FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Analysis of the CVE-2023-4863 vulnerability in libwebp
Student:	Pavel Holý
Supervisor:	Ing. Josef Kokeš, Ph.D.
Study program:	Informatics
Branch / specialization:	Information Security 2021
Department:	Department of Information Security
Validity:	until the end of summer semester 2024/2025

Instructions

1) Research the CVE-2023-4863 vulnerability in the libwebp library.
2) Describe the general concept of the vulnerability: What is the vulnerable component, how and when does the vulnerability manifest, which applications are vulnerable, which are known fixes and workarounds, etc.
3) Study the vulnerability in detail, with a particular focus on the possibility of a remote code execution through a malformed picture: Since the vulnerability takes a form of an out-of-bounds write, which memory areas can be affected? What needs to be an application's structure to make it susceptible to the RCE attack? Which general and specific protective measures need to be disabled to make the attack possible?
4) If possible, prepare a proof-of-concept of an application that can be exploited in this fashion.
5) Discuss your results.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 30 January 2024 in Prague.

Bachelor's thesis

ANALYSIS OF THE CVE-2023-4863 VULNERABILITY IN LIBWEBP

Pavel Holý

Faculty of Information Technology
Department of Information Security
Supervisor: Ing. Josef Kokeš, Ph.D.
May 15, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Holý Pavel. *Analysis of the CVE-2023-4863 vulnerability in libwebp.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

List of abbreviations ix

Introduction 1

1 Buffer overflow 2
- 1.1 Process's memory layout . . . 2
- 1.2 Buffer overflow on the stack . . . 4
  - 1.2.1 User stack's layout . . . 4
  - 1.2.2 Exploiting buffer overflow on the stack . . . 6
- 1.3 Buffer overflow on the heap . . . 7
  - 1.3.1 Heap's structure in glibc . . . 7
  - 1.3.2 Exploiting buffer overflow on the heap . . . 9
- 1.4 Protection mechanisms . . . 10

2 The libwebp library 12
- 2.1 CVE-2023-4863 . . . 12
  - 2.1.1 CVE Program . . . 12
  - 2.1.2 The vulnerability . . . 13
- 2.2 WebP image format . . . 14
  - 2.2.1 WebP header . . . 14
  - 2.2.2 Lossless bitstream . . . 15
- 2.3 Huffman codes . . . 16
  - 2.3.1 Huffman trees . . . 17
  - 2.3.2 Huffman coding using tables . . . 18
  - 2.3.3 Canonical representation of Huffman codes . . . 19
  - 2.3.4 Huffman codes in libwebp . . . 21
- 2.4 The implementation . . . 21
  - 2.4.1 `ReadHuffmanCodes` . . . 22
  - 2.4.2 `ReadHuffmanCode` . . . 23
  - 2.4.3 `ReadHuffmanCodeLengths` . . . 23
  - 2.4.4 `BuildHuffmanTable` . . . 24

3 Analyzing the vulnerable component 29
- 3.1 Overflow reach . . . 30
  - 3.1.1 Complete second-level trees . . . 30
  - 3.1.2 Incomplete second-level tree . . . 34
  - 3.1.3 Calculating the reach . . . 36
- 3.2 Overflow value . . . 38

4 Exploiting the vulnerable component **40**
4.1 Planning the exploit 40
4.2 Arranging the heap 42
4.3 Creating the malicious image 43
4.4 Creating the malicious file 45

Conclusion **46**

Bibliography **47**

Contents of attached files **51**

List of Figures

1.1 Virtual address space of a 64bit Linux process 4
1.2 User stack in a 64bit Linux process 6
1.3 Structure of a used chunk 8
1.4 Dynamic memory allocation 9

2.1 WebP simple file format containing lossless bitstream 15
2.2 Lossless bitstream structure 17
2.3 Two possible Huffman trees, decoding the word *assessment* 18
2.4 Two-level tables implementation, 4-bit root table 20

3.1 Comparison of arbitrary and optimal second-level trees 31
3.2 Constant gain demonstrated on two pairs of second-level trees 32
3.3 Positions where leaves at layer 7 are placed 35
3.4 Four bytes of `HuffmanCode` corresponding to a leaf in the distance tree 39

4.1 Arranging the heap 44

List of Tables

2.1 Release dates of web browser versions containing the fix 14
2.2 Sizes of the different Huffman trees in libwebp 21

3.1 Positions where leaves at layer 7 are placed, and the reach increases 34
3.2 Best reach and number of used leaves of an incomplete tree for each of the layers 36
3.3 Analysis of table sizes of the first four tables required to achieve buffer overflow . 37
3.4 Number of leaves in layers leading to maximum complete table sizes 38
3.5 Maximum overflows for each transition layer, U. L. stands for unpaired leaf . . . 38

4.1 Canonical representation of the tree that enables the exploit 42

List of code listings

1.1 Code susceptible to buffer overflow . . . 3
1.2 Dynamic memory allocation . . . 8

2.1 Structure `HuffmanCode` . . . 22
2.2 Function `ReadHuffmanCodes` . . . 22
2.3 Function `ReadHuffmanCode` . . . 24
2.4 Function `ReadHuffmanCodeLengths` . . . 25
2.5 Helper functions for `BuildHuffmanTable` . . . 26
2.6 Function `BuildHuffmanTable` . . . 27

3.1 Building optimal trees . . . 33
3.2 Building incomplete tree . . . 36

4.1 Function calls affecting the heap during the decoding of our WebP image . . . 42
4.2 Arranging the heap; `mmap` and function pointer wrappers . . . 43

I especially want to thank my supervisor, Ing. Josef Kokeš, Ph.D., for his invaluable guidance, support, and feedback throughout the duration of this research. Next, I extend my gratitude to my parents and friends for their unwavering support. Lastly, I want to acknowledge Bc. Vojtěch Krejsa, whose thesis served as a source of inspiration.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 15, 2024

Abstract

This thesis provides an analysis of the CVE-2023-4863 heap buffer overflow vulnerability in the libwebp library, a widely used library for working with the WebP image format. As the overflow occurs in the lossless image decompression based on the Huffman tree coding, the thesis describes the possibilities of Huffman codes that lead to buffer overflow. Next, the possibility of leveraging the vulnerability for malicious intents is explored. As a result, the thesis contains a Linux proof-of-concept application, which incorporates the vulnerable libwebp component and can be exploited to remote code execution. The existence of such exploitable application emphasizes the importance of the vulnerability.

Keywords CVE-2023-4863 vulnerability, libwebp, Huffman code, remote code execution, heap buffer overflow, exploit, Linux

Abstrakt

Tato práce analyzuje zranitelnost CVE-2023-4863, která se týká přetečení bufferu na haldě v knihovně libwebp, což je hojně používaná knihovna pro práci s obrázky ve formátu WebP. Protože se toto přetečení bufferu projeví při dekompresi bezztrátového obrázku, který je založený na Huffmanově kódování, tak tato práce popisuje možné Huffmanovy kódy, které způsobí přetečení bufferu. Dále prozkoumává možnost zneužití této zranitelnosti pro škodlivé úmysly. Jako výstup tato práce obsahuje Linuxovou proof-of-concept aplikaci, která používá zranitelnou komponentu z libwebp a může být zneužita pro spuštění útočníkova kódu. Existence takové zneužitelné aplikace zdůrazňuje důležitost této zranitelnosti.

Klíčová slova zranitelnost CVE-2023-4863, libwebp, Huffmanův kód, vzdálené spuštění kódu, přetečení bufferu na haldě, exploit, Linux

List of abbreviations

API	Application Programming Interface
ASLR	Address Space Layout Randomization
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
LIFO	Last In First Out
LSB	Least Significant Bit
MSB	Most Significant Bit
OOB	Out Of Bounds
OS	Operating System
RIFF	Resource Interchange File Format

Introduction

We live in a time where technology is used in more areas than ever before. With it, the number of bad actors increases as well. These actors often leverage software vulnerabilities, one of them being buffer overflow. If exploited successfully, this vulnerability can lead to denial of service attacks or even execution of the attacker's code. Many modern programming languages significantly reduce the possibility of a buffer overflow. However, there are still a lot of programs written in unsafe languages, which means they might be susceptible to a buffer overflow, so this topic is very relevant even today.

We believe it is crucial to minimize the risks of cyberattacks. One of the ways of doing so is by raising awareness of the impact a vulnerable component can have. This can be achieved by exploring publicly disclosed vulnerabilities and explaining them in greater detail than has been done before.

A buffer overflow vulnerability known under the identifier CVE-2023-4863 in the libwebp library was published in September 2023 [1]. Libwebp is a widely used library that works with images in the WebP format. It is used by many programs, from image manipulation tools to commonly used web browsers. There are rumors that its vulnerability can even be exploited to code execution, although there is no proof-of-concept of such exploitation publicly available.

The aim of this thesis is to analyze the CVE-2023-4863 vulnerability. We will identify the vulnerable component and describe the conditions required for this vulnerability to manifest. As information about the vulnerability is limited, our main goal is to describe the causes of this buffer overflow in greater detail. Finally, we will explore the possibility of achieving remote code execution and, if successful, prepare an exploitable proof-of-concept application. We do not plan to attempt to exploit an already existing application.

