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

https://www.websand.eu

Deliverable D5.1
Integrated WebSand Architecture

Abstract

This deliverable reports on the architecture design of the WebSand framework. The framework provide the basic infrastructure and baseline security services for all WebSand enabled applications. Furthermore, the framework consist of four main components, each taking effect in a dedicated phase during processing an HTTP request/response pair.

Deliverable details

Deliverable version: *v1.0*
Classification: *public*
Date of delivery: *31.05.2012*
Due on: *31.05.2012*
Editors: *Martin Johns, Sebastian Lekies*
Total pages: *32*

List of Contributors:
Bastian Braun, Lieven Desmet, Martin Johns, Sebastian Lekies, 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

Document revision history

	Responsible	Date
Initial revision	Martin Johns	May 30th, 2012
Review		June ___, 2012
Final version	Martin Johns	June ___, 2012

Executive Summary

In this document, an overview of the WebSand framework's architecture is given. The WebSand framework provides the general infrastructure, which allows an integration of the technical measures which are developed individually in work packages WP2, WP3, and WP4.

The framework consists of four main components:

- An HTTP request pre-processing unit,
- an API-layer for direct interaction with the application's business logic,
- an HTTP resonse post-processing unit,
- and elements for enforcing security properties during client-side execution.

The technical contributions of WP2 - WP4 are encapsulated in security modules. The framework's server-side components provide clearly defined interfaces, which can be utilized by the modules to hook into the HTTP request processing and provide APIs to the business logic.

Besides discussing the general architecture of the framework, this document shows how baseline security services, such as secure Web session management, are provided and outlines how the first technical prototypes (D2.2, D3.2, and D4.2) can be integrated into an application scenario using the framework.

Contents

List of Figures

1 Introduction

In this document, a general overview over the architecture and the underlying design decisions of the WebSand framework is given.

1.1 The bigger picture

However, before we will dive into this specific topic, we briefly revisit the bigger picture. The goal of the WebSand project is to empower developers to create sophisticated modern Web applications, that realize application scenarios that exceed the common client/server paradigm of the so called *Web 1.0*. Instead, we consider situations, in which multiple principals interact within the realm of a single application, using a heterogeneous set of interaction methods, ranging from backend B2B networking, over a series of sequential HTTP redirects, up to direct JavaScript API level communication within a single Web page. Hence, WebSand primary target is the developer community.

1.2 The WebSand framework

The core of WebSand is a server-side framework, which enable the development of secure Web applications for such non-trivial scenarios. This framework will be the topic of this document. Furthermore, WebSand's main contribution are a set of dedicated technologies and approaches, that allow the implementation of *secure Web interaction*, *information flow security*, and *secure composition*. It is the duty of the WebSand framework, to provide an infrastructure that is capable of incorporating these isolated measures so that they can function together within a single application scenario.

1.3 Technical foundation

The current prototypical implementation of the WebSand framework relies on the Java's Application server paradigm, which is a subset of the J2EE architecture [3]. For this reason, all practical examples, which show actual program code, in this document will utilize Java as programmin language and the JSP templating format.

However, please note that the concepts of the WebSand framework, which are discussed in this document, are not bound to the Java plattform. Instead, the underlying mechanisms can be utilized with any Web application plattform, such as PHP or Ruby on Rails.

1.4 Document organisation

This document describes the initial architecture of the framework. It is organized as follows: First, in Section 2.2, we a general overview of the framework's structure and components. Furthermore, in this section, we discuss how baseline services, such as Web session management are handled automatically by the framework. Then, we explore the four main components of the framework respectively in Sections 3, 4, 5, and 6. Finally, in Section 7 we outline, how the prototypes of the deliverables D2.2, D3.2, and D4.2 can be integrated in the framework. An API level view on the framework is given in Appendix A.

2 Design Overview and General Architecture

In this section we give a brief, coarse-grained overview of the general design, the individual components, and how they are interconnected.

2.1 High level overview

A WebSand application consists of three distinct components: The *general framework*, the *security modules*, the application's *business logic* and optional *client-side components*. For further reading of this document, it is essential to understand the distinct roles of these components:

2.1.1 General framework

The *general framework* provides the essential infrastructure, on which all further processing HTTP communication relies on: For one, the framework is responsible to handle incoming HTTP request and preprocess the received data.

The framework routes the received data to the responsible *security modules* (see below) for further processing and security decisions. Finally, the framework assembles and post-processes the outgoing HTTP responses, partially based on functionality provided by the security modules.

Please note, appart from the fundamental security services, the general framework does not implement specific security functionality. This is handled within the modules.

Further details on the framework are provided in Section 2.3.

2.1.2 Security modules

The *security modules* are the heart of WebSand's decision and enforcement approach. All individual technical measures, such as enforcement of information flow security or verification of control flow integrity, are handled in dedicated modules.

