Evil HTTP Compression - Compression Bombs

Denial-of-Service Attacks against User-Agents and Content Inspection Devices

Author: Geoff Jones, Cyberis Limited
Email: geoff.jones@cyberis.co.uk
Web: www.cyberis.co.uk

Abstract

The majority of web browsers and other HTTP User-Agents in use today and supporting compression are vulnerable to various denial-of-service conditions - namely memory, CPU and free disk space consumption - by failing to consider the high compression ratios possible from data with an entropy rate of zero. Using multiple rounds of encoding[1], a 43 Kilobyte HTTP server response will equate to a 1 Terabyte file when decompressed by a receiving client - an effective compression ratio of 25,127,100:1.

Several techniques will be outlined in this paper demonstrating how browsers handle such highly compressed data when loading resources in-line (e.g. HTML content and images) compared to when instructed to save a file directly to disk. A number of techniques to bypass required user interaction will also be detailed, as will the specific capabilities and idiosyncrasies of common browsers.

This vulnerability is a common weakness that affects multiple vendors. gzip bombs have been publicly reported for nearly 10 years[2], although nearly all current browser versions are still susceptible. In recent years the threat landscape has changed somewhat, with next-generation firewalls and mobile User-Agents presenting interesting new targets to would-be attackers. New compression schemas, such as Shared Dictionary Compression over HTTP (SDCH[3]), also increase the available attack surface to an adversary.

Introduction

HTTP compression is a capability widely supported by web browsers and other HTTP User-Agents, allowing bandwidth and transmission speeds to be maximised between client and server. Supporting clients will advertise supported compression schemas, and if a mutually supported scheme can be negotiated, the server will respond with a compressed HTTP response.

Compatible User-Agents will typically decompress encoded data on-the-fly. HTML content, images and other files transmitted are usually handled in memory (allowing pages to rendered as quickly as possible), whilst larger file downloads will usually be decompressed straight to disk to prevent unnecessary consumption of memory resources on the client.

Gzip (RFC1952) is considered the most widely supported[4] compression schema in use today, and has been used in the majority of attacks detailed in this paper, although other content encoding

[1] 4 rounds of gzip encoding
[2] http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html
[3]SDCH was proposed as a new compression schema in 2008
[4] http://www.vervestudios.co/projects/compression-tests/results

schemes can be exploited in exactly the same way. This paper does not detail specific vulnerabilities in the individual compression algorithms, rather the specific handling of compressed responses with regards to user interaction, automatic file download and memory allocation.

Testing Framework - GzipBloat

The author has written a testing framework, both for generic HTTP response tampering and various sizes of gzip bombs. GzipBloat (https://www.gitbhub.com/cyberisltd/GzipBloat) is a PHP script to deliver pre-compressed gzipped content to a browser, specifying the correct HTTP response headers for the number of encoding rounds used, and optionally a 'Content-Disposition' header. A more generic response tampering framework - ResponseCoder (https://www.github.com/cyberisltd/ResponseCoder) - allows more fine grained control, although content is currently compressed on the fly - limiting its effectiveness when used to deliver HTTP compression bombs.

Testing Methodology

The most popular desktop browsers in June 2013 as recorded by StatCounter[5] are shown below:

Internet Explorer	Firefox	Chrome	Safari	Opera
25.44 %	20.01 %	42.68 %	8.39 %	1.03 %

These five desktop browsers formed the basis of all tests, though it is important to note that any User-Agent that supports any form of HTTP compression is potentially vulnerable to the techniques listed in this paper. Where possible, multiple versions of the browsers were tested, with comments detailing differing behaviours of note. Linux, Windows and MAC versions of each browser were tested where available, although the majority of tests were conducted on a fully-patched version of Windows 7 (x64).

Three mobile based browsers were also be tested, namely Android's Webkit browser (version 4.0.4-XXLPH), Chrome on Android (version 18.0.1025469) and Safari on IOS (IOS 6.1.3).