The content of the chapters is as follows:

Chapter 1 In this chapter, we provide a theoretical background of buffer overflows, specifically a buffer overflow on the stack and on the heap. We also summarize various protection mechanisms that make exploiting of a buffer overflow more difficult.

Chapter 2 In this chapter, we first summarize the available information about the CVE-2023-4863 vulnerability. Next, we look at the WebP file format, with a focus on the parts relevant to the vulnerability. As the format uses the Huffman coding, we describe the coding as well. Finally, we analyze the code from the libwebp library that is responsible for the overflow and put it into the context of the file format.

Chapter 3 As the process behind creating data that cause overflow is not trivial, we dedicate this chapter to the analysis of what data can be overflown and where.

Chapter 4 In this chapter, we utilize the observations made in the previous chapter and create an application where the CVE-2023-4863 vulnerability can be exploited to code execution.

Chapter 1
Buffer overflow

Buffer is an area of memory that is used to temporarily store data. The underlying memory has a fixed size, whether it was allocated statically or dynamically. Many programming languages have built abstractions on top of this low-level memory access, and these data structures often implement boundary checks. If such a check is implemented and the programmer tries to access the memory at the position before the start or after the end of the buffer, no memory access will happen and an exception can be raised instead. These languages usually implement other memory safety features as well, so they are often referred to as *memory safe languages*. An example of a data structure with this behavior could be `list` in the Python programming language, which raises the `IndexError` exception upon trying to access data at an invalid offset. [2, 3, 4]

On the other hand, there are also languages that do not perform any checks and simply let you read or write directly the memory at the given index. Such languages are still widely used as well as omitting those checks usually leads to less code having to be executed on each memory access, which in turn improves execution time. If, for example, an application performs a lot of memory operations and time efficiency is required, it might be a valid reason to use such a language. It is then the programmer's responsibility to correctly implement checks that ensure the accessed memory is within the bounds of the buffer. Commonly used languages in this category are C and C++[1]. [5]

Buffer overflow, which is a specific type of *OOB* (*out of bounds*) write, occurs whenever the program writes to the memory at a position outside the buffer's memory area. As other data can reside next to the buffer, buffer overflow can be utilized to alter this data. Depending on what data has been overwritten, this can have various impacts, from altering a variable's value to executing arbitrary code. Creating a program susceptible to buffer overflow is trivial in memory unsafe languages and very difficult in memory safe languages for the reasons described above (although not impossible, as can be seen in [6]). A simple example of a C code susceptible to buffer overflow is shown in Code listing 1.1. [7, 4]

1.1 Process's memory layout

Before describing buffer overflow in specific areas of memory, we will introduce them first. As the concept of virtual memory influences the process's memory layout on modern machines, we start with its description. Unless stated otherwise, everything in this thesis is described in the context of x86_64 Linux. [8]

[1] Technically, the C and C++ specification marks out of bounds access as an undefined behavior, meaning there is no guarantee for how will the compiled binary behave. However, in practice, it usually accesses the memory in the same way as a legitimate in bounds access would.

Code listing 1.1 Code susceptible to buffer overflow

```
int n
int arr[10];
scanf("%d", &n);
for(int i = 0; i < n; i++)
  scanf("%d", &n[i]); // overflow occurs if n > 10
```

Virtual memory is an abstraction over the physical memory. It provides each process with the illusion that it has exclusive access to the main memory. This view of the memory is called the *virtual address space*. These virtual addresses are then being translated to physical addresses, per process. The translation is managed by the OS (operating system) (with support from hardware), making it completely transparent to the process. This mechanism has at least two significant advantages. First, as the virtual address space is not occupied by other processes, the process can place various memory areas to the same addresses between runs. Second, the process cannot directly access the memory of other processes simply because those regions are not mapped to its virtual address space in the first place, which improves security. [8]

As described in [9], the virtual address space is divided into three parts:

User space virtual memory The first part, located at the bottom of the address space, is the actual memory where the process's variables, functions, constants, etc., are stored. This is also the only part out of the three that the process can access.

Non-canonical virtual memory The middle part is not currently utilized and is reserved for future use. It is orders of magnitude larger than the other two parts, occupying almost all of the address space[2].

Kernel space virtual memory The *kernel* is the part of the OS that is always present in the memory. The top of the address space is reserved for it.

As summarized in [8], we now focus on the different areas of memory that make up the user space virtual memory; we show the layout in Figure 1.1. The areas are as follows:

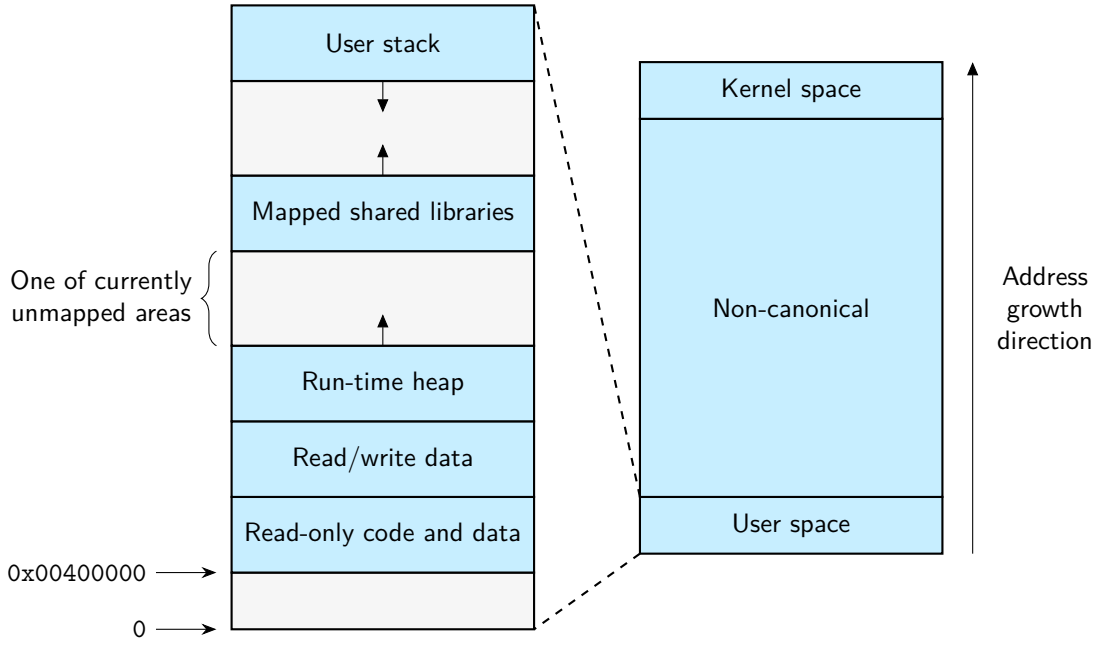
Code and data The program's code is located near the bottom of the address space and always starts at the address `0x00400000`. It is followed by data corresponding to the global and static C variables. This memory area is initialized directly from the content of the executable file.

Heap Heap follows immediately after, and any memory that is dynamically allocated during the run-time of an application is stored in it. Unlike the code and data areas, the size of the heap is not fixed and can expand and contract dynamically, depending on the amount of storage the application requires from the heap. Heaps will be explored in greater detail in section 1.3.

Shared libraries Programs usually rely on some commonly used, already existing functions. If the whole code of such functions was always included in every compiled binary that uses it, both the size of the binary and, subsequently, the amount of occupied memory required to store these functions during run-time would increase. This problem is addressed by the shared libraries. If multiple programs dynamically link the same shared library and are run afterward, the shared library is loaded to memory only once, which fixes the mentioned problem. It is through this memory area that the process can access the content of the shared library.

[2] This is not an issue, as even a small part of the 64bit address space can still be plenty big. When using virtual memory with 4-level page tables, the user space and kernel space virtual memories each occupy 128 TiB, compared to almost 16 EiB of non-canonical virtual memory (1 EiB = 10^6 TiB).

Figure 1.1 Virtual address space of a 64bit Linux process, inspired by [8]



Stack The user *stack* is located at the top of the user's virtual address space. The compiler uses this memory area for implementing function calls and for storing local variables. Similarly to the heap, the user stack's size changes during the execution of the program. In particular, it increases whenever a function is called and decreases when the function completes. More on stacks in section 1.2.

1.2 Buffer overflow on the stack

In general, stack is a data structure that supports two data manipulation operations—push and pop. Push adds a value to the stack. Pop retrieves the most recently pushed value, which has not been popped yet, and removes it from the stack. Data structures with this behavior are called *LIFO* (*Last In First Out*). [10]

Function calls usually require storing information, such as their local variables. On top of that, any function can call other functions. Whenever such nested calling of functions takes place, the most deeply nested function is the one that finishes its execution first. This resembles the LIFO functionality. For this reason, function calls are implemented using a stack in the user stack memory area. The part of the user stack corresponding to a particular function call is referred to as its *stack frame*. The user stack grows to lower addresses, as we have shown in Figure 1.1. [10]

1.2.1 User stack's layout

Before we focus on the user stack's layout in detail, we will describe three important registers, as outlined in [8]:

Stack pointer Stored in register `%rsp`[3], this value points to the top of the stack. More specifically, it holds the memory address from which the value will be retrieved when we call the pop instruction.