Depending on a module's scope, incoming or outgoing HTTP data is routed through the module. This way, the framework's pre-processor and post-processor can be outfitted with specific security functionality.

Furthermore, each module exposes an API to the Web application which allows to explicitly regulate the module's actions and to query further information from the module, in the context of the currently processed request.

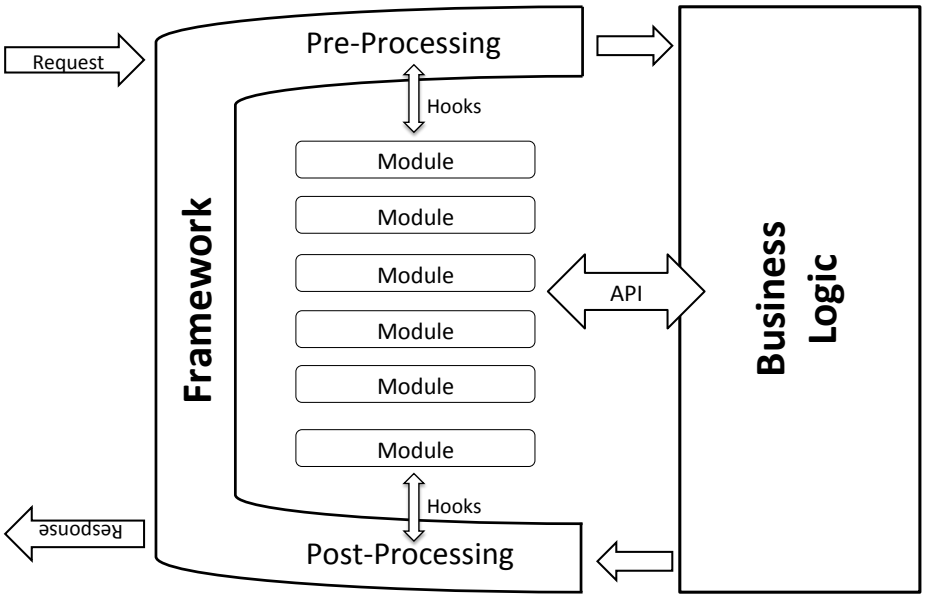

Figure 1: Framework components

2.1.3 Business logic

The *business logic* implements the actual Web application. Using the pre-processed incoming request data, which is provided by the framework, it computes the relevant actions, conducts changes in the application's state, and composes the HTTP/HTML response.

To interact with the security modules and the framework, an API is provided which exposes the respective interfaces to the business logic's code (see Sec. 4). This way, Web site elements, such as mash-up components, or policy configurations, such as information flow labels for DOM properties, can be added by the developer from within his application code.

2.1.4 Client-side components

Appart from the server-side code which consists of the framework, modules and business logic, WebSand also takes effect on the client side, i.e., in the Web browser. Depending on the respective technical measure, such client-side components can be JavaScript libraries, which provide client-side security guarantees, browser enhancement, e.g., in the form of browser extensions, or dedicated devices, such as mobile phones.

2.2 Staged request/response handling

As motivated above, the WebSand framework consists of a set of specialized security techniques, each implemented as security modules and client-side components. Each of these techniques realizes a dedicated security feature, e.g., enforcing secure information flow properties.

The implementation and enforcement of WebSand's security features happens at four distinct stages: Pre-processing, server-side execution, post-processing, and client-side execution. A given security technique can cause effects or expose interfaces at multiple of these stages.

Using the live-cycle of a HTTP request/response pair, we show how this approach take effect on each of these individual stages:

1. **Pre-processing:** *Incoming HTTP request processing unit*

   The incoming HTTP data is received and transparently pre-processed by the general framework. Security modules can register to receive incoming data (either all or only subsets, which match certain criteria). The framework routes the request data accordingly trough the registered modules. These modules then analyse the data and, if necessary, cause the framework to reject the request (e.g., in case of a identified security violation) or, if applicable, modify the request with further meta-data which will be used at the further stages. After each module has processed the request, in general the resulting internal representation is passed on to the next stage.

   For specific use-cases, modules can register to take over the handling of specific request completely (e.g., based on the URL of the incoming request). In such cases, the request data is not passed on to the server-side execution stage. Instead, the corresponding HTTP response is composed directly by the security module.

2. **Server-side execution:** *API-level processing*

   At this stage, the application's business logic is executed. For this execution, the pre-processed representation of the incoming request data is utilized.

   The WebSand security modules expose an API to the developer. The methods of the API can be used within the business logic to directly interact with the modules. This way the developer can query a module for further information, update the module's internal state, or configure the module to take certain actions, e.g., in the post-processing stage.

   Furthermore, at this stage, the outgoing response, in the form of an internal object-based representation is composed. Using module API,

the developer can outfit this response with module provided code, e.g., to include secure Web mashup components in the response's HTML content.

