OMOS: A Framework for Secure Communication in Mashup Applications [*]

Saman Zarandioon Danfeng (Daphne) Yao Vinod Ganapathy
Department of Computer Science
Rutgers University
Piscataway, NJ 08854
{samanz,danfeng,vinodg}@cs.rutgers.edu

Abstract

Mashups are new Web 2.0 applications that seamlessly combine contents from multiple heterogeneous data sources into one integrated browser environment. The hallmark of these applications is to facilitate dynamic information sharing and analysis, thereby creating a more integrated and convenient experience for end-users. As mashups evolve from portals designed to offer convenient access to information on critical domains, such as banking, shopping, investment, enterprise mashups, and web desktops, concerns to protect clients' personal information and trade secrets become important, thereby motivating the need for strong security guarantees. We develop a security architecture that provides high assurance on the mutual authentication, data confidentiality, and message integrity of mashup applications. In this paper, we describe the design and implementation of OpenMashupOS (OMOS), an open-source browser-independent framework for secure inter-domain communication and mashup development.

1. Introduction

Mashup applications are emerging as a Web 2.0 technology to seamlessly combine contents from multiple heterogeneous data sources; their overall goal is to create a more integrated and convenient experience for end users. For example, `http://mapdango.com` is a mashup application that integrates Google Maps data with relevant information from WeatherBug, Flickr, Eventful, etc. By entering a location, the user is presented with an integrated view of the weather of the location, events happening in surrounding area, photos that others took in the area, and so on. There are two main types of architectures for mashup applications, namely, server-side and client-side architectures. As the name indicates, server-side mashups integrate data from different sources at the server-side, and return the aggregated page to the client. For example, Facebook mashup APIs are mainly based on server-side integration [8].

However, the main drawback of server-side mashups is the requirement of complete trust on the mashup server by the client. Typically, the client needs to delegate authorization to the mashup server to act on its behalf.

In comparison, a client-side architecture, as illustrated in Figure 1, enables consumers and service providers to communicate within a browser, thus reduces the amount of trust that one has to place on untrusted third-party integrator. OpenSocial provides a client-side mashup API [17]. Throughout this paper, we focus on client-side mashup architecture, as emerging mashup applications using AJAX techniques hold the promise of the next technical wave of the future [2]. AJAX, short for asynchronous JavaScript with XML, is a technique that allows a Web page to retrieve contents from the Web server and update the page asynchronously using JavaScript. AJAX mashups are able to present a rich user interface and interactive experience with multiple data sources with minimal transmission delays.

Client-side mashup architectures allow information mashup to happen within the client's browser through the use of JavaScript. A mashup application should be able to access and integrate contents from different sources. In general, there is a trade-off between the security and functionality in today's mashup applications. In order to achieve higher security guarantees, a source should not be allowed to access contents of another domain. The `frame` or `iframe` element in the current browsers realizes this separation by forbidding one frame from accessing another frame with a different source domain. However, frame environments make it awkward for cross-frame communication and thus information integration in mashups.

To address this problem, several client-side mashup architectures have recently been proposed, including MashupOS [19], Subspace [11] and SMash [13].The main goal of

[*]This work has been supported in part by the Rutgers University Computing Coordination Council Pervasive Computing Initiative Grant, Rutgers University Research Council Grant, and a grant from Rutgers School of Arts and Science.

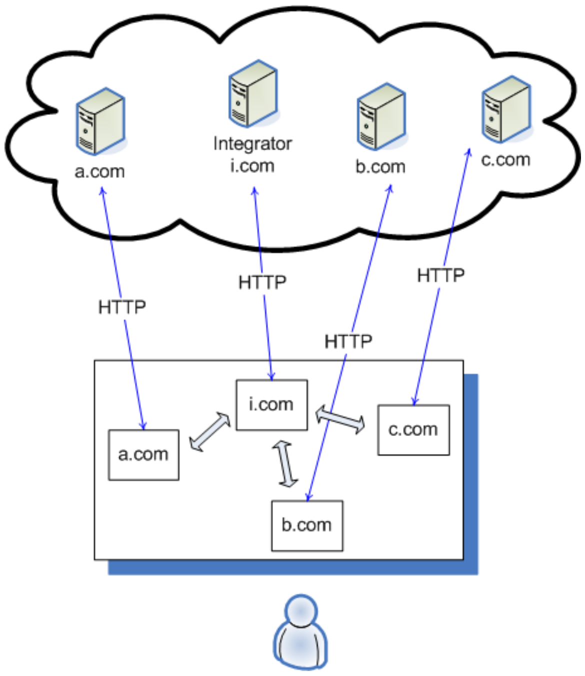

Figure 1. Client-side mashup architecture. The rectangle represents the browser on the client's local computer where contents from heterogeneous data sources such as a.com and b.com are mashed up.

these solutions is two-fold: to isolate content from different sources in sandboxes, such as frames, and to achieve frame-to-frame communication.

SMash [13] uses the concepts in publish-subscribe systems and creates an event hub abstraction that allows the mashup integrator to securely coordinate and manage content and information sharing from multiple domains. The mashup integrator (i.e., the event hub) is assumed to be trusted by all the web services. The event hub implements the access policies that governs the communication among domains.

MashupOS [19] introduces a sophisticated abstraction that enables web components from different domains to securely communicate. OMash [6], inspired by MashupOS, tries to simplify the abstraction and remove the reliance on same origin policy (explained in Section 2). However, implementing these abstractions requires adding new elements to the HTML standard and changing browsers to support them.

