Server-driven Outbound Web-application Sandboxing
FP7-ICT-2009-5, Project No. 256964

https://www.websand.eu

Deliverable D2.1
Secure Interaction Specification

Abstract

This deliverable reports on the results of the WebSand work package on Secure Web Interaction. It provides a wording definition and technical specification of the term and builds the basis for the development of the server-side framework that achieves Secure Web Interaction.

Deliverable details

Deliverable version: *v1.0* Classification: *public*
Date of delivery: *30.09.2011* Due on: *30.09.2011*
Editors: *Bastian Braun* Total pages: *51*

List of Contributors:
Bastian Braun, Philippe De Ryck, Lieven Desmet, Stefan Gentsch, Wouter Joosen, Martin Johns, Sebastian Lekies, Peng Liu, Nick Nikiforakis, Frank Piessens, Joachim Posegga, Walter Tighzert, Steven Van Acker, Jan Wolff

Project details

Start date: *October 01, 2010* Duration: *36 months*
Project Coordinator: *Martin Johns*
Partners: SAP, Siemens, CHALMERS, KUL, UNI PASSAU

Executive Summary

After the consolidation of the state-of-the-art, the present deliverable D2.1 provides details about secure interaction specification. These details include the requirements on the server-side WebSand framework which is supposed to provide secure web interaction, the addressing mechanism of this framework, and the policy attributes that rule the secure handling of requests by the framework. As a next step, we will develop and implement the framework prototype. The documentation and first case study on the policy enforcement and requirements on the client side will be given in the next deliverable D2.2. This includes the applicability of the addressing mechanism, the policy attributes, and the operational requirements as they are identified in this document.

In this first deliverable of WP2, we start with defining the term *Secure Web Interaction* (Sec. 1) which is crucial to be clear for the following sections and the upcoming activities in WP2. We argue that Secure Web Interaction is ensured if all incoming requests carry information that allows the Web server to uniquely determine the session and cross-domain context, track authentication and authorization, and enforce control-flow integrity.

Five real-world scenarios serve as a benchmark to show the effectiveness of our proposed means to achieve secure web interaction (Section 2). They deal with distributed two-factor authentication and authorization, authorization delegation with OAuth, cross-domain interaction, control flow over several domains, and race conditions respectively.

The basis for advanced approaches in the course of secure web interaction is session security. We summarize our recent progress in that field. To sum up, we identified issues in the usage of cookies as session authentication credentials and provided countermeasures to Cross-Site Scripting (XSS) and Session Fixation attacks. These attacks aim to steal an authenticated session either by gaining knowledge of the session cookie of the victim or by setting an attacker-controlled session cookie at the victim. Finally, an often overseen problem related to server-side session storage has been identified and will be avoided in the WebSand framework implementation. This problem allows the attacker to share his authentication status between different applications that are hosted on the same server.

The specification of secure web interaction in terms of operational and policy requirements for authentication, authorization, cross-domain communication, and control-flow integrity is given in Section 4. With operational requirements, we define requirements on the functionality of the WebSand framework. Such requirements are, for instance, an API for the application to provide the control-flow graph to the framework. On the other hand, pol-

icy requirements describe prerequisites on the policy language concerning its expressive power and enforcement capabilities.

In Section 5, we define our customized addressing mechanism. It allows fine-grained and unambiguous identification of resources of the application's external interface. For standard conformity, clients address all resources by URLs. An internal, application-specific component, the Gatekeeper, translates the incoming HTTP requests into framework-compatible request objects including the requested resource and respective request attributes like HTTP headers and URL parameters. The addressing mechanism is essential for creating policies that can define properties, attributes, and restrictions in respect to client interaction with these resources. The policies address the requests' attributes and are enforced based on the internal request object representation.

Section 6 unifies the required policy attributes needed for specifying the security requirements with respect to authentication, authorization, cross-domain communication, and control-flow integrity. We demonstrate that the policy attributes have the ability to define secure web interaction with respect to the scenarios given in Section 2.

Contents

List of Figures

1 Introduction

1.1 Motivation

The Web is ever evolving. There are now multitudes of server-side Web technologies and on the client-side the Web browsers are constantly expanding the set of available capabilities and features. Furthermore, the initial two-party client/server model of the Web makes place for an increasing number of multi-party scenarios.

However, the underlying basic interaction scheme of the Web has not significantly changed since its birth in 1990. The client(s) and server(s) communicate using a series of HTTP request/response pairs. In addition, one of the defining characteristics of HTTP has also not changed: The inherent statelessness of the HTTP protocol and its inability to provide reliable context information which span more than one single request/response-pair.

As a consequence, up to this date, it is the Web application's duty to ensure that basic security sensitive characteristics, such as robust session management or authentication tracking, are enforced. This enforcement happens solely on the application layer without any substantial support from the protocol layers. All the application's external interface sees is a stream of independent incoming HTTP requests which might or might not belong to a preexisting user/session/authentication context. The precise security characteristics for each incoming HTTP request have yet to be determined. The only information the server can rely on for this process, are its current server-side state and the information contained in the HTTP request itself. Thus, it is the application's (or the utilized application framework's) responsibility to ensure that all necessary information are available in the incoming HTTP requests, which are needed to allow correct and secure reactions.

This deliverable explores the features and capabilities of the WebSand framework in respect to enforcement of robust, policy-driven secure Web interaction, to enable the server-side application's HTTP handling to meet the security challenges of modern Web applications.

1.2 Secure Web Interaction

In the context of WebSand, we summarize all security decisions and actions that directly relate to incoming or outgoing HTTP traffic under the term *Secure Web Interaction*.

1.2.1 The four pillars of Secure Web Interaction

Within WebSand, we concentrate on the main areas of Web interaction:

- **Authentication and Authorization:** For each incoming HTTP request the authentication/authorization context has to be established prior to causing any state changing action on the server-side and prior to generating any HTTP response. This in particular includes non-trivial scenarios, which might implement features such as federated identity management, authorization delegation, or two-factor authentication.

- **Cross-domain interaction:**[1] In the past, the Web's same-origin policy was preventing all client-side cross-domain HTTP interaction. With the introduction of modern browser-based capabilities, such as Flash's `crossdomain.xml`-mechanism or CORS [20], this has changed fundamentally. New paradigms have been widely adopted in the meantime (see [9] for current deployment figures). These expanded interaction capabilities create new security challenges. Hence, if permitted by the application, cross-domain interaction policies have to be well maintained and well integrated in the whole server-side security interaction model.

- **Control-flow integrity:** In the context of Web applications, any given control-flow consists of a series of connected HTTP requests/responses. Each of these incoming requests corresponds to one to-be-committed server-side actions. Modern Web applications utilize sophisticated workflows that require the enforcement of strict control-flow requirements, in order to ensure the integrity of the server-side state of the respective workflow. HTTP provides no means to enforce such request channeling. Hence, malicious parties could aim to send HTTP requests out of their intended order to create server-side state confusion, which might lead to security-sensitive issues. In consequence, the Web application itself or the Web application's underlying framework has to conduct the control-flow enforcement.

In addition, it is crucial to embed all Web interaction security solutions into a framework that provides **Secure Session Management**. Secure

[1]Please note: Whenever we refer to *cross-domain interaction* in this document, we refer to client-side cross-domain communication, i.e. client-side code from one domain context requests content from a different domain. Server-driven cross-domain flows are handled as part of the control-flow integrity considerations.

session management is the fundamental pillar on which all further Web interaction is based. If this foundation is faulty, all further security measures may fail (e.g., as we have demonstrated in [13]).

1.2.2 The WebSand approach to Secure Web Interaction

The policy-driven solution, which we will develop in this document, is based on a fundamental observation: *All four major fields of secure Web interaction are, on a technical level, handled in a mostly identical fashion*: Identifier tokens are obtained from the data contained in the received HTTP request in order to determine the context in which the request has to be interpreted. Based on the values of the tokens, the corresponding security decision is taken.

While the precise nature of such tokens can vary (the applicable set includes at least cryptographically generated tokens, random nonces, and domain values), the basic mechanism is the same for all four purposes, namely session management, authentication/authorization tracking, cross-domain interaction, and control-flow enforcement.

This observation allows us to design a solution which handles all aspects of secure Web interaction in a unified fashion and to introduce a single reference monitor which is the central point of enforcement.

1.3 Organisation

This document is organized as follows: In Section 2, we present five generic application scenarios which illustrate the various facets of secure Web interaction and help to identify the connected security challenges. These scenarios will be used through the remainder of the deliverable to aid the understanding of the presented solution design. In Section 3, we document our findings in the field of secure session management, which will significantly influence the design of the WebSand framework. Based on the presented scenarios, we deduct operational and policy requirements in Section 4 which have to be fulfilled by the WebSand framework. In Section 5, we present a first architectural outline of WebSand's Web interaction enforcement. In particular, we discuss the two main components: A modular gatekeeper, which serves as the central reference monitor for incoming HTTP requests, and a uniform addressing mechanism that allows reliable assignment of request resources to policy rules. We provide the information at this point, as an insight in the mechanism's basic architecture is helpful to understand the secure Web interaction policy mechanism, which we then present in Section 6. The policy mechanism takes up the identified policy requirements from Section 4 and integrates them into a uniform, declarative format. Subsequently, we show how this policy mechanism can be utilized in connection with the outlined enforcement architecture to solve the security challenges of Section 2's application scenarios. We end the document with a conclusion in Section 7.

1.4 Related WebSand Publications

Several of the insights and solutions that we will outline in this document have been supported and substantiated by academic WebSand publications. Please refer to [19, 7, 9, 8, 13, 14] for details.

2 Overview of Web Interaction Scenarios

This section provides an overview over the scenarios that are used as a basis for deriving concrete requirements for work package 2. Each scenario presents a special use case that should be implementable in a secure fashion with the help of the WebSand framework.