3. **Post-processing: Outgoing HTTP response processing unit**

   In the post-processing stage, the internal response object is translated into the actual HTTP response. Similar to the pre-processing stage, security modules can register to take effect during this stage. The response object is routed through each registered module. These module can now analyse and modify the response data. Potential actions, can be for instance, the source-to-source translation of the response's JavaScript content, the setting of additional HTTP response headers, or the updating of a module's internal state, based on the encountered data in the response object.

4. **Client-side execution: Measures taking effect in the Web browser**

   Finally, a subset of enforcement techniques function on the client-side, i.e., in the user's Web browser or other devices which are owned by the user.

   In this stage, three different enforcement approaches are available to be utilized by the individual security measures:

   - *Server-side provided code:* The WebSand modules or the business logic can cause the deployment of dedicated JavaScript libraries or code snippets in the application's HTML content, which implement security features. An example in this area is the restriction of untrusted JavaScript using function wrappers for the DOM [5].
   - *Enhanced browser components:* In application scenarios with a well defined, closed group of users (e.g., for company internal applications), the usage of enhanced Web browsers, with additional security capabilities, e.g., in the form of browser extensions, is a promising direction. Thus, WebSand supports in usage of such components in the client-side execution stage. Outgoing HTTP responses can carry meta information, e.g., instantiated security policies, or extension-specific mark-up elements, which are supplied through the API or post-processing stage. This information and markup controls how the client-side measures function during Web page rendering and JavaScript execution. An example for this approach is the WebJail [1] extension.

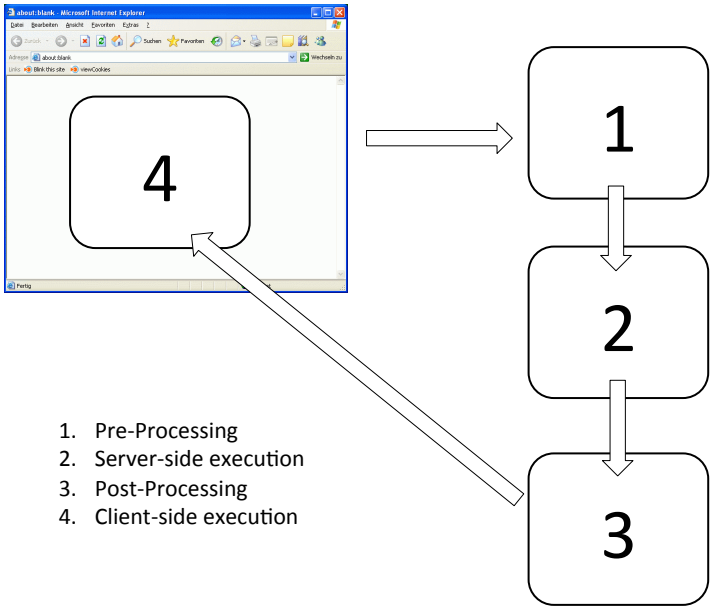

Figure 2: The four distinct stages of enforcement

- *Utilization of dedicated devices:* Finally, under circumstances that demand extensive security guarantees, it is also possible to integrate dedicated devices, such as mobile phones, into the client-side stage. These devices can take over certain steps, e.g., to provide two-factor authentication[2].

Example: As stated above, a given security module can cause effects on multiple of these stages. Take for example the module for control flow integrity enforcement. This module enforces strict workflows during interaction with the application through ensuring that HTTP requests are received in an expected order. These workflows are represented in the module as finite state machines. The module will register for the *pre-processing stage*, to examine all incoming requests, if they satisfy the application's control flow requirements. Furthermore, it will expose an API on the *Server-side execution stage*, which allows the developer to explicitly trigger state transitions in the module's internal state-machine representation of the application's control flow. Finally, such state transitions can be triggered automatically during the *post-processing stage*, based on defined criteria concerning the outgoing HTTP response.

2.3 Framework architecture

Figure 3 depicts the general architecture of the Websand framework. As shown, the picture is divided into two separate blocks. The first one contains Web Application specific building blocks such as the Gatekeeper and the Application Logic, while the second block contains the components of the security framework.

Application Logic : Besides the actual business logic, this building block also contains the application specific Gatekeeper. The Gatekeeper intercepts incoming HTTP requests and outgoing HTTP responses. As the properties of these requests and responses are very application depended, the Gatekeeper conducts a rewriting of these entities into an internal security framework representation. This behavior ensures a good interoperability of the security framework and the Web application. Following this step, the Gatekeeper delegates the request/response objects to the security framework for pre- and post-processing by utilizing the so-called Gatekeeper API.

Security Framework For pre- and post-processing the security framework offers a request processor that determines, in cooperation with the so-called Module Selector, which security modules are relevant for the current request/response pair. The modules contain the implemented security logic and provide these functionalities via two distinct APIs.

