In the Compression Hornet's Nest: A Security Study of Data Compression in Network Services

Giancarlo Pellegrino
CISPA, Saarland University, Germany
gpellegrino@mmci.uni-saarland.de

Davide Balzarotti
EURECOM, France
davide.balzarotti@eurecom.fr

Stefan Winter
TU Darmstadt, Germany
sw@cs.tu-darmstadt.de

Neeraj Suri
TU Darmstadt, Germany
suri@cs.tu-darmstadt.de

Abstract

In this paper, we investigate the current use of data compression in network services that are at the core of modern web-based applications. While compression reduces network traffic, if not properly implemented it may make an application vulnerable to DoS attacks. Despite the popularity of similar attacks in the past, such as *zip bombs* or *XML bombs*, current protocol specifications and design patterns indicate that developers are still mostly unaware of the proper way to handle compressed streams in protocols and web applications. In this paper, we show that denial of services due to improper handling of data compression is a persistent and widespread threat. In our experiments, we review three popular communication protocols and test 19 implementations against highly-compressed protocol messages. Based on the results of our analysis, we list 12 common pitfalls that we observed at the implementation, specification, and configuration levels. Additionally, we discuss a number of previously unknown resource exhaustion vulnerabilities that can be exploited to mount DoS attacks against popular network service implementations.

1 Introduction

Modern web-based software applications rely on a number of core network services that provide the basic communication between software components. For instance, the list includes Web servers, email servers, and instant messaging (IM) services, just to name some of the more widespread ones. As a consequence of their popularity, Denial of Service (DoS) may have very severe consequences on the availability of many web services. In fact, according to the 2014 Global Report on the Cost of Cyber Crime [35], the impact of application DoS is dramatic: 50% of the organizations have suffered from such an attack, and the average cost of a single attack is estimated to be over $166K US [35].

For performance reasons, many network services extensively use data compression to reduce the amount of data transferred between the communicating parties. The use of compression can be mandated by protocol specifications or it can be an implementation-dependent feature. While compression indeed reduces network traffic, at the same time, if not properly implemented, it may also make applications vulnerable to DoS attacks. The problem was first brought to users' attention in 1996 in the form of a recursively highly-compressed file archive prepared with the only goal of exhausting the resources of programs that attempt to inspect its content. In the past, these *zip bombs* were used, for example, to mount DoS attacks against bulletin board systems [1] and antivirus software [2, 57].

While this may now seem an old, unsophisticated, and easily avoidable threat, we discovered that developers did not fully learn from prior mistakes. As a result, the risks of supporting data compression are still often overlooked, and descriptions of the proper way to handle compressed messages are either lacking or misleading. In this paper, we investigate the current use of data compression in several popular protocol and network services. Through a number of experiments and by reviewing the source code of several applications, we have identified a number of improper ways to handle data compression at the implementation, specification, and configuration levels. These common mistakes are widespread in many popular applications, including Apache HTTPD and three of the top five most popular XMPP servers. Similar to the zip bombs of 20 years ago, our experiments show that these flaws can easily be exploited to exhaust the server resources and mount a denial of service attack.

The task of handling data compression is not as simple as it may sound. In general, compression amplifies the amount of data that a network service needs to process, and some components may not be designed to handle this volume of data. This may result in the exhaustion of re-

sources for applications that were otherwise considered secure. However, in this paper we show that these mistakes are not only caused by unbounded buffers, and neither are they localized into single components. In fact, as message processing involves different modules, improper communication may result in a lack of synchronization, eventually causing an excessive consumption of resources. Additionally, we show similar mistakes when third-party modules and libraries are used. Here, misleading documentation may create a false sense of security in which the web application developers believe that the data amplification risks are already addressed at the network service level.

To summarize, this paper makes the following contributions:

- We show that resource exhaustion vulnerabilities due to highly-compressed messages are (still) a real threat that can be exploited by remote attackers to mount denial of service attacks;
- We present a list of 12 common pitfalls and susceptibilities that affect the implementation, specification, and configuration levels;
- We tested 11 network services and 10 third-party extensions and web application frameworks for a total of 19 implementations against compression-based DoS attacks;
- We discovered and reported nine previously unknown vulnerabilities, which would allow a remote attacker to mount a denial of service attack.

This paper is organized as follows. In Section 2, we introduce the case studies. Then, in Section 3, we discuss the security risks associated with data compression, revisit popular attacks, and outline the current situation. In Section 4, we detail the current situation and present a list of 12 pitfalls at the implementation, specification, and configuration levels. Then, in Section 5, we describe the experiments and present previously-unknown resource exhaustion vulnerabilities. In Section 6, we review related works, and finally, in Sections 7 and 8, we outline future work and draw some conclusions.

2 Data Compression

Data compression is a coding technique that aims at reducing the number of bits required to represent a string by removing redundant data. Compression is *lossless* when it is possible to reconstruct an exact copy of the original string, or *lossy* otherwise. For a detailed survey on compression algorithms please refer to Salomon et al. [45]. Since the focus of our paper is on the incorrect *use* of compression and it is independent of the algorithm itself, we will discuss our finding and examples using the popular *Deflate* algorithm.

Prot.	Network Service	Native	External
XMPP	ejabberd	✗	-
	Openfire	✗	-
	Prosody	✗	-
	jabberd2	✗	-
	Tigase	✗	-
HTTP	Apache HTTPD	✗	-
	mod-php	-	✗
	CSJRPC	-	✗
	mod-gsoap	-	✗
	mod-dav	✗	-
	Apache Tomcat	-	✗
	Axis2	-	✗
	CXF	✗	✗
	jsonrpc4j	-	✗
	json-rpc	-	✗
	lib-json-rpc	-	✗
	Axis2 standalone	✗	-
	gSOAP standalone	✗	-
IMAP	Dovecot	✗	-
	Cyrus	✗	-

Table 1: Case studies and Implementations

Deflate is a lossless data compression technique that combines together a Huffman encoding with a variant of the `LZ78` algorithm. It is specified in the Request For Comments (RFC) number 1951 [13], released in May 1996, and it is now implemented by the widely used `zlib` library [19], the `gzip` compression tool [18], and the `zip` file archiver tool [22], just to name few popular examples.

Deflate is widely used in many Internet protocols such as the HyperText Transfer Protocol (HTTP) [17], the eXtensible Messaging and Presence Protocol (XMPP) [42], the Internet Message Access Protocol (IMAP) [11], the Transport Layer Security (TLS) protocol [26], the Point-to-Point Protocol (PPP) [60], and the Internet Protocol (IP) [33]. The list includes both text-based and binary protocols. However, since the first category contains fields of arbitrary length where the decompression overhead is more evident, we decided to focus our study on three popular text-based protocols: HTTP, XMPP and IMAP. For each protocol we selected a number of implementations, summarized in Table 1. The columns *Native* and *External* show if the compression is natively supported by the application or if it is provided by an external component.

HTTP - Starting from version 1.1, HTTP supports compression of the HTTP *response* body using different compression algorithms (including Deflate) [17]. While the specification only covers the compression of the response body, we manually verified that several HTTP server im-

HTTP server	No.	Perc.
Apache HTTPD	248	24.8%
NGINX HTTPD	202	20.2%
Google HTTPD	81	8.1%
MS IIS HTTPD	64	6.4%
Apache Tomcat	22	2.2%
Others (20 servers)	102	10.2%
Unknown	218	21.8%
Errors	63	6.3%
Tot. no. of domains	1000	100%

(a) HTTP servers of the first 1000 domains of the Alexa DB of 2013-10-05.

XMPP server	No.	Perc.
ejabberd	56	52.8%
Openfire	11	10.4%
Prosody	9	8.5%
jabberd2	3	2.8%
Tigase	2	1.9%
Other (1 server)	1	0.9%
Unknown	1	0.9%
Errors	23	21.7%
Tot. no. of domains	106	100%

(b) XMPP servers of the 106 domains from xmpp.net of 2013-09-03.

