Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permited without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Web App Penetration Testing and Ethical Hacking (Security 542)"
at http://www.giac.org/registration/gwapt

The Risks of Client-Side Data Storage

GIAC (GWAPT) Gold Certification

Author: Edwin Tump, edwin.tump@govcert.nl
Advisor: Tim Proffitt

Accepted: ----

Abstract

Web applications can use a plethora of mechanisms to store information on the PC's of their visitors. This information changed from simple ID's in cookies in the past to complete client-side databases with possibly confidential information in modern web applications. The growth in local data storage introduces new risks for both web applications and end users. Because web applications more and more depend on client-side data and have no control over what users do with it, they run the risk of using corrupted data in their application logic. As JavaScript allows access to all of this client-side data, Cross-Site Scripting (XSS) vulnerabilities become more and more serious, while the seriousness of this type of vulnerability is still often underestimated by developers. The introduction of more data and more technologies on web clients also brings traditional server-side vulnerabilities like XSS and SQL injection to clients, resulting in web applications suffering from client-side SQL injection (csSQLi) and client-side XSS (csXSS). This paper describes the different technologies available for client-side data storage, the risks associated with it and the measures that web developers should consider when using client-side data.

© 2011 SANS Institute, Author retains full rights.

© 2011 The SANS Institute Author retains full rights.

1. Introduction

Ever since the introduction of cookies as the HTTP state management mechanism, websites store data on the systems of their end users. The original idea behind cookies was that web applications would now be able to relate HTTP requests to previous requests. By storing a unique session identifier on both the client (in the form of a small text file, the "cookie") and the server, the stateless HTTP protocol suddenly became stateful. Cookie usage has changed over time and now web applications not only use this mechanism for session fixation but also to e.g. track users, create web applications with offline capabilities and speed up the performance of web applications by reducing server load and limiting the data that must be exchanged between client and server.

Not only cookie-usage itself has changed, also alternate technologies were introduced to store information on the systems of end users. These new technologies often make use of alternative ways to store data (e.g. in a local database) and typically offer much more storage capacity than cookies. It has made client-side data storage a quite popular part of modern web applications and this in turn lead to a changed risk landscape.

This paper will detail these risks associated with client-side data storage. It has a strong focus on the risks from the viewpoint of web applications. Risks that are mostly a concern to the end users of a web application are not taken into account. Typical risks that are mostly a concern to end users include privacy issues like the ability to track the browsing behaviors of users over different websites. Although very interesting, these kind of issues are out-of-scope for this paper.

Warning
This paper contains code examples with the sole purpose to illustrate the workings of client-side data storage. These examples are not complete nor do they contain appropriate checks and validations. Therefore, the examples used should never be used in any production environment.

Edwin Tump, edwin.tump@govcert.nl

© 2011 The SANS Institute Author retains full rights.

2. Client-side data storage

Before jumping into the details of client-side data storage risks, this paper will first define what client-side data storage actually is and which technologies are available to implement it. It will then give an overview of the risks associated with its use and end with an outline of the measures website owners should consider in order to avoid the described risks from happening.

2.1. Client-side data characteristics

All client-side data storage technologies, regardless the implementation, share some characteristics that must be considered first when defining the risks of it. First of all it's important to note that client-side data storage is initiated by the web application and that the data is mostly user-specific. This means that e.g. caching mechanisms implemented by the browser, although they do store data on the client, are not treated as client-side data storage technologies in this paper. With caching, the storage is initiated by the browser instead of the web application and most of the time it concerns generic data (e.g. image files) instead of user-specific data.

Of course, every client-side data technology supports the storage of data on the system of the end user. The maximum storage size differs between the technologies. Cookies e.g. allow for the storage of 4 kilobytes of data per cookie, with a maximum of 20 cookies per origin (Cristol, 1997). This 80 kilobytes storage-limitation for an origin is almost negligible when compared to newer technologies like HTML 5 Web SQL Databases that support 5 megabytes (the equivalent of 1.280 cookies) of storage per origin by default (Hickson, 2010).

The maximum storage duration can often not be specified. Cookies are an exception to this rule as it allows for an expiry to be set on the data, so that the cookie will be automatically discarded by the browser once the data in the cookie has expired. Unfortunately, most of the other storage technologies discussed in this paper do not support such an expiry mechanism so that data will remain on the system of the user as long as the web application, the browser or the user do not delete it.

© 2011 The SANS Institute Author retains full rights.

To prevent web applications from reading each others data, a mechanism known as the same origin policy applies to all of the storage technologies. By implementing the same origin policy, browsers check and record the origin of all the data they store based on the combination of at least the hostname of the web application (e.g. www.microsoft.com), the port number on which the web application runs (e.g. 80) and the protocol or scheme through which the data was delivered (typically http or https). When a web application wants to access some locally stored data, the browser will check the current origin and the origin of the data and only allow access if these match. Some storage technologies allow for an even stricter origin specification by also allowing a path in the origin (e.g. "/secure").

Different client-side data storage technologies use different storage mechanisms to store the data in. This can be e.g. a simple clear text based file, an XML formatted file or a binary database file. These files are created within the profile of the logged-in user so that (theoretically) only this user has access to this information.

The way locally stored data is used, depends on the specific web application. Most of the time, the data is used as input by the web application running on the server-side. Sometimes information stays local and is loaded by client-side scripts (e.g. Javascript) to create dynamic web pages based on the locally stored data.

2.2. Client-side data storage definition

Based on the characteristics of client-side data storage that were set out in the previous paragraph, the following definition of client-side data storage will be used throughout this paper:

Definition
Client-side data storage is the storage of mostly user-specific data on the system of a web user whereby this storage is initiated by a web application and executed and controlled by the browser or a browser plug-in.

© 2011 The SANS Institute Author retains full rights.

2.3. Well-known client-side data storage technologies

Web developers have different technologies at their disposal to store information on the systems of web users. Some of these technologies are supported by default by the browser, other technologies are only available after the user installs an additional plug-in in his browser. For the selection of plug-ins for this paper, statistics from StatOwl[1] are used. Of course, only plug-ins that offer storage capabilities are selected. Table 1 lists the most popular plug-ins according to StatOwl (based on 2010 usage statistics) and an indication whether or not this plug-in is included in this paper.

Plug-in	Support (2010)	In this paper?
Adobe Flash	96.43%	Yes
Oracle Java	79.33%	Yes
Windows Media Player	67.43%	No *(media player/no offline storage capabilities)*
Apple Quicktime	60.63%	No *(media player/no offline storage capabilities)*
Microsoft Silverlight	52.65%	Yes
Adobe Shockwave	30.67%	No
Google Gears	6.13%	Yes

Table 1: browser plug-ins

Table 1 shows that Adobe Flash is by far the most popular browser plug-in today as nearly every user on the web (more than 96 users on every 100 users), have this plug-in installed. Next to Adobe Flash, this paper will also discuss Oracle Java, Microsoft Silverlight and Google Gears.