Frame pointer Stored in register `%rbp`, this value points to the beginning of the most recently allocated stack frame, i.e., the stack frame of the currently executing function. The reason for keeping such a value is that the content of the stack frame can be addressed as a relative offset from the frame pointer. Although compilers often use the `%rbp` for storing the frame pointer, it is a general-purpose register and, as such, can be theoretically used for other purposes as well [10].

Instruction pointer Stored in register `%rip`, this value contains the address of the instruction that is currently being executed. After the instruction is completed, the instruction pointer is automatically incremented to point to the next instruction. [2]

When a function is called by executing the `call` instruction, the natural flow of the program is disrupted, as we want to execute the instructions of the called function (sometimes referred to as the *callee*) instead of the instruction following the `call`. To achieve this, the `call` instruction changes the instruction pointer to point to the first instruction of the callee. The frame pointer must be updated as well because we are entering a new function which has a new stack frame. When the callee finishes and wants to return to the *caller*, we must restore the registers to their original states. This means the instruction pointer should point to the next instruction after the `call`, and the frame pointer should point to the stack frame of the caller. For these values to be restored, we must store them before overwriting them with values related to the callee. As we have to do this backup once for each function call, it is convenient to push these values to the stack as well. The address of the next address after the `call`, which is called the *return address*, is pushed to the stack automatically by the `call` instruction and is later popped by the `ret` instruction, which finishes the execution of the callee. The frame pointer is stored in the `%rbp`. Because this is a general-purpose register, it cannot be pushed to the stack by the instruction `call` itself. Instead, it is pushed to the stack manually at the beginning of the callee and popped back to `%rbp` before the `ret` instruction is executed. [8]

Now that we have described everything relevant, we summarize the layout of the stack, which can be seen in Figure 1.2. When a function is called, the caller passes up to six parameters through registers. The remaining ones are pushed to the stack, and the `call` instruction pushes the return address. From now on, we are in the callee's stack frame, where the callee pushes the current value of the frame pointer and allocates space for local storage by subtracting a constant value from the stack pointer[4]. [8]

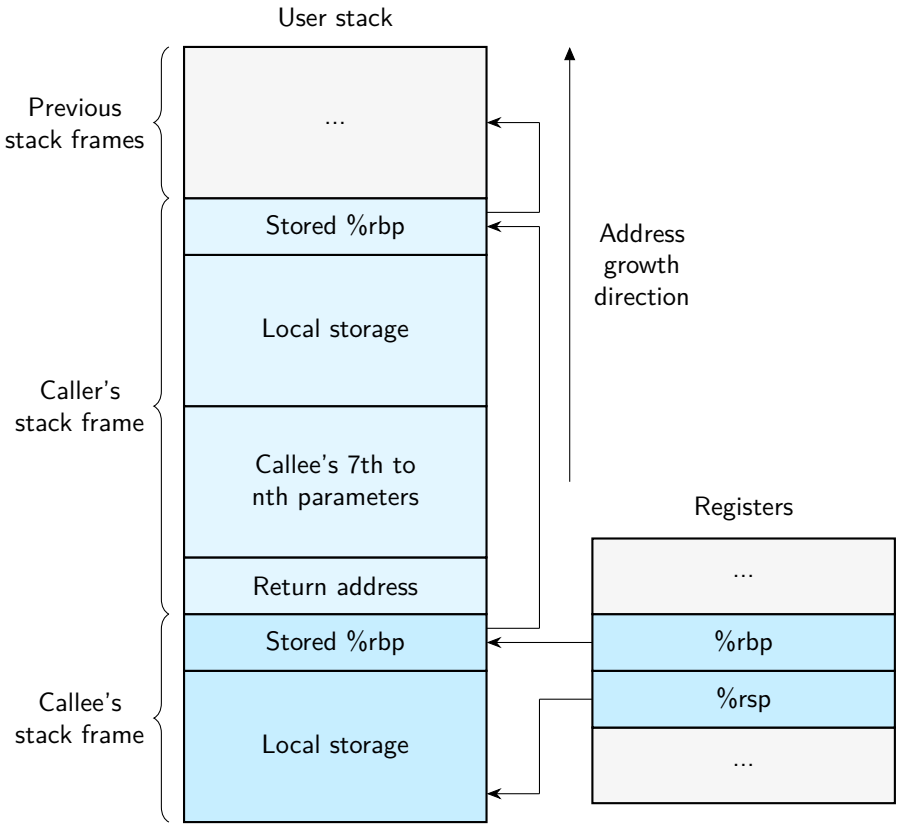
If the function is simple enough, meaning all parameters and local variables can be stored in registers and the function does not call any other function, it is possible that it does not need to allocate any space on stack at all. [8]

The last two things we did not explain regarding function calls are passing return value and storing the content of temporary registers so that their values are not lost when calling a function that also uses them. If possible, the return value is passed through a register. Otherwise, it is stored on the stack in the caller's stack frame. The temporary registers are saved to the stack and restored later. There is, however, a distinction as to who is responsible for doing so. The registers that are saved and later restored by the caller are called *caller-saved*, and those that are saved and restored by the callee are *callee-saved*. On x86_64, the callee-saved registers are `%rbx`, `%rbp`, and `%r12` to `%r15`. The caller-saved registers are `%r10` and `%r11`. [8]

[3] As mentioned previously, this applies to the x86_64 instruction set architecture—other instruction set architectures may differ.

[4] We subtract because the stack grows to lower addresses.

Figure 1.2 User stack in a 64bit Linux process



1.2.2 Exploiting buffer overflow on the stack

Now that we understand the stack's layout, we can focus on exploiting buffer overflow on the stack. When a buffer is located on the stack, it is placed in the local storage of the current function's stack frame. Because the stack grows to lower addresses, when we overflow beyond the end of the buffer, we are accessing memory on higher addresses, which means we go deeper into the stack. [10]

The most interesting value we can overwrite is the stored return address because this value determines what instructions will execute after we return from the function. By changing it, we force the execution of some other code. As to what code we can execute, we have several options, as presented in [10]:

Attacker's code If executed successfully, this variant provides us with the fewest limitations regarding the code we can and cannot inject. We need to place data containing the binary values of the instructions we wish to execute somewhere in the process's memory, usually in some buffer. This can be the buffer we are currently overflowing or some other buffer (which does not have to be susceptible to buffer overflow at all). Finally, we overwrite the return address on the stack with the address of our stored code.

Function in a shared library For various reasons, the first method is not always achievable. We can instead execute an already existing function from a shared library by changing the return address to the function's address. The most promising candidate we can use is the libc library, as it is almost always available and provides useful functions, e.g., the `system` function, which we can use to execute our command [11].

Function in the binary While this method is less flexible than the other two, it can still be useful. For instance, if a function is executed only after we are successfully authenticated, we can launch this function directly by setting the return address to the address of this function, resulting in us bypassing the authentication.

In practice, several security protection mechanisms are enabled, which we often have to overcome to achieve code execution. We provide an overview of these mechanisms in section 1.4.

Even if we cannot achieve code execution, we can crash the application. This can be done by setting the return address to some invalid value, for example, to all zeroes. When the currently executing function finishes, the CPU (central processing unit) tries to dereference the null pointer, which causes a segmentation fault. Unless the application explicitly catches the segmentation fault signal, this signal results in the process being terminated. [10, 12]

While the return address is the most interesting value we can target on the stack, it is not the only one. For example, other local variables are also stored on the stack. If the compiler has placed them on higher addresses than the buffer, we can target their values. However, this approach depends entirely on the meaning of these local variables. [8]

1.3 Buffer overflow on the heap

The heap is another area of memory that we can use to store data. Allocating data in the heap is referred to as *dynamic memory allocation*. As described in [8] and [10], we want to use the heap instead of the stack in the following scenarios:

- If we require a large amount of memory, the heap is a better option than the stack as the stack is generally smaller.
- If we do not know during compile-time the amount of memory we will require.
- If we need to use the memory even after the function that has allocated it returns.

