Deconstructing ColdFusion

Chris Eng and Brandon Creighton

July 29, 2010

Contents

Abstract

This paper contains practical advice for both developers and security practitioners for identifying security vulnerabilities in ColdFusion applications. We provide a survey of ColdFusion architecture, including request lifecycle, error handling, and variable scoping, then uses code samples to illustrate how these architectural features affect common vulnerability patterns. Finally, we explore the composition of compiled ColdFusion Java classes as well as ColdFusion's proprietary classfile format.

Introduction

ColdFusion Background and History

ColdFusion was originally developed in 1995 by Allaire Corporation (Wikipedia contributors, 2010). At the time, websites were comprised mostly of static content. CGI scripts could be used to implement basic application logic, but the skill level required to write a Perl or shell script was beyond the capabilities of most web designers, who only knew HTML. Part of the motivation for ColdFusion was to provide a mechanism for web designers to create more complex applications, including the ability to connect an HTML page to a backend database. They created a custom language called the ColdFusion Markup Language (CFML), which resembled HTML but added features typically seen in programming languages – variables, loops, if/then statements, etc.

ColdFusion 1.0 was only available on Windows and had its own custom web server, but as the product evolved, interoperability became more of a focus. ColdFusion 6, released in 2002, was rewritten entirely in Java to be compatible with the increasingly popular J2EE standard. All ColdFusion files were transparently compiled to Java classes at runtime, and developers had the option of packaging their ColdFusion applications as a WAR or EAR file, enabling it to be deployed on any J2EE application server. Macromedia, who had acquired Allaire a year earlier (Driver, 2001), bundled the JRun application server (another of their acquisitions) with ColdFusion 6.

In April 2005, Adobe acquired Macromedia (Adobe Systems, Inc., 2005). Unsurprisingly, in subsequent releases, ColdFusion has added features enabling integration with Flash, Flex, AIR, and other prevalent technologies. Adobe's website cites more than 777,000 developers and 12,000 companies using ColdFusion today, with a client list that includes Cisco, Verizon, Boeing, Citi, eBay, and the Department of Homeland Security (Adobe Systems, Inc., 2010).

Aside from Adobe's customer list, there is scant detail on the ColdFusion user population. One study suggests that the heaviest concentration of ColdFusion applications lies in the education, healthcare, and retail sectors (WhiteHat Security, Inc., 2010). Anecdotally, we are also aware that US federal and state government agencies also use ColdFusion to a reasonable degree.

Motivations

The primary motivation for publishing this research is to provide a practical resource for developers and code auditors. ColdFusion has received surprisingly little application security coverage, despite the fact that the technology has existed for 15 years, has had nine major releases, and is still being actively marketed and maintained.

There have been numerous papers and presentations addressing ColdFusion security, but most of them neglect to cover application security in any depth. Instead, they focus on enumerating the infrastructure footprint and describing server deployment issues such as restricting administrative interfaces and execution privileges (Davis, 2006). Presentations of this nature are extremely useful to penetration testers, but they are less beneficial to ColdFusion developers and code auditors.

Adobe's official whitepaper on securing ColdFusion 8 applications briefly addresses a few common web vulnerabilities, but the bulk of the documentation is used to describe authentication methods and other security features (Lee, Melven, & Sargent, 2007). In fact, the section on injection vulnerabilities is only two pages, and there is no dedicated section on cross-site scripting (XSS), one of the most widespread web application vulnerabilities for the majority of the past decade. The ColdFusion 9 developer guide, covering the most recent version, contains a chapter called "Securing Applications", but most of the content is lifted directly from the aforementioned ColdFusion 8 whitepaper.

There was a short-lived effort through the OWASP organization to create an Enterprise Security API for ColdFusion developers (Dean & Shelton, 2009), but this project appears to have been abandoned after roughly one month and was never formally released.

Platform Architecture and CFML

The Broad View

ColdFusion remains fairly true to its 1990s roots as a system comprised of individual dynamic pages. There are two primary types of files parsed by the engine: .cfm files ("pages") and .cfc files ("components"). There is a direct, straightforward mapping between HTTP requests and files; a request to a ColdFusion server for /foo.cfm results in the app-relative file named foo.cfm being loaded and executed. In addition, it is possible to run other code at certain stages of a request's lifecycle through use of the Application.cfc or Application.cfm files.

Pages and components are written in CFML, a set of HTML-style tags that perform various functions. Content outside of these tags is written to the response body, as in ASP or PHP. Additionally, there exists a separate language for writing code, called CFScript. This is a procedural, JavaScript-esque language that can access built-in and user-defined functions and objects. Many tasks can be accomplished in either CFML or CFScript; however, some functionality provided by CFML tags isn't easily available in CFScript.

In addition to being invoked directly, pages may be included using the `<cfmodule>` or `<cfinclude>` tags. Pages can also be embedded inside other code using custom tags; for example, `<cf_foo>` is a custom tag that is implicitly translated to `<cfinclude template="foo.cfm">` except that when using the shorthand notation, changes made to variables are not visible in the calling page. Components, on the other hand, provide the ability to create sets of functions to be used by other code, with the option of data sharing through instances of a component-typed object.

Both pages and components may define functions using the `<cffunction>` tag or through a separate syntax in CFScript. These are called UDFs, or user-defined functions, in the ColdFusion world. UDFs defined in pages cannot be called externally (unless included with `<cfinclude>`); however, components' functions can be made accessible to other code.

Page Lifecycle

Before a page or component is executed, the server first searches for and executes a file named either Application.cfc or Application.cfm. Application.cfc takes precedence, if both are present. The server searches the directory of the requested file first; if neither Application.cfc nor Application.cfm is present there, it traverses upwards until it reaches the top-level directory defined by the server, stopping if one is found. These files are used to set application- and directory-wide variables, configure application settings, provide content filtering, route requests dynamically, provide authentication, and handle errors.

