An Authorization Model For The Web Programming Language Qwel

Lulwa Ahmed El-Matbouly

lulwa@cmu.edu

Advisors:

Thierry Sans tsans@qatar.cmu.edu

Soha Hussein sohah@qatar.cmu.edu

CARNEGIE MELLON UNIVERSITY

SCHOOL OF COMPUTER SCIENCE

May 3, 2013

Abstract

With the fast growth of web technology, it is becoming easier for developers to design and deploy complex web applications. However, securing such web applications is becoming an increasing complex task as current technology provides limited support. Developers are required to reason about distributed computation and to write code using heterogeneous languages, often not originally designed with distributed computing in mind nor built-in security features.

Qwel is an experimental type-safe functional programming language for the web that has dedicated primitives for publishing and invoking web services. In this paper, we propose to extend Qwel with a decentralized authorization model allowing service providers to secure web applications written in Qwel. This extension provides web developers with built-in primitives to issue credentials to users and to express access control policies. Therefore, when a protected web service is deployed, the security policy is evaluated dynamically based on the credentials supplied by the user invoking this web service. As a result, we show how these new language features can be used to implement common scenarios as well as more sophisticated ones.

1 Introduction

With the fast growth of web technology and cloud computing, it is becoming increasingly popular to move software and data to the cloud. In this paradigm, the software is no longer a standalone application installed on the user's computer but it is offered as a web application. For instance Google Docs is an office suite (word processor, spreadsheet and presentation) that can be used through a web browser. From the user perspective, Google Docs is not very different from standalone office suites like Microsoft Office or Open Office. However, from the developer perspective, building web applications is a significant shift in the way to design, implement and deploy software. Indeed, correctness and security have always been the main concerns but it takes a new dimension in the context of web applications. For instance, a bug or a crash that occurs in a standalone application may impact the platform's owner only. However, a bug or a crash that occurs on the server side of a web application may impact all users registered to the service. In the same way, a vulnerability in standalone application may expose data from the platform's owner only. However, a vulnerability in a web application may expose data of all users registered to the service.

Securing web applications is a complex task and attacks targeting web applications are on the increase [6]. If we look at the range of vulnerabilities affecting web applications [5], we can classify them into two families: Injection vulnerabilities such as SQL injection, cross-site scripting, cross-site request forgery, content spoofing and Response splitting are resulting from an incorrect handling of unexpected user inputs. Incomplete mediation vulnerabilities such as information leakage, insufficient authorization and predictable resource location are resulting from bad application design and/or misconfiguration of the platform in controlling user access to data or resources.

These attacks are hard to mitigate as current technology provides limited support. Developers are required to reason about distributed computation and to write code using heterogeneous languages, often not originally designed with distributed computing in mind nor built-in security features. In [4], Sans and Cervesato proposed Qwel, a small functional programming language extended with primitives for mobile code and remote procedure calls, two distinguishing features of web programming. The initial goal was to provide the developer with a programming language to write both client side and server code ensuring adequate interactions between them. Since Qwel is type safe language, it is more likely to mitigate injection attacks when user's inputs do not match the appropriate type. However, in its original version, Qwel does not have built-in features to mitigate incomplete mediation attacks.

In this paper, we propose to extend Qwel to provide the developer with language-level security features to control access to web applications. The main contributions of this work are 1) an extension of Qwel syntax and semantics with distributed access control mechanisms proposed by Abadi and al. [1] 2) a formalization of the policy interpreter based on the sequent calculus logic and 3) an implementation of the policy interpreter. The rest of the paper is structured as follows: Section 2 summarizes existing work in distributed access control. Section 3 introduces Qwel and lays the motivations for extending it with new security primitives for access control. Section 4.4 describes the extended Qwel syntax. Section 5 shows how this extension can be used to express common access control policies as well as more sophisticated ones. Section 6 provides a formalization of the language semantics. Section 7 concludes and provides an outline of future developments.

2 Related Work

Access control is a restriction of operations on resources such as files and services to specific users. In calculus for access control, Abadi et al. [1] present a calculus that combines authentication which is the problem of determining the identity of the requester (principal), and authorization which is the problem of determining if the principal is allowed to access certain service. Basically, access control models consist in a set of logical that grants permission to principals access resources. In [1], the concept of *principal* can be:

- Users and machines.
- Channels, such as input devices and cryptographic channels.
- Conjunction of principals, of the form $[A \wedge B]$.
- Groups, define groups of principals. The use of the group is to decide whether a principal is a member of a group.
- Principals in roles, of the form $[A \text{ as } R]$. Where principal A may adopt the role R and act under the name $[A \text{ as } R]$.
- Principals on behalf of principal, of the form $[B \text{ for } A]$. Where principal A may delegate authority to B, and B can then act on behalf of A, using the identity $[B \text{ for } A]$.

Each object have an access control list (ACL), where a request to an object will be granted if the principal is authorized according to this list. Determining whether a request from a principal granted or denied is based on the logical model that extends the algebra of principals.

3 Motivating Example

Consider an example where *Alice*, a student at *Univ* needs to submit an assignment for her course through a web portal called *Submission*. To avoid plagiarism, her professor ask to check her own assignment using an online service called *NoPlagiarism*. As a proof, her professor requires her to submit her assignment along with the similarity report obtained previously.