Subspace [11] suggests an efficient techniques for `www.mashup.com` to use a JavaScript Web service from `webservice.com` by sandboxing the Web service in a frame that is originated from a *throwaway* sub-domain (e.g. `webservice.mashup.com`) and communicating with it by shortening document.domain to a common suffix and passing a JavaScript object that can be used for secure communication. The main drawback of this approach is that, due to the same origin policy, the Web service running from `webservice.mashup.com`, cannot use XMLHTTPRequest to communicate with its resources on backend site (`webservice.com`) and this restriction limits the use of AJAX Web services.

However, none of these solutions provide a flexible and secure point-to-point inter-domain communication mechanism that can be used in today's browsers. In this paper, we fill in this gap by considering the following design goals:

- To be compatible with all major browsers without any change or extension to the browsers.
- To provide a powerful abstraction that is flexible and easy to understand and use by mashup developers.
- To guarantee mutual authentication, data confidentiality, and message integrity in mashup applications. (Defined in Section 2.)

The novel features of our approach are as follows. First, we present key-based protocol for secure asynchronous point-to-pint inter-origin communication. Second, we separate communication layer from access control layer, therefore the framework can be used using different access control mechanism. Third, we present a layered communication abstraction for inter-frame communication fashioned after the networking stacks that is both powerful to use and easy to understand. Additionally, the framework does not require any browser change, so it is a good candidate for secure development of today's mashup applications.

The following techniques, enable us to achieve all of our design goals. Our key-based protocol satisfies the security requirements and prevents attackers from phishing, forging, tampering, and eavesdropping on cross-domain communications. Since we do not require new HTML elements, OMOS is compatible with current browsers. The layered abstraction hides implementation details from mashup developers and the API allows anyone to extend and improve any part of the mashup framework. OMOS' communication API and component-based development also make the development of complex AJAX applications much easier. (Reusable components are called mashlets in OMOS, Section 2.)

An additional advantage gained by using our techniques is that the mashup integrator (i.e., mashup site) need no longer be trusted by all the content providers (i.e., web services). This is possible because the frames from different web services are able to directly and securely communicate within the user's browser. Therefore, with OMOS it is possible to create new types of mashup applications that may involve and integrate sensitive and personal data without fully trusting the mashup integrator. For example, banking, shopping, and financial planning applications contain important personal information that users want to have high assurance on the controlled sharing of data. Allowing different domains to communicate in a secure fashion minimizes the potential risks of information exposure due to corrupted websites such as compromised mashup integrators, and untrusted contents from other data sources. We have implemented and evaluated the performance of the OMOS framework on four types of browsers. These initial experiments show that the communication channels are able to deliver high throughput without affecting the end user's browsing experience.

The paper is organized as follows. Basic concepts are defined in Section 2. The architecture and implementation of OMOS framework are presented in Section 3. The security analysis is in Section 4. In Section 5, we describe the experimental results. Related work is explained in Section 6. We give the conclusions and future work in Section 7.

2 Definitions

We define mashlets, gadgets and mashup applications. A *mashlet* is recursively defined as a HTML frame hosting JavaScript service that contains zero or more mashlets. The root mashlet is always visible and is usually called a mashup container. Every mashlet is controlled by and loads contents from its originating domain. Conceptually, mashlets are analogous to processes or daemons in the operating system, binary components (e.g COM/DCOM, DLL) in component-based architectures or web service providers in service oriented architectures (SOA). A *gadget* is a mashlet that is visible in the browser. A *mashup application* is a gadget that integrates data received from other mashlets [1].

Two most important aspects of mashup applications are interaction and security. Interaction refers to the ability of a mashlet to interact with its siblings, children, and parent mashlets. Security requires that a mashlet should not be able to access private information, such as DOM elements, events, memory, and cookies, of any other mashlet that is running under a different domain. In particular, a mashlet should not be able to listen to the communication between two other mashlets running under different domains. We call this requirement *data confidentiality*. In today's browsers, the same origin policy (SOP) [18] is designed to protect data confidentiality of domains against each other; in other words, SOP prevents documents or programs from one origin to access or alter documents loaded from another origin. SOP restrictions on JavaScript that govern the access to inline frames (`iframes`) [2] forbid JavaScript in one mashlet including the root mashlet to read or modify the contents in another mashlet. However, SOP is restrictive and rigid for mashup applications in general. Mashlets from different domains are isolated and cannot communicate or interact unless specifically allowed. Most of existing mashup applications circumvent this restriction either by creating server-side mashups, which is a less flexible approach, or by allowing complete access from other domains. Recently, researchers also demonstrated the vulnerabilities associated with carelessly attempting finer-grained origins [10].

Mutual authentication is another important security requirement in cross-domain mashlet communication. We define *mutual authentication* in mashup applications as the requirement that two mashlets that are communicating with each other must be able to verify each other's domain name.

Mashup applications should also satisfy the *message integrity* requirement that means that any tampering of the messages between two mashlets should be detected. OMOS satisfies the three requirements of data confidentiality, mutual authentication, and message integrity, by leveraging the security restrictions available in current browsers and by developing a lightweight key establishment protocol.

3 Architecture and Implementation of OMOS

In this section, we first give an overview of OMOS, and present its layered communication stack for inter-mashlet communication. Finally, we present some important implementation details of our technique.