Application.cfc allows users to define specially-named UDFs to be called at certain points in an application's life cycle. The list is below:

- `onApplicationStart()`: run once when an application starts, before the first request. It can access that request's variables and write output to its response.
- `onApplicationEnd()`: run when an application is explicitly shut down. This is not associated with any request.
- `onSessionStart()`: run when session handling is enabled and a new session starts. Associated with request.
- `onSessionEnd()`: run when a session times out or is cancelled. Not associated with request.
- `onRequestStart()`: called before every request. Associated with request.
- `onRequest()`: called after onRequestStart code ends. Associated with request.
- `onRequestEnd()`: called after request has been processed. Associated with request.
- `onMissingTemplate()`: called when an unknown page has been requested. Associated with request.
- `onError()`: called when an uncaught exception occurs. This may or may not be associated with a request.

Aside from Application.cfc, there are other ways to run code at some of these events:

- If Application.cfm exists and is chosen, and a file named OnRequestEnd.cfm exists in the same directory as that Application.cfm, OnRequestEnd.cfm is run after a request.
- The `<cferror>` tag provides error handling similar to the `onError()` function; this tag may be specified anywhere, including Application.cfm.

Variables and Expressions

ColdFusion variables exist in dynamically-scoped namespaces. These namespaces are called “scopes” by ColdFusion documentation. Scopes and variables are case-insensitive. Users can explicitly reference a variable by prefixing its name with the name of its scope. In this example, the URL scope contains the set of GET variables for a request; this XSS-laden code outputs the name of the GET variable `myvar` directly.

```
<cfoutput>
	variable: #URL.myvar#
</cfoutput>
```

However, it is not necessary to fully-qualify all variables. When an unscoped variable is referenced, the runtime searches several scopes for a variable of the same name. This can be handy for developers but can lead to security problems; see the next section for more details.

A list of common scopes is below. This list is not complete; there are others that exist only in certain contexts.

- Variables: The binding stack of variables local to the currently-running page or component. Unqualified variables bound using `<cfset>` or via CFScript are located here, as are the return values of many CFML tags.
- Application: Application-wide variables.
- Arguments: Named arguments to a UDF.
- Attributes: The set of attributes passed to a custom tag invocation.
- Caller: Available within custom tags; this references the calling page’s Variables scope
- Request: Persistent and global across all code for the lifetime of a request. This is **not** populated with request variables like PHP’s $_REQUEST; instead, it facilitates data sharing across complex meshes of components, custom tags, and included pages.
- This: Members of a structure or component instance.
- URL: HTTP GET variables.
- Form: HTTP POST variables.
- Cookie: HTTP cookies.
- CGI: The set of CGI variables; however, as noted below, some variables in this scope may be manipulated by HTTP header values!
- Server: Persistent and global across all applications running on the server.
- Session: Persistent across the lifetime of a session.
- Client: Persistent across all requests by a given HTTP client; keyed off of persistent cookies. May be stored client-side or server-side (configurable).

Variable Typing

Most ColdFusion variables belong to one of two type categories: simple or complex (for a better reference, see Lichtin, 2010). Simple variables have weak dynamic typing. Predominantly, simple types exist as strings; implicit conversions are used whenever a value of a different type is required. Built-in functions as well as UDFs can have type requirements, but are not required to. Virtually all operators require types; for example, the + operator performs arithmetic addition while the & operator performs string concatenation. Conversions make this transparent to users, resulting in code like:

```
<cfset x="3" + "4">     <!--- x is 7 --->

<cfset y=3 & 4>         <!--- y is "34" --->
```

Complex variables are containers; they have members and/or indices. Examples are arrays, structures, open files or directories, SQL/LDAP queries, and other special-function objects.

There is also a set of opaque types; these include proxies to Java/COM/.NET objects, binary data, and other types whose values can only be manipulated through special functions or custom CFML tags.

Building on its dynamically-scoped foundation, ColdFusion allows users to calculate variable names for operations like assignments at runtime. In CFML, this is often facilitated with the use of string interpolation; in CFScript, some functions take variable names as values. Here are CFML and CFScript examples of assignment with dynamically-calculated variable names:

```
<cfset "#z#"="foo">       <!--- z contains variable name --->

SetVariable(x, "bar");    <!--- x contains variable name to be set --->
```

ColdFusion also has mechanisms for dynamic evaluation of expressions. The primary one are the `Evaluate()` and `PrecisionEvaluate()` functions; in addition, conditional dynamic evaluation is available with the `IIF()` function.

Practical Tips for Developers and Security Practitioners

In this section, we will provide specific, practical guidance for those trying to build or break ColdFusion applications. We start by enumerating the sources of untrusted data, and we demonstrate ColdFusion coding patterns that lead to cross-site scripting and SQL injection, two of the most prevalent web application vulnerabilities. We finish with a discussion of some vulnerability classes that are unique to ColdFusion, including methods for exploiting unscoped, undefined, and persistent variables.

Sources of Tainted Data

The most common pattern for injection-style web vulnerabilities involves data from an untrusted source being used by the application without properly sanitizing or encoding it first. In order to detect these vulnerabilities, it's important to understand where the untrusted (or "tainted") data originates. Earlier, we described the different variable scopes that exist in ColdFusion. Certain scopes *always* contain tainted data, others *sometimes* contain tainted data.

