HAL open science

Security in OpenSocial-Instrumented Social Networking Services

Matthias Häsel, Luigi Lo Iacono

To cite this version:

Matthias Häsel, Luigi Lo Iacono. Security in OpenSocial-Instrumented Social Networking Services. 11th IFIP TC 6/TC 11 International Conference on Communications and Multimedia Security (CMS), May 2010, Linz, Austria. pp.40-52, 10.1007/978-3-642-13241-4_5 . hal-01056369

HAL Id: hal-01056369

https://inria.hal.science/hal-01056369v1

Submitted on 18 Aug 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Security in OpenSocial-instrumented Social Networking Services

Matthias Häsel[1] and Luigi Lo Iacono[2,⋆]

[1] XING AG, Hamburg (Germany)
matthias.haesel@xing.com
[2] Europäische Fachhochschule (EUFH), Brühl (Germany)
l.lo_iacono@eufh.de

Abstract. Securing social networking services is challenging and becomes even more complex when third-party applications are able to access user data. Still, adequate security and privacy solutions are imperative in order to build and maintain trust in such extensible social platforms. This paper discusses security issues in the context of OpenSocial-instrumented social networking services. It shows that the OpenSocial specification is far from being comprehensive in respect to security. Resulting weaknesses and shortcomings are emphasized and discussed. Finally, the paper attempts to fill these gaps by proposing extensions to the OpenSocial specification and recommendations for social networks that implement OpenSocial.

1 Introduction

Social networking services (SNS) such as Facebook, mixi or XING aim at building online communities of people who share interests and/or activities. They are technically accomplished through networking software that maps a social graph [1]. Most SNS are Web-based and build on proprietary solutions and data formats. An emerging trend is that SNS can be enriched by third-party applications that access their social graph. Interoperability is therefore an issue if these applications should reach more than a single SNS only. OpenSocial is a set of programming interfaces for developing applications that are interoperable within the context of different SNS [2].

When third-party applications are able to access user data, adequate security and privacy solutions are imperative. This paper provides an analysis of the security mechanisms integrated into OpenSocial and shows that protection is focused on data exchange between SNS and third-party applications only with protection mechanisms related to the integrity and authenticity of requests. The paper wants to raise awareness for security issues in an OpenSocial context. It gives recommendations on how to avoid flaws in implementing OpenSocial and integrating applications into an SNS by making use of standard security mechanisms which are not included in the OpenSocial specification. Finally, the paper considers more

⋆ This work was performed while Luigi Lo Iacono was with NEC Laboratories Europe, NEC Europe Ltd.

advanced security techniques such as making the access to sensitive social data transparent to the user and integrating OpenSocial applications into a platform's access control system.

2 OpenSocial-instrumented Social Networking Services

OpenSocial provides a means to extend or enrich SNS with third-party applications [2]. Using standard Web technologies, developers can create applications that run on SNS that have implemented OpenSocial. Until it was made public in November 2007, OpenSocial was driven primarily by Google, but is now managed by the non-profit OpenSocial Foundation. The Foundation consists of representatives of all major enterprises active in the social networking domain (except Facebook, which has developed a proprietary platform)[3] and specifies the APIs required for enhancing an SNS with third-party applications. The list of SNS which support OpenSocial (referred to as "containers" in OpenSocial jargon), can be accessed from the OpenSocial Website.[4] Prominent examples include iGoogle, LinkedIn, mixi, MySpace, orkut, Yahoo!, and XING.