3.1 Overview

Our goal is to support secure, asynchronous, inter-mashlet communication in browser environments. Much of our design in OMOS is lead by existing inter-process communications in networking, e.g., TCP. That is, we model the cross-domain frame-to-frame interactions (i.e. a frame communicating with another frame of a different domain) in a manner similar to cross-domain process-to-process interactions in networking paradigm. We develop a layered communication model for the purpose of cross-domain frame-to-frame communications that can be easily extended.

The OMOS framework can be viewed as a container for mashlets that manages their construction, destruction and resources, also provides them with services such as communication, persistence, user interface, user authentication and pub-sub messaging. Services that OMOS provides to mashlets are analogous to services that operating systems provide to desktop applications through well-defined APIs. OMOS runs entirely in the browser, requires no browser plug-in, and supports all main stream browsers, including

[1]This definition concentrates on client-side mashups

[2]Frames that can be inserted within a block of texts.

Firefox, Internet Explore, Safari, and Opera. Figure 2 illustrates how mashlets using OMOS interact.

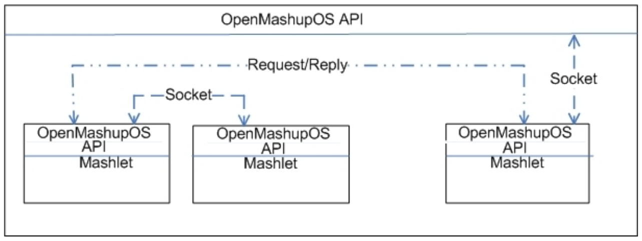

Figure 2. Interactions between mashlets in OMOS **framework. Each mashlet connects to the integrator using a socket connection that** OMOS **uses to provide services to mashlets.**

OMOS uses iframes to implement mashlets. For each mashlet loaded from a distinct domain, existing SOP restriction guarantees the confidentiality and isolation of mashlets. Also OMOS provides mashlets with a flexible, reliable, asynchronous and secure communication service that guarantees data confidentiality, data integrity, and mutual authentication using a layered communication stack.

3.2 Layered Communication Stack

Our OMOS architecture abstracts the mashlet communication and provides mashup developers with a powerful and flexible API. We borrow the concepts in networking to design a communication stack in OMOS. The administrative communications between mashlet and the parent mashlet (i.e., integrator) are done using a *socket connection*. Most of the OMOS service calls through JavaScript APIs lead to a communication through this socket connection. As a result, we are able to support modularity and transparency. Complex implementation details are hidden from the outside. For example, the request to get the DOM address of a specific domain name (or principal) is invisible to mashup developers. Figure 3 depicts the communication layers in OMOS architecture, namely from bottom to top, Datalink layer, Mashup Datagram Protocol (MDP) layer, and Mashup Hypertext Transfer Protocol (MHTTP) layer.

At the Datalink layer, communications are realized in a direct frame-to-frame fashion, which needs to be compliant with restrictions imposed by browsers. For example, the size of data to be transferred is limited depending on the type of the browser and the communication method, and DOM location of an iframe (e.g., parent.frame[3]) is used for addressing. We further discuss the Datalink layer services and implementation techniques in Section 3.4.

The purpose of Mashup Datagram Protocol (MDP) layer

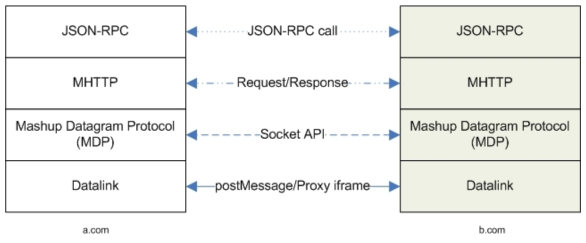

Figure 3. Communication stack in OMOS**. The arrows and their texts are the communication methods for the layers. Note that all the communications between two mashlets take place within the end user's browser.**

is to abstract the Datalink layer details. MDP provides the *logical client-side communication* between two mashlets, in such a way that from a mashlet's perspective, it is directly sending arbitrary sized data to another mashlet. Yet, in reality, the data may be fragmented, defragmented, and reordered which are all handled in the lower Datalink layer. Mashup applications use the logical communications provided by the MDP layer to send data to each other, without worrying about the implementation details of browser types, restrictions, etc. In MDP layer domain names and port numbers are being used for addressing. OMOS exposes services provided by this layer using socket APIs that is very similar to Java socket API for conventional TCP/IP communication. OMOS uses socket connection for administrative communication between mashlets and their parents. During the bootstrapping process, when a mashlet is first loaded, it gets the communication parameters from the segment identifier of its URL provided by the integrator, i.e., parent mashlet. Then the mashlet creates a socket connection to the integrator service on port zero (dedicated for this purpose). Through this bootstrapping process, the integrator establishes connections to all the mashlets that it contains. The integrator uses these connections to provide the services that the mashlets need, e.g., finding the DOM location of a specific mashlet, changing the width and height of their iframes, or resolving domain names to frame address, etc.

Mashup HyperText Transfer Protocol (MHTTP) is the top layer in the communication stack of OMOS. MHTTP provides stateless *request* and *reply* types of communication and abstracts all the details of socket programming. It is very common for service consumers that need to send a request to a service provider and get the corresponding response. It is easy for service providers to define the interface for these types of services with MHTTP.

We use JSON-RPC protocol on top of the MHTTP layer [12]. JSON-RPC is a simple lightweight remote procedure call protocol that is very efficient in AJAX applications [2]. This layer makes it easy to use existing JavaScript services. Instead of directly injecting JavaScript code, service consumer includes the service in a sandbox mashlet and hosts the mashlet in a safe throwaway subdomain. Then the service consumer uses JSON-RPC to call the service and retrieves the result without giving the script full access to resources available in the main domain.