- Always tainted
    - URL.any_variable
    - FORM.any_variable
    - COOKIE.any_variable

    - FLASH.any_variable
    - CLIENT.any_variable
      (only when client variables are enabled and storage is cookie-based)
- Sometimes tainted
    - CGI.some_variables (e.g. CGI.PATH_INFO, CGI.QUERY_STRING, etc.)
    - SESSION.some_variables (the application logic *may* copy tainted data into this scope)
    - REQUEST.some_variables (the application logic *may* copy tainted data into this scope)
    - CFFUNCTION arguments (when access="remote")

For any other scopes not listed, it is generally safe to assume that the data is not tainted. For unscoped variables, look for an assignment associated with each usage of the variable; if no assignment can be found, assume the value is tainted.

Cross-Site Scripting

Cross-site scripting (XSS) is pervasive in ColdFusion applications. The canonical example of XSS is using tainted data to construct an HTML page without properly encoding that data. For example, the following ColdFusion code would output the value of the POST variable `search`:

```
<cfoutput>
      You searched for: #FORM.search#
</cfoutput>
```

If the user were to enter JavaScript code, e.g. `<script>alert("xss")</script>`, into this field, the JavaScript would execute in the context of the user's web browser. In this case, the developer should have HTML encoded the tainted value before displaying its value.

XSS is easy to defend against provided developers are given appropriate tools and guidance. With ColdFusion, unfortunately, developer guidance is often misleading and/or incomplete.

The Problem with scriptProtect

The Adobe guidelines for writing secure ColdFusion applications states that using a mechanism called scriptProtect can "thwart most attempts of cross-site scripting" (Lee, Melven, & Sargent, 2007). scriptProtect can be enabled at the application level or the server level, and it works by inspecting variables in certain scopes (any combination of FORM, URL, CGI, and COOKIE) for strings commonly used in XSS attacks. It replaces blacklisted strings such as `<script` or `<object` with the string `<InvalidTag`, thereby nullifying certain attacks.

Going back to the canonical XSS example, adding scriptProtect would look like this:

```
<cfapplication scriptProtect="all">
<cfoutput>
      Your search: #FORM.search#
</cfoutput>
```

Requesting this page with POST data containing `search=<script>alert("xss")</script>` would return the following response:

```
Your search: <InvalidTag>alert(”xss”)</script>
```

Because `<script` was replaced with `<InvalidTag`, the resulting string no longer causes JavaScript execution.

Unfortunately, this mechanism only protects against the simplest XSS attacks. Because scriptProtect is blacklist-based, it can be easily circumvented by choosing an attack that includes none of the blacklisted values. For example, consider requesting the same page with POST data containing `search=<img%20src=”http://i.imgur.com/4Vp9N.jpg” onload=”alert('xss')”>`, which would return the following response:

```
Your search: <img src=”http://i.imgur.com/4Vp9N.jpg” onload=”alert('xss')”>
```

In this scenario, the attack string does not match the blacklist, so it is allowed through untouched. The JavaScript code is executed immediately after the image loads.

The blacklist used by scriptProtect can be extended by editing a file called neo-security.xml. By default it uses the following regular expression:

```
&lt;\s*(object|embed|script|applet|meta)
```

However, it is impossible to enumerate every possible combination of characters that might be used in an attack, which is why blacklists are not considered an effective defense against XSS. Enabling scriptProtect doesn't make an application any less secure, but the level of protection it provides creates a false sense of security for a typical developer.

Insufficient Built-in Encoding Functions

The best practice for eliminating XSS is to use contextual encoding/escaping depending on how the untrusted data is being used (e.g. in the document body, within a script tag, inside an attribute, etc.). The OWASP XSS Prevention Cheat Sheet (OWASP Foundation, 2010) covers this in detail and will not be reproduced here.

ColdFusion provides two built-in functions for performing HTML encoding: `HTMLEditFormat()` and `HTMLCodeFormat()`. These functions replace the <, >, ", and & characters with their HTML entity equivalents; `HTMLCodeFormat()` also wraps the entire string in a <pre> block. The limitation of these functions is that they are completely ineffective against unquoted or single-quoted tag attributes, or within existing script tags. For example, here are some code snippets that use HTMLEditFormat() to encode tainted data yet remain vulnerable to XSS:

```
<img #HTMLEditFormat(URL.foo)#>

<img alt='#HTMLEditFormat(URL.foo)#'>

<script>#HTMLEditFormat(URL.foo)#</script>

<script>var x='#HTMLEditFormat(URL.foo)#';</script>
```

There is another built-in function called `XmlFormat()` that is identical to `HtmlEditFormat()` but also encodes the single quote character. This provides better protection but still won't prevent XSS in all situations. Despite this, `XmlFormat()` will always be more effective than `HtmlEditFormat()`, though this fact is not at all intuitive to a developer due to the misleading function names.

ColdFusion should provide a more complete set of contextual encoding/escaping functions, similar to what the OWASP XSS Prevention Cheat Sheet provides. This would at least give ColdFusion developers the tools necessary to easily eliminate XSS from their applications.

XSS Caused by Default Pages

ColdFusion error pages can contribute to cross-site scripting in unexpected ways. Consider the following code snippets:

```
<cfoutput>#int(URL.count)#</cfoutput>
```

```
<cfset safenum=NumberFormat(FORM.num)>
```

Both of these examples use casting functions to ensure that tainted value is numeric. Now consider what happens when URL.count or FORM.num contain non-numeric characters – ColdFusion generates a verbose error page intended to help the developer debug the problem. The error page contains a message such as:

```
The value foo cannot be converted to a number.
```

There is no problem if the tainted data was literally "foo"; however, no encoding is performed so it is possible to inject JavaScript using other values. For example, if the request contained a GET parameter of `<img src="x" onerror="alert(1)">` the JavaScript alert would fire. The default error page does have scriptProtect enabled, which has already been shown to be ineffective.