Of course, browsers also support some storage mechanisms out-of-the-box. A well-known example of this is the cookie-mechanism supported by every browser. With the introduction of HTML 5, newer storage mechanisms as Web SQL Databases, Web Storage and IndexedDB are also implemented in some browsers (Pilgrim, 2011). These technologies will also be addressed in this paper. Each technology will be described in view of the client-side data storage characteristics outlined in §2.1.

[1] http://www.statowl.com/

© 2011 The SANS Institute Author retains full rights.

2.3.1. Cookies

The cookie mechanism was introduced in the 1990s as one of the first (if not *the* first) client-side data storage technology to overcome the stateless nature of HTTP and is described in RFC 2109 entitled 'HTTP State Management Mechanism' (Cristol, 1997). By using two HTTP headers ('Cookie' and 'Set-Cookie'), a website can store a unique session identifier on the system of the user '*to create stateful sessions with HTTP requests and responses*'. The 'Set-Cookie' response header is sent by the webserver to the client to initiate storage of information by the browser of the client. The 'Cookie' request header is then automatically inserted into requests to the website by the browser, if matching information for this website is found locally based on the same origin policy. Cookies are still supported by all browsers and used by a lot of websites. The RFC states that browsers should limit the number of cookies per unique host or domain name to 20 with a maximum of 4,096 bytes per cookie. This means a website could potentially store 80 KB (20 x 4,096 bytes) of information on the system of a user. A cookie typically looks like this:

Example 1: cookie example

```
Set-Cookie: name=value; domain=.domain.com; expires= Sun, 15-Nov-2012
14:50:38 GMT; path=/secure/>
```

The cookie must include a name and a value to be stored on the client. The other cookie attributes shown in the example above (domain, expires and path) are optional. The domain attribute allows the website to specify for which hosts and domains the cookie is valid. In the example shown above (example 1), the cookie is valid for all websites configured under the .domain.com domain, which includes www.domain.com, a.domain.com, b.domain.com, etc. The domain attribute thus plays an important role in the implementation of the same origin policy for cookies. Browsers enforce some restrictions to the domain specified. The domain cannot be different from the source domain that sets the cookie and browsers do not allow setting cookies on just a TLD like

© 2011 The SANS Institute Author retains full rights.

'.nl' or '.com' (Boneh, 2009)[2]. 'Expires' indicates when the information in the cookie will expire after which the browser should automatically delete the data. The path attribute allows for further narrowing the origin that is allowed to read and write the data.

Example 2 shows how easy it is to set and retrieve cookie values through JavaScript. The `setCookie()` function sets the value of the cookie parameter_name to something the developer specifies and the `getCookie()` function simply retrieves the current cookie string for this site. The cookie set, expires after 36.000.000 * 24 milliseconds, which is equal to 24 hours or 1 day.

Example 2: cookie usage

```
<script>
  function setCookie (strCookieValue) {
    var today  = new Date();
    var expire = new Date();
    expire.setTime(today.getTime() + 3600000 * 24);
    document.cookie = "parameter_name='" + strCookieValue + "';
      expires=" + expire.toGMTString();
  }

  function getCookie () {
    return document.cookie();
  }
</script>
```

2.3.2. IE UserData

From Internet Explorer version 5, Microsoft supports a dynamic HTML (DHTML) behavior called UserData to store information on a system. According to Microsoft, dynamic DHTML behaviors are '*components that encapsulate specific functionality or behavior on a page*' (Microsoft). Calling the IE UserData behavior,

[2] Browser developers have been struggling with this for a long time because this rule should also apply to generic subdomains under the TLD like .co.uk and .org.au. An interesting discussion on how to solve this in Firefox 1 can be found here: https://bugs.launchpad.net/ubuntu/+source/firefox/+bug/44062

© 2011 The SANS Institute Author retains full rights.

results in Internet Explorer storing data in an XML file on the users' system. Storage is typically limited to 128 KB per document and 1024 KB per domain[3] for regular websites.

Example 3 shows how userData can be used within a web page. The example shows a function named `setIEUserData()` that will set the value of the parameter `parameter_name` to something specified by the developer.

Example 3: IE userData usage

```
(…)
<style>.userData {behavior:url(#default#userdata);}</style>
(…)
<form name="oPersistForm">
  <input type="userData" type="hidden" id="oPersistInput">
</form>
(…)
<script>
  function setIEUserData (strUserData) {
    oPersist = oPersistForm.oPersistInput;
    oPersist.load ("oXMLBranch");
    oPersist.setAttribute ("parameter_name", strUserData);
    oPersist.save ("oXMLBranch");
  }
</script>
```

2.3.3. Adobe Flash

Plug-ins allow for extended functionalities in a browser like video and audio support and other multimedia applications. Not only do these plug-ins extend the functionalities of the browser, they also introduce new storage mechanisms on systems. Adobe Flash, Oracle Java, Microsoft Silverlight and Google Gears are some of the most popular plug-ins that come with storage capabilities.

Adobe Flash supports the concept of Local Shared Objects (LSO) (Adobe). An LSO is often referred to as a "Flash cookie" (EPIC, 2005), indicating that sites you visit

[3] Storage limits differ based on the security zone the specific website is in. Intranet websites can e.g. store up to 10.240 KB of information, whereas websites in the 'Restricted' zone can only store 640 KB.

© 2011 The SANS Institute Author retains full rights.

can create small data files on your computer storing all kinds of information, just like with traditional cookies. By default, Flash allows for the storage of 100 KB of data per domain. However, websites can obtain more storage space after the user approves this. The information is stored in binary LSO files that can be identified by the .sol extension they use.

Flash implements the same-origin policy so that by default only sites that stored data on a computer can access this data. However, by specifying a cross-domain policy file (`crossdomain.xml`), website owners can make the data they store on a computer also available to other source domains (Adobe, 2010).

It's not possible to set expiration on data that's stored in an LSO. This means that information contained in an LSO will stay on the users' computer as long as the website and the end user do not delete it.

A final note about Adobe Flash is that it allows for cross-browser storage. Unlike other storage mechanisms, Flash data is stored in the general profile of the user and not in the browser-part of the profile. This means that the Flash plug-ins in all the browsers installed on the computer, make use of the same Flash store and thus can share data. Example 4 shows a piece of ActionScript – the language used to create Flash applications - that's used to store a price in a LSO.

Example 4: ActionScript LSO usage

```
var totalprice:SharedObject = SharedObject.getLocal ("totalprice");
if (counter.data.value == undefined) {
  totalprice.data.price = 6;
} else {
  totalprice.data.price = totalprice.data.price + 6;
}
counter.flush();
```

2.3.4. Oracle Java

Oracle Java (formerly Sun Java) is a popular programming language that not only supports stand-alone applications on the system of a user but also web applications

© 2011 The SANS Institute Author retains full rights.

through the users' web browser. In case a Java application is started through the browser, this application is called a Java applet.

In theory, the Java Virtual Machine (JVM) – the component responsible for executing Java bytecode – has access to the system of the user to read or write files or to create databases. This way, a Java applet can theoretically store information on the system of a user through a website. However, the default Java system policy enabled on most systems doesn't allow an applet access to the local file system. In order for an applet to read or write information it needs special permissions like `FilePermission` (standard I/O) or `NetPermission` (access to cookie information) (Austin, 2000). Example 5 shows a piece of Java-code, taken from the Oracle-website[4] that will create a file on the system of the user. This example also illustrates that an exception can occur if the applet does not have sufficient permission to access the file system.

