Security Guidance for Writing and Deploying Silverlight Applications

By Jesse Collins

[Overview 2](#_Toc223228900)

[Threat Modeling and the Security Development Lifecycle 2](#_Toc223228901)

[Background of Web Security 2](#_Toc223228902)

[Same-Origin Policy 2](#_Toc223228903)

[Cross-Site Scripting Attacks 3](#_Toc223228904)

[Cross-Site Request Forgeries (CSRF) 3](#_Toc223228905)

[A CSRF Mitigation: Nonces 4](#_Toc223228906)

[Silverlight and Web Security 5](#_Toc223228907)

[Changes from version 1.0 to version 2 5](#_Toc223228908)

[How Silverlight Works 5](#_Toc223228909)

[XAML 5](#_Toc223228910)

[XAPs 6](#_Toc223228911)

[The Silverlight Sandbox 6](#_Toc223228912)

[EnableHtmlAccess 7](#_Toc223228913)

[ExternalCallersFromCrossDomain 8](#_Toc223228914)

[Silverlight Networking 8](#_Toc223228915)

[Cross-Domain Policy Files 8](#_Toc223228916)

[How to Maximize Safety of Cross-Domain Access 9](#_Toc223228917)

[Trusting Third-Party Domains 10](#_Toc223228918)

[LANs and Security Zones 10](#_Toc223228919)

[Internet Explorer and the XDomainRequest Object 11](#_Toc223228920)

[Sockets 11](#_Toc223228921)

[FAQ 12](#_Toc223228922)

[How can I safely display a Silverlight ad on my Web site? 12](#_Toc223228923)

[Is it safe to load arbitrary XAML in my Web page? 12](#_Toc223228924)

[Is it safe to load arbitrary XAML from managed code? 12](#_Toc223228925)

[Is it safe to display arbitrary media in XAML? 13](#_Toc223228926)

[Is it safe to allow users to upload arbitrary XAPs to my Web site? 13](#_Toc223228927)

[How can I tell if a file is a Silverlight application? 13](#_Toc223228928)

[Is it safe to render XAML or run XAPs on my server? 14](#_Toc223228929)

[How can I make sure my XAP is loaded only from a specific domain? 14](#_Toc223228930)

[Is it safe to hide secrets in my XAP? 14](#_Toc223228931)

[Does the PasswordBox control protect the password in memory? 14](#_Toc223228932)

[Where can I find documentation for these features and APIs? 14](#_Toc223228933)

[Conclusion 15](#_Toc223228934)

[Special Thanks 15](#_Toc223228935)

# Overview

Silverlight has been designed with security in mind. In many situations, you can deploy Silverlight applications on your Web site without incurring much new security risk. However, there are scenarios where extra care and attention must be taken. For these situations, it’s imperative that Web developers and system administrators understand the nuances of the technology, and make informed decisions about deployment.

To keep the learning curve shallow, Silverlight attempts to conform to existing metaphors and expectations about Web security wherever possible. If you’re already familiar with Web security, you probably already have some expectations and assumptions about how Silverlight might behave in different situations. Ideally these expectations hold true, but since the security of your Web site may be at stake, it’s best to double-check and verify against the documentation. This document will help you do that.

This document helps you develop and deploy Microsoft Silverlight 2 applications that are secure as possible. There are other interesting Silverlight security topics, but this document focuses on the security of Silverlight deployments. It is not intended, for example, to provide advice to end users about installation and maintenance of Silverlight, or give insight into the security of the runtime itself.

First, this document covers some Web security basics like the same-origin policy, cross-site scripting attacks, and cross-site request forgery. Then, Silverlight-specific topics are discussed. At the end of this document is an FAQ-style section that addresses specific scenarios and concerns you might have.

## Threat Modeling and the Security Development Lifecycle

Every solution uses different technologies in different environments, and therefore has different security needs. It’s important that you have a holistic, structured approach to the security of your product. Microsoft recommends that you use a process similar to the [Microsoft Security Development Lifecycle (SDL)](http://msdn.microsoft.com/en-us/security/cc448177.aspx), which is the process under which Silverlight itself has been developed and hardened.

Among other things, the SDL requires that threat models be written to describe and scrutinize the security threats to your product. Part of this process is about keeping track of other technologies your solution uses, and verifying that the security capabilities of those technologies match your expectations.

# Background of Web Security

Security on the Web has a few subtleties and nuances that you should understand before deploying any Web-based technology. This section covers a few important ideas relevant to Web security before getting to Silverlight specifically. Feel free to skim this section if you’re already familiar with Web security.

## Same-Origin Policy

*Same-origin policy* (also called site-of-origin policy) is a Web security concept that there is an implicit trust level between resources that exist on the same domain. A Web page loaded from http://contoso.com/index.html is allowed access to other URLs on http://contoso.com, but not http://fabrikam.com. If the two URLs share the same domain name (and scheme and port number), they are said to have the same origin, or share a site-of-origin. You also might say that this is a *first-party* request.

In the Web browser, same origin means two things.

1. If an HTML page loads an **iframe** element (or other scriptable object) with a same-origin HTML page, either HTML page is allowed to interact with the DOM (document object model) of the other page.
2. An HTML page is allowed to make HTTP requests back to its site-of-origin (generally with the **XmlHttpRequest** or **XHR** object).

Silverlight adopts this notion of resources having access rights to the domain on which they are hosted. Put another way, in a Web page, the security boundary is the domain name.

