JSFS -
Unlimited JavaScript Access to Computer Resources Without Browser Plugins

Wolfgang Imig, jsfsproject@gmail.com, 2014-03-25

Abstract: JSFS (JavaScript File System Access) provides a secure solution to access computer resources by JavaScript. Instead of using a browser plugin technology (e. g. ActiveX or Java Applet), it communicates with a local helper application – JSFS Agent - over a remote server. The browser configuration does not need to be changed in order to work with JSFS.

Table of Contents

1 Architectural Overview

1.1 The Problem

For security reasons, JavaScript in web browsers provides only restricted access to computer resources. E. g. it is not possible to store files in an arbitrary folder, start applications or communicate with an USB device. Browser plugins can be used in order to achieve extended access to computer resources. They can be written as Java Applets, ActiveXes, Flash Player or Silverlight applications. But enabling this technologies in the browser configuration opens up more possibilities for attackers. That is why their usage is often forbidden in enterprise environments.

1.2 The Solution

Abbildung 1 shows an architectural overview of the JSFS solution. On the client side, the web browser shows the web pages sent from "Your Web Application", the web application that requires access to resources on the client computer. This access is performed by the help of JSFS Agent, an application installed on the client computer. It is started when the user logs on to the operation system and opens a long-poll request to the JSFS Dispatcher on the server. This service connects the browser and the agent over the server computer.

If JavaScript code acquires access to a computer resource, it sends a request to the JSFS Dispatcher which forwards the request to the JSFS Agent by responding to its long-poll request. The result is returned back to the JSFS Dispatcher by the next long-poll from the JSFS Agent. On receiving the long-poll, the JSFS web service passes the result to the primary JavaScript request.

It is also possible to use the reverse communication direction, whereby the JSFS Agent triggers JavaScript code in the Browser. This can be useful e.g., if the JSFS Agent has to watch a folder. When a new file is inserted, the JSFS Agent notifies the JavaScript code which in turn could notify "Your Web Application".
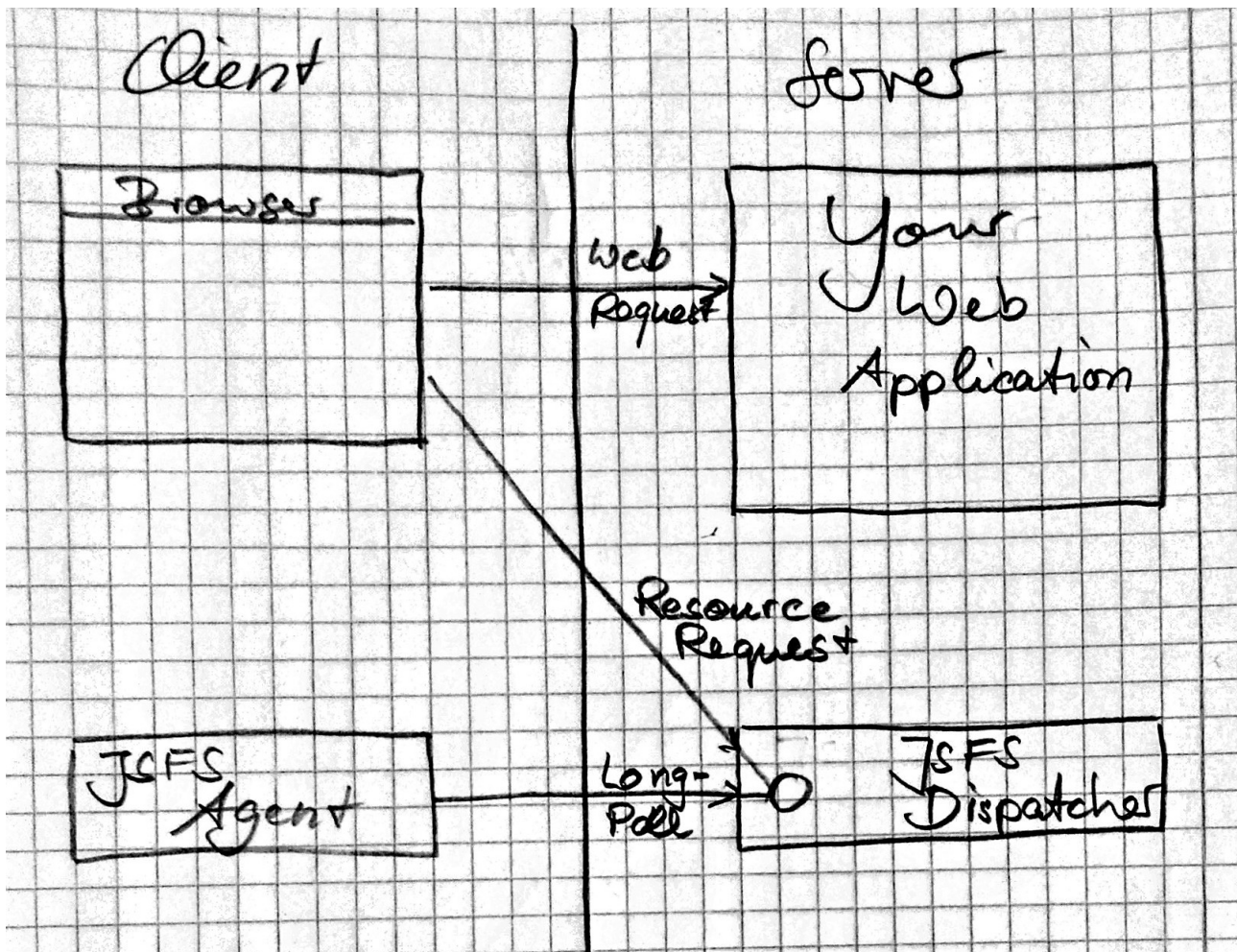
Abbildung 1: Architectural Overview

1.3 Field of Use

1.3.1 Focus on Enterprise Applications

One should be aware of the fact that all communication between the browser and JSFS Agent is sent over the network and through JSFS Dispatcher (see hint below). For this reason, JSFS is rather a solution for applications running on a LAN than applications using internet connections or a cell phone network. But JSFS could even provide an acceptable user experience for internet or mobile applications, if the transferred amount of data and the number of requests are small.

The development targets are aimed at companies:

- reliable security

- ease of deployment

- low system requirements

- simple extendability

- acceptable performance

Hint: We could also run JSFS Dispatcher on the Client, and we could even combine it with JSFS Agent. But this would make the installation procedure more complicated. Furthermore, attacks to

JSFS Dispatcher could not be recognized by an administrator (see section 5).

1.3.2 JSFS is Provided as Starting Point for Development

The JSFS solution is provided with full source code that can be modified to fit special requirements. For developing a JSFS Agent, starter examples in Java and C++ (MSVC 2012) are provided. They can be easily extended with additional functions or service interfaces.

