FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Analysis of the Zlib's CVE-2022-37434 Vulnerability
Student:	Vojtěch Krejsa
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

1) Research the available information about the CVE-2022-37434 vulnerability.
2) Study the parts of the Zlib library and its file formats that are relevant to the vulnerability.
3) The vulnerability causes a buffer overflow and may cause a crash of an application. Explore the vulnerability and find out when and how this happens.
4) Analyze the available fixes for the vulnerability and discuss their reliability.
5) Explore the possibility of exploiting the vulnerability to execute an attacker's code. In case this is not possible, give reasons for it, otherwise propose conditions that would allow such an attack to succeed and prepare a proof-of-concept of it.
6) Discuss your results.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 6 December 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Analysis of the Zlib's CVE-2022-37434 Vulnerability

Vojtěch Krejsa

Department of Computer Systems
Supervisor: Ing. Josef Kokeš, Ph.D.

May 7, 2023

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Ing. Josef Kokeš, Ph.D., for his guidance, support, and advice. I would also like to thank Ladislav Marko, Bc. Viet Anh Tran, and Bc. Michal Dvořák for their assistance in proofreading the text of this thesis. Last but not least, a big thank you goes out to my family, my friends, and my significant other for their love and support throughout my studies.

Declaration

In Prague on May 7, 2023

Citation of this thesis

Krejsa, Vojtěch. *Analysis of the Zlib's CVE-2022-37434 Vulnerability.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstract

In early August 2022, a critical vulnerability identified as CVE-2022-37434 was discovered in the widely used Zlib compression library. The vulnerability is described as a heap buffer overflow. Some sources even argue that it could be exploited to execute arbitrary code. However, there is no available evidence confirming this claim. In this thesis, a detailed analysis of the vulnerability focusing on its exploitability to code execution is performed. The analysis is performed on the Ubuntu 22.04 LTS operating system with the glibc 2.35 memory manager and on Windows 10, version 22H2, with its default memory manager. The analysis results confirm that the vulnerability can indeed be exploited to code execution. In this thesis, it is described how it can be achieved. For demonstration purposes, virtual environments have been prepared.

Keywords CVE-2022-37434 vulnerability, Zlib, buffer overflow, gzip, code execution, exploit

Abstrakt

Na začátku srpna roku 2022 byla v široce používané kompresní knihovně Zlib objevena kritická zranitelnost s označením CVE-2022-37434. Zranitelnost je popisována jako přetečení bufferu na haldě. Některé zdroje dokonce uvádějí, že by mohla být zneužita až ke spuštění libovolného kódu. Neexistují však žádné dostupné důkazy, které by tuto skutečnost potvrzovaly. V této práci je provedena detailní analýza zranitelnosti s důrazem na možnosti jejího zneužití právě pro spuštění kódu. Analýza je provedena na operačním systému Ubuntu 22.04 LTS s paměťovým manažerem glibc 2.35 a na Windows 10 verze 22H2 s výchozím paměťovým manažerem. Výsledky analýzy potvrzují, že zranitelnost lze skutečně pro spuštění kódu zneužít. V této práci je popsáno, jak toho lze dosáhnout. Pro účely demonstrace byla připravena virtuální prostředí.

Klíčová slova zranitelnost CVE-2022-37434, Zlib, přetečení bufferu, gzip, spuštění kódu, exploit

Contents

List of Figures

Listings

Introduction

The amount of information stored, processed, and transmitted continuously increases. Along with that, the number of software products is growing as well, and so is the number of vulnerabilities. Software vulnerabilities allow attackers to infiltrate and take control over target systems, compromising the security, confidentiality, and integrity of stored information. Although many developers and companies try to minimize vulnerabilities in their software, creating perfectly secure software is nearly impossible. Cyber attacks are thus gradually becoming one of the key issues of the modern world.

Nowadays, vulnerabilities are being disclosed publicly, allowing developers to fix them faster and making the general public aware. In addition, it helps to convince the users why they should update to a newer software version.

On August 5, 2022, a critical vulnerability identified as CVE-2022-37434 was discovered in the widely used Zlib compression library. According to its description, it is a buffer overflow vulnerability. Some sources even argue that it could be exploitable to code execution. However, at the time of writing this thesis, there is no public exploit nor any evidence to confirm that claim. In the light of the extensive use of the Zlib library, we were intrigued by this fact, so we decided to explore the vulnerability more thoroughly.

Our goal is to study the Zlib library, identify its vulnerable parts, and determine under which circumstances a buffer overflow can be triggered. Our main goal is to describe in more detail when and how the overflow occurs and determine whether the CVE-2022-37434 vulnerability can indeed be exploited to code execution and, if so, to prepare a proof-of-concept of it.

The structure of the thesis is as follows:

In Chapter 1, we present the vulnerability disclosure system and how vulnerabilities are scored. Next, we introduce buffer overflow vulnerabilities and how they can be exploited. Finally, we present available information about the CVE-2022-37434 vulnerability.

In Chapter 2, we present the gzip file format and the parts of the Zlib library relevant to the CVE-2022-37434 vulnerability.

In Chapter 3, we describe how the vulnerability behaves, how the overflow can be triggered, and how partial conditions for code execution can be achieved.

In Chapter 4, we summarize the results of our analysis and propose conditions under which code execution can be achieved. We also present our CVE-2022-37434 vulnerability severity score.

Chapter 1

Vulnerabilities

The CVE-2022-37434 vulnerability is one of many publicly disclosed vulnerabilities. In this chapter, we aim to introduce vulnerabilities in general, how they are reported, when and how they are disclosed, how they are scored, and how the general public can learn about the new ones. Furthermore, the goal is to introduce buffer overflows, as the CVE-2022-37434 vulnerability belongs to the category of buffer overflow vulnerabilities.

First, we will introduce the Common Vulnerabilities and Exposures program, which is used for identifying and cataloging vulnerabilities. Next, we will present the widely used Common Vulnerability Scoring System v3.1, used to classify the severity of vulnerabilities. Then we will describe the buffer overflow vulnerabilities, including exploitation techniques and protections against them. Finally, we will present the available information about the CVE-2022-37434 vulnerability.

1.1 Common Vulnerabilities and Exposures

Common Vulnerabilities and Exposures (CVE) is a community-based effort that maintains a database of publicly disclosed cybersecurity vulnerabilities. Each vulnerability is described in a CVE Record and is uniquely identified with a CVE Identifier. CVE Records are stored in the CVE List, the catalog of all CVE Records identified by or reported to the CVE Program. The list is publicly available, and anyone can access information about all disclosed vulnerabilities in this list. [1], [2]

CVE was launched in 1999 and is operated by the MITRE Corporation, funded by the Cybersecurity and Infrastructure Security Agency (CISA), part of the U.S. Department of Homeland Security (DHS). The CVE program aims to increase the volume of created CVE Records and to produce them as quickly as possible after discovering a vulnerability. [1]

As shown in Fig. 1.1, the number of published CVE Records increases over time. This is due to several factors. The amount of software is growing, and at the same time, more people are looking for security flaws and reporting them. Some software vendors even encourage people to search for vulnerabilities in their software via so-called bug bounty programs[1].

CVE provides a convenient and reliable way for all third parties interested in a particular vulnerability to exchange security information about it. Using CVE Records with CVE Identifiers ensures that two or more parties can confidently refer to the same vulnerability when discussing or sharing information.

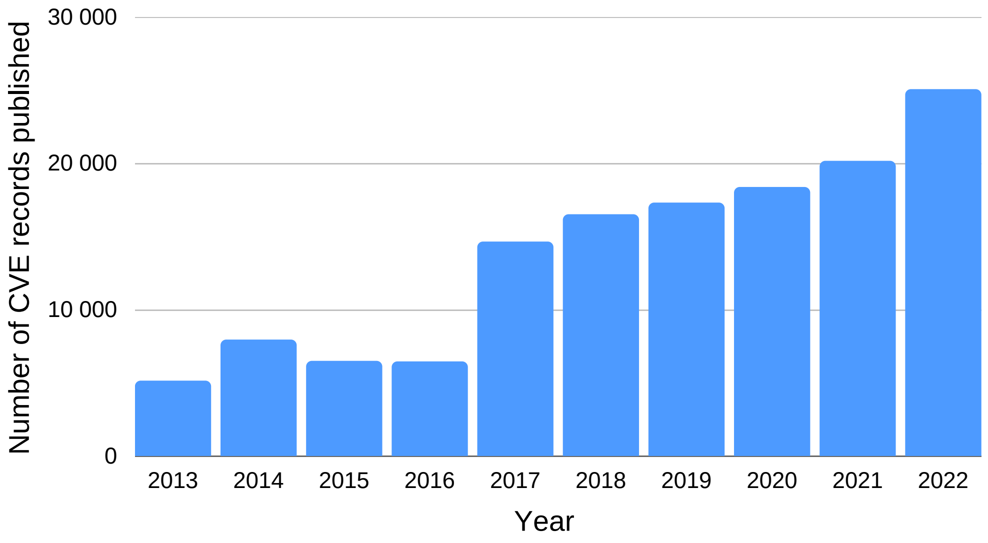


Figure 1.1: Number of CVE Records published in the past ten years [4].

1.1.1 Difference Between Vulnerabilities and Exposures

Vulnerability is a flaw in a software, hardware, firmware, or service component resulting from a weakness that can lead to the exploitation of a given system and causes a negative impact on confidentiality, integrity, or availability. An example of a vulnerability could be a buffer overflow vulnerability, which can lead to the execution of arbitrary code. [2], [5]

Exposure is an incident in which an unauthorized activity has taken advantage of a vulnerability. A vulnerability only implies potential exploitation, while exposure means the weakness has been exploited to perform an unauthorized action. [5]

[1]A *bug bounty program* is a public challenge issued by software vendors to reward individuals for reporting security vulnerabilities in their software. These programs allow developers to discover and fix vulnerabilities before the general public becomes aware, preventing them from being exploited. [3]

1.1.2 CVE Records

A CVE Record contains a brief description of a vulnerability associated with a CVE Identifier and references to more information about it, such as vendor advisories. It generally does not contain technical information and has no score for the vulnerability. However, additional information about the vulnerability, such as a CVSS score, information about a fix, and other essential details needed for the mitigation of the vulnerability, can be found in other databases, such as the U.S. National Vulnerability Database (NVD). [6]

As mentioned above, each CVE Record is assigned a number known as a CVE Identifier so that it is possible to refer to the vulnerability. Those identifiers are assigned by CVE Numbering Authorities (CNAs). The CVE Identifier takes the form CVE-[Year]-[Number], where the Year represents the year the CVE was published, and the Number is the number assigned by a CNA. [6], [7]

Thus, the CVE-2022-37434 vulnerability was published in 2022 and was assigned the serial number 37434.

Each CVE Record is associated with one of the following three states:

- **Reserved** – The initial state for a CVE Record. The CVE Record already has a CVE Identifier assigned by the responsible CNA but is not yet published.
- **Published** – In this state, the CVE Record is publicly disclosed in the CVE List. It must meet all the requirements to be published. It must contain an identifier, a brief description, and at least one public reference.
- **Rejected** – This state indicates that the CVE Record should no longer be used. A Rejected CVE Record remains in the CVE List so that users can know that it is invalid. [7]

1.1.3 Criteria for the Establishment of a CVE Record

For a CVE Record to be created, it must meet certain conditions. The complete, exhaustive list of requirements can be found in [8]. Only the most important aspects are mentioned below.

CVE Records must be assigned to independently fixable vulnerabilities. If one vulnerability can be fixed without fixing the other, a separate record should be created for each. Thus, one CVE Record should describe precisely one independently fixable vulnerability. At the same time, it must be true that the reported vulnerability negatively impacts the system's security. This can be acknowledged by the software vendor or the vulnerability reporter with a proof-of-concept of a security breach. However, the ultimate decision is always with the responsible CNA.

1.1.4 Reporting and Publishing CVEs

The entire CVE reporting process is divided into several steps:

1. First, a security researcher or an organization must discover a new vulnerability. Then the discoverer reports the vulnerability to a CNA and requests a CVE Identifier.

2. Once the identifier is reserved, the CVE Record is in the reserved state, which is its initial state. At this point, stakeholders can already use the CVE Identifier for early-stage vulnerability coordination and management. The CVE Record is not yet published at this stage.

3. Next, the CNA submits the details of the vulnerability. These details include affected products and versions, vulnerability type, impact, and at least one public reference.

4. Once the minimum required data elements are included in the CVE Record, the responsible CNA publishes it to the CVE List. As of this moment, the CVE Record is publicly disclosed. [7]

1.1.5 Risks Associated with CVEs

It should be noted that by publishing a CVE Record, attackers are also informed about the security flaw in the system. Whether it is better to disclose the vulnerability and give attackers an opportunity to attack everyone who has not yet patched it or to conceal its existence is a matter of debate.

It is generally accepted that the benefits of publishing outweigh the risks. Just because a vulnerability is not disclosed does not mean it is not being exploited. The description of the vulnerability in the CVE Record is intentionally brief so that it does not give out too much detail on how to exploit it. Disclosing and sharing information about vulnerabilities helps speed up their mitigation process. Generally, the details about a vulnerability are only published after a patch or fix is released. [6]

1.2 Scoring Vulnerabilities

Although any vulnerability in a system is undesirable, some of them are more severe than others. There may even be multiple vulnerabilities in a system simultaneously, so resolving them all at once may be impossible. In addition, the cost of fixing a vulnerability can even be greater than the potential damage to all users, so it is legitimate not to fix it at all or to fix it only partially to reduce its impact on the security of the given system. Therefore, objectively assessing each vulnerability's severity is essential. Vulnerabilities can then be ranked from the most severe and mitigated in that order.

To assess the severity of a vulnerability, it is advisable to utilize some of the available vulnerability scoring systems. Using a scoring system has one significant advantage – no essential security aspect is forgotten.

The vulnerability assessment can introduce certain subjectivity biases. Thus, different organizations may assign various severity scores to the same vulnerability.

There are several scoring systems, such as DREAD, Common Vulnerability Scoring System (CVSS), and Common Weakness Scoring System (CWSS). However, we will focus solely on CVSS, specifically its version 3.1, as it is currently the most widely used. NVD generally publishes CVSS v3.0 or CVSS v3.1 and CVSS v2.0 scores for each CVE Record.

1.2.1 Common Vulnerability Scoring System v3.1

The Common Vulnerability Scoring System is an open framework operated by the Forum of Incident Response and Security Teams (FIRST). It is used to evaluate the severity of software vulnerabilities. CVSS is currently in its third major version (v3.1), designed to address shortcomings in the previous versions.

The scoring system consists of three groups of metrics: Base, Temporal, and Environmental. The Base group represents vulnerability characteristics constant over time and across different environments. The Temporal group reflects vulnerability characteristics that change over time, and the Environmental group represents unique vulnerability characteristics within a given environment. The individual metrics have a small number of possible values to make the result less dependent on the subjectivity of the evaluator. [9]

Since NVD only provides the Base score for the vulnerabilities [10], we will not focus on Temporal and Environmental scores further. However, NVD provides a CVSS calculator, allowing users to add Temporal and Environmental metrics into their scoring, which better reflects the impact on their system.

Publicly available CVSS scores are in the vast majority Base scores only, which do not reflect all the factors in specific environments. Such a score can tell how dangerous the vulnerability is in general scope but cannot express the danger in the specific organization's context. This is why organizations should supplement the Base Score with Temporal and Environmental Scores specific to their use of the vulnerable product, making the severity score more accurate to the organization's environment. [9], [11]

The CVSS v3.1 Base score generates a numerical score ranging from 0 to 10, which can be modified by scoring the Temporal and Environmental metrics. The higher the value, the more severe the vulnerability. A CVSS score can also be represented as a vector string, a textual representation of the metric values used to derive the vulnerability score [9]. An example of such a vector string is shown in Fig 1.2.

1.2.2 CVSS Base Metrics

The CVSS Base metric group represents the characteristics of a vulnerability that are constant over time and across environments. It comprises three subsets of metrics: the Exploitability metrics, the Scope, and the Impact metrics.