From a developer's perspective this behavior is unexpected and counter-intuitive. In the process of trying to improve their data validation, they inadvertently create a XSS vulnerability.

XSS Caused by Custom Error Pages and Exception Handling

Luckily, avoiding the default ColdFusion error page is easy. The application can define a custom error handler using the `<cferror>` tag:

```
<cferror template="errorhandler.cfm" type="request">
```

Now, instead of displaying the default error page, ColdFusion will load errorhandler.cfm when an error occurs. However, there are still ways to make mistakes. The template for the custom error page has access to an object called `error` with member variables containing information about the error that just occurred. A typical custom error page might contain code like the following:

```
An error occurred at #error.dateTime# on this page: #error.template#
Details of the error: #error.message#
Please contact customer support at 800-xxx-yyyy
```

Unfortunately, #error.message# contains the same text as the default error page, which means that the custom error page will be vulnerable to XSS. Developers should avoid using #error.message# or #error.diagnostics# on custom error pages, or properly encode those values before displaying them.

Exception handling can also be used to detect and handle errors within a page using the <cftry> and <cfcatch> tags. Here is an example of exception handling being used to catch casting errors:

```
<cftry>
  <cfoutput>#int(URL.count)#</cfoutput>
  <cfcatch>Exception caught: #cfcatch.message#</cfcatch>
</cftry>
```

Similar to the "error" object that is available on error templates, an object called "cfcatch" is available inside any <cfcatch> block that contains similar information. #cfcatch.message#, like #error.message#, contains the non-encoded error text, and is therefore vulnerable to XSS. Developers should avoid using #cfcatch.message#, or properly encode the value before displaying it.

Amazingly, none of the ColdFusion documentation warns developers of the XSS risks associated with custom error pages or <cfcatch> blocks.

SQL Injection

ColdFusion only provides a few tags for querying databases. Backend databases are configured through the administrative GUI and referenced by CFML code using aliases. The <cfquery> tag is used to execute an ad-hoc SQL query, as shown here:

```
<cfquery name="getContent" dataSource="myDataSource">
  SELECT * FROM pages WHERE pageID = #URL.Page_ID# OR
  title = '#URL.Title_Search#'
</cfquery>
```

Notice that GET parameters are used to construct the SQL query. In this example, SQL injection can occur in the #URL.Page_ID# field but not the #URL.Title_Search# field. The reason for this is that ColdFusion automatically escapes single quotes on ColdFusion expressions inside <cfquery> tags, as a defense against SQL injection. Since #URL.Title_Search# is enclosed in single quotes and ColdFusion escapes any single quotes in the tainted value, an attacker cannot break out of the quoted string and manipulate the query syntax. For #URL.Page_ID#, however, escaping single quotes does not help because there are no quotes to break out of. This can be addressed by putting single quotes around #URL.Page_ID# in the query, but a more effective solution is to use parameterized queries.

The <cfqueryparam> tag is used to bind ColdFusion expressions to data types similar to how one would use prepared statements in other web development frameworks. One or more <cfqueryparam> tags can be nested within a <cfquery> tag to specify constraints on each of the variable elements. For example:

```
<cfquery name="getContent" dataSource="myData">
  SELECT * FROM pages WHERE pageID =
  <cfqueryparam value="#URL.Page_ID#" cfsqltype="cf_sql_integer">
```

```
        </cfquery>
```

Now that `#URL.Page_ID#` is defined as an integer using the “cfsqltype” tag attribute, it is no longer possible to inject arbitrary SQL into the ad-hoc query. Interestingly, the “cfsqltype” attribute is listed as optional in the ColdFusion documentation; however, even when that attribute is omitted, SQL injection is no longer possible.

For stored procedures, the `<cfstoredproc>` and `<cfprocparam>` tags can be used in similar fashion to `<cfquery>` and `<cfqueryparam>`.

Unscoped and Undefined Variables

Earlier we discussed the notion of variable scopes and the fact that variables can be referenced without supplying an explicit scope. Presumably this design was chosen to make CFML development easier, but one side effect of the decision is that the attack surface is increased. There are several situations where unscoped or undefined variables can lead to security vulnerabilities.

Search Order for Unscoped Variables

Before describing exploit scenarios, it’s important to understand how ColdFusion deals with unscoped variables. If a variable is referenced without an explicit scope, e.g. `#myvar#`, ColdFusion iterates over a predefined series of scopes trying to find that variable. Scopes are checked in the following order (Adobe Systems, Inc., 2009):

1. Local (function-local, UDFs and CFCs only)
2. Arguments
3. Thread local (inside threads only)
4. Query (not a true scope; variables in query loops)
5. Thread
6. Variables
7. CGI
8. Cffile
9. URL
10. Form
11. Cookie
12. Client

So the ColdFusion runtime first looks for a local variable called `#myvar#`, then `#CGI.myvar#`, `#URL.myvar#`, and so on, until it finds one that exists. Other valid scopes, such as Request, Session, Server, and others are not included in the search order. Access to those variables must be explicitly scoped.

Exploiting Unscoped Variables

The ramification of using unscoped variables is that application logic may be manipulated in ways the developer did not intend. For example, the developer may be assuming that `#myvar#` is being populated by a POST variable whereas in reality it could be populated by a CGI variable or a GET variable. Many if not all security vulnerabilities stem from making bad assumptions, and this is no exception.

Consider this code sample in which an authorization decision is made based on the existence of a local variable. Assume that earlier in the function, the code checked the user's role and set a local variable called `isAllowed` to 1.

```
<cfif IsDefined("isAllowed")>
  DoImportantStuff()
<cfelse>
  Sorry, you are not permitted to access this functionality.
</cfif>
```

The developer assumes that the call to `IsDefined()` will only be checking the value of the local variable called `isAllowed`. However, even if the application logic never set that variable, putting `importantVar=[anything]` in the URL (or the POST data, or a cookie) will essentially cause the `IsDefined()` check to succeed, and the user will be permitted to access functionality that should be restricted.