1.3.3 Examples

For example, a JSFS Agent could perform the following tasks within web pages:

- Watch directories [*]

- Start an application in order to edit a document, get notified when the user has finished. [*]

- Upload files from arbitrary directories controlled by JavaScript.[*]

- Encrypt or digitally sign a document before it is uploaded to the server

- Control hardware devices like scanners, printers, card readers, etc.

- Check software protection dongles

[*] This functionality is included in the supplied example implementation.

1.3.4 Cannot show UI on Web Page

The JSFS Agent cannot show a user interface component integrated in a web page. However, it might have views and might allow user interaction like any other program running on the client computer.

1.4 Code Example

The following example tries to give an idea of how to work with the JSFS solution in JavaScript code.

The example in Abbildung 2 shows the JavaScript function "listMyDocuments" that lists the files in the folder "My Documents" on a Windows computer. JSFS Agent is represented by the object "jsfs" which provides the function "findFiles" for listing the files and sub directories of a folder. It sends a resource request to JSFS Agent and returns its results synchronously.

For simplicity, the example shows a synchronous invocation of "findFiles". But all service functions can be called asynchronously too.
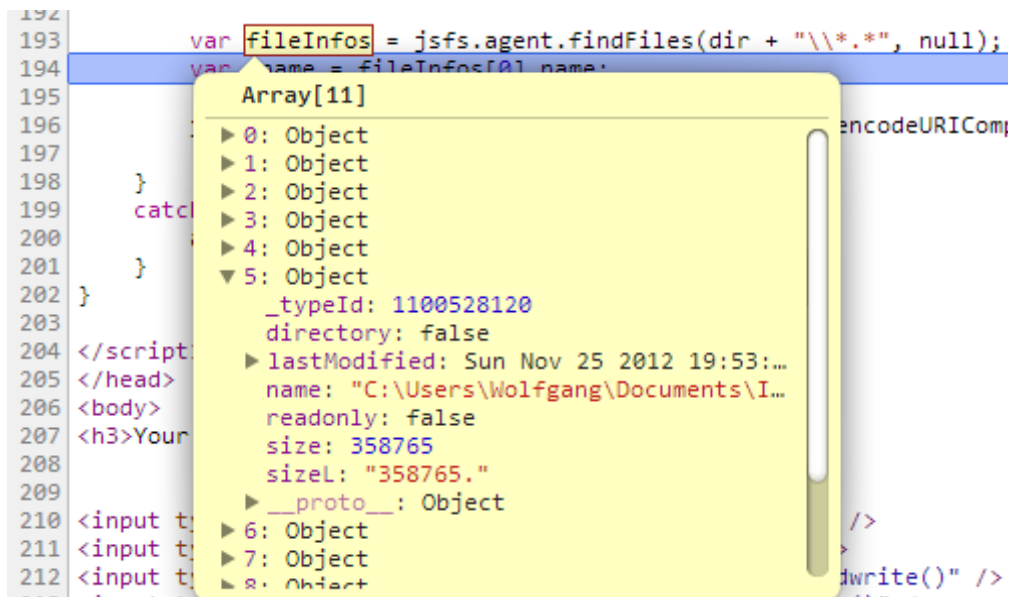
Abbildung 2: Synchronous example for listing files in a folder

1.5 Authentication, Connecting Browser to Agent

Referring to Abbildung 1, a resource request from the browser has to be directed to JSFS Agent running on the same computer in the same user context. This is achieved by using a token during communication between the browser and JSFS Agent, which is unique for each user.

Both, the browser and JSFS Agent, acquire the (same) token from "Your Web Application", see Abbildung 3, "Auth. Request". In an enterprise network, a single-sign-on mechanism over HTTP, e.g. with Kerberos or NTLM, can help to securely assign the token.

The JSFS Agent registers its services with the JSFS Dispatcher, see Abbildung 3, "RegisterService (Token)". If the browser requests a connection to the agent, see Abbildung 3, "getService (Token)", the JSFS Dispatcher is able to return the right stub.
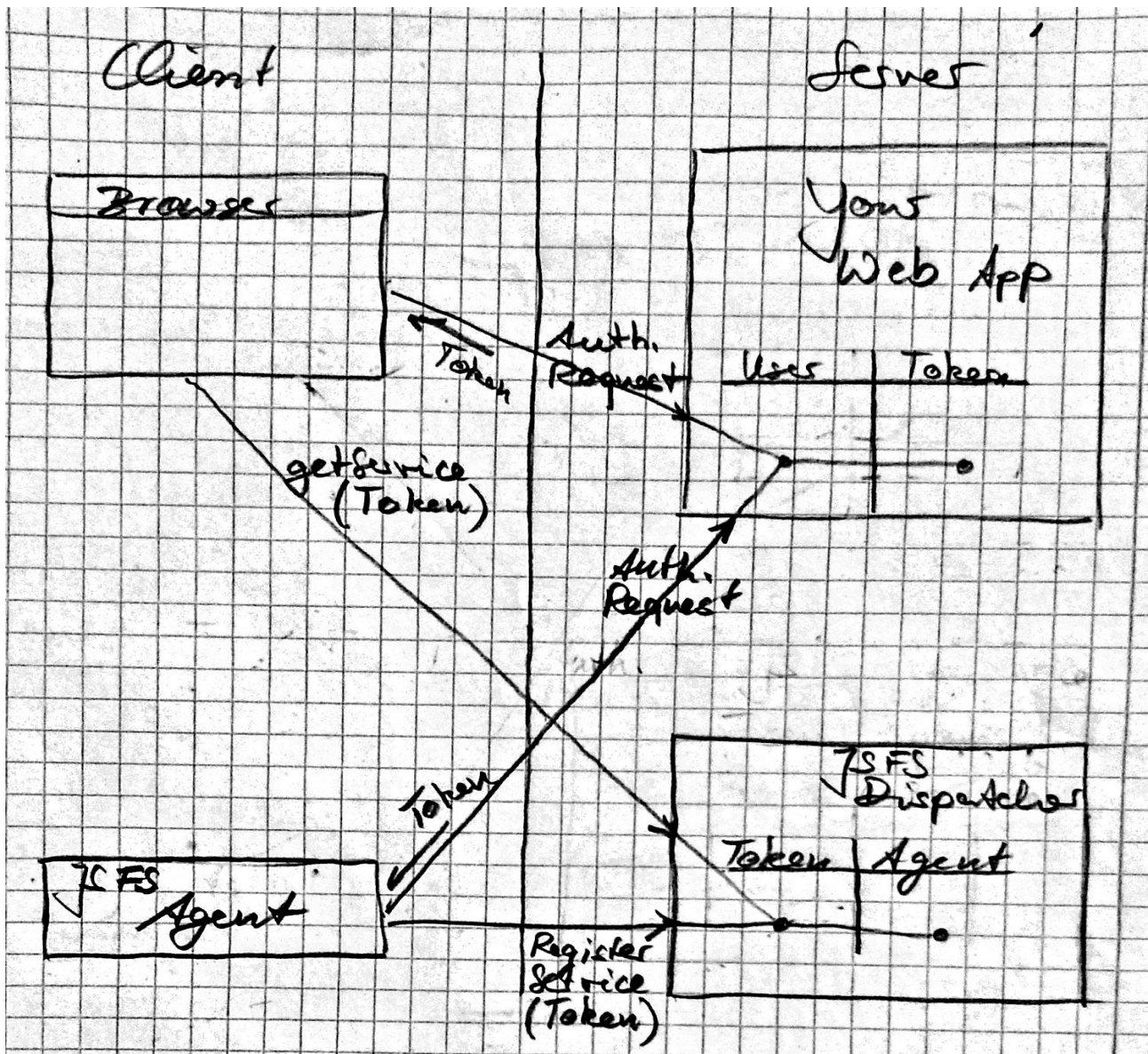
Abbildung 3: Authentication