Below is a brief overview and meanings of the metrics used in the Base Score calculation. A detailed description of each metric, including examples, can be found in the official documentation [9]. A graphical representation of the CVSS v3.1 calculation with a list of all possible values for each metric is shown in Fig. 1.2.

When scoring Base metrics, it should be assumed that an attacker has advanced knowledge of the weakness of the target system.

1.2.2.1 Exploitability Metrics

The Exploitability metrics reflect the characteristics of the vulnerable component. They consist of four metrics:

- **Attack Vector (AV)** – Reflects the context by which vulnerability exploitation is possible. This metric value will grow with the distance (logically and physically) an attacker can be in exploiting the vulnerable component. The possible values are Network, Adjacent, Local, and Physical.

- **Attack Complexity (AC)** – Reflects the complexity of successful vulnerability exploitation. It considers conditions needed to exploit the vulnerability but cannot be controlled by an attacker. As more of these conditions exist, the higher the complexity gets. The evaluation of this metric does not consider any user requirements – these are captured in the User Interaction metric. The possible values are Low and High.

- **Priviledges Required (PR)** – Describes the level of privileges an attacker must possess to exploit the vulnerability. The possible values are None, Low, and High.

- **User Interaction (UI)** – Captures whether user participation is required for a successful attack. It determines whether an attacker can exploit a vulnerability independent of other users. The possible values are None and Required. [9]

1.2.2.2 Scope

The Scope metric describes whether exploitation of the vulnerability can affect resources beyond the security scope managed by the vulnerable component. If such a possibility exists, the resulting score is higher. The possible values are Changed and Unchanged. [9]

1.2.2.3 Impact Metrics

Impact metrics describe the impact of a successful exploit on the component being attacked. The evaluation should reflect reasonable outcomes that we are confident an attacker can achieve. They consist of three metrics:

- **Confidentiality (C)** – Measures how much access an attacker gains to information resources by exploiting the vulnerability. Confidentiality refers to restricting access to information to authorized users only. The possible values are None, Low, and High.

- **Integrity (I)** – Measures how much an attacker can modify data on the compromised system. Integrity refers to the trustworthiness and veracity of information. The possible values are None, Low, and High.

- **Availability (A)** – Measures the impact on the impacted component's availability resulting from successful exploitation. The possible values are None, Low, and High. [9]

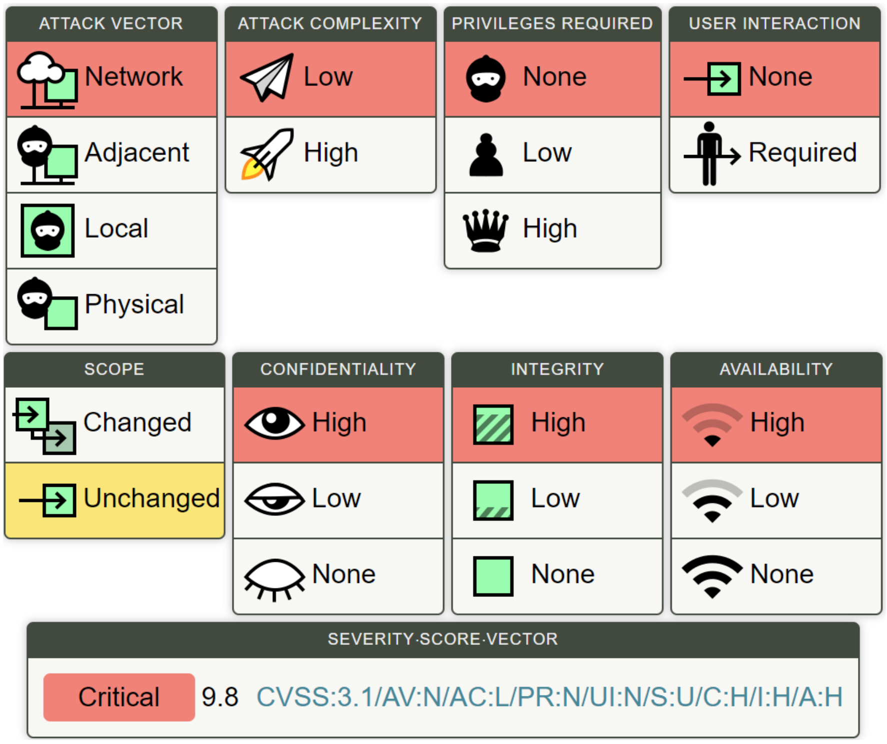
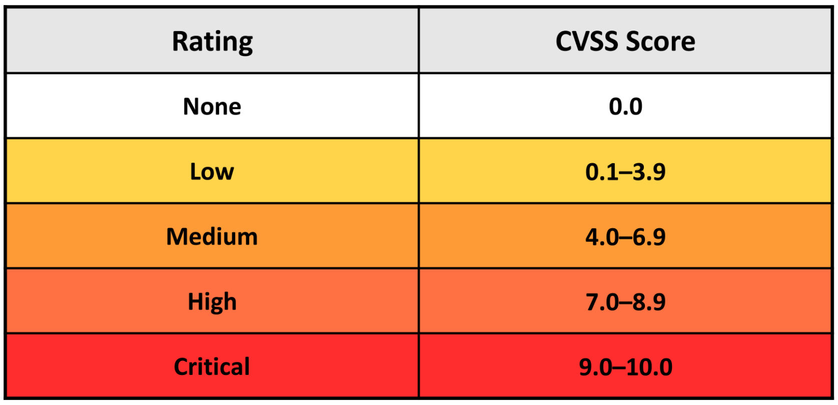


Figure 1.2: Visualization of the CVSS v3.1 Base score for the CVE-2022-37434 vulnerability provided by NIST [12]. The picture is taken from [13].

1.2.3 CVSS Qualitative Ratings

Converting the CVSS numeric value into a qualitative rating (text value) may be advantageous in some cases. This approach is especially useful when communicating with less technical stakeholders since referring to a vulnerability as critical rather than a vulnerability with a high CVSS score simplifies convincing the stakeholders of its severity.

FIRST maps CVSS scores to the qualitative ratings shown in Fig. 1.3. As can be seen, some security vulnerabilities may be assigned a CVSS score of zero, raising questions as to why such a vulnerability exists in the first place. Generally, these are reported bad habits that violate secure coding and are not vulnerable by themselves but may be vulnerable in combination with another bug.



Rating	CVSS Score
None	0.0
Low	0.1–3.9
Medium	4.0–6.9
High	7.0–8.9
Critical	9.0–10.0

Figure 1.3: Qualitative Severity Rating Scale [9].

1.3 Buffer Overflows

The vulnerability in the Zlib library described in CVE-2022-37434 is a buffer overflow vulnerability. To comprehend the buffer overflow vulnerabilities, it is first necessary to define the term buffer.

Buffer is a limited, contiguous allocated set of memory. Buffers are often used to hold data while moving it from one program section to another. [14]

A buffer overflow occurs when the amount of data written to a buffer exceeds the capacity of that buffer[2]. A buffer is located in memory, and an overflow causes a write beyond the buffer, overwriting whatever is stored there. [15]

[2]It should be emphasized that buffer overflows are solely the programmer's fault. Programs behave deterministically and do precisely what the programmer wrote. However, programmers are human and make mistakes, which may then lead to buffer overflows.

The buffer overflow exploitation is based on the principle that the vulnerability permits overwriting memory in a manner unintended by the program's author. By overwriting certain structures in memory, it is possible to change the flow of a program and, under certain circumstances, even gain control over it to execute arbitrary code.

Buffer overflow is one of the best-known forms of software vulnerability. Despite being well-known among software developers, it is still quite common. However, buffer overflows are not easy to discover, and even when one is found, it is generally complicated to exploit, mainly due to existing protections against them. Therefore, nowadays, buffer overflow-type vulnerabilities generally lead to a program crash. [16]

Buffer overflows are typically seen in C and C++ languages. This is because these languages have no inherent bounds-checking for buffers, shifting the responsibility to the programmer. However, programmers do make mistakes and give these errors a chance to occur. [14]

If the responsibility were passed to the compiler, the resulting programs would be slower because of the integrity checks [15]. This is one of many reasons why these languages are still widely used, especially in applications optimized for time and space. They give the programmer a significant level of control but, at the same time, a considerable responsibility.

Buffer overflow vulnerabilities usually occur in a code that relies on external data to control its behavior, for example, if the program processes user input.

1.3.1 Buffer Overflow Exploitation

There are many techniques to exploit buffer overflows. They vary by the memory segment in which the overflow occurs, the operating system, the programming language, and the architecture. However, all the techniques have one thing in common – they all aim to overwrite some interesting structures in the overflowed memory, attempting to take control over the program flow.

Buffer overflows may be exploited to cause an application to crash[3]. Another possible exploitation is to overwrite the data behind the overflowed buffer. If there were a bank account number directly behind the buffer, it would undoubtedly be enough for an attacker if they could overwrite it with their value. But perhaps the most interesting thing that can be achieved by exploiting a buffer overflow is gaining control over the execution flow of the program and forcing it to execute an injected code.

[3]These types of attacks are referred to as denial-of-service attacks. Even a mere application crash is a severe vulnerability for institutions such as banks or hospitals.

1.3.1.1 Stack Buffer Overflows

A stack buffer overflow is an overflow that occurs in a memory segment referred to as a stack.

Stack is a LIFO (Last In First Out) data structure that stores data. The last element is processed first, and the first is processed last. Each program contains a stack segment that fulfills the purposes of such a stack. The stack is essential for the program when calling functions. When a function has completed executing its instructions, it returns control to the original function caller. This concept is most efficiently implemented with the stack. [14], [15]

Stack buffer overflow attacks do not vary much across different systems. This makes protections against them more effective. However, it also makes them easier to exploit. The stack buffer overflow exploitation techniques were described in detail in the paper *Smashing The Stack For Fun And Profit*[4].

Most stack buffer overflow attacks target overwriting the return address stored on the stack. Return addresses are an interesting target because when a program finishes executing a function, it continues at the address specified in the return address. Thus, by overwriting it, an attacker can control where the program continues after the function ends, for example, at the address where they inject their code via user input. [14]

Exploitable targets on the stack include, but are not limited to:

- Return addresses
- Exception handlers
- Function pointers
- Values of local variables [14]

Since the vulnerability analyzed in this thesis is not a stack-based buffer overflow, we will not discuss stack overflows further.

1.3.1.2 Heap Buffer Overflows

A heap buffer overflow is an overflow that occurs in a memory segment referred to as a heap.

Heap is a writable memory segment that a programmer can directly control. The heap is not of fixed size – it can grow larger or smaller as needed. It is used when the programmer does not know, during compilation, how much memory will be needed, or for allocating larger chunks of memory. [14], [15]

[4]`http://phrack.org/issues/49/14.html`

Heap buffer overflows differ from the stack ones not only in where they occur but also in how they are being exploited. This is because the heap has a much more complicated structure, and its implementation depends heavily on the memory manager used. Thus, the methods of exploiting buffer overflows on the heap vary across different memory managers and their versions. Therefore, there is no universal way to describe heap exploitation.

Heap overflow attacks tend to be more difficult to perform because of the complexity of the heap structure and the needed knowledge about the memory manager used. However, nowadays, heap overflows are being exploited more than the stack ones. This is mainly because the various heap implementations make developing universal protections against heap overflows difficult.

One possible heap overflow exploitation technique is based on overwriting the internal heap structures located between allocated chunks of memory. By doing so, the memory manager can be forced to behave in a non-desirable manner. These techniques are theoretically very interesting but difficult to perform and heavily dependent on the memory manager. Several articles have been published in the past on how to exploit heap buffer overflows by overwriting these internal structures – *Vudo malloc tricks*[5], *Once upon a free()*[6], *The Malloc Maleficarum*[7], and *Malloc Des-Maleficarum*[8].

However, we will not focus on exploiting the heap by overwriting its internal structures. Instead, we will focus on how a heap overflow can be exploited by overwriting a function pointer located in the heap memory, which is a far more typical way of heap exploitation.

Unlike standard pointers, a function pointer points to code, not data. Generally, a function pointer holds an address of a function. This technique is based on overwriting the value of the function pointer with the address where the injected code is located. When the function initially stored in this function pointer is called, the injected code is executed[9].

1.3.1.3 Overwriting a Function Pointer

A simplified example of such function pointer override is shown in Fig 1.4. This example considers a 64-bit application, so the function pointer takes eight bytes. It is also assumed that when writing to the buffer, the size of the data being written is not checked. Thus, the 16-byte buffer is vulnerable to an overflow, and copying 24 bytes into it leads to overwriting the function pointer behind it. Considering the little-endianness of the CPU and the

[5]`http://phrack.org/issues/57/8.html`

[6]`http://phrack.org/issues/57/9.html`

[7]`https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt`

[8]`http://phrack.org/issues/66/10.html`

[9]In fact, the injected code would have to be additionally located in an executable memory segment.

ASCII encoding, the original value of the function pointer `0x5555555555555189` is overwritten to `0x4242424242424242` caused by that overflow[10].

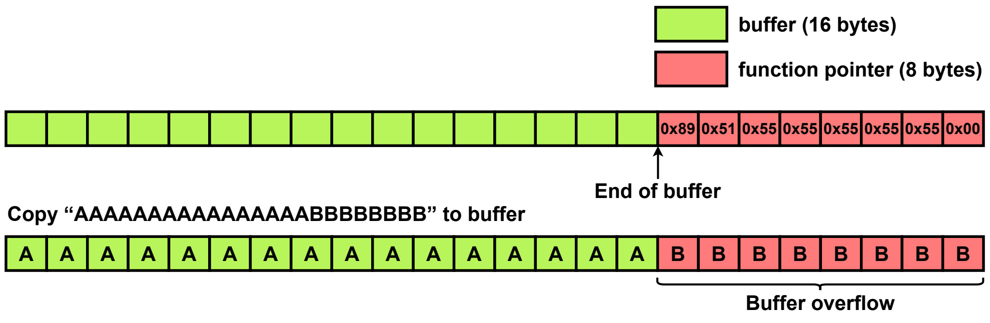

buffer (16 bytes)
function pointer (8 bytes)
0x89
0x51
0x55
0x55
0x55
0x55
0x55
0x00
End of buffer
Copy "AAAAAAAAAAAAAAAABBBBBBBB" to buffer
A A A A A A A A A A A A A A A A B B B B B B B B
Buffer overflow

Figure 1.4: Example of a buffer overflow leading to a function pointer override.

If an attacker could control what data is copied to the buffer, they could control what value the function pointer is overwritten with. Such controllable values are, for example, user inputs. As shown in the example above, the buffer did not store the B character but its binary representation. Thus, to override a function pointer with a specific value, an attacker only would need to find characters whose encoding corresponds to the desired binary values.

However, some values may be problematic to enter. One such value is `0x00` since it also represents the end of a string in C/C++. The most straightforward approach to deal with such values is to modify the injected code not to contain them. However, this may not be feasible in certain situations, such as when overwriting memory with a specific value is needed (e.g., when overwriting an address). Nevertheless, user input does not always have to be entered as a string. It can also be entered as a binary file into which these problematic characters can easily be inserted.

We emphasize that the function pointer does not have to be located immediately after the buffer to be overridden this way. If the memory behind the buffer is writable (i.e., allocated with write permissions), and the overflow does not exceed that memory, the application will not crash. Thus, if a function pointer is located in a contiguous writable memory region behind the buffer, and the overflow allows writing to this region, the pointer can be overwritten.

1.3.2 Buffer Overflow Protections

Due to the severity of buffer overflow vulnerabilities, many protections have been developed to protect memory from these attacks. These protections do not prevent the overflow itself but try to prevent arbitrary code execution when an overflow occurs. It should be noted that these protections do not prevent

[10]The overwritten value is `0x4242424242424242` since the B character has a hexadecimal value of `0x42` in the ASCII encoding.

the execution of such an attack but only make it more difficult to perform. Buffer overflow protections are complementary, and overcoming multiple ones simultaneously is significantly more difficult.