Another exploit scenario involves applying inconsistent scopes at different parts of the code. Consider this application logic that processes a user login and sets a Client scoped variable to capture whether the user has administrative privileges:

```
<cfif AuthenticateUser(FORM.username, FORM.password) and
      IsAdministrator(FORM.username)>
  <cfset Client.admin = "true">
<cfelse>
  <cfset Client.admin = "false">
</cfif>
```

Because Client scoped variables persist across pages and site visits, other CFML pages in the application use that variable as an authorization check before performing administrative actions:

```
<cfif admin eq "true">
  Put privileged functionality here!
<cfelse>
  Sorry, only admins can access this!
</cfif>
```

Notice that the developer forgot to specify the variable scope! Instead of checking the `admin` variable in only the Client scope, the ColdFusion runtime will use the scope search order to find the variable. If an `admin` variable is found in another scope before reaching the Client scope (sixth in the search order), that variable will be used instead of the one the developer intended. This flaw could be easily exploited by putting `admin=true` in the URL.

Developers should use explicit scopes for all variables to avoid these types of vulnerabilities. Code reviewers should look for vulnerabilities by scouring the application for any unscoped variables or scope mismatches.

Exploiting Undefined Variables

Undefined variables are another source of potential vulnerabilities. The issue is related to confusion between the `<cfparam>` and `<cfset>` CFML tags. The `<cfparam>` tag tests for the existence of a variable and provides a default value if a value is not already assigned, while the `<cfset>` tag explicitly assigns a value in all situations. It's not uncommon to see code like the following:

```
<cfparam name="pagenum" default="1">
<cfoutput>
  Now showing page #pagenum#.
</cfoutput>
```

In this example, the `pagenum` variable is assigned a default value of 1 if it doesn't already exist. Because the variable is not previously defined and because it is also unscoped, it can be overridden with a GET or POST variable by the same name.

Developers and code reviewers should assume that undefined, unscoped variables are populated with tainted data.

Environment Variables

Environment variables, or CGI variables, are interesting in ColdFusion because the behavior differs significantly from most web or application servers. Normally, there is a predefined list of CGI variables, and most are immutable, that is, they cannot be manipulated by the end user issuing the HTTP request. In ColdFusion, however, HTTP headers can be used to overwrite many legitimate CGI variables and to create an unlimited number of additional variables in the CGI scope.

Let's first look at how CGI variables work in typical situations. Here is a full HTTP request for /index.cfm on example.com:

```
GET /index.cfm HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.4)
Host: example.com
```

During the execution of index.cfm, the following variables would exist in the CGI scope, along with a bunch of others[1]:

- `CGI.HTTP_HOST` ⇒ example.com
- `CGI.HTTP_USER_AGENT` ⇒ Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.4)
- `CGI.SCRIPT_NAME` ⇒ /index.cfm
- `CGI.SERVER_PROTOCOL` ⇒ HTTP/1.1
- etc.

Now let's alter the original HTTP request to include an additional header:

```
GET /index.cfm HTTP/1.1
```

[1] `<cfdump var="#cgi#">` can be used to generate a full listing

```
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.4)
HTTP_HOST: evil.com
FOOBAR: I can make my own variables!
Host: example.com
```

This request is perfectly valid and will still cause index.cfm to be executed. However, if the code were to reference the variable `CGI.HTTP_HOST`, the value would be "evil.com" instead of "example.com". Most (but not all) of the other variables in the CGI scope can be overwritten in similar fashion. In addition to overwriting `CGI.HTTP_HOST`, this request creates the variable `CGI.FOOBAR` by simply injecting an extra HTTP header into the request. Neither of these behaviors is documented, and it is unclear why arbitrary HTTP headers would be allowed to manipulate CGI variables to this extent.

One interesting attack scenario would be if a website were using Basic or NTLM authentication and an attacker could use an HTTP header to overwrite `CGI.AUTH_USER` and/or `CGI.REMOTE_USER`, which would normally be populated by the web server. For example:

```
GET /page_requiring_basic_authentication.cfm HTTP/1.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.4)
AUTH_USER: admin
REMOTE_USER: admin
Host: example.com
```

When authentication is not configured on the web server, both `CGI.AUTH_USER` and `CGI.REMOTE_USER` can be overwritten successfully using this method. However, if Basic or NTLM authentication is enabled, only `CGI.AUTH_USER` can be overwritten. ColdFusion's `GetAuthUser()` function also returns the correct username, not the overwritten value. Due to time limitations, these authentication scenarios were only tested on the IIS web server, so there is a possibility that other servers may behave differently.

Developers should be careful not to assume `CGI.REMOTE_USER` and other CGI variables contain trusted values.

Persistent Variables

The intended use case for Client variables is to allow an application to maintain a user's state across multiple visits to a website. By default, these variables are stored in the system registry on the server, but they can also be stored in HTTP cookies or a backend database. The `<cfapplication>` tag in Application.cfm must be configured to enable Client storage; by default it is disabled. Within the `<cfapplication>` tag, the "clientStorage" attribute specifies where the data is stored. For example:

```
<cfapplication
  name = "application_name"
  clientManagement = "Yes"
  clientStorage = "Cookie" ...>
```

If the "clientStorage" attribute is omitted, ColdFusion will use the system registry by default. If it is set to "Cookie", then an HTTP cookie called CFCLIENT_CLIENT will be issued. The cookie contains unencrypted, hash-separated name-value pairs. Here is an example of how to use Client variables in a CFML page:

```
<cfset Client.foo = "some_value">
<cfset Client.bar = "some_other_value">
```

And here is the resulting HTTP cookie:

```
Set-Cookie:
CFCLIENT_CLIENT=bar%3Dsome%5Fother%5Fvalue%23foo%3Dsome%5Fvalue%23;
expires=Sun, 15-Jul-2040 20:37:55 GMT;path=/
```

Any of the variables embedded in the CFCLIENT_CLIENT cookie could be manipulated, and the application would never know the difference due to the lack of integrity checks (e.g. an HMAC).