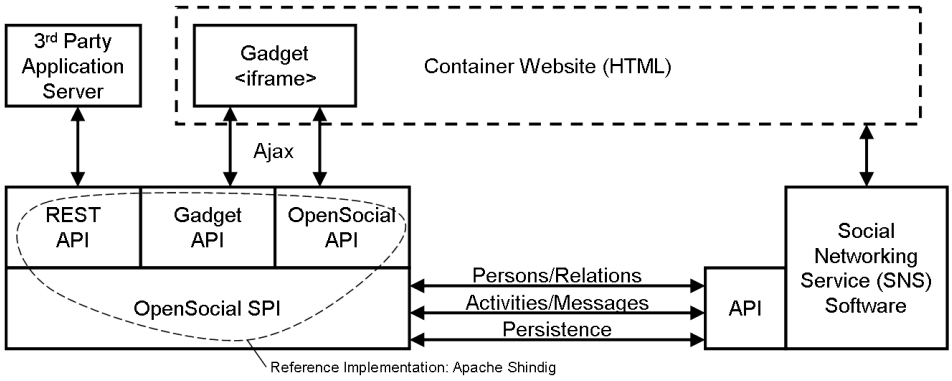

Fig. 1. OpenSocial Reference Architecture

The OpenSocial reference architecture is shown in Figure 1 and gives an overview of the involved technologies and components, as well as the relations and interactions between these components. OpenSocial applications are commonly based on the Gadget architecture[5] originally developed by Google, which has been expanded upon by interfaces enabling the access to the social data within the context of any given container. Gadgets are XML documents containing HTML and JavaScript code along with metadata. The XML specification of a Gadget is rendered by the

[3] http://developers.facebook.com/
[4] http://www.opensocial.org/
[5] http://code.google.com/intl/en/apis/gadgets/index.html

container and integrated into its own website. Communication between the Gadget and the container operates in such instances via standardized Ajax requests [3], which are defined in the `opensocial.*` and `gadgets.*` namespaces of the JavaScript-based OpenSocial [4] and Gadget [5] API respectively.

2.1 OpenSocial Compliance and Basic Services

Containers must implement all of the components shown in Figure 1 to be compliant with OpenSocial. Apache Shindig[6] is a reference implementation of the entire OpenSocial stack that operators of social network sites can refer back to. Shindig is open source and developed in both Java and PHP. The interconnection between the existing social networking software and Shindig takes place via the so-called OpenSocial Service Provider Interface (SPI), whose classes are extended in such a way that they access the networking software.

OpenSocial applications and the SNS software exchange three kinds of data which correspond to the three basic services that OpenSocial-instrumented platforms offer to their applications:

1. **People/Relationships**: The OpenSocial API enables direct access to the social graph of the container in the form of individual user objects. Two user instances that are made directly available to an application are the *viewer* and the *owner*. The viewer object refers to the current user, while the owner object is associated with the user in whose profile context the application is executed. Moreover, social applications have access to the targeted connections within the social network to be able to support the exchange of information and the interaction between users.
2. **Activities/Notifications**: Activities represent interactions between the user and the Gadget. Containers usually display a user's activities to all friends of that user within their update feeds. With Notifications, applications can make use of a container's messaging system by passing one-to-one messages to the container that will be sent on behalf of the current user.
3. **Persistence**: Data is often created during the course of user interaction that has to be saved on a persistent basis. For this purpose OpenSocial includes a persistence layer, allowing developers to store simple name/value pairs for each application or for each application and user. The container is responsible for the actual implementation of this persistence layer, which remains hidden from the social application developer.

Shindig makes available these services in its implementation of the OpenSocial JavaScript API. It also implements the Gadget JavaScript API which is used for security, communication and the user interface, and includes a Gadget rendering server that transforms the XML specification of an application into JavaScript and HTML code and makes this available via HTTP. Finally, with the OpenSocial gateway server component, Shindig provides an implementation of an additional server-side, RESTful API.

[6] http://incubator.apache.org/shindig/

2.2 OpenSocial Application Types

Three main types of OpenSocial applications can be destinguished (see Figure 2): Social mashups, social applications, and external applications.