Each browser was subjected to the following tests:

1. In-line request of a 1TB gzip encoded file with 4 rounds of encoding ('Content-type: text/html') - test file size 43k
2. In-line request of a 1TB gzip encoded file ('Content-type: text/html') - test file size 1GB
3. File download ('Content-Disposition: attachment') of of a 1TB gzip encoded file with 4 rounds of encoding - test file size 43k
4. File download ('Content-Disposition: attachment') of of a 1TB gzip encoded file - test file size 1GB
5. 1TB gzip compressed SDCH dictionary with 4 rounds of encoding - test file size 43k

[5] http://gs.statcounter.com/

All test cases were delivered by Cyberis' GzipBloat framework.

Results

Desktop Browsers

	Internet Explorer	Firefox	Chrome/Chromium	Safari	Opera
1TBx4 HTML	Not supported	See 3	See 6	Not supported	See 10
1TB HTML	See 1	See 3	See 6	See 9	See 11
1TBx4 FILE	Not supported	See 4	See 7	Not supported	See 12
1TB FILE	See 2	See 5	See 7	See 9	See 12
1TB SDCH	Not supported	Not supported	See 8	Not supported	Not supported

Key	Not supported	PASS - No effect or an error message displayed. Closure of browser tab permitted. No performance issues observed.	FAIL - Operating system resource exhaustion that can be recovered by termination of process (automatic or manual).	FAIL - Operating system denial-of-service requiring manual intervention to recover.

Test Notes

1. Memory exhaustion - operating system will eventually prompt to close the browser, once all physical and virtual memory is exhausted. UI very slow to respond. CPU usage also high.

2. File download prompt will only be displayed once response has been decompressed. Prior to this point, the download will fill the disk with a temporary file, *with absolutely no indication of the download occurring.* Download will continue until all available disk space is consumed. Tested on multiple versions of Internet Explorer, including IE11 preview on Windows 8.1. Navigation away from the page or closure of the browser *does not remove the temporary file*. CPU usage also high, although memory usage normal. Manual removal of temporary file (located in Temporary Internet Files) is required to recover the operating system. NB: clearing of Temporary Internet Files via Internet Explorer or Control Panel does not remove the file - command line access is required.

3. Memory and CPU exhaustion, although browser seems to recover without crash. UI very slow to respond.

4. Disk, memory and CPU exhaustion, operating system was inoperable during the download. No user interaction required to exploit. Low memory warning on Windows observed. Disk usage possible attributable to swap file usage. Operating system recovered after test, no temporary files remained.

5. File download prompt displayed, although browser continues to write a temporary file to disk, *prior to user confirming the download.* Download continues until all disk space is consumed, or user cancels the download dialog. High CPU usage, memory spikes. Once all disk space was consumed, temporary file was removed and the browser recovered from the download.

6. 'Aw, Snap!' displayed on Windows (Chrome 28.0.1500.71 m) shortly after load (CPU and memory spikes temporarily). Chromium on Linux (Version 28.0.1500.71 (209842)) consumes all CPU and available memory, running into swap space. UI very slow to respond.

7. Disk space completely exhausted - after which the download terminates with the error message 'Failed - Disk full' and the temporary file is removed. CPU usage moderate, memory usage normal.

8. Shared Dictionary Compression over HTTP (SDCH)[6] - if the server responds with an 'Get-Dictionary' header pointing to a gzip bomb (see appendix A), Chrome requests the SDCH dictionary in the background. *No user interaction is required.* On Windows (Chrome 28.0.1500.71 m), the response only results in a spike of memory and CPU for a limited time. Chromium on Linux (Version 28.0.1500.71 (209842)), all available memory is consumed (including swap space) and CPU usage is high. UI very slow to respond.

9. CPU/memory exhaustion leading to browser crash. Only limited testing conducted on this platform.

10. File download terminates shortly after commencing. No adverse effects on operating system. No error message displayed, unless 'view-source' is used. CPU usage high, memory normal.

11. File download terminates shortly after commencing. No adverse effects on operating system. Error message displayed indicating page has crashed. CPU usage high, memory normal.

12. Disk space completely exhausted - after which the download terminates with the error message 'Your hard disk is full. Please save to another location…' and the temporary file is removed. CPU usage moderate, memory usage normal.