Qwel is an experimental programming language for the web proposed by Sans and Cervesato in [4]. At its core, Qwel is a basic functional language extended with primitives for publishing and calling web services. Using Qwel, the example can be implemented as follows:

- *NoPlagiarism.com* publishes a web service that takes a document as argument and returns the corresponding similarity report (figure 1).
- *Submission.org* publishes a service that stores a document and its similarity report both given as argument (figure 2).
- Assuming that these two services are deployed beforehand, *Alice* calls the service *similarityReport@NoPlagiarism.com* with her homework, obtains the similarity report in return and then forwards it to the *submit@submission.org* service (figure 3).

```
publish doc : string
  let
    report = calculateSimilarity(doc)
  in
    report
  end
```

Figure 1: similarityReport@NoPlagiarism.com

publish s as $\langle doc, report \rangle \quad = \quad store(s)$

Figure 2: submit@Submission.org

let
  $doc \quad = \quad$ "Once upon a time ..."
  $simReport \quad = \quad$ call $similarityReport@NoPlagiarism.com$ with doc
in
  call $submit@Submission.org$ with $\langle simReport, doc \rangle$
end

Figure 3: Alice combining web services to submit her assignment

Beyond the functional aspects of this example, we would like to express security policies. For example, *NoPlagiarism* could express that "only students from *Univ* can get a similarity report". In the same way, *Submission* could express that "only students from *Univ* can submit their assignments" and that "similarity reports must have been issued by *NoPlagiarism*".

However, Qwel has no language features allowing service providers to express such policies. Hence, we will extend Qwel with the built-in primitives that enables developers to 1) express a local security policy protecting a published service and 2) issue credentials to users. The language interpreter will grant access to a service if and only if the local security policy is satisfied according to the credentials carried by the principal calling the service.

4 Extending Qwel With a Distributed Access Control Model

In the example above, *NoPlagiarism* wants to ensure that "only students from *Univ* can get a similarity report". In this scenario, *NoPlagiarism* does not know who is a student at *Univ*. Instead, it will expect *Univ* to issue a proof a.k.a a credential saying that *Alice* is a student. This is a typical example of distributed access control where parties can express security policies locally based on credentials issued by others. In this section, we extend the Qwel syntax with the distributed access control model proposed by Abadi and al. in [1].

4.1 Credentials

A credential is a collection of claims. For instance, *Alice* is a student according to *Univ* and *NoPlagiarism* is the issuer of the similarity report. Previous work [1] introduces the the modality says to represent such a claim. A claim is a relation between a fact, defined as a predicate, and the principal emitting such a predicate (e.g $Univ$ says $student(Alice)$). In our model, predicates can take as attributes other principals $student(Alice)$ and/or values $issuer(report)$. Claims are told to be true if and only if they are part of a credential (e.g cred($Univ$ says $student(Alice)$)). A credential cred$(e_0, \ldots, e_n)$ can have one or many claims. However, principals cannot create arbitrary credentials on behalf of others. The programmer does not use the constructor cred directly. Instead, we define say e that takes a fact and returns a credential that contains a claim emitted locally. For instance, say $student(Alice)$ creates a new credential cred($Univ$ says $student(Alice)$) when evaluated at *Univ*.

Since a credential cred$(e_0, \ldots, e_n)$ can have one or many claims, we define $e_0 \ \oplus \ e_1$ that combines different credentials. For example, once *Alice* has obtained a credential from *Univ* saying that she is a student and another from *NoPlagiarism* saying that *NoPlagiarism* is the issuer of the report returned to her. *Alice* can combined these two credentials into one and submit it to *submit@Submission.org*

$$\text{cred}(Univ \text{ says } student(Alice)) \ \oplus \ \text{cred}(NoPlagiarism \text{ says } issuer(report))$$

4.2 Access Control Policies

An access control policy restricts who can access a published service. In our model, an access control policy is an expression defining a constraint on the credentials carried by the service caller. For instance, *NoPlagiarism* can express that *Alice* must be a student to call its service *similarityReport*. To define this elementary policy, we define $\mathsf{pol}(Univ \text{ says } student(Alice))$ that specify that *Alice* must carry a credential in which *Univ* claims that she is a student.

To express more complex policies, we add logical operators such as $e_0 \wedge e_1$, $e_0 \vee e_1$, $\exists\ x.e(x)$ that allow the developer to combine constraints and express more complex policies. For instance, *Submission* can express that *Alice* must be a student from *Univ* and *NoPlagiarism* must be the issuer of the similarity report:

$$\mathsf{pol}(university \text{ says } student(Alice)) \wedge \mathsf{pol}(plagiarism \text{ says } issuer(report))$$

4.3 Issuing Credentials and Evaluating Policies

As introduced above, $\mathsf{pol}(Univ \text{ says } student(Alice))$ is the access control policy that says that *Alice* must be a student to call *similarityReport@NoPlagiarism.com*. However, it is unlikely that *NoPlagiarism* has to explicitly mention *Alice* (and any other potential other principal) in its policy. Instead, *NoPlagiarism* should write that anybody calling its service must be a student. Therefore, we need to be able to write an access control policy based on a variable representing the principal calling the service. To do so, we redefine the construct publish $w.x : \tau \Rightarrow e$ in such a way that w will be instantiated with the principal calling the service during the evaluation.

Finally, to check if an access control policy is satisfied based on the credential submitted as argument. We introduce the construct $\mathsf{check}(e_0, e_1)$ that will verify that the credentials e_0 satisfies the policy e_1.