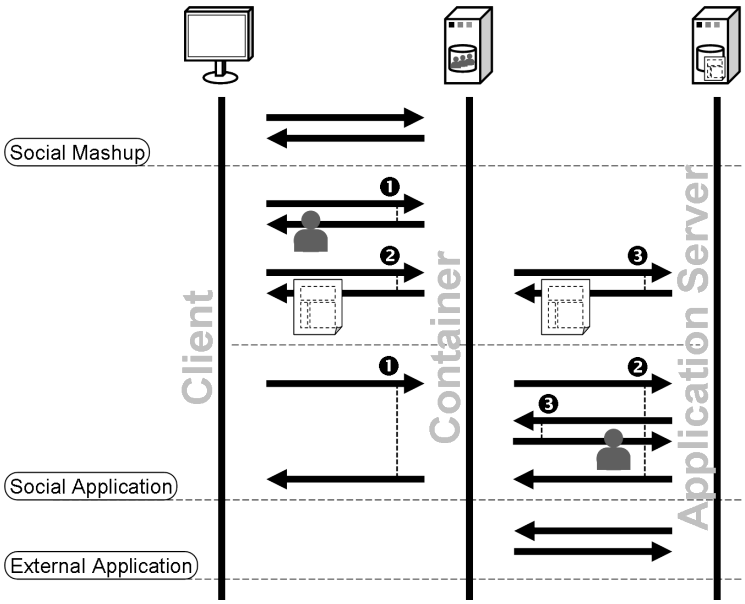

Fig. 2. OpenSocial Application Types and Associated Communication Patterns

A *social mashup* is a lightweight OpenSocial Gadget which runs inside a container. Since such applications do not rely on external third-party servers, they are extremely scalable, and the social data does not leave the protection sphere of the container. However, social mashups are limited in terms of data storage and/or processing capabilities and do not support interaction scenarios without user intervention. An example is an enhanced visualization mechanism to give users a view of all direct connections of the own social graph. A social mashup is typically created using client-side technologies only, such as HTML, JavaScript, and CSS.

A *social application*, while also being a Gadget, relies on an external third-party server for processing, storing and/or rendering data. Social applications use server-side technologies such as PHP, Python, Java, Perl, .NET, or Ruby on Rails. They provide advanced functionality and support interaction scenarios without user intervention. This comes, however, with the cost of being less scalable, and may require the processing and storage of social data by third parties outside of the protection and control sphere of the container. As an example, consider an application that allows the user to create polls, letting their friends within the container vote for different alternatives. To enable the user to choose which of their contacts should participate in the poll, the application would retrieve the owner and

his/her friends via the OpenSocial API (step 1 in the upper part of Figure 2). In the background, the user data is requested from the container's networking software via the appropriate SPI call. The application would then send the poll data and a list of the participants' user IDs to its server using a `gadgets.io.makeRequest()` call to the Gadget API (step 2), with the container proxying the data to the application server (step 3). Subsequently, the application would send out a notification to each invited friend via the OpenSocial API, which could be translated by the container into site messages containing a link to the poll. Participants clicking on that link would be identified by requesting the owner object (again, step 1), and would then be asked to vote. The application would pass the vote to its application server using another `makeRequest()` call (steps 2 and 3). Note that in this realisation, the application server does not process personal data. For the application's logic to work, it is sufficient to associate in the application server the container's user identifiers with the polls and votes created.

An alternative version of the polls example is depicted in lower part of Figure 2. As a first step, instead of utilizing the JavaScript API, the Gadget sets off a proxied `makeRequest()` call, transferring the owner's user identifier to the application server (steps 1 and 2). The application server requests the container's RESTful API (step 3) which responds with the owner and/or a list of their friends. The people data is combined with the application data, producing fragments of HTML and JavaScript that form the response to the container's request started in step 2 and the gadget's request started in step 1, respectively. Note that in this scenario, the application server processes (and may store) personal data retrieved via the RESTful API. In fact, the polls application does not require personal data to be transfered to the application server. However, there are use cases that cannot be realized with client-side APIs only. As an example, consider that polls would be limited to a certain period of time, and the application would be required to send a message (with a link to the results) to all participants after expiry. As the creator would not be interacting with the application at the time of expiry, the message needs to be triggered by the application server, rendering a client-side API useless.

Finally, an *external application* such as a website or mobile application runs outside a container, but still consumes social data through the RESTful API. Users can grant access to their personal data without needing to add the application on the SNS Website. Note that external applications are not necessarily based on the Gadget architecture. An example is an application that feeds or synchronizes the contacts database of a mobile phone with the data available from the social graph. This application type grants the most flexibility—almost all languages and platforms can take advantage of the social data—but also includes the highest risks of unauthorized access.

2.3 OpenSocial Application Lifecycle

From a developer's perspective (see left side of Figure 3), deploying a social mashup or application is equivalent to passing its XML specification to a container. Most containers require that developers submit their application to be reviewed before it is made available in their directory. Containers have different processes for granting

developers access to their sandbox environments and reviewing applications. On Orkut, for example, developers can sign up for a developer sandbox and submit their application online, while on XING developers have to hand in a product concept before they are granted access to the sandbox.

