ADOBE® AIR® HTML Security

Legal notices

Contents

AIR HTML Security Overview

The Adobe® AIR™ HTML security model is based on developer feedback and an evolving understanding of the security issues inherent in developing HTML and Ajax applications. The development of Rich Internet Applications for AIR presents new challenges because AIR applications provide, unlike Web-based RIAs, access to the desktop. This means that AIR applications can read and write files, draw to the screen, communicate with the network, and so on.

Security is a key concern of Adobe, users, system administrators, and application developers. For this reason, Adobe AIR includes a set of security rules and controls to safeguard the user and application developer. This white paper presents the security considerations in developing HTML-based AIR applications.

For details on AIR security, see the Adobe AIR Security white paper.

The security challenges of RIAs

One of the main threats facing HTML applications, whether desktop or web, is that of injection attacks that result in malicious code execution, such as cross site scripting (XSS). Code is usually injected via a few common vectors such as URL handling (`javascript:` and other dangerous schemes), `eval()` and assigning external HTML content to DOM elements such as `innerHTML` and `outerHTML`. Malicious code can also be injected into a trustworthy service (such as Google Maps) if the loading operations happen over HTTP.

These vulnerabilities are exposed when a developer assumes that the application is consuming harmless content when in fact it contains malicious code. As an example, an HTML-based AIR application that extracts comments from an online blog and displays them might allow users to write comments such as the following:

```
<a href="#" onclick="var f=new air.File('c:\\something.txt'); f.deleteFile();">I'm cool!</a>
```

If the user clicks the link, files could be deleted. The damage could be much worse and security vulnerabilities are common in browser-based HTML and Ajax applications.

The assumption that damage from malicious content loaded via JavaScript Object Notation (JSON) can be limited by existing domain sandbox restrictions is incorrect. All data within even a granular DOM sandbox boundary such as a `frame` or `iframe` operates in that sandbox regardless of its origin. Even the most granular frame-defined DOM sandbox boundary can only function successfully if the content is placed in a different origin domain. Moreover, common design and implementation patterns encourage the use of these dangerous practices.

For traditional Ajax applications, this has been acceptable, since applications that run within the browser are sandboxed. Isolated from the user's computer and bound by a same origin policy, browser applications have a unique symmetry.

Rich internet applications, which have access to feature-rich APIs, present unique challenges. Even a simple application, if poorly coded, could contain an injection flaw resulting in a severe compromise of the user's machine. For any RIA platform, the damage potential from a malicious attack is directly proportional to the value its runtime provides. The more powerful and feature-rich are the APIs in an RIA platform, the greater the risk.

Security protection through sandboxes

To mitigate these risks, AIR now provides a comprehensive security architecture that defines permissions for each internal and external file in an AIR application. Permissions are granted to files according to their origin and are assigned to logical security groupings called *sandboxes*. The runtime uses these sandboxes to define access to data and executable operations:

- Code in the application sandbox has access to AIR APIs, has a limited ability to generate code dynamically after a page loads, and cannot load content via a `script` tag.
- Code in all other sandboxes (referred to as non-application sandboxes) essentially mimics typical browser restrictions and behaviors for local trusted HTML.

The application sandbox

The primary function of the application sandbox is to enable interaction between AIR APIs and files in trusted directories. Isolation from code in non-application sandboxes is maintained by disabling the JavaScript parser after initial page load so that code such as `eval()` throws an exception. Setting `innerHTML` is permitted, but any script it contains is ignored. This provides broad functionality without exposing AIR APIs directly to externally loaded data.

Developers reference trusted directories within the application sandbox using the `app:/` URL scheme. Even some files in the application directory can be placed in a non-application sandbox if they are loaded into a `frame` or `iframe` using the AIR `sandboxRoot` and `documentRoot` attributes if the `frame` or `iframe`. All files included within an AIR installer are automatically installed in app:/ and therefore are part of the application sandbox. When the application runs, these files are given the full set of AIR application privileges, including interaction with the local file system. Content outside `app:/` is not part of the application sandbox and is therefore treated the same way as it is in the browser.

AIR APIs provide a secure means to safely bridge an evolved browser security model with a desktop-like security model. The AIR sandbox bridge model enables broader functionality and less risk than is associated with typical browser-based applications.