Example 5: creating a file from Java

```
Path file = ...;
try {
  file.createFile(); //Create the empty file with default permissions
} catch (FileAlreadyExists x) {
  System.err.format("file named %s already exists%n", file);
} catch (IOException x) {
  //Some other sort of failure, such as permissions.
  System.err.format("createFile error: %s%n", x);
}
```

Permissions for applets are stored in a Java policy file (e.g. `C:\Program Files\Java\jre6\lib\security\java.policy`) and can be changed by using the Java Policy Tool (Oracle, 1995) from Oracle.

[4] http://download.oracle.com/javase/tutorial/essential/io/file.html

© 2011 The SANS Institute Author retains full rights.

2.3.5. Microsoft Silverlight

Silverlight was introduced by Microsoft in 2007. It allows for '*creating engaging, interactive applications*' (Microsoft) on the web and is a competitor to Flash. Just as with Flash, Silverlight allows for the storage of data through these applications. Data is stored in so-called Isolated Storage (IS) (Microsoft, 2009) which is basically a virtual file system mapped to a directory on the real file system of the user. It derived from the same mechanism in Microsoft .NET. Because a website has this virtual file system at its disposal, it can create, read, write, delete, and enumerate files and directories inside the virtual file system, just as with a real file system.

Silverlight distinguishes between application stores and site stores. An application store is mapped to a particular Silverlight application. Access to the application store is granted based on the application identity, which is basically the full URL of the Silverlight application (e.g. http://www.example.com/silverlightapp.xap). A site store is not only accessible to the current Silverlight application, but also to other applications running in the same site. Silverlight employs the well-known same-origin policy based on scheme, hostname and port number to control access to site stores. Through the use of Silverlight policy files (`clientaccesspolicy.xml`) and even Flash policy files (`crossdomain.xml`), developers can influence the access to data and allow certain other websites to access the data they store on a client (Microsoft, 2010).

Silverlight uses quota to limit the amount of information that can be stored. By default, websites can store up to 1 MB of data per domain. But just as with Flash, users can decide to increase the amount of data they allow, if needed.

Example 6, written by Jeremy Likness (Likness, 2009), shows an example of how it's possible to write arbitrary data to a file in Isolated Storage.

© 2011 The SANS Institute Author retains full rights.

Example 6: writing to a file from Silverlight

```
private static void _SaveToDisk(byte() buffer, string fileName)
{
   using (IsolatedStorageFile iso =
      IsolatedStorageFile.GetUserStoreForApplication())
   {
      using (
         IsolatedStorageFileStream stream = new
            IsolatedStorageFileStream(fileName, FileMode.CreateNew,
               iso))
      {
         stream.Write(buffer, 0, buffer.Length);
      }
   }
}
```

2.3.6. Google Gears

Google introduced Gears in May 2007 to – amongst other reasons – enable offline web applications (Gears Team, 2010). The idea is that you can still use your favorite web application, even if you don't have an internet connection. One of the issues with offline web applications is that you might need a lot of data on the client system that's normally only available online. To support offline storage of data, Gears incorporates client-side database support based on SQLite (Gears Team). From Javascript, web applications can store information in a SQLite database on the client, provided the user installed the Gears plug-in for their browser or makes use of the Google-browser Chrome. Information in the database can be queriedbased on SQL statements. Example 7 shows how, from JavaScript, a local Google Gears database can be opened and queried for specific products.

Web applications can store a lot of information in the Gears enabled client-side database. An example use of this functionality is built into Gmail, where you have the option to locally store your e-mails. When enabled, Gmail will automatically synchronize your online mails with your offline database. This way, users can still read their e-mail,

© 2011 The SANS Institute Author retains full rights.

even if they don't have an active internet connection. The amount of information that can be stored in the Gears database doesn't seem to be limited by the software.

Google Gears employs an origin model based on the protocol/host/port tuple (Gears Team). By using so called cross-origin workers it's possible to '*make a request to a cross-origin server, even though the browser usually restricts this*' (Gears Team).

In February 2010, Google announced that they shifted their effort '*towards bringing all of the Gears capabilities into web standards like HTML5*'. Although Gears was still supported after then, this support was '*necessarily constrained in scope*' (Fette, 2010).

Example 7: reading from a Google Gears database

```
var db_gears = google.gears.factory.create('beta.database');
db_gears.open('products');
var rs = db_gears.execute ("SELECT * FROM books WHERE title like '?'",
  title_search);
while (rs.isValidRow()) {
  book_title = rs.field(0);
}
rs.close();
```

2.3.7. HTML 5 Storage Technologies

The latest additions to client-side data storage include new technologies around HTML 5. HTML 5 is the successor to HTML 4.01 that became a W3C recommendation in 1999. Although not part of the base HTML 5 standard, three data storage mechanisms are closely related to it: Web SQL Databases, Web Storage and Indexed Database API.

The Web SQL Database standard (Hickson, 2010) very much follows the storage mechanism of Google Gears as can be seen from example 8 on the next page. It uses a client-side database that can be queried through SQL. Information stored in a Web SQL Database is protected based on the same origin policy and cannot exceed 5 MB. All

© 2011 The SANS Institute Author retains full rights.

current implementations of Web SQL Databases in browsers are based on SQLite as the database. Because of this, the specification has reached an impasse and, according to the standard, '*the Web Applications Working Group does not intend to maintain it further*' (Hickson, 2010).

Example 8: reading from an HTML 5 Web SQL Database

```
var db = openDatabase ('mydb', '1.0', 'Web SQL Database', 2097152);
db.transaction (
  function (tx) {
    tx.executeSql ('SELECT * FROM books WHERE title like '?'",
      (title_search));
    function (tx, results) {
      (…)
    }
  }
);
```

Web Storage is another HTML 5 related standard (Hickson, 2011) and it's already implemented by most modern browsers. It allows for the storage of name/value pairs, just as with traditional cookies. It also allows for the storage of 5 MB of data whereby the same origin policy is used to protect this data. Web Storage distinguishes between local storage and session storage. Data stored in session storage is only available to the current session and the current browser window or browser tab. If you e.g. open the same website in different browser windows, session information in one window will not be available in another window. Local storage is available to all the windows opened (all sessions) by the user. Accessing web storage is very easy as illustrated in the example below where JavaScript is used to retrieve a stored value ('session_id') from local storage.

© 2011 The SANS Institute Author retains full rights.

Example 9: HTML 5 Web Storage

```
localStorage.getItem('session_id');
```

The final HTML 5 related storage standard is the Indexed Database API standard, or IndexedDB in short (Mehta, 2010). Just as with Web SQL Databases, it uses a database to store its information in. This allows for storage of significant amounts of data. However, unlike Web SQL Databases, IndexedDB does not use queries (SQL) to access the database. Instead, it makes use of keys and indexes to store and retrieve key/value pairs. As it uses databases and key/value pairs, IndexedDB can be seen as a compromise between Web SQL Databases and Web Storage. Data is protected through the use of the same origin policy. The specification does not yet outline any limitations on the size of the information store. Support for this standard by browsers is still very limited.