IMAP server	No.	Perc.
Dovecot	31	42.5%
Courier	19	26.0%
Zimbra	6	8.2%
Cyrus	3	4.1%
MS Exchange	2	2.7%
Others (5 servers)	6	8.2%
Unknown	6	8.2%
Servers discovered	73	100.00%

(c) IMAP servers of the first 1000 domains of the Alexa DB

Table 2: Service detection for HTTP, XMPP, and IMAP servers

plementations additionally support the compression of the *request* body. Table 2a shows the result of the HTTP *service detection*[1] in order to identify the most popular HTTP server implementations among the top 1000 domains of the Alexa Top Sites database[2]. From Table 2a, we selected Apache HTTPD 2.2.22 [53] and Apache Tomcat 7 [52] as they are available for GNU/Linux. The former supports message decompression via the module `mod-deflate`, while the latter can be extended with third-party filters. In this paper, we used the 2Way HTTP Compression Servlet Filter 1.2 [37] (2Way for short) and Webutilities 0.0.6 [32].

In our experiments, we considered three use cases that may benefit from request compression: distributed computing, web applications, and sharing static resources. For Apache HTTPD, we selected gSOAP 2.8.7 [59] to develop SOAP-based RPC servers, CSJRPC 0.1 [9] to develop PHP-based JSON RPC servers, the PHP Apache module [55] (mod-php, for short) to develop PHP-based web applications, and WebDAV [21] (as implemented by the built-in Apache module mod-dav) to share static files. For Tomcat, we selected Apache CXF 2.2.7 [51], Apache Axis 2 [50], jsonrpc4j 1.0 [15], json-rpc 1.1 [41], and libjson-rpc 1.1.2 [7].

We test web servers with the following HTTP request:

```
POST $resource HTTP/1.1\r\n
Host: $domain\r\n
Content-Encoding: gzip\r\n
\r\n
$payload
\r\n
```

where `$resource` is the path to the resource, `$domain` the web server domain, and `$payload` is the compressed payload, or the *compression bomb*. The type of payload varies according to the implementation under test, i.e., JSON or SOAP message requests, and HTML form parameters. For example, the SOAP compression bomb is the following:

```
<soapenv:Envelope [...]>
   $spaces
   <soapenv:Body>[...]</soapenv:
      Body>
</soapenv:Envelope>
```

where `$spaces` can be, for example, a 4GB-long string of blank spaces. Once compressed, this payload is reduced to about 4MB with a compression ratio of about 1:1000, a value close to the maximum compression factor that can be achieved with Zlib, i.e., 1:1024 [19]. It might be possible to generate payloads with higher ratios, for instance, by modifying the compressor to return shorter, but still legal, strings. However, in this paper, we did not investigate this direction and leave this as future work.

XMPP - XMPP is an XML-based protocol offering messaging and presence, and request-response services [43, 44]. XMPP is at the core of several public and IM services, such as Google Talk, in which users exchange text-based messages in real-time. We performed service detection on the list of XMPP services available at `xmpp.net`[3]. Table 2b shows the result of the service detection. We selected the five most popular XMPP servers for our tests: ejabberd 2.1.10 [38], Openfire 3.9.1 [27], Prosody 0.9.3 [56], jabberd2 2.2.8 [54], and Tigase 5.2.0 [58].

To test XMPP servers, we used a similar trick as used in SOAP compression bombs. The highly-compressed XMPP message (i.e., *xmppbomb*) is the following:

```
<?xml version='1.0' ?>
  <stream:stream
```

[1] *Service detection* is a technique to identify the name of a network service by analyzing the server response against a database of fingerprints. In this paper, we used the Nmap Security Scanner tool [30]).

[2] See `http://www.alexa.com/`

[3] `xmpp.net` maintains a publicly accessible list of XMPP services: `http://xmpp.net/services.xml`. We retrieved it on 2013-09-03 and it contained 106 domains.

```
$spaces
to='server' xmlns='jabber:
    client'
[...]>
```

where $spaces can be a 4GB-long string of blank spaces. Also in this case, the compression ratio is about 1:1000.

IMAP - IMAP is a command-response protocol that allows a client to manage emails and mailboxes on a remote server [11]. The protocol supports compression algorithms to reduce the size of both commands and responses [23]. We obtained a list of popular IMAP servers from the first 1000 domains of the Alexa database. We first resolved the Mail eXchange (MX) domain and then we performed service detection. Table 2c shows the results of the service detection. As opposed to HTTP, we report percentages of the total number of the discovered servers because the majority of MX domains do not offer IMAP access to the email boxes. Two of the five IMAP servers were available for GNU/Linux and supported data compression: Dovecot 2.0.19 [16] and Cyrus 2.4.12 [8].

To test for IMAP, we crafted an email which can be compressed with a ratio of about 1:1000. The structure of the email is the following:

```
From: sender@foo\r\n
To:   receiver@foo\r\n
Subject: I am a bomb!\r\n
$spaces
```

where $spaces can be a 4GB-long string of blank spaces[4].

3 Decompression Security Risks

Applications can use the Deflate decompression algorithm in three main ways. First, they can invoke the functions provided by widely available libraries, such as `zlib` for `C` and `java.util.zip` for Java. Second, they can adopt a high-level wrapper built around one of the previously mentioned libraries (e.g., the `zlib` module in Python or the `Zlib` module in PHP). Finally, applications can implement their own version of the Deflate algorithm. Either way, in this paper we show that it is not trivial to properly decompress user-generated data streams. The risk arises from three aspects of the compression/decompression process:

1. Decompression is a computationally intensive task entailing an extensive use of CPU, memory, and disk space. If not properly limited, this process can be abused to stall an application and cause a denial of service;

2. Decompression amplifies the amount of data that software needs to process. Other components may not be designed to handle this volume of data. Therefore, the use of other functionalities on compressed data may result in the exhaustion of resources for components that were otherwise considered secure;

3. Compressed data can often be pre-computed by an attacker, thus creating a largely unbalanced scenario in which the input can be sent very fast, but the server needs to invest a lot of resources to process it. Moreover, the compressed input can often be meaningless or even malformed, because applications are often designed to discard bad inputs only *after* they are entirely decompressed.

3.1 The Past: Zip Bombs and Billion Laughs

Abusing data amplification to cause application or system denial of services is an old trick. The first documented DoS attack via a highly-compressed file archive dates back to 1996 when an attacker uploaded a malicious compressed file archive (a zip bomb) to the Bulletin Board System (BBS) of Fidonet, waiting for the system administrator to decompress it [1]. The classic zip bomb was a 42-kilobyte zip file archive that contained five nested layer of compressed files whose total size amounted to 4.5 petabytes. In 2001, zip bombs were used by attackers as email attachments [57] to disable anti-virus software designed to scan incoming messages [2].

A second popular exploitation of data amplification flaws was the so-called Billion Laughs attack [49] in 2003 (CVE-2003-1564). The Billion Laughs attack, also called the Exponential Entity Expansion attack, is an attack that exploits resource exhaustion vulnerabilities of XML document parsers when processing recursive entity definitions. An attacker may exploit this behavior by crafting a valid XML document (an XML bomb) which will cause the parser to generate an exponential amount of data. This results in CPU monopolization and memory exhaustion that can be exploited to mount a denial of service attack. This vulnerability was first reported in 2003 as a weakness of `libxml2` (CVE-2003-1564), an XML parser library. The same vulnerability was later discovered in some network servers, e.g., in 2009 in WebDAV as implemented by Apache HTTPD (CVE-2009-1955), and in a number of XMPP servers in 2011 (see, for example, CVE-2011-1755 and CVE-2011-3288).

[4] Due to limitations of the IMAP servers, in our experiments we used a 2GB-long string.