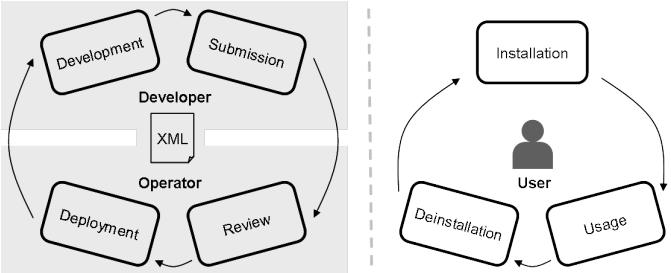

Fig. 3. OpenSocial Application Lifecycle

From a user's perspective (see right side of Figure 3), first-time usage of an OpenSocial application usually involves some kind of installation process whereby the user explicitly permits the application to access the data in their personal user profile. Events such as installation or deinstallation are desirable yet difficult for developers to track. The OpenSocial specification therefore allows developers to specify URLs that the container will POST event data to when these events are triggered. Besides installation and deinstallation, such events include rate limiting, directory listing changes, and blacklist/whitelist notifications. By tracking the data sent to the specified URL, application providers can accurately track the number of installs, remove database entries upon uninstall, and get automatic notifications if their application exceeds a quota or is marked as spam by the container.

3 Analysis of the OpenSocial Security Specifications

Security is a crucial aspect in SNS due to the sensitive and personal nature of the social data exchanged via these platforms. In the context of OpenSocial-instrumented SNS, security becomes even more important because third-party applications (possibly) access the social data. This section will introduce and analyse the built-in security mechanisms in the OpenSocial specification.

The focus of this analysis is on OpenSocial-specific issues. More general security problems related to Rich Internet Applications (RIA) and Web applications in general such as Cross-Site Scripting (XSS) or issues related to the execution of client-side JavaScript code are not included. Such aspects are not unique to OpenSocial and subject of other initiatives such as the Open Web Application Security Project (OWASP) [6].

3.1 Focus of the OpenSocial Specification regarding Security

Taking the application types and their communication patterns introduced in Section 2 into account, it becomes clear that OpenSocial primarily introduces the communication path between the container and the third-party application server. The communication link between the container and the client is the SNS proprietary channel. Security considerations within the OpenSocial specification are therefore not targeted to the SNS proprietary link as it is seen as a container-specific implementation detail and assumed to be protected by appropriate policies and means. In the case of a social mashup or application e.g., the content of the Gadget XML—which contains HTML and JavaScript code—is rendered into the container's Website. To prevent DOM manipulations, the Gadget is usually embedded into an `<iframe>` served from an alternate domain.

The container is involved in all communications and in some cases it is mediating from the client-side application to the application server. Thus, the container plays a decisive role in terms of security and privacy especially in the presence of external application servers and should therefore carefully inspect all communications and strictly enforce its policies. This is manifested in the OpenSocial specification which recommends that the container should always validate the transferred parameters before passing them to the application server.

3.2 Message Integrity and Authentication

The communication between the container and the application server will most commonly include at least the IDs of the users currently interacting with an OpenSocial application on the container (see Section 2). Since `makeRequest()` calls are just JavaScript transmitted via HTTP GET or POST, it is possible for any user to create `makeRequest()` calls with whatever arguments they wish [7].

To protect IDs from manipulations by malicious users and to ensure that parameters truly originate from the container, OpenSocial contains a method to communicate IDs to the application server in a verifiable manner by using the OAuth API authorization protocol [8]. More specifically, requests are signed based upon OAuth's parameter signing mechanism. Through the method of parameter signing, it is possible for an application to request the container to send IDs along with a digital signature that allows third-party servers to verify that the parameters passed are legitimate. Note, however, that the response flow from the application server back to the client-side application is not protected by any OpenSocial-specific security mechanisms.

When a request is signed by the container, it adds additional data before forwarding it to the remote application server. This data contains information such as the IDs of the application making the request, the owner and viewer, and information that the application server can use to verify that the information added was not tampered with since the container sent the request. Two signature algorithms are specified: the secret-key based HMAC-SHA1 [9] and the public-key based RSA-SHA1 [9] method. To generate the signature, a so-called Signature Base String is

generated first, which is a consistent reproducible concatenation of the request elements into a single string. The parameters contained in the POST or GET request (`oauth_signature` excluded) are ordered and concatenated into a normalized string which finally forms the input to the digital signature algorithm. An example for the required steps to generate the signature base string and the signature is shown in Figure 4.