While it may seem virtually impossible to cause arbitrary code execution with all protections enabled and that they definitively solve buffer overflows, it is still possible, and buffer overflows are still a severe problem even nowadays. Moreover, an attacker often does not need to execute code to exploit the vulnerability. As we already mentioned, if there was a variable behind the buffer specifying a bank account number, the attacker would be perfectly happy to modify that value and would not need to run any code.

The most commonly used buffer overflow protections are described below.

1.3.2.1 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) is a security feature controlled by the operating system that places important parts of a process, such as libraries, executables, the stack, and the heap, at random addresses each time the application is run. [14]

Even though an attacker can overwrite a return address or a function pointer, they do not know what value to overwrite it with since the address space is randomized. The protection of ASLR is based on the principle that it reduces the attacker's probability of correctly guessing the correct address. However, the attacker may still attempt to guess the correct value. For example, in small address spaces, ASLR can be overcome by brute-force attacks. [14]

ASLR is very effective in combination with the NX-bit and has a low impact on program performance. Currently, there is no legitimate reason to disable this protection.

1.3.2.2 NX-bit/XD-bit

The NX-bit (Non-eXecutable bit) allows processors to mark memory pages as non-executable. AMD introduced this feature in 2003, and Intel refers to this functionality as XD-bit (eXecute Disable). The idea is simple. Only pages that contain code will be executable, and the others will be marked as non-executable since there is no reason to run code from them. The processor will refuse to execute instructions in a memory marked as non-executable. [14], [17]

With the NX-bit protection enabled, more than redirecting the program flow to the injected code is required. In addition, the injected code must be placed in executable memory, which significantly complicates the attack. On Microsoft systems, this protection is referred to as Data Execution Prevention (DEP).

Unfortunately, there exists a technique that overcomes this protection quite effectively. This technique is referred to as Return Oriented Programming (ROP). This is why it is advisable to use multiple protections simultaneously. As with ASLR, there is no legitimate reason to disable this protection.

1.3.2.3 Stack Canaries

A stack canary is designed to protect the stack. The name canary comes from miners. They used to take a canary with them when they went down the mine. The canary indicated that there were poisonous gases in the mine as it was more sensitive and died first if there were some, giving the miners time to escape. [14]

A stack canary works on a very similar principle. It is a randomly generated value placed between the return address and local variables on the stack. This value is checked before returning from a function. If the value is overwritten, there is a high probability that the return address is overwritten as well. In such circumstances, it is safer to terminate the program directly. [14]

This technique cannot be applied to a heap, as it cannot be determined where such a canary should be placed and when it should be checked.

1.3.2.4 Guard Pages

A guard page is an unmapped memory page with no access rights placed between allocated memory blocks used to detect buffer overflows on the heap. It is not allowed to read, write or execute instructions from a guard page. An exception is raised if this page is accessed, causing a segmentation fault. [18], [19]

One possible approach is to place guard pages immediately after a buffer, thus detecting an overflow of even a single byte. This can be done by placing each buffer at the end of a memory page and placing a guard page immediately after that page.

While these guard pages increase memory requirements, they can efficiently detect overflows. It is, therefore, up to the developer to consider whether and to what extent to use them. They can be helpful, at least for testing purposes.

1.3.2.5 Encoding and Decoding Function Pointers

By overwriting a function pointer, an attacker can change the program flow. The protective technique works on a similar principle as ASLR. It does not prevent the pointer from being overwritten, but it reduces the likelihood that an attacker will be able to guess the correct value to overwrite it with. This is achieved by encrypting the function pointer. Instead of storing a function pointer, the program keeps its encrypted form. When the function pointer needs to be used, it gets decrypted. Thus, an attacker would need to break the encryption to redirect the pointer to the injected code. [18]

The Windows API provides the `EncodePointer()` and `DecodePointer()` functions that implement this functionality.

1.4 CVE-2022-37434

In this thesis, we focused on the vulnerability described in CVE-2022-37434[11] published on August 8, 2022, which is a vulnerability in the widely used Zlib compression library.

According to the CVE Record, the vulnerability is described as a heap buffer overflow via a large gzip header extra field in the `inflate()` function. In addition, the `inflateGetHeader()` function must be called for the overflow to occur. If this function is not called, the header information, including the extra field that causes the overflow, is discarded. [20]

NIST assigned CVE-2022-37434 a CVSS v3.1 score of 9.8 [10], which puts it into the category of critical vulnerabilities. Some other companies rate this vulnerability less severely. For example, SUSE assigned CVE-2022-37434 a CVSS v3.1 score of 8.1 [21], and Red Hat a 7.0 [22]. However, these ratings are often lower mainly because no exploit taking advantage of the vulnerability has been published yet.

It is not explicitly stated in the CVE Record that the vulnerability can lead to code execution. However, given the high CVSS v3.1 score, it can be assumed that it can. Some third parties using the Zlib library, such as Apple [23] or Debian [24], report in their security updates that the CVE-2022-37434 vulnerability could indeed lead to code execution.

One of the references in the CVE Record is a GitHub repository created by Evgeny Legerov, who discovered the vulnerability. Unfortunately, this repository has been deleted and no longer exists. The reasons for the deletion are not known. The repository did not include an exploit proving code execution but served as a proof of concept that a heap buffer overflow can be triggered.

A patch remediating the CVE-2022-37434 vulnerability has already been published. Its reliability will be discussed in the final chapter. The fix consists of two commits. The first commit[12] introduced a new bug into the code – a `NULL` pointer dereference. The second commit[13] only fixes this bug.

According to the description of the first commit, an overflow can occur if the extra field of the gzip header is larger than the space provided by the user when calling `inflateGetHeader()`. If the extra field data are delivered in multiple calls of `inflate()`, there could be a buffer overflow of the provided space.

The vulnerability was found in the library in version 1.2.12 but was introduced in the code in version 1.2.2.1, meaning the library had been vulnerable for almost terrifying twelve years. On October 13, 2022, Zlib version 1.2.13 with the fix remediating the vulnerability was published.

To summarize, we have found that the CVE-2022-37434 vulnerability is a buffer overflow that can occur during gzip decompression using the Zlib

[11] `https://www.cve.org/CVERecord?id=CVE-2022-37434`
[12] `https://github.com/madler/zlib/commit/eff308a`
[13] `https://github.com/madler/zlib/commit/1eb7682`

function `inflate()`. Currently, there is no known exploit demonstrating the exploitation of this vulnerability, and apart from the information mentioned above, no further information is available.

In the following chapters, we will describe the vulnerability in more detail and present how it can be exploited to code execution.

Chapter 2

The Zlib Library

The CVE-2022-37434 vulnerability was discovered in the Zlib library[14]. Zlib is a free, open-source software library for lossless data compression and decompression. The library's authors are Jean-loup Gailly and Mark Adler, who are also the authors of gzip[15]. Zlib is written in the C programming language and can be used on virtually any hardware and operating system, which makes it very popular and widely used across systems. In the past, a list[16] of all applications that used Zlib has been maintained on the official Zlib website. However, this list contained so many applications that it was impossible to continue. [25], [26]

Nowadays, most operating systems and software that uses compression use Zlib, either directly or indirectly. Well-known software using the Zlib library includes the Linux kernel, the Apache HTTP Server, and the OpenSSL cryptographic library.

In this thesis, we have examined the Zlib library in more detail to analyze the CVE-2022-37434 vulnerability's security impacts and exploitability. In this chapter, we will introduce selected parts of the library that are relevant to the vulnerability. First, we will describe the gzip file format. Being familiar with this file format helps in understanding how the library handles it and how a malicious file should be created to trigger an overflow. Next, we will present the important structures and functions we used to demonstrate how the vulnerability can be exploited to execute arbitrary code. Finally, we will describe how Zlib processes the gzip header since the vulnerability is found in the code segment that processes it.

We performed the analysis on the library version 1.2.12[17] and all information provided in this chapter applies to this version of the library. While most results are likely to apply to older versions, they may differ slightly.

[14]`https://www.zlib.net/`

[15]`http://www.gzip.org/`

[16]`https://zlib.net/apps.html`

[17]`https://github.com/madler/zlib/tree/v1.2.12`

2.1 Zlib's Data Formats

The Zlib library uses the `DEFLATE` algorithm for compression and decompression and supports three data formats: the deflate format, the zlib format, and the gzip format. The deflate format is simply a deflate stream with no wrapper. The zlib and gzip formats are wrappers around the deflate stream, providing additional information about the compressed data. [27]

The main difference between the zlib and gzip wrappers is that the zlib wrapper is more compact as the gzip wrapper holds extra information such as a filename, comment, or extra field. These wrappers also differ in the integrity check algorithm. The gzip format uses CRC-32, while the zlib format uses Adler-32, which runs faster. [28]

By default, the library uses the zlib format [27]. This is important information since, according to the CVE-2022-37434, a buffer overflow can be triggered when decompressing a gzip file which is not the Zlib's default data format. This does not mean the library is not vulnerable, but it reduces the number of potentially vulnerable applications.

All three data formats Zlib uses are described by RFCs (Request for Comments). The zlib format is documented in RFC 1950[18], the deflate format is documented in RFC 1951[19], and the gzip format is documented in RFC 1952[20]. Since the vulnerability is related to the gzip format, we will only focus on the gzip format.

2.1.1 Gzip File Format

The following description of the gzip file format is based on the RFC 1952.

The gzip file format is a lossless compressed data format independent of CPU type, operating system, file system, and character set. It was designed for single-file compression and is compatible with the format created by the gzip utility. This format uses the `DEFLATE` algorithm for compression but can easily be extended to other algorithms. [29]

The gzip file consists of a series of so-called members. The members appear one after another in the file, with no additional information before, between, or after them. The format of such a member is shown in Fig. 2.1. It consists of several fields. Some fields are fixed size, some are variable size, some are mandatory, and some are optional. The presence of optional fields is specified via the `FLG` field, and Fig. 2.1 shows which fields are optional. The meaning and size of each field are as follows:

[18]https://www.rfc-editor.org/rfc/rfc1950
[19]https://www.rfc-editor.org/rfc/rfc1951
[20]https://www.rfc-editor.org/rfc/rfc1952

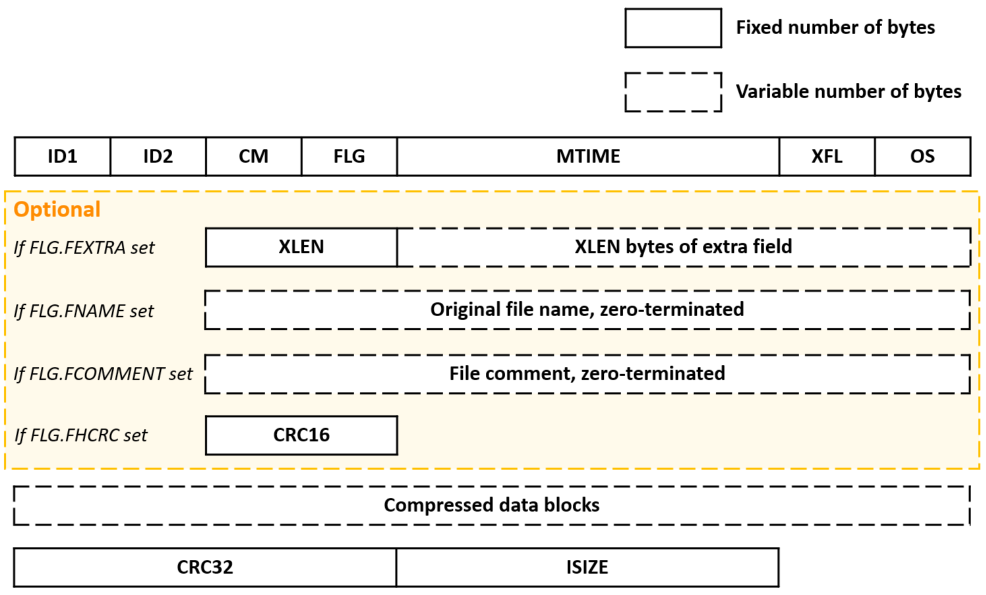


Figure 2.1: Visualization of the gzip file format [29].

- **`ID1, ID2`** fields are the size of one byte. Their value is fixed – `ID1` has the value `0x1f`, and `ID2` has the value `0x8b`. They are used to identify the gzip file.
- **`CM (Compression Method)`** field is the size of one byte. It identifies the compression method used. The deflate compression method has the value `0x08`.
- **`FLG (Flags)`** field is the size of one byte. The individual bits are used as flags indicating if optional fields are present. The bits are arranged so that the most significant bit (bit 7) is on the left.
  - bit 0: `FTEXT`

    If set, the file is probably ASCII text.
  - bit 1: `FHCRC`

    If set, an optional field `CRC16` for the gzip header is present.
  - bit 2: `FEXTRA`

    If set, optional extra fields are present.
  - bit 3: `FNAME`

    If set, an original file name terminated with a zero byte is present.
  - bit 4: `FCOMMENT`

    If set, a comment terminated with a zero byte is present.

  - bit 5–7: reserved

    These bits are reserved and must be set to zero.

- **`MTIME (Modification Time)`** field is the size of four bytes. It specifies the most recent modification time of the original file. The time is in Unix format.

- **`XFL (Extra Flags)`** field is the size of one byte. These extra flags are available for use by specific compression methods.

- **`OS (Operating System)`** field is the size of one byte. It identifies the operating system on which the compression was performed. It can help determine the end-of-line conventions for text files.

- **`XLEN (Extra Length)`** field is of the size of two bytes. If the `FEXTRA` flag is set, it specifies the optional extra field length.

- **`CRC32`** field is the size of four bytes. It contains the uncompressed data's CRC32 value. It is used to check if the data was not corrupted during decompression.

- **`ISIZE (Input Size)`** field is the size of four bytes. It contains the size of the original input modulo 2^{32}. [29]

The presence of the optional extra field is a necessary condition for the CVE-2022-37434 vulnerability to be exploitable. For this reason, from now on, we will only consider gzip files containing an extra field. It is used to store extra data in the header of the compressed file, and some applications use it to extend the gzip format.

The optional extra field consists of a series of subfields. The form of each subfield is shown in Fig. 2.2, and the size and meaning of each field are as follows:

- **`SI1, SI2`** fields are the size of one byte. They provide a subfield identifier, typically two ASCII letters.

- **`LEN`** field is the size of two bytes. It specifies the length of the subfield data, excluding the four initial bytes. [29]

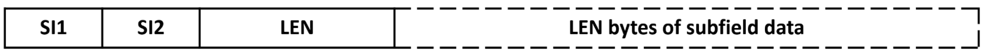


Figure 2.2: Visualization of the extra field subfields [29].

2.2 Zlib's Structures

The Zlib library uses several structures when compressing and decompressing a gzip file. We will present two of them – specifically the `z_stream` and `gz_header` structures. Both are used when using `inflate()` for gzip decompression with a previous call to `inflateGetHeader()`.

2.2.1 z_stream

Zlib uses the `z_stream` structure to store the context of ongoing compression and decompression, and it is therefore passed to most Zlib functions as an argument. Before any compression or decompression can begin, `z_stream` should be initialized with one of the initialization functions provided by the Zlib library. Before calling an initialization function, the application must initialize `zalloc`, `zfree`, and `opaque`. Those variables are function pointers, and an application can pass through them pointers to its functions for custom memory management. If these values are set to `Z_NULL` (Zlib's zero constant), the standard library functions `malloc()` and `free()` are used. [27]

Since the CVE-2022-37434 vulnerability does not depend on custom memory management, we will always set these variables to `Z_NULL`.

The input to be processed is passed to the compression and decompression methods via `next_in` and `avail_in`, and the output is returned via `next_out` and `avail_out`. The application must update `next_in` and `avail_in` when `avail_in` has dropped to zero and must update `next_out` and `avail_out` when `avail_out` has dropped to zero. The library sets all the other fields, and the application must not update them. [27]

The meaning of the individual `z_stream` variables is described in Listing 2.1.