1.6 Security Considerations

1.6.1 Authentication

The browser and JSFS Agent can authenticate against "Your Web Application" by using any standard HTTP authentication mechanism. The agent example implementation uses BASIC authentication in order to keep the example simple. In production environments on a LAN, a single-sign-on mechanism could be used, otherwise the user has to log-on twice: the first time for JSFS Agent and the second time in the browser.

If a malicious web page on the *internet* tries to access JSFS Agent, it has to authenticate against "Your Web Application" first. This would require a cross domain request which is declined by the browser. In order to workaround this barrier, the attacker could try a JSONP request or insert a proxy server between the browser and "Your Web Application".

- A JSONP request cannot be successful because it requires a special JavaScript response

format that invokes a callback function.

- The proxy server approach does not help, if JSFS is used on an intranet, because "Your Web Application" is not visible for the proxy (because it runs on the *internet*). If JSFS is used on the *internet*, the user would have to supply login credentials when navigating to the page of the attacker. This is similar to a phishing attack. It should be recognized by the user.

1.6.2 Same Origin Policy

For security reasons, browsers enforce that requests made by the XMLHttpRequest object can only be sent to the server address that has delivered the JavaScript code.

The easiest way to fulfill this requirement is by running "Your Web Application" and JSFS Dispatcher on the same web server.

If "Your Web Application" is an ASP.NET or PHP application etc., or JSFS Dispatcher cannot run on the same web server for other reasons, JSFS Dispatcher can be configured to allow [CORS] (Cross Origin Resource Sharing). This mechanism allows to bypass the Same Origin Policy and is enabled in web.xml. Another solution for this configuration can be obtained by using a revers proxy. Thereby the "preflight" requests would be avoided.

1.6.3 Cross Site Scripting Attack

If "Your Web Application" allows Cross Site Scripting Attacks, an attacker can misuse JSFS Agent. In order to minimize the damage in case of a successful attack, your JSFS Agent should obtain only functionality very special to you application.

1.6.4 Cross Site Request Forgery Attack

JSFS requests are complex JSON objects and use their own 128 bit random session IDs in addition to the application servers session IDs. A Cross Site Request Forgery attack cannot control JSFS Agent directly.

1.6.5 Attack JSFS Dispatcher

The result of authentication is a token that connects the browser with JSFS Agent. This token is build from a 128bit random number. Thus, a brute-force attack that tries out many tokens will most likely not succeed. Furthermore, such an attack is detected in JSFS Dispatcher. If function call "getService (Token)", see Abbildung 3, fails, the application server session object created for the browser connection is immediately invalidated.

Another target for an attacker could be the "registerNotifyService(token)" function. An attacker could call "registerNotifyService(token)" repeatedly in order to cause an Out-Of-Memory error in JSFS Dispatcher. To prevent this, the function should not store anything if the token was not used by a "registerService" call before.

The attacks described above can only disturb the system but cannot take control over JSFS Agent. Furthermore, they are not possible, if JSFS Dispatcher is embedded inside "Your Web Application" and both use [BYPS] for communication. In this case the interface of JSFS Dispatcher is only accessible with a valid session in "Your Web Application".

1.6.6 Execution Rights for JSFS Agent

JSFS Agent runs in the security context of the logged in user and does not require special client computer configurations or any extended execution privileges.

Especially, it is not necessary to open a TCP port on the client computer. JSFS Agent connects to JSFS Dispatcher and to "Your Web Application" the same way as the browser does.

1.6.7 HTTPS

It is highly recommended that a HTTPS connection is used between the components. This ensures, that the browser and the JSFS Agent are connected to the right servers. Furthermore, the traffic cannot be captured by a Man-in-the-middle attack.

1.7 Supported Programming Platforms

JSFS Dispatcher can only be developed in Java and there are no plans to support other programming platforms. A J2EE Servlet Engine (like Tomcat) is required to run the JSFS Dispatcher.

JSFS Agent can be written in Java for any platform and in C++ for Windows (Visual Studio 2012). The C#.NET platform could also be used, but currently no starter example is provided. An implementation for QT/C++ might come later.

1.8 Communication Layer: BYPS

Serializing and transferring data uses the [BYPS] communication layer. It provides optimized binary and JSON data transfer over HTTP. A reverse HTTP mechanism (Long-Polling) allows server-to-client and client-to-client communication.

BYPS uses a javadoc-based generator for building the serialization code. The interface and data definitions are specified in Java files. Thus, no special interface definition language has to be learned (as far as one has basic Java knowledge).

The generator can create serialization code for Java, JavaScript, C# and C++ (currently only MSVC 2012, C++11 Standard).

2 Setup Development Environment

Go through the following steps to setup a development environment in order to compile and debug the JSFS modules.

2.1 Java Software Development Kit

Download a recent Java [JDK] of version 7 or newer.

2.2 Download Tomcat 7.0 Application Server

Download [TOM7]

On Windows: download the ZIP file - not the installer package.

Unpack the ZIP archive into an arbitrary directory.

2.3 Download Eclipse IDE for J2EE Development

Eclipse can be downloaded at: [ECLIPSE]

Download "Eclipse IDE for **Java EE Developers**".

2.4 Download JSFS

– Create a main directory (referenced as "maindir") for the packages to be downloaded.

– Download [JSFS]

– Extract the ZIP file into "maindir", rename "maindir/jsfs-master" to "maindir/jsfs".

– Copy tools.jar from sub-directory "lib" of the JDK installation into "maindir/jsfs/lib"

2.5 Create Eclipse Workspace

– Start Eclipse

– Choose "maindir/jsfs" as workspace directory

– Click Icon "Workbench"

2.6 Configure Eclipse for Using Tomcat

– Start Eclipse

– Go to: Window – Preferences – Server – Runtime Environments

– Add...