3.2 The present

Despite the popularity of previous attacks, a quick look at current protocol specifications, coding rules, and design patterns suggests that developers remain mostly unaware of the risks of using data compression. The risks of using compression are often overlooked, and guidelines on the proper way to handle compressed messages are either misleading or completely missing. In the rest of this section we briefly describe what protocol specifications, design patterns, and secure coding practices mention about the security issues related to data compression. Then, in Section 4, we show how this lack of common knowledge and understanding about the possible decompression attack vectors leads to a multitude of mistakes in many popular applications and protocols. Finally, in Section 5, we show experiments and the software vulnerabilities that we discovered on our case studies.

3.2.1 Protocol Specifications

A closer look at the specifications of the case studies revealed that none of them discuss potential security issues related to the use of data compression at the protocol level.

The Deflate specifications are mainly concerned with data integrity issues, suggesting developers implement means of validating the integrity of compressed data [13]. The HTTP protocol is concerned with loss of data confidentiality and unauthorized access, e.g., via path traversal attacks [17]. HTTP also addresses other DoS-related issues, such as broken clients when handling the status code 100 and with HTTP proxies [17]. The XMPP stream compression [25] does not describe any XMPP-specific security concern due to the use of data compression. Instead, it refers to SSL/TLS [26], which is concerned with data leakage, buffer overrun in the compression library, and enforcing packet size limits for uncompressed data. However, the specifications do not elaborate on how these concerns apply to XMPP, and therefore the developer may be left to personal interpretation that, ultimately, translates into vulnerable implementations. Finally, the IMAP compression specification [23] refers again to the SSL/TLS specifications for everything related to the decompression process.

3.2.2 Security Design Patterns

Security design patterns [3] are used to prevent vulnerabilities during the software design phase as well as to mitigate security risks. They address security concerns at a high level of abstraction (i.e., *DoS Safety*, *Compartmentalization*, and *Small Process* [24]) but unfortunately lack the details to address the specific concerns at the implementation level.

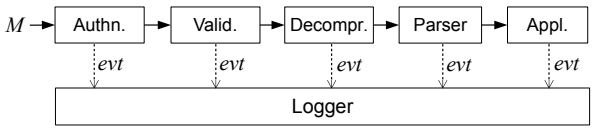

Figure 1: Message Pipeline

3.2.3 Secure Development Rules

Secure development rules suggest ways to write secure source code via secure code patterns (See, for example, coding rules for Java [29] and C++ [46]) or by means of software testing (see, for example, the OWASP Testing Guide [31]). The only existing rule on processing securely compressed data suggests a technique to validate a zip archive file before decompressing it. Sadly, the rule proposes an insecure technique, making the applications that implement it vulnerable to DoS via disk space exhaustion. We give more details of this rule in Section 4.2.

4 Common Pitfalls

In our study, we analyzed the implementation and documentation of our case studies, the protocol specifications (i.e., [11, 13, 17, 23, 25, 42]), and the software development best practices (i.e., [3, 24, 29, 31, 46]) looking for proper and incorrect ways to handle data compression. In this section, we distill our findings into 12 common pitfalls we observed at the implementation, specification, and configuration levels. Table 3 shows the list of pitfalls and maps them to the implementations of Table 1 that are affected by them.

4.1 Implementation Level

We start our survey of common compression-related mistakes by looking at the software implementation. As mentioned in Section 3, the decompression of a user-provided input is a delicate task that is prone to many errors. Software developers may be unaware of or underestimate the risks involved in this process, leading to implementation mistakes that can introduce denial of service vulnerabilities in the final product.

In this section, we list common pitfalls using a pipeline that processes incoming compressed requests. A pipeline is a linear chain of processing units in which the output of a unit is the input of the following one. Data pipelines are used to process incoming messages in blocks, in which each unit processes a single piece of information at the time, and provides an input to the next unit. Data compression can be used at different stages of message processing. For instance, it can be the first processing unit,

Prot.	Network Service		Implementation						Specification			Configuration		
			Impr. Input Val.	No Authn.	Int.-Unit Comm.	Log. Msgs.	Unbound. Mem.	Unbound. CPU	Misl. Doc.	Err. Best-Pract.	API Incons.	Insuf. Options	Default Values	Decentr. Pars.
XMPP	ejabberd		-	-	-	-	-	-	-	-	-	-	-	-
	Openfire		-	-	-	✗	✗	✗	-	-	-	✗	✗	-
	Prosody		-	✗	-	-	✗	✗	-	-	-	✗	-	-
	jabberd2		-	-	-	-	-	-	-	-	-	-	-	-
	Tigase		-	-	-	-	✗	✗	-	-	-	✗	✗	-
HTTP	Apache HTTPD	Static document	✗	-	-	-	✗	✗	✗	-	-	✗	-	-
		mod-php scripts	✗	-	✗	-	-	✗	✗	-	-	✗	-	-
		mod-php CSJRPC	✗	-	✗	-	-	✗	✗	-	-	✗	-	-
		mod-gsoap	✗	-	✗	-	-	✗	✗	-	-	✗	-	✗
		mod-dav	-	-	-	-	-	✗	-	-	-	-	-	-
	Apache Tomcat	Axis2	✗	-	-	-	✗	✗	✗	-	✗	✗	-	✗
		CXF	✗	-	-	✗	-	✗	✗	-	✗	✗	-	✗
		jsonrpc4j	✗	-	-	-	-	✗	✗	-	✗	✗	-	✗
		json-rpc	✗	-	-	-	✗	-	✗	-	✗	✗	-	✗
		lib-json-rpc	✗	-	-	-	✗	-	✗	-	✗	✗	-	✗
	Axis2 standalone		✗	-	-	-	✗	✗	✗	-	✗	✗	-	✗
	gSOAP standalone		✗	-	✗	-	-	✗	✗	-	✗	✗	-	✗
IMAP	Dovecot		-	-	-	-	-	-	-	-	-	-	-	-
	Cyrus		-	-	-	-	-	-	-	-	-	-	-	-

Table 3: Distribution of the pitfalls within the implementations under test

as it is for XMPP messages, or it can be placed after the message parsing, as implemented by the HTTP message processing.

Figure 1 shows a generic data pipeline to process an incoming message M. This pipeline is extracted from our case studies and is used to guide the description of the pitfalls. The pipeline has four processing units: user *authentication*, message *validation*, message *decompressor*, and message *parser*. The user authentication verifies that the request is sent by an authenticated user. This step may not be present in certain protocols, e.g., HTTP. The message validation unit implements a decision procedure to establish whether the incoming message can be accepted. The decompressor implements the data decompression algorithm to reconstruct the original message. Finally, the message parser performs a syntactic analysis of the message according to the rules of the communication protocol. As we already mentioned, the actual order of the blocks can be different, to reflect the protocol specification. The four units can send messages to the logger in order to store errors or unusual events in a log file. Finally, the output of the pipeline is consumed by the *application logic*. Here, by application we refer to an abstract representation of a message consumer. The consumer can be the rest of the software, e.g., a web application, as well as an additional component used to support application software, e.g., a web-based RPC framework.

4.1.1 Improper Input Validation during Decompression

One of the first mistakes we observed in our study is related to the erroneous way in which the size of incoming compressed messages is validated. We observed three ways to validate the message size: validation of the compressed message size, validation of the decompressed message size, and validation of the compression ratio of the message.

The first approach is the most straightforward to implement. Unfortunately, it is hard to estimate the size of a message by looking at its compressed form. For instance, while accepting an input no longer than 1MB could be insufficient for many types of data (e.g., when uploading a compressed picture), the same value is already sufficient for an attacker to generate extremely large decompressed output (e.g., by compressing a very repetitive string of bytes) that may cause an application denial of service.