## Cross-Site Scripting Attacks

A *cross-site scripting* (XSS) attack is an attack where a malicious attacker is able to inject code (generally JavaScript) in the context of a Web page on a domain other than his or her own.

As described in the previous section, Web browsers enforce the same-origin policy to prevent this from happening, but it is quite common for flaws in Web applications to unwittingly allow this to occur anyway. The most common XSS attack is *JavaScript injection*, where an attacker submits a specially-crafted payload to a server that allows users to post content on Web pages. For example, to attack a message board Web application, an attacker might submit a post with an embedded **script** element. When a user views this post, the JavaScript inside the **script** element will execute in the context of that page. This code has access to the page’s DOM, as well as, the victim’s cookies.

XSS may or may not be a security problem, depending on the needs of the Web site. For static Web pages with no JavaScript, XSS is not a concern because there’s not any attackable surface area. For other Web sites, it may be possible to perform defacements through a malicious payload in the query string. For still others, like the previous message board attack example, it may be possible to steal users’ cookies, which will often lead to complete compromise of that user’s account.

Since Web browsers, Silverlight, and other plug-ins respect the same-origin policy, today XSS attacks are primarily against server-side application code. However, throughout this document you’ll see places where a developer should be specifically concerned about XSS threats, and what the dangers might be. Since the security requirements for every site are different, it’s important that you carefully consider your own requirements and communicate them clearly.

## Cross-Site Request Forgeries (CSRF)

A *cross-site request forgery* (CSRF) is an attack where a malicious Web site makes an HTTP request to a target Web site in such a way that the Web site thinks that the request is part of a legitimate user’s session. In a typical CSRF attack, a user is logged into both the target Web site and the malicious Web site in different tabs or browser windows. When the malicious site sends a Web request to the target Web site, the Web browser puts the same cookies and HTTP authentication on that request as it would for legitimate traffic on that Web site. If the target Web site only uses cookies or session data to validate the request, the malicious application can make specific requests to the Web site in order to tamper with the user’s account. If a site uses persistent cookies liberally, the attack may be much easier because the user does not have to be logged in at the time.

Figure 1 illustrates such an attack. Here, the victim has two browser windows open, each with an active session to a Web site. One session is for http://contoso.com, and the other is for http://fabrikam.com. The Web page at http://fabrikam.com commits a CSRF attack by sending HTTP requests to endpoints at http://contoso.com. The danger is that http://contoso.com might consider that request to be part of the victim’s legitimate session.

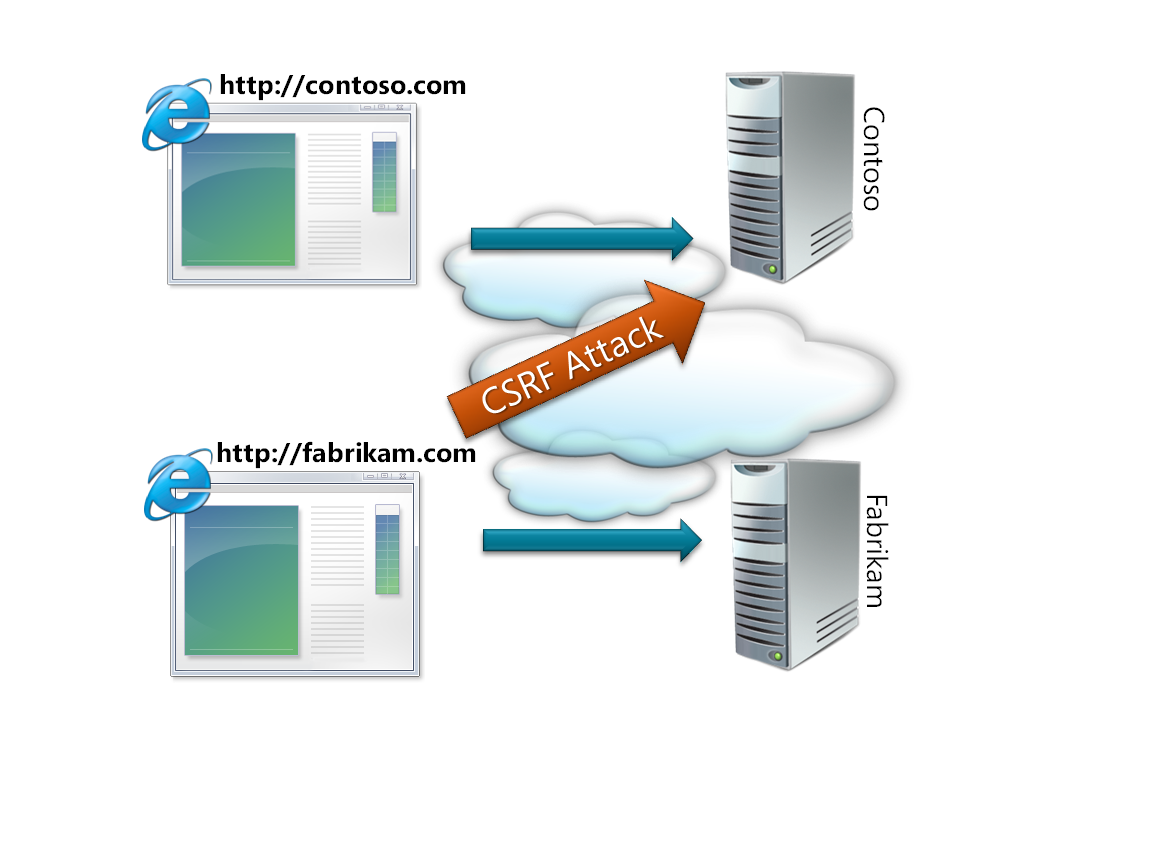


Figure - A cross-site request forgery (CSRF) is an attack where an HTTP request to a victim domain masquerades as legitimate traffic