Whenever we want to store something in the heap, we do not simply choose an address in the heap memory area and start writing there. This is because the heap itself is managed by the *dynamic memory allocator* (we will refer to it as the *allocator*). When we need to allocate memory in C, we make a request by calling `malloc`. The `malloc` function takes one parameter—the amount of memory we require, in bytes. This request is then processed by the allocator. It finds an appropriate place in the heap with enough space to satisfy our request. Finally, a pointer to this area is returned to us from the `malloc` call. From now on, we can access the allocated memory for as long as we like, as it will not be automatically unallocated (unlike the memory on the stack). When we no longer need the memory, we should call the `free`[5] function and pass it the pointer previously obtained from `malloc`. By doing this, we inform the allocator that the memory is no longer required, which enables the allocator to assign it to us again in future `malloc` calls. [8]

1.3.1 Heap's structure in glibc

The structure of the heap entirely depends on the implementation of the allocator. We will focus on the implementation provided by the GNU C Library, or glibc for short. We chose this implementation as it is used in all major Linux distributions. [8, 13]

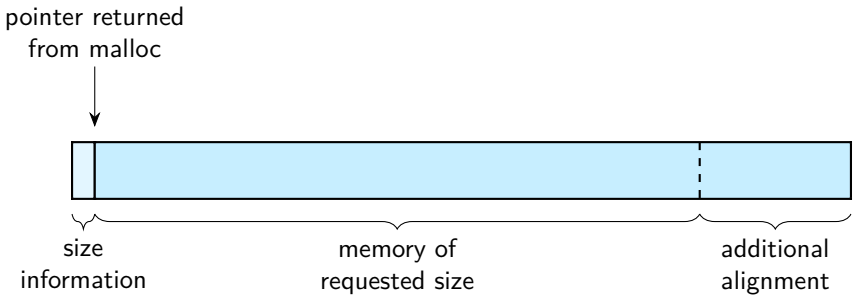
The heap is divided into blocks of memory called *chunks*. The last chunk, called the *top chunk*, is typically the biggest one. According to [14], each other chunk is in one of two states:

[5] The C++ equivalents to `malloc` and `free` are `new` and `delete`.

Used This chunk corresponds to memory allocated by the program (for example, by `malloc`). The chunk's size is stored exactly at the beginning of the chunk. As the size is always aligned to multiples of eight, its three lower bits are used for additional information. Immediately after, the area of memory returned by `malloc` starts. Its size is at least equal to the amount the application requested, although it might be larger because of the allocator's size alignment requirements. We show this structure in Figure 1.3.

Free This chunk is not currently allocated. Because the allocator keeps a list of all free chunks, the memory of this chunk is re-purposed for storing additional allocator information, for instance, pointers to other free chunks.

Figure 1.3 Structure of a used chunk



When we request memory, the allocator will either reuse an already existing free chunk (to reduce fragmentation) or create a new chunk at the end by splitting the top chunk (this is a major oversimplification; a full description can be found in [14]). We demonstrate this behavior in Figure 1.4 and in Code listing 1.2. We refer to the chunks with capitalized variants of the pointer variable names. Assuming we have no free chunks, `A`, `B`, and `C` are placed immediately one after the other. Then the chunk `B` is switched to the free state. The chunk `D` is too large to fit in the free chunk, so it is placed at the end as well. When we create chunk `E`, we see that the allocator behaves differently for different magnitudes of allocated sizes. In our example, we test three different magnitudes, controlled via the `n` variable. We present the observed behavior:

n = 1 `E` is placed in the free chunk, occupying the whole chunk.

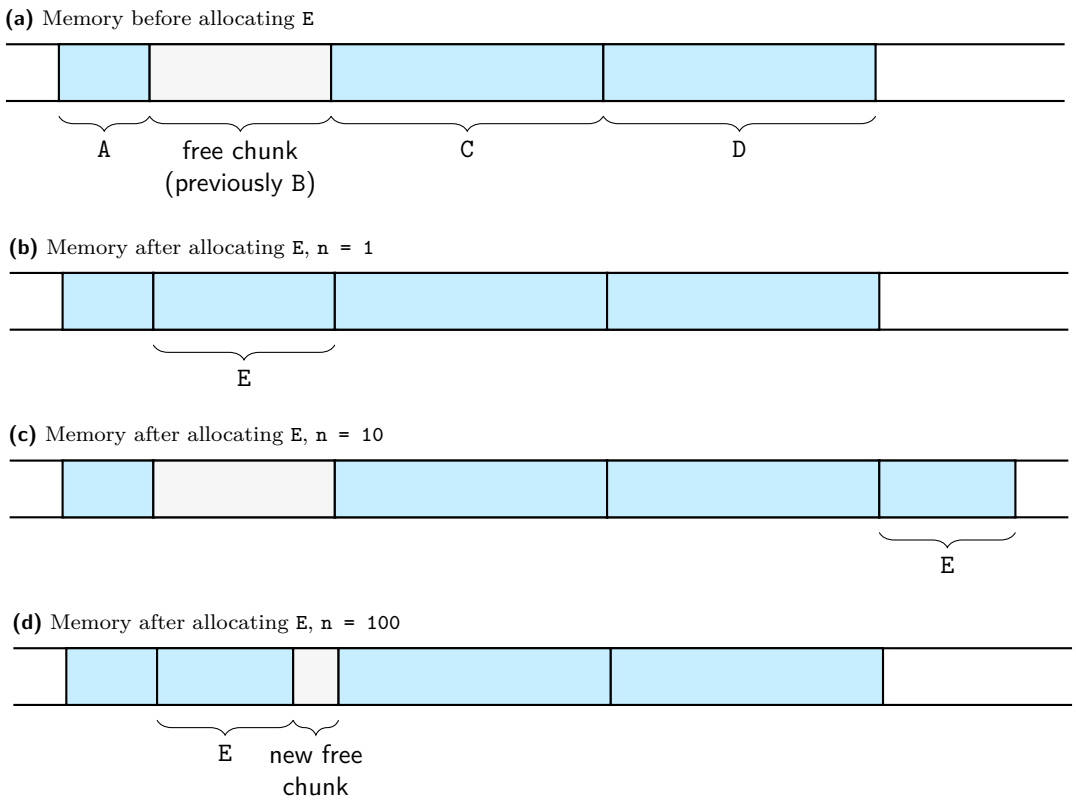
n = 10 `E` gets a new chunk, placed at the end after `C`. Free chunk remains intact.

n = 100 `E` is placed in the free chunk, which is split first, creating a new free chunk between `E` and `C`. This allows future allocations to be placed there.

Code listing 1.2 Dynamic memory allocation

```
int n = 1; // 10, 100
void* a = malloc(10 * n);
void* b = malloc(20 * n);
void* c = malloc(30 * n);
free(b);
void* d = malloc(30 * n);
void* e = malloc(15 * n);
```

Figure 1.4 Dynamic memory allocation



Allocator's data structures used for storing free chunks are called *bins*. When a chunk is freed, it is assigned to a selected bin, based on its size. Bins of different sizes are treated differently. This is why the example behaved differently for various values of `n`. [14]

For the sake of completeness, the glibc allocator implementation also supports multiple heaps, each of which belongs to an *arena*. As this is not relevant to this thesis, we will not discuss it further. Detailed descriptions of bins, heaps, and arenas can be found in [14].

1.3.2 Exploiting buffer overflow on the heap

Whenever we have a buffer on the heap, it is located in a used chunk. If we overflow, we overwrite information located after the end of the buffer. As we have shown in Figure 1.3, the only other part of the used chunk located after the buffer itself is padding, which does not contain any information. This implies we must focus on altering the content of the following chunks instead. Two types of information are located in chunks—allocated memory owned by the user application, and allocator's metadata stored at the start of a used chunk and in free chunks. [14, 10]

Our first option is to overwrite data in the allocated blocks of memory. This approach depends entirely on the meaning of the information that is stored there. If, for instance, a function pointer was located there, we could even succeed in code execution. Options as to what value we can set this pointer to are described in section 1.2.2. Nonetheless, the possibilities must be examined individually in the context of a specific application, so we will not go into further details in this section. [14]

The second option is to alter the allocator's metadata. As this depends entirely on the allocator's implementation, the exploitation methods will vary between implementations of the standard C Library. An example of one such way to take advantage of the allocator's metadata is altering it and utilizing the old implementation of the unlink macro (now fixed). More information can be found in [15].

If we want to overflow to another specific chunk, we must find out where it is located relative to the chunk containing the vulnerable buffer. Unlike on the stack, determining the relative offset of two chunks is not trivial. As we have discussed previously, the exact positions where the chunks will be placed are influenced by numerous factors, one of which is the set of currently free chunks. For example, let us have a function where we make two allocations right after each other, and before we return, we free both of them. Invoking this function multiple times throughout the application will (in general) yield different relative offsets between the allocated blocks, as other heap operations may have happened in between the calls that influenced the heap's state. Furthermore, we must be careful not to corrupt the allocator's metadata, which may be placed between the susceptible and the target chunks. When the allocator detects this corruption, it will likely terminate the program. [14]

On the other hand, the allocator (in glibc) has a fixed algorithm for determining where to put the next chunk. This means that if an application executes the same sequence of `malloc` and `free` calls each time it is launched, the relative offsets between individual allocated blocks across multiple application runs will remain constant. [14]

1.4 Protection mechanisms

Multiple protection mechanisms aimed at reducing the possibility of buffer overflow exploitation have been created over the years. The mechanisms we present fall into one of two groups—those done entirely in software and those utilizing some feature of the OS. These mechanisms serve only as an additional layer of security. By no means do they completely eliminate the possibility of buffer overflow exploitation, and we should treat them as such.

We provide an overview of a few selected protection mechanisms, as they are described in [10]:

NX bit NX stands for *Non-eXecutable*. This bit defines whether code can be executed in various regions of memory. The idea is to have parts of memory writable or executable, but not both simultaneously (this concept is also referred to as $W \hat{} X$). The stack, the heap, and the memory area storing global variables are writable, but not executable. The memory areas containing code are executable, but not writable. This mechanism is implemented in the OS with support from hardware. Even if we are able to modify the instruction pointer and inject our code somewhere in a buffer, executing this code will be more difficult as the part of the memory with the injected code will not be executable. However, if we want to execute an already existing code instead, this protection has no effect as this code is in an executable area of memory.