4.4 Full Extended Syntax

To summarize, Qwel is extended with the following constructs:

$$
\begin{array}{lrcl}
\text{Type} & \tau & := & \mathsf{world} \mid \tau \rightsquigarrow \tau' \\
& & \mid & \mathsf{fact} \mid \mathsf{claim} \mid \mathsf{credential} \mid \mathsf{policy} \\
\\
\text{Expression} & e & ::= & \mathsf{url}(\mathsf{w}) \mid \mathsf{here} \\
& & \mid & \mathsf{url}(\mathsf{w}, u) \mid \mathsf{publish}\ w.x : \tau \Rightarrow e \mid \mathsf{call}\ e_1\ \mathsf{with}\ e_2 \mid \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w} \\
& & \mid & p(e_1, \ldots, e_n) \mid e_1 \Rightarrow e_2 \mid e_0\ \mathsf{says}\ e_1 \\
& & \mid & \mathsf{say}\ e \mid \mathsf{cred}(e_0, \ldots, e_n) \mid e_0 \oplus e_1 \\
& & \mid & \mathsf{pol}(e) \mid \exists\, x : \tau.e_1 \mid e_0 \wedge e_1 \mid e_0 \vee e_1 \\
& & \mid & \mathsf{check}(e_0, e_1)
\end{array}
$$

We have not introduced the construct $e_1 \Rightarrow e_2$ yet. This construct was suggested by [1] and will be illustrated in an example shown in 5.2

4.5 Syntactic Sugar

For convenience, we extend the syntax with syntactic sugar allowing the developer to define a protected service:

$$
\begin{array}{l}
\mathsf{publish}\ w.x : \tau \times \mathsf{credential} \Rightarrow e_0 \\
\mathsf{protect}\ e_1
\end{array}
\quad \triangleq \quad
\begin{array}{l}
\mathsf{publish}\ w.x : \tau \times \mathsf{credential} \Rightarrow \\
\\
\end{array}
\quad
\begin{array}{l}
\mathsf{if}\ \mathsf{check}(\mathsf{snd}\ x, e_1)\ \mathsf{then}\ e_0 \\
\mathsf{else}\ \mathsf{raise}\ AccessDeniedException
\end{array}
$$

5 Examples

5.1 Example 1: University Submission System

Based on the extension suggested above, we are now able to write in Qwel the example presented in section 3:

1. To get the *Univ* credential, *Alice* calls *getStudentCred@Univ.edu*. The server checks if *Alice* is a student and returns a credential saying that the caller of the service is a student (figure 4).
2. To get the similarity report, *Alice* sends her university credential and her assignment to the *similarityReport@NoPlagiarism.com*. In return, she gets the similarity report and a credential specifying that *NoPlagiarism.com* is the issuer of the similarity report.
3. Finally, to submit her assignment, she combines the credentials obtained from *Univ* and from *NoPlagiarism*. Thus, she forwards the similarity report and the aggregated credential to the submission service.

```
let
  doc                          =  "Once uppon a time ..."
  univCred                     =  call getStudentCred@Univ.edu with ()
  ⟨simReport, plagCred⟩        =  call similarityReport@NoPlagiarism.com with ⟨doc, univCred⟩
  univPlagCred                 =  univCred ⊕ plagCred
in
    call submit@Submission.org with ⟨simReport, univPlagCred⟩
end
```

Figure 4: Alice calls for the services

```
publish w.x as x : unit ⇒
  if checkStudent(w)
     then  say student(w)
     else  raise AccessDeniedException
```

Figure 5: getStudentCred@Univ.edu

```
publish w.x as ⟨doc, cred⟩ : string × credential ⇒
  let
     report     =  ⟨doc, calculateSimilarity(doc)⟩
     plagCred   =  say issuer(report)
  in
    ⟨report, plagCred⟩
  end
protect
  pol(Univ.edu says student(w))
```

Figure 6: similarityReport@NoPlagiarism.com

5.2 Example 2: Managing Medical Reports at the Hospital

Consider an example of an hospital in which medical reports are managed electronically. In figure 8, the hospital system publishes a service that returns the medical report corresponding to the id given as argument. This service is ruled by the following policy:

publish $w.x$ as $\langle report, cred \rangle$: (string $\times$ string) $\times$ credential $\Rightarrow$ $store(s)$
protect
  pol($Univ.edu$ says $student(w)$) $\wedge$ pol($NoPlagiarism.com$ says $issuer(report)$)

Figure 7: submit@Submission.org

- **Rule 1**: the patient can access his own medical report
- **Rule 2**: any doctor working for the hospital can access any medical report
- **Rule 3**: anybody that is explicitly allowed by the patient can access to the patient medical report

Rule 1 and rule 2 can be implemented using the constructs introduced previously. However, rule 3 can be seen as a delegation rule: "anybody speaking on behalf of the owner can access the medical report". For instance, *Alice* can delegate authority to her *Grandma* in order for her to access Alice's medical report. For that purpose, we introduce the construct $w_1 \Rightarrow w_2$ (also defined by [1]) that allows a principal to delegate authority to another principal.

To satisfy rule 1, *Alice* must obtain a credential from the hospital saying that she is the owner of a certain medical report with a specific id (figures 9 and 12).

To satisfy rule 2, *Bob* must obtain a credential from the hospital saying that he is a doctor (figures 11 and 10).