– Choose: Apache Tomcat 7.0,

– Next >

– Enter the directory where the downloaded Tomcat ZIP archive has been unpacked.

– Finish.

2.7 Import JSFS Projects into Eclipse Workspace

– Start Eclipse

– Edit file: "maindir/jsfs/lib/byps.userlibraries

– Replace: "d:/git/jsfs" by "maindir/jsfs", Save.

– Go to: Window – Preferences – Java – Build Path – User Libraries

– Click "Import..."

– Choose "maindir/jsfs/lib/byps.userlibraries".

– Click "OK"

– Go to: File – Import...

– Choose: General – Existing Projects into Workspace

- Choose: Select root directory
- Browse...
- Select "maindir/jsfs"
- Check all listed projects
- Finish

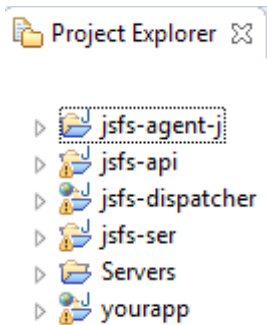The Project Explorer should show this projects now:

Abbildung 4: Eclipse Project Explorer with JSFS projects

The projects have the following purposes:

jsfs-agent-j	JSFS Agent written in Java.
jsfs-api	Interface definition of the JSFS Agent and JSFS Dispatcher. This interface is evaluated by the BYPS generator. The ANT script byps.xml runs the generator which produces all its output into the sub directories of the jsfs-ser project. Except the JavaScript output files, they are written into the files jsfs*.js in the folder yourapp/WebContent.
jsfs-dispatcher	JSFS Dispatcher
jsfs-ser	The BYPS generator creates this classes. They are used to serialize the request and response data in the communication between the browser, the JSFS Agent and the JSFS Dispatcher.
Servers	Definition for running and debugging a Tomcat. The tomcat-users.xml defines the role "yourapp-role" and a user "user" with password "pwd" that is used to authenticate to the token service in "Your Web Application". A logger configuration can be found in the context.xml file.
yourapp	This is a dummy web project that represents the application which uses JSFS. In the diagrams above it is named "Your Web Application". This project contains the JavaScript code in index.html that invokes the JSFS Agent. Furthermore this application hosts the token generation service in the Servlet class JsfsTokenServlet.

2.8 Deploy Web Applications

- Start Eclipse
- Go to: Window – Show View – Other...

- Select: Server – Servers
- OK
- Right-click in Servers view – New – Server
- Select Tomcat v7.0 Server
- Next
- Add All >>
- Finish

- Start the server via context menu or toolbar button. Open a browser window and go to address: http://localhost:8080/yourapp/
- The "Your Web Application" should be displayed.

Hint: If another port should be used for Tomcat, the JSFS Agent example code as to be adjusted. For the Java example in jsfs-agent-j, in class com.wilution.jsfs.Main the members jsfsDispatcherUrl and yourWebappUrl have to be modified.

2.9 JSFS Agent in C++: Install Microsoft Visual Studio 2012

In order to compile the JSFS Agent for C++, the Microsoft Visual Studio 2012 (or newer) for C++ development has to be installed. The free Express edition is not sufficient since MFC is used. At least the Professional version is required which is available as a free trial version.

The sources of BYPS are required:

- Download [BYPS]
- Extract the ZIP file into "maindir" (see 2.4), rename "maindir/byps-master" to "maindir/byps".

The JSFS Agent for C++ can be found in "maindir/jsfs/jsfs-agent-cpp".

3 Walk through the Examples

This section discusses the supplied examples. The file names, program names etc. used in the example code fit to Windows. It should not be difficult to find appropriate values for other operating systems.

3.1 Start the Applications

- Start the Tomcat server in Eclipse
- Start one of the JSFS Agents, either the C++ or the Java version.
  In order to start the Java version of the JSFS Agent, run the class com.wilutions.jsfs.Main in the project jsfs-agent-j. This can be achieved by the following steps: Expand in Project Explorer jsfs-agent-j/src/com.wilutions.jsfs, Right-click Main.java, Run As, Java Application.
- The JSFS Agent places an icon in the system tray and shows a message like Abbildung 5. This message tells us that the connection to the JSFS Dispatcher has been established and a token has been received from "Your Web Application". The credentials needed to request a

token are currently hard-coded in the Main class. (In production environments, a SSO mechanism could be used where by the credentials are automatically obtained by the operating system and the Java VM).
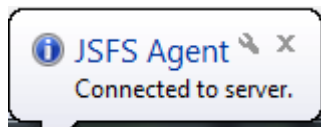
Abbildung 5: Tray icon and message from JSFS Agent

– Open a browser and navigate to address: http://localhost:8080/yourapp/index.html. The browser shows a page like this:
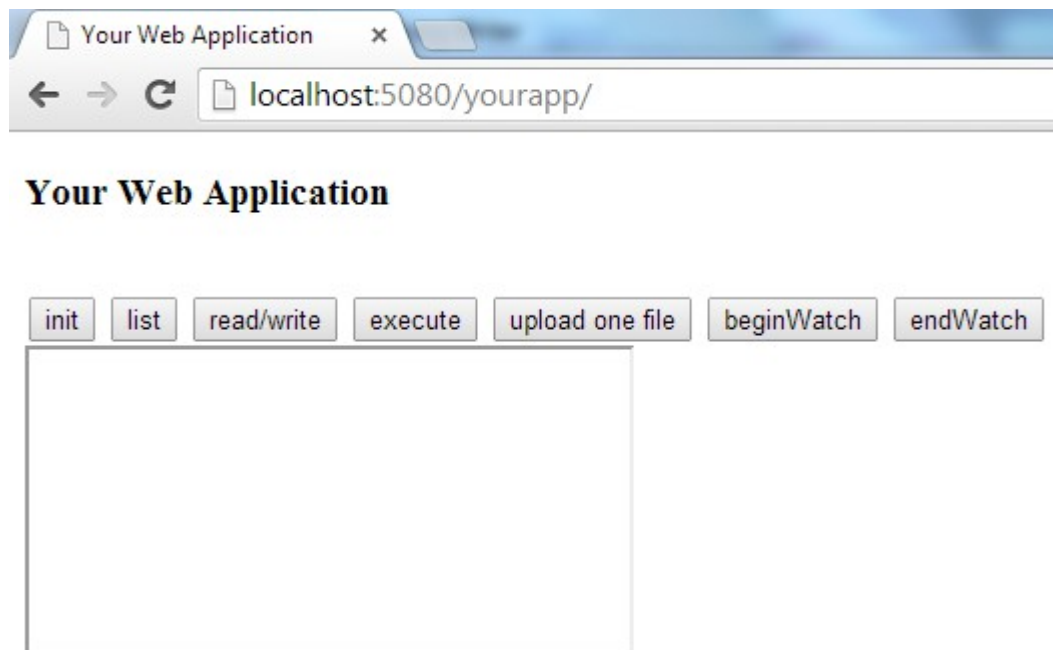
Abbildung 6: Your Web Application

– Start the browsers debugger (key F12 in Chrome and Internet Explorer).

