Web Same-Origin-Policy Exploration Lab
(Web Application: Collabtive)

1 Overview

The security model of existing web browsers is based on the same-origin policy, and provides some basic protection features to web applications. The objective of this labs is to help the students get a good understanding of the same-origin policy. The understanding will be a precursor for other web-related labs such as cross-site scripting and cross-site request forgery.

2 Lab Environment

You need to use our provided virtual machine image for this lab. The name of the VM image that supports this lab is called `SEEDUbuntu11.04-Aug-2011`, which is built in August 2011. If you happen to have an older version of our pre-built VM image, you need to download the most recent version, as the older version does not support this lab. Go to our SEED web page (`http://www.cis.syr.edu/~wedu/seed/`) to get the VM image.

2.1 Environment Configuration

In this lab, we need three things, are of which are already installed in the provided VM image: (1) the Firefox web browser, (2) the Apache web server, and (3) the `Collabtive` project management web application. For the browser, we need to use the `LiveHTTPHeaders` extension for Firefox to inspect the HTTP requests and responses. The pre-built `Ubuntu` VM image provided to you has already installed the Firefox web browser with the required extensions.

Starting the Apache Server. The Apache web server is also included in the pre-built `Ubuntu` image. However, the web server is not started by default. You need to first start the web server using the following command:

```
% sudo service apache2 start
```

The `Collabtive` Web Application. We use an open-source web application called `Collabtive` in this lab. `Collabtive` is a web-based project management system. This web application is already set up in the pre-built `Ubuntu` VM image. We have also created several user accounts on the `Collabtive` server. To see all the users' account information, first log in as the admin using the following password; other users' account information can be obtained from the post on the front page.

```
username: admin
password: admin
```

Configuring DNS. We have configured the following URLs needed for this lab. To access the URLs , the Apache server needs to be started first:

URL	Description	Directory
`http://www.soplab.com`		/var/www/SOP/
`http://www.soplabattacker.com`	Attacker	/var/www/SOP/attacker/
`http://www.soplabcollabtive.com`	Collabtive	/var/www/SOP/soplabCollabtive/

The above URLs are only accessible from inside of the virtual machine, because we have modified the `/etc/hosts` file to map the domain name of each URL to the virtual machine's local IP address (`127.0.0.1`). You may map any domain name to a particular IP address using `/etc/hosts`. For example you can map `http://www.example.com` to the local IP address by appending the following entry to `/etc/hosts`:

```
127.0.0.1       www.example.com
```

If your web server and browser are running on two different machines, you need to modify `/etc/hosts` on the browser's machine accordingly to map these domain names to the web server's IP address, not to `127.0.0.1`.

Configuring Apache Server. In the pre-built VM image, we use Apache server to host all the web sites used in the lab. The name-based virtual hosting feature in Apache could be used to host several web sites (or URLs) on the same machine. A configuration file named `default` in the directory "`/etc/apache2/sites-available`" contains the necessary directives for the configuration:

1. The directive "`NameVirtualHost *`" instructs the web server to use all IP addresses in the machine (some machines may have multiple IP addresses).

2. Each web site has a `VirtualHost` block that specifies the URL for the web site and directory in the file system that contains the sources for the web site. For example, to configure a web site with URL `http://www.example1.com` with sources in directory `/var/www/Example_1/`, and to configure a web site with URL `http://www.example2.com` with sources in directory `/var/www/Example_2/`, we use the following blocks:

```
<VirtualHost *>
    ServerName http://www.example1.com
    DocumentRoot /var/www/Example_1/
</VirtualHost>

<VirtualHost *>
    ServerName http://www.example2.com
    DocumentRoot /var/www/Example_2/
</VirtualHost>
```

You may modify the web application by accessing the source in the mentioned directories. For example, with the above configuration, the web application `http://www.example1.com` can be changed by modifying the sources in the directory `/var/www/Example_1/`.

Disabling Cache. The lab tasks require you to make some modifications to the web applications while you are using them. To ensure the web browser always fetches the page from the modified web application and not from the web browsers cache, you can disable the web browser's local cache as follows – Type `about:config` in the address bar and setup the following preferences in the web page you see:

```
browser.cache.memory.enable         /* set to false, default = true */
browser.cache.disk.enable           /* set to false, default = true */
browser.cache.check_doc_frequency   /* 1 = everytime. default = 3 =
                                       as needed */
```

You may re-enable the cache after your lab, so that there is no performance degradation of your browsing activities afterward.

2.2 Note for Instructors

This lab may be executed in a supervised fashion in a lab environment. In such cases, the instructor may provide some background information at the beginning of the lab such as:

1. How to use the Firefox browser and the `LiveHTTPHeaders` Extension.
2. How to use the pre-configured virtual machine.
3. Some background on JavaScript, Document Object Model (DOM), HTML basics, and `XMLHttpRequest`.