[6] http://www.blogs.zeenor.com/wp-content/uploads/2011/01/Shared_Dictionary_Compression_over_HTTP.pdf

Mobile Browsers

	Chrome (Android)	Webkit browser (Android)	Safari (IOS)
1TBx4 HTML	See 1	See 4	Not supported
1TB HTML	See 1	See 5	See 6
1TBx4 FILE	See 2	See 2	Not supported
1TB FILE	See 2	See 2	See 6
1TB SDCH	See 3	Not supported	Not supported

Key	Not supported	PASS - No effect or an error message displayed. Closure of browser tab permitted. No performance issues observed.	FAIL - Operating system resource exhaustion that can be recovered by termination of process (automatic or manual).	FAIL - Operating system denial-of-service requiring manual intervention to recover.

NB: Windows Phone 8 and Blackberry 10 devices were not available for testing

1. 'Aw, Snap!' error message displayed. Operating system seemingly unaffected.
2. Downloads the file, but free space remaining suggests the file was not decompressed correctly.
3. File requested, but no indication of free space being used.
4. Blank page displayed (possibly multiple rounds of decompression not supported?)
5. Browser crash. On reload of the browser, the same page is resumed, causing a further crash. Subsequent attempts do not reload the affected page.
6. Browser crash shortly after response is received.

Common Weakness

The results show that the most popular web browsers in use today are vulnerable to various denial of service conditions - namely memory, CPU and free disk space consumption - by failing to consider the high compression ratios possible from data with an entropy rate of zero (for example /dev/zero). Depending on the HTTP response headers used, vulnerable browsers will either decompress the content in memory, or directly to disk - only terminating when operating system resources are exhausted.

The most serious condition observed was an effective denial-of-service against Windows operating systems when a large gzip encoded file is returned with a 'Content-Disposition' header - recovery from the condition required knowledge of the Temporary Internet FIles directory structure and command line access. This seemed to affect all recent versions of IE, including IE11 on Windows 8.1 Preview.

Dangerous Assumptions

A number of potential reasons why this common weakness may exist in a product:

Dangerous Assumption #1 - Compressed data is generated 'on-the-fly'

Usually, compression schemas favour decompression with regards to speed - it is more computationally expensive to compress content on the server than it is to decompress the received content on the client. If one server handling many clients can encode content 'on-the-fly', it may be assumed that a receiving client should be able to decompress the content, especially when considering it is less computationally expensive to perform the decompression routine.

A malicious web host however, can perform all necessary compression routines off-line, configuring a web server to serve the already compressed content (with necessary 'Content-Encoding') headers to unsuspecting victims - the server no longer needs to compress each and every HTTP response. An attacker can take as much time as necessary to highly compress very large files ready for delivery.

Dangerous Assumption #2 - Compression is used to compress 'real data'

It is a fair assumption that most browsers would expect HTTP compression to be reducing the bandwidth requirements of 'real data'. As most HTML content is ASCII-based text, a typical compression ratio of 3:1 is not unusual. Binary data (for example images), may be even less, especially when considering modern formats that natively support compression (e.g. PNG). As in-line HTML content and images are relatively small, even complex web pages are unlikely to trouble the CPU and memory resources of a modern operating system - memory being the ideal place for decompression to occur, of course, for speed reasons.

Again, an adversary need not be concerned with real data - a large file containing nothing but zeros will suffice for a denial of service, and as it has an effective entropy of zero, it will compress very well (1027:1):

```
$ dd if=/dev/zero bs=10M count=1 | gzip -9 | wc -c
1+0 records in
1+0 records out
10485760 bytes (10 MB) copied, 0.172531 s, 60.8 MB/s
10208
```

Now, a number of User-Agents support multiple levels of content encoding (e.g. 2 more rounds of gzip compression).

```
$ dd if=/dev/zero bs=10M count=1 | gzip -9 | gzip -9 | wc -c
1+0 records in
1+0 records out
10485760 bytes (10 MB) copied, 0.149518 s, 70.1 MB/s
159
```

By passing it through gzip twice, we can now see a compression ratio of 65948:1. As the input file size increases, along with the number of encoding rounds, this ratio will continue to increase; a 1 Terabyte file with 4 rounds of gzip encoding will result in just a 43 Kilobyte response.

Dangerous Assumption #3 - The user/browser will probably say yes

Some modern browsers attempt to 'speed up' file downloads by commencing with a download to a temporary file prior to the user actually confirming the download - if the user subsequently cancels the request, the transfer will be terminated and the temporary file removed.