Example 10: HTML 5 IndexedDB

```
var db = open('books', 'Book store', false);
var index = db.openIndex ('BookAuthor');
var matching = index.get('fred');
if (matching) {
  alert (matching.isbn + '|' + matching.name);
}
```

(example taken from http://www.w3.org/TR/IndexedDB/)

Table 1 (next page) contains an overview of the technologies discussed and the main characteristics of these technologies.

© 2011 The SANS Institute Author retains full rights.

Technology	Implementation	Max Size/domain
Cookie	Brower supported (all browsers). Mostly cleartext files with the cookie-value.	20 cookies per origin, 4.096 per cookie (80 KB in total per origin)[5]
IE UserData	Browser supported (Internet Explorer only). Information is stored in XML-files.	640 KB ('Restricted') – 10.240 KB ('Intranet')
Adobe Flash	Browser plug-in. Information is stored in binary Local Shared Object files (.sol files).	100 KB
Oracle Java	Browser plug-in. Storage functionality is disabled by policy by default.	Unlimited
Microsoft Silverlight	Browser plug-in. Information is stored in 'Isolated Storage' which is a virtual file system mapped to the users' system.	1 MB
Google Gears	Browser plug-in. Storage in client-side database based on SQLite. Access to the database through queries (SQL).	Unlimited
Web SQL Databases	Browser supported (selected browsers only). Storage in client-side database. All current implementations are based on SQLite. Access to the database through queries (SQL).	5 MB
Web Storage	Default (selected browsers only). Use of name/value pairs to store and retrieve information.	5 MB
Indexed Database API	Browser supported (selected browsers only). Records (key + value) are stored in a database and accessed through keys and indexes.	Undefined

Table 2: storage technologies

Browsers do not support all of the technologies described in table 2. Table 3 lists the most popular browsers and indicates the technologies supported by these browsers.

[5] Although this is defined in the RFC, some browsers deviate from this standard. Internet Explorer 8 e.g. supports 50 cookies per origin with a maximum of 10 KB per cookie (see http://msdn.microsoft.com/en-us/library/cc197062%28VS.85%29.aspx)

© 2011 The SANS Institute Author retains full rights.

Please note that the overview is based on browsers running on Microsoft Windows XP SP3.

Technology	Apple Safari 5.0.1	Google Chrome 5.0	Microsoft IE 8.0/9.0	Mozilla Firefox 3.6.16/4.0	Opera 10.60
Cookie	Yes	Yes	Yes	Yes	Yes
IE UserData	No	No	Yes	No	No
Adobe Flash	Yes	Yes	Yes	Yes	Yes
Oracle Java	Yes	Yes	Yes	Yes	Yes
Microsoft Silverlight	Yes	Yes	Yes	Yes	Yes
Google Gears	Yes	Yes	Yes	Yes	No
Web SQL Databases	Yes	Yes	No	No	Yes
Web Storage	Yes	Yes	Yes	Yes	Yes
Indexed Database API	No	No	No	No / Yes	No

Table 3: storage technology support by browsers (Yes = supported)

3. Client-side data storage risks

The use of client-side data storage by a web application, introduces risks. This chapter describes the risks associated with client-side data storage. Each risk is illustrated with an example of how this risk can be abused by miscreants.

3.1. Client-side Cross-Site Scripting (csXSS)

Cross-Site Scripting (XSS) is a very prevalent vulnerability in modern web applications. According to the Open Web Application Security Project (OWASP), XSS is '*a type of injection problem, in which malicious scripts are injected into the otherwise benign and trusted web sites'* (KirstenS, 2010). This means that an attacker can execute scripts within the context of the website under attack. Different types of XSS exist, but they all have the same net result: they allow for the execution of malicious JavaScript in the browser of the user. You could argue that this is nothing special, as every website you

© 2011 The SANS Institute Author retains full rights.

visit is able to execute scripts on your PC. However, with XSS, the script is executed within the context of the website attacked, which means the attacker can gain access to resources (e.g. information) which are normally not accessible.

The traditional form of XSS is reflected XSS whereby the malicious script is sent to the web server by the user and this script is then reflected back to the user. The reason the user is sending this malicious script is because this user was offered a malicious link on which he clicked (e.g. a malicious link in an e-mail) or was redirected to (e.g. via a malicious or infected website). Even more powerful than reflected XSS is stored XSS. With stored XSS, an attacker succeeds in injecting a malicious script in a store used by the website. This can be a database, a file or any other storage mechanism on the server. If this store is used to display information to the user, the malicious injected script will be presented to each and every user visiting the website.

Reflected XSS and stored XSS depend on functionalities on the server: the server must reflect or store malicious scripts. There's another form of XSS that stays completely local and makes use of client-side storage capabilities: client-side XSS or csXSS in short (Sutton, 2009). The idea is that an attacker injects a malicious script in a local store on the client. Every time the webpage uses this locally stored information, the client will attack itself with the XSS payload. Figure 1 illustrates the different types of XSS described in this paper.

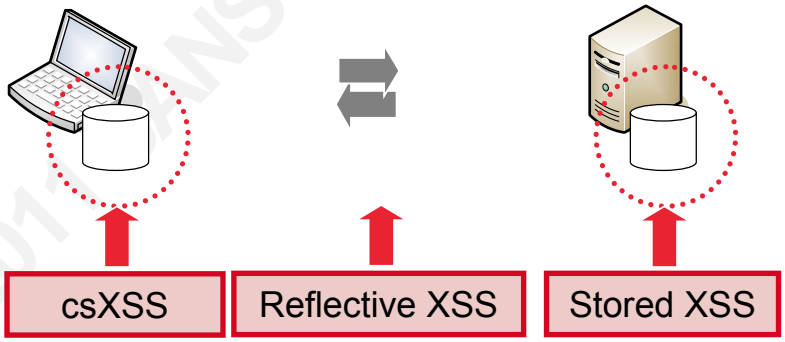

Figure 1: different types of XSS

Assume a website stores a list of all search terms the user searched for in a local Web SQL Database. Each time the user enters a search query, the website will call the

© 2011 The SANS Institute Author retains full rights.

`addToHistory()` function to save the query locally and will then execute the search on the server. The `addToHistory()` function is displayed in example 11.

Example 11: save search term in local database

```
<script>
  function addToHistory (strSearchTerm) {
    if (window.openDatabase) {
      var db = openDatabase ("searchterms", "1.0", "Web SQL Database",
        2 * 1024 * 1024);
      db.transaction (
        function (tx) {
          tx.executeSql ("INSERT INTO search_terms (term) VALUES (?);",
            (strTerm));
        }
      );
    }
  }
</script>
```

The example shows that search terms are stored in a table named `search_terms` and the column name is `term`. Queries are sent to the webserver through a specific queryparameter named `term`. If a user searches for the term 'mysearch' then the resulting URL with the query will be '`/search.pl?`**`term=mysearch`**'. A miscreant finds out that this website is vulnerable for csXSS. To exploit this, the attacker tries to convince the user to open the URL `/search.pl?term=`**`%3Cscript+src%3D%E2%80%9Dhttp%3A%2F%2Fwww.attacker.com%2Fattack.js%E2%80%9D%3E%3C%2Fscript%3E`**'. Although the query may look quite cryptic, it's just a URL-encoded version of the string '`<script src="http://www.attacker.com/attack.js"></script>`'. Now, when the user opens the specific link, the scripts will automatically be saved as a search term to the local database. As soon as the user now opens his search history, he will be automatically offered the malicious script which will result in his search history being exposed to the attacker.

