DOI: 10.1109/mic.2014.102

Authentication, Authorization, and Accounting in WebRTC PaaS Infrastructures. The Case of Kurento

Luis López-Fernández, Micael Gallego, and Boni García Universidad Rey Juan Carlos
David Fernández-López and Francisco Javier López NaevaTec

Server infrastructures for Web Real-Time Communications (WebRTC) are useful for creating rich applications. Developers commonly use them to access capabilities such as group communications, archiving, and transcoding. Kurento is an open source project that provides a WebRTC media server and a platform as a service (PaaS) cloud built on top. The authors present its API and analyze different security models for it, investigating the suitability of using simple access control lists and capability-based security schemes to provide authorization.

Web Real-Time Communications (WebRTC) is the umbrella term for several emergent technologies and APIs that aim to bring such communications to the Web.1 Although still in its infancy,2 WebRTC is a technological initiative getting considerable worldwide attention. One of the biggest challenges with WebRTC is security. Standardization bodies are investing huge efforts to address the security issues associated with performing calls on the Web,1 but these efforts are concentrated at the client. However, the use of WebRTC media infrastructures is becoming common practice. For example, WebRTC media gateways are typical in services that require protocol or format adaptations, as when integrating WebRTC with the IP Multimedia Subsystem (IMS).3 Multipoint control units (MCUs) are often used to support group communications,4 and recording media servers are helpful when persisting WebRTC calls.

Here, we concentrate on the infrastructure side of the problem by first reviewing the state of the art in this area and introducing common security models. Then, we introduce Kurento, a WebRTC media server, and Nubomedia, a platform as a service (PaaS) written on top of it. We experiment with different authorization models that we can implement in Nubomedia, concentrating on two: access control lists (ACLs) and capability-based security (CAP). We compare both models to show their advantages and drawbacks.

WebRTC Media Servers

In the past few years, the expectations arising from WebRTC technologies have brought a golden era to media server vendors. Just insert "WebRTC media server" into your preferred search engine, and you will obtain dozens of solutions. Despite this multiplicity, their features fall into just three categories5:

• Group communication capabilities include mixing and forwarding. This type of media server is called an MCU, following terminology from the ITU-T's H.323 recommendation.6

• Media archiving capabilities deal with recording audiovisual streams into structured or unstructured repositories and recovering them later for visualization.

• Media bridging capabilities refer to attaining interoperability among networks or domains having incompatible media formats or protocols. WebRTC-to-IMS gateways are the most popular in this area.

WebRTC Media Infrastructures in the Cloud In the past few years, cloud technologies have permeated the multimedia RTC market.7 Media processing is usually associated with high computational costs,8 so media servers are easily susceptible to scalability problems. This has pushed most vendors to follow different anything-as-a-service (XaaS) schemes.9 When looking to the WebRTC arena, the software-as-a-service (SaaS) model isn't especially interesting because it doesn't exploit WebRTC's real benefits (WebRTC is a development framework and not a specific service). Infrastructure-as-a-service (IaaS) models expose virtual instances of media servers but lack horizontal scalability.10 In WebRTC, clouds are making inroads through the PaaS model. The mainstream tendency is to reveal platform APIs that offer access to vast computational resources, exposing some of the capabilities enumerated earlier.

Security Requirements

Most WebRTC PaaS models4 split the service roles among three different entities: the PaaS provider, which deploys and maintains the PaaS infrastructure and offers its capabilities through an API; the application provider, which creates WebRTC-enabled applications based on the API; and the user, who accesses the application (and hence consumes the PaaS API) from a Web browser. Following this model, users access the PaaS resources, but the PaaS provider charges the application provider for it. Hence, users access the PaaS on behalf of the application provider. This requires the application provider to manage user identities and credentials, which lets it implement different types of business models without requiring the PaaS to own any kind of data about users. From a security perspective, the question is how this model can deal with the AAA problem: authentication, authorization, and accounting. Based on the description just given, the application provider — which owns the user identities — must implement authentication. Authorization and accounting, on the other hand, must be implemented by the PaaS because it receives and executes API requests. This requires some kind of mechanism to let application providers control and limit user access to PaaS resources. OAuth is the most well-known standard enabling access to resources on others' behalf.11 It allows users to give an application permission to access their private server resources without sharing their credentials. However, we can't apply OAuth to the described PaaS model because users don't own the server resources. The application provider does, and it must provide the user with the appropriate permissions to access the PaaS on its behalf. So, the OAuth roles are somehow inverted, making the OAuth protocol flow inapplicable.