In the case of a HTTP compression bomb, this obviously has significant implications for the free disk space of the underlying operating system. This is especially true if a process crashes, as any temporary files are likely to be left behind following an unclean exit. Manual removal of the leftover files may be necessary in such cases.

Related to this assumption are other background transfers that may be initiated by the browser. An SDCH-supporting web server for example[7], will instruct an SDCH client (e.g. Chrome) to request a dictionary in the background. As this is a background request initiated by the browser rather than the user, the 'Stop' button does not terminate the HTTP session as per a normal request. The whole browser must be terminated to prevent all available memory from being consumed.

Dangerous Assumption #4 - If it's good enough for you, it's good enough for me

A content inspection device sat between a server and client may attempt to decode compressed content as part of its normal duties. An architect or developer of such a system may decide that any arbitrary number of encoding rounds may be appropriate, which is understandable given some User-Agents [currently] support several thousand rounds[8] of encoding. Unfortunately, a content inspection device may be more critical in terms of availability (it probably supports many clients for example), and therefore the risk of failure should be deemed greater.

Now unfortunately, there is no ideal situation here - failure to decode multiple rounds is an obvious evasion technique, whilst decoding any arbitrary number may lead to denial-of-service. Probably the best solution if technically possible is to remove the 'Accept-Encoding' headers altogether and also drop responses that still have the 'Content-Encoding' header set[9] with a compression schema. Obviously this has implications for bandwidth consumption and therefore speed.

As previously mentioned, the threat landscape has changed in recent years, as now many devices perform such inspection (e.g. proxies, next-generation firewalls, WAFs, IDS/IPS etc), and may therefore be vulnerable to attack.

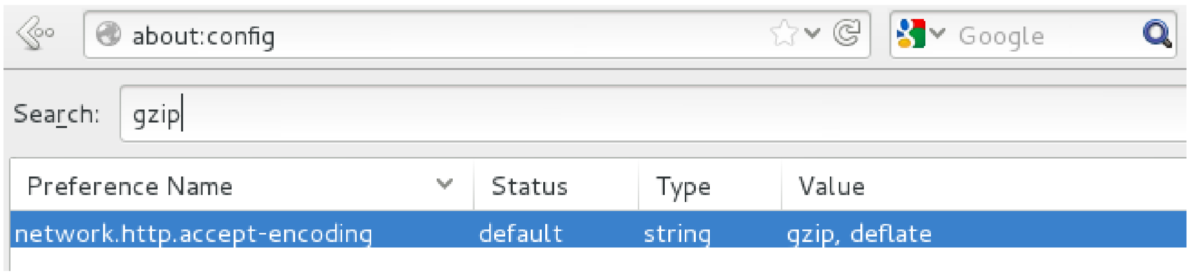
[7] Or a malicious host pretending to be a SDCH capable server
[8] Chrome supports several thousand, for example
[9] Simple removal of the HTTP request header may not be sufficient - an attacker owned server can obviously still return gzip encoded data, and the client will still believe they announced the fact they can accept a Gzip encoded response.

Solution

Several vendor recommendations can be made to mitigate the risk of HTTP compression bombs:

1. Consider restricting multiple rounds of compression encoding
2. If possible, determine the expected size of decompressed content before allocating memory to the task
    a. If there is insufficient memory available to decompress content, perform the decompression via repeated calls of the compression function.
    b. Consider setting a maximum sensible size for in-line HTML content that is delivered with compression
    c. If downloading a file, ascertain as soon as possible whether sufficient disk space exists. If it does not, terminate with an appropriate error message and remove all temporary files.
3. Consider enforcing a maximum decompression time limit for received content
4. If decompression fails when downloading a file, remove all temporary files written to disk
5. IE10/11 should prompt before downloading a gzip encoded file with a 'Content-Disposition: attachment' header
6. Consider limiting the size of temporary file that can be created prior to a user confirming a file download prompt.
7. Enforce sensible limits for SDCH dictionaries (the proposed standard[10] suggests at least 10MB of space on the client side for total dictionary size)
8. Content filtering devices could consider removing HTTP request/response compression directives

There are few mitigating factors for end-users, other than disabling supported compression schemas. This may or not be possible depending on the browser in use. Firefox is known to support this feature via *about:config* :



Search: gzip

Preference Name	Status	Type	Value
network.http.accept-encoding	default	string	gzip, deflate

No obvious configuration option appears in Internet Explorer or Google Chrome.