Edwin Tump, edwin.tump@govcert.nl

© 2011 The SANS Institute Author retains full rights.

3.2. Client-side SQL injection (csSQLi)

SQL injection is defined as the act of attacking databases by injecting SQL commands '*into data-plane input in order to effect the execution of predefined SQL commands'* (Nsrav, 2010). This means that SQL injection changes the syntax of the SQL statement in order to influence the effect of the original statement. Historically, SQL injection was only possible on databases running on the server-side of web applications. The reason for this is that databases were only placed on the server and controlled through server-side queries. The introduction of database-support on clients (Google Gears and HTML 5 Web SQL Databases), also introduces the possibility of SQL injection on the client-side: client-side SQL injection or csSQLi in short (Trivero, 2008).

To interact with a local database, websites make use of Javascript-calls. To select a list of records from an HTML 5 Web SQL Database, the developer should issue a Javascript-call like below:

```
t.executeSql('SELECT description FROM actions WHERE id=2')
```

Just as with traditional server-side scripting languages, it's possible to process user input in a query in two different ways: via dynamic strings or via placeholders. The use of placeholders is preferred as it prevents users from changing the syntax of the query through malicious input. Placeholders can be recognized by the question mark in the query, e.g.:

```
t.executeSql('SELECT description FROM actions WHERE id=?', (id))
```

In the example above, the query is static and only the ID that's searched for can be influenced by the user. Sometimes, developers choose to use queries based on dynamic strings instead of placeholders, which can lead to security problems, e.g:

```
t.executeSql('SELECT description FROM actions WHERE id=' + id)
```

By manipulating the ID in the example above, the end user (or an attacker) can change the syntax of the query. The consequences of this manipulation by an attacker depend on the application logic behind the query and the type of query used. However, most of the time, the usefulness of csSQLi is quite limited for an attacker as it does not

© 2011 The SANS Institute Author retains full rights.

allow for the retrieval of information. Altering a query will result in different results echoed back to the user, but not the attacker.

A popular mechanism that's often used in conjunction with SQL injection is a mechanism called stacked queries. A stacked query allows an attacker to execute his own query, fully irrespective of the original query the application executes. Stacked queries are added to an original query through the use of a semicolon. Below is an example of a stacked query that can be injected because of incomplete user input checking:

```
SELECT description FROM actions WHERE id=2; INSERT INTO another_table
(value1) VALUES ('injected_value');--
```

SQL injection with stacked queries can be very powerful, as it allows the attacker to execute arbitrary SQL-commands on the database. However, tests on the browsers examined in this paper, reveal that stacked queries are not accepted by these browsers. Tests were performed on both Google Gears and HTML 5 Web SQL Databases.

3.3. Client-side data corruption

As data is stored on the client, the web application has no control over what the owner of the system does with this data. By opening a storage file, the user can change and corrupt the original data. In the case of cookies, changing the information can be as easy as opening a clear text file located in the users' profile and changing the contents. Sometimes the user might need to install an additional tool before he can change the information on his system. An example of this is the installation of an SQLite client to change the information that's stored in a HTML 5 Web SQL Database or a Flash-tool to open and change LocalSharedObject files.

The impact of users changing their data will vary among web applications. It all depends on the use of the data within the application logic. Following are two examples that illustrate the possible impact of this.

© 2011 The SANS Institute Author retains full rights.

Client-side data corruption example 1
A website allows users to place and watch online videos. Once the browser of the user loaded the video, the user can change the volume of the audio. The preferred volume is then saved into a Flash LocalSharedObject-file. Every time a user opens a new video, the preferred volume is now read from the local Flash-store.

Client-side data corruption example 2
An organization opens a portal through which customers can get information about the products the organization sells. Customers can also place an order for these products. In order to create a session, the web application stores a unique session identifier in a cookie, along with an indication whether or not this user has administrative privileges on the website ('admin=yes' or 'admin=no').

The examples outlined above illustrate that client-side data corruption will not form a problem with the video-website. Worst case scenario is that the audio-volume cannot be read and is then changed back to the default. Data corruption is a much bigger problem in the second example. If a regular user alters his cookie by replacing '`admin=no`' with '`admin=yes`' in his local cookie-file, he will now have administrative access to some of the functions in the portal. It should be stressed that controlling access to administrative function in this way is bad practice anyhow because it can be easily controlled by the user.

3.4. Client-side data leakage

Because a web application has little control over the data stored on a PC, it cannot guarantee the confidentiality of the data. Of course, technical issues like XSS and SQL injection can put client-side data in danger, and allow attackers to retrieve this information. This leakage of data can be problematic in case the data stored is of a sensitive nature or when this data you can be used to get access to sensitive information (replay attack).

© 2011 The SANS Institute Author retains full rights.

The cause of client-side data leakage is not limited to XSS, SQL injection or some other vulnerability in the web application. Leakage can also take place when an attacker has access to the file system of the user. This can be the case when the computer of the user is infected with some piece of malware or when the user logs in to a web application from a shared computer (e.g. internet kiosk).

A complicating factor with client-side data leakage is that often it isn't possible to automatically delete client-side data after a certain amount of time. An exception to this rule is the expiration time that can be set on cookies. By specifying an expiration time on cookies, the browser will automatically invalidate or delete the data after the information has expired. With other mechanisms it's not possible to define such an expiration time so that information will theoretically stay on the computer of the user forever.

3.5. Same origin policy bypass

The same origin policy is an extremely important concept when it comes to protecting client-side data. Bypassing this policy can have serious consequences. Assume a web application enables a user to administer information about customers. Because the web application has offline capabilities, this information is not only stored on the server but also on the client. If an attacker succeeds in bypassing the same origin policy for this web application, the attacker now has access to the information about all the customers of this organization.

So how can an attacker bypass the same origin policy? First of all, all variations of XSS allow attackers to bypass the same origin policy because this type of vulnerability enables the execution of scripts within the context of the attacked domain.

Unfortunately, vulnerabilities in the web application are not the only causes of same origin policy problems. Vulnerabilities in web browsers can result in browsers not enforcing the same origin policy properly. Table 4 (see next page) shows an overview of same origin policy bypass vulnerabilities in browsers discovered in 2010, based on the

© 2011 The SANS Institute Author retains full rights.

information in the National Vulnerability Database (NVD)[6]. If an attacker succeeds in exploiting such a vulnerability in a browser, he can have appropriated the client-side data.