Implementing Security

Most WebRTC PaaS APIs implement AAA using a simple, token-based protocol, inspired by OAuth but with a different flow.4 Figure 1 depicts this protocol's message flow, which comprises the following steps:

1. The user's browser requests access to the application. This means that it asks for a specific webpage (HTML and JavaScript code), which is hosted at a Web application server (AS) that the application provider owns. During this step, the AS can require the user to authenticate using the mechanism it desires: formbased, HTTP digest, OAuth, and so on.

2. The AS holds a unique identifier of the application it's serving (appId) and a secret password associated with it. Prior to deploying the application, the application provider must obtain both from the PaaS provider.

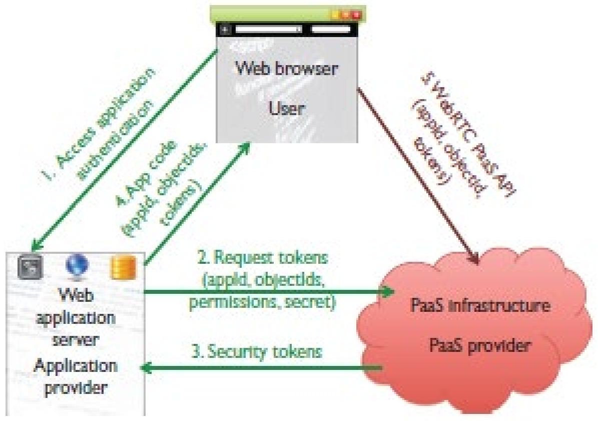
Web browser
User
1. Access application authentication
4. App code (appId, objectIds, tokens)
5 WebRTC PaaS API (appId, objectId, tokens)
Web application server
Application provider
2. Request tokens (appId, objectIds, permissions, secret)
3. Security tokens
PaaS infrastructure
PaaS provider

Figure 1. Conceptual model of the authentication, authorization, and accounting (AAA) mechanism. We implemented the model in a platform-as-a-service (PaaS) infrastructure using the token security mechanism.

Depending on the authentication result (step 1) and the application business logic, the AS determines what permissions to provide to the user. These permissions are associated with objects residing at the PaaS or created by the AS during this step (for example, a “room” for a group call, or a recorder capability). Next, the AS asks the PaaS for the appropriate tokens that grant these permissions.

3. The PaaS validates the application provider’s access rights using the secret password and generates the tokens, which are stored with their associated permissions and security rules at the PaaS.

4. The AS generates a response page to the user that contains the client-side application logic, appId, objectIds of all involved media capabilities, and tokens that let the user access those capabilities.

5. The user’s browser now executes the client-side application logic, issuing requests to the PaaS. Each request carries three parameters: appId (identifying the application), objectId (identifying the target object), and the appropriate tokens. Next, the PaaS performs the request authorization by checking whether the provided tokens grant the requested permissions that enable access to the target object.

As we expected with this scheme, authentication falls to the application provider, and the PaaS needs to deal only with authorization and accounting. To implement accounting, the traditional approach is to use call detail records: every time an application creates, releases, or accesses a media object, the platform records into a database the appId, operation type, and objectId. Later, a batch analysis lets providers implement diverse billing mechanisms based on this information. However, different possibilities exist for dealing with authorization. We review these next.

Authorization Models

The scientific literature has widely analyzed the authorization problem.12 Here, we concentrate on two basic mechanisms: ACLs and CAP.13

ACLs control which users in the system receive which permissions on which objects. To put them in place, we can just follow the scheme in Figure 1: the token acts as a unique identifier for a user session. Hence, during step 2, the AS requests one token associated with all the user’s permissions on all the necessary objects. Then, the PaaS stores in a database lists of objectIds mapped to tokens and permissions so that, for each objectId–token pair, the PaaS can recover the permissions. Consequently, when the user client issues a request (step 5), the PaaS can query the database and check whether the provided token has permission to execute the requested operation on the target object.

CAP, on the other hand, uses the token as a specific permission on a given object. The intuitive idea behind our CAP model is simple. Imagine that every feature of an object is protected by a "lock," and that the token is a "key" that lets the user open the lock. Adapting the Figure 1 scheme to CAP is straightforward. During step 2, the AS requests several tokens associated with all the permissions needed. When the PaaS creates an object, it generates and stores that object's full list of tokens. Hence, the PaaS just needs to send the appropriate token subset to the AS (step 3). The AS then gives these tokens to the user (step 4). Consequently, the user's requests (step 5) include the specific token subset that grants the permissions for the target object the operation requires. When the PaaS receives the request, it only needs to check that the provided tokens match the tokens stored at the object.