ASLR *Address Space Layout Randomization*. This feature randomizes the addresses the different memory areas are placed to, and is managed by the OS. It counts on the fact that we cannot determine the addresses of the code we want to execute. While randomization of every memory area is possible, it is often not implemented in that way as it can reduce performance. For instance, on Linux, the stack, the heap, and the shared libraries (when they are loaded to the memory) are randomized, but the code and data corresponding to global variables are not (unless we compiled the binary as a *relocatable object*). This means that if we execute one of the binary's functions, we will find it consistently on the same address as ASLR does not randomize its address. ASLR is enabled by default, and we can configure it by writing to `/proc/sys/kernel/randomize_va_space`, as described in [16].

Stack canaries The name *canaries* comes from mines. When a miner went down a mine, they brought a bird with them. When air was running out, the bird, being more sensible, died first, alerting the miner. Stack canaries have a similar purpose. In general, they take the form of some additional value placed on the stack somewhere between buffers and sensitive information (e.g., the return address). When we overflow a buffer on the stack, we cannot overwrite the sensitive information without overwriting the canary. By comparing the canary's current and original values, the code can discover an illegal write operation and terminate the application because the canary could not have been changed otherwise. This protection mechanism does not require any support from the OS because it only needs to add additional code for storing and later checking the canary. One such implementation, available in the GCC compiler, can be turned on with the `-fstack-protector` option. As with the other mechanisms, stack canaries do not completely eliminate the threat of buffer overflow exploitation. For example, if another vulnerable component allows us to read the canary's value first, we can then overflow, set the canary to the value we have obtained previously, and continue to overwrite the sensitive information. This way, no overflow happened from the code's point of view.

Chapter 2
The libwebp library

WebP is an image file format created by Google, whose main purpose is to be used on the web. It aims to provide superior compression (both lossy and lossless) compared to, for example, PNG or JPEG. This means one can preserve image quality while decreasing file size, which could be interesting as transmitting less data can improve the load times of websites. WebP has been adopted by most major web browsers (such as Google Chrome, Firefox, Opera, Edge and Safari). However, WebP is being used by other programs as well, for instance, Photoshop, Gimp and Blender. [17, 18, 19, 20, 21]

The WebP specification is open source, meaning anyone can implement their own library for WebP image manipulation. The most widely used implementation, created by Google itself, is called libwebp. It is this library that has been found vulnerable to heap buffer overflow. This vulnerability has been given the CVE-2023-4863 identifier. [17, 22]

In the following sections, we summarize available information about this vulnerability. Next, we follow up with a description of the WebP format and an overview of Huffman codes. We finish with an analysis of the libwebp code relevant to the buffer overflow.

2.1 CVE-2023-4863

Before we explore the CVE-2023-4863, we will briefly describe the *CVE Program*, as it is responsible for us calling this vulnerability by such name.

2.1.1 CVE Program

The abbreviation CVE stands for *Common Vulnerabilities and Exposures*. The purpose of the CVE Program is to identify, define and catalog cybersecurity vulnerabilities after they were publicly disclosed. [23, 24]

Each vulnerability tracked in the CVE Program is given its own unique entry in the catalog—the *CVE Record*. This record contains structured data about the vulnerability and a unique *CVE ID* assigned to it, whose format is CVE-YYYY-NNNN (YYYY is the year the record was created, and NNNN is an arbitrary number ensuring the CVE ID is unique). By using the identifier, we can avoid possible confusion when referring to vulnerabilities. [24, 23, 25]

Another thing we might want to do is to describe how severe a vulnerability is. Multiple severity rating systems have been created to make the severity rating meaningful and comparable between vulnerabilities. One such widely used system is *CVSS* (Common Vulnerability Scoring System). While the system exists in multiple versions, the version we come across the most is 3.1. This system takes in various information (how complex would an attack exploiting the

vulnerability be, what privileges would be needed, whether user interaction would be required, …) and turns it into a single number between 0 and 10—0 being no threat, 10 describing critical severity. By assigning the CVSS ratings to CVE Records, we can not only be certain we are all referring to the same vulnerability (thanks to the CVE ID) but also compare their CVSS ratings and determine which vulnerability requires immediate attention and which vulnerability "can wait." [26]

2.1.2 The vulnerability

Early in September 2023, the Citizen Lab, a laboratory at the University of Toronto, discovered an actively exploited vulnerability on an iPhone. They promptly sent their findings to Apple, which issued two CVEs[1] related to this issue. [27, 28]

Upon further investigation done by Apple, they sent a message to Google describing a vulnerability in the libwebp library. The next day, a commit providing fix to the issue was pushed to the libwebp git. A week later, this commit was merged into the libwebp version 1.3.2. Google assigned a CVE record to this vulnerability—the CVE-2023-4863[2]. [29, 30, 31, 1, 32]

As far as we can tell, there are two main sources that provide technical insight on the vulnerability. First, there is the mail from Apple, which contains two attachments. One is a technical report from the Apple team describing the libwebp code responsible for the overflow, the other is a proof-of-concept file that causes the overflow. Second, there is a blog post called *The WebP 0day*, together with another proof-of-concept file and programs for constructing this file and visualizing it. The latter was created by the combined effort of Ben Hawkes[3] and mistymntncop[4]. [29, 33, 34]

The blog post describes the nature of the vulnerability. The vulnerability is in the lossless variant of the WebP image. This compression uses the Huffman coding algorithm, which is based on a tree data structure. In practice, implementing this algorithm as an actual tree is not efficient. Instead, we can use an implementation based on tables. The authors of libwebp have precomputed the maximum possible size of the table that a *legitimate* image might require. The table is then created with this constant size. [33]

The problem lies in this table. It is possible for us to create a malicious file that, when we try to decode it by libwebp, passes all checks that test whether the format is correct, and yet it requires a bigger table than we actually have. Consequently, the code then proceeds to write past the end of the table, which is the cause of the buffer overflow. [33]

NIST has assigned this vulnerability a CVSS severity rating of 8.8, which is a further confirmation of its importance. There are rumors that this vulnerability may have been publicly exploited, although no public exploit exists. It is assumed the vulnerable code has been present for almost ten years, which means it could have been being exploited all this time. [35, 33, 30, 36]

Once the commit fixing the vulnerability was available, its propagation to the programs relying on it began. We have gathered available information regarding the times it took popular web browsers to get fixed; it is available in Table 2.1.

Regarding other mitigations besides the patch in the commit—as far as we know, unless we have completely eliminated the possibility of a WebP image being decompressed on our machine, there was no mitigation available that prevented a malicious WebP image from triggering a buffer overflow or even (potentially) exploiting it.

Although both Apple's mail and the blog post provide invaluable information useful for further analysis of the vulnerability, neither gives detailed instructions on how the various malicious files

[1] CVE-2023-41061 and CVE-2023-41064

[2] One more CVE record was created by Google, the CVE-2023-5129, but it was rejected as it was a duplicate of the CVE-2023-4863.

[3] `https://twitter.com/benhawkes`

[4] `https://twitter.com/mistymntncop`

Table 2.1 Release dates of web browser versions containing the fix [29, 30, 37, 38, 39, 40, 41, 42, 43]

Event	Date
vulnerability reported	September 6, 2023
libwebp fixed	September 7, 2023
Google Chrome fixed	September 11, 2023
Mozilla Firefox fixed	September 12, 2023
Microsoft Edge fixed	September 12, 2023
Brave fixed	September 12, 2023
Opera fixed	September 13, 2023
Tor Browser fixed	September 13, 2023

can be created, what values can be overflown and so on. We aim to provide a detailed description of the subject in this thesis. We will be analyzing the library at the last vulnerable commit, i.e., the commit 7ba44f80f3b94fc0138db159afea770ef06532a0.

2.2 WebP image format

The WebP format is based on the RIFF (*Resource Interchange File Format*) document format. The basic element of a RIFF file is a *chunk*. Each chunk contains three fields. First, the 32 bit code used for chunk identification. It consists of four ASCII characters, each of which occupies 8 bits. Next, the chunk's size in bytes, also stored in 32 bits. The size does not count this field, the identification field, or padding. Finally, the payload. This field holds the data itself, and its size equals to the size specified in the previous field. If the payload's size in bytes is odd, a single padding zero byte is appended. [44]

As the WebP file format does not operate on whole bytes but on individual bits instead, we must clarify the bit ordering. We read bytes in their natural order as they occur in the file. For each byte, we process its bits from the least significant bit (LSB) to the most significant bit (MSB). If a value in the format is stored in more than one bit, we construct the value by repeatedly reading the next bit. The first read bit goes to the LSB of the value and the last read bit to the MSB of the value.