3.2 Button "init": Connect browser to JSFS Dispatcher

– The browser is currently still not connected to the JSFS Dispatcher. Therefore, press button "init".

– As a result, the user name and password to login to the token service of "Your Web Application" has to be submitted: "user" and "pwd".

– If the login is successful, a message box with "OK" is displayed. Press OK.

Hint: The account user/pwd is configured in the tomcat-users.xml file that can be found in Eclipse, Project Explorer, Servers, Tomcat ...

3.3 Button "list": List files in a folder

– Look for function "fslist" in the debugger window and set a breakpoint at the line "jsfs.agent.listFiles..." as shown in Abbildung 7 for the Chrome browser.

```
55  /**
56   * List files in a directory.
57   * This function processes asynchonously.
58   */
59  function fslist() {
60
61      const  dir = "%HOMEDRIVE%%HOMEPATH%\\Documents";
62
63      jsfs.agent.findFiles(dir + "\\*.*", null, function(fileInfos, ex) {
64
65          if (ex) {
66              alert("Error: " + ex);
67          }
68          else {
69              var msg = "";
70              for (var i = 0; i < fileInfos.length; i++) {
71                  msg += fileInfos[i].name + "\n";
72              }
73              alert(msg);
74          }
75      });
76
77  };
```

Abbildung 7: Example for listing files

– Click button "list".

– The function findFiles is executed asynchronously at this line. If the result is available, the function object passed in the last parameter is invoked. Synchronous execution could be achieved by omitting the function parameter. Step with your debugger over this function call.

– In order to step through the code that receives the result, a further breakpoint has to be set, e.g. in line 65 of the screenshot in Abbildung 7. Continue program execution in the debugger.

– The second breakpoint shut be hit. Step over the next lines, watch the object contents as you like and continue program execution.

– An alert box should be displayed listing the files in the folder.

3.4 Button "read/write": Display file content in an IFRAME

This example shows that a file can be created an modified over the web page. Furthermore it displays the file content in an IFRAME.

– Find function "fsreadwrite". Set a breakpoint on "jsfs.agent.writeAllText", line 89 in screenshot Abbildung 8.

```
78
79  /**
80   * Read and write a text file.
81   * This function processes synchonously.
82   */
83  function fsreadwrite() {
84
85      const file = "%TEMP%\\jsfstext.txt";
86
87      try {
88          var text = "Text written at " + (new Date());
89          jsfs.agent.writeAllText(file, text);
90
91          var stream = jsfs.agent.readFile(file);
92          var fileView = document.getElementById("fileView");
93          fileView.src = stream.url;
94
95      } catch (ex) {
96          // JSFS Agent sets ex.code as Windows error code + 10000
97          alert(ex);
98      }
99
100 };
```
Abbildung 8: Example for writing and reading files

– Click button "read".

– Since the function "writeAllText" is not supplied with a function object at a last parameter, the function is executed synchronously. Step over this function in the debugger.

– Check in the file system, that the file has been created and written.

– The function "readFile" is executed next. This function reads the contents of the file and is defined with an InputStream object as return value. Since JavaScript does not provide stream objects, a wrapper object for an URL is returned. Step over "readFile".

– Watch the content of the returned object named "stream". The member "url" contains an address where the stream content can be downloaded, see Abbildung 9. This URL is only valid for some minutes and can only be read once.
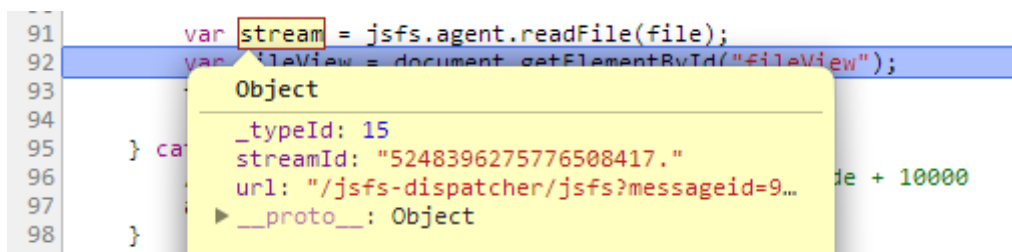
```
91          var stream = jsfs.agent.readFile(file);
92          var  ileView = document. getElementById("fileView");
93
94              Object
95      } ca
96          _typeId: 15                                          de + 10000
97          streamId: "5248396275776508417."
98      }   url: "/jsfs-dispatcher/jsfs?messageid=9…
            ▶ __proto__: Object
```
Abbildung 9: JavaScript stream object

– Continue program execution, the file content should be displayed in the IFRAME box under the buttons.

3.5 Button "execute": Start a Program on the Client

This example starts the command line program "sort.exe" which receives lines from standard input and print them alphabetically sorted on standard output. In this case, the result is not returned as a response to the browser request. It is sent to the browser over the "FileSystemNotify" interface. The JSFS Agent invokes the notify function of this interface, when the program has finished and submits the results. Thus, for transferring the result, the communication direction is reversed: the

JSFS Agent invokes the browser.

– Look for function "fsexecute" and set a breakpoint in the first line of the function as show in Abbildung 10.

```
102 /**
103  * Start a program on the client computer.
104  * This example starts the command line program "sort.exe" in ord
105  * The result is received in the FileSystemNotifyImpl.notify() fu
106  */
107 function fsexecute() {
108     try {
109         var opts = new com.wilutions.jsfs.ExecuteOptions();
110         opts.extraInfo = "DOS sort"; // This string is checked in
111         opts.standardInput = "BBB\nAAA\nCCC\n";
112         opts.captureOutput = true;
113         opts.captureError = true;
114         jsfs.agent.executeNotifyExit(["sort.exe"], opts);
115     }
116     catch (ex) {
117         alert(""+ex);
118     }
119 }
```