There’s a key quirk Web browsers have that makes CSRF attacks possible: browsers attach cookies to requests regardless of the source of the code issuing that request. For example, if the Web page on http://fabrikam.com loads an image from http://contoso.com, the request to the contoso.com server will include cookies for contoso.com. So if there’s a Web application endpoint http://contoso.com/deletealldata.aspx that deletes all the data for that account, a Web page on another domain can invoke this endpoint by pointing an **img** element at it. And if the user happens to be logged in to contoso.com, deletealldata.aspx will receive valid cookies for that user, just as though it were navigated to the way the developer intended. What’s worse, the attacker can also make POST requests with arbitrary data against fabrikam.com by using just HTML or JavaScript.

Because the most popular Web browsers send cookies and HTTP authentication with third-party Web requests by default, Web developers must harden their applications against CSRF threats. But how can you tell if a request is legitimate? Sites used to rely on the HTTP **Referrer** header, but this has proven to be insecure. The key is that even though an attacker can make requests against your domain with, for example, **img** elements, there is no way for her to read the *responses* to those requests.

**Note** There are some tricks for reading some data in HTTP responses from other domains. For example, **script** elements or cascading style sheets (CSS) can be used to read data in HTTP responses, but these only work in fairly unusual situations.

### A CSRF Mitigation: Nonces

To mitigate CSRF attacks, many Web developers use a *nonce* (*n*umber used *once*) to distinguish legitimate traffic from fake traffic. The concept is pretty simple. When the Web application serves up a Web page with a form on it (or some other type of data submission code), it generates a nonce—a randomly generated token—and adds it to a hidden field in the form. When the form POST comes back from the Web page, the number that’s returned is compared against the nonce that was previously generated; if they match, the request is valid.

Nonces only work because Web browsers don’t allow JavaScript to read the responses from cross-domain requests. If they did, malicious code would simply issue the first request to get the form and read the nonce themselves.

In general, Web developers aren’t as familiar with CSRF as they are with XSS, and partly because of this many security researchers agree that CSRF vulnerabilities are a growing problem. Here, the specific context of the vulnerability is even more relevant than for XSS. Every Web application will have unique entry points, and so be susceptible to unique attacks. For some applications, CSRF may be a mild inconvenience; for others, it may be severe.

# Silverlight and Web Security

Now that you have some background in Web security, this section covers how Silverlight fits in with Web security. First, this section briefly discusses the way Silverlight 2 works, and then gets deeper into the Silverlight features that are especially relevant to Web security.

This section is meant only to give a cursory tour of some of the Silverlight fundamentals, and covers only what’s necessary for this discussion. For a more thorough treatment of these topics, see <http://silverlight.net>.

## Changes from version 1.0 to version 2

Silverlight version 1.0 was a great technology for loading vector and media content, but it provided no way to run code. The application developer would write JavaScript code and the Web browser itself would execute that code.

In Silverlight 1.0, the application developer sets the **source** property of the Silverlight plug-in to the location of the XAML to be rendered. This XAML must be from the same origin domain as the Web page. Because of this same-origin policy, and the fact that XAML itself contains markup and no code, the potential for security impact is fairly low. There is no security boundary in this case.

With the functionality added in Silverlight version 2, things are different. Developers can now set the **source** property to a XAP file, which may include managed code, in addition to markup and other resources. Additionally, it is possible to load XAP files from domains other than the origin domain.

## How Silverlight Works

Silverlight is a browser plug-in that can be loaded in Microsoft Internet Explorer and Firefox on computers running Microsoft Windows, and Safari and Firefox on Apple Macintosh computer. To create an instance of Silverlight on a Web page, just use an **object** element and set the **type** property to **“application/x-silverlight”**, as shown in the following code.

<object type=”application/x-silverlight-2” ></object>

The **object** element representing Silverlight is sometimes referred to as the plug-in interface. Once you have a Silverlight instance, you can load a XAML or XAP file by setting the plug-in’s **source** property.

### XAML

**XAML** (eXtensible Application Markup Language) is an XML-based markup language for describing visual content. It’s roughly analogous to HTML.

Unlike HTML, it is not possible for XAML to contain code, but it *can* set up event handlers to JavaScript functions on the hosting Web page. This means that if you’re loading XAML you don’t necessarily trust, you need to consider the possibility that it may contain event handlers that can call your JavaScript functions.

Also note that the Silverlight runtime makes no guarantees about CPU or memory consumption when loading XAML. Malicious XAML content may be able to lock up your Web page to the point that your users may find it unusable.

For these reasons, it is generally recommended that you do not load untrusted XAML into your Web page. However, if you’ve carefully considered the consequences, it may be safe to do so, depending on your particular application.

### XAPs

Silverlight 2 introduces a second type of content named *XAP* files. XAPs are a way to package managed code. Conceptually, a XAP is a Silverlight application—complete with all the XAML, managed DLLs, and other resources that applications tend to contain.

The XAP file format is nothing more than a ZIP file containing Silverlight content. A file named AppManifest.xaml exists in every valid XAP and describes the content of that XAP. When the plug-in’s **source** property is set to a valid XAP, the Silverlight runtime downloads the XAP, parses it, and reads the AppManifest.xaml to determine what managed DLL to load to start the application.