3.3 Implementation Details

In this section, we describe some important implementation details of OMOS. Our descriptions of our communication stack are bottom-up, starting from the Datalink layer. More implementation details can be found at `http://OpenMashupOS.com` [16].

3.4 Datalink Layer

Datalink is the layer that does the actual transfer of data from one frame to another. OMOS currently uses iframe proxy or postMessage (if available) for cross-domain communication between frames. Other communication mechanisms can be implemented and easily plugged into the framework. In Opera and some especial configuration of other browsers, frame navigation is restricted that prevents two mashlets in different frame hierarchies from communicating directly. In this case, if the integrator is not trusted then the communication fails and OMOS will prompt the user to use a browser with permissive navigation policy; otherwise, the data link layer or the integrator mediates and routes the data link packets to the destination.

3.4.1 iframe Proxy and Key Establishment Protocol

For inter-frame communication, if postMessage API is not available, OMOS fails to *iframe proxy* techniques to do inter-frame communication. Browsers enforce a *write-only policy* on URL field of iframes, which means that a frame can *write* to the URL field of a frame with a different origin domain, but *not read*. The URL field of a frame can only be read by the frame itself or a frame of the same origin. Therefore, in OMOS, if iframe A originated from `a.com` wants to pass some data to iframe B from `b.com`, iframe A creates an internal temporary hidden iframe that points to a proxy page that is hosted on b.com and sets the fragment identifier to carry data (for example, `http://b.com/proxy.html#data`). As part of its OnLoad event, the proxy page reads the data from its URL and delivers that to iframe B. The iframe proxy gets removed afterward. This method has the following benefits over the approach that is used in [13]; it is event driven and does not require polling, therefore eliminates the delay between each poll and improves the performance by eliminating unnecessary timers. With this solution, we eliminate the click sound problem that IE has in SMash [3].

Although this event-based communication mechanism through iframe proxy has been documented elsewhere [7], [5], it is not known previously how to achieve mutual authentication in this communication method. When frame A writes `http://b.com/proxy.html#data` as the URL in the iframe proxy, A can make sure that frame B can get the data only if its domain is `b.com` (because of SOP); however when frame B receives data, there is no direct way to find out the origin of the received data. We develop a key establishment protocol in OMOS that is used by two frames to initiate a shared secret key. By leveraging the write-only property of frame URL, the key establishment protocol elegantly allows the two frames to verify each other's domain name (e.g., that iframe[A]'s domain is `a.com` and iframe[B]'s domain is `b.com`).

OMOS key establishment protocol is as follows. Let say frame[1] from `a.com` and frame[2] from `b.com` want to exchange a shared secret key. Frame[1] generates the secret key SK_1 and passes the key to frame[2] using a proxy from `b.com`. Since frame[2] can get the key only if it is originated from `b.com`, frame[2] can prove that its origin is `b.com` by responding back with SK_1. However, `b.com` still needs to verify that the origin of frame[1] is `a.com`. To do so, it generates a new secret key SK_2 and passes it along with SK_1 using `a.com`'s proxy then frame[2] can prove that its origin is `a.com` by responding with SK_2. At this point only `a.com` and `b.com` know SK_2 so they can use it as a shared secret for the rest of communication. Note that key establishment happens during three-way handshake in MDP layer that is described in the next section. Using this protocol, OMOS framework can provide mutual authentication capability in inter-mashlet communication.

Figure 4 illustrates this key establishment protocol. Data fields shown on the arrows between the two frames represent Datalink packets, which encapsulate MDP packets. SK_1 and SK_2 are session secrets chosen by frame[1] and frame[2], respectively for each communication session. EID is an identifier needed by Datalink layer for addressing destination object. Each frame also creates a serial number in each Datalink packet.

3.5 Other Datalink Layer Services

Besides key establishment for mutual authentication, the Datalink layer also provides services like reordering, (de)fragmentation, and (un)piggybacking to enable effi-

[3]A click sound is usually made in IE when a frame is redirected, which can be distracting if it occurs too frequently as the frames URL gets repeatedly updated for the data transfer purpose.

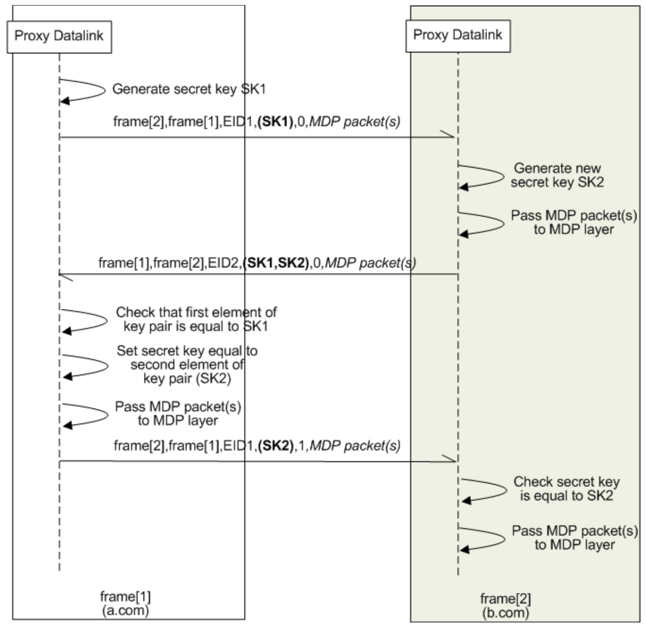
Proxy Datalink
Proxy Datalink
Generate secret key SK1
frame[2],frame[1],EID1,(SK1),0,MDP packet(s)
Generate new secret key SK2
Pass MDP packet(s) to MDP layer
frame[1],frame[2],EID2,(SK1,SK2),0,MDP packet(s)
Check that first element of key pair is equal to SK1
Set secret key equal to second element of key pair (SK2)
Pass MDP packet(s) to MDP layer
frame[2],frame[1],EID1,(SK2),1,MDP packet(s)
Check secret key is equal to SK2
Pass MDP packet(s) to MDP layer
frame[1] (a.com)
frame[2] (b.com)