2.1 Mobile Apps for Enhanced Security

In different scenarios, mobile phones are used as a trusted, secondary communication channel and as a second factor for authentication and authorization. For web mashups, a mobile app can provide a trusted channel that allows to implement stronger authentication and authorization mechanisms, for example two-factor authentication, user consent, and separation of duties (SoD). These security features are relevant for applications that have high security requirements, which includes financial applications and applications handling highly sensitive personal information, e.g. electronic health systems.

The WebSand framework is supposed provide functionality to implement enhanced security checks based on second factor authentication and authorization. Existing examples include the Google Authenticator, MobileOTP and online banking systems using mobile transaction numbers (mTAN, sm-sTAN). In the following, the Google Authenticator example is described in more detail to illustrate the usage scenario.

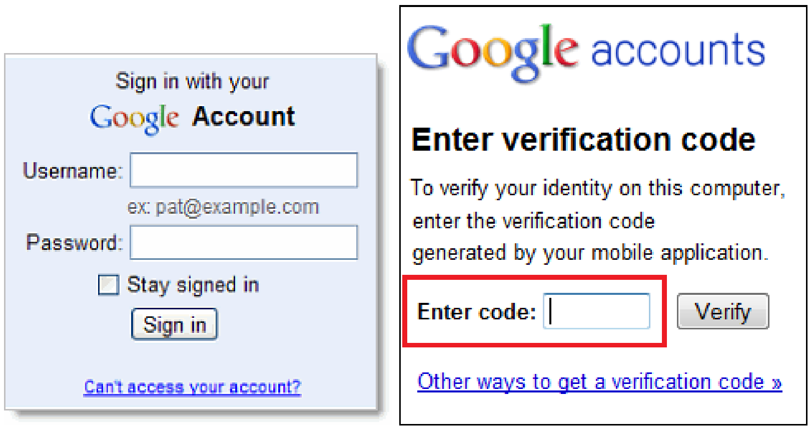
The Google Authenticator [4] provides 2-step authentication using mobile devices and one-time passcodes. The passcodes are generated based on the HMAC-Based One-time Password (HOTP) algorithm [11] and the Time-based One-time Password (TOTP) algorithm [12].

Implementations are available as apps for Android and Blackberry phones and as a PAM-module (Pluggable Authentication Modules).

The two-phase process provides stronger authentication than the default password-based login for access to a Google account, all the more if performed on an untrusted device like a public PC, e.g. in hotels and Internet cafes.

2.2 Delegation of Privilege in Distributed Workflows

Mashups that involve back-end communication of mashup components must provide a mechanism for delegation of a set of access control privileges. An example of this delegation of user privilege is used by Facebook and Twitter and makes use of the OAuth protocol [2]. The OAuth protocol is designed to provide a simple and secure way for users of a web application, in our case Twitter, to grant access privilege to a third party, in our case Facebook, to



Sign in with your Google Account

Username:

ex: pat@example.com

Password:

Stay signed in

Sign in

Can't access your account?

Google accounts

Enter verification code

To verify your identity on this computer, enter the verification code generated by your mobile application.

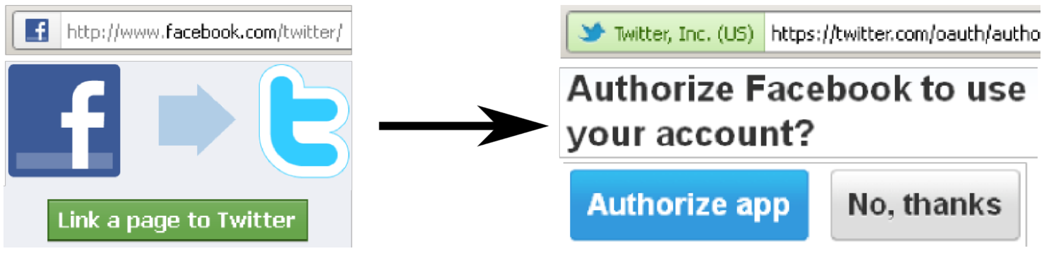
Enter code:

Verify

Other ways to get a verification code »

Figure 1: Graphical user interface of the Google Authenticator

their data and resources without forwarding their authentication credentials. In our example, the user wants that the posts he creates on Facebook should automatically also update his status on Twitter.



http://www.facebook.com/twitter/

Link a page to Twitter

Twitter, Inc. (US) https://twitter.com/oauth/autho

Authorize Facebook to use your account?

Authorize app

No, thanks

Figure 2: OAuth workflow for automatic Twitter updates originating from Facebook.

Figure 2 shows a screenshot of the related functionality on Facebook and the confirmation page on Twitter. To start the authorization workflow, the user clicks on the button with the label “Link a page to Twitter”. Facebook now communicates with the Twitter OAuth Service in the back-end and requests a so called ‘request_token’. After receiving the token, Facebook redirects the browser window of the user to the Twitter web page, where he has to authenticate. On Twitter, the user is asked whether he authorizes the requesting domain ‘facebook.com’ to have access to his Twitter account. After giving his consent, Facebook is able to use the ‘request_token’ to request an ‘access_token’, which allows access to the Twitter account on behalf of the user.

In the WebSand framework, support for OAuth will be implemented allowing delegation of access privileges to other mashup components. The focus will be on the case that a WebSand-equipped web application gives access to a user's personal account. Mashup components will thus be enabled to access another application's resources on behalf of the user as part of back-end workflows.

2.3 Cross-domain Interaction

In the recent years on-demand solutions such as SAP's Business By Design are becoming more and more important. By being provided over the Internet these solutions offer companies more flexibility when designing their internal IT systems. Instead of buying and installing software on-premise, the on-demand solution can be rented for a certain amount of time and be accessed through a web browser. As information systems often need to communicate with each other, a communication channel is needed between on-demand and on-premise systems. Often, however, there are multiple network barriers such as firewalls between those systems. Hence, alternative communication channels such as cross-domain requests are used in order to connect remote systems.

In the following, we consider an on-demand application that provides a feature to display a personalized catalogue, in which the customer can view items that are currently running low or are out of stocks in his warehouse. Furthermore, the customer is able to receive recommendations for items that might be of interest to him. In order to gain the necessary warehouse data the on-demand system queries the customer's on-premise warehouse system via client-side cross-domain requests through the user's web browser. In order to enable this indirect communication channel the on-premise system has to open up parts of its API to client-side cross-domain requests by setting up a so-called cross-domain policy. This can be accomplished by several technologies such as Adobe Flash, Silverlight or Cross-Origin Resource Sharing.

In order to support old legacy browsers as well as modern mobile browsers, the several different approaches have to be combined. So the server (here the on-premise system) has to setup multiple different policies for different environments. As each policy format is very different and as there are multiple security pitfalls in each technology, it is tedious to manually maintain these policies in a secure fashion. The WebSand framework will address this issue by offering an automatic way to maintain the different policies while avoiding the security pitfalls.

2.4 Online Shopping Workflow over Two Domains

A merchant implements the online shopping workflow following his business logic. The payment step, however, is external from his point of view. So, he redirects his customers to a *Cashier-as-a-Service (CaaS)*. The CaaS is a service provider that takes money from customers and passes it on to the merchant after discounting his rate. So, after being redirected to the CaaS, customers are supposed to supply payment details which are checked by the CaaS. Then, the workflow drives customers back to the merchant's domain. The merchant expects a payment confirmation by the CaaS. In case of success, customers receive an approval notification and the checkout details are displayed.

The employment of the WebSand framework is supposed to be at the merchant's site. The workflow step to the CaaS as well as the redirection back to the merchant's domain contain a number of pitfalls. For example, data that is part of the redirection must not be reusable and the respective customer and amount of money must be verifiable.

Popular examples for such a CaaS include Google Checkout [5], Amazon Payments [1], and PayPal [17] (cf. Figure 3).

Besides self-developed online stores, there are off-the-shelf merchant systems, e.g. nopCommerce [15] and Interspire [6]. However, the WebSand framework is supposed to be independent of the merchant application in this scenario.

2.5 Web Portal to Send Limited Number of Text Messages

A telco provider offers its customers to send a maximum number of text messages (SMS) per day for free. Provided that they access the service via its mobile or wired network, another authentication is not needed. For other access paths, customers have to authenticate (login) first. Then, they can access the text message preparation form. After providing the recipients' phone numbers and the respective text, the form is sent to the provider's gateway in order to be processed. However, sending of more messages is denied when the threshold is reached.

The provider stores the respective amount of sent messages in a database. Before a message is processed, the current quota is checked. The message is sent if the threshold is not yet reached. After successfully sending the message, the server increases the number of sent messages in the database.