XAPs can be loadedfrom a third-party domain. When this is the case, they are called *cross-domain XAPs*. As described in the next section, Silverlight has specific rules governing the hosting of cross-domain XAPs.

When a XAP is cross-domain, the site-of-origin for the XAP is set to the domain from which the XAP was loaded, *not* the domain from which the page was loaded. So, if a Web page on http://fabrikam.com loads a XAP from http://contoso.com, the application is allowed to make Web requests to contoso.com, and not to fabrikam.com.

## The Silverlight Sandbox

When your page creates a Silverlight plug-in and sets the **source** property to load a XAP, this Silverlight application doesn’t necessarily run in the same security context as the rest of the page. Conceptually, the Silverlight application is running in a “sandbox”. This means that the Silverlight application is running in such a way that it is isolated from the HTML page, and that Silverlight is able to control access across this boundary. This is an important change from Silverlight 1.0, where the application’s JavaScript code had to run in the same scope as the HTML page.

To be consistent with other technologies on the Web, Silverlight piggybacks on the browsers’ **iframe** model when dealing with XAPs. If the XAP is same-origin, by default free communication is allowed between the page and the XAP. If the XAP is a cross-domain, by default no communication is allowed. This can be adjusted by setting the **EnableHtmlAccess** and **ExternalCallersFromCrossDomain** properties.

As shown in Figure 2, **EnableHtmlAccess** controls access to the page and **ExternalCallsFromCrossDomain** controls access to the XAP. These properties are described in more detail in the following sections.

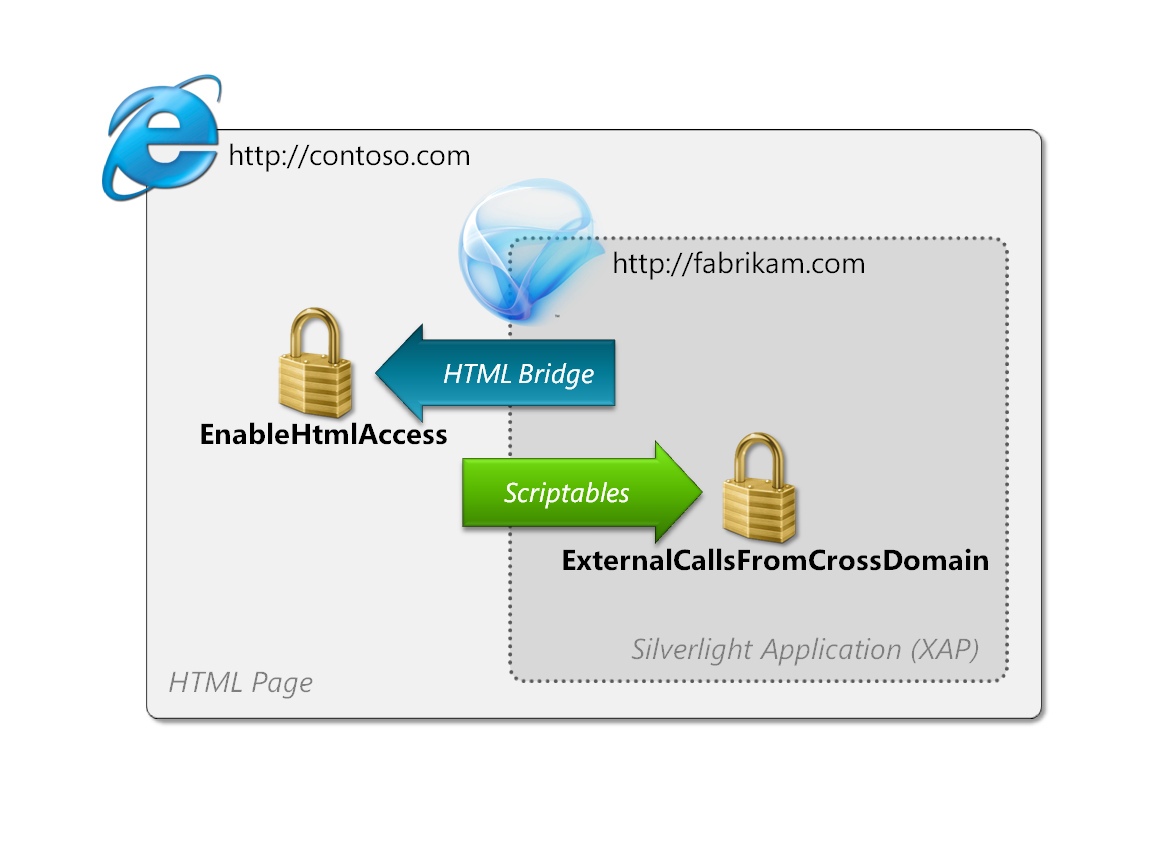


Figure - EnableHtmlAccess and ExternalCallsFromCrossDomain control access between the page and the XAP

**Note** If you haven’t already looked into the features of Silverlight 2, note that it includes some fairly rich functionality for communication between a XAP and its hosting page. Specifically, Silverlight’s [HTML Bridge](http://msdn.microsoft.com/en-us/library/cc645076(VS.95).aspx) feature allows XAP code to call directly into the HTML DOM from managed code.

### EnableHtmlAccess

By default, a XAP has access to the hosting HTML page if it is same-domain, but not if it is cross-domain. The [**EnableHtmlAccess**](http://msdn.microsoft.com/en-us/library/cc838264(VS.95).aspx) property on the plug-in allows the developer to control this behavior explicitly. You can set this property to true or false to get the behavior that you want.