1. **Gatekeeper API:** The Gatekeeper API is used during preprocessing to determine whether a request is of legitimate nature or not. More details on the request processing step can be found in Sections 3
2. **Application API** The Application API can be used to provide functionality of the security modules to the application logic runtime. More information on this API can be found in Section 4

The configuration parameters that are needed to select the relevant modules and to carry out the security critical steps are contained within the policies defined by the Web application maintainer.

2.4 Base-line security modules

While each technical work package (2,3,4) within the Websand framework provides several different modules to the security framework, there is functionality that is not part of Websand's core contributions, but has to be implemented into the security framework anyway. For example, the Session,

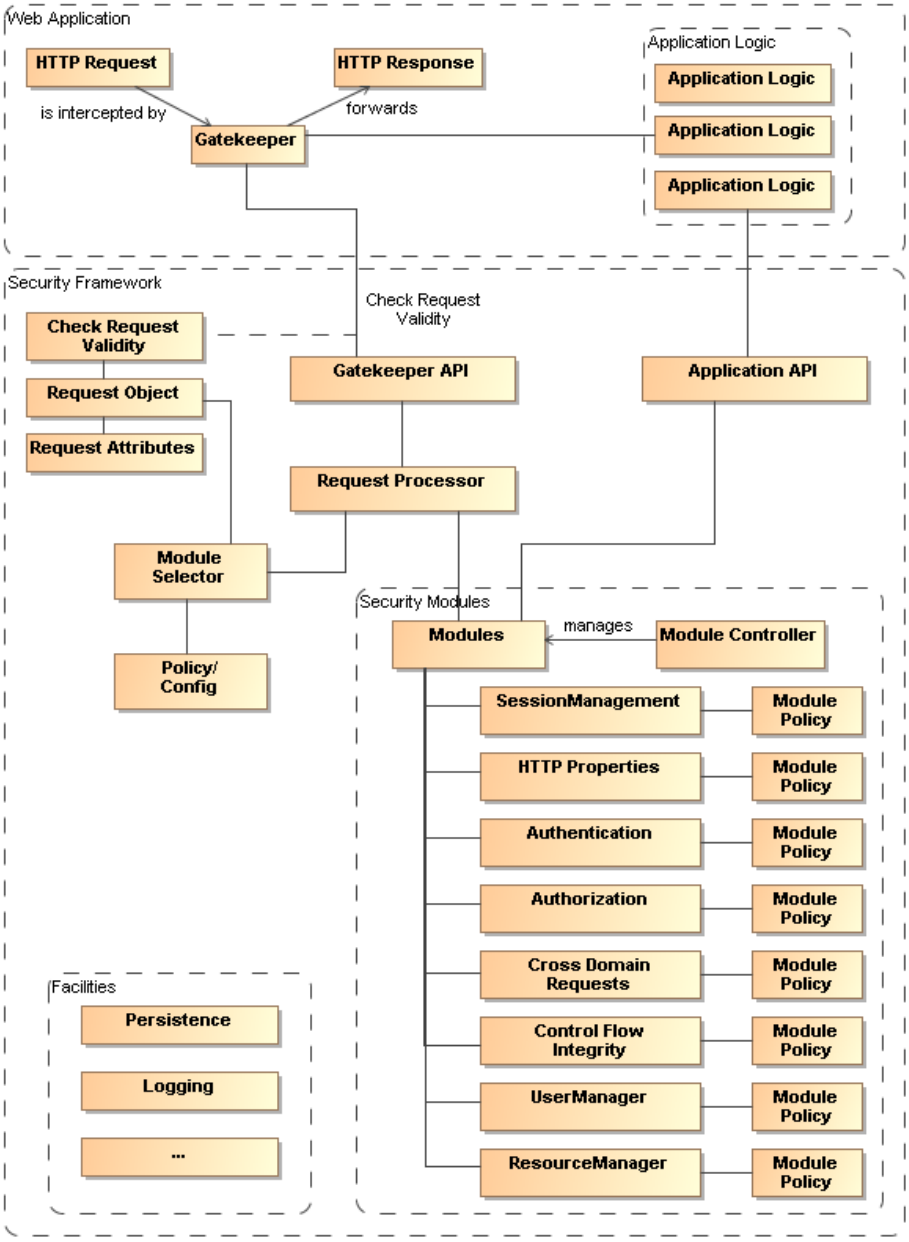

Figure 3: WebSand Framework

User and Resource management modules provide the basic functionality required by any other security module to function properly. Furthermore, we also regard the role-based access control module as such a base-line module required by any Web application.Hence, the minimum set of modules required to run the framework are composed of these base-line services:

- **Resource management:** By default the Websand framework rejects any HTTP request received by the Gatekeeper in order to avoid the leakage of private files from the Web server. In order to accept a request, the requested resource needs to be whitelisted within the so-called resource manager module. At each request this module checks whether the requested file is part of the legitimate Web application and if so grants the request. If a file does not exist or was not marked by the administrator to be part of the Web application it rejects the request.