CVE-ID	Description
CVE-2010-3934	The browser in Research In Motion (RIM) BlackBerry Device Software (…) does not properly restrict cross-domain execution of JavaScript.
CVE-2010-3259	WebKit (…) does not properly restrict read access to images derived from CANVAS elements, which allows remote attackers to bypass the Same Origin Policy.
CVE-2010-3178	Mozilla Firefox 3.5.x (…) and 3.6.x (…) do not properly handle certain modal calls made by javascript: URLs in circumstances related to opening an new window.
CVE-2010-2763	The XPCSafeJSObjectWrapper (…) in Mozilla Firefox (…) does not properly restrict scripted functions.
CVE-2010-2296	The implementation of unspecified DOM methods in Google Chrome (…) allows remote attackers to bypass the Same Origin Policy.
CVE-2010-1663	The Google URL Parsing Library (…) allows remote attackers to bypass the Same Origin Policy via unspecified vectors.
CVE-2010-1213	The ImportScripts Web Worker method (…) does not verify that content is valid JavaScript code.
CVE-2010-1206	The startDocumentLoad function (…) in Mozilla Firefox (…) does not properly implement the Same Origin Policy.
CVE-2010-0494	Cross-domain vulnerability in Microsoft Internet Explorer (…) allows user-assisted remote attackers to bypass the Same Origin Policy.
CVE-2010-0488	Microsoft Internet Explorer (…) does not properly handle unspecified “encoding strings”.
CVE-2010-0170	Mozilla Firefox (…) does not offer plugins the expected window.location protection mechanism.
CVE-2010-0162	Mozilla Firefox (…) does not properly support the application/octet-stream content type as a protection mechanism against web scfript in certain circumstances.

Table 4: Same Origin Policy bypassing vulnerabilities

[6] http://nvd.nist.gov/

© 2011 The SANS Institute Author retains full rights.

Same origin policy problems can also arise when a web application does not properly define the origin. The cookie mechanism e.g. allows a developer to set the origin through the 'domain' attribute of a cookie. Assume that the developer of mysite.domain.com uses a cookie to store information on the PCs of its users, whereby the cookie specifies that the domain is .domain.com. This can become a problem when the ownership of different (virtual) hosts under the domain.com domain is diffused. An attacker who sets up a website under evil.domain.com will in this case have access to the cookie information set by mysite.domain.com. This is typically a problem on websites like blogs and social media sites where every user has its own virtual host name (e.g. security.wordpress.com) under the same primary domain.

A similar problem exists on shared servers, where several users share web space on the same host. The web pages of these users typically have URL's like http://www.domain.com/~username/. As described in §2.1, traditional cookies allow for a path to be set so that the validity of a cookie can be constrained based on a path. If e.g. the path is defined as /~user1, then this cookie can only be read by the owner of the /~user1 branch of the website. HTML 5 storage technologies do not support such a mechanism which means that the origin cannot be narrowed down based on paths on the web server. If a developer uses HTML 5 storage technologies on a shared server, then this information can be read and changed by all the other users that own web space on the same shared server (Trivero, 2008).

Another way to bypass the same origin policy is by poisoning the DNS cache of a DNS server. If an attacker succeeds in injecting his own DNS records for a specific hostname, then the attacker can redirect every visitor of this website (who makes use of the vulnerable DNS server) to his own site. Because in this case the attacker mimics the website, he is able to execute random scripts within the context of the website attacked. This allows for unrestricted access to locally stored information for this website.

© 2011 The SANS Institute Author retains full rights.

4. Client-side data storage measures

To lower the risk exploitation of client-side data storage vulnerabilities, developers are advised to implement preventive measures. Based on the risks of the previous chapter, this chapter describes these measures.

4.1. Do not trust locally stored data

Because your web application has no control over the integrity of the data on the client, you should never trust locally stored data. Users should not be able to influence the application logic of the web application through locally stored data. Some examples of unwanted application behavior due to corrupted data were already illustrated in §3.3.

4.2. Consider the use of encryption

Encryption is a mechanism that's often used to achieve confidentiality of data. If a web application stores information locally, it can employ encryption to prevent others and malware from reading the cleartext representation of this data. The way the developer implements encryption for locally stored data is essential. If e.g. the key to decrypt the data is passed through JavaScript, the encryption is of no use. Therefore, locally encrypted data must be sent to the server first, after which the server decrypts this information based on a key that's only available to the server.

Mind that the use of encryption will not protect your data against every attack. If the server decrypts this information and echoes the cleartext information back to the browser, attackers can still gain access to this information by the use of XSS.

If you decide to encrypt data on the client, it's advisable to make use of a standard mechanism supported by the programming language you're using. This way you make sure you use a solid and proven solution and it also saves you from a lot of work. An example of this is `mcrypt` which allows for encryption through PHP[7].

[7] http://php.net/manual/en/book.mcrypt.php

© 2011 The SANS Institute Author retains full rights.

4.3. Consider the use of digital signatures

One advice that was already described in §4.1 is to not trust locally stored data. If your web application really needs to store information locally and you want to check the integrity of the data, you could make use of digital signatures. Just as with encryption, it's advisable to make use of a standard mechanism available instead of implementing your own one.

4.4. Prevent (cs)XSS

XSS allows an attacker to access information in local storage through malicious JavaScript. The impact of a traditional XSS vulnerability in a web application can be quite high, depending on the type of information stored on the PC of the end-user. If e..g. you have a web application that stores copies of e-mails locally, a XSS vulnerability in your web application will potentially allow an attacker to read all of these e-mails. Developers must be aware that, in these cases, attackers can do a lot more than just opening a harmless alert box on the users' PC.

XSS occurs when the web server doesn't sufficiently sanitize or encode the data that it receives or retrieves and echoes this data back to the user. The problem of XSS can be solved by introducing proper output encoding mechanisms in server-side scripts. However, the use of client-side data storage makes it possible that data used in the output, is never seen and sanitized by the server. This can happen when information is stored locally and retrieved by a client-side script as illustrated in §3.1. To prevent csXSS, the developer must introduce output encoding mechanisms on the client-side in the form of JavaScript-functions.

OWASP offers the Enterprise Security API (ESAPI)[8] that – among a lot of other functions - has built-in functions to encode output within different programming languages to prevent XSS. ESAPI libraries are currently available for Java, Microsoft .NET, PHP, ColdFusion, Python and JavaScript.

[8] https://www.owasp.org/index.php/Category:OWASP_Enterprise_Security_API

© 2011 The SANS Institute Author retains full rights.

What the ESAPI basically does in this case is encoding unsafe characters in the output. So e.g. the less-than-sign ('<') is HTML-encoded as '<'. This way, it's very hard for an attacker to inject tags in the output of the web application (and thus execute malicious JavaScript). Developers must use the functions offered by the ESAPI or otherwise encode output to prevent (cs)XSS.

4.5. Use parameterized queries

If you use a client-side database in your web application in combination with an SQL interface (Google Gears or HTML 5 Web SQL Database), you should use parameterized queries instead of dynamic strings.