To satisfy rule 3, *Alice's Grandma* must obtain a credential from Alice saying that she can speaks on her behalf (figures 13 and 14).

publish $w.s$ as $\langle id, cred \rangle$: int $\times$ credential $\Rightarrow$ $retrieve(id)$
protect
  pol($hospital$ says $owner(id, w)$)
  $\vee$ pol($hospital$ says $doctor(w)$)
  $\vee \exists\, w'$: world.$w \Rightarrow w' \wedge hospital$ says $owner(id, w')$

Figure 8: Get medical report service at the hospital

publish $w.x$: unit $\Rightarrow$
  let
    $id = getPatientId(w)$
  in
    say $owner(id, w)$
  end

Figure 9: getMedicalReportCred@hospital

let
  $doctorCred$ = call $getDoctorCred@hospital$ with ()
in
  call $getMedicalReport@hospital$ with $\langle 2136, doctorCred \rangle$
end

Figure 10: Call get medical report service by doctor (Bob)

```
publish w.x : unit ⇒
  if isDoctor(w)
    then say doctor(w)
    else raise AccessDeniedException
```

Figure 11: getDoctorCred@hospital

```
let
  reportCred = call getMedicalReportCred@hospital with ()
in
  call getMedicalReport@hospital with ⟨2136, reportCred⟩
end
```

Figure 12: Call get medical report service by the patient (Alice)

```
publish w.x : unit ⇒
  if isMyGrandma(w)
    then say w ⇒ here
    else raise AccessDeniedException
```

Figure 13: getAliceDelegationCred@Alice

```
let
  delegationCred = call getAliceDelegationCred@Alice with ()
in
  call getMedicalReport@hospital with ⟨2136, delegationCred⟩
end
```

Figure 14: Call get medical report service by grandma who speaks for Alice

6 Semantics

The static semantics of the proposed Qwel extension is defined in figures 15 and 16. The dynamic semantics is defined in figures 17, 18, 19 and 20. We formalized the policy evaluation based on an extended sequent calculus logic (figure 21). The sequent calculus [3] is a simple set of rules that can be used to show the truth of statements in first order logic. As a proof of concept, we have developed a policy evaluator implementing the sequent calculus rules (see appendix).

$$\frac{}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{url}(\mathsf{w}') : \mathsf{world}}\ \textsc{of\_url} \qquad \frac{}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{here} : \mathsf{world}}\ \textsc{of\_here}$$

$$\frac{}{\Sigma, u : \tau \rightsquigarrow \tau' \ @\ \mathsf{w}';\Gamma \vdash_{\mathsf{w}} \mathsf{url}(\mathsf{w}', u) : \tau \rightsquigarrow \tau'}\ \textsc{of\_url} \qquad \frac{\Sigma;\Gamma, w : \mathsf{world}, x : \tau \vdash_{\mathsf{w}} e : \tau'}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{publish}\ w.x : \tau \Rightarrow e : \tau \rightsquigarrow \tau'}\ \textsc{of\_publish}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \tau \rightsquigarrow \tau' \qquad \Sigma;\Gamma \vdash_{\mathsf{w}} e_2 : \tau}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{call}\ e_1\ \mathsf{with}\ e_2 : \tau'}\ \textsc{of\_call} \qquad \frac{\Sigma;\Gamma \vdash_{\mathsf{w}'} e : \tau}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}' : \tau}\ \textsc{of\_expect}$$

Figure 15: Typing rules for modified Qwel constructs

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \tau_1 \quad \ldots \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_n : \tau_n}{\Sigma;\Gamma \vdash_{\mathsf{w}} p(e_1,\ldots,e_n) : \mathsf{fact}}\ \text{of\_p} \qquad \frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{world} \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_2 : \mathsf{world}}{\Sigma;\Gamma \vdash_{\mathsf{w}} e_1 \Rightarrow e_2 : \mathsf{fact}}\ \text{of\_speaksfor}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 : \mathsf{world} \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{fact}}{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0\ \mathsf{says}\ e_1 : \mathsf{claim}}\ \text{of\_says}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e : \mathsf{fact}}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{say}\ e : \mathsf{credential}}\ \text{of\_say} \qquad \frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 : \mathsf{claim} \quad \ldots \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_n : \mathsf{claim}}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{cred}(e_0,\ldots,e_n) : \mathsf{credential}}\ \text{of\_cred}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 : \mathsf{credential} \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{credential}}{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 \ \oplus\ e_1 : \mathsf{credential}}\ \text{of\_join}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e : \mathsf{claim}}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{pol}(e) : \mathsf{policy}}\ \text{of\_pol} \qquad \frac{\Sigma;\Gamma, x:\tau \vdash_{\mathsf{w}} e : \mathsf{policy}}{\Sigma;\Gamma \vdash_{\mathsf{w}} \exists\, x:\tau.e : \mathsf{policy}}\ \text{of\_exists}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 : \mathsf{policy} \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{policy}}{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 \wedge e_1 : \mathsf{policy}}\ \text{of\_and} \qquad \frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 : \mathsf{policy} \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{policy}}{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 \vee e_1 : \mathsf{policy}}\ \text{of\_or}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{w}} e_0 : \mathsf{credential} \quad \Sigma;\Gamma \vdash_{\mathsf{w}} e_1 : \mathsf{policy}}{\Sigma;\Gamma \vdash_{\mathsf{w}} \mathsf{check}(e_0,e_1) : \mathsf{boolean}}\ \text{of\_check}$$