- **Session management:** The session management module provides all the functionality that is needed for session tracking to the application. It is responsible of extracting and setting of cookie values and rejecting requests with potentially malicious or insufficient session information. Furthermore, it implements best practices for session management such as the usage of a cryptographically strong session id generator and the setting of security related cookie flags(HTTPOnly, Secure). Within the Websand project several papers were written on session security [8], [4], [6], [7]. The results presented in these papers are also included within the session management module. So, for example, the Session management module also implements protection against session fixation as described in [4] or against misuse of shared session storage of Web applications [6].

- **User Management:** Any sophisticated Web application is in need to provide some kind of user management capability. So for example, an online shop has to implement a very secure system to store, edit and retrieve information of certain user accounts including potentially sensitive data. Thereby, the core functionality of such user management systems is application independent. Hence there is no need to reimplement this kind of security critical features again and again for each Web application. Therefore, the Websand framework provides this functionality to the Web application in the form of the user management module.

- **Role-based access control:** Besides the management of user data, modern Web application are also in need of access control systems in

order to restrict access to certain resources. One of the core features that is built upon the user and resource management modules is the role-based access control module. Via Websand's policy system an administrator is able to define a set of roles, assign these roles to certain users (integration with user management module) and request certain roles whenever security sensitive resources are accessed(integration with resource management)

3 Incoming Request Processing

When processing an HTTP request several security critical decisions have to be taken. While some of those can only be decided within the business logic of the applications, others can already be taken on a request basis. So whenever a request carries certain criteria, it can either be granted in legitimate cases or should be denied under potentially malicious circumstances. In order to tell such requests apart, the Websand framework conducts a request preprocessing in which each module has a saying in the process of determining the legitimacy of a request.

Each request that targets the Websand-protected application arrives at the gatekeeper first. The gatekeeper is an application-specific component that provides the glue code between the Web application and the Websand framework. By rewriting the external HTTP request into an internal framework representation, the gatekeeper ensures a good interoperability between the different modules of the Websand framework and the Web application.

After the rewriting step, the internal representation is passed to different modules based on pre-defined policies. So, for example if a resource requested by the client should only be available in a certain authentication context (according to the given policy), the Websand framework passes the request to the Authentication/Authorization modules in order to decide upon the legitimacy. Depending on the given policies, the module selector selects none, some or all of the available modules for further investigation. Each module is granted full access to the internal request object. Based on this information and additional configuration parameters it decides upon the legitimacy of the request. A request is only passed to the web application if all of the previously selected modules agree upon the legitimacy of the request (by returning a positive boolean flag). If one of the modules is in doubt about the legitimacy it is able to return a negative boolean flag and hence the request will not be passed to the application. Instead an error message will be displayed to the user.

In order to illustrate the intented behavior this paragraph will give a short example for the request processing step. The Web application on `http://example.org` hosts an admin panel under `http://example.org/admin.jsp` that offers an easy way to alter content and corresponding user permissions. As this is obviously a very sensitive functionality it should only be accessible to users that present a valid set of admin credentials to the Web application. Hence the administration panel is protected by the authorization/authentication modules of the Websand security framework. Each request that targets the Web application is processed by the gatekeeper first in order to translate it into the internal representation. One step within this pro-

cess for example.org is to extract the so-called session id of a user from the cookie value contained within the original HTTP request. The internal representation of the request is then passed to the request processor which selects the appropriate modules for the requested resource. If the request is targeted towards the admin panel the request processor is able to learn from the policies specified by the webmaster, that the requested resources require proper authorization/authentication. Hence it will select the authentication and authorization modules and pass the internal request object to these modules. If a session id was not present within the initial request, the modules will immediately indicate that the request is not a legitimate one and hence the gatekeeper will refuse the request. If a valid id was found the authorization/authentication modules will query the session module for the server-side session data of the specific id. With the received data the authentication module is able to gain an insight into the user's identity. Based on the given identity the authorization module is able to query the database for the corresponding access permissions and the permissions needed to access the admin panel. If the user provided permissions exceed the required permissions, the request is declared to be legitimate otherwise it is declared illegitimate.

4 API-level

The steps described in the previous section make use of the so-called gatekeeper API to verify the permissibility of an HTTP request. However, it is not always possible to determine what security critical decisions have to be taken by looking at the request object only. Sometimes also tight integration with the underlying business logic is needed to decide upon the legitimacy of a request or upon subsequent actions that need to be conducted as a result of actions carried out on the business layer (e.g. Deployment of an HTTP response header, adapting the state of the security framework). Therefore, each module of the Websand framework also offers an API to the business layer that can be used by the business logic to communicate with the security framework while business transactions are carried out.