Implementing AAA in Kurento and Nubomedia

Our target PaaS is based on Kurento, an open source software project devoted to building a WebRTC media infrastructure (www.kurento.org).

At the heart of Kurento there is a piece of software called the Kurento Media Server (KMS) that's based on pluggable media processing capabilities. These capabilities are exposed to application developers through abstractions called media elements. The media element toolbox is quite rich; elements expose features that let applications record, mix, augment, blend, route, and apply computer vision to streams, for example.

From an application developer perspective, media elements are black boxes: you just need to take the desired elements and connect them following the required topology. In KMS jargon, a graph of connected media elements is called a media pipeline. Figure 2 shows the pipeline of an application implementing a WebRTC loopback, which receives and sends WebRTC streams (WebRtcEndpoint), processes the media flow for tracking an object's movements (PointerDetectorFilter), and adds a special effect, such as a hat, on top of the detected faces (FaceOverlayFilter) when the object enters a specific region.

KMS exposes its capabilities through its Media API, which lets clients manipulate media pipelines through stubs. So, for every media element, the API exposes an object type with simple primitives for controlling its behavior. For example, Figure 2 shows the creation of the application topology, which requires invoking the connect primitive on the corresponding media elements. Given that connect is always invoked in the element acting as the source and takes as argument the sink, the specific code is

```
webRtcEndpoint. connect(pointerDetectorFilter)
pointerDetectorFilter. connect(faceOverlayFilter)
faceOverlayFilter. connect(webRtcEndpoint)
```

Based on Kurento, we are building a WebRTC PaaS platform called Nubomedia (www.nubomedia. eu). Using horizontal scalability,10 Nubomedia transforms KMS into a distributed media server, which exposes its capabilities through the Media API. To implement authorization in Nubomedia, we must understand that most WebRTC PaaS models have created specific roles and permissions associated with each of their capabilities. For example, when working with MCUs, users might have roles such as moderator or publisher, which grant permissions such as the ability to ban users

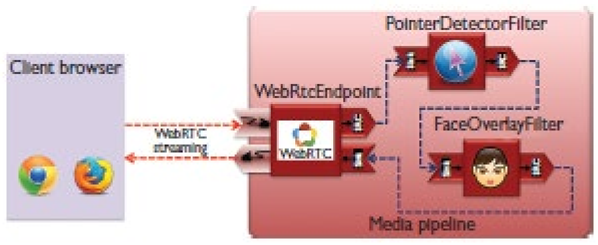

Figure 2. Kurento application using computer vision and augmented reality.The application creates a game in which users can interact with virtual objects to put on or remove a hat from their heads.This example is available at www.youtube.com/watch?v=5eJRnwKxgbY.

or publish streams, respectively. Hence, when a request comes into the PaaS asking for an operation on the MCU, the PaaS just checks permissions related to the MCU. This illustrates that these authorization models are "object oriented," in the sense that they're constrained to the capabilities that a single object can expose. With Nubomedia, the Media API enables developers to dynamically modify the pipeline topology and, with it, application functionality. Consequently, we also need a "topology orientation." This means that the connect primitive requires an authorization mechanism to discriminate who has the right to connect to whom. We've achieved this by introducing two additional permissions for each media element:

• READ_MEDIA. A user holding this permission on a given media element can invoke connect on that element, passing other elements as argument.

• WRITE_MEDIA. A user holding this permission for a given media element can invoke connect on other media elements, passing this media element as argument.

This gives the PaaS full control of the accessible pipeline topologies. The cost is an increase in the authorization logic complexity, given that connect invocations require checking permissions on two different objects.

We created minimal prototype implementations of the ACL and CAP schemes for Nubomedia. To understand the details, note that, as in most cloud platforms,8 the Nubomedia architecture has three layers:

• The load balancer receives client requests and distributes them among the available computational resources.