Figure 4. Key establishment protocol between two mashlets, frame[1] from a.com and frame[2] from b.com, through Datalink layer.

cient transfer of arbitrarily big data objects or frequent events/small objects. We explain them as follows.

- **Fragmentation:** Each browser defines Maximum Transfer Unit (MTU) size that specifies the maximum amount of data a frame can carry in its URL field. When the size of a MDP packet is larger than MTU, the packet should be fragmented to smaller chucks and then sent to the other end, which is called fragmentation. On the other side, the receiver's DataLink layer assembles these fragments and sends the resulting MDP packet up to MDP layer, which is called defragmentation. This service enables transfer of arbitrarily large data objects without interrupting responsiveness of user interface.

- **Reordering:** We observe that in some cases, depending on how event handling is implemented in the browser, packets sent using iframe proxy arrive out of order. Reordering ensures that MDP packets are delivered to MDP layer in the order that they are sent by the sender.

- **Piggybacking:** Piggybacking essentially refers to a lazy-send approach for transferring small data objects. OMOS needs to create a new iframe proxy for every data transfer between two iframes. When the sender has frequent small sized data objects, it is more efficient to collect them and send them together using only one iframe proxy, instead of sending them in multiple iframe proxies. To do so, OMOS automatically detects this case and keeps the small data objects in a queue and piggyback them on an single iframe proxy. This service dramatically improves the event rate.

3.6 MDP Layer

In OMOS, MDP (Mashup Datagram Protocol) is similar to transport layer protocols in TCP/IP (or UDP/TCP). However, note that all of the frame-to-frame communications occurred in OMOS take place in the end user's browser on the user's *local machine*, as OMOS supports client-side mashups. The inter-frame messages are represented by the thick arrows in Figure 1. An MDP communication has three phases: 1) Connection establishment (three-way handshake) 2) Communication (transferring actual data) 3) Disconnection (upon requests of one of the peers, closing the connection and releasing the resources). Figure 5 illustrates these three phases. Note that all mashlet-to-mashlet communications are asynchronous. Applications can communicate at the MDP layer using OMOS socket APIs. The APIs are asynchronous meaning that actions are executed in non-blocking scheme, allowing the main program flow to continue processing. Programs pass callback functions to handle events. Figure 5 shows a usual MDP communication scenario;

The following code illustrates how one can use OMOS socket APIs. For the mashlet at the service provider side:

```
var serverSocket = OpenMashupOS.ServerSocket(1111);
var ssCallback =
{
   onConnectionRequested: function(socket)
   { // define sCallback to handle events
     // including onDataReceived,
     // onTimeout,onError events
     // set callback object for server-side
     // socket endpoint
      socket.setCallback(sCallback);
     var currentTime = new Date();
     //send data to client that is connected
     //to this socket
      socket.send(currentTime.getTime());
   },
   onError: function(exp) {/*handle exception */}
}
serverSocket.accept(ssCallback);
```

For the mashlet at the service consumer side:

```
var sCallback =
{
  onConnected:   function()
                 {alert("Connected to server");},
  onDisconnected: function()
                 {alert("Disconnected.");},
  onDataReceived: function(data)
                  {
                    alert("Server's time is "+data);
                    socket.disconnect();
                  },
  onTimeout:      function()
                  { /*handle timeout   */ },
```

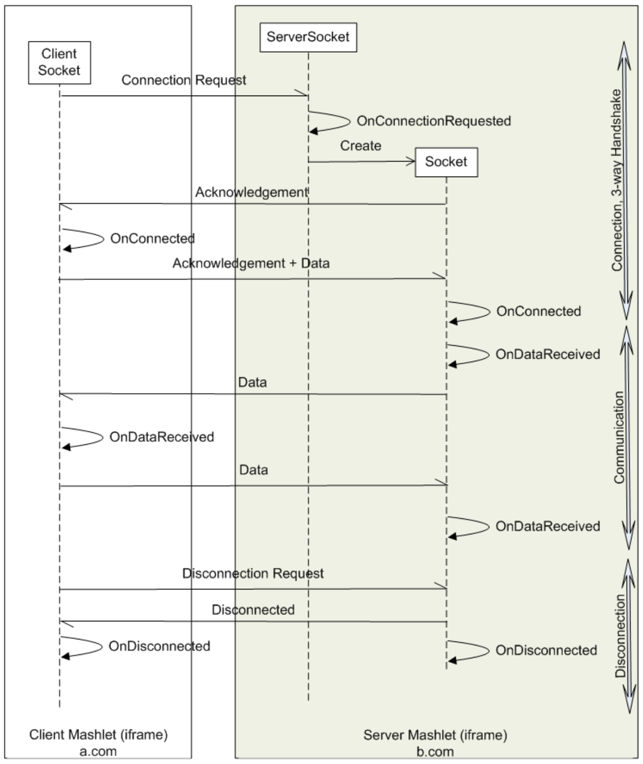

Figure 5. Connection establishment (three-way handshake), Communication, and Disconnection are three phases of a typical MDP communication session.

```
  onError:        function(exp)
                  { /*handle exception */ },
  timeout:        1000
}
var socket =  OpenMashupOS.socket("time.example.com",
                                  1111, sCallback);
```