In the following we give a short example when the so-called application API is needed: A user is authenticated to an application that handles flight bookings. In order to book a flight the user starts the booking process which is protected by the control flow integrity module. Hence, this module verifies that the user is only able to access the different steps within the booking process in the correct order to avoid a confusions of the internal state of the application. At each arriving HTTP request of the particular user, the request pre-processing ensures that only requests targeting the next step within the process are routed to the application. Any other request will be refused by the gatekeeper. In order to be able to verify the legitimacy of the request the control flow module holds a model of the underlying business process and tracks the state of the current user. For this example, we assume that the user only sends legitimate requests and hence at each request the control flow module will grant the HTTP request and will update the state according to the actions of the user. However, the security framework is not aware of the business critical data (only the business layer is aware of that). So, for example, if the user tries to book a flight that is already sold out, the security framework will not be aware of this fact and update the state and route the legitimate request to the application. When processing the request the business logic detects that the flight is not available anymore and sends the user one step back to choose another flight. Now, the application is in need to roll back the state of the security framework, so that the user is able to resend the request for a new flight. Otherwise, the security framework's state would be inconsistent with the application's state. Therefore, the application is able to use the application API to tell the framework that the state transition has not taken place due to the non-availability of the flight.

Listing 1 depicts two examples of the API usage within a JSP application (note: Listing 1 only shows pseudo code). While the first example illustrates

Listing 1 Exemplified usage of the API-layer for adding information flow policies and composition elements

```
<%
IFmod = WebSand.ApplicationAPI
                .getModule(InformationFlowSecurity.class);
IFmod.setHigh([...list of JS-properties...], IFpolicy);
%>

[...]
<...HTML markup...>
[...]

<%
WJmod = WebSand.ApplicationAPI
                .getModule(WebJailModule.class);
WJmod.addWebJail("http://googlemaps.com/map.js", WJpolicy);
%>

[...]
```

the usage of the Information Flow module, the second example shows the usage of the Webjail (Secure Composition) component. In order to receive the application API for a specific module, the Web application is able to utilize an API that is provided by the Websand framework (See Line 2 in Listing 1). After obtaining the reference to the API the Web application can simply call any function that is offered by the specific module API.

5 Outgoing Response Processing

Besides investigating the request or communicating with the business logic, the security framework is sometimes in need of altering the outgoing HTTP response in order to add security critical properties to it. Such a property can either be represented as an HTTP response header (e.g. X-Frame-Options header, Cross-Origin Resource Sharing policy header) or by additions added to client-side code or markup delivered to the browser within the response body. Therefore, the framework also offers an API to the security modules in order to alter HTTP response header fields and the response body.

Often, this functionality is needed to pass further information to the client-side enforcement mechanisms in the form of policies passed within HTTP response header fields. One example for such an HTTP header field is the Cross-Origin Resource Sharing (CORS) mechanism. CORS is a mechanism that offers a secure cross-domain communication with the help of JavaScript's XMLHttpRequest. In earlier versions of HTML and JavaScript reading the response of cross-domain requests was forbidden due to related security issues. However, at the same time this is a feature that is often needed for modern mashup-style communication. Therefore, the CORS mechanism was introduced with HTML5. In order to allow a cross-domain request, the receiving application has to attach a special HTTP header to the response. Based on the data provided within this header field the browser will either grant read access to the response or deny it. Hence this header is used as a special policy to instrument client-side enforcement mechanisms. In many cases the Websand framework will utilize such headers to either instrument browser-based security mechanisms or to transfer policies to Websand-based client-side components.

Another feature that relies on the response processing is response rewriting. In some cases, the Websand framework needs to alter the HTML response created by the Web application in order to add security related information. This is especially useful if security code is difficult to add to a Web site or if the security code would pollute most of the business logic with additional non-business related instructions. Hereby, response rewriting is very similar to techniques such as Aspect-Oriented Programming. Instead of manually polluting the business logic with boilerplate security code fragments, the developer only expresses his wish to add a certain security feature to his Web page by stating it within the Websand policy. The Websand framework will then take care of adding the features to a Web page whenever it is requested. As a big advantage, the developer has not to care about the security code and is thus able to concentrate on the business logic itself while ensuring a good maintainability of the code and reducing implementation errors.

6 Client-side Enforcement

Some security aspects of the WebSand framework can not be enforced on the server-side only. Secure composition of 3rd party web-mashup components (client-side mashups, as described in D1.2) is one example where server-side enforcement only is not sufficient. The server provides the security policy, ie what the 3rd party component is allowed to do and what not, but as the composition is done by the client, only the client itself is able to enforce this policy. Information flow is another example where client-side components are necessary in order to prevent the leakage of confidential information to 3rd party. This requires an enhanced browser able to perform the security checks and enforce the policies. This can be achieved using two approaches, scripts include or browser modification in case the former one is not possible.