Figure 16: Typing Rules for new Qwel constructs

$$\frac{}{\mathsf{url}(\mathsf{w}')\ \mathsf{val}}\ \text{val\_url} \qquad \frac{m}{j}\ \text{val\_url}\ n\mathsf{url}(\mathsf{w}',u)\ \mathsf{val}$$

$$\frac{}{\Delta\ ;\ \mathsf{here}\ \mapsto_{\mathsf{w}}\ \Delta\ ;\ \mathsf{url}(\mathsf{w})}\ \text{ev\_here}$$

$$\frac{}{\Delta\ ;\ \mathsf{publish}\ w.x:\tau \Rightarrow e\ \mapsto_{\mathsf{w}}\ (\Delta, u\ @\ \mathsf{w} \hookrightarrow w.x:\tau.e)\ ;\ \mathsf{url}(\mathsf{w},u)}\ \text{ev\_publish}$$

$$\frac{\Delta\ ;\ e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ \mathsf{call}\ e_1\ \mathsf{with}\ e_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{call}\ e_1'\ \mathsf{with}\ e_2}\ \text{ev\_call}_1 \qquad \frac{v_1\ \mathsf{val} \quad \Delta\ ;\ e_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_2'}{\Delta\ ;\ \mathsf{call}\ v_1\ \mathsf{with}\ e_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{call}\ v_1\ \mathsf{with}\ e_2'}\ \text{ev\_call}_2$$

$$\frac{v_2\ \mathsf{val}}{\underbrace{(\Delta^*, u\ @\ \mathsf{w}' \hookrightarrow w.x:\tau.e)}_{\Delta}\ ;\ \mathsf{call}\ \mathsf{url}(\mathsf{w}',u)\ \mathsf{with}\ v_2\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{expect}\ [\mathsf{url}(\mathsf{w}), v_2/w, x]e\ \mathsf{from}\ \mathsf{w}'}\ \text{ev\_call}_3$$

$$\frac{\Delta\ ;\ e\ \mapsto_{\mathsf{w}'}\ \Delta'\ ;\ e'}{\Delta\ ;\ \mathsf{expect}\ e\ \mathsf{from}\ \mathsf{w}'\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{expect}\ e'\ \mathsf{from}\ \mathsf{w}'}\ \text{exp}_1 \qquad \frac{v\ \mathsf{val}}{\Delta\ ;\ \mathsf{expect}\ v\ \mathsf{from}\ \mathsf{w}'\ \mapsto_{\mathsf{w}}\ \Delta\ ;\ v}\ \text{exp}_2$$