The second approach consists in checking the size of the message by setting a limit to the amount of decompressed data. While this is a better solution, the validation of the decompressed size of an object is not straightforward to implement. To the best of our knowledge, we are not aware of any technique that allows one to compute the uncompressed size without first decompressing the input. Unfortunately, if the application needs to fully decompress the data before checking its size, it cannot protect itself against DoS attacks. However, many li-

braries (such as `zlib` and other language-specific wrappers) allow the application to decompress the incoming message in chunks. The size of each chunk is controlled by the application, which creates two buffers to store the compressed and the decompressed data streams. These two buffers are passed to the decompressor function of the library, which reads from one and writes to the other. The decompressor function returns when there are no more input bytes to process or when the output buffer is full. At this point, the application can provide more data or empty the output buffer. This interface allows an application to decompress a large message a piece at a time, constantly monitoring the amount of decompressed data. If a threshold is reached, the application can reject the input without the need to fully decompress the entire message.

The third approach consists of calculating the compression ratio. The compression ratio is the ratio between the sizes of the compressed and the decompressed messages. If the value exceeds a certain threshold, the decompression is halted and the message discarded. However, the compression ratio may vary according to the redundancies contained in the original message and it may cause the rejection of valid inputs. Moreover, the problem of deciding the appropriate threshold for the ratio must be solved. This may not be a simple task, as its value may depend on the protocol itself and on the way it is used by the application. Developers may leave the choice to educated guesses, experience, or experiments. This can result in under- or overestimation of this parameter. The former may increase the risk of rejecting valid messages, while the latter may introduce an uncontrolled use of resources.

To summarize, the first approach is a security mistake, the second is correct (if properly implemented), and the third one is potentially risky.

Apache HTTPD and gSOAP standalone provide examples of all three of these approaches. The first, vulnerable, approach is used by mod-deflate when it decompresses data for mod-php or mod-gsoap. This approach is insecure and we will discuss the details of this vulnerability in Section 5.4. When a message enters Apache HTTPD, it is processed by a chain of filters which transform the message and perform additional checks. The core module of Apache HTTPD offers a basic filter that allows setting a limit on the size of any incoming request body[5]. If the body exceeds the limit, then the message is rejected. However, the limit refers only to the compressed body size and it may render applications vulnerable to resource exhaustion.

The second approach is also implemented by mod-deflate, this time when it decompresses XML messages for mod-dav. The core module of Apache HTTPD offers a second parameter to limit the size of XML body objects[6]. As opposed to the previous one, the limit correctly applies to the decompressed form of the body. The third approach is now implemented by the patched versions of mod-deflate and gSOAP standalone. mod-deflate allows the user to configure a threshold for the compression ratio and the number of violations of the ratio that are allowed. gSOAP instead has the ratio built into the source code. In this case, the decompression is halted upon one violation.

4.1.2 Use of Compression before Authentication

We observed a great variety of practices for enforcing user authentication before the message decompression. For example, authentication is mandatory in SSL/TLS [14], recommended in XMPP [25], and undefined in IMAP [23]. In some cases, the implementation may even diverge from the specifications to postpone the use of compression (e.g., as done by OpenSSH), or to use compression where not prescribed (e.g., decompression of HTTP requests). This great variety of cases clearly indicates the lack of a consistent best practice. This may lead developers to underestimate the risk and overlook the recommendation. For example, we discovered that Prosody accepts compressed messages before user authentication, thus violating the recommendation of the XMPP protocol [25].

4.1.3 Improper Inter-Units Communication

When a processing unit in the pipeline detects a fault, i.e., the buffer limit is reached, then the unit should halt and notify the other units and the logger of this event. The communication between units can be direct or indirect via a third-party component, i.e., the pipeline manager. If a unit does not halt the execution, then the application may continue to consume resources until the resources are exhausted.

We observed this problem in Apache HTTPD in the interaction between mod-deflate and mod-php. mod-php can limit the size of the incoming request body via the parameter `post_max_size`. This parameter applies to the amount of data received in input by mod-php. However, if the incoming message is compressed, then it is first decompressed and then passed to mod-php. In this case, once the limit is reached, mod-php has no means to signal mod-deflate to stop processing the incoming data. As result, mod-deflate will keep on decompressing data, thus wasting system resources. The same problem was also observed between mod-deflate and mod-gsoap. The

[5] Via the configuration parameter `LimitRequestBody`

[6] Via the configuration parameter `LimitXMLRequestBody`

developers of mod-php and mod-gsoap confirmed this behavior.

4.1.4 Logging Decompressed Messages

Log files are used to store events generated by a running program (or an entire operating system), and they are particularly important to monitor the execution of background processes that have little interaction with the user, as is the case for web services. The information stored in log files can cover a wide range of events, including warning messages, malfunction errors, and verbose reports of the current activity of a given process.

While the use of log files is a good practice, both developers and users should carefully select the frequency and the verbosity of the generated events. An excessive level of verbosity may be useful to debug unusual behaviors, but it may have side effects from a security perspective. In particular, when the unusual behavior is caused by compressed data, developers and users may underestimate the resources needed to generate and store the event and, as a result, the application can exhaust all the available resources.

For instance, we observed this type of issue in Apache CXF. Upon receiving an invalid request, Apache CXF stores the request in a temporary file, and then adds an entry in the log file containing the first 100 KB of the request. However, if the invalid message is compressed, Apache CXF decompresses the entire request on disk just to extract the 100KB header to log. As a result, in our experiments we observed that a single request of 4MB (containing 4GB of data after decompression) can cause Apache CXF to store on the disk 8 GB of temporary data.

4.1.5 Unbounded Resource Usage

In general, the best way to avoid DoS attacks against an application is to properly limit the size of decompressed inputs. However, whenever such thresholds need to be set very high because the application has to accept large amounts of user-provided data, the developers should carefully design the code to bound the CPU and memory usage of the decompression routine. We found unbounded CPU and memory usage in different applications. Below, we discuss these pitfalls separately.

Unbounded Memory - The data amplification introduced by a decompressor may be underestimated by the developers who may leave buffers uncontrolled on the size both in peripheral (e.g., input validation components) and internal components (e.g., message parsers). For example, we observed this type of mistake in Prosody and json-rpc. Prosody decompresses incoming messages in chunks. Each chunk is passed to the XMPP message parser, which internally accumulates the chunks without bounds before the processing. A single highly-compressed message results in consuming all the available memory and, finally, being terminated by the operating system. A similar behavior was observed in json-rpc. Upon receiving a JSON request, json-rpc accumulates the uncompressed request into a memory buffer until no more memory is available.

Unbounded CPU - Decompressors are *CPU-bound* software procedures, as they tend to monopolize the CPU usage for the entire duration of the operation. If an attacker can influence their execution, then she may degrade system performance and mount a CPU DoS attack.

Unfortunately, best practices in secure software coding [29, 46] do not provide techniques for controlling CPU-bound tasks. As a result, developers may not be aware of the risks and leave the CPU usage unbounded. One way to control CPU-bound operations is to introduce idle time intervals in which a task is suspended for a period of time. The size of the interval and the moment in which it is introduced can be decided at run-time by taking into account the current status of the process. For example, a task may introduce idle times in order to keep constant the bandwidth of the decompressor throughput.

We observed an unbounded CPU usage in many implementations of our case studies, including Apache mod-deflate, Apache CXF, Webutilities, 2Way, Prosody, and Tigase. On the other hand, we found that CPU controls via idle time intervals were already implemented by ejabberd and jabberd2.

4.2 Specification Level

In this section, we review common data compression pitfalls stemming from imprecise protocol specifications, misleading documentation, and erroneous best practices.

4.2.1 Misleading Documentation

Modern software is a collection of reusable components. The developers of each component should carefully document the security risks related to the usage of their own components in order to allow a more secure integration. For this purpose, we reviewed the documentation of Apache mod-deflate, Webutilities, 2Way Filter, Axis 2, and gSOAP. None of the above components discuss the security risks related to the use of data compression. Even worse, the user documentation of Webutilities and 2Way even reassure their users that (i) a developer can plug in the decompression "*without changing the source code*" [37], and (ii) that "*nothing [else is] needed*" [32] from the user. In general, misleading documentation may create a false sense of security in which a developer may believe that she does not need to address the problem

in her application because the possible security concerns are already addressed at the underlying level.