3 Background

Web browsers are essentially user agents that interact with web sites/web applications [1] on behalf of their users. Typically users visit a web site using the web browser – web browsers forward HTTP requests to the web site on behalf of their users and in turn display the web page returned by the web site in the response.

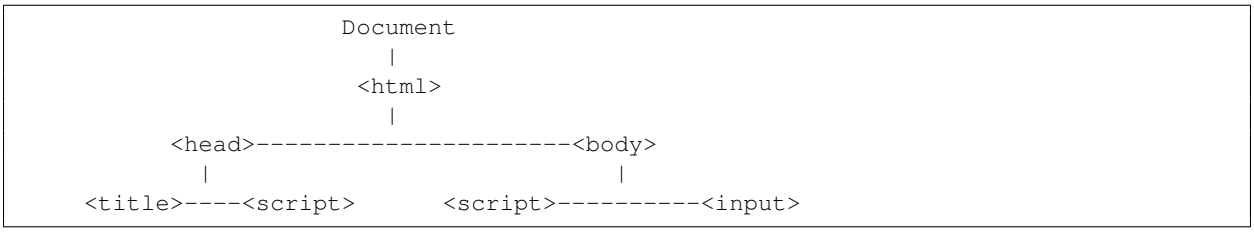
Web browsers use a security model called the same-origin policy (SOP) for enforcing some access restrictions on web applications. The SOP identifies each web site using its origin, which is a unique combination of ⟨protocol, domain, port⟩, and creates a context for each origin. For each origin, the web browser creates a context and stores the resources of the web application from the origin in the context. JavaScript programs from one origin are not allowed to access resources from another origin. Cookies and Document Object Model (DOM) objects are examples of web application resources for which SOP is applied. Furthermore, JavaScript programs may use the `XMLHttpRequest` API to send HTTP requests to web applications. The SOP is also extended to the use of `XMLHttpRequest` API.

First, we will provide some background on cookies, DOM objects, and `XMLHttpRequest` API. Then, we describe the lab tasks that will lead the students to investigate SOP and how it affects the use of cookies, DOM objects, and `XMLHttpRequest` API.

3.1 Document object model (DOM)

Web browsers expose the contents of the web page using the DOM API to JavaScript programs. Figure 1 contains a web page that illustrates the use of DOM API. HTML is a hierarchically structured document. Internally, the DOM object organizes the tags in the web page in the form of a tree. The original structure of the web page in Figure 1 is show in the following:

[1] We will use the terms web sites and web applications interchangeably



There are two functions in the web page namely `appendp` and `gethtmlchildren`. The `appendp` function adds a `h1` heading and a paragraph element to the body of the web page using the DOM API. The function `gethtmlchildren` displays all the tags that are children of the HTML tag.

```
<html>
  <head>
    <title>Self-modifying HTML</title>
    <script>
      function appendp()
      {
           var h1_node = document.createElement("h1");
           h1_node.innerHTML = "Self-modifying HTML Document";
           document.childNodes[0].childNodes[2].appendChild(h1_node);
           var p_node = document.createElement("p");
           p_node.innerHTML = "This web page illustrates how
                               DOM API can be used to modify a web page";
           document.childNodes[0].childNodes[2].appendChild(p_node);
      }

      function gethtmlchildren()
      {
           var entiredoc = document.childNodes[0];
           var docnodes = entiredoc.childNodes;
           for(i=0; i<docnodes.length; i++)
             alert(docnodes[i].nodeName);
      }
    </script>
  </head>
  <body name="bodybody" >
    <script> appendp(); </script>
    <input type="button" value="Display children of HTML tag"
           onclick=gethtmlchildren() >
  </body>
</html>
```

Figure 1: A web page with a JavaScript program that updates the web page dynamically

3.2 Cookies

Cookies are placeholders for server-provided data in the web browser typically used to track sessions. Each cookie is a key-value pair such as `"color=green"` and may have some optional attributes:

- `expires` attribute indicates the cookie's expiration date in the future.
- `max-age` attribute specifies the lifetime of the cookie in seconds.

- `path` attribute indicates the top directory under which the cookie is shared and accessible.
- `domain` attribute indicates the top domain level under which cookie can be accessed cross domain.
- `secure` is a Boolean attribute which enforces that the cookie is transmitted only using HTTPS or another secure protocol.

Web applications can create a cookie in the web browser using the `set-cookie` header in the HTTP response. After cookies are created, web browsers attach the cookies in all the subsequent requests to the web application. Also, JavaScript programs can access, modify, and create cookies. In a JavaScript program, All the cookies in the web application can be referenced using `document.cookie` object.

The HTTP protocol is by its nature stateless. Therefore, web applications use a session-management schemes for associating HTTP requests with a particular user and session. In cookie-based session-management schemes, web applications store the session identifier in a cookie in the web browser. The session cookie is an example of a resource that needs protection to ensure the integrity and correctness of the application.