Figure 17: Evaluation Rules for modified Qwel constructs

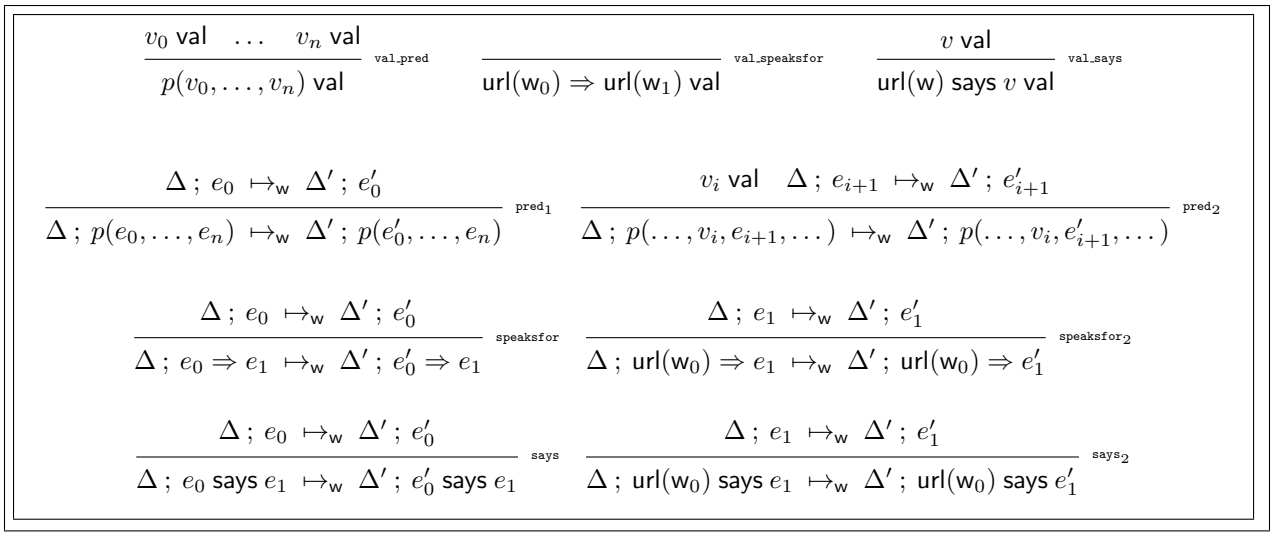

Figure 18: Evaluation Rules for Qwel claim constructs

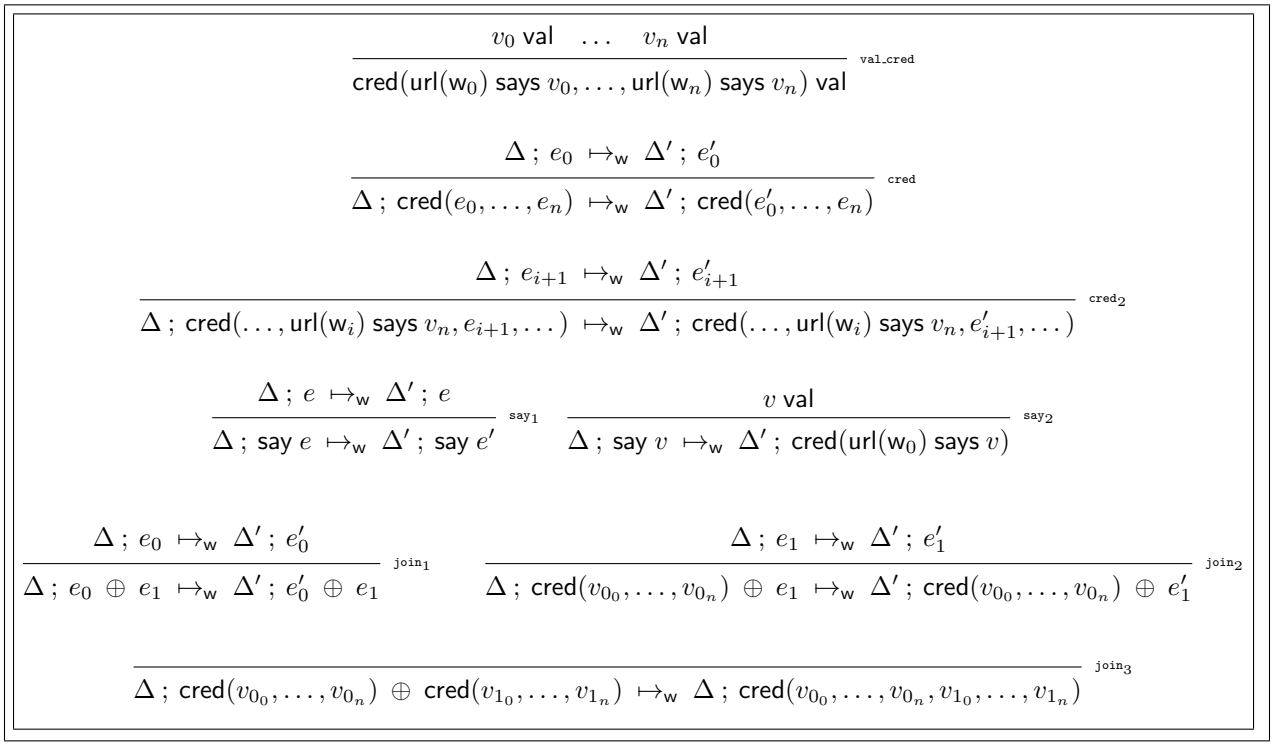

Figure 19: Evaluation Rules for Qwel credential constructs

7 Conclusion and Future Work

In conclusion, the main goal of the thesis is to extend Qwel language syntax and semantics providing developers with a mean to issue credentials and protect web services with access control policies. This extension defines an expressive language, yet simple and easy to use for the purpose of building web services with embedded security constraints.

There are several avenues for future work, in our model the caller is responsible for getting the credentials and sends them to the server that will try to prove that they satisfy the policy. This can be overwhelming for the server

$$\frac{v\ \mathsf{val}}{\mathsf{pol}(\mathsf{url}(\mathsf{w})\ \mathsf{says}\ v)\ \mathsf{val}}\ \text{val\_pol} \qquad \frac{}{\exists\, x : \tau.e\ \mathsf{val}}\ \text{val\_exists} \qquad \frac{v_0\ \mathsf{val} \quad v_1\ \mathsf{val}}{v_0 \wedge v_1\ \mathsf{val}}\ \text{val\_andp} \qquad \frac{v_0\ \mathsf{val} \quad v_1\ \mathsf{val}}{v_0 \vee v_1\ \mathsf{val}}\ \text{val\_orp}$$

$$\frac{\Delta\ ;\ e\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e}{\Delta\ ;\ \mathsf{pol}(e)\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{pol}(e')}\ \text{pol}$$

$$\frac{\Delta\ ;\ e_0\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_0'}{\Delta\ ;\ e_0 \wedge e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_0' \wedge e_1}\ \text{and}_1 \qquad \frac{v_0\ \mathsf{val} \quad \Delta\ ;\ e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ v_0 \wedge e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ v_0 \wedge e_1'}\ \text{and}_2$$

$$\frac{\Delta\ ;\ e_0\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_0'}{\Delta\ ;\ e_0 \vee e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_0' \vee e_1}\ \text{or}_1 \qquad \frac{v_0\ \mathsf{val} \quad \Delta\ ;\ e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ v_0 \vee e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ v_0 \vee e_1'}\ \text{or}_2$$

$$\frac{\Delta\ ;\ e_0\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_0'}{\Delta\ ;\ \mathsf{check}(e_0, e_1)\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{check}(e_0', e_1)}\ \text{check}{+}1 \qquad \frac{v_0\ \mathsf{val} \quad \Delta\ ;\ e_1\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ e_1'}{\Delta\ ;\ \mathsf{check}(v_0, e_1)\ \mapsto_{\mathsf{w}}\ \Delta'\ ;\ \mathsf{check}(v_0, e_1')}\ \text{check}_2$$