4.2.2 Erroneous Best Practices

Unawareness and underestimation of the risks in using data compression may also affect best practices. In our review of secure coding rules, we found out that the security risks specific to the decompression of compressed messages are not properly addressed. Design patterns are too generic to address the specificity of data compression, and secure code patterns address only the risks of storage exhaustion due to zip archive bombs [29]. Finally, testing guides only propose tests against information leakage vulnerabilities caused by the simultaneous usage of data compression and data encryption [31].

Interestingly enough, we discovered that the only available pattern on the topic is also insecure[7]. In fact, it suggests developers verify the decompressed size reported in the file headers before accepting a Zip archive. Unfortunately, this information can be easily forged by an attacker to contain any arbitrary value, thus successfully bypassing the security checks. We disclosed the flaw in the pattern to the authors, who marked it as vulnerable and provided a newer, secure version.

4.2.3 API Specifications Inconsistency

Data compression is an optional feature and it is transparent from the point of view of the application. However, in our review we found out that the use of compression may also violate the contract of other APIs. For example, the method `ServletRequest.getContentLength()` of J2EE 7 is supposed to return the length of the request body as it is made available through the input stream. The *input stream* refers to the object used by the servlet developer to access the content of the body of the message. This parameter may be used in the logic of the servlet to allocate a buffer, or to accept or reject the resource. Unfortunately, when the HTTP decompression is enabled, `getContentLength` returns a wrong value. In our experiments, we verified that `getContentLength` returns the value stored in the `Content-Length` HTTP header, while the input stream contains the much larger uncompressed body.

4.3 Configuration Level

In this section, we list common pitfalls in the way compression can be configured in different services.

[7] See https://www.securecoding.cert.org/confluence/display/java/IDS04-J.+Safely+extract+files+from+ZipInputStream

4.3.1 Insufficient Configuration Options

In Section 4.1, we described a number of secure approaches to handle data compression at the implementation level. These solutions allow one to control the resource consumption by setting limits to the amount of resources to be used. The actual threshold may vary depending from a number of factors. For example, a web application that manages an online storage service may require the web server to accept large input messages to upload big files. In order to allow use of network services in different scenarios, the resource limits need to be parametric and the proper thresholds should be selected by the user during the deployment phase.

However, we observed that the number of configuration parameters provided by common servers is often insufficient. This is mainly due to the lack of implementations of the resource consumption controls. For example, Prosody only allows the use of data compression to be enabled or disabled; it does not offer any parameters to specify the maximum size of decompressed data to be accepted, nor the output bandwidth of the decompressor.

4.3.2 Insecure Default Values

Recently, it has been demonstrated that compression may be problematic when used together with data encryption, as it can lead to information leakage (e.g., CRIME [39] and BREACH [36]). The exploitation of these flaws may depend on the deployment scenario and the capability of the attacker to choose the plaintext. For these reasons, the use of data compression should be at the discretion of the user, who should assess the characteristics of the deployment scenario and the usage of the service.

While data compression should be an optional feature, in our survey we observed network service configurations in which data compression was enabled by default, such as in Openfire and Tigase.

4.3.3 Decentralized Configuration Parameters

The time to identify and resolve an attack is critical to contain the costs of a cyber-incident [34]. The response to an attack may require changing the configuration of a running system, and this task is simplified if the security-relevant configuration parameters are easily accessible to the security response team. In our survey, we verified that this is not always the case. For example, compression in CXF can be enabled in two ways. First, it can be enabled by adding the decompressor filter in the configuration of the servlet (i.e., the `web.xml` file). Second, it can be enabled within the Java code of the service using a Java annotation. In both cases, in order to disable the compression, the security response team needs to modify the configuration of all the servlets or, in the worst

case, to modify even their source code. However, the response team may not have access to the source code. As a result, they may need to involve developers in this activity, which may increase the time required to react to an attack.

5 Analysis

In this section, we describe the experiments that we performed to detect resource exhaustion vulnerabilities. This section is organized as follows. In Section 5.1, we describe the experiment setup. Then, in Section 5.2 we present our results, and in Section 5.3 we discuss them. Finally, in Section 5.4, we present the complete list of vulnerabilities.

5.1 Experiment Setup

The services we tested in our experiments are developed using different programming languages, including C, C++, Java, PHP, Erlang, and Lua. While automatic code-based techniques could be used to detect software vulnerabilities, these techniques are language-dependent and therefore cannot be used in our analysis. For this reason, we followed a black-box testing approach in which we probed the implementation with malicious inputs, and then measured the resource consumption and the service availability.

Tests - Each test consists of two parts: *baseline test* and *attack*. The baseline test measures the point of reference for resource consumption and service availability when compression is not used and there are no attacks. Baseline measurements are sampled over a period of 60 seconds by probing the target with 4MB-long, honest protocol messages. The attack measures instead the resource consumption and service availability when stressing the implementation with malicious messages sent by one, 12, and 24 simultaneous attackers—a very low number compared with the number of clients that participate in most of the distributed DoS attacks. Malicious messages are presented in Section 2. Both the baseline and the attack requests have a 4MB payload in order to rule out the overhead of transferring the data over the network.

Testbed - We performed the experiments on a testbed of three machines to host, respectively, the server, the attackers, and the honest client. The server machine runs the IUT and the internal monitor, the attackers execute the test cases to send highly-compressed messages to the server, and the client executes baseline tests to measure the availability of the service.

To test the HTTP services, we used (i) a 4MB static file resource; (ii) two PHP scripts for mod-php, each using a different PHP interface[8] to read the content of the request body; and (iii) PHP, Java, and C++ classes and functions to be deployed as a web services with CSJRPC, Axis 2, CXF, jsonrpc4j, json-rpc, lib-json-rpc, and gSOAP (both mod-gsoap and standalone). To test the XMPP and the IMAP servers, we created user accounts for both the attackers and the honest client. In our tests, we considered different IUT configurations. For example, we tested Apache HTTPD, mod-php, Apache Tomcat, ejabberd, and jabberd2 with different maximum message sizes.

Prot.	Network Service		CPU	Mem.	Disk
XMPP	ejabberd		-	-	-
	Openfire		✗	✗	✗
	Prosody		✗	✗	-
	jabberd2		-	-	-
	Tigase		✗	✗	-
HTTP	Apache HTTPD	Static document	✗	✗	-
		mod-php scripts	✗	-	-
		mod-php CSJRPC	✗	-	-
		mod-gsoap	✗	-	-
		mod-dav	✗	-	-
	Apache Tomcat	Axis2	✗	✗	-
		CXF	✗	-	✗
		jsonrpc4j	✗	-	-
		json-rpc	-	✗	-
		lib-json-rpc	-	✗	-
	Axis2 standalone		✗	✗	-
	gSOAP standalone		✗	-	-
IMAP	Dovecot		-	-	✗
	Cyrus		-	-	✗

Table 4: Resource Consumption Vulnerabilities

Monitoring - We monitored the IUT with a combination of internal and external monitors. The external monitor measures the service availability in terms of number of honest messages processed per second. We used the client to continuously provide the server with honest messages and measure the server's response time. The internal monitor is a modified version of `pidstat` from the sysstat tool suite [20], which repeatedly polls the `/proc` filesystem. It measures (1) CPU usage, (2) virtual size (VSZ) and the resident set size (RSS) memory and (3) disk I/O of the processes associated with the IUT.