By setting **EnableHtmlAccess** to true, you allow the XAP to arbitrarily manipulate the DOM from managed code. This is the moral equivalent to plopping a **script** element in the middle of your document. If the XAP is malicious, it’s an instant XSS vulnerability. The XAP can sniff your page, tamper with your page, read your user’s cookies, and perform Web requests to your domain on your user’s behalf. The attacker gets CSRFs as well.

So be careful when setting **EnableHtmlAccess** to true. Make sure that you trust the XAP that will be loaded. Consider what would happen if the remote domain expired and a domain squatter acquired it. If **EnableHtmlAccess** is true, the XAP you load will have the ability to sniff your page, deface your page, and steal cookies. As always, carefully consider the needs of your specific application and discuss concrete threats with your team.

### ExternalCallersFromCrossDomain

**EnableHtmlAccess** regulates the XAP’s access to the page, but the XAP also needs to be able to control access from the page. Just as it can be dangerous for a Web page to load an untrusted XAP, it may also be problematic for a XAP to be loaded into a hostile page.

If the page and the XAP are hosted on the same domain, JavaScript running on the page is allowed to access the XAP by making calls against the scriptable members the XAP exposes. In cross-domain cases, the default value of the [**ExternalCallersFromCrossDomain**](http://msdn.microsoft.com/en-us/library/system.windows.deployment.externalcallersfromcrossdomain(VS.95).aspx) property is **NoAccess**, which means that there are no entry points to your XAP. The only way a XAP can be scripted cross-domain is if the **ExternalCallersFromCrossDomain** attribute on the XAP’s deployment manifest (AppManifest.xml) is set to **ScriptableOnly**.

Take care when setting **ExternalCallersFromCrossDomain** to **ScriptableOnly**. Although **ExternalCallersFromCrossDomain** is generally not as dangerous as **EnableHtmlAccess**, there may very well be specific XAPs where access to scriptable members must be tightly controlled. Take a hypothetical case where domain A loads a XAP from domain B and calls certain scriptable members on it. Perhaps the XAP makes special calls back to the domain B server on behalf of domain A. But by enabling scriptable access, an HTML page on any domain can call these entry points.

At this time, Silverlight doesn’t provide a way to control access to your XAP based on the calling domain. If you open up access for domain A, you open up access to the world. Always be careful when making and allowing cross-domain calls.

# Silverlight Networking

In Silverlight 1.0, applications were only allowed to make Web requests back to the site-of-origin. Not only does Silverlight 2 allow applications to make Web requests to third-party domains, but it also allows applications to open TCP sockets as well. This section describes these features and accompanying security considerations.

## Cross-Domain Policy Files

As you learned earlier in the “Background of Web Security” section, it would be a breach of Web security rules for Silverlight applications to allow arbitrary Web requests against any domain. This would open serious CSRF problems because malicious Silverlight applications would be able to generate fake, malicious traffic against any session-based Web site.

What is needed is a way for Web sites to explicitly indicate that cross-domain traffic is safe, and what kind of traffic is safe. Silverlight’s approach is to give Web sites the ability to explicitly declare that their domain allows cross-domain traffic.

A domain can allow Silverlight applications to make third-party Web requests to it by publishing a clientaccesspolicy.xml file at its root. (For example, you can see Microsoft’s policy file at <http://microsoft.com/clientaccesspolicy.xml>). The Web site can specify which URLs are allowed to call cross-domain, as well as which domains are allowed to make those calls. If you own two domains—contoso.com and fabrikam.com—and you want to allow contoso.com to communicate with a Web service on fabrikam.com, you can publish http://fabrikam.com/clientaccesspolicy.xml that looks like the following.

<?xml version="1.0" encoding="utf-8"?>

<access-policy>

<cross-domain-access>

<policy>

<allow-from>

<domain uri="http://contoso.com"/>

</allow-from>

<grant-to>

<resource path="/mywebservice.aspx" />

</grant-to>

</policy>

</cross-domain-access>

</access-policy>

Without getting too bogged down in the syntax of this format, you can see what access is allowed fairly easily. Here, fabrikam.com is allowing cross-domain access to http://fabrikam.com/mywebservice.aspx, but only if the XAP requesting that resource has been loaded from http://contoso.com. For more information about allowing cross-domain access, see [Making a Service Available Across Domain Boundaries](http://msdn.microsoft.com/en-us/library/cc197955(VS.95).aspx).

To perform the cross-domain Web requests, you can use Silverlight’s **Downloader** object from JavaScript, or use the **WebClient** or **HttpWebRequest** classes from managed code. Silverlight will first request the policy file from the target domain, and verify that your request meets its requirements. Note that Silverlight caches the policy file throughout the lifetime of the plug-in.

### How to Maximize Safety of Cross-Domain Access

When you allow cross-domain access, your server-side code can now be called from a XAP on another domain—potentially a malicious XAP, depending on the domains to which you grant access. Since this XAP can actually read the HTTP response, traditional CSRF-prevention techniques, such as nonces, won’t work. This XAP is effectively indistinguishable from a real user at this point. Remember that all the user’s cookies and HTTP authentication is sent along as well, so the session will appear to be valid.

So when you open up an endpoint to be available cross-domain, you need to take a careful look at the way you use cookies and HTTP authentication in your application. If your application is complex enough, you may want to consider adopting a methodology like threat modeling. Take a close look at the information and functionality you expose via these endpoints, and any side-effects that might happen when they’re hit.