Parameterized queries make use of placeholders whereby the browser will replace the placeholder with a dynamic value at runtime. An example use of parameterized queries was already illustrated in §3.2:

```
t.executeSql('SELECT description FROM actions WHERE id=?', (id))
```

4.6. Specify the data-origin as narrow as possible

If you specify the origin of the data too loose, the information you store on the PC could possibly be read by other web applications. The rule of thumb is to specify the origin as narrow as possible. So, if you use cookies in your web application, and the URL is http://secure.thisismywebsite.com, then specify the origin as secure.thisismywebsite.com and not thisismywebsite.com. In the latter case, cookies can also be read by www.thisismywebsite.com, other.mywebsite.com and etcetera. Obviously this is especially a problem on shared domains. Cookies also allow a path-variable to be set to further narrow down the origin.

Newer storage mechanisms as HTML 5 Web Storage only use the tradition tuple (scheme/host/port) to determine the origin. This means you should never use this kind of storage mechanism on shared servers where multiple users share web space within the same scheme/host/port tuple. This is often the case on servers where multiple users can create their own homepage on the some host (e.g. http://www.domain.com/~user).

Edwin Tump, edwin.tump@govcert.nl

© 2011 The SANS Institute Author retains full rights.

4.7. Limit the lifetime of client-side data

Except for cookies, the data storage mechanisms described in this paper do not support automatic deletion of data. This will result in the data being available on the system for a potentially long time. To prevent this from happening, web applications should delete data right after it's no longer needed. How this is done, depends on the mechanism used. Example 12 illustrates some of the (JavaScript-)actions that can be taken by a web application to remove data.

Example 12: delete information from local stores

- Cookies:

```
document.cookie = "sessionid=";   // there's no 'delete-cookie'-function
```

- Web Storage:

```
clear();                          // removes all items
removeItem("key");                //removes a single item
```

- Indexed Database:

```
removeObjectStore("store");       // removes the complete store
remove("key");                    // removes a single item
removeIndex("index");             // removes a single index
```

It should be noted that unfortunately not all mechanisms support a `clear()`- or `destroy()`-like function to clear all data in the repository or destroy the complete repository. With cookies e.g. you can not delete the cookie, but you must reset to value of the variable to nothing (NULL).

4.8. Test and assess

As with every web application, it's important to test your code on security vulnerabilities. As part of a penetration test, the pentester should first determine if the web application makes use of client-side data storage. This can be determined by

© 2011 The SANS Institute Author retains full rights.

monitoring the files created locally, checking JavaScript code and reverse engineering binary files (Flash, Java and Silverlight). Once it is determined that client-side data storage is used, the pentester must find out how information is stored and what kind of information is stored. The next step is to check how vulnerable the web application is for the risks described in chapter 3 of this paper.

To determine if the web application is vulnerable for csXSS, the pentester must find out if information is stored and retrieved locally without proper input filtering or output encoding. If a flaw is found in this process, the pentester should try to inject malicious JavaScript in the local data storage and, once this succeeds, develop Proof-of-Concept for that. Often, this is done through a CSRF-attack via a malicious website.

Client-side SQL injection (csSQLi) is only possible if the web application makes use of a storage mechanism that has an SQL interface (typically Web SQL Databases or Google Gears). It's quite easy to determine if the web application is vulnerable for csSQLi by evaluating the SQL calls made from JavaScript. If the calls make use of parameterized queries, then the web application is probably not vulnerable. Else, it is.

As stated earlier in this paper, data corruption is possible with every web application that uses local data storage because the user has full control over the information on his/her computer. The pentester must determine whether or not he can influence the web application logic by changing the information manually. Different tools exist to change this information, depending on the type of mechanism used. If e.g. one of the newer HTML 5 storage mechanisms is used, the pentester must probably install an SQLite client to manually change the information in the database. The pentester must evaluate the effects of changing the data and try out different variations. The effects really depend on the way the web application uses this information. It can lead to the bypassing of certain restrictions on the web application or enable the user to inject SQL statements if the information is used directly by the web application to make calls to a server-side database.

Just as with data corruption, data leakage is always a possibility with client-side data as information is stored on the PC of the user an can be opened by malware on the system of the user or another user on the same system. The pentester must evaluate if the

© 2011 The SANS Institute Author retains full rights.

information stored locally is of a sensitive nature and, if so, if this information is properly protected by encryption mechanisms. He must also check if the information can be used to e.g. login to the web application from another computer.

Finally the pentester must test if it is possible to bypass the same origin policy. The first step is to determine the parameters used by the web application to define the origin. If the origin is specified too loose, the pentester must find out if it's possible to create a malicious website within this origin. If the web application runs on a shared server, it might be possible to create a subsite on this server through which the pentester can get access to information stored by the web application. The web application must be scanned for XSS vulnerabilities because these vulnerabilities will probably allow the attacker to execute malicious JavaScript code in the context of the web application evaluated.

Table 5 summarizes the tests penetration testers should execute in order to evaluate the security of client-side storage mechanisms used by the web application.

Risks	Actions
csXSS	▪ Find out if information stored locally is echoed back to the user. ▪ If so, check if the output is properly encoded. ▪ If no proper encoding is done, try to inject scripts in the local database. ▪ Create a scenario where e.g. CSRF is used to automatically inject the malicious code through a malicious website.
csSQLi	▪ Determine if the web application makes use of an SQL-based local storage mechanism (Web SQL Database or Gears). ▪ Find out how queries are created: via dynamic strings or parameterized queries. ▪ Experiment with different inputs if the web application uses dynamic strings.
Data corruption	▪ Find out how information in local storage is used by the web application. ▪ Find out if digital signatures are used to protect the integrity of data. If so, evaluate the strength of the signature mechanism.

© 2011 The SANS Institute Author retains full rights.

Risks	Actions
	▪ Manually edit the information stored by the web application: − Change name/value pairs (cookies) − Change information in an LSO by loading .sol-files in a SOL editor (Flash) − Change information in a database by loading a database in a local SQLite-client (Web SQL, Web Storage, IndexedDB, Gears) − Change information in XML-files (Silverlight, UserData) ▪ Evaluate the effects of data corruption on the web application logic. Can you bypass restrictions? Can you SQL-inject statements on the server-side database? Et cetera.
Data leakage	▪ Evaluate the type of information that's stored locally. ▪ Find out if the web application contains a XSS of csXSS-vulnerability through which it's possible to execute malicious JavaScript within the context of the web application. ▪ If confidential information is stored locally without encryption, warn the web application owner of the risks. ▪ If information is encrypted, evaluate the strength of the encryption mechanism. Can it be broken? ▪ Can the information stored be used for replay attacks?
Same origin policy bypass	▪ Evaluate the origin set by the web application: − Domain and path variables (cookies) − Cross-domain policy (Flash and Silverlight) − Client access policy (Silverlight) − Cross-origin workers (Gears) ▪ Find out if you can create a malicious website within the origin specified (e.g. malicious.domain.com if origin is .domain.com or a subsite on a shared server). ▪ Find out if the web application contains XSS vulnerabilities that will access to the information.

Table 5: Pentesting client-side data storage

© 2011 The SANS Institute Author retains full rights.

5. Conclusion

There exist a lot of mechanisms to store information from a web application on a PC of an end-user. Some of these mechanisms are supported out-of-the-box by browsers, others must be enabled through the installation of browser plug-ins. Irrespective the technical implementation, the use of all of these storage mechanisms share some risks that must be addressed by the web application developer.