3.7 MHTTP Layer

OMOS provides the main functionality of the MHTTP layer through the versatile `asyncRequest` method that abstracts the same-domain and cross-domain HTTP calls to servers as well as the mashlet-to-mashlet communication. The latter happens inside the browser on the client's local machine. The implementation of the `asyncRequest` method is built on the existing XMLHttpRequest API in JavaScript. Currently, XMLHttpRequest only handles same domain mashlet-to-server interaction. Our `asyncRequest` realizes cross-domain requests by coupling XMLHttpRequest with our mashlet-to-mashlet communication mechanism (described in previous sections). Thereby, we are able to provide a nice and clean interface for all three types of calls, which are shown in Figure 6. The following code shows how we can use OMOS API to make

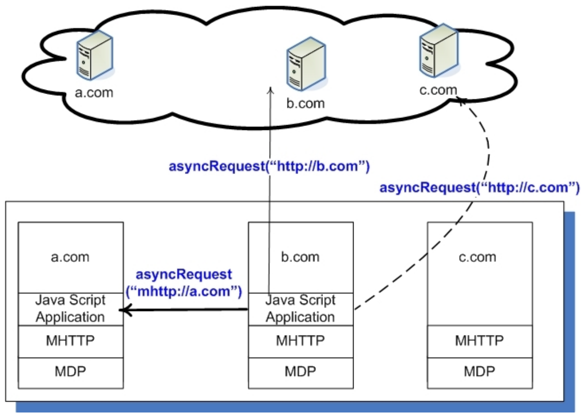

Figure 6. Illustration of the flexibility of asyncRequest method in OMOS that can be used to realize three types of requests from b.com: same-domain mashlet-to-server communication (solid thin line), cross-domain mashlet-to-server communication (dash line to server at c.com), and mashlet-to-mashlet communication (solid thick line to mashlet at a.com).

a MHTTP call:

```
var callback =
{
  onDataReceived: function(response)
                          { /*consume response */ },
  onTimeout:    function()
                          { /*handle timeout   */ },
  onError:       function(exception)
                          { /*handle exception */ },
  timeout:       1000
}
OpenMashupOS.asyncRequest('POST',
                "mhttp:5555//socialnetwork.com/service",
                callback,
                JsonRpcRequest
        );
```

4 Security Analysis

We analysis the security of OMOS from three aspects: data confidentiality, message integrity, and mutual authentication. We describe how frame phishing can be easily prevented in our framework.

Data Confidentiality OMOS satisfies the data confidentiality in inter-mashlet messaging by leveraging the browser's same origin policy and the write-only restriction on the URL field of iframe. The sender passes data through the URL of a proxy iframe from the domain of the intended receiver. As the URL can be read only by proxy's domain, no man-in-the-middle can read the message.

Message Integrity In OMOS, message integrity is realized by utilizing the browser's restriction on partial change of URLs and the shared key between two frames. To modify any data carried on URL, a mashlet needs to know the secret key, otherwise the packet is rejected and dropped at the destination. Thus, an unauthorized mashlet is unable to tamper with inter-frame messages.

Mutual Authentication Our key establishment protocol in Figure 4 guarantees that the mutual authentication between two frames, say frame[1] from `a.com` and frame[2] from `b.com`, is achieved in OMOS. Frame[2]'s origin is successfully authenticated, if and only if it sends back the secret key SK_1 sent by frame[1] through `b.com`'s proxy. Similarly, frame[1] proves its origin by sending SK_2 back to frame[2]. The confidentiality of communication ensures that frame[1] and frame[2] are the only two mashlets that know SK_2.

Detecting Frame Phishing Frame phishing refers to where a malicious frame in a mashup can change which frame is loaded in another part of the mashup [13]. For example, an attacker's frame can change `bank.com`'s frame to point to `attacker.com`, which may mislead the end user into disclosing sensitive information such as password or banking data. The mashlet's parent in OMOS can conveniently detect this type of frame phishing. A regular mashlet has an on-going socket session with its parent for administrative commands. In a normal scenario, disconnection of this session is initiated by mashlet or its parent and this session should be closed before mashlet gets unloaded. Therefore, if `attacker.com` redirects `bank.com` to a malicious frame, since the administrative session is still alive, `bank.com` mashlet, as part of its onunload even handler, will send a phishing attack notification to its parent. Therefore, parent mashlet can take the appropriate action and notify the end user of the threat by prompting an alert window, for example.

Access Control OMOS framework separates communication and access control mechanism. Therefore, different access control techniques can be used to control communications between mashlets. For example, a policy enforcement mashlet can govern communication of different mashlets similar to central even hub in SMash, or in a distributed fashion, each mashlet can control access to its services using a *dynamic whitelisting* technique. Due to the space limit, we do not elaborate on this aspect here in this paper.