Abbildung 10: Execute a Program

– An object of type "ExeucteOptions" is prepared before the "executeNotifyExit" function is invoked to start the sort application. The member "extraInfo" is set to an identifiert that makes it easier to find out, which kind of execution result in the "FileSystemNotify" implementation is received. Furthermore the input lines are set which have to be passed to the application. In order to receive the results, the standard output pipe has to be captured. Therefore, the member "captureOutput" is set. Before continuing the debugger, set a breakpoint where the result is received. This is in function "notify" of the object "FileSystemNotifyImpl", see Abbildung 11.

```
129  * by setting the skeleton class as the prototype.
130  */
131 var FileSystemNotifyImpl = function() {
132
133     this.notify = function(notifyInfo) {
134         if (notifyInfo.error) {
135             alert("Notify error: " + notifyInfo.error);
136         }
137         else if (notifyInfo.extraInfo == "Watch Desktop") { //if
138             var kind = "Nothing";
139             switch (notifyInfo.kind) {
140             case com.wilutions.jsfs.EWatchFolderNotifyKind.Entry
141             case com.wilutions.jsfs.EWatchFolderNotifyKind.Entry
142             case com.wilutions.jsfs.EWatchFolderNotifyKind.Entry
143             }
144             alert("File change: " + kind + ", file=" + notifyInf
145         }
146         else if (notifyInfo.extraInfo == "DOS sort") { // if (no
147             alert("Process terminated, exitcode=" + notifyInfo.e
148                     ", extraInfo=" + notifyInfo.extraInfo +
149                     ", stdout=" + notifyInfo.standardOutput);
150         }
151     };
152
153 };
154 FileSystemNotifyImpl.prototype = new com.wilutions.jsfs.BSkeletc
```

Abbildung 11: Notify interface implementation

- Continue debugger execution. The breakpoint in "notify" should be hit.
- Step through the function and watch the passed "notifyInfo" object. After continuing the execution an alert box shows the sorted lines.

3.6 Button "upload one file"

This example uploads a file to "Your Web Application". A file can be uploaded either by sending the file content as the body of a simple HTTP POST request or by sending a multipart/form-data encoded request. The former solution is to be preferred at least for the Java version of the JSFS Agent, since it does not require to load the entire file content into main memory.

- Find function "fsuploadFile" int the source code and set a breakpoint at the call to "jsfs.agent.uploadFile".

```
104
185  /**
186   * Upload a file.
187   */
188  function fsuploadFile() {
189      try {
190          const url = "upload?";
191          const dir = "%HOMEDRIVE%%HOMEPATH%\\Documents";
192
193          var fileInfos = jsfs.agent.findFiles(dir + "\\*.*"
194          var fname = fileInfos[0].name;
195
196          jsfs.agent.uploadFile(fname, url + "fname=" + enco
197
198      }
199      catch (ex) {
200          alert(ex);
201      }
202  }
203
```

- Click button "upload one file". The debugger execution stops at the breakpoint.
- The code before has requested the files in the users "Documents" folder and will take the first one for upload.
- Before continuing the program, set a breakpoint in Eclipse in project "yourapp" at "Java Resources/ src/your.app/UploadServlet.java" in function "doPost". This function will receive the upload request from JSFS Agent.
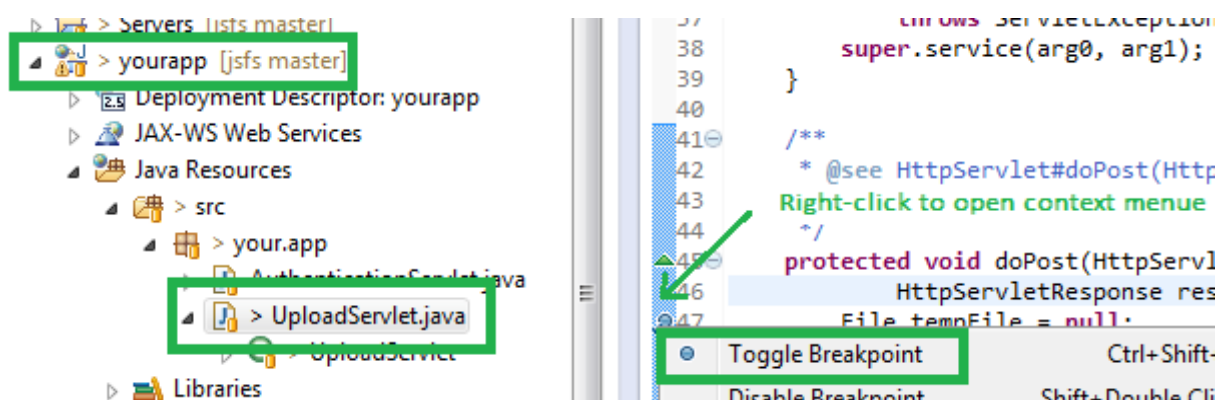
- Now, continue program execution in the browsers debugger.
- The breakpoint in Eclipse should be hit.
- Step through the code and memorize the name of the temporary file (move the cursor over variable "tempFile" to see the file path).

- Then continue program execution by pressing the key F8.
- Check whether the file was uploaded: open the "tempFile" in the file system.

3.7 Button "beginWatch": Watch a Folder

This example watches a folder in the file system and shows an alert box, if a file is created, deleted or modified. The notifications about changes are transmitted over the "FileSystemNotify" service.

- Find function "beginWatch" in the source code and set a breakpoint at the call to "jsfs.agent.beginWatchFolder".

```
162  * call function endwatch().
163  */
164  var watchFolderId = 0;
165  function beginWatch() {
166
167      var dir = "%HOMEDRIVE%%HOMEPATH%\\Desktop";
168
169      watchFolderId = jsfs.agent.beginWatchFolder(dir,
170              false,              // Do not watch sub folders
171              "Watch Desktop"     // This helps FileSystemNotifyImpl.n
172      );
173
174      alert("The browser displays a message box, when you create, delet
175  }
```

Abbildung 12: Begin watching a folder

- Click on button "beginWatch". The debugger execution stops at the breakpoint.

- The first parameter of the function "beginWatchFolder" specifies the directory to be watched. In this example, the desktop folder is supplied. The next parameter tells the function whether sub-directories should be watched too. The third parameter is an identifier that helps to find the right handler code in the "FileSystemNotifyImpl.notify". The return value is a handle to internal to the watcher object in the JSFS Agent. It has to be released with "FileSystemService.endWatchFolder".

- Continue program execution. An alert box shows a message which tells that the folder is being watched now. Click OK.

- Set a breakpoint in the function "FileSystemNotifyImpl.notify" as shown in Abbildung 11.

- Create a new file on the desktop. The breakpoint should be hit.

- Step through the function and watch the passed "notifyInfo" object. After continuing the execution an alert box shows the modification.

- If a file is renamed, three notifications are received: "EntryDeleted", "EntryCreated", "EntryModified".

- Stop watching the folder with a click on "endWatch".

3.8 Authentication

The authentication procedure is defined by BYPS, the underlying communication framework. It expects an implementation of the BAuthentication interface which has the following functions:

authenticate	Performs authentication with the server.
	For the JSFS application, it requests a token from "Your Web

	Application" by sending a HTTP GET request to the Servlet "your.app.AuthenticationServlet" at URL ".../auth".
isReloginException	Checks whether the given exception indicates that a new authentication should be processed.

Any exception that occurs during a request to the services of the "BClient" object is routed through this function. If the function returns true, the framework re-authenticates and performs the request again.

Since the JSFS application does not define special exceptions that denote an invalid session, the function forwards the exception to a default function in the framework.