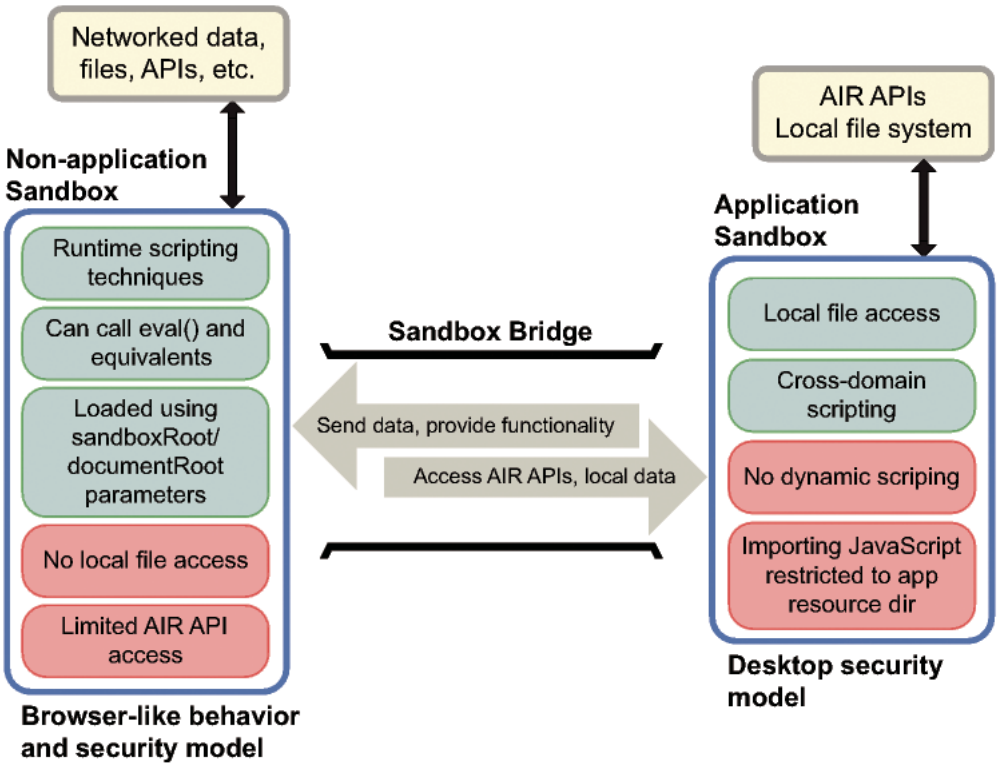
The non-application sandboxes

The primary function of the non-application sandboxes is to limit access to local files and AIR APIs unless permitted by the developer. Content loaded from a network or Internet location is automatically assigned to this sandbox. The non-application sandboxes can use the `eval()` function, execute code via `innerHTML` assignments, and so on. Code in non-application sandboxes is bound by a same-origin policy and, by default, cannot do cross-domain data loading via `XMLHTTPRequest`. However, code in a non-application sandbox cannot directly execute AIR APIs.

Developers are encouraged to write code that runs in the application sandbox, even if some restrictions for this sandbox make it behave a little differently than the browser model.

The sandbox bridge

At their simplest, HTML-based AIR applications that use network content consist of a root user interface (UI) file, the data and files they download, and some application files (in the application sandbox). However, if the external content and some user interface files operate in non-application sandboxes and the application files operate in an application sandbox, how does the non-application sandbox content and application sandbox content interact with each other?



The AIR solution is the *sandbox bridge* API, which enables indirect communication between different sandboxes. Developers can craft functions that call AIR APIs and then expose those "mediating" functions on the sandbox bridge. These bridge functions essentially "sit on the bridge" and wait for code in the non-application sandbox to call them.

This architecture accomplishes two goals. First, non-application sandbox content does not have direct access to the AIR APIs and application content is protected from external data which is unknown and potentially malicious. Second, only those APIs that the developer explicitly chooses to expose via the bridge mechanism are exposed.

While the sandbox bridge does not guarantee security, developers hold the keys for reducing risk in how they expose the AIR APIs to untrusted content. Much like a client-server model, the server should never trust the client, and the client should never trust data loaded from third parties. Likewise, developers should not use or trust code loading on the server as that could compromise the server, which in turn has access to the operating system. So for the client-server model in both the browser and in the AIR sandbox bridge, primitive system APIs, like `writeFile()` and `readFile()`, should always be protected from attack.

Adobe AIR security is designed with the web application development community in mind. Adobe supports standards and methodologies that help to mitigate security risks. Organizations such as the OpenAjax Alliance are further helping to publicize these best practices:

- Isolate untrusted content in a frame or `iframe`, as leveraging the same-origin policy makes it difficult for attackers to access the full DOM tree.