$$\frac{v_0\ \mathsf{val} \quad \ldots \quad v_n\ \mathsf{val} \quad v\ \mathsf{val} \quad v_0, \ldots, v_n \models v}{\Delta\ ;\ \mathsf{check}(\mathsf{cred}(v_0, \ldots, v_n), v)\ \mapsto_{\mathsf{w}}\ \Delta\ ;\ \mathsf{true}}\ \text{check}_3 \qquad \frac{v_0\ \mathsf{val} \quad \ldots \quad v_n\ \mathsf{val} \quad v\ \mathsf{val} \quad v_0, \ldots, v_n \not\models v}{\Delta\ ;\ \mathsf{check}(\mathsf{cred}(v_0, \ldots, v_n), v)\ \mapsto_{\mathsf{w}}\ \Delta\ ;\ \mathsf{false}}\ \text{check}_4$$

Figure 20: Evaluation Rules for Qwel policy constructs

$$\frac{}{\Gamma, p(x_1, \ldots, x_n) \vdash p(x_1, \ldots, x_n), \Delta}\ \text{pred} \qquad \frac{}{\Gamma, \mathsf{w}_1 \Rightarrow \mathsf{w}_2 \vdash \mathsf{w}_1 \Rightarrow \mathsf{w}_2, \Delta}\ \Rightarrow \qquad \frac{\rho \vdash \rho'}{\Gamma, \mathsf{w}\ \mathsf{says}\ \rho \vdash \mathsf{w}\ \mathsf{says}\ \rho', \Delta}\ \text{says}$$

$$\frac{\Gamma, \rho_1, \rho_2 \vdash \Delta}{\Gamma, \rho_1 \wedge \rho_2 \vdash \Delta}\ \wedge\text{L} \qquad \frac{\Gamma \vdash \rho_1, \Delta \quad \Gamma \vdash \rho_2, \Delta}{\Gamma \vdash \rho_1 \wedge \rho_2, \Delta}\ \wedge\text{R}$$

$$\frac{\Gamma, \rho_1 \vdash \Delta \quad \Gamma, \rho_2 \vdash \Delta}{\Gamma, \rho_1 \vee \rho_2 \vdash \Delta}\ \vee\text{L} \qquad \frac{\Gamma \vdash \rho_1, \rho_2, \Delta}{\Gamma \vdash \rho_1 \vee \rho_2, \Delta}\ \vee\text{R}$$

$$\frac{\Gamma \vdash \rho_1, \Delta \quad \Gamma, \rho_2 \vdash \Delta}{\Gamma, \rho_1 \to \rho_2 \vdash \Delta}\ \to\text{L} \qquad \frac{\Gamma, \rho_1 \vdash \rho_2, \Delta}{\Gamma \vdash \rho_1 \to \rho_2, \Delta}\ \to\text{R}$$

$$\frac{\Gamma, \rho(x) \vdash \Delta}{\Gamma, \exists\, x.\rho(x) \vdash \Delta}\ \vee\text{L} \qquad \frac{\Gamma \vdash \rho(z), \Delta}{\Gamma \vdash \exists\, x.\rho(x), \Delta}\ \vee\text{R}$$

Figure 21: Policy evaluation

when dealing with multiple parallel service calls. In Proof carrying authorization (PCA) [2], the service provider sends its policy to the caller. The latter must build a proof that his/her credentials satisfy the policy. If such a proof can be derived, this proof is sent sent back to the server. Hence, the server simply need to verify the soundness of the proof

rather than trying to find one. In the future, we want to adapt the PCA model in Qwel.

References

[1] M. Abadi, M. Burrows, B. Lampson, G. Plotkin, J. Kohl, C. Neuman, and J. Steiner. A calculus for access control in distributed systems, 1991.

[2] Ljudevit Bauer. *Access control for the web via proof-carrying authorization*. PhD thesis, Princeton, NJ, USA, 2003. AAI3107865.

[3] Gerhard Gentzen. Investigations into logical deduction. *American philosophical quarterly*, 1(4):288–306, 1964.

[4] Thierry Sans and Iliano Cervesato. QWeSST for Type-Safe Web Programming. In Berndt Farwer, editor, *Third International Workshop on Logics, Agents, and Mobility — LAM'10*, volume 7 of *EPiC*, pages 96–111, Edinburgh, Scotland, UK, 15 July 2010. EasyChair Publications.

[5] WhiteHat Security. Website Statistics Report, 2012.

[6] Symantec. Internet Security Threat Report, 2013.