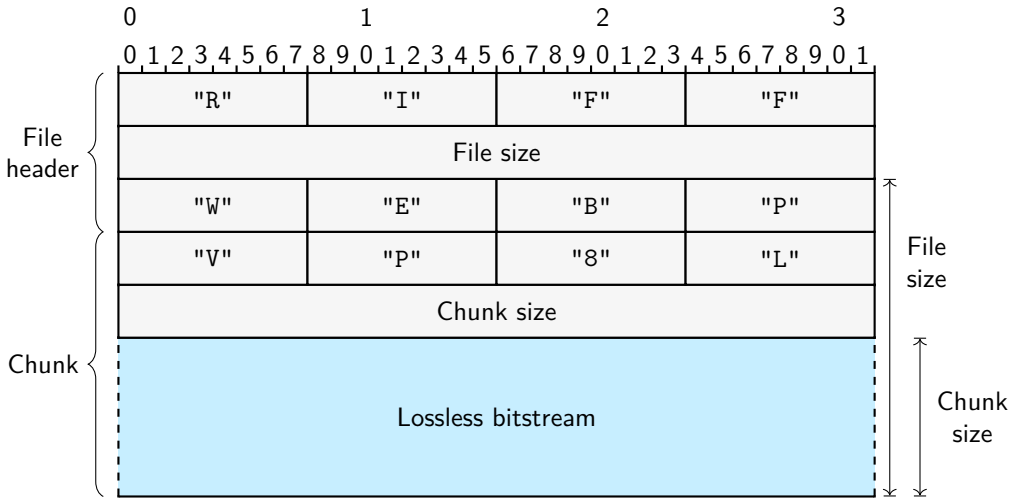
2.2.1 WebP header

All WebP files share the same header. It starts with the `"RIFF"` bytes, followed by the file size in bytes minus 8, and ends with the bytes `"WEBP"`. After the header, various chunks follow. [44]

There are two WebP file formats—*simple file format* and *extended file format*. Both formats contain the image data. In addition, the extended file format can contain optional chunks providing supplementary information, such as color profile, animation control data (WebP can describe animated sequences, similarly to GIF), and Exif and XMP metadata. The image data, also referred to as the *bitstream*, can be either *lossy* or *lossless*. As the vulnerability affects the lossless bitstreams and as it does not require any of the optional chunks from the extended file format, we focus solely on the simple file format containing a lossless bitstream. [44, 33]

In the simple file format, there is only one chunk after the header. It is identified by the `"VP8L"` bytes. The bitstream is stored after the chunk's size in the chunk's payload. We provide an illustration of the file in Figure 2.1.

Figure 2.1 WebP simple file format containing lossless bitstream, inspired by [44]



2.2.2 Lossless bitstream

To achieve compression, i.e., a reduction in the file size, the lossless bitstream implements three methods, as outlined in [45]:

Prefix-coded literals Each pixel consists of four channels—red, green, blue, and alpha (transparency). Each channel can have a value from 0 to 255, meaning it requires 8 bits, so a total of 32 bits per pixel. We can perform a frequency analysis for each of the channels. Based on the frequencies, we can create an encoding where the more frequent the value is, the shorter sequence of bits we assign to it.

LZ77 backward reference If the image contains sequences of pixels that appear multiple times in the image, we can leverage this information for further saving of the file size. This method does not store any information about the pixel. Instead, it stores two values—*length* and *distance*. The length indicates how many pixels should be copied, and the distance describes where to copy the pixels from.

Color cache code This method holds a set of recently used pixels in the image. If the next pixel is already in the cache, we can copy it from there. This way, referring to the recently used pixels can sometimes be more efficient than the other two methods.

We can use all the described methods (or their subset) in one bitstream, which allows for better compression as the individual method's usefulness depends on the specific pixel layout of the image. These methods need to store additional information (e.g., the encoding of the Prefix-coded literals). It might be useful for us to store different encoding information for different parts of the image. The format supports this with *Meta Prefix codes*. However, as it is not relevant to the vulnerability, we will not discuss it further. [45]

Before we compress the image, we may want to apply transforms, which can then lead to better compression. The lossless bitstream supports four transforms, all of which work on a per-pixel basis. The transforms are reversible so that their effects can be undone during decoding. Referencing [45], we provide brief descriptions:

Predictor transform This transform builds on the idea that neighboring pixels are often similar. If we use this transform, the next pixel is predicted from the already decoded pixels, and only the deviation from the actual value is stored.

Color transform This transform decorrelates the red, green, and blue values of each pixel. It keeps the green value, transforms the red value based on the green value, and transforms the blue value based on the green and the red values.

Subtract Green transform This transform subtracts the green value from the red and the blue values. It is redundant, as the same effect can be achieved by the Color transform. However, we might still want to use it, as it does not need to store any additional transform data.

Color Indexing transform We can use this transform if the image does not contain more than 256 unique pixels. In such a case, the unique pixels are stored in an array. The transform then works as follows. The green value stores the index to this array, the red and the blue values are set to 0, and the alpha value is set to 255.

We can use any combination of the transforms, but each transform can be used at most once. [45]

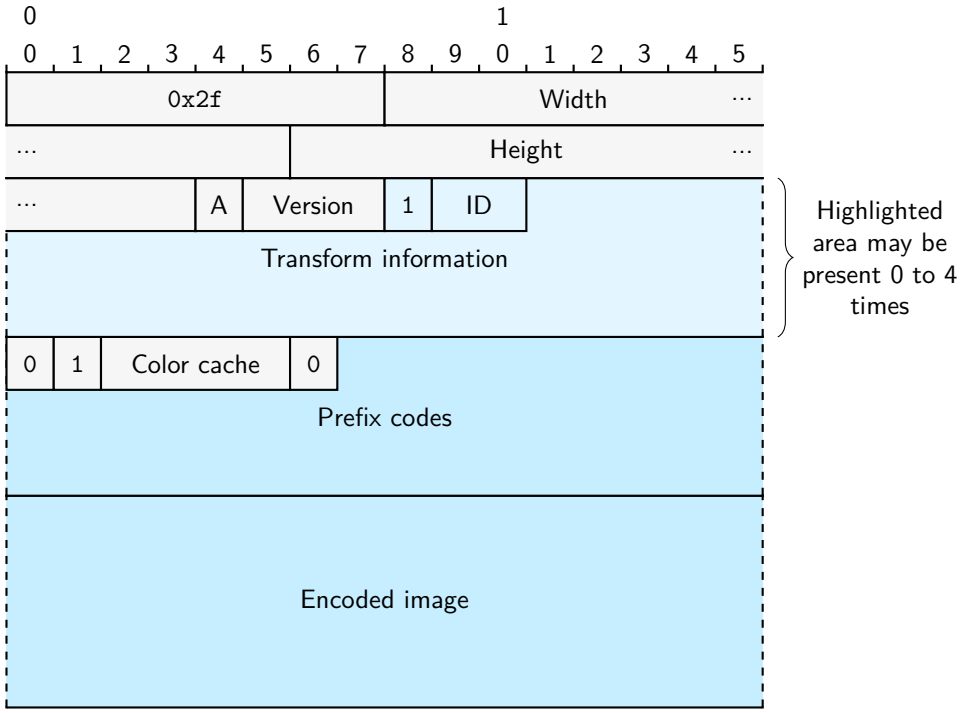
According to [45], we provide a description of the structure of the lossless stream and show it in Figure 2.2:

1. Signature value (8 bits). Contains the value `0x2f`.
2. Image width (14 bits). Width of the image minus one. For example, if this field contains the value 9, the width of the image is 10.
3. Image height (14 bits). Height of the image minus one.
4. Is alpha used (1 bit). This bit serves only as a hint and should not impact decoding. It should be 0 if the alpha value of all pixels is 255 and 1 otherwise.
5. Version (3 bits). Must be equal to 0.
6. Transform (variable size). This field can be present zero or multiple times, each time describing one transform. Each time the field is present, it is indicated by a single 1 bit followed by two bits, which determine the selected transform out of the four we have described above. Further transform specific information can follow, depending on the transform. When there are no more transforms, it is denoted by a single 0 bit.
7. Color cache (variable size). If the color cache is used, a single 1 bit is followed by four bits determining the color cache size. Otherwise, a single 0 bit is present.
8. Meta prefix codes (variable size). Single 1 bit indicates the presence of the meta prefix codes, followed by their data. Otherwise, a single 0 bit is present.
9. Prefix codes (variable size). The encoding information. We will describe it in detail in the following section, as it is the part of the format containing the component susceptible to the overflow.
10. Encoded image (variable size). The actual data used for decoding the individual pixels.

2.3 Huffman codes

One possible algorithm we can use to construct prefix codes is called Huffman coding. However, the Huffman algorithm itself is not relevant to the vulnerability. Furthermore, as the libwebp functions and structures that work with the prefix codes usually contain the word Huffman in their names, and as the public often (incorrectly) refer to prefix codes as Huffman codes anyway, we use the two terms interchangeably. [46]

Figure 2.2 Lossless bitstream structure with color cache present and without meta prefix codes, inspired by [44]



We call the values we want to encode *symbols*. The pool of these symbols is the *alphabet*, and the *alphabet size* is the number of the symbols. Huffman coding creates a pool of sequences of bits, where each sequence is mapped to a symbol from the alphabet, and each such symbol is represented by exactly one sequence. The sequences, in general, have different lengths. By *length* of a symbol, we understand the length of its sequence of bits. The special requirement of Huffman coding (or prefix codes in general) is that any sequence must not be a prefix of another sequence. If we satisfy this requirement, we do not need to store additional information where one encoded symbol ends and another begins.