Modern multi-threaded web servers process different requests almost in parallel. Malicious customers could try to send a large number of requests

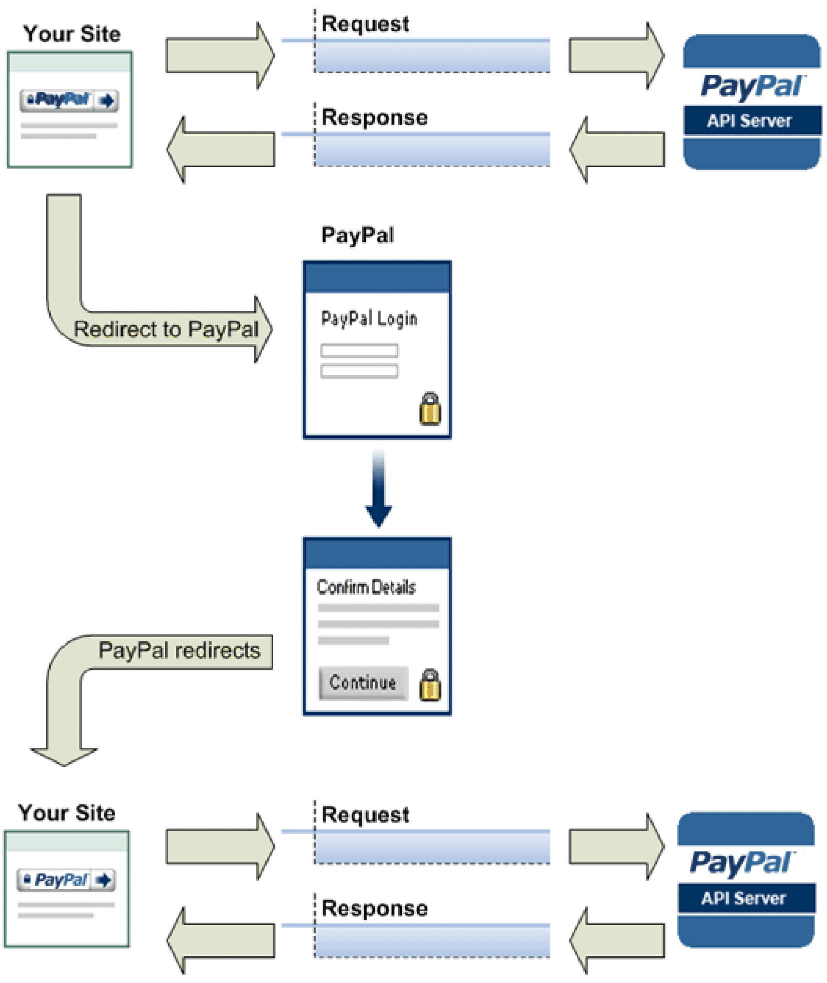


Figure 3: Cross-Domain Workflow for a Web Shop with PayPal[2]

at the same time to exploit a race condition. In this scenario, more messages might be processed (and thus sent) in a short time before the database record is updated to the threshold value because the database update happens after sending the message to prevent counting failed transmissions.

The problem is even more complicated due to the fact that web users can access the application's API (i.e. URLs) in arbitrary sequence. This way, the presence of race conditions in web applications is harder to detect than in local programs.

There are numerous providers for sending free SMS in the Web. In fact, most of them are localized. The above described scenario is taken from a real world case study [16].

3 Session Security

A solution that enables the enforcement of secure web interaction has to rely on one basic pillar: Session Security. The concept of session tracking is indispensable for the web application paradigm as most security critical interactions and decisions are taken based on the identity of a user. This identity is proven to the server by presenting a valid session ID that is only known to the legitimate user. However, in the wild many attacks such as session fixation exist that aim on hijacking a user's session in order to conduct malicious actions in his name. If an attacker is able to obtain a valid session, the server is by no means able to decide whether a request came from a legitimate user or not. For mechanisms that enforce secure web interaction this is a major point of concern as they heavily rely on the legitimacy of a requestor. Hence without a secure session handling such mechanisms are useless. Therefore, this section explores pitfalls and protection mechanisms regarding session security that will be implemented within the WebSand framework to enforce secure session handling. Thereby, this section is based on three papers created within the WebSand project [7, 14, 13]. If we are able to identify further threats regarding session management, we will address these threats in future work during the Websand project.

3.1 Separation of Application and Session Data

One very common attack nowadays is cross-site scripting (XSS). A main goal of such an attack is to steal a user's session identifier. In order to do so, an attacker injects a piece of JavaScript code into a website that reads out the session cookie via the *document.cookie* directive and sends the received value to a web page that is controlled by the attacker. Later on, the adversary can use the obtained cookie in order to impersonate a legitimate user. The basic problem behind this attack is that the cookie values are handed over to JavaScript via the *document.cookie* directive. On the one hand, this is a necessary feature that is used to process user-specific application data which is stored within a cookie with JavaScript. On the other hand this opens an attack vector for adversaries to steal session information in the outlined fashion. As there is no need to pass session data to JavaScript a new opt-in feature was introduced to prevent the hand over of cookie data to JavaScript. In order to do so a webmaster has to set the so called HTTPOnly flag on a cookie. By setting this flag, the cookie is passed to the browser, but the browser protects the cookie value from being read by *document.cookie*. However, a recent study [14] showed that only 22,3 % of all session cookies are protected by the HTTPOnly flag. This number suggests that there is still a significant attack vector for stealing session information. Therefore, the WebSand framework turns this opt-in flag into an opt-out feature. Technically speaking, this means that the WebSand framework automatically deploys the HTTPOnly flag for any cookie leaving the web server in order to protect any session cookie that might appear. As it is sometimes desirable to pass cookie data to a piece of JavaScript the WebSand framework offers a way within its policy to opt-out the flag for single cookies. So by specifying the cookie name within the policy a webmaster has to intentionally remove the cookie protection.

3.2 Abusing Locality in Shared Web Hosting

As shown in [13] web applications tend to leave the default session storage location unchanged. Hence, server-side session information is often stored in the same location (e.g. */tmp* on UNIX systems). For a web application running on a server in isolation this is not a problem, but for shared hosting environments this fact opens up an additional attack vector for malicious adversaries. If an attacker controlled web application and a victim's web application share the same session storage the attacker can arbitrarily tamper the victim's sessions stored in this location. In order to do so, an attacker forces the victim's application to issue a new session ID for example by registering and logging-in to a low privileged account. After obtaining the corresponding session ID the attacker advises his application to use this session ID to store arbitrary data to it (e.g. Administrator = true or UserId = 1). In this fashion the attacker can change his identity or escalate his privileges in the victim's application depending on the underlying application logic. As this is obviously very undesired a security framework for web applications needs to enforce a strong separation between different applications running on the same machine. Therefore, the Websand framework will strictly isolate the underlying session storage in order to avoid the outlined attack.

3.3 Session Fixation Protection

Another common attack to hijack a session is session fixation [19]. Thereby, a victim is tricked into authenticating a session ID (SID) that is known by the attacker. Thereby, the attack basically works as follows (see Figure 4):

1. The attacker obtains a SID value from the server (1,2)
2. He tricks the victim to issue an HTTP request using this SID during the authentication process (3,4)

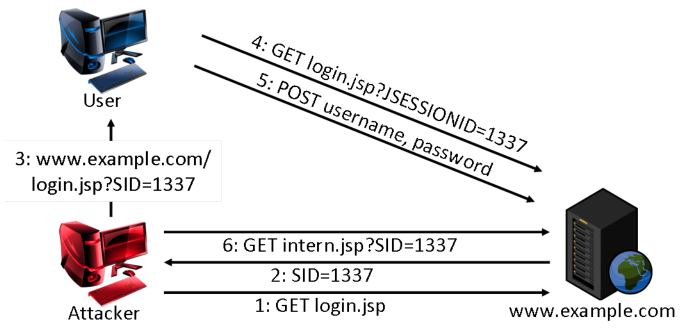

User
3: www.example.com/ login.jsp?SID=1337
Attacker
4: GET login.jsp?JSESSIONID=1337
5: POST username, password
6: GET intern.jsp?SID=1337
2: SID=1337
1: GET login.jsp
www.example.com

Figure 4: Exemplified Session Fixation attack

3. The server receives a request that already contains a SID. Consequently, it uses this SID value for all further interaction with the user and along with the user's authorization state (5).

4. Now, the attacker can use the SID to access otherwise restricted resources utilizing the victim's authorization context (6).

In order to counter the attack, a web application needs to issue a new session ID whenever the authentication state of a user changes. This still enables an attacker to trick the user into using a previously known SID, but if the user authenticates, the ID is renewed and consequently the attacker cannot escalate the privileges of the ID known to him. Although the WebSand framework focuses on security by construction and thus expects a web application to address such a vulnerability during development, we present an approach that even secures vulnerable web applications from being exploited. In order to do so, the WebSand framework intercepts incoming requests and establishes a second level session identifier management. In addition to the SIDs that are set by the application, the interceptor issues a second identifier (the 'proxy SID' - PSID). Whenever an HTTP request without a PSID value is received by the interceptor, this request is regarded to be the user's very first request to the application. If the request carries any stale SID values, such data is discarded. For the corresponding HTTP response a fresh PSID value is generated and attached to the response via set-cookie (see Figure 5). In the course of the following HTTP communication, the application's responses are monitored for outgoing SID values that are to be assigned from the application to the user. If such a value is detected, the combination of the PSID and SID value is stored by the interceptor. From now on, only requests that contain a valid combination of these two values are forwarded to the application (see Figure 6). Requests that are received with an invalid combination of SID/PSID are treated as if they would carry no session information. Consequently, they are stripped off all cookie headers before sending them to the application and are outfitted with a fresh PSID value upon response.

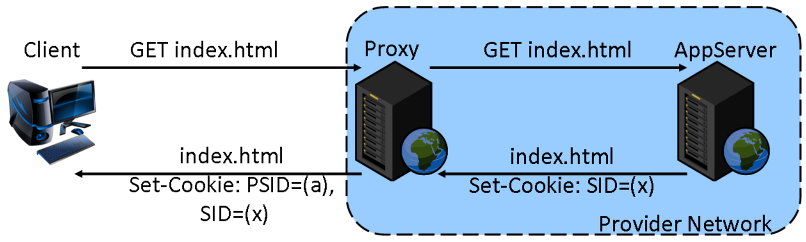


Figure 5: Introduction of the proxy session identifier.

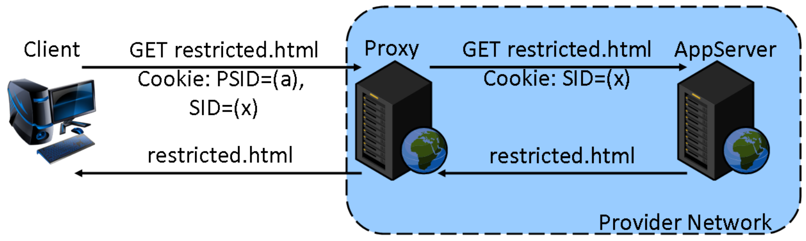


Figure 6: Verification of the proxy session identifier.