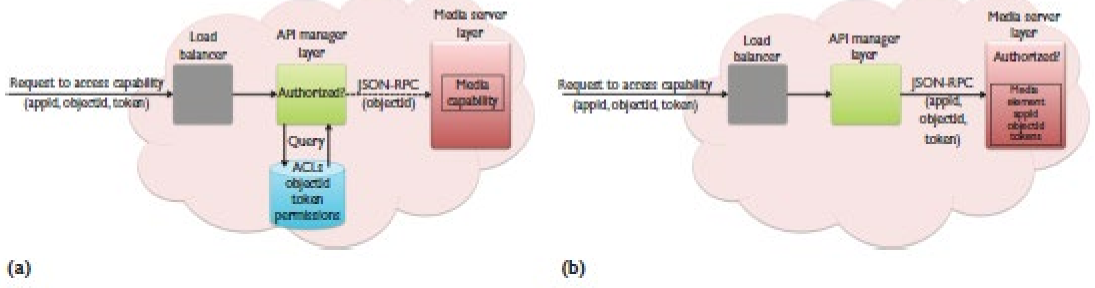
Request to access capability (appid, objectid, token)
Load balancer
API manager layer
Authorized?
JSON-RPC (objectid)
Media server layer
Media capability
Query
ACLs objectid token permissions
(a)
Request to access capability (appid, objectid, token)
Load balancer
API manager layer
JSON-RPC (appid, objectid, token)
Media server layer
Authorized?
Media element appid objectid tokens
(b)

Figure 3. Architecture of (a) an access-control-list-based authorization and (b) a capability-based authorization. We implemented both models in Nubomedia.

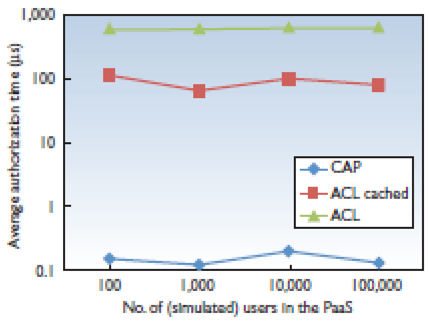

Figure 4. Experimental authorization benchmarks obtained for a running instance of Nubomedia.

Diamonds indicate the capability-based security (CAP) authorization times (local to the specific media server instance holding the accessed object).Triangles indicate access control list (ACL) authorization times when there is a shared authorization database for all API manager instances (remote database). Squares show ACL authorization times when a cache of the authorization database

is maintained on each API manager instance (local database).The experiment considers users sending requests to a pipeline with 10 media elements having five permissions each. Each experiment consists of 1,000 requests per user, 60 percent of which have positive authorization.

• The API manager layer supplies the PaaS API semantics. With Nubomedia, this layer is in charge of aspects such as locating the best media server for placing a media element or coordinating communication among media elements.

• The media server layer holds the media elements where operations such as media reception, processing, transcoding, or recording take place. It comprises several KMS instances whose number can grow or shrink to adapt to the offered load.

We can implement ACLs on top of this architecture following the model depicted in Figure 3a. We can see from the figure that the best location for the authorization logic is at the API manager layer. The tokens and their associated permissions are generated there and stored in a database, where the authorization logic can query them later.

With CAP, the best architectural option is to let media elements hold the CAP tokens themselves as part of their internal state (see Figure 3b). Hence, the authorization logic is placed directly on the media server. Unlike with ACLs, in CAP, the number of tokens per object is independent of the number of users. Consequently, each object needs to store only a fixed, small number of them.

Discussion

To compare the two proposed models' strengths and weaknesses, we generated benchmarks for our Nubomedia prototype ACL and CAP implementations. We used a test instance of Nubomedia deployed on a 100Base-T LAN with three i7 dual-core/8-Gbyte boxes executing Ubuntu 14.04 and Java 7. We based the ACL databases on MySQL v5.5.37.

As Figure 4 shows, CAP authorization has a significant advantage in terms of speed. The explanation for this is straightforward: when using ACLs, each API request must receive authorization by querying a database, which is a slow operation. Given that the Nubomedia connect primitive requires checking permissions on two different objects (that is, issuing two queries), this issue in particular hurts its performance. On the other hand, CAP authorization takes only the necessary time to compare a couple of tokens held in memory, which is much faster.

However, the proposed CAP scheme also has drawbacks. The most important is that ACLs can provide more fine-grained authorization logic. The intuition behind this is simple: our CAP model is agnostic to user identities. Hence, any authorization logic requiring user differentiation isn't possible. To understand why, imagine a simple scenario in which an application provides live WebRTC event retransmission (such as a football match). The application provider lets premium users visualize the event from different perspectives, but restricts free users to a single camera. This scenario is associated with a pipeline such as the one Figure 5 depicts. Without losing generality, we assume only two cameras (F for free and P for premium) and two users. Following our CAP scheme, the permissions the AS provides should be those Table 1 shows. However, CAP tokens might be copied and exchanged, and user P might wish to share her camera P token with user F, letting him visualize the premium content. When working with ACLs, we can avoid this. ACL tokens are associated with user sessions. Hence, we might use a quota mechanism to avoid user F accessing the premium content with user P's token.