2.3.1 Huffman trees

We can describe the workings of the Huffman coding on Huffman trees. As the Huffman tree is a *binary tree*, which is a special type of *tree*, which is a special type of *graph*, we will introduce the simplified basics of graph theory related to the subject first.

A graph is a mathematical structure consisting of *vertices* and *edges*. Each edge connects two vertices and no two edges connect the same pair of vertices. When visualizing graphs, we usually draw vertices as circles and edges as lines between the circles.

All vertices connected by an edge to a vertex are called *neighbors* of that vertex.

We define *path* between two vertices a and b as a sequence of unique vertices $a, v_1, v_2, \ldots, b$, such that there exists an edge connecting each pair of neighboring vertices in the sequence.

If exactly one path exists for each pair of vertices in a graph, then we call the graph a tree.

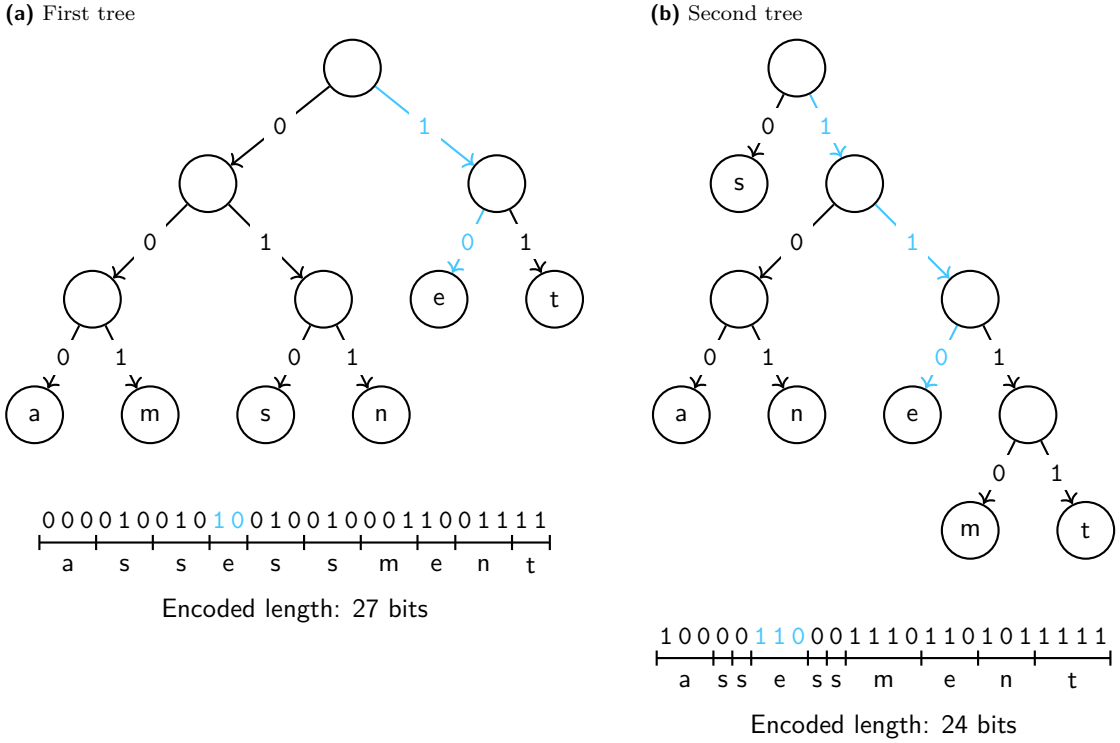
A binary tree is a tree where we select one vertex as the *root*, each vertex except the root has its *parent* (the next vertex on the path from this vertex to the root), and each vertex can have up to two *children* (the other neighbors besides the parent), where we differentiate between the *left* or the *right* child. If a vertex has no children, we call it a *leaf*. Otherwise, we refer to

it as an *inner vertex*. The *layer* of a vertex is its distance from the root (the root is at layer 0, its children are at layer 1, …). By *depth* of a tree, we mean the maximum layer in the tree. We typically draw binary trees by placing the root vertex at the top and expanding the graph downwards.

We now demonstrate the process of decoding a bitstream encoded by Huffman coding on a Huffman tree, which is a binary tree with the symbols stored in its leaves. Beginning at the root, we read the bitstream bit by bit. If we read a zero, we descend to the left child, and if we read one, we descend to the right child. Whenever we get to a leaf, we read the symbol stored in it, which is the decoded symbol. We immediately move back to the root and repeat the process. [46]

As long as we comply with the unique prefix condition, we can choose the bit sequence mapping, and thus the shape of the Huffman tree, arbitrarily. We provide an example of two possible Huffman trees in Figure 2.3. As we can see, differently shaped trees can provide various levels of compression. To maximize the compression, the more often a symbol occurs, the shorter sequence we want to assign to it. This is exactly what the Huffman coding algorithm does, and running it on the word *assessment* produces the second tree in the figure.

Figure 2.3 Two possible Huffman trees, decoding the word *assessment*



2.3.2 Huffman coding using tables

While decoding the bitstream by reading one bit at a time is possible, it is not very fast. We can improve it by implementing the Huffman tree as a table instead. Let d denote the tree's depth. We create a table that has 2^d entries (indexing starts at 0). By using such size, we can index the table with every possible binary number of length d (including those with leading zeros). Later, when we are decoding the image, we read d bits from the bitstream and store it in a number i. By using the ordering we have described previously (first read bit is stored in the LSB of the

number), the first "move" while descending the tree is determined by the LSB of i. For example, let $d = 6$ and the length of the symbol S we are currently decoding be $l = 4$. We want to use i as an index to the table and immediately get the symbol. The problem is we have read two more bits than we needed, which are already part of the following symbol and thus can be in any of the possible $2^{d-l} = 2^2 = 4$ states. We compensate for this by storing this symbol in four table entries (indexes are all states of two bits "joined" by the $l = 4$ bits representing the symbol).

Each of the entries has two fields—*value* and *bits*. The value field stores the symbol S that the entry represents, and the bits field says by how many bits it is represented (l). This simplifies the decoding process as follows. We look at the first d bits (but do not remove them from the bitstream yet), which we use as an index to the table. The indexed entry gives us the decoded symbol and the number of bits l it was represented by. We now remove l (not d) bits from the bitstream and repeat the process.

This approach speeds up the decoding process significantly. The unfortunate property it has is that the table size grows exponentially with the tree's depth. A compromise between decoding speed and required memory is to use *multi-level tables*. We focus on two-level tables and describe them as they are implemented in libwebp, although the concept can be easily generalized to more levels.

Again, we use some number of bits to index the table; let us denote it by d_1. However, this time, d_1 is less than the depth of the tree, resulting in a smaller table. We refer to this table as the *first-level table* (or the *root table*). If the entry's symbol is represented by no more than d_1 bits, then we read the symbol from the entry and proceed the same as with the original single-level table. Otherwise, if the symbol is represented by strictly more bits than d_1, then the entry's fields serve different purposes. Another table with the same shape (entries storing value and bits), called a *second-level table*, is present. We call the number of bits used for indexing it d_2. Information where to find this table is stored in the entry's value field. The entry's bits field stores the value $d_1 + d_2$. We now remove d_1 bits from the bitstream and use the next d_2 bits to index the second-level table as usual. Depending on the shape of the tree, it is common to have multiple second-level tables.

In Figure 2.4, we provide an illustration of a two-level tree with a 4–bit root table ($d_1 = 4$) and with two second-level tables with d_2 equal to 1 and 2. It uses the libwebp implementation that places the tables directly next to each other, and the value field pointing to a second-level table contains the distance between the root table's entry and the start of the corresponding second-level table. [47]