To provide protection against Session Fixation, the interceptor monitors the HTTP requests' data for incoming password parameters. If a request contains such a parameter, the interceptor assumes that an authentication process has happened and renews the PSID value, adds an according Set-Cookie header to the corresponding HTTP response, and invalidates the former PSID/SID combination. This way, only the PSID/SID combination is renewed whereas the server-side session record remains unchanged. The SID is even renewed if the authentication attempt fails. This, however, is no threat as the new SID does not carry any security assumptions. The WebSand policy language needs to provide a configuration option for the session cookie names and the requests that trigger a change of the authentication state.

4 Deducing Requirements for Secure Web Interaction

This section provides details about the requirements on the WebSand framework and the WebSand policy mechanism respectively. The requirements on the WebSand framework are listed as ‘operational requirements’, those on the WebSand policy mechanism as ‘policy requirements’. We derived these requirements from the scenarios given in Section 2 and the lessons learned from our recent advances on session security (Section 3). Requirements are given with respect to authentication and authorization (Section 4.1), cross-domain interaction (Section 4.2), and control-flow integrity (Section 4.3).

4.1 Authentication and Authorization

This section describes the operational and policy requirements with respect to authentication and authorization. These requirements are motivated by the challenges posed by the scenarios *Mobile Apps for Enhanced Security*, described in Section 2.1, and *Delegation of Privilege in Distributed Workflows*, described in Section 2.2.

4.1.1 Operational Requirements

While the username/password authentication mechnism is the de-facto standard for authentication, some applications require a higher level of security for authentication and authorization. Three different types of requirements have been identified that are especially important in the mashup context.

1. federated identity management
2. stronger authentication
3. stronger authorization
4. delegation of privilege

Federated Identity Management Support for external identity management allows to delegate the management of users and their authentication credentials to an external service provider, which reduces implementation effort and can provide single-sign-on functionality with applications that use the same service provider. In the WebSand framework, it is planned to support OpenID[18], which is the established, open standard for decentral authentication and identity management.

Stronger Authentication Examples for applications that require stronger authentication can be all applications that operate on sensitive data or functionality, e.g. management of personal information like electronic health records or management interfaces for industrial plants. Some implementations of stronger authentication schemes use mobile phones as a secondary, trusted channel, e.g. Google's mobile authenticator [4] for two-factor authentication. Here the possession of the mobile device and the secret key that is stored in the device serves as a second factor to increase the confidence in the identity of the client. In some cases, stronger authentication is only required for sensitive functionality that has a high impact when abused. When accessing sensitive functionality like this, the user is required to reauthenticate using the stronger mechanism, and is only allowed to continue the workflow after successful completion.

Stronger Authorization Stronger authorization is typically required when sensitive actions are to be performed for example adding a new administrative user or changing of the contact email address. Accessing sensitive functionality would then trigger a workflow that explicitly asks for the user's consent to perform the sensitive action on the server side, which may require re-authentication or the use of a trusted secondary communication channel. Examples of this are online banking with mobile TANs and business processes that require the application of the separation of duties principle asking for the consent of at least two different users. In a business context, it is important for auditability to be able to prove that a user gave his explicit consent to sensitive actions that have been performed. The requirement of stronger authorization is closely linked to the requirement of stronger authentication, because authorization always relies on the authenticity of the requesting entity.

Delegation of Privilege In some mashup scenarios, the user needs to delegate the privilege to perform specific actions on his behalf against a third party. This allows to create mashups with workflows running in the background as server-to-server communication, but without giving up the user's authentication credentials. An example is the scenario of Twitter integration with Facebook described in Section 2.2.

4.1.2 Policy Requirements

For representation in a policy, the requirements for stronger authentication, stronger authorization and delegation of privilege are subsequently described more formally.

Sensitive operations can be described by the following properties, which must be expressable by the security policies:

- the *identity* of the requesting principal,
- the *identifier* of the operation or action to be performed, and
- the *identifier* of additional properties.

The requesting principal represents the identity of the user of the active session, which is necessary for access control checks. The policy must be able to express the requirement of authentication for non-public resources and must facilitate the use of external identity providers via OpenID. For more sensitive actions, it must be possible to request stronger authentication mechanisms, for example two-factor authentication.

For access control, the operation or action to be performed is represented by a unique identifier that represents the logical operation to be performed by the web application. In combination with the principal basic access control policies can be defined. For Mash-up scenarios that contain back-end workflows without direct user involvement, it must be possible for the mashup to delegate a subset of the user's privileges to third-party systems. One possible implementation mechanism is given by the OAuth protocol used by Twitter.

Additional properties should be available to model specific requirements, for example membership in a specific organisational unit or association of users to tenants in multi-tenant applications. Additional attributes should be available to request stronger authorization or explicit user consent checks using a secondary communication channel.

4.2 Cross-Domain Interaction

This section covers the operational and policy requirements that are related to cross-domain interactions. Thereby, the requirements are derived based on the scenario outlined in Section 2.3 and two papers created within the Websand project [9, 8].

4.2.1 Operational Requirements

Client-side cross-domain requests are requests that are created within the user's browser across domain boundaries. Hence, these requests are outfitted with the user's session cookies and thus executed within the corresponding authentication context. In general, three mechanisms exist that can be leveraged to conduct such requests: Flash, Silverlight and Cross-Origin Resource

Sharing (CORS). Basically, all these approaches follow the same server-side opt-in security model: In order to grant cross-domain access to a resource, a server has to setup a policy that defines a set of access grants to one or more foreign domains. The main differences between the three approaches can, thereby, be divided into three different categories:

1. *Availability for different environments:* Nowadays the browser landscape is situated in a transitional phase. While a lot of older browsers make use of plug-ins in order to enrich their functionality, modern browsers and especially those deployed in mobile environments do not support plug-ins anymore, but rather focus on new HTML5 capabilities for novel use cases. This fact has a severe impact on cross-domain interactions as plug-in technologies like Flash and Silverlight are only available in legacy browsers and CORS is only available in modern browsers capable of HTML5.

2. *Policy transport mechanism:* While Silverlight and Flash provide their policies within a file, CORS utilizes HTTP request and response headers to transfer the policy to the client. While policy files are easier to deploy, HTTP headers allow a much more fine-grained categorization mechanism.

3. *Policy expressiveness:* While CORS provides a very expressive and mighty policy language that can be applied on request level, Silverlight and Flash provide a mechanism that can only be applied on folder level.

In order to provide cross-domain access to a heterogeneous environment, servers have to offer more than one policy in order to cover the whole browser spectrum. At the same time, multiple security pitfalls exist in each technology that web developers have to be aware of. As shown in [9], web developers tend to issue insecure policies by utilizing a wildcard that grants cross-domain access to any other domain in the web. A wildcard alone however is not sufficient enough to cause insecurities, but in many cases these access grants are also given to resources that contain personalized information which is only available to the user. There are ways to offer cross-domain services in a secure fashion, but these ways are on the one hand hardly known to web developers and on the other hand sometimes difficult to deploy. Thus, one requirement on the WebSand framework is to take over the task of generating and deploying such cross-domain policies in a secure fashion. Thereby, the web administrator specifies a list of domains for each resource that should be shared via cross-domain requests. The WebSand framework then generates and deploys the different policy formats and at the same time avoids security pitfalls, like sharing private data to any other domain. One challenge here, will be the harmonization of the different approaches that allows a fine-grained, but still easy-to-use configuration.

4.2.2 Policy Requirements

Cross-domain interactions depend on four different factors:

1. *Target domain*: The domain that deploys the cross-domain policy and serves the requested resource
2. *Requested resource*: The resource that is passed back to the requestor's domain
3. *Requestor's domain*: The domain that initiates the cross-domain request and receives the response (that could contain sensitive data)
4. *Header fields*: HTTP request header fields sent along with the HTTP request for the requested resource.

In order to generate the different cross-domain policies, the WebSand policy language needs to offer a mechanism that can be used to specify a set of access grants. An access grant thereby exists of a list of resources (requested resource) and a list of domains (requestor's domain) to which cross-domain requests are granted. Additionally, a cross-domain request can carry a set of custom header fields. As these fields are often used for security sensitive checks these fields must explicitly be allowed by the cross-domain policy. In order so, the web developer should be able to define a set of HTTP header fields that can be sent to the requested resource.

4.3 Control Flow Integrity

This section discusses operational and policy requirements in respect to control-flow integrity. The requirements are derived from the scenarios described in Section 2.4 and Section 2.5.

4.3.1 Operational Requirements

In the beginning, the web was meant to be a mechanism for delivering and connecting static documents via a computer network. In the recent years, however, the basic technologies of the web (HTTP, HTML) were more and more used to build sophisticated applications such as online shops or social networks. As opposed to classical desktop applications, these web applications face one major shortcoming when it comes to control-flow integrity: While a desktop application has full control over it's execution sequence a web application cannot enforce the sequence in which it's functionalities are called. Though control-flow integrity can be enforced for one specific resource on the web server, it cannot be enforced easily for functionality that is spread across several server resources. The main reason for this shortcoming is the fact, that an adversary could chain a series of requests in an arbitrary sequence. With the upcoming workflow-based web applications this problem becomes a serious vulnerability. Web developers can only make assumptions upon the sequence in which requests are arriving at the web server. Thus, WebSand aims at addressing this issue by implementing an approach that enforces control-flow integrity across multiple requested resources.

Control-Flow Graph: In order to enforce control-flow integrity, a machine-readable representation of the underlying business logic is needed. A workflow can be represented by a finite state automaton consisting of several states and transitions (see Figure 7 for an example). While the automaton's states represent generic business logic related server-side states within a workflow, the transitions represent the requests that a client sends to the server. In each state, only specific transitions and thus only specific requests are allowed.