5.2 Results

Table 4 shows a summary of the results of our experiments on the 19 implementations. Out of them, only four implemented the compression in a secure way. Both ejabberd and jabberd2 keep a constant resource usage even during multiple simultaneous attacks. In fact, through a manual source code analysis, we were able to

[8] `php://input` interface and `$HTTP_RAW_POST_DATA`

verify that both servers implement two separate mechanisms to limit the use of memory and CPU usage during decompression. Table 4 shows a possible disk-based DoS attack against Dovecot and Cyrus; however, this is not to be considered a vulnerability, as IMAP servers are designed to store on disk the email used as the attack vector. All the other 15 services we tested showed an uncontrolled increase in at least one of the three system resources, making them potentially vulnerable to decompression-based DoS attacks. All results were reported to (and confirmed by) the developers of the corresponding applications and libraries.

Table 5 shows an excerpt of our experiments on three vulnerable implementations: Prosody, Apache HTTPD, and Apache CXF with the WebUtilities filter. For each implementation in Table 5, we performed four experiments (col. *Attackers*): the baseline and three attacks respectively with one, 12, and 24 parallel attackers. For each experiment, we report the requests response time (col. *Resp.*), the median value of the CPU usage (col. *Mdn*), the maximum virtual size memory allocated (col. *VSZ*), the maximum resident set size[9] allocated (col. *RSS*), and the total amount of data written to disk (col. *WR*). The columns *mult* report the ratio between the measured value during the attack and the baseline. In the rest of this section we detail the results of Table 5.

Prosody allocates up to 7.8GB of RSS memory and 22GB of VSZ memory when processing a single malicious request. The process is then killed by the operating system due to a system out of memory error. Even worse, Prosody also exhibits the same behavior when we sent the malicious message before the user authentication. Similarly as seen for Prosody, the measurements for Apache CXF show a significant resource utilization: starting from 0.03 GB of the baseline, Apache CXF can write about 1 TB, which is 3243 times the baseline. These value indicates that Apache CXF may be vulnerable to disk space exhaustion. Other services, while still potentially vulnerable, had a more controlled behavior. For instance, Apache HTTPD monopolizes the CPU at about 100% for 17 seconds with a single attacker, and up to 140 seconds by sending 24 malicious payloads in parallel.

5.3 Experiment Results Discussion

In this section, we discuss three factors that play an important role in our black-box experiments.

First, the baseline sets the amplitude of the proportions with the measurements done during the attack. The choice of the baseline is crucial because it can affect the conclusion of the analysis. As we already explained, the choice of the baseline was to offer a reference point that rules out network delay. This results in ratio values that cannot be directly transferred to real-size servers.

Second, the quantification of the severity of the observed degradation heavily depends on a number of variables that our testbed does not realistically reproduce, e.g., number of CPU cores, size of main memory, disk space, and average load of the server. Small-size servers can be DoSed with few requests, while large and powerful servers may be able to sustain a higher load before showing signs of resource exhaustion. As a result, to obtain an externally visible effect, it may be necessary to use a larger number of simultaneous attackers.

Finally, it is hard to develop an automated procedure to detect DoS vulnerabilities on the basis of the data we collected. The measures do not offer an accurate view of the internal behavior of the application, and the figures depend on so many factors that sometimes it is hard to make a final conclusion. For this reason, we manually verified each case, often complementing the experiments with a source code analysis of the affected components. Moreover, we discussed each problem with the developers, and obtained confirmation of each vulnerability reported in this paper.

5.4 Vulnerabilities

Our experiments led to the discovery of nine vulnerabilities. After we completed our experiments, our results were also reproduced on other three additional XMPP network services (M-Link, Metronome, and MongooseIM), discovering resource exhaustion vulnerabilities also in these products as well.

We followed the principle of responsible disclosure and informed the developers, the community, and the security response teams. In most of the cases, developers reacted to our first reports and worked on a patch. If developers were unresponsive for over a month, we tried a second time and then alerted the US CERT to support the disclosure. Eventually, all the developers acknowledge the reported vulnerabilities. The way in which each product was patched is described in the rest of this section.

5.4.1 HTTP

Apache HTTPD - The component that caused CPU and memory consumption is mod-deflate. This affected mod-php, CSJRPC, and mod-gsoap applications. Unfortunately, mod-php and mod-gsoap developers are unable to solve this issue on their components, as they are unaware of a suitable interface to control mod-deflate. As a result, we escalated the issue to the Apache Security Team. The security team acknowledged the presence of the vulnerability, fixed it in Apache HTTPD 2.4.10, and disclosed

[9]The total RAM size of the server machine is 8 GB.

Prot.	Network Service	Attacks		CPU		Memory				Disk	
		No.	Resp.	Mdn	mult.	RSS	mult.	VSZ	mult.	WR	mult.
XMPP	Prosody	*baseline*	> 0 s	1%	x1	0.01 GB	x1	0.05 GB	x1	0.00 GB	-
		1	204 s	18%	x18	7.68 GB	x1397	10.71 GB	x210	0.00 GB	-
		12	222 s	21%	x21	7.60 GB	x1383	18.40 GB	x361	0.00 GB	-
		24	531 s	34%	x34	7.71 GB	x1402	22.96 GB	x451	0.00 GB	-
HTTP	Apache HTTPD	*baseline*	1 s	21%	x1	0.05 GB	x1	0.55 GB	x1	0.00 GB	-
		1	17 s	114%	x6	0.10 GB	x2	0.78 GB	x1	0.00 GB	-
		12	72 s	297%	x14	0.58 GB	x11	2.36 GB	x4	0.00 GB	-
		24	142 s	229%	x11	0.84 GB	x15	2.75 GB	x5	0.00 GB	-
	Apache CXF WU	*baseline*	1 s	57%	x1	0.33 GB	x1	3.03 GB	x1	0.03 GB	x1
		1	149 s	55%	x1	0.38 GB	x1	3.03 GB	x1	9.10 GB	x317
		12	1135 s	109%	x2	0.73 GB	x2	6.09 GB	x2	84.90 GB	x2958
		24	1296 s	109%	x2	0.44 GB	x1	3.08 GB	x1	93.07 GB	x3243

Table 5: Excerpt of the experiment results

it publicly (CVE-2014-0118). Our contribution was also rewarded by the bounty program of Hackerone[10].

Apache HTTPD developers implemented two new mechanisms to control CPU and memory consumptions that can be configured via the following new parameters: `DeflateInflateLimitRequestBody`, `DeflateInflateRatioLimit`, and `DeflateInflateRatioBurst`. The first parameter limits the maximum amount of memory that can be allocated to decompress incoming HTTP requests. The second enforces a ratio between the compressed and decompressed message. This mechanism also allows specifying the number of tolerated violations of the ratio before halting the decompression. While this mechanism limits the use of resources in the presence of highly-compressed messages (not necessarily compression bombs), it does not limit the amount of CPU used by the decompressor.

gSOAP - gSOAP standalone suffers from uncontrolled CPU usage. The developers acknowledged the presence of the vulnerability and released a patch. The patch implements a ratio-based technique similar to the mechanism implemented now by mod-deflate. However, as opposed to mod-deflate, the ratio is not parametric but it is built into the source code.

Webutilities and 2Way - Webutilities and 2Way filters are the components that cause unbounded CPU usage in Axis 2, CXF, and jsonrpc4j. After our reports, the developers of Webutilities fixed the vulnerability using a CPU throttling mechanism that can be configured via a new parameter, `decompressMaxBytesPerSecond`. This mechanism monitors the throughput in bytes per second of the decompressor and, if the limit is reached, it introduces idle time intervals in which the decompressor is suspended from execution for a short period of time. The developers of 2Way acknowledged the presence of the issue and are working on a patch (still unavailable at the time of writing).