Listing 2.1 Structure `z_stream` [27].

```c
typedef struct z_stream_s {
    z_const Bytef *next_in; // next input byte
    uInt      avail_in;  // number of bytes available at next_in
    uLong     total_in;  // total number of bytes read so far
    Bytef     *next_out; // next output byte will go there
    uInt      avail_out; // remaining free space at next_out
    uLong     total_out; // total number of bytes output so far
    z_const char *msg;   // last error message
    struct internal_state FAR *state; // not visible by apps
    alloc_func zalloc;   // used to allocate internal state
    free_func  zfree;    // use to free internal state
    voidpf     opaque;   // argument passed to zalloc and zfree
    int        data_type;  // best guess about the data type
    uLon269g   adler;      // Adler-32 or CRC-32
    uLong      reserved;   // reserved for future use
} z_stream;
```

2.2.2 gz_header

Zlib uses the gz_header structure to pass gzip header information of a gzip stream to and from Zlib routines. Using the gz_header, an application can precisely define the format of a gzip header of a compressed file. Each optional field's presence and content can be defined by passing a pointer to this field in the appropriate variable. If the header should not contain an optional field, the application must set the corresponding variable to Z_NULL. [27]

Zlib provides the deflateSetHeader() function to set a gzip header of a compressed file and the inflateGetHeader() function to retrieve the gzip header from a compressed file.

The individual gz_header variables match the fields of the gzip format described above, and according to their name, it is possible to recognize which variable corresponds to which field. In addition to the gzip header, the gz_header structure contains extra_max, name_max and comm_max variables. They are used only when retrieving the header. Some optional fields in the gzip header are of variable lengths, and to retrieve them, the application must provide memory into which these fields should be loaded. The available memory for each field is passed as a pointer to this memory via the extra, name, and comment variables. If these variables are not Z_NULL, then extra_max, name_max, comm_max contain the maximum size of memory in bytes provided for each field [27].

The meaning of the individual gz_header variables is described in Listing 2.2.

Listing 2.2 Structure gz_header [27].

```
typedef struct gz_header_s {
    int      text;      // true if compressed data believed to be
                        // ASCII text
    uLong    time;      // modification time
    int      xflags;    // extra flags
    int      os;        // operating system
    Bytef    *extra;    // pointer to extra field or Z_NULL
    uInt     extra_len; // extra field length
    uInt     extra_max; // space at extra (if not Z_NULL)
    Bytef    *name;     // pointer to file name or Z_NULL
    uInt     name_max;  // space at name (if not Z_NULL)
    Bytef    *comment;  // pointer to comment or Z_NULL
    uInt     comm_max;  // space at comment (if not Z_NULL)
    int      hcrc;      // true if the CRC is present
    int      done;      // true when done reading gzip header
} gz_header;
```

2.3 Zlib's Functions

The Zlib library provides many functions for compression and decompression, but we will only focus on a subset of these functions related to the analyzed vulnerability.

The library uses the `DEFLATE` compression algorithm, a combination of the `LZ77` algorithm and Huffman encoding. However, the compression algorithm itself is irrelevant to the vulnerability since the buffer overflow occurs during the gzip header decoding, i.e. before an actual data decompression is performed. Because of this, we will not describe the code sections implementing this algorithm further.

Zlib refers to its compression method as deflate and decompression method as inflate. The prefix "inflate" or "deflate" makes it possible to distinguish whether a function relates to compression or decompression. Fig. 2.3 shows the deflate and inflate process from a high-level perspective. Although the CVE-2022-37434 vulnerability only affects the decompression process, we will also describe several functions that are used for compression as we will use them to create a malicious file.

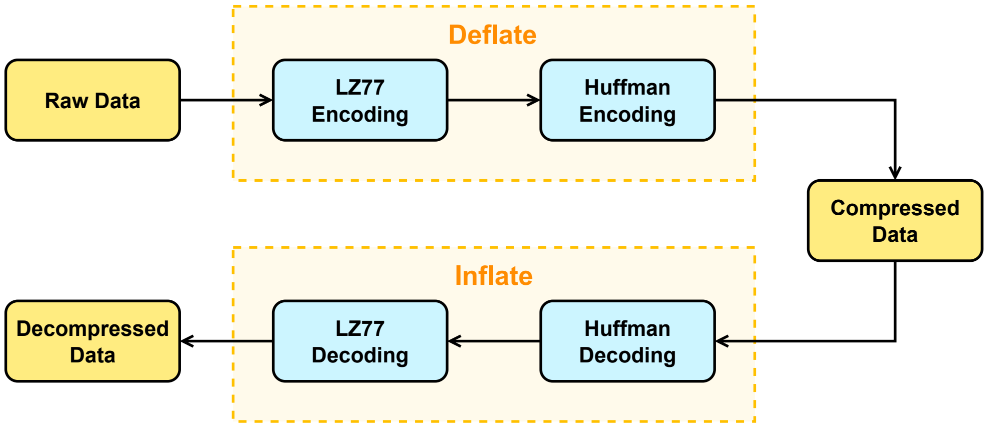


Figure 2.3: Inflate and Deflate process from high level [25].

2.3.1 deflateInit2(), inflateInit2()

As mentioned above, `z_stream` should be initialized before the actual compression or decompression can begin. For this purpose, the Zlib library provides initialization functions, which include `deflateInit2()` and `inflateInit2()`. These functions must be called before the call to `inflate()` or `deflate()`, depending on the operation to be performed [27].

The `deflateInit2()` and `inflateInit2()` functions are extensions of `deflateInit()` and `inflateInit()`, allowing setting more compression or decompression options. What is important for the CVE-2022-37434 vulnerability is that these extended functions, unlike the basic ones, allow setting the gzip encoding and decoding.

As we mentioned, an overflow can only occur in the library when decompressing a gzip file, meaning that `inflateInit2()` must be used for the CVE-2022-37434 vulnerability to be exploitable. The `deflateInit2()` may not be used, but we use it to create a malicious file. However, that malicious file could be created by other means as well.

Since the CVE-2022-37434 vulnerability does not depend on compression options, they can be set to their default values[21]. The gzip encoding or decoding is achieved by adding 16 to the `windowsBit` argument for both functions. The default value of `windowsBit` is 15, so with the addition of 16, we will set it to 31.

2.3.2 deflate(), inflate()

The Zlib library performs the actual compression using the `deflate()` function, and the decompression is done using the `inflate()` function. Both of these functions take as arguments a `z_stream` structure, through which the input is passed, and the output is returned, and a `flush` argument, which is used to specify how the data should be flushed to the output buffer. Both allow the input to be processed by parts and can be called repeatedly to process the whole input[22]. [27]

The `deflate()` function tries to compress as much of the provided data as possible and stops when the input buffer is empty or the output buffer is full. The function processes more input, generates more output, or does both simultaneously. The caller should ensure that at least one of these actions is possible before calling `deflate()`. [27]

Of all the available options for the `flush` argument of `deflate()`, for simplicity, we will only use the `Z_NO_FLUSH` and `Z_FINISH` options.

The `Z_NO_FLUSH` option lets `deflate()` decide how much data to accumulate before producing output to maximize compression, and `Z_FINISH` is used in the last call to `deflate()`. Therefore, `deflate()` may not produce output on every call. [27]

The `inflate()` function is crucial for the CVE-2022-37434 vulnerability as part of the `inflate()` function is the gzip header processing, which is the vulnerable part of the Zlib library.

`inflate()` tries to decompress as much of the provided data as possible and stops when the input buffer is empty or the output buffer is full. Just like `deflate()`, it processes more input, generates more output, or does both

[21]However, it may be useful to use `Z_NO_COMPRESSION` for `deflateInit2()` to store data in a gzip wrapper without compression when creating a malicious file that aims to inject code into an application. We will address this in the next chapter.

[22]It is even necessary for `inflate()` to be called repeatedly to trigger an overflow in the library.

simultaneously. The caller should ensure that at least one of these actions is possible before calling `inflate()`. [27]

We will always set the `flush` argument of `inflate()` to the `Z_NO_FLUSH` option since `inflate()` can detect the end of the stream itself, so there is no need to use a different option for the vulnerability demonstration purposes. Just like `deflate()`, `inflate()` may not produce output on every call [27].

2.3.3 deflateEnd(), inflateEnd()

The Zlib library may allocate some data structures for a `z_stream`, for example, when initializing it. The `deflateEnd()` and `inflateEnd()` functions are used to free all data structures that the library has allocated for a given `z_stream`. These functions discard any unprocessed input and do not flush any pending output. [27]

2.3.4 deflateSetHeader(), inflateGetHeader()

With the use of the `deflateSetHeader()` function, an application can provide a custom gzip header for a file to be compressed. This function may be called after `deflateInit2()` and before the first call to `deflate()`. It is passed a `z_stream` to which the given gzip header belongs and a `gz_header` through which the header is passed. The caller should ensure that if `name` and `comment` are not `Z_NULL`, the corresponding strings are terminated with a zero byte and that if `extra` is not `Z_NULL`, `extra_len` bytes are available there. [27]

Calling `deflateSetHeader()` is optional. If the function is not called, Zlib will use its default gzip header[23] [27]. We will use this function when creating a malicious file.

The `inflateGetHeader()` function is used to retrieve a gzip header of a compressed file into the provided `gz_header` structure. It may be called after `inflateInit2()` and before the first call to `inflate()`. Before calling `inflateGetHeader()`, the caller must assign to the `name`, `comment`, and `extra` variables pointers to the memory into which the optional gzip header fields are to be loaded. If these variables are not `Z_NULL`, the caller should also set variables `name_max`, `comm_max` and `extra_max`, specifying the maximum size of the provided memory space. [27]

According to the documentation, only the truncated data should be retrieved if the optional fields exceed the provided space [27]. This is the crucial information since the CVE-2022-37434 vulnerability violates this expected behavior, which can lead to a buffer overflow.

Without calling `inflateGetHeader()`, the entire gzip header is discarded [27], and the vulnerable section of `inflate()` is not reached. This is why applications that do not call `inflateGetHeader()` are unaffected by the vulnerability.

[23]If the gzip wrapper is used for compression.

2.4 Gzip Header Processing

The header processing is part of the `inflate()` function and is the vulnerable part of the library. `inflate()` uses a state machine to process as much input data and generate as much output data as possible in a single call [30]. Specifically, the state that retrieves the extra field from a gzip header is vulnerable. The rough structure of this state machine is shown in Listing 2.3.

Listing 2.3 `inflate()` state machine pseudocode [30].

```
for (;;)
    switch (state) {
        ...
        case STATEn:
            if (not enough input data or output space to make
                progress)
                return;
            ... make progress ...
            state = STATEm;
            break;
        ...
}
```

When `inflate()` is called again, the same state is attempted again [30], which makes it possible to split the compression into multiple steps.

There are many `inflate()` states, but we will only focus on the `EXTRA` state, which retrieves the extra field. Within the next chapter, we will present a more detailed analysis of this code section.

Fig. 2.4 shows how `inflate()` progresses through each state when decompressing a gzip file. For clarity, transitions to error states are not captured, and only the part that processes the gzip header is shown. The decoding of the header is followed by states that process the compressed data and the gzip trailer. If a gzip file is being processed, the states are always processed in that order[24].

Some code sections of individual states may be skipped depending on the presence of optional fields and the previous call to `inflateGetHeader()`. This is why the vulnerability does not affect applications that do not call `inflateGetHeader()` – the vulnerable code section is skipped.

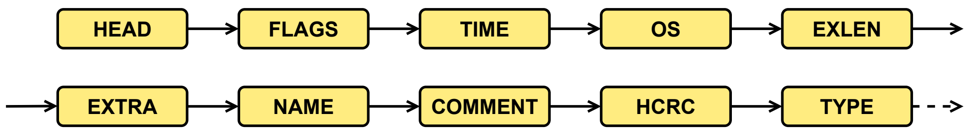


Figure 2.4: Visualization of gzip header processing [31].

[24]Except an error occurs.

Chapter 3

Exploiting the Vulnerability

One of the objectives of this thesis is to explore the possibility of exploiting the CVE-2022-37434 vulnerability to execute an attacker's code. After a more thorough analysis, we have found that code execution is indeed possible, and in this chapter, we will describe how it can be achieved.

First, we will introduce the `EXTRA` state, which is the vulnerable part of the `inflate()` function, and describe how this state works and why it is vulnerable in the first place. Next, we will describe how the vulnerability behaves and under what circumstances a buffer overflow can be triggered. Then we will focus on how to arrange the memory so that the overflow does not cause an application to crash. Finally, we will describe how a code may be injected into an application via a malicious gzip file.

We performed the analysis on Linux with the glib 2.35 memory manager and Windows 10, version 22H2, with its default memory manager. For Linux, we chose the Ubuntu 22.04 LTS distribution.

As in the previous chapter, the information presented in this chapter applies to the Zlib library version 1.2.12.

3.1 State EXTRA

The vulnerable part of the Zlib library is the `EXTRA` state of the `inflate()` function that is used to retrieve an extra field from a gzip header. Listing 3.1 shows the code of this vulnerable state. To understand how the `EXTRA` state works, it is essential to understand the variables it uses.

The `inflate()` function is passed a `z_stream` as an argument. Initially, `inflate()` copies several values from the variables of the passed `z_stream` into its local variables so that it can access them faster. Updated values are copied back to the `z_stream` before returning from `inflate()`. One such variable is `state`, which is a pointer to the internal state of `inflate()`. This internal state is used to maintain the context of the ongoing decompression.

Other such variables are `have` and `next`, which correspond to the `avail_in` and `next_in` variables from the passed `z_stream`.

The `state->head` variable is a pointer to a `gz_header` structure into which the gzip header information should be retrieved. The individual variables of the `gz_header` structure have already been described above. Without the previous call to `inflateGetHeader()`, the `state->head` is set to `Z_NULL`.

Listing 3.1 State `EXTRA` of the `inflate()` function [30].

```c
case EXTRA:
    if (state->flags & 0x0400) {
        copy = state->length;
        if (copy > have) copy = have;
        if (copy) {
            if (state->head != Z_NULL &&
                state->head->extra != Z_NULL) {
                len = state->head->extra_len - state->length;
                zmemcpy(state->head->extra + len, next,
                        len + copy > state->head->extra_max ?
                        state->head->extra_max - len : copy);
            }
            if ((state->flags & 0x0200) && (state->wrap & 4))
                state->check = crc32(state->check, next, copy);
            have -= copy;
            next += copy;
            state->length -= copy;
        }
        if (state->length) goto inf_leave;
    }
    state->length = 0;
    state->mode = NAME;
```

- `state->check` is a protected copy of check value.
- `state->flags` holds the gzip header method and flags.
- `state->head` is a pointer to a `gz_header` where the gzip header information should be retrieved.
- `state->length` determines how many bytes still need to be processed for the given state. For the `EXTRA` state, this value is initialized in the `EXLEN` state, in which the size of the extra field is loaded. This value can be at most 65 535 because the `XLEN` field in the gzip header specifying the size of the extra field is the size of two bytes, and a larger value cannot be written into two bytes.
- `state->mode` specifies the current inflate mode.
- `state->wrap` is used to hold additional information about the ongoing compression. Its bit 0 is true for the zlib wrapper, bit 1 is true for the gzip wrapper, and bit 2 is true to validate the check value.

- `state->head->extra` is a pointer to the memory into which the extra field should be retrieved.
- `state->head->extra_len` specifies the extra field length.
- `state->head->extra_max` specifies the maximum size in bytes of the memory provided for `extra`.
- `copy` specifies the number of stored bytes that can be copied.
- `have` specifies how many bytes are available in the input. It corresponds to the `avail_in` variable of the passed `z_stream`.
- `len` determines how many bytes have already been loaded within the `EXTRA` state.
- `next` is a pointer to the next input. It corresponds to the `next_in` variable of the passed `z_stream`. [31], [30]

Now we can explain how the `EXTRA` state works. To begin with, an overflow can occur when the `zmemcpy()` function is called (line 769). This function is not vulnerable by itself. However, combined with the previous code, it can be called with argument values that the authors did not intend, leading to an overflow. Thus, for an overflow to occur at all, line 769 must be reachable.

First, whether the gzip file being decompressed contains an extra field is checked (line 762). At the time of executing the `EXTRA` state, this information is stored in the `state->flags` variable. If the gzip file does not contain an extra field, there is nothing to process, and the vulnerable code that stores the extra field is skipped. Therefore, the gzip file must contain an extra field for an overflow to be triggered.