Developers should be careful when using HTTP cookies as the storage mechanism for Client variables. Generally speaking, cookies should never be used to store sensitive information, and this case is no exception.

Direct Invocation of UDFs

Recall from earlier that every method in a .cfc file is a potential entry point, depending on how it is defined. For example, the following URL will invoke method xyzzy on an anonymous instance of component foo.cfc, with arguments arga="vala" and argb="valb" (arguments can also be passed as POST variables, but the method name must be in the URL):

```
http://example.com/foo.cfc?method=xyzzy&arga=vala&argb=valb
```

Not all UDFs can be invoked directly like this. Most, in fact, are designed to be called only by other .cfc or .cfm pages but are not intended to be entry points. Whether or not UDFs are directly accessible from the web is based on how the `<cffunction>` tag is configured, specifically the value of the "access" attribute. Below is a sample UDF that converts Fahrenheit to Celsius:

```
<cffunction name="ftoc" output="false" access="remote">
  <cfargument name="temp" required="yes" type="numeric">
  <cfreturn ((temp-32)*5/9)>
</cffunction>
```

The value of the "access" attribute is "remote", which means that the UDF can be accessed from remote clients through a URL, Flash, or a web service. If the "access" attribute is not supplied, it defaults to "public", which allows the function to be called by locally executing pages or components but not invoked from outside the application. Other, more restrictive settings for the "access" attribute include "private" and "package".

Vulnerabilities exist when sensitive functions are implemented as UDFs with the access attribute set to "remote". Developers should select the most restrictive access level possible when implementing UDFs.

ColdFusion Behind the Curtain

Since ColdFusion made the jump to J2EE in ColdFusion MX (i.e. ColdFusion 6.0), all code has been compiled to Java classes and executed through a traditional J2EE workflow. Each page and component is compiled to an individual class; in addition, each user-defined function gets its own class. This compilation is typically done on initial page load; however, ColdFusion comes with a utility to precompile individual files. The CF Administrator console can also bundle up applications, including the ColdFusion runtime, as WAR or EAR archives for easy deployment.

Precompilation and Analysis

The script named `cfcompile`, bundled with ColdFusion, compiles source files into Java class files – either one at a time or entire directories. As mentioned above, a single source file may result in multiple classes. All Java classes resulting from a file named F are concatenated and placed in the output directory in a single file, also named F. This non-standard arrangement is handled by custom class loaders inside the ColdFusion runtime; it is also a hassle for researchers interested in analyzing compiled ColdFusion files.

We couldn't locate a utility that properly dealt with these concatenated class files, so we wrote one. It is written in Java, has no external dependencies, and properly handles entire ZIP-based archives. Source code is included. Its name is cfexplode, and you can find it at:

```
http://code.google.com/p/cfexplode/
```

Usage is straightforward: specify an output directory and a list of cfcompiled files or a zip archive containing them. Non-cfcompiled class files are fine as well; they'll be parsed included in the output directory as well. Simple usage log for a single file being split up into three related classes (one for the page, and one for each UDF defined inside):

```
% file index.cfm
index.cfm: compiled Java class data, version 45.3
% mkdir outdir
% java -jar cfexplode.jar outdir index.cfm
% ls -l outdir
total 40
-rw-r--r-- 1 cstone cstone  3534 2010-07-23 15:24 index.cfm.0.class
-rw-r--r-- 1 cstone cstone  2095 2010-07-23 15:24 index.cfm.3534.class
-rw-r--r-- 1 cstone cstone 31234 2010-07-23 15:24 index.cfm.5629.class
```

Exploded classes can be loaded into normal analysis tools (e.g. JD-GUI, JAD, IDA Pro).

Class Structure

The general contract for the base classes extended by pages and components is fairly simple: setup inside static initialization blocks, methods to register UDFs and function map names to classes, methods to perform initial bindings of Java private variables to ColdFusion variable objects, and one main entry point for the entire page. In the case of components and pages, the entry point is `runPage()`. UDFs are similar; their entry point is `runFunction()`, which is passed information about the current local variable binding stack and the calling page. Arguments to the function are passed in and accessible through the Arguments variable scope, and type validation occurs by the runtime.

Pages and components communicate to the HTTP response through a plain old javax.servlet.jsp.JspWriter, held by the superclass, which also has an associated javax.servlet.jsp.PageContext. These facilitate integration with JSP tag libraries, which are also allowed in ColdFusion pages. Often the compiler "factors out" sections of the main method of a page into separate methods; these methods are named `_factor0()`, `_factor1()`, etc. and called explicitly by `runPage()` or `runFunction()`.