Apache CXF - Apache CXF suffers from a disk exhaustion vulnerability. We reported the issue to the Apache Security Team. The security team acknowledged the presence of the vulnerability in two branches of the software, fixed it in version 2.6.14 and 2.7.11, and disclosed it publicly (CVE-2014-0109 and CVE-2014-0110). This vulnerability is described in Section 4.1.4.

json-rpc and lib-json-rpc - json-rpc and lib-json-rpc suffer from an uncontrolled memory vulnerability. Upon receiving a JSON request, both frameworks try to store all of the uncompressed data in a single memory buffer, causing an out of memory error. The developers acknowledged the issue and are currently working on a patch.

5.4.2 XMPP

The disclosure of the XMPP vulnerabilities was conducted with the involvement of the XMPP community. We supported the community in coordinating the disclosure and preparing a common secure notice about the multiple vulnerabilities. In total, our experiments directly discovered four vulnerabilities in three XMPP servers, and three other servers were found vulnerable during the disclosure. All the vulnerabilities are fixed and new versions of the servers are already available[11]. Also in this case, our contribution and efforts were rewarded by the bounty program of Hackerone[12].

Openfire - Openfire does not properly restrict the resources used in processing incoming XMPP messages (see, CVE-2014-2741 and VU#495476).

Prosody - Prosody suffers from memory exhaustion due

[10] See https://hackerone.com/reports/20861

[11] See `http://xmpp.org/resources/security-notices/`

[12] See https://hackerone.com/reports/5928

to an uncontrolled buffer (CVE-2014-2745) and allows unauthenticated users to use data compression (CVE-2014-2744).

Tigase - Tigase does not properly limit the memory used to process incoming XMPP messages (CVE-2014-2746).

M-Link - M-Link does not properly restrict the resources used in processing incoming XMPP messages (CVE-2014-2742).

Metronome - Metronome suffers from unbounded memory consumption (CVE-2014-2743) and allows the use of compression before user authentication (CVE-2014-2744, shared with Prosody).

MongooseIM - MongooseIM does not properly restrict the resources used in processing incoming XMPP messages (CVE-2014-2829).

6 Related Work

In this section, we review works that are related to this paper from different perspectives. In particular, we discuss attacks that exploit data amplification, leakage due to the use of data compression, worst-case complexity, and bandwidth exhaustion.

Data Amplification Attacks - To the best of our knowledge, zip bombs [1] and XML bombs [49] are among the first documented abuses of data amplification. We already discussed the details of these attacks in Section 3. Data amplification can also be achieved by using external servers for which the response size is bigger than the request size [40]. An attacker can spoof the network address of a victim and sends small request packets to a large number of servers. These requests trigger voluminous response traffic that accumulates on the network link of the victim and leads to bandwidth exhaustion. Similar to an asymmetry in the request/response traffic volume, our work considers an asymmetry in creating and processing compressed data for attack amplification.

Compression and Encryption - Data compression can lead to information leakage when used together with encryption. CRIME [39] and BREACH [36] are two attacks that exploit the change of size of a ciphertext due to the compression of the plaintext. These attacks target the SSL/TLS layer when used to carry HTTP conversations and rely on an attacker that is capable of performing a chosen-plaintext attack. These attacks and our work show orthogonal security issues in using data compression. While CRIME and BEAST aim at breaking the SSL/TLS encryption layer, our paper addresses software vulnerabilities due to the data amplification of decompression algorithms.

Algorithmic Attacks - Resource exhaustion can also result from the worst-case performance of the data structure algorithms [12] and rule matching algorithms [47]. Similarly to these attacks, in this paper we exploit the worst-case scenario of decompression algorithms in which the attacker can cause resource exhaustion with a compression rate of 1:1000.

Bandwidth Exhaustion - A variety of DoS attacks target network bandwidth exhaustion [6, 28, 48]. The Coremelt attack [48] and the Crossfire attack [28] achieve DoS through network bandwidth exhaustion of a targeted Autonomous System backbone routers. The attacker does not connect to the victim, but instead she uses machines under her control to exchange data over the link used by the victim. As both Coremelt and Crossfire work by exposing network links to high traffic, their effects can arguably be mitigated by using compression. However, the results presented in this paper demonstrate that there is a catch to such strategies, as compression bears risks for the communication's end nodes. Büscher and Holz [6] show that the vast majority of attacks launched by the DirtJumper/Ruskill botnet target HTTP port 80 [6] and that an average number of 185 DoS threads is sufficient to saturate the link between the botnet and the victim. Our results indicate that service disruptions can be achieved with a much lesser number if compression is used, thereby circumventing the detection and analysis proposed in the paper [6].

7 Future Work

As future work, we plan to investigate two directions: secure data compression, and automated detection techniques for resource exhaustion vulnerabilities.

Solving the mistakes presented in Section 4 is only part of the problem. Data compression introduces an unbalanced scenario in which the sender can generate *offline* compressed messages while the receiver needs to perform *online* decompression. This gives a large advantage to the attacker. The first direction of our research is to tackle this problem by studying new techniques that introduce *fairness* in data compression. The idea is to allow the receiver to decompress an incoming message when it has evidence that the sender has performed the compression online. The evidence can be provided by the means of single-use or session-based compression keys.

The problem of detecting resource exhaustion vulnerabilities has been addressed in the past [4, 5, 10]. However, existing techniques are fragmented and suffer from a number of limitations which hinder their use: they cannot scale to real programs (See [5]), are not fully automated (See [4]), and can be applied only to a subset of

the resource exhaustion vulnerabilities (See [10]). As a second research direction, we plan to develop an intelligent fuzzer which combines code analysis and blackbox testing: The code analysis can be responsible for extracting a set of program constraints that the fuzzer would then use to generate inputs using a constraint solver.

8 Conclusion

In this paper, we presented a study on the current use of data decompression in three popular network services that are at the core of modern web-based applications. We analyzed 19 network services and extensions, protocol specifications, and documentation looking for proper and incorrect ways to handle data compression. We grouped our findings into 12 common pitfalls that we observed at the implementation, specification, and configuration levels. Furthermore, in our tests we discovered and reported nine previously unknown vulnerabilities. While these problems have been now being patched, we believe that this paper shows how the risks of supporting data compression are still too often overlooked, even by very popular web and network services.

Acknowledgments

We thank the anonymous reviewers for the valuable comments. We also thank Gabriel Serme (SAP) for the valuable discussion that originated this line of research, and various members of the standardization and software development groups for fruitful discussions. In no specific order, we thank the Apache Security Team, Daniel Kulp (CXF), Stanislav Malyshev (PHP), Matthew Wild (Prosody), Andrzej Wójcik (Tigase), Guus der Kinderen (Openfire), Peter Saint-Andre (XMPP Standards Foundation), Philipp Hancke (XMPP Standards Foundation), Kevin Smith (XMPP Standards Foundation), Waqas Hussain (XMPP Standards Foundation), Robert van Engelen (gSOAP), and Rajendra Patil (Webutilities). This project was supported in part by the German Ministry for Education and Research (BMBF) through funding for the project 13N13250, H2020 ESCUDO-Cloud, and TU Darmstadt CASED/EC-SPRIDE.

References

[1] ACCESS DENIED. DFS Issue 55. `http://textfiles.com/magazines/DFS/dfs055.txt`, 1996.

[2] AERASEC NETWORK SERVICES AND SECURITY GMBH. Decompression Bomb Vulnerabilities. `http://www.aerasec.de/security/advisories/decompression-bomb-vulnerability.html`, 2009.

[3] ALUR, D., MALKS, D., AND CRUPI, J. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2001.

[4] ANTUNES, J., NEVES, N., AND VERISSIMO, P. Detection and prediction of resource-exhaustion vulnerabilities. In *ISSRE 2008* (Nov 2008), pp. 87–96.

[5] BURNIM, J., JUVEKAR, S., AND SEN, K. Wise: Automated test generation for worst-case complexity. In *ICSE 2009* (May 2009), pp. 463–473.

[6] BÜSCHER, A., AND HOLZ, T. Tracking DDoS Attacks: Insights into the business of disrupting the Web. In *Proc. LEET* (2012).