Here are a few ideas about how you might make sure that your endpoints are safe when exposed cross-domain.

* **Use static resources** – If your Web endpoint doesn’t have any code, there’s shouldn’t be much risk of CSRF. You may want to pre-generate your content.
* **Don’t use state** – CSRF relies on cookies or HTTP authentication. If you don’t use either of these, you shouldn’t have a problem. But remember that session information is generally tracked by cookies, so this means using session data is off-limits as well.
* **Use alternate authentication** – It might be possible to use some other kind of shared secret between the two domains to verify access. But remember that authentication protocols are very difficult to design correctly. Also remember that it is fairly easy for an attacker to reverse engineer your XAP, so be sure not to keep important secrets in your code. This approach is only advisable in the simplest of situations.
* **Verify the endpoint is innocuous** – Depending on your situation, it might be *desired* behavior that cookies and authentication are honored regardless of the requesting domain. If you’re not giving out the user’s private data, or taking some significant action on his or her behalf, allowing cross-domain access *may* have tolerable security risk. Again, every application is different, and has different security needs.

When you allow cross-domain access to a Web site, it’s possible to open up the whole domain rather than just one endpoint. This is generally not recommended. It’s usually much safer if you can scope access down to a particular Web page or Web service. This allows you to audit and scrutinize one specific piece of code for CSRF threats, and frees you from examining every resource on the domain. Even if there’s no problem today, perhaps someone else will add a resource tomorrow that hasn’t been properly hardened.

If you must open up cross-domain access, you may want to consider creating a new sub-domain to contain this policy. For example, you could set up a sandboxed domain at http://api.fabrikam.com. This confines the cross-domain access to this specific sub-domain, and will hopefully serve as a reminder to future colleagues that code on this domain is accessible cross-domain. For even greater security, you might consider hosting this API on a domain with a different top-level domain, for example, http://api.fabrikam.*net*. This ensures that cookies won’t be shared between the two domains.

**Note** Remember that http://fabrikam.com and http://api.fabrikam.com are treated as completely different domains.

### Trusting Third-Party Domains

Where possible, it is recommended that you restrict cross-domain access to the specific domains that you expect to be using your service. If your Web site or Web service is only meant to be called from an application on http://fabrikam.com, you might as well just allow access to that domain, rather than the whole world.

However, if a domain in your list has an XSS vulnerability, you may be inadvertently exposing your endpoint to any site that wants to call it.

Suppose a fabrikam.com Web page has a query-string XSS vulnerability that allows an attacker to run JavaScript code in the context of that page. Assuming the attacker can get a user to visit her malicious site, the attacker can navigate the page to the fabrikam.com page and inject her malicious JavaScript. The JavaScript that the attacker injects creates a Silverlight object and makes a Web request against your domain. Since your clientaccesspolicy.xml file indicates that fabrikam.com is allowed to make this request, and the request is in fact coming from fabrikam.com, the call will succeed.

**Note** Remember, it’s possible to write JavaScript-only Silverlight applications that don’t require XAPs.

So, when you trust a specific domain, you’re also trusting that it doesn’t have any XSS vulnerabilities. It’s probably wise to assume they exist, and write your code as though you expect any Web site to be able to make requests against your application.

### LANs and Security Zones

Even if an endpoint doesn’t rely on cookies or authentication, you still run the risk of greater exposure due to *physical* access. If your Web application is behind a network address translator (NAT), firewall, or Web proxy, your clientaccesspolicy.xml file may be exposing your Web endpoint to the Internet, even if you didn’t want it to. If you run a Web application on your company’s local network—and allow wide-open cross-domain access—a Silverlight application from any Web site is now able to make requests against your domain.

For example, suppose an internal line-of-business Web service is running on http://humanresources on your corporate network. Perhaps someone adds a clientaccesspolicy.xml file so that internal Silverlight applications can access endpoints on it. But if it allows access from any domain, this means that Web pages on the *Internet* can make request to these endpoints as well. So even though an attacker cannot access http://humanresources directly, because the technology your IT department has put in place to prevent it, the attacker may be able to access it from one of your employees’ computers with a Silverlight application.

Silverlight provides a “speed bump” to help limit exposing your local Web applications to the Internet, but it isn’t fool-proof. The speed bump involves security zones and works only in Internet Explorer on Windows.

Internet Explorer has the concept of *security zones* to help categorize security configurations. To view the security zones in Internet Explorer, click the Tools menu, click Internet Options, and then click the Security tab. Users can establish different security rules depending on the source of the Web content.

For Silverlight, you’re primarily concerned with the difference between the Internet zone and the Local intranet zone. The Local intranet zone is meant to describe your local area network (LAN). A LAN represents computers that aren’t directly exposed to the Internet and are often behind a network address translator (NAT) or firewall. The Internet zone, of course, describes the servers on the Internet.

So, to help protect intranet Web applications from malicious sites on the Internet, Silverlight does not allow a Silverlight application from the Internet zone to access an endpoint in the Local intranet zone.

But unfortunately, security zone determination is not fool-proof, because it is really impossible to infer the network topology from a single client in a general way. So do not rely on this behavior, even if it seems to work for your organization. You simply cannot guarantee this safeguard will work on every network. Also note that this logic isn’t available on all platforms.