Variable Scoping

As discussed earlier, variables are dynamically scoped in ColdFusion, and references can also be dynamic. Instead of mapping every ColdFusion variable to a corresponding Java variable, the current binding stack is passed around from object to object. Instances of the class coldfusion.runtime.Variable represent bindings. In an effort to speed up variable accesses, the compiler often creates private Variable references inside pages to store pointers to bindings. Here's an example decompilation of this source file, named vtest.cfm:

```
<cfset "z"="y">
<cfset "#z#"="foo">
<cfoutput>
  z: #z# <br/>
  y: #y# <br/>
  url.var: #url.var# <br/>
</cfoutput>
```

This results in one class being generated, since there are no UDFs defined in the page. This is a cleaned-up version of a JD-CORE (Dupuy, 2010) decompilation. The only omission is the `getMetadata()` accessor.

```
public final class cfvtest2ecfm39616334 extends CFPage {
  private Variable Y;
  private Variable Z;
  static final Class class$coldfusion$tagext$io$OutputTag;
  public static final Object metaData;

  static {
    class$coldfusion$tagext$io$OutputTag =
      Class.forName("coldfusion.tagext.io.OutputTag");
    metaData = new AttributeCollection(new Object[0]);
  }

  protected final void bindPageVariables(VariableScope varscope, LocalScope
locscope) {
    super.bindPageVariables(varscope, locscope);
    this.Y = super.bindPageVariable("Y", varscope, locscope);
    this.Z = super.bindPageVariable("Z", varscope, locscope);
  }
  protected final Object runPage() {
    Throwable t8, t7;
    Object t6, value;
    int mode0;
    JspWriter out = this.pageContext.getOut();
    Tag parent = this.parent;
    super.bindImportPath("com.adobe.coldfusion.*");
    super._set("z", "y");
    super._whitespace(out, "\n");
    super._set(Cast._String(super._autoscalarize(this.Z)), "foo");
    super._whitespace(out, "\n");
    OutputTag output0 =
(OutputTag)super._initTag(class$coldfusion$tagext$io$OutputTag, 0, parent);
    super._setCurrentLineNo(3);
    output0.hasEndTag(true);
    try {
      if ((mode0 = output0.doStartTag()) != 0) {
        do {
          out.write("\n  z: ");
          out.write(Cast._String(super._autoscalarize(this.Z)));
          out.write(" <br/>\n  y: ");
          out.write(Cast._String(super._autoscalarize(this.Y)));
          out.write(" <br/>\n  url.var: ");
          out.write(Cast._String(super._resolveAndAutoscalarize("URL", new
String[] { "VAR" })));
          out.write(" <br/>\n");
        } while (output0.doAfterBody() != 0);
      }
      if (output0.doEndTag() == 5)
        return null;
    } catch (Throwable localThrowable1) {
      output0.doCatch(localThrowable1);
    } catch (Throwable localThrowable2) {
      throw localThrowable2;
    } finally {
      Object t9 = returnAddress;
      output0.doFinally();
    }
    super._whitespace(out, "\n");
    return null;
  }
}
```

In this case, there are two variables: z and y. These are reflected by the two Variable fields bound by bindPageVariables(): this.Y and this.Z. These bindings are used to retrieve the current value of the corresponding ColdFusion variable at that point in time:

```
// set a variable whose name is the contents of variable "z" to the
// string "foo"
super._set(Cast._String(super._autoscalarize(this.Z)), "foo");
```

The function `super._autoscalarize(Variable)` resolves the given Variable from the local scope and returns the result; `Cast._String()` attempts to convert the value to a String, and throws a ColdFusion exception (and a Java exception) if it fails.

When the name of the assigned variable is known at compile-time, the code to set it is simpler:

```
// set variable "z" to the string "y"
super._set("z", "y");
```

In addition to `super._set()`, Variable objects have a `set(String)` method that is often invoked directly; this is also used to set the value of variables.

When a fully-qualified variable is referenced – in this case `url.var`, which refers to the GET variable named `var`, if present – the variable resolution function differs again:

```
out.write(Cast._String(super._resolveAndAutoscalarize("URL", new String[] {
  "VAR" })));
```

Several CFML tags also modify variables in the local scope: `<cfquery>`, `<cfloop>`, `<cfset>`, and `<cffile>` are a few examples. Typically, the variable to modify is passed as a string to the runtime classes that implement these tags.

WAR Structure

ColdFusion applications can be bundled as WAR or EAR files using the CF Administrator web-based tool. Inside the corresponding WAR's web.xml file, it's clear that pages and components are handled by separate servlets: coldfusion.CfmServlet and coldfusion.xml.rpc.CFCServlet (*.CFR ColdFusion report files are also handled by CFCServlet).

These servlets – invoked by the wrapper class coldfusion.bootstrap.BootstrapServlet – locate the classes being requested based on the URL (and parameters, in the case of direct invocation of CFC methods); retrieve variables from HTTP headers, POSTed content, and GET variables and populate the corresponding variable scopes; instantiate the called class and run all setup methods; and finally, call the `runPage()` or `runFunction()` method.

This series of events happens toward the end of processing of a coldfusion.filter.FusionFilter classes (NB: these are not javax.servlet.Filters). It should be noted that the compiled page/function classes are also considered Fusion Filters. Code to propagate persistent data across sessions/clients (e.g. in the Session and Client scopes) lives in these filters as well.

Other Bundled Servlets

A typical WAR, as bundled by the CF Administrator on ColdFusion 9, contains servlets in addition to the page-handling functions described above. These servlets support various ColdFusion-specific functions. Here is a subset of servlets, along with their default URL mappings:

- *.jsp: JSPLicenseServlet; this is a passthrough for jrun.jsp.JSPServlet

- /flex2gateway/*, /flashservices/gateway/*, /CFFormGateway/*: FLEX/plain Flash Remoting gateways for CFC methods
    - Mappings work like this: /flashservices/gateway/path1.path2.component -> path1/path2/component.cfc
    - Gateways can be used in Flash through the ActionScript call NetServices.createGatewayConnection()
    - This is used internally by `<cfgrid>` and other built-in ColdFusion tags that generate Flash-based UI automatically
- /CFIDE/GraphData.cfm: GraphServlet: generates data graphs and charts; used by the `<cfchart>` tag.
- /CFFileServlet/*: CFFileServlet: handles and serves up files from a cache directory; used by `<cfimage>`
- /cfform-internal/*: FLEX FileManagerServlet; serves a handful of dynamically-generated images and js files
- /WSRPProducer/*: WSRP portlet management Axis service

Conclusions

ColdFusion was designed to be simple for web "developers" to use, but this simplicity comes at the expense of being quite complicated underneath.   It's easy to make coding mistakes or overlook vulnerabilities during a code review if you don't understand ColdFusion internals such as request lifecycle, error handling, and variable scopes. This is true of any web application framework but ColdFusion is particularly onerous.