For the sake of good usability it should be possible for a user to enter multiple different control-flows at the same time as well as entering the same control-flow twice. Furthermore, it is also possible that one resource is part of multiple control-flows. In order to assign incoming requests to existing control-flow graphs, the WebSand framework utilizes a token-based approach. Whenever such a request arrives, it has to carry a unique token that identifies the corresponding workflow. If a request does not include this token it is considered as a request that is either not part of a workflow or that arrived out of order.

Enforcing Control-Flow Integrity Upon a Graph: The afore-mentioned graph representation allows the WebSand framework to understand an application's requirements regarding control-flow integrity. After the application delivers a specification of it's control-flow graphs to the WebSand framework, the framework tracks the application's state by investigating incoming HTTP requests and responses. If a user session is situated within a workflow, incoming requests that are related to this workflow are mapped to state transitions in the graph. The corresponding mapping of requests to transitions is thereby defined within WebSand's policy. If a valid mapping can be established the transition is carried out and the request is forwarded to the requested resource. If the request would cause a transition that is not defined in the control-flow graph, the WebSand framework rejects the request and thus enforces control-flow integrity.

Example:

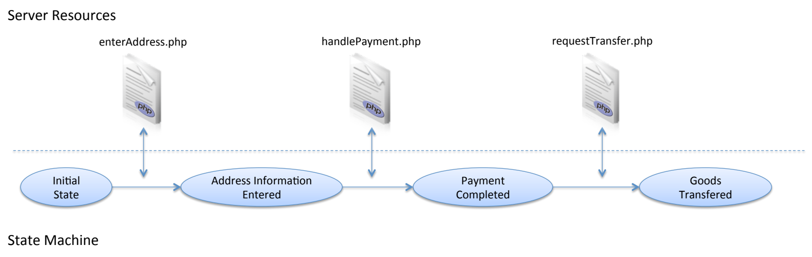


Figure 7: (Simple) Mapping of server resources to a control-flow graph

Figure 7 shows an exemplary control-flow graph for a shopping cart check-out process of an online shop consisting of four different states. After clicking the checkout button the user is prompted to insert his address data. In a second step, the payment of the ordered goods is conducted and in a third step the transfer of the goods is automatically triggered. In case this application is vulnerable to attacks that aim on subverting the control-flow, an attacker could skip over the payment step and directly trigger the physical transfer of arbitrary goods. In order to enforce control-flow integrity WebSand monitors the incoming requests. After entering the "Address Information Entered" step, only requests to handlePayment.php are allowed. Requests to requestTransfer.php would be rejected by the framework as such a transition is not foreseen in the control-flow graph.

Critical Section: In general it should be possible for a user to enter a control-flow more than once at the same time to provide good usability. For example, a user of an online banking application should be able to conduct several money transfers in parallel in order to easily copy and paste the recipient data (this is not possible in many online banking applications today). However, there are parts of such control-flows that should not be entered simultaneously to avoid state confusions or race conditions. Thus, the WebSand framework has to be able to enforce critical sections in a workflow. In this context, a critical section is a clearly specified subgraph which can only be entered once per user at the same time.

Ensuring User-Intended Actions: In the past, several types of attacks against Web applications have been documented, that cause the Web application to mistake adversary initiated HTTP requests to be intended actions initiated by the attacked victim. Examples for such attacks include phishing (submitting the victims's credentials to the application), click-jacking (tricking the victim to interact with the application's front end without his knowledge), cross-site scripting (hijacking the victim's browser to interact with the Web application), and cross-site request forgery (causing the victim's Web browser to send HTTP requests that carry the victim's authentication context to the application).

Unlike other problems in the field of secure Web interaction, this class cannot be solved on the server-side alone. Instead, it has to take the application's front-end into consideration, as a user's actions and intents are deeply interwoven with the application/device the user interacts with (which is in most cases the Web browser but can also include various other HTTP enabled applications and devices). Due to the vast heterogeneity of potential Web front ends of applications/devices, a universally applicable solution to this type of threat has not yet been developed, and it is highly doubtful that such a general solution is even possible.

To protect security sensitive actions within a workflow from attacker initiated HTTP requests, the WebSand framework requires the capability to demand and verify proof that the incoming HTTP request was indeed triggered by the client-side because of intended actions of the user. As motivated above, the technical solution how such a verifying step can be implemented, is highly dependent on the utilized front-end technology that the user is currently interacting with. The situation on a mobile browser on a smart phone might differ completely from a full-fledged desktop browser or an internet appliance. It is the framework's duty to translate the abstract demand for a given workflow step to be protected into the applicable technical solution which applies to the encountered client-side execution context (or, if the currently utilized client-side environment does not offer the required security characteristics, for instance if the front-end is an outdated Web browser in an internet cafe, to terminate the workflow due to security reasons).

4.3.2 Policy Requirements

The main goal of the policy mechanism is to define the application specific security requirements in a way that the WebSand framework is able to enforce the necessary security measurements. For control-flow integrity these requirements are mainly encoded within the control-flow graph. Hence, WebSand's policy language must be expressive enough to define such a graph in a sophisticated fashion (details can be found in Section 6). Such a graph can be described as a set of states and a set of transitions between the different states. As mentioned in Section 4.3.1, it must be possible to enforce critical sections for critical subgraphs. This means that the policy mechanism must provide features in order to mark subgraphs of a workflow as a critical section. Finally, the policy mechanism must be capable to flag a workflow step to require special protection, to ensure user-intended actions.

5 Framework Design

This section describes the software architecture of the WebSand framework for secure web interaction and the addressing mechanism for resources of the web application's external interface, which allows to formulate security policies that define properties, attributes, and restrictions in respect to interaction with these resources.

5.1 Framework Architecture

Figure 8 shows a functional view of the architecture of the WebSand security framework.

The WebSand security framework for secure web interaction provides functionality for two main purposes, each with its own application programming interface: the gatekeeper API and the application API. The gatekeeper API serves as the entry point for the gatekeeper, which are components of web application code that intercept HTTP request and response messages on their way in and out of the application. The gatekeeper serve as an adapter between a web application's specific usage of HTTP and the generic WebSand security framework. It is the responsibility of the gatekeeper to extract the security relevant information out of the HTTP messages and to forward this information in form of request objects and associated security attributes to the security framework. While the gatekeeper API handles the processing of HTTP messages, the application API provides direct access to the functionality of the security modules. The application API is intended to be called directly by application code, for example to perform authorization checks in the business logic.

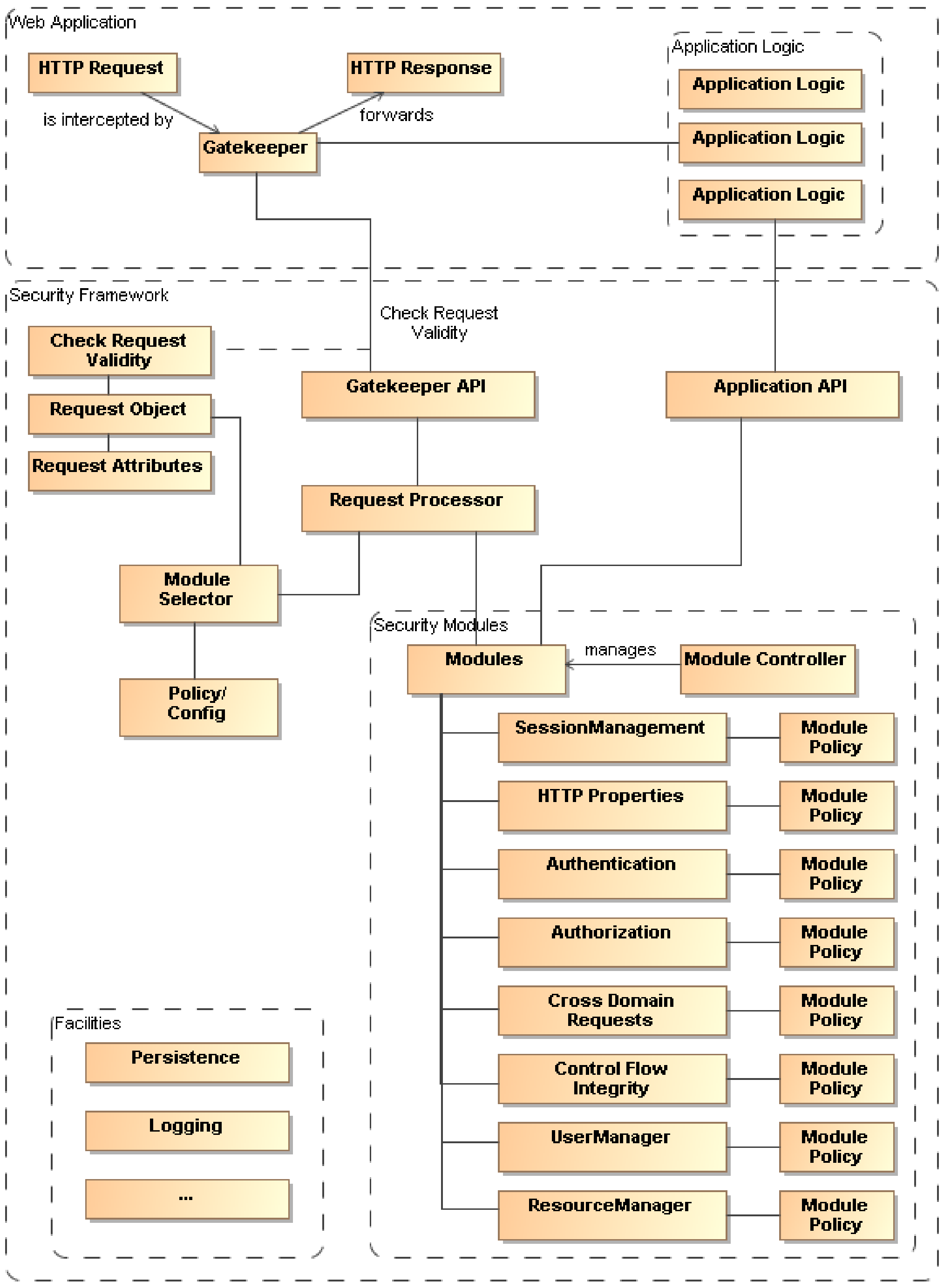


Figure 8: Software architecture of the WebSand security framework.