3.3 XMLHttpRequest

JavaScript programs may use the `XMLHttpRequest` API to send HTTP requests for a target URL. The following is a simple JavaScript program that uses the `XMLHttpRequest` API:

```
<script>
   xhr = new XMLHttpRequest();
   xhr.open(POST,"http://www.soplabcollabtive.com/admin.php?action=addpro",true);
   xhr.send(null);
</script>
```

The above JavaScript program sends a HTTP POST request to a URL using the `open` and `send` methods. The Same-Origin Policy also applies to the target URL used in the `send` methods.

4 Lab Tasks

4.1 Task 1: Understanding DOM and Cookies

The objective of this task is to get familiar with the DOM APIs that can be used for modifying cookies and web pages.

1. Figure 1 illustrates the use of some DOM API. Write a JavaScript function that traverses and displays the entire DOM tree for the web page in Figure 1. The function should show the `h1` heading and paragraph added to the document by the *appendp* function.
2. The `Collabtive` web application uses a cookie-based session management scheme. Identify the name of the session cookie in `Collabtive`. Using the `LiveHTTPHeaders` extension to find out when the web application creates the session cookie in the web browser; please provide a snapshot of the interactions. The `Collabtive` web application can be accessed using the URL `www.soplabcollabtive.com`.
3. Read the source code of `www.soplab.com/cookie.html`, and understand how to store, read and process the cookies. Write your own JavaScript in `cookie.html` to display the number of times that the web page has been visited by the current user.

4.2 Task 2: SOP for DOM and Cookies

The objective of this task is to illustrate how web browsers identify the origin of web applications and how access restrictions are applied on DOM objects and cookies.

To illustrate SOP for DOM and cookies, we use a web page, located at `www.soplab.com/index.html`. The web page displays two web pages inside its frames:

```
<frameset rows="*,75">
  <frame src="about:blank" name="main">
  <frame src="navigation.html">
</frameset>
```

The first frame displays a web page located at `www.soplab.com/navigation.html` and asks the user to provide the URL for another web page to be displayed in the next frame. When the user provides the URL, a JavaScript program in the first frame displays the requested web page in the second frame. Furthermore, `navigation.html` has two JavaScript programs that read the source code and cookies of the web page in the second frame. JavaScript programs in `navigation.html` can reference the DOM object and the cookies of the web page in the second frame using `parent.main.document` and `parent.main.document.cookie` respectively. Essentially, we have one web page that is accessing the resources of another web page. Recall that the SOP restricts JavaScript programs from one origin from accessing resources in another origin. We will use this web page in the forthcoming tasks to understand the SOP-based access restrictions on cookies and DOM objects.

1. Provide the following URLs to the web page in the first frame and report whether you are able to access its cookies and DOM objects from the first frame.
   - `www.soplab.com/index.html`
   - `www.soplab.com/navigation.html`
2. Try to use some cross-domain URL such as `www.google.com` in the URL bar of the first web page and report whether you are able to access its cookies and DOM objects.
3. The web server is listening on two ports - 80 and 8080. Provide `http://www.soplab.com:8080/navigation.html` to the first frame, and report whether you are able to read the DOM object and cookies for the web page in the second frame.
4. Not only are the cookie and contents of the frame under the restriction of SOP, several other objects are also restricted, such as the `History` object and the URL of the frame. Test them on `www.soplab.com/index.html`.

4.3 Task 3: SOP for `XMLHttpRequest`

We have seen a simple example that uses the `XMLHttpRequest` API. A slightly more complex example is contained in `www.soplab.com/navigation.html`. Once you have familiarized yourself with the `XMLHttpRequest` API, you can do the following:

1. Understand the JavaScript program in `navigation.html`, and then verify whether the SOP is also extended to the target URL of HTTP requests that you can create using `XMLHttpRequest` API. Report your observations in the report.
2. What are the dangers of not extending the SOP to the HTTP requests created using `XMLHttpRequest` API. For full credit, you should describe some possible attacks.

Task 4: Exceptions from SOP

There are some exceptions to SOP. In this task you will explore such exceptions.

- Some HTML tags can also trigger a HTTP request within a web page. For example, the `img` tag in a HTML page triggers a HTTP GET request. The question is whether SOP is applied here to restrict the targets of the HTTP request. Please investigate the following HTML tags: `frame`, `iframe`, `img`, and `a`. Verify your hypothesis using experiments, and report your observations. You can craft a web page in `www.soplabattacker.com` to make requests to `www.soplab.com`.

5 Submission

You need to submit a detailed lab report to describe what you have done and what you have observed. Please provide details using `LiveHTTPHeaders`, `Wireshark`, and/or screenshots. You also need to provide explanation to the observations that are interesting or surprising.