[10] http://www.blogs.zeenor.com/wp-content/uploads/2011/01/Shared_Dictionary_Compression_over_HTTP.pdf

Conclusion

With the growth of mobile data connectivity, improvements in data compression for Internet communications has become highly desirable from a performance perspective, but extensions to these techniques outside of original protocol specifications can have unconsidered impacts for security.

Although the techniques presented in this paper have presented a known threat for a number of years, the growing ubiquity of advanced content inspection devices, and the proliferation User-Agents which handle compression mechanisms differently, has substantially changed the landscape for these types of attack.

The attacks demonstrated here will provide an effective denial-of-service against a number of popular client browsers, but the impact in these cases is rather limited.

Ultimately, the greater impact of this style of attack is likely to be felt by intermediate content inspection devices with a large pool of users. Although outside of scope of this exercise, the results of this initial testing indicate that it is likely a number of advanced content inspection devices may be susceptible to these decompression denial-of-service attacks themselves, potentially as the result of a single server-client response. In an environment with high availability requirements and a large pool of users, a denial-of-service attack which could be launched by a single malicious Internet server could have a devastating impact.

Appendix A - Server Responses

Test case 1 - 1TB file with four rounds of gzip encoding (no 'Content-Disposition' header):

```
curl -I 'http://127.0.0.1/gzipbloat/gzipBloat.php?rounds=4&infile=1T.gzipx4'

HTTP/1.1 200 OK
Date: Tue, 16 Jul 2013 12:37:45 GMT
Server: Apache
X-Powered-By: PHP/5.4.17-pl0-gentoo
Content-Encoding: gzip, gzip, gzip, gzip
Content-Length: 43758
Content-Type: text/html
```

Test case 2 - 1TB file gzip encoded (no 'Content-Disposition' header):

```
curl -I 'http://127.0.0.1/gzipbloat/gzipBloat.php?infile=1T.gzip'

HTTP/1.1 200 OK
Date: Tue, 16 Jul 2013 12:39:29 GMT
Server: Apache
X-Powered-By: PHP/5.4.17-pl0-gentoo
Content-Encoding: gzip
Content-Length: 1067044016
Content-Type: text/html
```

Test case 3 - 1TB file with four rounds of gzip encoding ('Content-Disposition: attachment'):

```
curl -I
'http://127.0.0.1/gzipbloat/gzipBloat.php?rounds=4&infile=1T.gzipx4&filename=zeros.
txt'

HTTP/1.1 200 OK
Date: Tue, 16 Jul 2013 12:40:52 GMT
Server: Apache
X-Powered-By: PHP/5.4.17-pl0-gentoo
Content-Disposition: attachment; filename="zeros.txt"
Content-Encoding: gzip, gzip, gzip, gzip
Content-Length: 43758
Content-Type: text/html
```

Test case 4 - 1TB file gzip encoded ('Content-Disposition: attachment'):

```
curl -I
'http://127.0.0.1/gzipbloat/gzipBloat.php?infile=1T.gzip&filename=zeros.txt'

HTTP/1.1 200 OK
Date: Tue, 16 Jul 2013 12:41:40 GMT
Server: Apache
X-Powered-By: PHP/5.4.17-pl0-gentoo
Content-Disposition: attachment; filename="zeros.txt"
Content-Encoding: gzip
Content-Length: 1067044016
Content-Type: text/html
```

Test case 5 - 1TB SDCH dictionary with four rounds of gzip encoding:

```
curl -I 'http://127.0.0.1/gzipbloat/sdch.php'

HTTP/1.1 200 OK
Date: Tue, 16 Jul 2013 12:42:20 GMT
Server: Apache
X-Powered-By: PHP/5.4.17-pl0-gentoo
Get-Dictionary:
/gzipbloat/gzipBloat.php?infile=1T.gzipx4&filename=dictionary.sdch&contenttype=appl
ication/x-sdch-dictionary&rounds=4
Content-Type: text/html
```