5.2 Framework Components

5.2.1 Gatekeeper

The gatekeeper component is the part of the application code that integrates the security framework into the request processing of the application. Intercepted HTTP messages must be processed by the gatekeeper and security relevant information must be extracted. This information is then forwarded to the gatekeeper API in the form of request objects and associated request attributes. The gatekeeper also needs to handle the decision of the security framework to accept or deny the request. The implementation of the gatekeeper depends on the web framework that is used to implement the application. In a Java web container, a possible implementation would be a ServletFilter or a Valve.

5.2.2 Application Program Interfaces

The security framework exports two main programming interfaces to the web application, the gatekeeper API and the application API. The gatekeeper API is called by the gatekeeper to verify the permissibility of an HTTP message. The application API provides direct access to functionality of the security modules to application logic. It can be used, for example, to implement access control checks on the business layer.

5.2.3 Request Objects and Attributes

In the framework, one has to differentiate between HTTP requests and internal requests. HTTP requests originate from the client and are intercepted by the gatekeeper. The gatekeeper in turn creates an internal request object, which is then passed on to the request processor via the gatekeeper API. This request contains those attributes which are needed by the security modules to perform the actual security checks. These attributes are directly extracted from HTTP requests, such as the client IP address, GET/POST parameters and the requested URI.

5.2.4 Request Processing

The Module Selector evaluates the policy and determines the set of security checks that are necessary to decide whether a request can be accepted or must be dropped. Requests to public resources, for example the login page, may not need an access control check. In some cases, the security modules will have dependencies on request attributes that are generated by other modules. For example, the authorization module relies on the user identity, but most requests will not a-priori contain that attribute because the session management is responsible to deduce it from the session.

5.2.5 Modules

The Security Modules are modular software components that encapsulate the functionality of one security aspect. Each module has an interface used for request processing and one interface that is exported via the application API. For example, the module for authorization may contain an implementation for role-based access control.

A special module is the ResourceManager module, which manages a list of application resources on which access control checks can be defined. Resources represent entities that can be addressed by requests and also include actions that can be performed, such as adding a new user, submitting a report, or processing a financial transaction. The addressing mechanism is a central aspect of the framework and is described in more detail in the following section.

5.3 Addressing Mechanism

In this section, an addressing mechanism for application resources is described that serves to identify application resources in the security policies and determine when security checks are performed. There are three default resource types that can be addressed by the mechanism: web resources, functionality and data objects, but the addressing mechanism can be extended by application specific attributes. In the following, the proposed addressing mechanism is described.

5.3.1 Syntax

The addressing mechanism that is proposed is similar to the concept of URNs, as defined in RFC2141 [10], and URIs, as defined in RFC3986 [3]. The main idea is to identify resources by a string of characters, which consists of a namespace identifier and an resource identifier part. Default namespaces are defined for standard resource types: `url`, `action`, and `data`. Custom namespaces may be defined by client applications.

The two parts are separated by a colon, and the namespace part is required to only contain upper-case and lower-case characters in US-ASCII encoding. The resource identifier part is only restricted to be a valid unicode sequence. The maximum total lenght of resource addresses should be configurable by the client application, a default value of 2000 bytes is proposed. The following strings are examples for valid resource addresses:

- `url:http://websand/admin/doit?action=adduser`
- `action:admin_adduser`
- `data:table-admin-users`

5.3.2 Application Resources Types

There are three different types of application resources that typically need to be addressed by web applications: web resources, functionality and data objects. Each of these types maps to one of the default namespaces identifiers of the addressing mechanism: `url`, `action`, and `data`. Custom resource types can be defined to satisfy special requirements, for example multi-tenancy. To be recognizable by the security framework, all resources need to be registered by the web application on start-up.

Web Resources Web resources are handled by the `url` namespace, which contains the URL in the resource identifier part. This allows direct handling of HTTP resources and covers the external interface of web pages and web services.

Which parts of the URL will be included in the resource identifier depends on the application code that integrates the security framework. For dynamic pages or services, the URL will often need to be trimmed to not contain the URL parameters, e.g.
`url:http://websand/admin/adduser?username=johndoe`. In case of URLs that act as command processors, the URL might contain the part of the URL that specifies the command, e.g.
`url:http://websand/admin/do?action=adduser`. This mechanism offers flexibility to client applications in the way the framework is integrated.

Functionality and Data Resources Functionality and data elements are both addressed by a default namespace identifier and an unstructured resource identifier. Resources that are supposed to be handled by the security framework must be registered before use. In principle, this kind of identifiers only serve as keys to elements in a set of known resources. After the registration of a resource, it is possible to define policy rules that address this resource.

Custom Namespaces The freedom to define new namespaces and arbitrary identifiers makes the mechanism very flexible. It would be, for example, possible for an application to define a namespace `orgunit` to model the organisational structure of a company. This would allow to define access control restrictions based on the organisation, for example by performing an access control check on the resource `orgunit:sales`.

5.4 Usage in the Framework Architecture

The addressing mechanism is used mainly in three different places of the architecture of the WebSand security framework: the gatekeeper API, the application API, and the policy representation. In these places the addressing mechanism is needed to identify resources of the web application at the programming interface or the configuration mechanism. Often, the gatekeeper components that process HTTP messages may plainly use the URL from the HTTP requests to specify the target resource in the request attribute, for example: `url:http://websand/admin/do?action=adduser`. But the gatekeeper might also parse the URL and attach several request attributes of different types to the request object, for example:
`url:http://websand/admin/do` and `action:adduser`, which would allow policy rules for both resources.

5.5 Summary

This section describes an addressing mechanism for application resources that is used to identify application resources on the external interface of the security framework and in security policies. The mechanism is able to handle different resource types and is flexible enough to be extended with custom resource types.

6 Secure Web Interaction Policies

This section describes the policy attributes that are necessary to implement a policy that is capable of fulfiling the requirements deducted in Section 4.

6.1 Policy Format

As described in Section 5.1, the WebSand framework follows a modular design principle. Each module covers one functionality such as checking for valid user credentials or deploying cross-domain capabilities. Depending on the requirements for a specific request, modules are activated to perform their checks or simply ignored if a particular check should not be performed. The security requirements that are needed to decide upon this fact need to be specified within a policy. Basically, the policy consists of a set of rules that trigger the activation of a certain module upon a request that is targeted towards a certain resource and that fulfill certain conditions. Therefore, such a rule is structured into four different elements:

1. *Module name:* A name that identifies the corresponding module and sets the semantics for the elements *Conditional Criteria* and *Additional Parameters*

2. *Resources:* A list of resources for which the corresponding module is triggered. Thereby, the resources are identified by the addressing mechanism described in Section 5.3.

3. *Conditional Criteria:* An expression that returns a boolean value. Depending on the evaluation of the expression, the module triggers a respective security event for the request. Such events could be *allow* and *deny* as well as appending HTTP headers to the respective response or logging the event. The conditions can thereby address the context of the request such as the origin and authentication information in terms of GET and POST data or HTTP header fields. It is also possible to define custom evaluation criteria. These criteria are individually designed for each module.

4. *Additional Parameters:* In this element, additional parameters can be set that are passed to the module in order to influence the exact behavior. As opposed to the Conditional Criteria, Additional Parameters do not directly influence the outcome of the module's security decision but are needed by the module to take a decision in the first place. The

format and purpose of these parameters is highly dependent on the corresponding module that is triggered by the policy rule. See Section 6.2 for details.

The following section defines the modules and the corresponding conditional criteria and additional parameters for authentication, authorization, cross-domain interaction and control-flow integrity.

6.2 Required Policy Attributes

In this section, we list the WebSand policy attributes which are required for secure web interaction. For this purpose, we utilize the policy format which was described in Section 6.1. The listed policy attributes directly correspond to the identified policy requirements from Section 4.

6.2.1 Authentication and Authorization

In the following, the requirements described in Section 4.1 for authentication and authorization are transformed into a description of policy requirements.

The authentication module is responsible for authentication and managing different authentication mechanisms, for example username/password and OpenID.

- **Module Name: AuthenticationModule**

  This modules enforces authentication for protected resources and, if successfull, provides the user's principal as an attribute to other modules, for example the authorization module.

- **Resources:** the set of resources, for which authentication is required.

- **Conditional Criteria:** none

- **Additional Parameters:** the authentication mode. If not specified, the default authentication mode for the application is used, for example username/password or OpenID.

The authorization module enforces access control checks.

- **Module Name: AuthorizationModule**

  This modules performs access control for non-public resources. It depends on the authentication module to provide the user's principal.

- **Resources:** the set of resources, for which access should be granted.
- **Conditional Criteria:** none
- **Additional Parameters:** the access control model and additional model specific parameters, for example 'role-based access control' and the required role. Authorization of delegated privileges using OAuth are represented by a special module. Optional parameters might be specified to model Section 5.

For stronger authorization, the explicit consent module can be used to request confirmation that the requested action should be performed.

- **Module Name: ExplicitConsentModule**

  This modules performs a stronger authorization check by triggering a workflow that asks a specified user for explicit consent to perform the required action.
- **Resources:** the set of resources, for which access should be granted.
- **Conditional Criteria:** none
- **Additional Parameters:** the user's principal which should be asked for his consent and a query string.

The OAuth Service Access module allows to define access control policies based on privileges that have been previously granted to third-party services, which are thereby enabled to invoke service calls on the user's behalf.