The exceptions that may occur in the functions of the "jsfs.agent" object are not routed through the "isReloginException" since the "jsfs.agent" is not a service of the "BClient" object. |
| getSession | Returns an object that represents the current session and that is sent with every request. This functionality is not used for the JSFS application. |

The functions "authenticate" and "getSession" can be implemented synchronously or asynchronously. It depends on how the "start" function of the "BClient" object is called. In this example, the synchronous mode is used because the code to handle it is less complex, see file "jsfs-conn.js".

On the server side, the Servlet "your.app.AuthenticationServlet" is responsible for creating the authentication token. This is build by computing the MD5 hash from a string which consists of the user name plus the remote address plus a secret number. Since the Browser and JSFS Agent connects with the same user and from the same remote address, both receive the same token.

– Set a breakpoint in the "authenticate" function found in file "jsfs-conn.js".

– Set a breakpoint in "AuthenticationServlet.doGet".

– Click "init" and step through the program.

4 Customize JSFS Agent

This section explains, how to add functionality to the example implementation of JSFS Agent. It is already mentioned, that BYPS is used as a communication layer between the browser, JSFS Agent and JSFS Dispatcher. Thus, this section also provides some basic knowledge about BYPS.

The service interfaces of the JSFS Agent and the JSFS Dispatcher can be seen as entities of an API from the perspective of the JavaScript code in the browser. For this reason, in the terms of BYPS the interfaces and classes used in communication are called an API.

The language to be used to define an API is Java. Hence, no special interface definition language has to be learned.

Based on the API definition, the BYPS generator creates code for serializing classes and for providing a remote-procedure-call mechanism. This code is written into the project "jsfs-ser". The generator is started by Ant script byps.xml in project "jsfs-api" (see [ANT] for more information about Ant).

4.1 JSFS API

It is a good practice to collect all the API entities in a single project, here it is the the project "jsfs-api". The entire API must have one descriptor class named BApi with at least a name and a version number. In this case, the API descriptor is the class com.wilutions.jsfs.BApi.

In general, API classes can only consist of data members and cannot have functionality – except trivial getter and setter functions. All the functionality must be placed in interfaces.

In order to create backward compatible APIs, the version number must be incremented when new symbols are added. The new symbols must be tagged with the javadoc comment "@since" followed by the version number since when they are available. Other modifications, like removing symbols, changing the type of data members or changing the parameter list of functions, break compatibility and should be avoided.

4.2 Example: Add a Data Member to FileInfo

This example shows the steps to add a data member, e.g. to the FileInfo class. Adding a function to an interface can be done the same way.

 – Start Eclipse

 Increment the API version number:
 – Open the class com.wilutions.jsfs.BApi in the code editor.
 – Set version number e.g. "1.0.0.1".

 Add class member:
 – Open class com.wilutions.jsfs.FileInfo from the project "jsfs-api" in the code editor. Add a new "protected" member of type "boolean" named "hidden".
```
/**
 * True for hidden file or directory.
 * @since 1.0.0.1
 */
protected boolean hidden;
```
 – Generate a getter and setter function via the context menu – Source – Generate Getters and Setters...
 – Save the files

 Run the BYPS generator:
 – Right-click on byps.xml – Run As – Ant Build.
 – The console windows in Eclipse should show the generation process.
 – When finished, **refresh the project "jsfs-ser"** by Right-click – Refresh.

 Extend the JSFS Agent:
 – Open file "FileInfoHelper.java" in project "jsfs-agent-j".
 – In function "makeFileInfo" add line "fi.setHidden(file.isHidden());"

– Save "FileInfoHelper.java"

Extend the C++ Agent:

– Open "jsfs-agent-cpp" in Visual Studio.
– (Open file "jsfs-ser/src-api/JSFS-api.h" to check whether FileInfo class has been extended)
– Open file "FileSystemServiceImpl.cpp" and go to function "fileInfoFromWin32FindData".
– Add line "fi->setHidden((fd.dwFileAttributes & FILE_ATTRIBUTE_HIDDEN) != 0);".
– Compile project.

Test your modification:

– E.g. store a hidden file in the directory used in "fslist" and click the "list" button. Check in the debugger window that the new member is properly set.

4.3 Add a New Service Interface

In order to add a new service interface, a new Java interface has to be defined in the "jsfs-api" project. The interface can be created in an arbitrary package but not in the default package. It has to extend "byps.BRemote". The interface functions must be declared to throw a byps.RemoteException.

If the interface should be implemented on the client side, which is here the JSFS Agent or the JavaScript side, it must be tagged with "@BClientRemote", see "FileSystemNotify" or "FileSystemService".

If only the server side, here JSFS Dispatcher, will provide an implementation, no special tag is required.

New interfaces must also be tagged with a @since value.

4.4 Add a New Value Class

As already mentioned, classes are used to transmit data and cannot have functionality (all the functionality is placed in interfaces). This is why classes are called "Value Classes" in the terms of BYPS.

Value Classes have to implement the marker interface "java.io.Serializable" and have to define the unique (generated) "serialVersionUID" member. The lower 4 bytes of this value are taken to compute the internally used type ID. This type ID must also be unique in the domain of an API, otherwise the generator will stop with an error message. If you miss to define a "serialVersionUID", the generator creates one set to a 4bit wide random number.

New classes must tagged with the @since tag as all other new symbols.

4.5 Example: Send Information from JSFS Agent to Server (JSFS Dispatcher)

Define a New Function

– Open "DispatcherService.java" at "jsfs-api/src/com.wilutions.jsfs"
– Add a new function as follows:

```java
public String sendInfo(HashMap<String, String> props, InputStream stream, byte[]
bytes)
        throws RemoteException;
```

– Run "byps.xml" in project "jsfs-api".

– Right-click "jsfs-ser", "Refresh"

Implement the New Function in JSFS Dispatcher

– Open "JsfsDispatcher" at "jsfs-dispatcher/Java Resources/src/com.wilutions.jsfs"

– Add an implementation of the function defined above. E.g. right-click somewhere in the
editor window, "Source – Override/Implement Methods...", check "sendInfo(.... without
BAsyncResult)", click "OK".

```java
@Override
public String sendInfo(HashMap<String, String> props, InputStream stream, byte[]
bytes) throws RemoteException {

    System.out.println("props=" + props);
    System.out.println("bytes=" + Arrays.toString(bytes));

    try {
        FileOutputStream fos = null;
        try {
            fos = new FileOutputStream(File.createTempFile("jsfs", ".txt"));
            int c = 0;
            while ((c = stream.read()) != -1) fos.write(c);
        }
        finally {
            if (fos != null) fos.close();
            if (stream != null) stream.close();
        }
    }
    catch (IOException e) {
        throw new BException(BExceptionC.IOERROR, "Failed to transfer stream", e);
    }

    return props.toString();
}
```