Then it is determined how many bytes of the extra field can be copied, and this information is further held in the `copy` variable. Ideally, copying all remaining bytes of the extra field is possible. The number of remaining bytes to copy is given by the `state->length` variable. If not enough bytes are available on the input, this value is reduced to the number of bytes available (lines 763–764).

Next, whether a `gz_header` is available is checked. If so, it is then checked if memory to store an extra field has been provided (lines 766–767). Otherwise, there is nowhere to save the extra field, and the vulnerable code is skipped. A `gz_header` is provided via the `inflateGetHeader()` function, and without calling it, `state->head` is set to `Z_NULL`. This gives two more conditions that must be met to trigger an overflow. There must be a previous call to `inflateGetHeader()`, and the application must provide memory to store an extra field.

If the memory to store an extra field is provided, the number of bytes of already processed extra field data is calculated and stored in the `len` variable.

This is followed by a call to `zmemcpy()` function, which copies the specified number of bytes from source to destination (lines 768–771). The `zmemcpy()` function takes three arguments. The first argument is a pointer to the destination array where the content is to be copied, the second argument is a pointer to the source of data to be copied, and the third argument specifies the number of bytes to be copied.

Within the `EXTRA` state, the arguments are passed as follows: the destination is specified as the sum of `state->head->extra` and `len`, the source is specified as `next`, and the number of bytes to be copied depends on the memory size provided for `extra`. The extra field data is, therefore, copied from the input to `extra` at offset `len`, and only as many bytes as can fit in `extra` should be copied. However, this expected behavior is violated under some circumstances, resulting in an overflow.

If the already stored extra field data together with the data to be copied would not fit in `extra`, the value of the third argument is determined as the result of subtracting `len` from `state->head->extra_max`. The idea behind this operation is to copy only as many bytes as can still fit in `extra`. The problem is that there is no check whether `len` is smaller than `state->head->extra_max` before calling `zmemcpy()`. That subtraction may lead to data type underflow, as both these variables are of `unsigned int` data type and `len` may be greater than `state->head->extra_max`.

If `len` is greater than `state->head->extra_max`, the maximum amount of the extra field data has already been stored, and there should be no further writes to `extra`. However, an underflow of the `unsigned int` data type occurs instead, as the result of the subtraction is negative. The `zmemcpy()` function is subsequently requested to copy nearly four gigabytes of data, causing an overflow of the provided memory. Considering the maximum possible size of an extra field, the result of the subtraction cannot be less than -65 535, which for an `unsigned int` corresponds to a value close to four gigabytes. So if `len` happens to be greater than `state->head->extra_max`, such a call to `zmemcpy()` causes an overflow. The following section will focus on how this overflow can be triggered.

Here the question may arise whether, when an underflow occurs, this value is always close to four gigabytes, even on 32-bit and 64-bit architectures, and whether it cannot be influenced in some way. We will come back to this later, but for now, we can reveal that it is always almost four gigabytes regardless of 32-bit and 64-bit architecture.

The remaining part of the `EXTRA` state is not so interesting for the vulnerability. After `zmemcpy()` is called, if `FHCRC` flag is set, the CRC32 check value is calculated and stored in `state->check` (lines 773–774). Subsequently, the information about the input data is updated to match the progress made. The number of bytes available on the input is reduced by the number of bytes that have been processed, the pointer to the input is shifted, and the number of bytes that still need to be processed within the `EXTRA` state is decreased accordingly (lines 775–777).

Finally, it is checked if the entire extra field has been processed. If not, the execution continues in `inf_leave`, where the total counts and the check value are updated, followed by a return from the `inflate()` function. In this case, the execution will continue at the `EXTRA` state the next time `inflate()` is called. If the entire extra field has already been processed, the state is changed to `NAME`, and `inflate()` starts processing that state.

3.2 Triggering an Overflow

Having analyzed the behavior of the `EXTRA` state, we can now take a closer look at how an overflow can be triggered in the library. As a reminder, to trigger an overflow, the `len` variable must be greater than `state->head->extra_max` at the time of calling `zmemcpy()`. Given how the value of the `len` variable is calculated, this can only be achieved if the extra field is larger than the memory provided for `extra`, and, therefore, we will focus only on these scenarios.

3.2.1 Possible Scenarios

In the first processing of the `EXTRA` state, the `state->head->extra_len` variable is equal to `state->length`. The `len` variable is therefore equal to zero, which means that it cannot be greater than `state->head->extra_max`. Thus, an overflow can only be triggered if the extra field is processed in more than one `inflate()` call.

If any extra field data is available for processing in the `EXTRA` state, the value of the `state->length` variable decreases with each call to `inflate()` (line 777). This way, the entire extra field is gradually processed.

Another possible scenario is the following. The extra field is larger than the memory provided for `extra`, and the entire extra field has not yet been processed in the previous call to `inflate()`. All the extra field data retrieved fit in `extra`, so it was stored there. Then the `inflate()` is called again. During this call, the remaining part of the extra field is processed. However, the remaining part does not fit in `extra`, so it must be truncated. At this point, `len` is smaller than `state->head->extra_max`, so this scenario does not cause an overflow. Since the extra field is completely processed, the `inflate()` function starts processing the next state.

The last and the most interesting scenario, since it causes an overflow, is the following. The extra field is larger than the memory provided for `extra`, and the entire extra field has not yet been processed in the previous call to `inflate()`. The last retrieved extra field data did not fit into `extra`, so only the truncated part of it was stored. More than `state->head->extra_max` bytes must have been processed since all the retrieved data did not fit into `extra`. Then, the `inflate()` function is called again. At this point, `len` is necessarily greater than `state->head->extra_max`. The sum of `len` and `copy` is greater than `state->head->extra_max`, so by the condition from line 770, the number of bytes to copy is determined as the result of subtracting `len` from `state->head->extra_max`. However, since `len` is greater, the data type underflow occurs, resulting in an overflow.

3.2.2 Overflow Demonstration on a Specific Example

To better understand how an overflow can be triggered, we will demonstrate it with a specific example. For these purposes, we have chosen an example from the official Zlib documentation from the *Zlib Usage Example* chapter [32]. This chapter describes how the authors intend the `inflate()` function to be used. Therefore, it can be expected that this is how users use it as well. Listing 3.2 shows the pseudocode of this example[25].

Compared to the original example, we made minor changes. The difference is the use of `inflateInit2()` instead of `inflateInit()` and the added call to `inflateGetHeader()`. We made these changes since we want to use `inflate()` in the gzip decompression mode.

The pseudocode does not show how the individual structures are initialized for better clarity. The description of how they should be initialized is given in the previous chapter. Also omitted are code sections that handle the decompression errors. The complete code can be found in the proof-of-concept programs in the attached files.

Listing 3.2 Pseudocode of decompression using the `inflate()` function [32].

```
int inf( FILE * source, FILE * dest ) {

    ... z_stream  initialization ...
    ... gz_header initialization ...

    inflateInit2(...);       // init for gzip decompression
    inflateGetHeader(...);   // provide a gz_header

    // decompression loop
    do {
        ... read input from the source file ...
        ... provide loaded input to inflate() ...
        do {
            ... provide output space for inflate() ...
            inflate(...);
            ... write the generated output to the dest file ...
        } while ( inflate() can generate output );
    } while ( inflate() not reached the end of compressed data );

    inflateEnd(...);
    return Z_OK;
}
```

[25]Our proof-of-concept programs are also based on this example.

The `inf()` function decompresses a gzip stream from the input file and writes decompressed data to the output file. Internally, it uses the `inflate()` function. Before every call to `inflate()`, the application should provide input and available output space. Otherwise, there may be no progress made. This is done by passing the input and output buffers via the `z_stream` structure. We will set these buffers to the size of 32 KB. The input file is, therefore, read in 32 768-byte blocks, except the last block, which may be smaller.

The size of the input buffer is essential. If it were too large, the extra field would always be processed in a single call to `inflate()`, which, as we have already discussed, is the case that does not lead to an overflow. We consider the buffer size of 32 768 bytes a reasonable value that another programmer could use as well, as it is a nice rounded number of a reasonable size that is a multiple of two. At the same time, an overflow can still be triggered with such an input buffer size.

For the vulnerable part of the `EXTRA` state to be reachable, there must be a previous call to the `inflateGetHeader()` function, via which a `gz_header` structure to store gzip header information is provided. An application should provide the memory to retrieve optional fields from the header or set the appropriate `gz_header` variables to `Z_NULL`. We will further assume that memory of 32 768 bytes has been provided for `extra`, `name`, and `comment`. The values of `extra_max`, `name_max`, and `comm_max` are therefore 32 768.

The memory size provided for `extra` is essential since it directly determines whether the vulnerability is exploitable. If the provided memory were larger than 65 535 bytes, an overflow could not occur since `len` would never be greater than `state->head->extra_max`. According to the documentation [27], only the truncated data will be stored if an optional field is larger than the memory provided. Thus, providing a memory for optional fields of this size is perfectly legitimate.

Although it would not be necessary to provide memory for `name` and `comment` to demonstrate the vulnerability, we try to approximate as closely as possible how another user might perform the decompression using `inflate()`. If they already decided to retrieve the gzip header, it is expected that they would want to retrieve the entire header, including all available optional fields.

With these sizes of buffers and the memory provided for `extra`, an overflow can be triggered, for example, when decompressing a gzip file with an extra field of size 65 535 bytes. An example of how such a file might look is shown in Fig. 3.1. Only the extra field of the optional fields is present since it is the only one on which the vulnerability depends. However, any other optional fields could be present as well.

We emphasize that this is not the only possible combination of input buffer size, memory size provided for `extra`, and the size of an extra field of the gzip file that leads to an overflow. There are many such combinations, and this is only one of them.

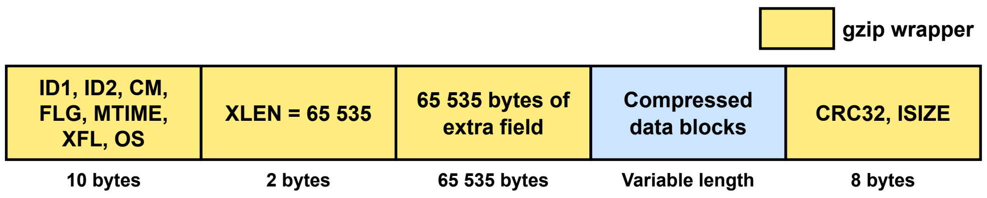


Figure 3.1: Structure of a gzip file with an extra field of size 65 535 bytes.

Assuming that the input buffer and memory provided for `extra` are the sizes of 32 768 bytes, the processing of a valid gzip file with an extra field of size 65 535 bytes by the `inf()` function is as follows (we will focus on what happens in the `EXTRA` state of the `inflate()` function):

1. In the first call to `inflate()`, an input of size 32 768 bytes is provided. By the time the `EXTRA` state is first executed, the first 12 bytes have already been processed by previous states[26]. Thus, the `EXTRA` state is given the remaining 32 756 bytes as input. The size of the extra field is 65 535 bytes, so currently, the entire available input contains extra field data. All input data fits into `extra` and is copied there.

2. In the second call to `inflate()`, the execution continues in the `EXTRA` state since the extra field has not yet been fully processed, and another 32 768 bytes are provided on input. The entire input again contains extra field data, but this time, there is insufficient space in `extra`, so based on the condition on line 770 of Listing 3.1, only 12 bytes are stored. However, all 32 768 bytes of the input are processed within this call, as they all are extra field data.

3. In the third call to `inflate()`, there are 11 more bytes to be processed from the extra field, so the execution again continues in the `EXTRA` state. Either 32 768 bytes or less are available on the input, depending on the compressed data size. In this call, the value of the `len` variable is calculated as 65 524, and the value of `state->head->extra_max` is 32 768 (the memory size provided for `extra`). This is exactly the case when `len` is greater than `state->head->extra_max` before calling `zmemcpy()`. In this particular scenario, 4 294 934 540 bytes are copied, leading to an overflow of the provided space.

As can be seen, the compressed data size does not affect whether an overflow occurs. Such an overflow causes, in most cases, a crash of an application, but under certain circumstances, it may not. If an overflow occurs and the application does not crash, the vulnerability may even be exploited to code execution. However, even the mere fact that it is possible to cause an application crash is a big problem, as attackers can perform denial-of-service attacks.

[26]It is always 12 bytes given the fixed size of the fields preceding the extra field.

3.3 Code Execution

In this section, we will describe how the CVE-2022-37434 vulnerability can be exploited to code execution. The principle of the attack is based on the fact that an attacker can take advantage of the overflow to overwrite memory locations that they would normally not be able to write to. These memory locations may contain interesting objects whose overwriting may influence the program flow. However, code execution can only be achieved if the overflow does not result in a crash of the application.

First, we will discuss how many bytes are overflowed and what conclusions can be drawn regarding exploiting the vulnerability on 32-bit and 64-bit architectures. Next, we will describe how the memory must be arranged so that the overflow does not cause an application to crash. Finally, we will present how a malicious file can be created whose decompression via the `inf()` function will lead to arbitrary code execution.

3.3.1 Analysis of the Overflow Sizes

We can finally explain why, when an overflow occurs in the Zlib library due to the CVE-2022-37434 vulnerability, it is always an overflow of almost four gigabytes. This conclusion is based on a thorough analysis that we performed on the vulnerable code parts at the assembler level. The generated assembler is compiler dependent – we used the gcc compiler version 11.3.0[27]. The analysis was performed for both 32-bit and 64-bit versions.

First, we analyzed the `EXLEN` state responsible for retrieving the extra field length. The extra field length is specified by two bytes. However, its retrieved value is stored in the `state->length` variable, which is of `unsigned` data type that takes four bytes in both analyzed cases. We focused on whether it is not possible to influence the upper two bytes and thus change the retrieved value.

The extra field length is retrieved via the `NEEDBITS()` macro, which loads the required number of bits from the input into a variable of `unsigned long` data type. In the analyzed cases, the `unsigned long` takes four bytes on a 32-bit system and eight on a 64-bit system. The retrieved extra field length value is first stored in `unsigned long`, then copied to `unsigned`. Either way, only the lower four bytes are reflected in the final value.

To retrieve the bits, `NEEDBITS()` uses the `movzx` instruction, which copies the content of the source operand (register or memory location) to the destination operand (register), and zero extends the value. This instruction ensures the value is always retrieved correctly, regardless of the previous register value.

Based on the assembler analysis performed, the `NEEDBITS()` macro works as expected – the requested bits are retrieved, and the unused upper bits are set to zero. Therefore, we conclude that for the analyzed cases, the upper two

[27] `https://gcc.gnu.org/gcc-11/`

bytes cannot be influenced in any way and that the retrieved extra field length corresponds to the value specified in the gzip header of the decompressed file.

The third argument of the `zmemcpy()` function, which specifies the number of bytes to be copied, is of the `size_t` data type. The size of `size_t` varies depending on whether the application is 32-bit or 64-bit. On a 32-bit system, `size_t` takes four bytes, and on a 64-bit system, eight. One might wonder if the number of bytes by which the buffer is overflowed does not vary depending on whether the application is 32-bit or 64-bit since the result of subtracting `len` from `state->head->extra_max` is stored in `size_t`, whose size is architecture dependent.

The answer is no. It does not vary. On both 32-bit and 64-bit systems, `len` and `state->head->extra_max` are four-byte unsigned variables. According to the assembler, the result of their subtraction is first saved as a four-byte value, and only then it is converted and stored in `size_t`. Thus, the calculated value remains the same in both cases. On a 64-bit system, the upper four bytes are zero-extended.

Thus, the number of bytes overflowed is the same for both 32-bit and 64-bit systems. For an overflow to occur, the result of the subtraction must be negative. As we have already discussed, the result of the subtraction cannot be less than -65 535. Considering these facts, there is always an overflow of almost four gigabytes.