- **Module Name: OAuthServiceAccessModule**

  This modules performs access control based on OAuth and also authentication of the requesting service.
- **Resources:** the set of resources, for which access should be granted based on OAuth.
- **Conditional Criteria:** none
- **Additional Parameters:** OAuth parameters, for example the OAuth consumer for which access should be allowed.

6.2.2 Cross-Domain Interaction

As mentioned in Section 4.2 one requirement of the WebSand framework is to deploy cross-domain policies in a secure fashion. Following the policy format defined in Section 6.1 we deduct the following rule definition.

- **Module Name: CrossDomainModule**

  This module configures the deployment of necessary cross-domain policies. On the one hand, this rule is used at startup time of the framework to generate and deploy the necessary policies for Silverlight and Flash. Furthermore, the rule is used on a request-basis to deploy the CORS header to the response on an incoming cross-domain request.[3]

- **Resources:** Set of resources for which client-side cross-domain access is enabled.

- **Conditional Criteria:** The main conditional criteria for a policy entry of this type is a list of whitelisted external domains, which are permitted to access the specified resource in a client-side cross-domain fashion.

  In order to trigger this module for incoming requests, such requests have to carry the necessary CORS request headers and the domain specified within the header has to be whitelisted in the instantiated policy.

- **Additional Parameters:** In order to tell the WebSand framework what kind of custom request header fields are allowed for this cross-domain request, a webmaster can specify a set of header field names within the additional instructions element. On the technical level this information will be used for compiling the static Flash and Silverlight policy files, as well as for potentially happening CORS pre-flight communication [20].

6.2.3 Control Flow Integrity

Section 4.3 explains the requirements concerning the enforcement of control-flow integrity within the WebSand framework. We transfer these requirements to the policy format provided in Section 6.1.

- **Module Name: ControlFlowIntegrityModule**

[3]An example application of this policy can be found in Section 6.3.3

This module enforces the control-flow graphs that are deployed by the web application. For every incoming request, the module determines whether the respective step in the control-flow is granted or not. Requests that belong to a workflow are fitted with tokens to keep apart different workflows in the same session. After processing a request, the module conducts a transition and, thus, tracks the application's state.

- **Resources:** A set of resources which are part of a control-flow graph.
- **Conditional Criteria:** As a user session is not situated within a workflow at startup time, the first request that is made towards a control-flow integrity protected resource must be targeted at an entry point of a workflow. Any subsequent request can be processed if the current state allows the requested transition.
- **Additional Parameters:** Additional information that must be provided to the *ControlFlowIntegrityModule* includes whether the transition belongs to a critical section. This information is crucial to prevent race conditions.

Next, a module is needed to ensure that security-critical actions can only be triggered if evidence is given for the user's intent to send the respective HTTP request. For resources that are registered by the module through corresponding policy entries, the module verifies the user's intent and that the request is not triggered by an attacker before running the respective action.

- **Module Name: IntendedActionsModule**

  This module controls the access to dedicated critical actions which have to be secured more thoroughly. Such actions generally have a particular high impact. Thus, the web application has to ensure that the user indeed intended the respective action and that it has not been triggered by an attacker.
- **Resources:** A set of resources for which user intent has to be ensured.
- **Conditional Criteria:** In order to be processed, a request needs to carry attributes that can be used to prove the user's intent (details were provided in Section 4.3).
- **Additional Parameters:** none

6.3 Application to Scenarios

We will show how the policy attributes defined in the preceding sections help to compile secure web interaction policies. Therefore, the scenarios from Section 2 serve as problem statements.

6.3.1 Mobile Apps for Enhanced Security

In the scenario *Mobile Apps for Enhanced Security*, described in Section 2.1, modes of authentication and authorization are proposed which are based on apps running on mobile phones. To integrate the planned feature for stronger authentication and authorization, the web application that integrates the WebSand framework has to configure the 'ExplicitConsentModule' for the ressources to be protected. For two-factor authentication, this would be the login-URL. The associated policy would need to have an additional entry that requires the 'ExplicitConsentModule' to trigger the workflow that asks for the user's consent. Since the workflow is asynchronous, the application must actively support this out-of-band authorization of sensitive actions.

In the following, the workflow for two-factor authentication is described as an example.

1. The user accesses the main page and enters his credentials, e.g. username / password.

2. The application processes the request, reads the additional 'Explicit-ConsentModule' requirement from the policy, and starts the out-of-band authentication workflow.

3. The WebSand framework sends the request for explicit user consent to the preregistered mobile phone of the user.

4. In the server-side, internal response object, the need for the out-of-band process is signalled to the web application, which displays a message and prompts the user to confirm the request for explicit consent on his mobile phone.

5. The user checks his mobile phone and acknowledges the request for authentication to the web application.

6. The app on the mobile phone signals to the WebSand framework that the explicit user consent has been acknowledged. The acknowledgement is forwarded to the web application.

7. The web application queries the WebSand framework for the status of the explicit user consent, either by polling or triggered by user interaction.

8. When the explicit user consent has been accepted by the WebSand framework, the web application triggers a new authentication request in the user's browser, which can now be processed.

6.3.2 Delegation of Privilege in Distributed Workflows

The scenario *Delegation of Privilege in Distributed Workflows*, desribed in Section 2.2, describes the need for a mechanism for delegation of user privileges to a third party. These delegated privileges can then be used in back-end server-to-server communication to perform operations on the user's behalf. Since this type of communication is targeted at resources that expose a programming API via web services, the OAuthServiceAccessModule will only be used on a small number of ressource elements in the policy.

As the semantics of the programming API to which access privileges are delegated are determined by the web application, it must also implement the user interface for delegation of privilege. In the user interface for delegation of privilege, it is important to communicate clearly what set of privileges is delegated, to which third-party and for what purpose. The application of the WebSand framework facilitates that by exporting a programming interface in the application API, which allows to delegate privilege, and by providing the OAuthServiceAccessModule, which allows to express the need for delegated privilege in the policy for protection of service endpoints.

6.3.3 Cross-domain Interaction

In Section 2.3 an example scenario is given that needs to be accomplished in a secure fashion with the help of the WebSand framework. Basically, the scenario covers two domains: The domain used by the on-demand sales application and the domain used by the on-premise warehousing system. For this section we consider the domain `example-company.sales-on-demand.com` to be the domain of the on-demand system and `warehousing.corp` as the internal intranet DNS name of the warehousing system. As there is a firewall in between those two domains, the respective systems cannot use a direct communication channel, but have to rely on client-side cross-domain requests. For this purpose, multiple different techniques namely Silverlight, Flash and CORS are utilized in order to cover the broad browser spectrum used within the company. In order to ensure the smooth communication between the systems, the warehousing system has to open up it's API by setting up a distinct server-side policy for each of the mentioned techniques. The cross-domain policy is thereby enforced on the client-side either by the respective plug-in (Silverlight, Flash) or by the browser itself (CORS). Only if the requesting domain is whitelisted within the corresponding policy the request is allowed, otherwise it is blocked within the user's browser. In order to handle the deployment of these policies in a secure fashion, the WebSand framework is used to take over the responsibility of generating and deploying these policies. The API of the warehousing system basically consists of set of server-side scripts that are located in the folder "API" in the root directory of the server i.e. the complete URL for the API folder is `warehousing.corp/API/`. Thereby, the API consists of two files called *RetrieveProductInformation.php* and *RetrieveStockData.php* In order to access the API in a proper way, the application deployed on `example-company.sales-on-demand.com` has to additionally add some custom HTTP headers to it's cross-domain requests (Namely CUSTOM_HEADER1 and CUSTOM_HEADER2). In order to configure the WebSand framework according to these requirements the company sets up following policy rule within the WebSand's policy file:

- **Module Name: CrossDomainModule**
- **Resources:**
  url:http://warehousing.corp/API/RetrieveStockData.php
  url:http://warehousing.corp/API/RetrieveProductInformation.php
  url:https://warehousing.corp/API/RetrieveStockData.php
  url:https://warehousing.corp/API/RetrieveProductInformation.php
- **Conditional Criteria:**
  OriginDomain == "example-company.sales-on-demand.com"
- **Additional Parameters:**
  AllowedHeaderFields = "CUSTOM_HEADER1, CUSTOM_HEADER"

Actually, WebSand processes this rule multiple times. Once at startup time in order to generate the respective Flash and Silverlight policy files and whenever a request arrives that targets one of the listed resources in order to trigger the deployment of the necessary CORS headers onto the HTTP response. Besides, triggering and unifying this deployment process for the different techniques, WebSand conducts sanity checks upon the complete WebSand policy file. If it detects a security issues it prevents the deployment of cross-domain policies and thus avoids sensitive data to be leaked to an untrusted application. One such security issue could be a cross-domain access grant for a resources that has additional security requirements such as the enforcement of a trusted path. As such a path cannot be guaranteed by a cross-domain request this inconsistency will be prevented by WebSand. Hence, WebSand is not only able to unify and ease the deployment process, but it also increases security of the existing cross-domain access grants by conducting cross-checks with other security requirements.

6.3.4 Online Shopping Workflow over Two Domains

In this section, we show the necessary policy rules to secure the merchant web application in the scenario given in Section 2.4.

The merchant first defines valid workflows. We simplify the scenario and reduce the workflow to the most interesting steps. The customers are allowed to access all article descriptions in arbitrary sequence and put arbitrary many items in the cart. After a cart addition, a cart summary with all costs including shipping and taxes is shown. Then, either more items can be inspected or a checkout can be requested. In the latter case, the user is redirected to the CaaS provider. As the following steps are off-domain and, thus, out of scope for the framework, the user's return is the next action concerning control-flow integrity. A summary of the purchase is presented in case the return request fits control-flow restrictions.