- Data loaded to a frame from a different domain should be given a unique JavaScript execution context and DOM tree.
- Do not generate and execute code dynamically.
- Avoid inserting HTML content that is not from a trusted source.
- Use JSON securely.

  AIR provides the mechanisms for developers to follow these best practices and thereby enhance the security of their HTML-based applications. For more information, see Ajax and Mashup Security at the OpenAjax Alliance.

  The developer must explicitly bridge sandboxes.

Summary of sandbox privileges

Capability	**Application Sandbox**	**Other (non-application) sandboxes**
Direct access to AIR APIs?	Yes	No
Access to application sandbox functions that use AIR APIs via the bridge?	N/A	Yes
Can load remote script such as `<script src='http://www.example.com/some_code.js'>`?	No	Yes
Can, by default, execute cross-domain requests (XHR)	Yes	No
Supports dynamically loading strings as code after the `load` event: `eval()` function, `setTimeout('string', milis)`, `javascript:` URLs, attribute handlers on elements such as `onclick='myClick()'`) that are inserted via `innnerHTML`, and so on.	No	Yes
Ajax frameworks will work without any changes?	Some frameworks	Yes

Moving ahead securely

The AIR security model aims for a backwards compatibility with legacy Ajax models while reducing the risk associated with exposing system APIs to data beyond the developer's control. In the browser client-server model, the browser handles much of the presentation and some dynamic third-party data loading. The server handles most of the sensitive processing and provides OS-like facilities such as storage, cryptography, and so on. By providing two sandboxes that communicate via indirectly exposed functions, Adobe AIR makes it less likely that attackers can access end-user machines and data.

Leveraging Ajax frameworks

Because a non-application sandbox is effectively the HTML browser sandbox, Ajax developers should find porting of existing applications and patterns to AIR relatively straightforward. For example, you could take an existing crossword game from the web and insert it entirely into a non-application sandbox frame.

While Ajax frameworks running in non-application sandboxes should run just as they do in a web browser, some framework versions may not completely work in the application sandbox if they rely on `eval()` and cross-domain code loading. It largely depends on how much a given framework depends on `eval()` and similar behaviors, as well as which parts of the framework the developer uses.

These restrictions do not prevent using `eval()` with JSON object literals. This allows your application content to work with the JSON JavaScript library. However, application sandbox content is restricted from using overloaded JSON code (with event handlers). For other Ajax frameworks and JavaScript code libraries, check to see if the code in the framework or library works within these restrictions on dynamically generated code. If they do not, you will need to include any content that uses the framework or library in a non-application security sandbox.

Migrating applications

What if you want to take that simple crossword game and implement functions like `saveGame()` and `loadGame()`? Doing so requires accessing AIR APIs, and you implement the relevant APIs in the application sandbox. Functions and data in the application sandbox are then exposed to the non-application sandbox via a sandbox bridge. At a high level, the process involves the following:

1. Rename the original root content file to `someName.htm`.
2. Create a file named `myRoot.htm` and include an `IFRAME` or `FRAME` that points to `someName.htm`.
3. Write functions that call the AIR APIs. It's safer not to call the APIs directly.
4. Create a `parentSandboxBridge` that exposes the application sandbox functionality to the non-application sandbox.