5 Experiments

The goal of the experiments is to test the performance of OMOS library in various browsers, in particular, on how fast data can be transferred from one frame to another frame of a different origin. We are mostly interested in testing the communication channel between two frames as it is the basic building block for mashup applications. We ran experiments on a machine with the following configurations. Intel Core 2 CPU, 980 MHz, 1.99 GB RAM, Microsoft Windows XP 2002 SP2, Firefox v2.0.0.14, Internet Explorer v7.0.5730.13, Opera v9.27, and Apple Safari v3.1.1. The values reported are the averaged results over five runs.

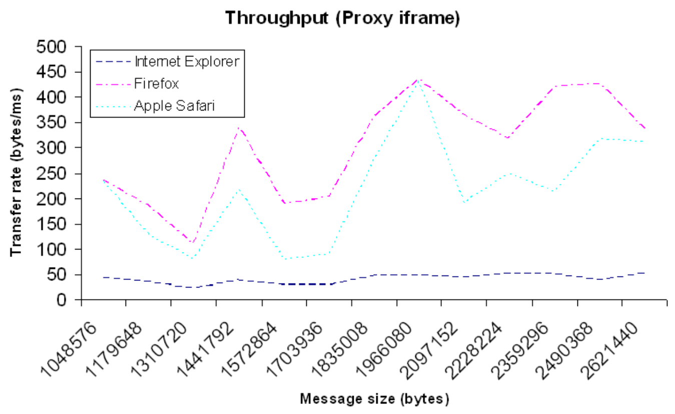

Figure 7. The figure shows the throughput between two mashlets with iframe proxies in FireFox, IE, and Safari. X-axis is the size of MDP packets.

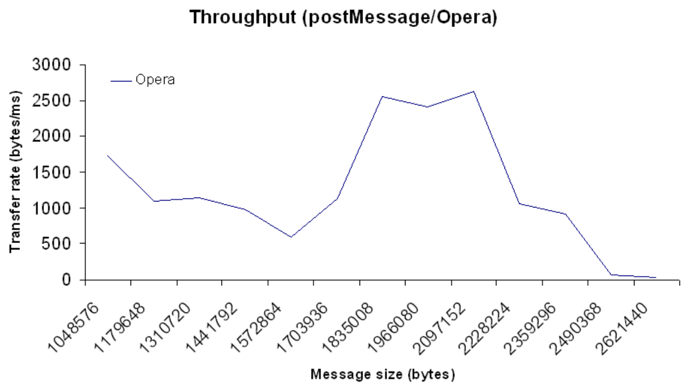

Figure 8. The figure shows the throughput between two mashlets with PostMessage in Opera. X-axis is the size of MDP packets.

Figure 7 shows the throughput as the size of messages increases in FireFox, IE, and Safari. FireFox and Safari have similar performance in terms of throughput as they both can achieve around 430 KB/s of transfer rate. Recall that MDP layer can handle arbitrarily large data objects. The underlying Datalink layer handles the URL limitation by fragmentation and defragmentation. For IE, the throughput is much lower and can achieve the transfer rate of 50 KB/s. The slowdown in IE is due to the URL limit (2KB) imposed by IE, as there is overhead in the Datalink layer to fragment and defragment large messages into small packets that can be fit into 2KB URL. The mashlets communicate through iframe proxies described in our protocol. In general, larger message sizes give higher throughput for all three types of browsers. Opera gives high throughput, due to the native support of inter-frame messaging (i.e., postMessage [3]), it is shown in a separate graph in Figure 8. Figure 8 shows that Opera gives throughput as high as 2500 KB/s with message sizes around 2MB. However, the performance then degrades as the message sizes increase. The transfer rate eventually drops to zero as the message size reaches around 2.6MB. The root cause of this poor performance of Opera with larger message sizes is currently not clear to us. From the throughput results, 2 MB seems to be the optimal message size.

Even though using the larger message sizes (i.e., frame URL) for transferring data leads to higher throughput, we observed that using very large message sizes leads to less responsive user interface and thus affects the surfing experience of the end user. Based on our experiences, the maximum message size should be around 100 KB to ensure responsive browser interface. Therefore, there is a trade-off between performance and usability. IE's URL limit affects the rate of information transferred and significantly slows down the data transfer. In comparison, for all the other three browsers, the frame URL can be very large (> 2MB). OMOS is able to find the suitable size of frame URL automatically.

6 Related Work

The authors of MashupOS recognized that existing browser has a limited all-or-nothing trust model and protection abstractions suitable only for a single-domain system [19]. They proposed new abstractions for the content types and trust relationships in the current browser environments. In MashupOS, new native HTML tags are introduced to HTML page. These tags can be added and removed dynamically using JavaScript, so mashups with dynamic layout are possible. To demonstrate the feasibility, the authors have implemented their abstraction using browser plug-in for IE in such a way that browser at compile time converts them to standard HTML tags and simulates their functionality. The main difference between MashupOS and OMOS is that MashupOS provides a modified browser, whereas we create library supports that applications can use within current browsers. In Subspace [11], JavaScript web services are placed into iframes that are originated from "throw-away" subdomains of mashup integrator. This approach is not flexible in general, as web services need to run under the subdomain of the integrator, and cannot directly perform XMLHtmlRequest calls to their backends.

Keukelaere *et al.* developed SMash that is a secure component communication model for cross-domain mashups called SMash [13]. In SMash, all of the communications are through the mashup integrator, which is also called hub. The hub mediates and coordinates all the communications via tunnel frames among the participating frames. The hub also enforces access policies. It prevents frames from eavesdropping on or tampering the others' communication channels. SMash inter-frame communication is supported through a tunnel frame pointing to the integrator's domain that each frame needs to create in order to communicate with the integrator.