3.3.2 Analysis of the Affected Memory Areas

At this point, we have yet to discuss what the overflow overwrites. Looking again at line 769 of Listing 3.1, the data are copied from the input (variable `next`) to `extra` at offset `len`. However, once an overflow occurs, `len` is greater than `state->head->extra_max`, so all the data to be copied will be written beyond `extra`. Thus, this is not a typical overflow scenario where the data is written into a buffer, and the buffer boundaries are not checked during the write operation, causing the overflow. In our case, the data is written directly outside the buffer. A visualization of the overflow is shown in Fig. 3.2.

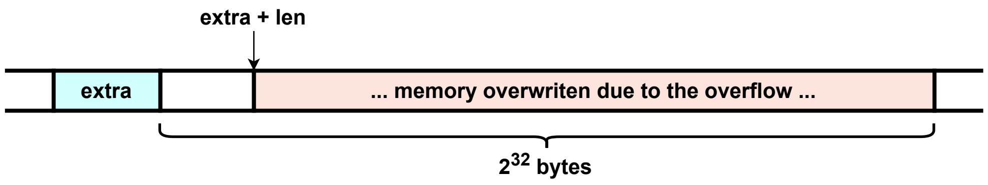


Figure 3.2: Visualization of the overflowed memory.

The greater the value of `len`, the fewer bytes are overflowed. The memory is always overflowed from `extra` at offset `len` up to the position of 2^{32} bytes from the end of `extra`. As a reminder, `len` can only contain values from 0 to 65 535.

3.3.3 Exploitability on 32-bit and 64-bit Architectures

The overflow is caused by the `zmemcpy()` function, which is called at the time when it should not be called anymore as there is no more space in `extra`. This function performs both read and write operations. If it fails to perform either of these operations, the application crashes. To prevent the crash, the source data must be read from memory with read permissions and written to memory with write permissions.

Thus, based on what we already know about the examined overflow, to avoid a crash, there must be almost four gigabytes of readable memory behind the input buffer and almost four gigabytes of writable memory behind `extra`[28]. We will consider a simplified scenario, where there must be exactly four gigabytes (2^{32} bytes) of readable and writable memory behind the buffers, as the maximum number of bytes by which these memories can theoretically be smaller (65 535 bytes) is negligible compared to the rest.

On a 32-bit architecture, a process has a maximum of four gigabytes of virtual address space. However, a part of this space is reserved for the system. For example, Windows and Linux operating systems have reserved one or two gigabytes [33], [34]. Thus, a process can allocate a maximum of around three gigabytes of memory.

When `zmemcpy()` is requested to copy almost four gigabytes of data on a 32-bit architecture, it will necessarily attempt to read or write to memory that is either unallocated, has no read or write permissions, or will try to access the `NULL` address. All of these cases lead to a crash of an application. Therefore, exploiting the CVE-2022-37434 vulnerability to code execution on a 32-bit architecture is not possible. However, the vulnerability can still be exploited for denial-of-service attacks.

On a 64-bit architecture, a process has a maximum of 2^{64} bytes of virtual address space, meaning there may be enough allocated memory behind the input buffer and `extra` so that an overflow does not cause the application to crash. If other conditions are met, code execution can be achieved.

3.3.4 Memory Layout Preventing a Crash

If there are 2^{32} bytes of contiguous memory with read permissions directly behind the input buffer and 2^{32} bytes of contiguous memory with write permissions behind `extra`, the overflow caused by the CVE-2022-37434 vulnerability will not result in a crash, which under certain circumstances can lead up to code execution. We will describe how to achieve such a memory layout in the heap memory segment[29]. We emphasize that the following demonstrations highly depend on the memory manager used.

[28]These memory blocks may overlap.

[29]An overflow could also occur in the stack memory segment if the input and `extra` buffers were allocated on the stack.

Fig. 3.3 shows an example of a memory layout that satisfies the conditions to prevent a crash. We will now demonstrate how it can be achieved on a Linux operating system with the glibc 2.35 memory manager and on Windows 10, version 22H2, with its default memory manager.

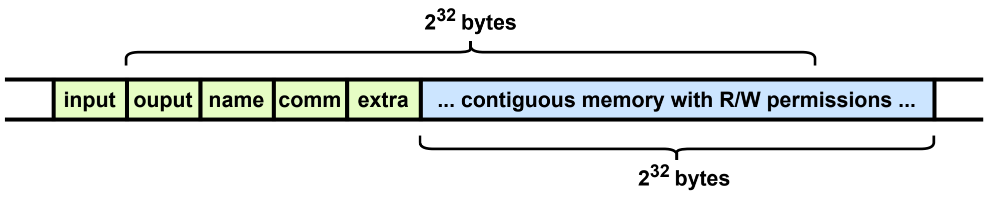


Figure 3.3: Example of a memory layout that prevents the overflow triggered by the CVE-2022-37434 vulnerability from causing an application crash.

3.3.4.1 Arranging Memory on Linux with glibc 2.35

We will now describe how to create the desired memory layout on Linux using the glibc's `malloc()`. The glibc's `malloc()` divides the memory it receives from the operating system into chunks of various sizes and accomplishes the allocation by assigning one or more consecutive chunks. Each chunk has its size and can be either in use or free. Freed chunks are stored in lists called bins so that they can be subsequently quickly and efficiently reused for other allocation requests. When allocating, bins are first searched for an available free chunk of sufficient size in bins, and if such a chunk exists, it is assigned. Otherwise, `malloc()` asks the operating system for additional memory. [35]

This behavior can be leveraged to create the desired memory layout. An example of how it can be achieved on Linux with glibc 2.35 is shown in Listing 3.3.

Listing 3.3 Creating the memory layout shown in Fig. 3.3 on Linux with glibc 2.35.

```
#define CHUNK 32768

int inf ( FILE * source , FILE * dest) {
    // This memory is divided between multiple buffers
    char * memory = malloc ( 5 * CHUNK );
    ...
}

void arrangeMemory ( char ** data ) {
    data = malloc ( 132000 * sizeof ( char * ) );
    for ( size_t i = 0; i < 132000; ++i )
        data[i] = malloc ( CHUNK )
    for ( size_t i = 0; i < 5; ++i )
        free ( data [i] );
    inf (...);
}
```

The `arrangeMemory()` function is assumed to be executed first. Within the `arrangeMemory()` function, a large contiguous memory region is created by calling `malloc()` in a loop[30]. Creating the contiguous large memory region in this manner makes freeing only part of it possible. So afterward, the first few chunks are freed so that their size matches the memory size allocated within the `inf()` function. Finally, the `inf()` function is called. When `inf()` then calls `malloc()`, it is assigned the just-freed memory chunks. Thus, the memory allocated within `inf()` is located directly in front of the large contiguous memory region, and the created memory layout matches the desired layout shown in Fig. 3.3.

In this example, we used the `malloc()` and `free()` functions, which are standard C library functions available on most operating systems and are, therefore, the most commonly used functions for dynamic allocation and deallocation. The memory assigned by `malloc()` has read and write permissions. The example is particularly interesting because we forced the memory manager to create a specific memory arrangement using legitimate operations. If an attacker could control when an application allocates and deallocates memory, he could force the memory manager to create such an arrangement.

The example is intended to illustrate a real-life scenario that could occur. Let's consider a situation where an application runs on a server and processes user requests. One of its services is decompressing a gzip file, for which it might use the `inf()` function. However, the application also provides other services that involve memory allocations and deallocations. If an attacker managed to force the application to allocate enough memory (e.g., by sending the same request that allocates memory many times), then caused part of the memory to be freed, and then requested the decompression of the gzip file, they could potentially achieve the desired memory layout.

3.3.4.2 Arranging Memory on Windows 10, version 22H2

Achieving the desired memory layout on a Windows 10, version 22H2, with its default memory manager is much more complicated, as the pages of a process's virtual address space can be in one of the following states:

- **Free** page can be reserved, committed, or both simultaneously. Attempting to read from or write to a free page results in an access violation exception.

- **Reserved** page is reserved for future use and cannot be used by other allocation functions. It is inaccessible, so attempting to read from or write to a reserved page creates an access violation exception. It is available to be committed.

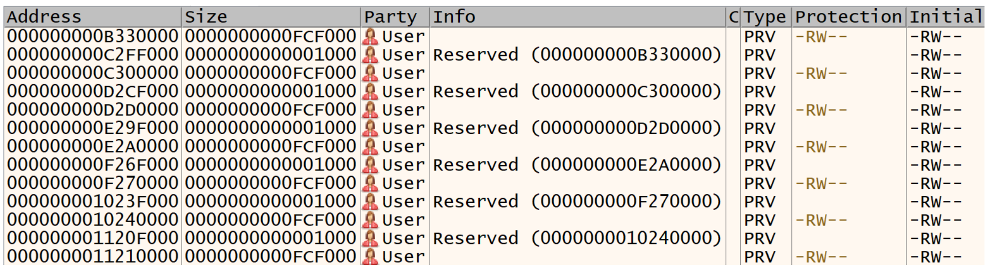
[30]It is taken advantage of that the glibc heap arranges the memory allocated this way sequentially, resulting in a contiguous memory region.

- **Commited** page is accessible, and the access is controlled via the memory protection constants. [36]

This mechanism is particularly useful when reserving space for a large buffer that is not all needed immediately but needs to be contiguous. However, it severely limits the ability to create the memory layout shown in Fig. 3.3. Nevertheless, it is still possible. It is only more complicated, making the vulnerability more difficult to exploit.

The `malloc()` function implemented using the Windows 10, version 22H2, heap manager differs from the glibc's `malloc()`. However, many of its properties are the same. Just like the glibc's `malloc()`, it first tries to assign previously freed memory from bins when allocating, which can be leveraged for arranging the specific memory layout.

Considering that property, we tried to use the same strategy as in the previous example to create the desired memory layout on Windows 10, version 22H2. However, we could not succeed using the `malloc()` and `free()` functions because of the reserved pages that the Windows heap manager leaves between committed chunks once they reach a specific size. Fig. 3.4 shows the virtual memory after such an attempt. As can be seen, there are always `0x1000` bytes of reserved memory without access between read-and-write memory chunks, making the allocated memory non-contiguous.



Address	Size	Party	Info	C	Type	Protection	Initial
000000000B330000	0000000000FCF000	User			PRV	-RW--	-RW--
000000000C2FF000	0000000000001000	User	Reserved (000000000B330000)		PRV		-RW--
000000000C300000	0000000000FCF000	User			PRV	-RW--	-RW--
000000000D2CF000	0000000000001000	User	Reserved (000000000C300000)		PRV		-RW--
000000000D2D0000	0000000000FCF000	User			PRV	-RW--	-RW--
000000000E29F000	0000000000001000	User	Reserved (000000000D2D0000)		PRV		-RW--
000000000E2A0000	0000000000FCF000	User			PRV	-RW--	-RW--
000000000F26F000	0000000000001000	User	Reserved (000000000E2A0000)		PRV		-RW--
000000000F270000	0000000000FCF000	User			PRV	-RW--	-RW--
000000001023F000	0000000000001000	User	Reserved (000000000F270000)		PRV		-RW--
0000000010240000	0000000000FCF000	User			PRV	-RW--	-RW--
000000001120F000	0000000000001000	User	Reserved (0000000010240000)		PRV		-RW--
0000000011210000	0000000000FCF000	User			PRV	-RW--	-RW--

Figure 3.4: Virtual memory after several consecutive allocations in small chunks via the `malloc()` implemented using the Windows 10, version 22H2, heap manager.

We have noticed that the `malloc()` implemented with the Windows heap manager always reserves a memory address range, from which it then assigns memory. The memory range may grow as the memory allocation requests increase. Depending on the requested memory chunk size, the assigned memory may be contiguous even after multiple calls of `malloc()`, but only until it reaches a specific size. Then, a chunk of reserved memory is left after that contiguous committed memory chunk, and additional memory is allocated only after the reserved chunk. This process is repeated until the required amount of memory is allocated.

We have been trying to create the desired memory layout using `malloc()` and `free()` in many different ways. Still, we could not succeed. We found that allocating a large contiguous chunk in a single call to `malloc()` is possible. However, such large chunks also have a reserved memory region behind them. Thus, the ability to force the memory manager to create a specific memory layout is lost by doing so.

Conclusively, we cannot state with certainty that achieving the desired memory layout is impossible using the `malloc()` implemented using the Windows 10, version 22H2, heap manager. Still, based on our analysis, we believe it is. A more detailed examination of the `malloc()` implementation is out of the scope of this thesis.

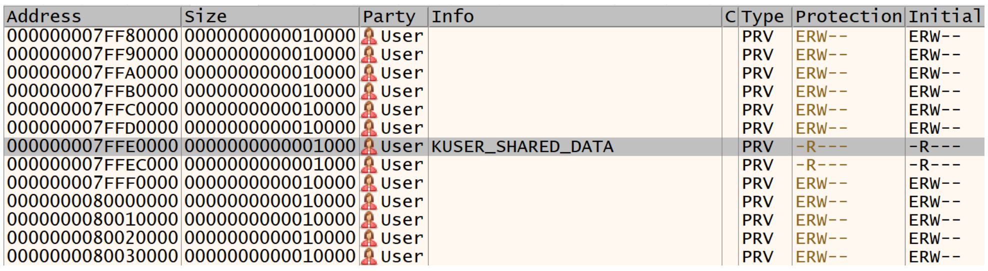
We think the reserved pages left between read-write memory chunks serve as guard pages. After all, one 4 096-byte guard page once every almost 16 megabytes is reasonable, even at the expense of more memory consumed, and may prevent such abuse.

Although we could not succeed in creating the desired memory layout using the `malloc()` and `free()` functions, we did succeed using the `VirtualAlloc()` and `VirtualFree()` functions. These are low-level functions from the Windows API that are used to manage dynamic memory directly.

In most cases, the `malloc()` and `free()` functions are more likely to be used for dynamic memory management, partly because they do not depend on the operating system and partly because they are easier to use and do not require knowledge of low-level memory management. However, `VirtualAlloc()` and `VirtualFree()` are also used and are even necessary in some situations, for example, when it is needed to work with memory protection constants.

The `VirtualAlloc()` function can be passed an address from which to allocate memory via the first argument. However, since it is not needed to achieve the desired memory layout, we will set this argument to `NULL`, leaving the system to determine the allocated address.

`VirtualAlloc()` always assigns memory aligned on the system allocation granularity boundary and only allocates memory blocks in multiples of the system granularity [37]. The system allocation granularity boundary is `0x10000` bytes on many 64-bit Windows systems [38], and the system granularity is mostly `0x1000` bytes. We found that the assigned memory is contiguous if allocated in the system allocation granularity size chunks.

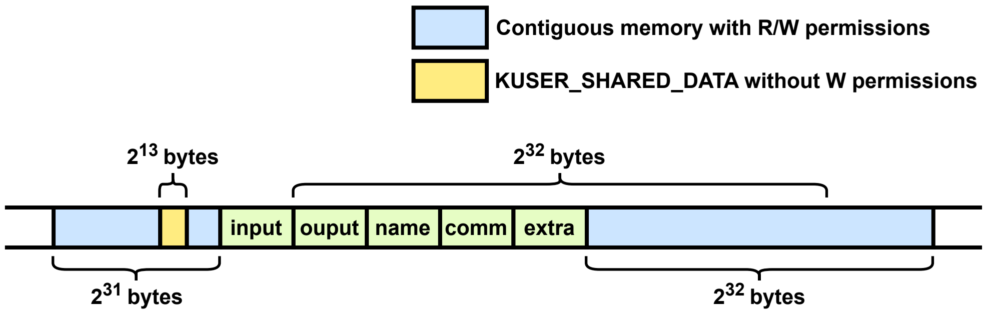


Address	Size	Party	Info	C	Type	Protection	Initial
000000007FF80000	0000000000010000	User			PRV	ERW--	ERW--
000000007FF90000	0000000000010000	User			PRV	ERW--	ERW--
000000007FFA0000	0000000000010000	User			PRV	ERW--	ERW--
000000007FFB0000	0000000000010000	User			PRV	ERW--	ERW--
000000007FFC0000	0000000000010000	User			PRV	ERW--	ERW--
000000007FFD0000	0000000000010000	User			PRV	ERW--	ERW--
000000007FFE0000	0000000000001000	User	KUSER_SHARED_DATA		PRV	-R---	-R---
000000007FFEC000	0000000000001000	User			PRV	-R---	-R---
000000007FFF0000	0000000000010000	User			PRV	ERW--	ERW--
0000000080000000	0000000000010000	User			PRV	ERW--	ERW--
0000000080010000	0000000000010000	User			PRV	ERW--	ERW--
0000000080020000	0000000000010000	User			PRV	ERW--	ERW--
0000000080030000	0000000000010000	User			PRV	ERW--	ERW--

Figure 3.5: `KUSER_SHARED_DATA` pages without write access located at address `0x7FFE0000` interrupting the allocated contiguous memory region.