Most of the risks are not new and exist ever since the introduction of cookies in the 1990s. But some recent developments have changed the risk landscape. The fact that some applications store and retrieve data locally – without the need of server intervention – makes it possible to circumvent server-side filters and encoders. This introduces traditional server-side vulnerabilities to the browser (XSS → csXSS, SQLi → csSQLi). The use of client-side filters and encoders is required to battle these threats. Although client-side SQL injection may sound like a serious vulnerability, the effect of this is pretty limited. And next to that, all storage mechanisms that are theoretically vulnerable for csSQLi are no longer actively supported.

The growing amount of information also makes this information more interesting for miscreants. Traditionally information stored on a PC by a web application was mostly restricted to some identifiers like session ID's. With the introduction of offline web applications, the PC could now also contain complete databases with customer information or personal information like e-mails. This makes client-side data an interesting target for miscreants and increases the damage of XSS-vulnerabilities in the web application that serve as a vehicle to get access to this information.

Finally the absence of data expiration mechanisms in modern client-side data solutions results in data left behind on PC's. It's essential that web applications not only create and change information in local data stores but also pay attention to the active removal of data once it's not needed any longer.

Edwin Tump, edwin.tump@govcert.nl

© 2011 The SANS Institute Author retains full rights.

6. References

Adobe. "What are local shared objects?." *Adobe*. Adobe, n.d. Web. 19 Apr 2011. <http://www.adobe.com/products/flashplayer/articles/lso/>.

Adobe. "Adobe Cross Domain Policy File Specification." Adobe, 21 January 2010. Web. 19 Apr 2011. <http://learn.adobe.com/wiki/download/attachments/64389123/CrossDomain_PolicyFile_Specification.pdf?version=1>.

Austin, Calvin, and Monica Pawlan. *Advanced Programming for the Java 2 Platform*. 1st ed. Addison Wesley Longman, 2010. 400. Print.

Boneh, Dan. "Cookie Same Origin Policy." Stanford University, 30 January 2009. Web. 19 Apr 2011. <http://crypto.stanford.edu/cs142/lectures/10-cookie-security.pdf>.

EPIC. "Local Shared Objects -- "Flash Cookies"." *EPIC - Electronic Privacy Information Center*. Electronic Privacy Information Center, 21 July 2005. Web. 19 Apr 2011. <http://epic.org/privacy/cookies/flash.html>.

Fette, Ian. "Hello HTML 5." *Gears API Blog*. Google, 19 February 2010. Web. 19 Apr 2011. <http://gearsblog.blogspot.com/2010/02/hello-html5.html>.

Gears Team, . "Database API - Gears API - Google Code." *Google Code*. Google, n.d. Web. 19 Apr 2011. <http://code.google.com/apis/gears/api_database.html>.

Gears Team. "Gears and Security - Gears API - Google Code." *Google Code*. Google, n.d. Web. 19 Apr 2011. < http://code.google.com/intl/pl/apis/gears/security.html>.

Gears Team, . "Gears FAQ - Gears API - Google Code." *Google Code*. Google, n.d. Web. 19 Apr 2011. <https://code.google.com/apis/gears/gears_faq.html>.

Gears Team. "GearsHistory - Timeline of major events." *Google Code*. Google, 4 February 2010. Web. 19 Apr 2011. <https://code.google.com/p/gears/wiki/GearsHistory>.

Hickson, Ian. "Web Storage." W3C, 8 February 2011. Web. 19 Apr 2011. < http://www.w3.org/TR/2011/WD-webstorage-20110208/>.

Hickson, Ian. "Web SQL Database." W3C, 18 November 2010. Web. 19 Apr 2011. <http://www.w3.org/TR/2010/NOTE-webdatabase-20101118/>.

Edwin Tump, edwin.tump@govcert.nl

© 2011 The SANS Institute Author retains full rights.

'KirstenS','Jmanico', Williams, Jeff, 'Wichers', 'Roman', and Weidman, Adar. "Cross-site Scripting (XSS)" Open Web Application Security Project (OWASP), 20 October 2010. Web. 19 Apr 2011. < https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29>.

Kristol, D., and L. Montulli. "RFC 2109: HTTP State Management Mechanism." IETF, February 1997. Web. 19 Apr 2011. <http://www.ietf.org/rfc/rfc2109.txt>.

Likness, Jeremy. "Saving Bitmaps to Isolated Storage in Silverlight 3." Code Project, 31 July 2009. Web. 18 Apr 2011. <http://www.codeproject.com/Articles/38636/Saving-Bitmaps-to-Isolated-Storage-in-Silverlight-.aspx>.

Mehta, Nikunj, Jonas Sicking, Eliot Graff, and Andrei Popescu. "Indexed Database API." W3C, 19 August 2010. Web. 19 Apr 2011. <http://www.w3.org/TR/2010/WD-IndexedDB-20100819/>.

Microsoft. "DHTML Behaviors." Microsoft, n.d. Web. 19 Apr 2011. <http://msdn.microsoft.com/en-us/library/ms531078%28v=vs.85%29.aspx>.

Microsoft. " Isolated Storage (Silverlight QuickStart)." *The Official Microsoft Silverlight Site*. Microsoft, 5 March 2009. Web. 19 Apr 2011. <http://www.silverlight.net/learn/quickstarts/isolatedstorage/>.

Microsoft. "Making a Service Available Across Domain Boundaries." *MSDN*. Microsoft, May 2010. Web. 19 Apr 2011. <http://msdn.microsoft.com/en-us/library/cc197955%28v=vs.95%29.aspx>.

Microsoft. "Silverlight Overview." *The Official Microsoft Silverlight Site*. Microsoft, n.d. Web. 19 Apr 2011. <http://www.silverlight.net/getstarted/overview.aspx>.

'Nsrav','Wichers', 'KirstenS', 'Suei8423', Bergman, Neil and Siman, Maty. "SQL Injection" Open Web Application Security Project (OWASP), 1 March 2010. Web. 19 Apr 2011. <https://www.owasp.org/index.php/SQL_Injection>.

Oracle. "Policy Tool - Policy File Creation and Management Tool." *Oracle*. Oracle, 1995. Web. 19 Apr 2011. <http://download.oracle.com/javase/1.3/docs/tooldocs/win32/policytool.html>.

Pilgrim, Mark. "Local Storage - Dive Into HTML5." N.p., n.d. Web. 19 Apr 2011. <http://diveintohtml5.org/storage.html>.

Edwin Tump, edwin.tump@govcert.nl

© 2011 The SANS Institute Author retains full rights.

Sutton, Michael. "A wolf in sheep's clothing." ZScaler Research, 19 February 2009. Web. 18 Apr 2011. <http://zscaler.com/presentations/A%20Wolf%20in%20Sheep%27s%20Clothing.pdf>.

Trivero, Alberto. "Abusing HTML 5 Structured Client-side Storage." SecDiscover, 20 July 2008. Web. 18 Apr 2011. <http://packetstorm.orion-hosting.co.uk/papers/general/html5whitepaper.pdf>.

© 2011 SANS Institute, Author retains full rights.

© 2011 The SANS Institute Author retains full rights.