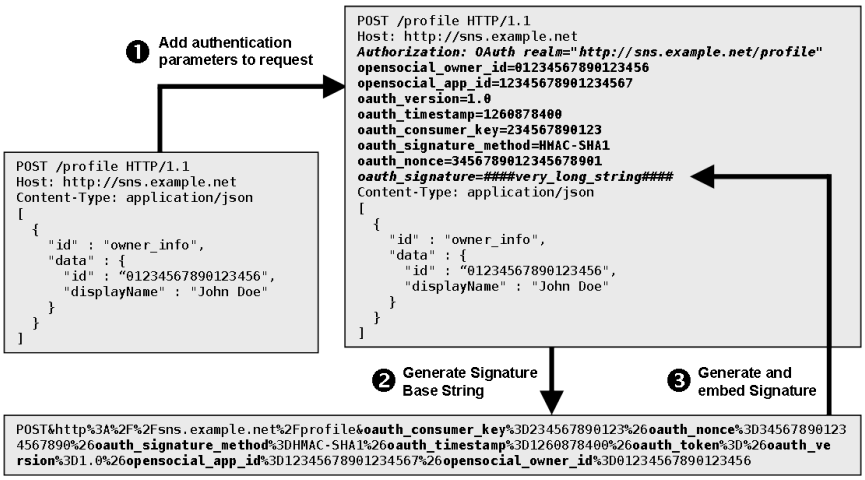

Fig. 4. Signature Generation based on the OAuth Parameter Signing Mechanism

Note that the OAuth parameter signing algorithm is applied to the HTTP parameters, but not to the application data (here JSON-encoded) included in the POST as depicted in Figure 4. Assume that the owner ID in the application data is altered by the user and that the container sets the owner ID to the HTTP parameters according to some context information established for the session with the user. If the container in this scenario does not cross-check whether the same owner ID appears in the application data as the one stored in the HTTP parameters, the parameter signing might get meaningless. If this happens, the owner ID is sent twice to the application server (once signed in the HTTP parameters and once unsigned in the application data). In this case the developer needs to compare these IDs carefully in his verification code. If this is not performed and the application server code verifies the signed parameters and executes the business logic based on the application data without cross-checking the consistency of IDs, the forgery of IDs and henceforth the unauthorized retrieval of social data is still feasible. Such an attack is very similar to the XML Signature Wrapping attack which was first described by Fournet at the DIMACS workshop in 2005 [10] and first published by McIntosh and Austel at SWS workshop in 2005 [11]. This threat renders a large set of applications that rely on the use of XML Signature [12] as a security enforcement technology critically insecure in cases in which the validation procedure does not

carefully perform other security-related checks in addition to the pure verification of the digital signature. The real-world impact of this threat was first shown by Gruschka and Lo Iacono at the ICWS conference in 2009 [13] when they described a similar vulnerability in the Amazon EC2 Cloud services.

3.3 Message Confidentiality

In order to protect the data flow against unauthorized access during the transmission and to maintain it confidential in the presence of passive attacks performed by eavesdroppers, communication needs to be encrypted. As in the case of the integration and authentication of requests, container-specific mechanisms are deployed for the communication between the client-side application and the container. For the communication between the container and the application server the OpenSocial specification does not state how to protect messages against eavesdropping, not even in terms of guidelines or recommendations. It is up to the container to define, implement and enforce an appropriate security policy.

3.4 Identity Management and Access Control

To ensure that the requestor identity cannot be spoofed in a client initiated data flow, OpenSocial containers usually expect a short lived security token that is made up by the owner ID, viewer ID, and application ID, among others. As far as a rendered application is concerned, the token should be entirely opaque; it should only be interpretable by the container's Gadget renderer and APIs. However, the OpenSocial specification itself does not state any details on how this token should be encoded. Neither does it give a recommendation on the token lifetime or its encryption.

Similarly, security tokens can be applied in an application server initiated data flow. In this scenario, the client retrieves the token via the JavaScript API and sends it to the application server via a proxied `makeRequest()` call. The application server using the security token can then make calls to the container's RESTful API, passing the security token as a URL parameter or within the HTTP request header. Again, OpenSocial does not provide any recommendations on identity management in a RESTful scenario. E.g., containers could use OAuth to validate whether a request has been initiated by a certain application server. Note that in such as scenario, the application server is able to arbitrarily set the identity of the user for whom the data is requested.

4 Recommended Security Improvements and Extensions

This section takes up the identified security issues of OpenSocial and provides suggestions on how to improve the current situation. The following proposals include both enhancements to the OpenSocial specification and recommendations to SNS that implement OpenSocial.