In comparison to SMash where a tunnel frame exists in every mashlet, we create an iframe for every round of communication and send the information encoded in the fragment identifier during an onLoad event. Therefore, unlike SMash, we do not need a polling mechanism and the communication in OMOS is event-driven. Polling creates negative impacts on the performance of single-threaded browser. We support mutual authentication in our inter-mashlet communication that prevents an attacker from frame spoofing. In our OMOS, cross-domain frames can communicate directly without the participation of the mashup integrator. Therefore, the trust assumption put on the mashup integrator can be relaxed.

Recently, a secure postMessage method is proposed by Barth, Jackson, and Mitchell [3]. They have proposed a protocol to fix an authentication vulnerability in several (polling-based) inter-frame communication protocols including SMash, and Windows Live communication protocol [15]. The communication protocol used in OMOS dose not have this issue, as is explained in Section 4.

Cross-site request forgery (XSRF), which is also known as the confused deputy attack against a Web browser [1], is a malicious attack again websites by exploiting browser vulnerabilities. In a XSRF attack, a malicious website can launch an iframe to make requests on behalf of the user to another website with which the user's authenticated session is still valid. For example, the request may be to transfer funds from the user's bank or to change the user's Gmail configuration. A secure browser *OP browser* that prevents and detects XSRF was presented by [9]. Simple alternatives are for websites to set a short expiration period on authenticated sessions, and to educate users to close authenticated sessions upon finishing.

Singh and Lee presented a browser design inspired by μ-kernel based OS [14] that allows flexible and finer customization. The main design difference between Singh-Lee browser and OP browser is that OP browser is process-based whereas Singh-Lee browser is within the same address space that makes it possible for the browser to provide memory isolation for browser components. As with other mashup solutions (SMash and MashupOS), OMOS depends on the security of browser to correctly operate. Therefore, a secure browser such as OP would be complementary to our techniques in realizing web security.

7 Conclusions and Future Work

We presented our design and implementation of a secure and efficient communication framework OMOS for mashup applications. OMOS works in unmodified browsers and ensures the message authentication, integrity, and confidentiality in cross-domain inter-frame communications. We gave a detailed security analysis of our communication mechanism based on iframe proxies. We demonstrated through experiments that OMOS gives high data transfer rates in most types of browsers.

For future work, we would like to design and support more service abstraction in OMOS. We would like to enable identity management in OMOS by supporting secure *single sign-on authentication* for users. Different service provider mashlets require user authentication and authorization. Having a protocol for single sign-on will relieve user from entering username and password multiple times. In this case, user provides his or her identification to a trusted mashlet in the container and then the mashlet provides that information to other web services' mashlets as needed. We will investigate the user-defined delegation problems in mashup environments addressed by Close [4] who proposed a simple Web-Key solution. We also plan to provide efficient services to mashlets similar to the services that operating systems provide to applications such as the management of persistence and resources.

8 Acknowledgements

The first author would like to thank the help of professors at Bahai Institute for Higher Education (BIHE).

References

[1] D. Ahmad. The confused deputy and the domain hijacker. *IEEE Security and Privacy*, 6(1):74–77, January/February 2008.

[2] Asynchronous JavaScript and XML Tutorials. `http://developer.mozilla.org/en/docs/AJAX`.

[3] A. Barth, C. Jackson, and J. C. Mitchell. Securing browser frame communication. In *Proceedings of the 17th USENIX Security Symposium*, 2008.

[4] T. Close. Web-key: Mashing with permission. In *W2SP 2008: Web 2.0 Security and Privacy. Held in conjunction with the 2008 IEEE Symposium on Security and Privacy*, 2008.

[5] CrossFrame, JavaScript Yahoo API.

[6] S. Crites, F. Hsu, and H. Chen. Omash: Enabling secure web mashups via object abstractions. In *15 ACM Conference on Computer and Communicatios Security*, 2008.

[7] DOJO Library. Part 5. `http://dojotoolkit.org/book/dojo-book-0-4/`.

[8] Facebook API. `http://developers.facebook.com/`.

[9] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, May 2008.

[10] C. Jackson and A. Barth. Beware of finer-grained origins. In *W2SP 2008: Web 2.0 Security and Privacy. Held in conjunction with the 2008 IEEE Symposium on Security and Privacy*, 2008.

[11] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proceedings of the 16th International Conference on World Wide Web*, pages 611–620, 2007.

[12] JSON-RPC 1.1 Specification. `http://json-rpc.org/wd/JSON-RPC-1-1-WD-20060807.html`.

[13] F. D. Keukelaere, S. Bhola, M. Steiner, S. Chari, and S. Yoshihama. SMash: Secure component model for cross-domain mashups on unmodified browsers. In *Proceedings of the 17th International Conference on World Wide Web*, 2008.

[14] K. S. W. Lee. On the design of a web browser: Lessons learned from operating systems. In *W2SP 2008: Web 2.0 Security and Privacy. Held in conjunction with the 2008 IEEE Symposium on Security and Privacy*, 2008.

[15] Microsoft Windows Live Contacts. `htp://dev.live.com/mashups/trypresencecontrol/`.

[16] OpenMashupOS Project, `http://OpenMashupOS.com/`.

[17] OpenSocial API. `http://code.google.com/apis/opensocial/`.

[18] Same Origin Policy. `http://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[19] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *ACM Symposium on Operating Systems Principle (SOSP)*, pages 1–16. ACM Press, 2007.