A SML implementation of the policy evaluator

```
type world = string

type value = string

type pr = string

datatype expression = v of value
                    | w of world

datatype proposition = Pred of world * pr  * expression list
                     | speaksfor of world * world * world

datatype formula = prep  of proposition
                 | andf of formula * formula
                 | orf of formula * formula
                 | Exists of expression * formula

(* listequal: list * list -> boolean*)
fun listequal [] [] = true
    | listequal (x::l1) (y::l2) = (x=y) andalso listequal l1 l2
    | listequal _  _ = false


(* getValueDomain: proposition list -> proposition list *)
fun  getValueDomain(Pred(w',q,l)::pl) =
            (List.filter (fn v(e) => true | w(e) => false) l)@getValueDomain(pl)
    | getValueDomain(speaksfor((w1,w2,w3))::pl) = getValueDomain(pl)
    | getValueDomain([]) = []

(* getWorldDomain: proposition list -> proposition list *)
fun getWorldDomain(Pred(w',q,l)::pl) = w(w')::(List.filter (fn v(e) => false | w(e) => true) l)
                                       @getWorldDomain(pl)
  | getWorldDomain(speaksfor((w1,w2,w3))::pl) =  w(w1)::w(w2)::w(w3)::getWorldDomain(pl)
  | getWorldDomain([]) = []


(* replaceValueList: value * value * expression list -> expression list *)
fun replaceValueList(v1,v2,v(expr)::l) =  if(v1=expr)
                                          then v(v2)::replaceValueList(v1,v2,l)
                                          else v(expr)::replaceValueList(v1,v2,l)
    | replaceValueList(v1,v2,w(expr)::l) = w(expr)::replaceValueList(v1,v2,l)
    | replaceValueList(v1,v2,[]) = []

(* replaceValue: value * value * formula -> formula *)
fun replaceValue(v1,v2, prep(Pred(w',q,l))) = prep(Pred(w',q,replaceValueList(v1,v2,l)))
  | replaceValue(v1,v2, prep(speaksfor(w3,w4,w5))) = prep(speaksfor(w3,w4,w5))
  | replaceValue(v1,v2, andf(f1,f2)) = andf(replaceValue(v1,v2,f1),replaceValue(v1,v2,f2))
  | replaceValue(v1,v2, orf(f1,f2)) = orf(replaceValue(v1,v2,f1),replaceValue(v1,v2,f2))
  | replaceValue(v1,v2, Exists(v(expr),f)) = if (expr = v1) then Exists(v(expr),f)
                                             else Exists(v(expr),replaceValue(v1,v2,f))
  | replaceValue(v1,v2, Exists(w(expr),f)) = Exists(w(expr),replaceValue(v1,v2,f))

(* replaceWorldList: value * value * expression list -> expression list *)
fun replaceWorldList(w1,w2,w(expr)::l) =  if(w1=expr)
                                          then w(w2)::replaceWorldList(w1,w2,l)
                                          else w(expr)::replaceWorldList(w1,w2,l)
  | replaceWorldList(w1,w2,v(expr)::l) = v(expr)::replaceWorldList(w1,w2,l)
  | replaceWorldList(w1,w2,[]) = []

(* replaceWorld: value * value * formula -> formula *)
fun replaceWorld(w1,w2, prep(Pred(w',q,l))) = if (w'=w1)
                                              then prep(Pred(w2,q,replaceWorldList(w1,w2,l)))
```

```
                                                 else prep(Pred(w',q,replaceWorldList(w1,w2,l)))
| replaceWorld(w1,w2, prep(speaksfor(w3,w4,w5))) = let
                                                    val w3' = if (w3=w1) then w2 else w3
                                                    val w4' = if (w4=w1) then w2 else w4
                                                    val w5' = if (w5=w1) then w2 else w5
                                                  in
                                                    prep(speaksfor(w3',w4',w5'))
                                                  end
| replaceWorld(w1,w2, andf(f1,f2)) = andf(replaceWorld(w1,w2,f1),replaceWorld(w1,w2,f2))
| replaceWorld(w1,w2, orf(f1,f2)) = orf(replaceWorld(w1,w2,f1),replaceWorld(w1,w2,f2))
| replaceWorld(w1,w2, Exists(w(expr),f)) = if (expr = w1) then Exists(w(expr),f)
                                           else Exists(w(expr),replaceWorld(w1,w2,f))
| replaceWorld(w1,w2, Exists(v(expr),f)) = Exists(v(expr),replaceWorld(w1,w2,f))

val removeDuplicates:(expression list -> expression list) =
       List.foldl (fn (x,b)=> if (List.exists (fn y=>(y=x)) b) then b else x::b) []



(* prove: proposition list * formula -> boolean*)
fun prove(model, policy) =
  let
    (* check: expression list * expression list * proposition list * formula list -> boolean *)
    fun check(d, d', model, prep(p)::f) = (List.exists (fn x => case (x,p)
                                            of (Pred(w',q,l),Pred(w'',q',l')) => (w''=w')
                                                andalso (q=q') andalso (listequal l l')
                                             | (speaksfor(w1,w2,w3),speaksfor(w1',w2',w3')) =>
                                                 (w1=w1') andalso (w2=w2') andalso (w3=w3')
                                             | _ => false)
                                                    model) orelse check(d,d',model,f)
      | check(d, d',  model, andf(pol1,pol2)::f) = check(d,d', model,pol1::f)
                                                   andalso check(d,d', model, pol2::f)
      | check(d, d', model, orf(pol1,pol2)::f) = check(d,d', model, pol1::pol2::f)
      | check(d, d', model, Exists(v(x),pol)::f) =
            check (d,d', model,(List.map (fn v(y):expression => replaceValue(x,y,pol)
                                                  | w(y) => pol) d')@f)
      | check(d, d', model, Exists(w(x),pol)::f) =
            check (d,d', model,(List.map (fn w(y):expression => replaceWorld(x,y,pol)
                                                  | v(y) => pol) d)@f)
      | check(d, d', model, []) = false

  in
      check(removeDuplicates(getWorldDomain(model)),
            removeDuplicates(getValueDomain(model)),
            model, [policy])
  end
```