Takeaways for Developers

- Use scriptProtect if you want, but don't expect it to be effective at preventing XSS.
- Never deploy an application with the default error page; use a custom error page and remember to encode variables like `error.message` and `error.diagnostics`.
- Use `<cfqueryparam>` or `<cfprocparam>` for all SQL queries.
- Set restrictive access permissions on UDFs defined in .cfc files.
- Never use unscoped variables; always use explicit scoping.
- Do not confuse `<cfparam>` and `<cfset>`.
- Remember that most CGI variables can be manipulated, and be particularly careful not to trust `CGI.AUTH_USER`.
- Do not use cookie-based storage for Client scope variables.
- Strip out unnecessary ColdFusion components before deploying an application.

Takeaways for Security Practitioners

- If you come across a default ColdFusion error page while penetration testing, it will almost always be vulnerable to XSS.

- On any .cfm page that reflects a user-supplied parameter, inject `<script>` and look for `<InvalidTag>` in the response to determine if scriptProtect is enabled; use `<img>`, `<a>`, or other tags with JavaScript event attributes to bypass scriptProtect.
- Inject into numeric parameters when attempting SQL injection exploits; strings are automatically single quote escaped without developer intervention.
- In a code review, any mismatches in variable scoping may be a potential vulnerability (e.g. setting a variable with an explicit scope but accessing it unscoped later).
- You can override a lot more CGI variables than you are accustomed to.
- The presence of a `CFCLIENT_CLIENT` cookie indicates that cookie-based storage is being used; keep an eye out for trivial cookie poisoning attacks.
- ColdFusion-generated Java classes are pretty ugly; use the cfexplode utility to help reverse engineer any WAR or EAR files you obtain.

Takeaways for Adobe

Developer security documentation for ColdFusion could be vastly improved in the following ways:

- Provide more guidance on web vulnerabilities; the current information is disproportionately focused on security features.
- More clearly describe the limitations of scriptProtect so developers are not given a false sense of security; saying that it blocks "most attempts" at XSS is disingenuous at worst, misleading at best.
- Fix the default error page to eliminate the XSS risk.
- Warn developers about the XSS risk of custom error pages. Specifically identify the members of the `error` object that are tainted.
- Provide a more extensive library of built-in encoding methods, using the OWASP XSS Prevention Cheat Sheet as a guide.
- Discuss the security implications of unscoped variables, rather than suggesting that explicit scoping only has performance benefits.
- Do not allow arbitrary CGI variables to be created and/or overwritten.
- Discuss the security implications of using cookie-based Client storage.

Bibliography

Adobe Systems, Inc. (2005, April 18). *Adobe to Acquire Macromedia*. Retrieved July 19, 2010, from Adobe Systems Inc. web site: http://www.adobe.com/aboutadobe/invrelations/adobeandmacromedia.html

Adobe Systems, Inc. (2010, July). *Customer Stories: Who's Using ColdFusion*. Retrieved July 19, 2010, from Adobe Systems, Inc. web site: http://www.adobe.com/products/coldfusion/customers/

Adobe Systems, Inc. (2009). *Developing ColdFusion 9 Applications.* Retrieved July 19, 2010, from Adobe Systems, Inc. web site: http://help.adobe.com/en_US/ColdFusion/9.0/Developing/coldfusion_9_dev.pdf

Davis, A. (2006). ColdFusion Security. *EUSecWest 2006.* Amsterdam.

Dean, J., & Shelton, B. (2009, June). *OWASP Enterprise Security API for ColdFusion.* Retrieved from http://code.google.com/p/owasp-esapi-coldfusion/source/list

Driver, M. (2001, January 17). *Commentary: Macromedia-Allaire a match made in heaven*. Retrieved July 19, 2010, from CNET: http://news.cnet.com/Commentary-Macromedia-Allaire-a-match-made-in-heaven/2009-1023_3-251093.html

Dupuy, E. (2010). *Java Decompiler project (freely available Java decompilation library and tools)*. Retrieved from http://java.decompiler.free.fr/

Lee, E., Melven, I., & Sargent, S. (2007, October). *ColdFusion 8 Developer Security Guidelines.* Retrieved from Adobe Systems, Inc. web site: http://www.adobe.com/devnet/coldfusion/articles/dev_security/coldfusion_security_cf8.pdf

Lichtin, H. (Unknown). *Using ColdFusion Variables.* Retrieved July 22, 2010, from Adobe Systems, Inc. web site: http://www.adobe.com/devnet/server_archive/articles/using_cf_variables.html

OWASP Foundation. (2010, June 8). *XSS Prevention Cheat Sheet.* Retrieved July 22, 2010, from OWASP web site: http://www.owasp.org/index.php?title=XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet

WhiteHat Security, Inc. (2010, May). *WhiteHat Website Security Statistics Report, 9th Edition.* Retrieved from WhiteHat Security, Inc. web site: http://www.whitehatsec.com/home/assets/WPstats_spring10_9th.pdf

Wikipedia contributors. (2010, July 16). *ColdFusion.* Retrieved July 19, 2010, from Wikipedia, The Free Encyclopedia: http://en.wikipedia.org/wiki/ColdFusion