– Press "CTRL-SHIFT-O" to add missing imports

Call the New Function in JSFS Agent, Java

– Open "FileSystemImpl.java" in "jsfs-agent-j/src/com.wilutions.jsfs"

– Lookup function "writeAllText"

– Edit the function body:

```java
@Override
public void writeAllText(String path, String text) throws RemoteException {
    internalWriteAllText(path, text, false);

    File file = getFile(path);
    HashMap<String, String> props = new HashMap<String, String>();
    props.put("hello", "jsfs");
```

```java
    try {
      bclient.getDispatcherService().sendInfo(props,
          new FileInputStream(file), text.getBytes());
    } catch (FileNotFoundException e) {
      throw new BException(BExceptionC.IOERROR, "Transfer stream failed");
    }
  }
```

Try Out

- Set a breakpoint in function "writeAllText" in "FileSystemImpl.java" at "jsfs-agent-j/src/com.wilutions.jsfs"

- Set a breakpoint in function "sendInfo" in "JsfsDispatcher" at "jsfs-dispatcher/Java Resources/src/com.wilutions.jsfs"

- Start "Tomcat v7.0 Server ..."

- Start JSFS Agent: Right-click "jsfs-agent-j/src/com.wilutions.jsfs/Main.java", "Debug As – Java Application"

- Start a browser and navigate to "http://localhost:8080/yourapp"

- Click "init", login with "user" and "pwd".

- Click "read/write"

- The breakpoint in "FileSystemImpl.java" is hit

- Press "F8" to continue the program

- The breakpoint in "JsfsDispatcher" is hit

- Press "F6" to step through the code or "F8" to continue.

4.6 Generator Can Check for Incompatible Modifications

The BYPS generator creates an XML file containing the entire API definitions: bapi_new.xml. This file can be used by the generator in the next run in order to detect, whether the modified API is compatible to the previous one. Therefore, the file must be renamed to "bapi.xml", before the generator is started the next time.

You may want to rename file "bapi_new.xml" to "bapi.xml" every time when you check in the code into the source code control system.

4.7 Remove Obsolete Functions Before Release

The original source code of JSFS Agent is written as an example for general purpose file system access.

In order to constrain the potential damage an attacker could cause in case of a successful attack, you may want to remove all obsolete example functions and expose only those functions that are actually required.

4.8 Cross-Domain Usage

When using JSFS Dispatcher on a different server than "Your Web Application", the browser has to send cross domain requests to JSFS Dispatcher. In order to allow this kind of requests, the JSFS Dispatcher has to respond with the "Access-Control-Allow-Origin" header in each request. Tomcat provides a Servlet filter – the CORS filter – to achieve this. This filter can be easily added to the

JSFS Dispatcher's web.xml file. But the CORS filter has two disadvantages:

– Each request hast to be "pre-fighted" by a HTTP-HEAD request that tests whether the call is allowed. This reduces performance.

– Firefox does not support to send the HTTP session credentials in synchronous requests, thus all requests have to be asynchronous which makes the code much more difficult to write.

Another solution is to insert a reverse proxy between the browser and the servers. The browser talks solely with the proxy and thus does not have to send cross domain requests. This solution causes more effort in installation but allows to write synchronous requests.

5 Related Work: Gibraltar

A similar project for accessing hardware resources from JavaScript was published by Microsoft Research in 2012, see [MSGIB]. The authors suggest to run a HTTP server, the "device server", as an extension component on the "Client".

An advantage of Gibraltar over JSFS is that data flow between the browser and the helper component (device driver resp. JSFS Agent) is faster.

But Gibraltar has some disadvantages compared to JSFS:

- The device server is visible to all web pages viewed in the browser.
  - If a malicious web page tries to misuse the device server and the device server asks the user to confirm access, the user might accidentally click the wrong button because she does not expect a question at that time (This problem applies to HTML5 too).
  - In contrast, JSFS Agent is not visible to any web page or application. The malicious web page would have to authenticate to "Your Web Application". This attack is discussed in 1.6.1: if it is possible at all, the user would have to enter login credentials.

- Authorization relies on the "Referer" HTTP header.
  - If the attacker is able to fake the "Referer" header, she is also able to open a session to the device server by sending a domain that has already been registered with the device server (see section 2.1 and Figure 2 in [MSGIB]). The "capability token" - which is the session token, what else? - does not help to remedy this problem.
    - Assume domain x.com is authenticated to access the camera.
    - Assume web site x.com is not opened in the browser.
    - Assume user opens the malicious web site y.com
    - y.com sends OPEN_SESSION with faked req.referer=x.com.
    - active_tokens[x.com] will be set to a new token ("TOKEN:123")
    - y.com sends DEVICE_CMD with req.referer=x.com, req.device=camera and req.token="TOKEN:123".
    - access_hardware() will be called.
  - Android (at least version 4.1.2) does not have a firewall. An HTTP server is visible to all devices connected to the WLAN. This makes it easy to access Gibraltar with a faked "Referer" header.
  - The [HTTP] definition recommends in section 15.1.3: "...that the user be able to select whether or not the "Referer" field is sent." and some lines below: "Clients SHOULD NOT include a "Referer" header field in a (non-secure) request if the referring page was

transferred with a secure protocol." Hence, in order to use Gibraltar, the user must allow the "Referer" header and the device server should be accessed over HTTPS.

- The device server is a general purpose component which makes it extremely interesting for a large range of attackers. In contrast, JSFS Agent can be constrained to the special requirements of the web application which is normally only useful for a small number of attackers.

- Complex installation procedure of the device server:

  ○ Installation requires firewall configuration in order to allow the device server listening on a TCP port and allowing requests from localhost but disallowing requests from other addresses.

  ○ Installation on Terminal Servers require "Virtual IP Addressing" to allow multiple HTTP servers listen on the same TCP port.

6 Bibliography

References

CORS: Mozilla Developer Network, HTTP access control (CORS), , https://developer.mozilla.org/en-US/docs/HTTP/Access_control_CORS

BYPS: Wolfgang Imig, , , https://github.com/wolfgangimig/byps.git

JDK: ORACLE, , , http://www.oracle.com/technetwork/java/javase/downloads/index.html

TOM7: Apache Software Foundation, , , http://tomcat.apache.org/download-70.cgi

ECLIPSE: Eclipse Foundation, , , http://www.eclipse.org/downloads/

JSFS: Wolfgang Imig, , , https://github.com/jsfsproject/jsfs.git

ANT: Apache Software Foundation, ANT, , http://ant.apache.org/

MSGIB: Kaisen Lin, David Chu, James Mickens, Li Zhuang, Feng Zhao, and Jian Qiu, Gibraltar: Exposing Hardware Devices to Web Pages Using AJAX, 2012, http://research.microsoft.com/apps/pubs/default.aspx?id=161386

HTTP: , Hyper Text Transfer Protocol RFC 2616, , http://tools.ietf.org/html/rfc2616