4.1 Application Type Selection

In Section 2.2 and 3.1 it became clear that social mashups have advantages in respect to security and privacy in comparison to the two other application types, since no social data is transmitted to an external server, but the personal data remains inside the protection sphere of the container. Thus, it is recommended that the container as well as the user should carefully examine the underlying communication pattern of each application before deciding to integrate or make use of it, respectively. For the container, this means that an acceptable procedure needs to be in place which controls and governs the integration of applications, i.e., the container needs to ensure during the review phase that an OpenSocial applications does not access more user data than required in order to protect its users against unauthorized data gathering.

4.2 Data Minimisation and Pseudonymisation

An important aspect SNS operators and users should check before deploying or installing an application respectively, is the kind of social data that is going to be used by the application outside the container's premises (if any). It needs to be controlled and ensured that the released data items are the minimum required to perform the task. Minimising the data amount exchanged with external entities is a measure to reduce the risk of unauthorized data gathering and data aggregation. This should be included in the specifications as a recommendation to make SNS operators aware of this risk.

To further increase privacy protection, the following additional aspect must be considered by the OpenSocial specification and the container. In most cases, applications require some form of ID in order to perform their tasks (see the polls example in Section 2.2). The OpenSocial specification does not define how this ID should look like. It is therefore possible that a container simply uses its internal user IDs, opening the door for coalition or multi-application attacks in which a single entity (by using a set of applications or a set of entities forming a coalition) will be able to correlate the obtained data using these IDs. To prevent such kind of attacks and to protect the users' privacy more effectively, the specification should recommend not to use internal IDs since these are personal identifiable information, but to replace them with application-specific pseudonyms instead.

4.3 Application Integration and Code Signing

As introduced in Section 3.1, `<iframe>` elements are a common means to integrate social mashups and applications into the container's Website without allowing client side scripts to perform attacks. Since this is not explicitly stated in the specification, an according recommendation should be added to raise the awareness and point SNS operators to additional resources on this topic.

Another issue is that the JavaScript code is often not stored at the container, but loaded from the Web. The JavaScript code within the Gadget XML specification is commonly reduced to a URL referencing the actual source. This renders the

review process of the SNS operator meaningless if the application code can easily be altered after it has been reviewed and added to the container. The adoption of code signing technology are required here, so that the source code can be verified by the container and in case of a dispute, the developer can not deny having released the code providing the necessary evidence for liability cases.

4.4 Communication Security

As discussed before, the specified means to protect the communication in an OpenSocial setting are limited to parameter signing of HTTP requests. No means to protect the HTTP responses are included, which might open doors for new attacks.

Moreover, OAuth targets security service integrity and authentication, but does not provide a guideline how to protect the confidentiality of user data. Since communication between the client-side application and the container is protected by container-specific means, OpenSocial does not interfere here. Still, for communication between the container and the application server, the specification should provide a guideline how to encrypt data. This is not only required to raise the awareness in confidentiality issues among SNS operators and application developers, but also in order to reduce the variety of different approaches that will be taken by distinct containers, which is contradicting one of the goals of OpenSocial, to create a write-once-run-everywhere (or at least learn-once-write-anywhere) environment since application developers need to adapt to container-specific demands.

The OpenSocial specification must be enhanced to additionally sign responses to enable the container to verify integrity and authenticity. Further means are out of scope of the OpenSocial specification, but still required in order to reach a certain level of security as well as compatibility among containers. Thus, the following recommendations should be carefully considered by SNS operators:

- Use HTTPS for Gadget rendering, Gadget JavScript API, OpenSocial JavaScript API, and RESTful API
- Make HTTPS and signed messages (including event messages) mandatory for application providers' APIs (especially for social and external applications)
- Compare IDs in the signed parameters with the ones used in the unsigned application data and advise application providers to so as well
- Make sure that security tokens are implemented in a safe way

Some of these recommendations are out of the control sphere of the container. Still, containers have means to check and enforce such rules by analyzing the Gadget's XML specification.

4.5 Access Control and Delegation

Access to social data by external entities via the RESTful API is an aspect to be carefully considered. Unfortunately, the OpenSocial specification does not give much guidance here. Authentication of external entities such as application servers or mobile applications is handled by mechanisms established by the container and

initiated during the application deployment process. Further aspects should be defined within the specification to form a basis for effective and cross-platform access control systems. This includes mechanisms to check whether a user granted access to his/her personal data to a specific application. The specification should recommend that in any case, a container should make personal data only available to external entities of such users who have explicitly provided their consent (opt-in).