As shown in Fig. 3.5, there are two `KUSER_SHARED_DATA` pages at address `0x7FFE0000` without write permissions interrupting the allocated contiguous memory. They are always placed to this fixed address by the Windows operating system when ASLR is disabled [39], so not much can be done with them. The problem is that the `VirtualAlloc()` starts assigning memory before these pages, and the contiguous allocated chunk of memory is interrupted by these pages after about two gigabytes.

Nevertheless, this problem can be easily bypassed by allocating six gigabytes of memory instead of four and freeing the memory chunks only after these two non-writable pages. That also results in a layout preventing the overflow from crashing an application. The layout should match the memory layout shown in Fig. 3.6. The blue memory regions represent the memory allocated via `VirtualAlloc()`.



Contiguous memory with R/W permissions
KUSER_SHARED_DATA without W permissions
2^13 bytes
2^32 bytes
input
ouput
name
comm
extra
2^31 bytes
2^32 bytes

Figure 3.6: Example of another memory layout preventing the overflow triggered by the CVE-2022-37434 vulnerability from causing an application crash.

Listing 3.4 Creating the memory layout shown in Fig. 3.3 on Windows 10, version 22H2, with the `VirtualAlloc()` and `VirtualFree()` functions.

```
#define CHUNK 65536

int inf ( FILE * source , FILE * dest ) {
    // This memory is divided between multiple buffers
    char * memory =  VirtualAlloc ( NULL , 2 * CHUNK , MEM_COMMIT |
                        MEM_RESERVE , PAGE_EXECUTE_READWRITE );
    ...
}

void arrangeMemory ( char ** data ) {
    data = ( char ** ) malloc ( 100000 * sizeof ( char * ) );
    for ( int i = 0; i < 100000; i++ ) {
      data[i] = VirtualAlloc ( NULL , CHUNK , MEM_COMMIT |
                        MEM_RESERVE , PAGE_EXECUTE_READWRITE );
    }
    for ( int i = 32768; i < 32770; i++ )
      VirtualFree ( data[i] , 0, MEM_RELEASE );
    inf (...);
}
```

An example of how to achieve the memory layout shown in Fig. 3.6 on Windows 10, version 22H2, with the `VirtualAlloc()` and `VirtualFree()` functions is shown in Listing 3.4. Again it is assumed that the `arrangeMemory()` is executed first. It is taken advantage of the fact that the memory allocated via the `VirtualAlloc()` function in allocation granularity-sized chunks is contiguous and that `VirtualAlloc()` first tries to assign previously freed memory.

3.3.5 Creating a Malicious File

Finally, we can describe what a malicious file[31] exploiting the CVE-2022-37434 vulnerability to code execution might look like. We have already shown an example of a file that triggers an overflow. However, we only focused on making the file trigger an overflow and did not care about its content which is essential for the code execution.

So far, we have yet to deal with what the memory is overwritten with. Looking again at line 769 of Listing 3.1, the second argument of `zmemcpy()`, which determines where the data are copied from, is the `next` variable. The `next` variable points to the next available input a thus corresponds to some part of the gzip file being processed. At the time the overflow occurs, depending on the input buffer size and the presence and size of each gzip file field, the input buffer may contain in this order: extra field data, name field data, comment field data, `CRC16` field data, compressed data, `CRC32` value, and `ISIZE` value.

To execute code not initially present in the application at its launch, the code must be injected into it. However, it is not injected as source code but as binary machine instructions, also known as opcodes, as the processor can only handle them[32]. Each instruction is uniquely identifiable and is represented by one or more bytes of opcodes. Typically, the code is first written in the assembler programming language and the corresponding opcodes are generated based on it, making the process quite simple. An attack consists of injecting a sequence of opcodes somewhere in memory and redirecting the flow of the application to execute these instructions.

In the case of the CVE-2022-37434 vulnerability, the code can be injected into an application via the content of a valid gzip file. It can be done in multiple ways. The first option is via the `CRC32` and `ISIZE` variables, as they can take on arbitrary values. However, their values should correspond to the compressed data, so for the file to be valid, compressed data would have to be created to match the specific values. In addition, only eight bytes can be stored in these fields. For this reason, it would be more worthwhile to store

[31]Even though we refer to these files as malicious, they are still legitimate gzip files meeting the RFC 1952 specification.

[32]Even source code in high-level programming language must be first translated into these opcodes before it can be executed. The translation is done during the compilation process.

opcodes as compressed data and calculate the appropriate `CRC32` and `ISIZE` values accordingly.

Storing opcodes as compressed data is pretty simple. It can be done by creating a gzip file with uncompressed content [40]. Storing uncompressed data in a gzip file is typically used when the data needs to be saved in gzip format so that it can be sent over a network or saved to disk but does not need to be compressed. As a result, the opcodes are stored in uncompressed form in a deflate stream with a gzip wrapper around it.

The only problem that arises is that even uncompressed data must contain a five-byte deflate header to be a valid deflate stream, and the value of these five bytes is hard to control. These five bytes specify that the content of a deflate stream is uncompressed and determines its size [41]. Nevertheless, the values of subsequent bytes can be controlled.

This way, injecting up to 65 535 contiguous arbitrary bytes (one deflate block) is possible[33]. However, we have already discussed that for an overflow to occur, the entire extra field, whose maximum size is 65 535 bytes, must not be processed on a single `inflate()` call. Therefore, the input buffer size remains the main limiting factor of how many bytes can be injected.

Fig. 3.7 shows what the input buffer content might look like when processing a gzip file with opcodes stored as compressed data[34]. As can be seen, the entire input buffer content, except for the five bytes of the deflate header, can be controlled. If the compressed data were shorter, it might not fill the remaining part of the input buffer. However, we aim to show that since it is possible to control the size and the content of the compressed data and the extra field, it is possible to control the contents of almost the entire input buffer.

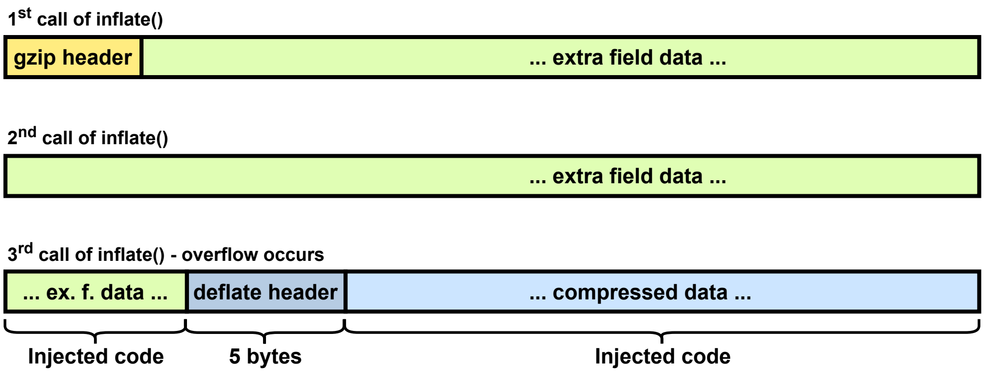


Figure 3.7: Visualization of the input buffer content during individual `inflate()` calls when processing a malicious gzip file that injects code via compressed data.

[33]This block may be followed by another one, but with a new deflate header.

[34]The number of `inflate()` calls may vary depending on the input buffer size and the size of the extra field.

It is also possible to create a non-valid gzip file that contains the opcodes right after the extra field (without a deflate header). However, such a file could not be decompressed in most cases, as the compressed data would mostly not be a valid deflate stream. Therefore, when decompressing, `inflate()` would return an error. Nevertheless, it remains a possibility, and an overflow would still occur since the actual processing of the compressed data occurs only after the overflow is triggered.

Another option is to store the code in the name and comment optional fields. There is an advantage that these fields can be arbitrarily long. The problem is that both these fields are terminated with a zero byte (`0x00`), and if a zero byte were a part of the injected code, it would terminate the field. The remaining part would be regarded as another field, possibly corrupting the deflate stream and making the gzip file non-valid. Although most codes can be created so that their opcodes do not contain zero bytes, sometimes using the zero bytes is unavoidable, especially for rewriting specific addresses.

The last option to store the code is in the extra field, whose content can be arbitrary. By the time the overflow occurs, either only extra field data is in the input buffer (and then it is possible to control the content of the entire buffer), or the rest of the extra field, compressed data, and the gzip trailer are there. However, a significant portion of the input buffer can be controlled even in the latter case.

If the name and comment fields are omitted, and the compressed data is small, the remaining part of the processed gzip file may not fill the rest of the input buffer. In such a case, the remaining part of the input buffer contains the extra field data from the previous `inflate()` call, which can be controlled.

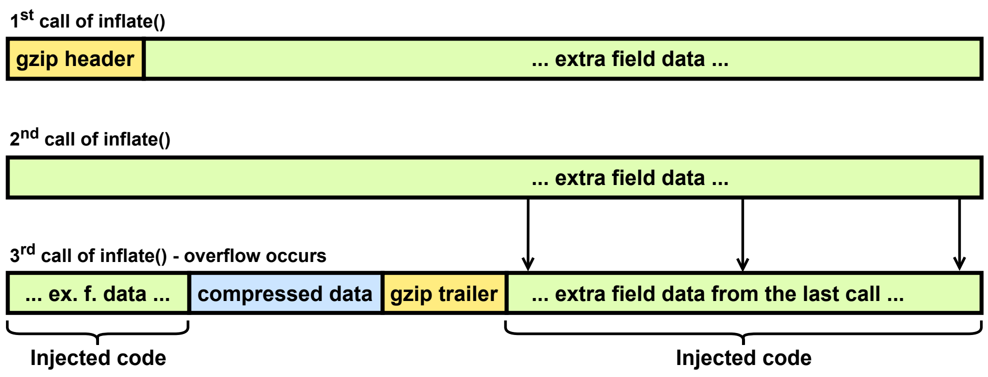


Figure 3.8: Visualization of the input buffer content during individual `inflate()` calls when processing a malicious gzip file that injects code via the extra field data.

Fig. 3.8 shows what the input buffer content might look like when processing a gzip file with opcodes stored in the extra field data. Our goal is to show the scenario where before the overflow is triggered, the remaining part of the gzip file is loaded into the input buffer and does not fill it. Thus, the bytes from the previous call are left in the input buffer, whose values can be controlled. Again, the number of `inflate()` calls may vary depending on the input buffer size and the size of the extra field.

Creating a specific file to inject code into a particular application always depends on the input buffer size. Depending on this size, it is then possible to modify the size and content of the individual fields of the malicious file so that the code is injected into the desired location of the input buffer. When an overflow occurs, the beginning of the overflowed memory is overwritten with either all or part of the input buffer (depending on where the `next` variable is pointing).

At this point, it is all down to what was present in the input buffer before the overflow occurred and the attacker's ingenuity. If, for example, a function pointer were located in the memory being overwritten, it may be possible to exploit it to execute the injected code by overwriting its value with the address of the injected code[35]. However, there are many more ways to exploit the CVE-2022-37434 vulnerability.

[35]This is how we demonstrated the code execution.

Chapter 4

Results of the Analysis

In this chapter, we will present the results of our CVE-2022-37434 vulnerability analysis.

First, we will summarize the conditions that would allow an attack that results in code execution to succeed. Next, we will comment on the exploitability of the CVE-2022-37434 vulnerability in OpenSSL, as OpenSSL uses the Zlib library and, therefore, may be vulnerable. Then we will discuss the available fixes. Finally, we will present our CVSS v3.1 severity rating for the vulnerability.

4.1 Conditions Allowing an Attack to Succeed

We will focus on attacks that aim to overwrite a function pointer in the overflowed memory with the address of the injected code. When the function initially stored in the function pointer is then called, the injected code is executed. This is how we demonstrated the code execution in the proof-of-concept programs.

However, the attack may be more complicated due to the buffer overflow protections, which are typically enabled. Suppose code execution is to be achieved by overwriting a function pointer. In that case, it is necessary to overcome the ASLR protection since it is required to know what address the function pointer should be overwritten with, and the NX-bit protection since the injected code must be located in an executable memory segment so that it can be executed.

We have decided not to describe how these protections can be overcome, as we do not want to give too detailed a description for potential attackers. However, it should be remembered that these protections may not work in every situation and do not make such an attack impossible. They only make it harder. The proof-of-concept demonstrations should serve as a compelling evidence of why it is not advisable to disable these protections.

Thus, for demonstration purposes, we disabled the ASLR protection and ensured the executability of the memory where the code is injected. On Linux, we ensured the memory executability via the `mprotect()` function, and on Windows, we directly allocated the memory as executable. The ASLR protection can be disabled at the operating system level.

The conditions that would allow achieving arbitrary code execution by overriding a function pointer by exploiting the CVE-2022-37434 vulnerability are as follows:

- The application must be 64-bit.
- The application must use the Zlib's `inflate()` function for decompression.
- The application must use `inflate()` to decompress files in the gzip file format.
- There must be a previous call to `inflateGetHeader()`, and memory must be provided for `extra`.
- The entire extra field, whose maximum size is 65 535 bytes, must not be processed in a single `inflate()` call. The input buffer must therefore be smaller than 65 547 bytes (65 535 bytes for the extra field and 12 bytes for the gzip header).
- There must be at least 2^{32} bytes of readable memory behind the input buffer and 2^{32} bytes of writable memory behind `extra`.
- There must be a function pointer in the overflowed memory.
- The address where the code is injected must be known or detectable.
- The memory region into which the code is injected must be executable.

It should be noted that there may be other ways to exploit the vulnerability than by overwriting a function pointer. Once an attacker can control what the memory is overwritten with, it only depends on what was present in the memory before the overflow occurred and how they are able to exploit it. For example, the vulnerability may be exploited by overwriting the internal heap structures.

Moreover, often an attacker does not even need to know the address or have executable memory to exploit the vulnerability. If the overflowed memory contained information about the user account, such as access rights, an attacker could assign themselves administrator privileges. They would need neither the knowledge of the address nor an executable memory. In such a case, even buffer overflow protections would not help.

4.2 Proof-of-concept of Code Execution

In this thesis, we analyzed the vulnerability on two operating systems and two memory managers – specifically Ubuntu 22.04 LTS with the glibc 2.35 memory manager and Windows 10, version 22H2, with its default memory manager. For both operating systems, we prepared a proof-of-concept that it is possible to exploit the CVE-2022-37434 vulnerability to execute code. For demonstration purposes, we prepared virtual environments.

The virtual environment for Ubuntu 22.04 LTS is available directly in the .ova format in the attached files, but we also provided steps to create it manually. The virtual environment for Windows 10 cannot be published for licensing reasons. However, we provided the steps for creating it.

Each virtual environment includes two scripts that perform the entire attack. The scripts are independent of each other, and the attack can be performed using any of them. There are two scripts because we have prepared two programs that can create a malicious file – the first stores the code to be injected into the compressed data, and the other stores it in the extra field. The individual scripts differ in using different programs for creating the malicious file. The scripts do the following:

1. Compile *zlib_inflate.c* (a program that performs gzip decompression).
2. Compile *zlib_deflate_comp.c* or *zlib_deflate_extra.c* (a program that creates a malicious gzip file).
3. Create a malicious file using the program created in step two.
4. Turn off ASLR (This step applies only to the Linux scripts. On Windows 10, the ASLR must be disabled manually.).
5. Decompress the malicious file using the program created in step one. In this step, the overflow is triggered, leading to the execution of the injected code.
6. Print the “End of script” message and pause before the user presses any key.