[7] BUTKO, A. JSONRPC Java implementation. `https://code.google.com/p/libjsonrpc/`, 2014.

[8] CARNEGIE MELLON UNIVERSITY. Project Cyrus. `https://cyrusimap.org/`, 2014.

[9] CAZI, M. CSJSONRPC - A PHP JSON-RPC Server. `https://github.com/mojtabacazi/CSJRPC`, 2014.

[10] CHANG, R., JIANG, G., IVANCIC, F., SANKARANARAYANAN, S., AND SHMATIKOV, V. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *CSF '09* (July 2009), pp. 186–199.

[11] CRISPIN, M. Internet Message Access Protocol - Version 4rev1. RFC 3501 (Proposed Standard), Mar. 2003. Updated by RFCs 4466, 4469, 4551, 5032, 5182, 5738, 6186, 6858.

[12] CROSBY, S. A., AND WALLACH, D. S. Algorithmic DoS. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, 2011, pp. 32–33.

[13] DEUTSCH, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[14] DIERKS, T., AND ALLEN, C. The TLS Protocol Version 1.0. RFC 2246 (Proposed Standard), Jan. 1999. Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176.

[15] DILLEY, B. JSON-RPC for Java. `https://github.com/briandilley/jsonrpc4j`, 2014.

[16] DOVECOT SOLUTIONS OY. Dovecot - Secure IMAP Server. `http://www.dovecot.org/`, 2014.

[17] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[18] GAILLY, J.-L., AND ADLER, M. gzip. http://www.gzip.org/, 2014.

[19] GAILLY, J.-L., AND ADLER, M. zlib. http://www.zlib.net/, 2014.

[20] GODARD, S. SYSSTAT. http://sebastien.godard.pagesperso-orange.fr/, 2014.

[21] GOLAND, Y., WHITEHEAD, E., FAIZI, A., CARTER, S., AND JENSEN, D. HTTP Extensions for Distributed Authoring – WEBDAV. RFC 2518 (Proposed Standard), Feb. 1999. Obsoleted by RFC 4918.

[22] GORDON, E., SPIELER, C., WHITE, M., AND HAASE, D. Info-ZIP. http://http://www.info-zip.org//, 2014.

[23] GULBRANDSEN, A. The IMAP COMPRESS Extension. RFC 4978 (Proposed Standard), Aug. 2007.

[24] HAFIZ, M., ADAMCZYK, P., AND JOHNSON, R. E. Growing a pattern language (for security). In *Proc. Onward! '12* (2012), ACM, pp. 139–158.

[25] HILDEBRAND, J., AND SAINT-ANDRE, P. XEP-0138: Stream Compression. http://xmpp.org/extensions/xep-0138.html, 2009.

[26] HOLLENBECK, S. Transport Layer Security Protocol Compression Methods. RFC 3749 (Proposed Standard), May 2004.

[27] IGNITEREALTIME. Igniterealtime Openfire. http://www.igniterealtime.org/projects/openfire/, 2014.

[28] KANG, M. S., LEE, S. B., AND GLIGOR, V. The Crossfire attack. In *Proc. SP* (2013), pp. 127–141.

[29] LONG, F., MOHLNDRA, D., SEACORD, R. C., SUTHERLAND, D. F., AND SVOBODA, D. *The CERT Oracle Secure Coding Standard For Java*. SEI Series In Software Engineering. Addison-Wesley, 2012.

[30] LYON, G. Nmap Security Scanner. http://nmap.org/, 2014.

[31] MEUCCI, M. ET AL. The OWASP Testing Guide 4.0. https://www.owasp.org/images/5/52/OWASP_Testing_Guide_v4.pdf, 2014.

[32] PATIL, R. webutilities. https://code.google.com/p/webutilities/, 2014.

[33] PEREIRA, R. IP Payload Compression Using DEFLATE. RFC 2394 (Informational), Dec. 1998.

[34] PONEMON INSTITUTE LLC. 2013 Cost of Cyber Crime Study: United States. Tech. rep., Ponemon Institute LLC, Traverse City, Michigan 49629 USA, October 2013.

[35] PONEMON INSTITUTE LLC. 2014 Global Report on the Cost of Cyber Crime. Tech. rep., Ponemon Institute LLC, Traverse City, Michigan 49629 USA, October 2014.

[36] PRADO, A., HARRIS, N., AND GLUCK, Y. SSL, Gone in 30 Seconds - A BREACH Beyond CRIME. http://breachattack.com/#resources, 2013.

[37] PREDIC8 GMBH. 2Way HTTP Compression Servlet Filter. http://predic8.com/gzip-compression-filter.htm, 2014.

[38] PROCESSONE. ejabberd - the Erlang Jabber/XMPP Daemon. http://www.ejabberd.im/, 2014.

[39] RIZZO, J., AND DUONG, T. The CRIME Attack. https://docs.google.com/a/trouge.net/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit#slide=id.g1e59c14c_1_54, 2012.

[40] ROSSOW, C. Amplification hell: Revisiting network protocols for ddos abuse. In *Proc. NDSS* (2014).

[41] SAIKIA, R. JsonRpc - Easy and Lightweight Json-Rpc Client/Server. https://github.com/RitwikSaikia/jsonrpc/, 2014.

[42] SAINT-ANDRE, P. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), Oct. 2004. Obsoleted by RFC 6120, updated by RFC 6122.

[43] SAINT-ANDRE, P. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 3921 (Proposed Standard), Oct. 2004. Obsoleted by RFC 6121.

[44] SAINT-ANDRE, P. Mapping the Extensible Messaging and Presence Protocol (XMPP) to Common Presence and Instant Messaging (CPIM). RFC 3922 (Proposed Standard), Oct. 2004.

[45] SALOMON, D. *Data Compression: The Complete Reference*. Springer-Verlang, 2007.

[46] SEACORD, R. C. *Secure Coding In C And C++*. SEI Series In Software Engineering. Addison-Wesley, 2006.

[47] SMITH, R., ESTAN, C., AND JHA, S. Backtracking algorithmic complexity attacks against a NIDS. In *Proc. ACSAC* (2006), IEEE Computer Society, pp. 89–98.

[48] STUDER, A., AND PERRIG, A. The Coremelt attack. In *Proc. ESORICS*, M. Backes and P. Ning, Eds., vol. 5789 of *LNCS*. Springer, 2009, pp. 37–52.

[49] SULLIVAN, B. XML Denial of Service Attacks and Defenses. http://msdn.microsoft.com/en-us/magazine/ee335713.aspx, 2009.

[50] THE APACHE FOUNDATION. Apache Axis. http://axis.apache.org/, 2014.

[51] THE APACHE FOUNDATION. Apache CXF: An Open-Source Services Framework. http://cxf.apache.org/, 2014.

[52] THE APACHE FOUNDATION. Apache Tomcat. http://tomcat.apache.org/, 2014.

[53] THE APACHE FOUNDATION. HTTP Server Project. http://httpd.apache.org/, 2014.

[54] THE JABBERD TEAM. JabberD XMPP Server. http://jabberd2.org/, 2014.

[55] THE PHP GROUP. PHP. http://www.php.net/, 2014.

[56] THE PROSODY TEAM. Prosody IM. http://prosody.im/, 2014.

[57] THE REGISTER. DoS Risk from Zip of Death Attacks on AV Software? http://www.theregister.co.uk/2001/07/23/dos_risk_from_zip/, 2001.

[58] TIGASE INC. Open Source and Free XMPP/Jabber Software. http://www.tigase.org/, 2014.

[59] VAN ENGELEN, R. A. The gSOAP Toolkit for SOAP Web Services and XML-Based Applications. http://www.cs.fsu.edu/~engelen/soap.html, 2014.

[60] WOODS, J. PPP Deflate Protocol. RFC 1979 (Informational), Aug. 1996.