In the first place, we present the intuitive control-flow graph in Figure 9. A request to x leads to the overview page of the shopping cart, either by putting a new item to the cart or by just requesting an overview of the current shopping cart status. The resource a that is requested afterwards denotes a click on a “Checkout” button. This transaction needs a verified intended action to prevent attackers to shop on behalf of a victim. When in state 2, the user is immediately redirected to the CaaS in order to provide the payment details and acknowledge the money transfer. Finally, after a number of unknown steps in the CaaS domain, the user is redirected back to the merchant's site where he sends a request to resource c resulting in state 3. There, a summary and confirmation page could be shown that can be printed and serves as a receipt for the user. State 2 is a critical section because a malicious user driving two workflows to this state might mix up the parameters in the subsequent requests to c and probably present a payment acknowledgement token gained for a cheaper item to cause the merchant's web application to accept it for a more expensive item [21]. So, for each session, at most one payment notification must be expected.

Taking into account that only the control-flow in the local domain is subject to control-flow integrity enforcement by the WebSand framework, we can state that the redirection step b in fact causes a request to a foreign

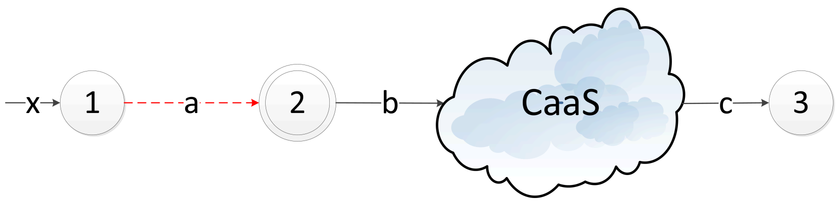


Figure 9: Workflow graph for the CaaS scenario.

domain and is thus out of scope for the *ControlFlowIntegrityModule*. So, the respective control-flow graph ignores the redirection and results in the graph given in Figure 10. Special requirements can be put on the request to c in the conditional criteria including cryptographic tokens in the request parameters and state 2 as the current state from the merchant's web application's point of view.

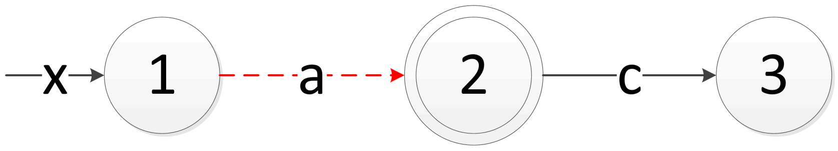


Figure 10: Control flow graph for the CaaS scenario.

To sum up, the access to the domain re-entry point can be secured by the *ControlFlowIntegrityModule* in conjunction with appropriate policy rules that are derived from the control-flow graph. We could show that a multi-domain control-flow does not cause more control-flow protection measures than a local control-flow except the check of cryptographic tokens in the request.

6.3.5 Web Portal to Send Limited Number of Text Messages

The web portal scenario given in Section 2.5 provides potential race condition vulnerabilities. That means a malicious user could request the same resource more often than it is allowed by the business logic. In the example scenario, the SMS sending API has to be protected against such an attack. The control-flow graph is provided in Figure 11.

So, the respective workflow starts with a request on the text input page x, either after login or by access via the provider's network, resulting in state 1. Then, a request to the SMS sending API a with attached information like message text, sender, and recipients leads to state 2 where a redirection to a confirmation page b is issued. Finally, the workflow ends in state 3. The crucial part in this workflow are states 1 and 2 together with requests to

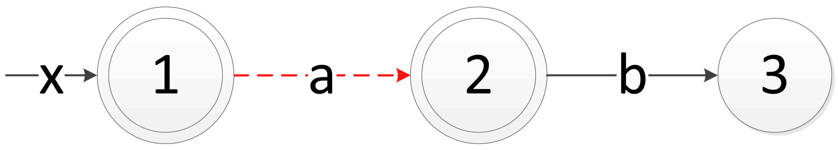


Figure 11: Control flow graph for the text message scenario.

a. The latter need a trusted path because these requests consume free SMS quota and, thus, cause some kind of cost. State 2 and 3 together build a critical section because a user accessing state 2 in parallel more often than his remaining quota could succeed to send more messages than allowed. The secure section only allows sequential sending of messages and prevents the exploitation of race conditions.

In the end, the definition and enforcement of critical sections in the policies effectively prevents exploitation of race conditions without major efforts. Together with the results from Section 6.3.4, we could show that the proposed policy mechanism is suitable to provide the intended security measures in terms of control-flow integrity.

7 Conclusion

This deliverable has reported on the specification of secure web interaction. First, we have defined secure web interaction to be ensured if all incoming requests carry information that allows the Web server to uniquely determine the session and cross-domain context, track authentication and authorization, and enforce control-flow integrity.

Five scenarios have been described that represent the problems which are supposed to be solved within the context of this work package. They deal with distributed two-factor authentication and authorization, authorization delegation with OAuth, cross-domain interaction, control-flow over several domains, and race conditions respectively.

The basis for advanced approaches in the course of secure web interaction is session security. We have summarized our recent progress in that field. To sum up, we identified issues in the usage of cookies as session authentication credentials and provided countermeasures to Cross-Site Scripting (XSS) and Session Fixation attacks. These attacks aim to steal an authenticated session either by gaining knowledge of the session cookie of the victim or by setting an attacker-controlled session cookie at the victim. Finally, an often overseen problem related to server-side session storage has been identified and will be prevented in the WebSand framework implementation. This problem allows the attacker to share or even escalate his authentication status between different applications that are hosted on the same server.

Based on the insights we have gained during research on the topic and the defined scenarios, we compiled security requirements that are needed to obtain secure web interaction. These requirements are divided into operational and policy requirements. While operational requirements define needs on the functionality and features of the WebSand framework, policy requirements specify what the policy language with its attributes has to fulfill. All these requirements have been provided with respect to authentication, authorization, cross-domain interaction, and control-flow integrity.

Then, the first design details of the WebSand framework have been provided. We have described the framework architecture with its components and their communication links. A central entity of the framework is an application-dependent gatekeeper which directs incoming requests to the actual WebSand framework and translates requests from their external representation to their internal format. The security modules work on the internal format. As a central aspect of the deliverable, we have presented the WebSand addressing mechanism that allows fine-grained, unambiguous identification of the resources of the web application's external interfaces.

Finally, the WebSand policy mechanism has been described. The policy format and the policy attributes have been explained. The policy format includes the name of the respective security module which is developed in the context of WebSand, the application's resources that are protected by the respective policy rule, a set of conditional criteria that direct the security module's outcome, and a set of additional parameters that are needed in order to facilitate the request handling in special cases. The suitability of the policy mechanism has been shown for each of the five scenarios.

References

[1] Amazon Payments. https://payments.amazon.com.

[2] M. Atwood, R. M. Conlan, B. Cook, L. Culver, K. Elliott-McCrea, L. Halff, E. Hammer-Lahav, B. Laurie, C. Messina, J. Panzer, S. Quigley, D. Recordon, E. Sandler, J. Sergent, T. Sieling, B. Slesinsky, and A. Smith. OAuth Core 1.0. [online], OAuthCore1.0, December 2007.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. http://tools.ietf.org/html/rfc3986, August 2005.

[4] Google Authenticator – two-step verification. http://code.google.com/p/google-authenticator.

[5] Google Checkout. https://checkout.google.com.

[6] Interspire. http://www.interspire.com.

[7] M. Johns, B. Braun, M. Schrank, and J. Posegga. Reliable Protection Against Session Fixation Attacks. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1531–1537. ACM, 2011.

[8] M. Johns and S. Lekies. Biting the hand that serves you: A closer look at client-side flash proxies for cross-domain requests. In *Proceedings of the 8th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[9] S. Lekies, M. Johns, and W. Tighzert. The state of the cross-domain nation. In *Proceedings of the 5th Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.

[10] R. Moats. Urn syntax. http://tools.ietf.org/html/rfc2141, May 1997.

[11] D. M'Raihi, M. Bellare, F. Hoornaert, D. Naccache, and O. Ranen. Hotp: An hmac-based one-time password algorithm. http://tools.ietf.org/html/rfc4226, December 2005.

[12] D. M'Raihi, S. Machani, M. Pei, and J. Rydell. Totp: Time-based one-time password algorithm. http://tools.ietf.org/id/draft-mraihi-totp-timebased-06.txt, September 2010.

[13] N. Nikiforakis, W. Joosen, and M. Johns. Abusing Locality in Shared Web Hosting. In *4th European Workshop on System Security (EU-ROSEC'11)*, 2011.

[14] N. Nikiforakis, W. Meert, Y. Younan, M. Johns, and W. Joosen. SessionShield: Lightweight Protection against Session Hijacking. In *3rd International Symposium on Engineering Secure Software and Systems (ESSoS '11)*, 2011.

[15] nopCommerce. http://www.nopcommerce.com.

[16] R. Paleari, D. Marrone, D. Bruschi, and M. Monga. On race vulnerabilities in web applications. In *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 126–142, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Paypal. https://www.paypal.com.

[18] D. Recordon and D. Reed. Openid 2.0: a platform for user-centric identity management. In *DIM '06: Proceedings of the second ACM workshop on Digital identity management*, pages 11–16, New York, NY, USA, 2006. ACM.

[19] M. Schrank, B. Braun, M. Johns, and J. Posegga. Session fixation: the forgotten vulnerability? In *Sicherheit 2010: Sicherheit, Schutz und Zuverlässigkeit*, pages 341–352. Gesellschaft für Informatik, 2010.

[20] A. van Kesteren (Editor). Cross-Origin Resource Sharing. W3C Working Draft, Version WD-cors-20100727, http://www.w3.org/TR/cors/, July 2010.

[21] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to Shop for Free Online - Security Analysis of Cashier-as-a-Service Based Web Stores. In *IEEE Symposium on Security and Privacy*, pages 465–480, 2011.