Script include The security libraries required to perform the checks and enforce the different policies are delivered by the server using script includes. This is the preferred way in the WebSand framework as it doesn't require any browser modification.

Enhanced browser When the client-side security checks have to extend functionalities of the browser and the browser do not provide an interface do to it or if the interface can not be called from a script, script includes are not sufficient. The browser has to be modified, either by modifying its code or by providing a browser's extension that contains the required functionalities.

Although the WebSand framework supports the two approaches, the usage of script include is preferred as less intrusive as a modified browser. As outlined in Section 7, only a few aspects of the WebSand framework require a modified browser.

7 Integration of WebSand Components

In this section we provide an overview on how the different prototypes responsible for secure web interaction (described in D2.2), information flow (described in D3.2) and secure composition (described in D4.2) can be swimmingly integrated in the WebSand framework. Each prototype requires at least the addition of a security module (see Sec. 2.1.2 for the definition of a security module). Some require the usage of a client component as the server may not be able to enforce all the security aspects (see Sec. 6). The rest is processed by the framework.

7.1 Secure Web Interaction

The secure web interaction prototype aims at outsourcing the security-context decision (robust session management, authentication tracking...) from the business logic to the security modules. Some are already part of the baseline security services (see Sec. 2.4), the others are described here.

Cross-domain interaction: each incoming request is checked by the incoming request processor for its cross-domain context. This is done by checking the value of the "Origin" HTTP header, if present. If this value is different from the application's domain, the cross-domain module will compare it to a whitelist of allowed domains created by the developer of the web application. Only if this domain is present in the list will the request be allowed. For full compatibility with the CORS standard, the "Access-Control-Allow-Origin" HTTP response header will be set by the module in the outgoing request processor.

Control flow integrity : the Control Flow Integrity module offers mechanisms to enforce a defined order of HTTP requests. More precisely the module doesn't forward requests to the web application if they don't match the underlying workflow. When the webmaster defines such a workflow he creates a control flow graph by choosing the respective resources, for example `/1.jsp`, `/2.jsp` and `/3.jsp`. After that he interconnects these resources so that a graph accrues. Now the module's manager component knows that these files are protected by a control flow graph and uses this graph when any of these resources is requested. The gatekeeper builds the so called FrameworkRequest - an internal independent representation of the original HTTP request - when a HTTP request arrives on the web server. This FrameworkRequest is then passed to all responsible modules typically to the Control Flow Integrity module, too. All responsible modules are determined

with the ModuleSelector by validating their policies. If the requested resource is not component of any graph, the module logically allows the request and forwards it to the web application. In case of an invalid request, for example when the user tries to skip a step, the module rejects the request and an error message will be sent to the client. Thereby the internal state of the module, e.g. the current position in graph, and also the web application is not affected. If the request was allowed by all responsible modules the module's API is used a second time: the SecurityContext that communicates with every module's validation method now tells the Control Flow Integrity module to update the current position of the user in his workflow. The workflows are determined by the session id and an unique token which is generated in the token factory of the module when a user enters a new graph. The single resources respectively nodes of a control flow are clearly identified by the resource name (e.g. "1.jsp") and the request's parameters (e.g. "action=booking"). These parameters' values (here: "booking") are validated with the use of regular expressions. Therefore the developer of the web application could also define that the value for example should only consist of numbers. Moreover an entire graph could only consist of one resource (e.g. "do.jsp") and just the parameters define the distinct order in the control flow. To gain a very flexible design graphs can be connected to each other and re-used several times as sub graphs in other workflows. Because a user can be in different or similar control flows at the same time - for example in different browser tabs - the module instantiates and stores the respective graphs with information about its user (e.g. session id) and the current processing position.

7.2 Secure Composition

The secure composition prototypes enforce least-privilege integration of 3rd party JavaScript code.

WebJail, the first prototype, enforces the least-privilege policy via a modified browser and a security module. When the developer wants to securely integrate a 3rd party component, he can use an API provided by the WebJail security module (see Listing 1) and define a policy in the form of an external file. The output processor will then generate the HTML iframe tag with a new policy attribute containing the url of the policy. The modified browser will then recognize this new attribute, download the policy and enforce it while loading the 3rd party component.

The second prototype gives the possibility to run 3rd party scripts in a sandbox environment and does not require any browser modification. Using the security module's API, the developer can define which 3rd party scripts he wants to include and provide a policy for each script. The outgoing response processor will include the necessary JavaScript libraries (consisting of the Secure ECMAScript library, a baseline API library containing wrappers to mediate access to security-sensitive APIs and a loader library to fetch the 3rd party script) in order to run the code in a sandbox and the policy in the response itself defined as JavaScript code. Any browser supporting JavaScript will then execute the 3rd party code in the provided sandbox.