Write your clientaccesspolicy.xml files to assume Internet-based Silverlight applications may be allowed to make HTTP requests to your servers. If you put an overly-permissive clientaccesspolicy.xml file on http://humanresources, for example, you may be inadvertently allowing malicious internet applications to make requests against that server.

**Note** In order to accommodate pre-existing Web services and endpoints that already support Adobe Flash, Silverlight also supports a subset of Flash’s crossdomain.xml format. For more information, see [HTTP Communication and Security with Silverlight](http://msdn.microsoft.com/en-us/library/cc838250(VS.95).aspx).

### Internet Explorer and the XDomainRequest Object

One easy way out of all these CSRF concerns is to take a look at Internet Explorer 8, and its new **XDomainRequest** (XDR) object. This object also allows cross-domain HTTP access, with some added constraints to prevent the kinds of CSRF problems discussed earlier.

Among other things, XDR requests are anonymous, meaning that they don’t include cookies or HTTP authentication. This means your existing Web endpoints continue to work as safely as before, “for-free”, without additional hardening in most cases.

## Sockets

Silverlight 2 also introduces the ability to establish TCP sockets. Sockets can be established to both the site-of-origin and to third-party domains that have explicitly allowed access through their clientaccesspolicy.xml files. As described in previous sections, take care when configuring access, and expose as few resources to as few consumers as possible. The following shows an example of a policy file that allows socket connections.

<?xml version="1.0" encoding ="utf-8"?>

<access-policy>

<cross-domain-access>

<policy>

<allow-from>

<domain uri="http://contoso.com"/>

</allow-from>

<grant-to>

<socket-resource port="4510" protocol="tcp" />

</grant-to>

</policy>

</cross-domain-access>

</access-policy>

Since sockets operate at a lower level than HTTP, the threats are a little different and more dependent on your specific environment. With sockets, CSRF is no longer so much of a factor because there’s no HTTP involved and there’s no cookies or HTTP authentication either. You can still use the socket APIs to craft a valid HTTP request, but you’ll have to set the cookies yourself.

The bigger danger with sockets is that you might accidentally allow access to some other important network application running on your server (or some other network appliance that might be sniffing traffic). In order to reduce the likelihood of port collisions like this, Silverlight restricts sockets to use ports only in the range 4502-4534. However, before deploying your clientaccesspolicy.xml file, make sure nothing else in your environment will be confused by traffic on these ports.

**Note** For sockets, Silverlight makes no attempt to restrict access based on security zone, so make sure that you’re not exposing resources on your local network to the whole Internet.

# FAQ

This section provides direct answers to specific scenarios that you may encounter.

## How can I safely display a Silverlight ad on my Web site?

It’s best to sandbox and lock down the Silverlight ad as much as possible. It’s difficult for ad providers to recognize ads that do malicious things, such as automatically navigating to a Web page or tampering with the surrounding page.

To display a Silverlight ad, it is recommended that you set **EnableHtmlAccess** on the plug-in to false, and **AllowHtmlPopupWindow** to true. This prevents the ad from performing arbitrary navigations and DOM interactions, but still allows it to pop up a new window if and only if the user clicks it. This means that the ad can’t automatically navigate to another page. Also, it means that the user has some idea of which entity on the page initiated the navigation because he or she has to click it.

## Is it safe to load arbitrary XAML in my Web page?

By setting the plug-in’s source to a XAML file, you can just load XAML content rather than a whole Silverlight application (XAP). This was the only content that Silverlight 1.0 supported through the **source** property.

XAML cannot contain any script or executable code, but note that Silverlight does not make any guarantees about CPU or memory consumption. A malicious XAML file may be able to render your Web application unresponsive.

Additionally, Silverlight will attempt to resolve event handlers in the XAML to JavaScript functions on the hosting page, including the global functions. For example, loading the following XAML will call the page’s **alert** function, which shows a message box.

<Canvas xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" Loaded="alert"/>

This XAML sets the canvas’ **Loaded** event to call the **alert** function on the hosting page. This technique could be used to call any function on the hosting page, although the attacker has very little control over the parameters passed to the function.

Since it’s impossible to be sure that there will never be a dangerous top-level function in your page, the best thing to do is to not load untrusted XAML. However, in some applications, loading untrusted XAML may be a worthwhile tradeoff. Just make sure you understand the potential risks.

## Is it safe to load arbitrary XAML from managed code?

Using Silverlight’s managed **XamlReader** class, you can easily load XAML by calling the **XamlReader.Load** method on a XAML string, and then adding the resulting object to your visual tree.

Even though XAML doesn’t contain any executable code, Silverlight doesn’t enforce any CPU or memory limitations when loading XAML. Therefore, if it’s critical that your application remains responsive, you’ll want to avoid loading untrusted XAML.

Also realize that XAML has the power to create managed types, and to invoke any public property setters on those types. If you have a Silverlight application named SilverlightApplication1, which has a class named MyClass, which has a property named MyProp, the following XAML will create an instance of MyClass and set the MyProp property when loaded.

<app:MyClass MyProp='set this string'

xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'

xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml'

xmlns:app='clr-namespace:SilverlightApplication1;assembly=SilverlightApplication1' />

At this time, it is not recommended to use **XamlReader.Load** to load untrusted XAML content. Silverlight currently does not yet provide a way to cleanly load untrusted XAML.

## Is it safe to display arbitrary media in XAML?

Images, video, and audio are generally safe to display, with the caveat that Silverlight makes no guarantees about CPU and memory consumption. It may be possible for malicious images or video to consume enough system resources to make the user’s computer unresponsive.