More detailed information about the demonstration programs and how to perform the attack is provided in the attached files.

The compiler used in both cases is the gcc version 11.3.0. However, we achieved code execution even with the gcc version 12.2.0. Therefore, we would like to emphasize that the CVE-2022-37434 vulnerability is not dependent on a particular compiler version, and changing just the compiler does not protect against the vulnerability.

On Ubuntu 22.04 LTS, we demonstrated the code execution by running a shell, and on Windows 10, by running a calculator. That may not look so terrifying. However, an actual attacker might not run a shell or a calculator. They might run something far more malicious. In the demonstration programs, the input buffer size is 32 768 bytes, and thus it is possible to execute more than 30 000 bytes of opcodes. This gives the attacker the ability to run almost anything. If the application were connected to the Internet, they might even be able to inject a code that downloads another malicious program from the Internet and executes it.

4.3 Zlib and OpenSSL

The OpenSSL cryptographic library uses Zlib and may, therefore, be vulnerable. OpenSSL can be used to implement a TLS (Transport Layer Security) client and server, and the TLS communication can be compressed. While compression in TLS is no longer recommended due to security risks and is even disabled by default in TLS 1.3, it can still be used. This could mean that an evil server may send a compressed malicious file to a legitimate client, or an evil client may send a malicious file to a legitimate server. Given that such an attack vector theoretically exists, we decided to examine it.

Although it seems theoretically possible, it cannot be implemented in practice since OpenSSL does not use the Zlib library for gzip compression and decompression, which is one of the necessary conditions for exploiting the CVE-2022-37434 vulnerability (see Section 4.1).

4.4 Reliability of Available Fixes

The CVE-2022-37434 vulnerability was remedied in Zlib version 1.2.13, released on October 13, 2022. That is quite a long time since the publication of the CVE Record on August 5, 2022. However, we believe the release was delayed because the first commit fixing the vulnerability introduced a new bug where the `NULL` pointer could be dereferenced. Thus, we assume the authors did not want to release a new library version containing additional bugs, so they waited to ensure the fix was correct.

The vulnerability was fixed by an added verification that the `len` variable is not greater than `state->head->extra_max` before calling `zmemcpy()`. As we discussed in the vulnerability exploitation chapter, the case when `len` is greater than `state->head->extra_max` leads to an overflow. Adding this condition ensures both that it is not possible to write outside `extra` when calling `zmemcpy()` and that the data type cannot be underflowed when determining the size of the data to be copied. Therefore, we consider the fix to be reliable.

The CVE-2022-37434 vulnerability had existed in Zlib since September 10, 2011, so the library had been vulnerable for almost terrifying twelve years. This proves that although some software is massively used and even open source, it can contain such a severe vulnerability. The CVE-2022-37434 vulnerability was difficult to detect due to the complexity of the decompression process via the `inflate()` function. It may have been introduced due to an oversight, but it is equally possible that it was introduced intentionally. That can, at the very least, raise some doubts.

As mentioned earlier, the Zlib library is widely used in many applications, operating systems, and software projects. Examples of some of these are:

- **Cryptographic libraries** – OpenSSL, GnuTLS
- **Database systems** – SQLite, PostgreSQL, MySQL
- **Multimedia software** – VLC media player, OBS, GIMP
- **Office suites** – Microsoft Office, LibreOffice
- **Operating systems** – Linux kernel, macOS
- **Web browsers** – Google Chrome, Mozilla Firefox
- **Web servers** – Apache, Microsoft IIS, NGINX

The CVE-2022-37434 vulnerability thus allows the potential exploitation of all software using vulnerable Zlib versions, which is highly concerning due to the library's widespread usage in various types of applications, including the large-scale ones.

Given Zlib's popularity and the severity of the CVE-2022-37434 vulnerability, it is essential to upgrade to the latest version of the library (currently Zlib 1.2.13) as soon as possible to avoid potential damage to users and systems using Zlib.

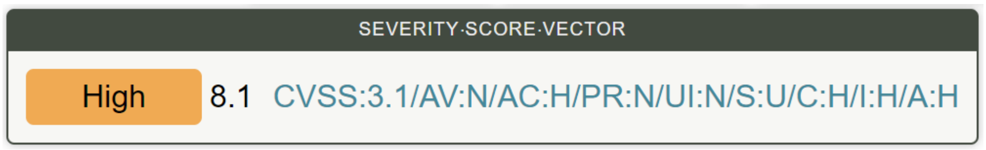
4.5 Our Vulnerability Severity Rating

Given that we have examined the CVE-2022-37434 vulnerability thoroughly, we consider it reasonable to assign it a severity score based on our analysis. We decided to use the CVSS v3.1 scoring system.

Given that the vulnerability can be exploited to code execution, we consider the Confidentiality, Integrity, and Availability metrics to be High. As can be seen from the demonstration programs, neither User Interaction nor Privileges are needed, so we scored both metrics as None. The attack can also be performed over a network, so we scored the Attack Vector metric as Network. The vulnerability exploitation cannot affect resources beyond the security scope managed by the security authority of the vulnerable component, so we scored the Scope metric as Unchanged.

The only change we would make from the current CVSS v3.1 Base score provided by NIST is in the Attack Complexity metric. Considering that an attacker would need to figure out the buffer sizes in the application, and the content of the overflowed memory, would need to find out if it is even possible to cause an overflow without crashing the application and would have to overcome the ASLR and the NX-bit protection to execute code, we consider the Attack Complexity to be High. However, as discussed in the previous chapter, performing denial-of-service attacks is pretty simple, so at the same time, we consider the Low rating that NIST assigned to the metric also reasonable.

Nevertheless, if the application only crashes, the Confidentiality and Integrity would not be High. Therefore, looking at vulnerability comprehensively, we find it more reasonable to score Attack Complexity as High. The resulting score vector is shown in Fig. 4.1.



SEVERITY·SCORE·VECTOR

High 8.1 CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H

Figure 4.1: CVSS v3.1 Base Score Vector of our CVE-2022-37434 vulnerability severity rating. The picture is taken from [13].

Conclusion

The goal of this thesis was to explore the CVE-2022-37434 vulnerability, find out when and how a buffer overflow can be triggered, and determine whether it could be exploited to code execution. After our vulnerability analysis, we concluded that code execution is indeed possible, and we have described how it can be done.

First, we introduced vulnerabilities in general, how they are published, and how they are scored. We focused on buffer overflow vulnerabilities, their exploitation techniques, and protections against them. Then, we presented the available information about the CVE-2022-37434 vulnerability.

In the second chapter, we described the gzip file format and the parts of the library relevant to the CVE-2022-37434 vulnerability.

In the third chapter, we discussed how the vulnerability could be exploited to code execution. We analyzed the vulnerable code segment, how the vulnerability behaves, and the scenarios that may lead to an overflow. Next, we described how a memory manager could be forced to arrange memory so that the overflow does not cause an application to crash. We also introduced several ways to create a malicious gzip file whose decompression may lead to an overflow and overwrite the overflowed memory with arbitrary values.

We performed the analysis on two operating systems and two memory managers – Linux with glic 2.35 memory manager and Windows 10, version 22H2, with its default memory manager. In both cases, we achieved code execution and prepared a proof-of-concept of it.

To demonstrate the exploitation of the CVE-2022-37434 vulnerability to code execution, we prepared virtual environments for both operating systems. We created scripts to automate the demonstrations.

Finally, we summarized the conditions that would allow exploitation resulting in code execution to succeed and provided our severity rating for the vulnerability.

The results of this thesis and the exploitation demonstrations should serve as a compelling evidence of why updating to the latest Zlib library version is advisable. In future work, it might be interesting to explore the possibility of achieving code execution by overriding the internal heap structures.

Bibliography

[1] CVE. About the CVE Program. [online], [Accessed 2023-02-01]. Available from: https://www.cve.org/About/Overview

[2] CVE. Glossary. [online], [Accessed 2023-02-01]. Available from: https://www.cve.org/ResourcesSupport/Glossary?activeTerm=glossaryCVEList

[3] HackerOne. What Are Bug Bounties? How Do They Work? [With Examples]. [online], Jul 2021, [Accessed 2023-02-04]. Available from: https://www.hackerone.com/vulnerability-management/what-are-bug-bounties-how-do-they-work-examples

[4] CVE. Published CVE Records. [online], [Accessed 2023-02-01]. Available from: https://www.cve.org/About/Metrics

[5] NOPSEC. Understanding the Difference Between Vulnerabilities and Exposures. [online], Jan 2022, [Accessed 2023-02-01]. Available from: https://www.nopsec.com/blog/difference-vulnerabilities-exposures/

[6] Balbix. What is a CVE. [online], [Accessed 2023-02-02]. Available from: https://www.balbix.com/insights/what-is-a-cve/

[7] CVE. CVE Process. [online], [Accessed 2023-02-02]. Available from: https://www.cve.org/About/Process#CVERecordLifecycle

[8] CVE. CVE Numbering Authority (CNA) Rules. [online], Mar 2020, [Accessed 2023-02-04]. Available from: https://www.cve.org/ResourcesSupport/AllResources/CNARules

[9] FIRST. Common Vulnerability Scoring System v3.1: Specification Document. [online], [Accessed 2023-02-04]. Available from: https://www.first.org/cvss/v3.1/specification-document

[10] NIST. Vulnerability Metrics. [online], [Accessed 2023-02-04]. Available from: `https://nvd.nist.gov/vuln-metrics/cvss`

[11] Balbix. What are CVSS Scores. [online], [Accessed 2023-02-04]. Available from: `https://www.balbix.com/insights/understanding-cvss-scores/`

[12] NIST. CVE-2022-37434 Detail. [online], Jan 2023, [Accessed 2023-02-01]. Available from: `https://nvd.nist.gov/vuln/detail/CVE-2022-37434`

[13] Chandan. CVSS v3.1 Base Score Calculator. [online], [Accessed 2023-02-03]. Available from: `https://chandanbn.github.io/cvss/`

[14] Anley, C.; Heasman, J.; Linder, F.; Richarte, G. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes, 2nd Edition.* USA: John Wiley & Sons, Inc., 2007, ISBN 9780470080238.

[15] Erickson, J. *Hacking: The Art of Exploitation, 2nd Edition.* USA: No Starch Press, 2008, ISBN 9781593271442.

[16] OWASP. Buffer Overflow. [online], [Accessed 2023-02-08]. Available from: `https://owasp.org/www-community/vulnerabilities/Buffer_Overflow`

[17] Wong, A. Execute Disable Bit – The BIOS Optimization Guide. [online], Aug 2017, [Accessed 2023-02-10]. Available from: `https://www.techarp.com/bios-guide/execute-disable-bit/`

[18] Seacord, R. *Secure Coding in C and C++, 2nd Edition.* Addison-Wesley Professional, 2013, ISBN 9780321822130.

[19] Plakosh, D. Guard Pages. [online], May 2005, [Accessed 2023-02-11]. Available from: `https://www.cisa.gov/uscert/bsi/articles/knowledge/coding-practices/guard-pages`

[20] CVE. CVE-2022-37434. [online], Oct 2022. Available from: `https://www.cve.org/CVERecord?id=CVE-2022-37434`

[21] SUSE. CVE-2022-37434. [online], Feb 2023, [Accessed 2023-02-13]. Available from: `https://www.suse.com/security/cve/CVE-2022-37434.html`

[22] Red Hat. CVE-2022-37434. [online], Feb 2023, [Accessed 2023-02-13]. Available from: `https://access.redhat.com/security/cve/cve-2022-37434`

[23] Apple. About the security content of macOS Ventura 13. [online], Oct 2022, [Accessed 2023-02-13]. Available from: `https://support.apple.com/en-us/HT213488`

[24] Debian. [SECURITY] [DLA 3103-1] zlib security update. [online], Sep 2022, [Accessed 2023-02-13]. Available from: `https://lists.debian.org/debian-lts-announce/2022/09/msg00012.html`

[25] Euccas, C. Understanding Zlib. [online], Jan 2019, [Accessed 2023-02-14]. Available from: `https://www.euccas.me/zlib/`

[26] Gailly, J.-l.; Adler, M. Zlib. [software], [Accessed 2023-02-14]. Available from: `https://zlib.net/`

[27] Gailly, J.-l.; Adler, M. zlib.h. [online], Mar 2022, [Accessed 2023-02-15]. Available from: `https://github.com/madler/zlib/blob/v1.2.12/zlib.h`

[28] Adler, M. How are zlib, gzip and zip related? What do they have in common and how are they different? [online], Dec 2022, [Accessed 2023-02-14]. Available from: `https://stackoverflow.com/questions/20762094/how-are-zlib-gzip-and-zip-related-what-do-they-have-in-common-and-how-are-they/20765054#20765054`

[29] Deutsch, P. GZIP file format specification version 4.3. RFC 1952, May 1996, doi:10.17487/RFC1952. Available from: `https://www.rfc-editor.org/info/rfc1952`

[30] Gailly, J.-l.; Adler, M. inflate.c. [online], Mar 2022, [Accessed 2023-03-02]. Available from: `https://github.com/madler/zlib/blob/v1.2.12/inflate.c`

[31] Gailly, J.-l.; Adler, M. inflate.h. [online], Mar 2022, [Accessed 2023-02-15]. Available from: `https://github.com/madler/zlib/blob/v1.2.12/inflate.h`

[32] Gailly, J.-l.; Adler, M. Usage Example. [online], Dec 2005, [Accessed 2023-03-16]. Available from: `https://www.zlib.net/zlib_how.html`

[33] Microsoft. Virtual Address Space (Memory Management). [online], Jul 2021, [Accessed 2023-03-26]. Available from: `https://learn.microsoft.com/en-us/windows/win32/memory/virtual-address-space`

[34] The Linux Kernel. Address space options for 32bit systems. [online], [Accessed 2023-03-26]. Available from: `https://linux-kernel-labs.github.io/refs/heads/master/lectures/address-space.html#address-space-options-for-32bit-systems`

[35] glibc wiki. MallocInternals. [online], Aug 2022, [Accessed 2023-03-27]. Available from: `https://sourceware.org/glibc/wiki/MallocInternals`

[36] Microsoft. Page State. [online], Jul 2021, [Accessed 2023-03-28]. Available from: https://learn.microsoft.com/en-us/windows/win32/memory/page-state

[37] Microsoft. VirtualAlloc function (memoryapi.h). [online], Jul 2022, [Accessed 2023-04-01]. Available from: https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc

[38] Microsoft. Why is address space allocation granularity 64KB? [online], Oct 2003, [Accessed 2022-04-02]. Available from: https://devblogs.microsoft.com/oldnewthing/20031008-00/?p=42223

[39] Microsoft. Randomizing the KUSER_SHARED_DATA Structure on Windows. [online], Apr 2022, [Accessed 2023-04-01]. Available from: https://msrc.microsoft.com/blog/2022/04/randomizing-the-kuser_shared_data-structure-on-windows/

[40] Adler, M. Can gzip only store a file like zip and not compress it? [online], Mar 2021, [Accessed 2022-04-02]. Available from: https://stackoverflow.com/a/42934744

[41] Deutsch, P. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996, doi:10.17487/RFC1951. Available from: https://www.rfc-editor.org/info/rfc1951

Appendix A

Acronyms

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
CISA	Cybersecurity and Infrastructure Security Agency
CNA	CVE Numbering Authority
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
CWSS	Common Weakness Scoring System
DEP	Data Execution Prevention
DHS	Department of Homeland Security
FIRST	Forum of Incident Response and Security Teams
LIFO	Last In First Out
NIST	National Institute of Standards and Technology
NVD	National Vulnerability Database
RFC	Request For Comments
ROP	Return Oriented Programming
TLS	Transport Layer Security

Appendix B

Contents of attached files

```
README.pdf ............................... the PDF file with description
impl ............................ the directory with the implementation
├── poc ........... the directory with the proof-of-concept demonstrations
└── virt ............ the directory with the prepared virtual environment
thesis .................................. the directory with the thesis
├── BP_Krejsa_Vojtech_2023.pdf ......... the thesis text in PDF format
└── src ................. the directory of LaTeX source codes of the thesis
```