For more fine-grained access policies that limit the access to particular data items, appropriate mechanisms still need to be defined and specified. An initial approach for a more fine-grain access to user data by OpenSocial applications has been developed by StudiVZ.[7] The basic idea is that the user can pass "configurable" user profiles (so-called ID cards) to the application and by this means control the disclosure of their profile data (also a means to minimize the exposed data as stated in Section 4.2). Although this approach gives the user a maximum of control, it increases complexity for the user and may decrease the value provided by an application or might even effect its functioning due to the lack of relevant data. Practice will show how this approach will be accepted by users and application providers.

Delegation of access rights to external entities is another point which requires a more indepth analysis and appropriate specification. In case of external applications, standard three-legged OAuth mechanisms can be used [8]. In cases where an external entity needs to access social data without a user interaction, the definition of delegation tokens which can be used in conjunction with the two-legged OAuth protocol is still an open issue.

4.6 Transparency Enhancements

Crucial questions regarding privacy are whether, why and how an application processes and/or stores what kinds of personal data. Unfortunately, a procedure which empowers the user to make an informed decision whether to install an application is not in the scope of the OpenSocial specification. As a result, applications make warning statements (if at all) in an proprietary manner, requiring the user to carefully read and understand their privacy policies. In the case that the data leaving the control sphere of the container is made explicit to the user at the time of installation, it is often not possible for the user to reassess the privacy policy during application lifetime. The extension of the Gadget XML specification to include such information would provide a standardized way of describing whether user data will be processed or stored by a third party. The container would be able to extract this information and present it in a container-specific and consistent manner, providing decision support to the user.

5 Conclusions

Securing SNS is challenging and becomes even more complex when third-party applications are able to access a user's data. This paper analyzed and discussed

[7] http://www.vzlog.de/2009/10/neue-details-zu-opensocial-bei-studivz/ (German)

security implications in the context of OpenSocial-instrumented SNS. It showed that the OpenSocial specification is far from being comprehensive in respect to security. The paper introduced issues and depicted possible vulnerabilities resulting from these shortcomings. It presented different approaches to fill these gaps and proposed additions to the OpenSocial specification as well as a set of recommendations for containers that implement OpenSocial.

Although some of the proposals are still in an early stage and certainly require further research and development efforts, the paper contributes findings to raise awareness for existing security issues and provides suggestions and recommendations for resolving these issues. The results and suggestions will be brought to the attention of the OpenSocial Foundation with the goal to enhance upcoming versions of the specification and encourage the creation of guidelines for non-normative aspects.

References

1. Boyd, D., Ellison, N.: Social network sites: History, and scholarship. Journal of Computer-Mediated Communication **13**(1) (2007) 210–230
2. Häsel, M.: Opensocial: An enabler for social applications on the web. Communications of the ACM **nn**(n) (2010) nn–nn
3. Garrett, J.: Ajax: A new approach to web applications. Technical report, Adaptive Path Inc. (2005)
4. OpenSocial and Gadgets Specification Group: Opensocial specification v0.9. Technical report, OpenSocial Foundation (April 2009)
5. OpenSocial and Gadgets Specification Group: Opensocial gadgets api specification v0.9. Technical report, OpenSocial Foundation (April 2009)
6. Wiesmann, A., van der Stock, A., Curphey, M., Stirbei, R., eds.: A Guide to Building Secure Web Applications and Web Services. The Open Web Application Security Project (2005)
7. Arrington, M.: First opensocial application hacked within 45 minutes Available online at: http://www.techcrunch.com/2007/11/02/first-opensocial-application-hacked-within-45-minutes/ (last accessed on Nov. 27, 2009).
8. Hammer-Lahav, E., ed.: OAuth Core 1.0 Revision A. (2009)
9. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press (2001)
10. Fournet, C.: Verification tools for web services security. In: DIMACS Workshop on Security of Web Services and E-Commerce. (2005)
11. McIntosh, M., Austel, P.: XML signature element wrapping attacks and countermeasures. In: SWS '05: Proceedings of the 2005 Workshop on Secure Web Services, New York, NY, USA, ACM Press (2005) 20–27
12. Bartel, M., Boyer, J., Fox, B., LaMacchia, B., Simon, E.: XML-Signature Syntax and Processing. W3C Recommendation (2002)
13. Gruschka, N., Lo Iacono, L.: Vulnerable cloud: Soap message security validation revisited. In: Proceedings of the IEEE International Conference on Web Services (ICWS). (2009)