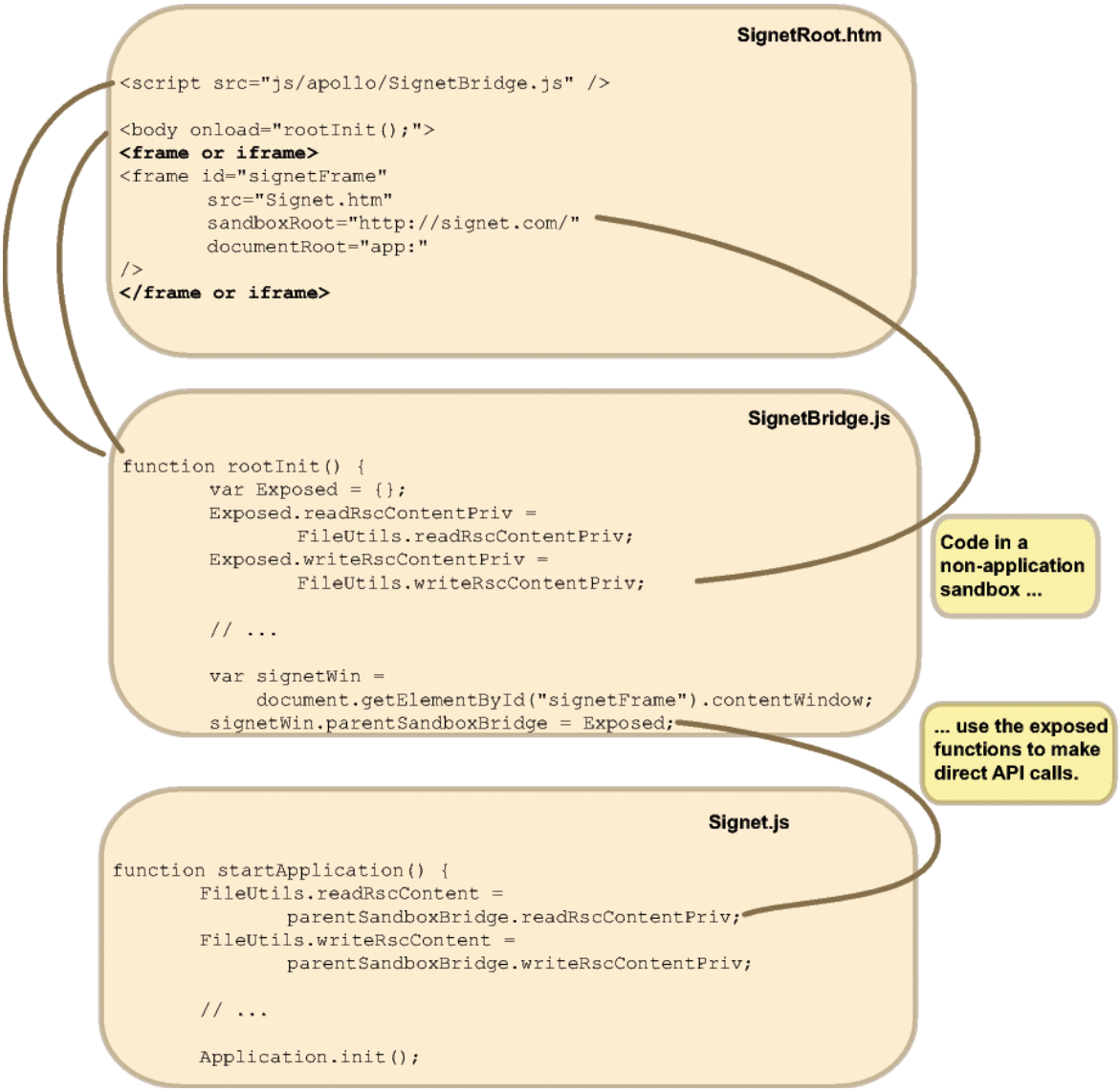
   Communication between the two sandboxes can be two way: `parentSandboxBridge` and `childSandboxBridge` enables traversing the bridge in both directions. This gives developers more control over content flow between the application and the application environment.

A real use case

Signet (a bookmark manager for del.icio.us users) is an HTML-based AIR application that implements the AIR security model in the following way:

1. First, `SignetRoot.htm` was created to contain a frame with id `signetFrame` and three required attributes that identify available domains.
2. In `SignetRoot.htm`, `onload` calls the `SignetBridge.js` which contains all the exposed functions.
3. In SignetBridge.js, the functions are exposed to `signetFrame` via `parentSandboxBridge`.
4. In `Signet.js`, the exposed functions make direct AIR API calls. For example, `writeRscContentPriv` calls `writeRscContent`.

While the methodology is relatively straightforward, the effect is powerful: remote and local domains are defined for two sandboxes, the bridge is started during onload, and the APIs are indirectly exercised through mediating functions via the bridge API.



Security: a shared responsibility

Ultimately, RIA security rests with everyone involved in the application life cycle. End users trust the developer to adhere to best practices and standards.

Aside from recommended development methodologies, security for AIR applications begins with installation on the user's computer:

- The initial installation workflow cannot be modified.

- Applications share a streamlined and consistent installation process that is consistent across all web browsers and operating systems.
- Installation is administered by the runtime which cannot be manipulated by the installed application.
- Installer files must be digitally signed so that users can verify the code's origin and determine the application's access privileges.
- The runtime provides memory management to minimize vulnerabilities such as buffer overflows and memory corruption.

Responding to any security concern ultimately adds to the developer's burden. However, the AIR security model attempts to reduce the complexity of writing and maintaining secure code.

Refer to the AIR developer documentation for a list of best practices. Among others, these include limiting the use of external files to those which are necessary, not using data from network sources for certain operations, and so on.