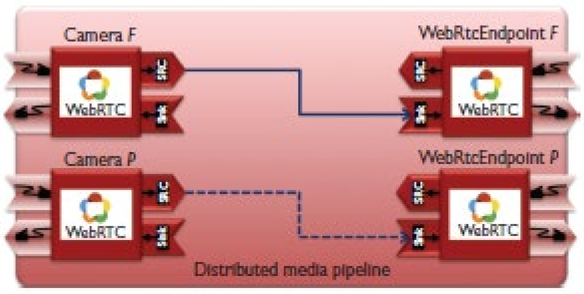

Figure 5. Media pipeline. The pipeline is associated with an application in which users might have an incentive to share capability-based security tokens.

Table 1. Capability-based security (CAP) tokens.*

Media element	CAP token	User F	User P
Camera F	READ_MEDIA	✓	✓
	WRITE_MEDIA		
Camera P	READ_MEDIA		✓
	WRITE_MEDIA		
WebRtcEndpoint F	READ_MEDIA		
	WRITE_MEDIA	✓	
WebRtcEndpoint P	READ_MEDIA		
	WRITE_MEDIA		✓

*The application server provides the tokens to different users for the event retransmission example application. The shaded cell indicates the position of the authorization breach.

In general, CAP is appropriate for applications in which users have no incentive to share their CAP tokens. Security models based on this type of principle are commonly used in the Web (that is, people don't have incentive for sharing their HTTP cookies when accessing their banks online). Hence, CAP authorization can probably be used safely in most scenarios involving WebRTC call models (such as videoconferences and group calls), but it isn't appropriate for those applications combining different levels of access to resources (free, premium, and so on), nor for applications requiring revokable permissions in a per-user scheme (for example, revoking a given permission from one user but maintaining it for others).

Acknowledgments

This work is supported by the European Commission under projects FI-WARE FP7-2011-ICT-FI, GA-285248, and NUBOMEDIA FP7-ICT-2013-1.6, GA-610576.

References

1. S. Loreto and S.P. Romano, "Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts," IEEE Internet Computing, vol. 16, no. 5, 2012, pp. 68–73.
2. A.B. Johnston and D.C. Burnett, WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web, Digital Codex, 2012.
3. IP Multimedia Subsystem (IMS); Stage 2, 3GPP tech. specification 23.228, work in progress, 2014.
4. P. Rodríguez et al., "Vaas: Videoconference as a Service," Proc. IEEE 5th Int'l Conf. Collaborative Computing: Networking, Applications, and Worksharing, 2009, pp. 1–11.
5. T. Melanchuk, An Architectural Framework for Media Server Control, IETF RFC 5567, June 2009; http://tools. ietf.org/html/rfc5567.
6. G.A. Thom, "H. 323: The Multimedia Communications Standard for Local Area Networks," IEEE Comm., vol. 34, no. 12, 1996, pp. 52–56.
7. Y. Wen et al., "Cloud Mobile Media: Reflections and Outlook," IEEE Trans. Multimedia, vol. 16, no. 4, 2014, pp. 885–902.
8. I. Ahmad et al., "Video Transcoding: An Overview of Various Techniques and Research Issues," IEEE Trans. Multimedia, vol. 7, no. 5, 2005, pp. 793–804.
9. J. Cáceres et al., "Service Scalability over the Cloud," Handbook of Cloud Computing, Springer, 2010, pp. 357–377.
10. L.M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynami- ally Scaling Applications in the Cloud," ACM SIGCOMM Computer Comm. Rev., vol. 41, no. 1, 2010, pp. 45–52.
11. D. Hardt, The OAuth 2.0 Authorization Framework, IETF RFC 6749, Oct. 2012; http://tools.ietf.org/html/ rfc6749.
12. V. Suhendra, "A Survey on Access Control Deployment," Security Technology, Springer, 2011, pp. 11–20.
13. R.S. Sandhu and P. Samarati, "Access Control: Principle and Practice," IEEE Comm., vol. 32, no. 9, 1994, pp. 40–48.