7.3 Information Flow Security

The business logic manipulates data whose confidentiality and integrity are at stake. This section presents how the information flow prototypes can be integrated in the WebSand framework. Both of them do not require the usage of the incoming request processor but client-side components.

FlowFox is a modified web browser that implements information-flow control for web scripts based on the secure multi-execution technique. The web application developer is able to define a fine-grained policy using the FlowFox security module. The module will create a separate policy file where the different security levels for DOM calls are defined using labels. When rendering the page, the browser will download the policy and enforce it to control the information-flow.

JSFlow is an information-flow aware JavaScript interpreter implemented in JavaScript. The developer define the sources, sinks and secrets using the JSFlow security module's API and the outgoing request processor will include those annotations in the JavaScript code. In order to use the JSFlow interpreter and not the native browser's interpreter, an extension like Zaphod[1] for the Firefox browser has to be installed in the browser to switch the JavaScript rendering engine. JSFlow will then monitor the information flow during the execution in the browser and stop it if necessary to prevent insecure information flow.

[1] https://addons.mozilla.org/en-US/firefox/addon/zaphod/

8 Conclusion

In this document, we explained the architecture and design of the WebSand framework. The framework consists of four main components, namely the pre-processing unit, the API-layer interaction, the post-processing unit, and client-side measures, that take effect in the user's browser. For all these components, the functionality scope and their position within the enclosing architecture have been discussed. Finally, we outlined, how the first WebSand prototypes, which were the subject of deliverables D2.2, D3.2, and D4.2, can be integrated into Web applications using the framework.

The WebSand framework will provide the infrastructure, which allows to create the use cases applications, which will be implemented in the third year of the project, using the individual security measures of WP2, WP3, and WP4.

A Appendix

The Appendix lists the Gatekeeper API and the Application APIs of the already integrated modules. Experimental modules are not yet included in this list, but will be added in the next deliverables.

A.1 Gatekeeper API

Listing 2 Gatekeeper API in Java IDL

```
module GatekeeperAPI
{
    interface GatekeeperAPI
    {
        boolean isDBAndConfigInitValid();
        isValid(FrameworkRequest req);
        void shutDown();
    };
};
```

A.2 Application API

Listing 3 Authentication API in Java IDL

```
module ApplicationAPI
{
    interface AuthenticationInterface
    {
        void addUser(Credentials credentials);
        boolean authenticate(Credentials credentials,
                    HttpServletRequest req);
    };
};
```

Listing 4 ControlFlow API in Java IDL

```
module ApplicationAPI ControlFlowInterface
{
    interface
    {
        boolean isRequestAllowed(FrameworkRequest request);
        void movePositionOfUser(FrameworkRequest request);
        void loadControlFlowGraph();
        void deleteControlFlowGraph(int graphToken);
    };
};
```

Listing 5 CrossDomain API in Java IDL

```
module ApplicationAPI
{
    interface CrossDomainInterface
    {
        boolean isWhitelisted(String origin);
    };
};
```

Listing 6 Rbac API in Java IDL

```
module ApplicationAPI
{
    interface RbacInterface
    {
        void addRole(String role);
        boolean doesRoleExist(String role);
        void addUserToRole(String user,
                           String role);
        boolean isUserInRole(String user,
                           String role);
        void addRoleToResource(String role,
                           String resource);
        boolean isResourceInRole(String role,
                           String resource);
        boolean doesUserHaveAccess(String user,
                           String resource);

    };
};
```

Listing 7 ResourceManager API in Java IDL

```
module ApplicationAPI
{
    interface ResourceManagerInterface
    {
        void addResource(String resource);
        boolean doesResourceExist(String resource);
        List<String> getResources();
    };
};
```

Listing 8 SessionManager API in Java IDL

```
module ApplicationAPI
{
    interface SessionManagerInterface
    {
        void createNewSession(HttpServletRequest req);
        void invalidateSession(HttpSession session);
    };
};
```

Listing 9 UserManager API in Java IDL

```
module ApplicationAPI
{
    interface UserManagerInterface
    {
        void addUser(String user);
        boolean doesUserExist(String user);
        List<String> getUsers();
    };
};
```

References

[1] S. V. Acker, P. D. Ryck, L. Desmet, F. Piessens, and W. Joosen. WebJail: Least-privilege Integration of Third-party Components in Web Mashups. In *Proceedings of the ACSAC 2011 conference*, 2011.

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

[3] E. Jendrock, S. Bodoff, D. Green, K. Haase, M. Pawlan, and B. Stearns. The j2ee tutorial. , 2002.

[4] 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.

[5] J. Magazinius, P. Phung, and D. Sands. Safe wrappers and sane policies for self protecting javascript. In *15th Nordic Conference on Secure IT Systems*, 2010.

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

[7] 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.

[8] 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.