## Is it safe to allow users to upload arbitrary XAPs to my Web site?

Recall that the site-of-origin for a XAP is the domain from which the XAP was loaded—not the domain on which it is displayed. This means that a XAP hosted on http://fabrikam.com may make any arbitrary HTTP request it wants back to http://fabrikam.com. This is the same-origin principle at work. So if the XAP is malicious, it can easily make CSRF attacks against this domain, even if the page loading it is from contoso.com.

To make matters even more interesting, remember that a XAP file is just a ZIP file with specific contents. Microsoft Word’s DOCX format is based on ZIP as well. This means that it is possible to construct a file that is both a valid DOCX file *and* a valid XAP.

To mitigate this problem, the Silverlight runtime needs to make sure that the Web server realizes the content its serving will be interpreted as a XAP. To accomplish this, Silverlight requires that XAPs have a MIME-type of **“application/x-silverlight-app**” when they’re loaded cross-domain. If this MIME type is not set, the XAP will fail to load.

This may still be an issue for you if you happen to be hosting Silverlight applications on a domain where you also want to allow your users to post content. In this case, you may need to write some code to sniff the upload. For more information, see the next question.

So it may be possible to safely host untrusted XAPs, if you’re in either of the following situations.

* Your Web server does not set the **“application/x-silverlight-app**” MIME type.
* You have no Web content on that domain that may be subject to CSRF threats.

## How can I tell if a file is a Silverlight application?

As mentioned in the previous question, there may be security risks associated with allowing users to upload XAPs to your Web site. You may find yourself in a situation where you need to filter user uploads and sniff them to see if they look like Silverilght applications.

Every valid Silverlight application is a ZIP archive packaged with the deflate algorithm. There also must be a file named AppManifest.xaml at the root of that ZIP archive. If the file in question is either not a ZIP archive, or does not contain an AppManifest.xaml entry at the root of the archive, you can safely conclude Silverlight will not interpret it as a XAP.

It is important to use a ZIP library that is hardened against malicious input when processing user-uploaded files. The simple act of uncompressing an archive is a deceptively complex operation.

## Is it safe to render XAML or run XAPs on my server?

The key problem with rendering XAML on the server is the threat of denial-of-service (DoS). Since it may be possible to construct a malicious XAML file that makes Silverlight consume large amounts of memory and CPU cycles, it may not be appropriate for server environments at this time. You might also consider running Silverlight in a separate process in order to enforce quotas.

## How can I make sure my XAP is loaded only from a specific domain?

At this time, Silverlight does not provide a mechanism for enforcing that a XAP is loaded only by a particular domain.

When **EnableHtmlAccess** is false and the XAP cannot access the hosting page, the XAP is not able determine the origin domain of its hosting page. If you need functionality like this, you may need to require that the loading page set **EnableHtmlAccess** to true, and write code in your XAP to verify that it’s being hosted on the correct domain.

To get the name of the hosting domain, validate the value of the **HtmlPage.Document.DocumentUri** property. Other methods of checking the hosting page may be prone to spoofing.

To determine the domain, you can also check the HTTP **Referrer** header on the server side before providing the XAP to the client. This does not prevent people from manually downloading the application, but will hamper other Web sites from simply embedding your application on their pages.

## Is it safe to hide secrets in my XAP?

No. Silverlight 2 doesn’t support any encryption of XAPs. A XAP file is simply a ZIP archive containing DLLs, XAML, and other resources.

You may consider using a managed obfuscator, such as Dotfuscator, to make it more difficult to extract data from your DLLs, but be warned that this is just a speed bump and is not fool-proof.

## Does the PasswordBox control protect the password in memory?

No. Silverlight 2 doesn’t make any effort to encrypt or obfuscate the user’s password in memory. Note that Windows Presentation Foundation (WPF) does have a mechanism for this in its **SecurePassword** property, which uses the **SecureString** class.

## Where can I find documentation for these features and APIs?

Here are some links to Silverlight documentation that may help you with securing your applications. Many of these links were referenced in this document.

**Making a Service Available Across Domain Boundaries**: <http://msdn.microsoft.com/en-us/library/cc197955(VS.95).aspx>

**Working with Sockets**: <http://msdn.microsoft.com/en-us/library/cc296248(VS.95).aspx>

**Cryptographic Services in Silverlight**: <http://msdn.microsoft.com/en-us/library/cc265159(VS.95).aspx>

**EnableHtmlAccess**: <http://msdn.microsoft.com/en-us/library/cc838264(VS.95).aspx>

**ExternalCallersFromCrossDomain**: <http://msdn.microsoft.com/en-us/library/system.windows.deployment.externalcallersfromcrossdomain(VS.95).aspx>

**AllowHtmlPopupWindow**: <http://msdn.microsoft.com/en-us/library/cc974117(VS.95).aspx>

**DocumentUri**: <http://msdn.microsoft.com/en-us/library/system.windows.browser.htmldocument.documenturi(VS.95).aspx>

# Conclusion

Silverlight has been designed with security in mind. To enable some important scenarios—especially scenarios involving cross-domain communication—Silverlight has some features that are possible to use insecurely. By understanding the common threats in the Web space and the ways these threats relate to Silverlight, it should be relatively simple to create and deploy Silverlight applications with minimal security risk.

# Special Thanks

A number of volunteers have reviewed this document to help ensure accuracy and readability. Contributors include Marco Matos, Reid Borsuk, Sam George, Mark Shlimovich, Spencer Low, Ben Pryor, and Robert Lyon.