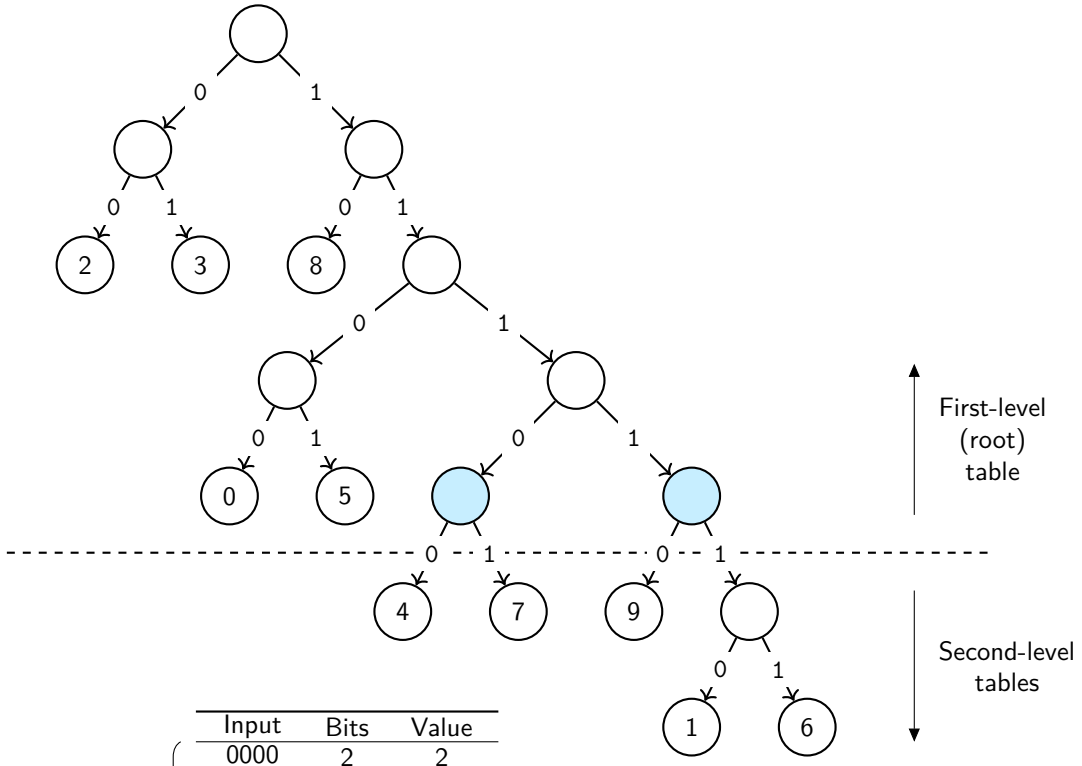
2.3.3 Canonical representation of Huffman codes

There are two aspects we need to consider regarding the representation of Huffman trees. One is how we store them in memory, which we have addressed by the two-level tables. The other is how to store the tree in the bitstream. As the whole idea of a compressed image is to reduce its total size, and as the shape of the tree is stored in the bitstream as well, we want to choose a representation of the tree that contains as little redundant information as possible. One possible solution is to use the canonical representation. [45]

For each symbol in the tree, we remember only its length, i.e., the layer of its leaf. When reconstructing the tree, we sort the symbol-length pairs, primarily by length and secondarily by symbol (this assumes the symbols in the alphabet themselves can be sorted, i.e., for each pair of distinct symbols, we can say which comes first). We then iterate over the sorted pairs, constructing the tree. For each symbol we iterate over, we create a leaf vertex in the "leftmost" available position in the specified layer (adding inner vertices above if necessary).

We cannot store all trees in this way. First, the tree expands "to the right". Second, the symbols in the same layer are placed in ascending order. However, these properties do not limit us in any way, as we can reorganize any tree to this shape. The symbols may then be represented by different bits than before, but their lengths remain unchanged.

Figure 2.4 Two-level tables implementation, 4-bit root table



	Input	Bits	Value
First-level table	0000	2	2
	0001	2	8
	0010	2	3
	0011	4	0
	0100	2	2
	0101	2	8
	0110	2	3
	0111	5	9
	1000	2	2
	1001	2	8
	1010	2	3
	1011	4	5
	1100	2	2
	1101	2	8
	1110	2	3
	1111	6	3
Second-level table	0	1	4
	1	1	7
Second-level table	00	1	9
	01	2	1
	10	1	9
	11	2	6

Furthermore, if we know the alphabet in advance, we can omit the information about the individual symbols and only store their lengths. For instance, if we always have an alphabet with symbols from 0 to $n-1$ for some n, we can store only the n and the lengths of the symbols (we first store the length of symbol 0, then the length of symbol 1, and so on). Using the graph from Figure 2.4, we would store 10, followed by $4, 6, 2, 2, 5, 4, 6, 5, 2, 5$.

2.3.4 Huffman codes in libwebp

The main reason we talk about Huffman codes is that they are used in the libwebp library to encode the picture. There are actually five separate trees—*green*, *red*, *blue*, *alpha*, and *distance*. Their alphabets are numbers from zero to the alphabet size minus one. We note these sizes in Table 2.2, as they are described in [45]. *color_cache_size* can be a value from 0 to 2048. The implementation stores these trees as two-level tables in one big buffer, one after another.

Table 2.2 Sizes of the different Huffman trees in libwebp

Table	Alphabet size
Green	$256 + 24 + color\_cache\_size$
Red	256
Blue	256
Alpha	256
Distance	40

The actual decoding of the image is not important to us as the overflow happens earlier, during the Huffman tree reconstruction. Still, we at least indicate how the decoding works because it explains the atypical size of the green tree.

Upon decoding a symbol with the green tree, we decide what to do next. It determines which of the compression methods we have described in section 2.2.2 is currently being used. Let us call the decoded symbol S. In line with [45], depending on the value of S, the decoding continues as follows:

$S < 256$ Use prefix-coded literals. Use S as the value of the green channel. Next, read from the bitstream to decode another three symbols from the red, blue, and alpha trees. The decoded symbols are the values for the red, blue, and alpha channels, respectively.

$256 \leq S < 256 + 24$ Use LZ77 backward reference. Depending on the value of S, read further bits. These bits, together with the symbol S, describe the length. Then decode one more symbol from the distance tree using the bitstream. This second symbol is the distance.

$256 + 24 < S$ Use color cache code. Use $S-256-24$ as the index into the color cache, retrieving the value for the current pixel.

2.4 The implementation

As the WebP format is fairly complex, the libwebp library contains many functions and structures. However, only a few of them are related to the vulnerability. We focus on the relevant parts in this section. For the sake of clarity, the pseudocodes we present are simplified to capture the important logic/code and not burden us with irrelevant parts and implementation details. We conduct the analysis on the last vulnerable commit, i.e., the commit 7ba44f80f3b94fc0138db159afea770ef06532a0.

We call a function for reading bits `ReadBits(n)`, and the bit ordering is in line with the ordering in section 2.2. In other words, reading $n > 1$ bits by calling `ReadBits(n)` behaves the same as `ReadBits(1) | (ReadBits(n - 1) << 1)`. We assume we have already read and processed the bitstream up to the color cache information (we presented the format in section 2.2).

2.4.1 ReadHuffmanCodes

Code listing 2.1 Structure `HuffmanCode` [48]

```
typedef struct {
  uint8_t bits;   // number of bits used for this symbol
  uint16_t value; // symbol value or table offset
} HuffmanCode;
```

Code listing 2.2 Function `ReadHuffmanCodes` [49]

```
#define FTS (630 * 3 + 410) // fixed table size
const uint16_t kTableSize[12] = {
  FTS + 654, FTS + 656, FTS + 658, FTS + 662,  FTS + 670,  FTS + 686,
  FTS + 718, FTS + 782, FTS + 912, FTS + 1168, FTS + 1680, FTS + 2704
};
const uint16_t kAlphabetSize[5] = {256 + 24, 256, 256, 256, 40};
int ReadHuffmanCodes(int color_cache_bits) {
  if(ReadBits(1)) { ... read and use meta prefix codes ... }
  const int max_alphabet_size = kAlphabetSize[0]
    + ((color_cache_bits > 0) ? 1 << color_cache_bits : 0);
  const int table_size = kTableSize[color_cache_bits];
  int* code_lengths = calloc(max_alphabet_size, sizeof(int));
  HuffmanCode* huffman_tables = malloc(table_size * sizeof(HuffmanCode));
  HuffmanCode* huffman_table = huffman_tables;
  int ok = 0;
  for(int i = 0; i < 5; i++) { // green, red, blue, alpha, and distance codes
    int alphabet_size = kAlphabetSize[i];
    if(i == 0                  // if processing green codes
      && color_cache_bits > 0) // and color cache is used
      alphabet_size += (1 << color_cache_bits); // increase alphabet size
    int size = ReadHuffmanCode(alphabet_size,
                               code_lengths,
                               huffman_table); // read one code
    if(size == 0)
      goto Error;
    huffman_table += size; // point to the next table
  }
  ok = 1;
Error:
  free(code_lengths);
  if(!ok)
    free(huffman_tables);
  return ok;
}
```

The first function we look at is `ReadHuffmanCodes` (Code listing 2.2). We start by reading the meta prefix codes. As previously stated, the meta prefix codes do not influence the vulnerability, so we do not give them further attention and assume they are not present.

`max_alphabet_size` stores the maximum possible alphabet size, which is the alphabet of the green code. It is always at least $256 + 24$, possibly more for color cache, as we have outlined in section 2.3.4. The array `kAlphabetSize` stores the five alphabet sizes (without color cache).

The implementation uses two-level tables. The root table and the second-level tables are all stored in one buffer, one after another. Moreover, the five two-level tables themselves are also stored one after another, meaning there is only one big buffer to accommodate all of them. The maximum possible buffer sizes the tables can require are precomputed in the `kTableSize` array. As we can have 12 different values of `color_cache_bits`, this table stores 12 possible table sizes. The value we use is stored in `table_size`.

`code_lengths` will be used later when building individual tables, but for optimization reasons, it is allocated only once here. Its size is set to the maximum possible alphabet size.

`huffman_tables` is the buffer for storing all five tables. The structure `HuffmanCodes` (Code listing 2.1) represents one entry of the table and has the value and bits fields as we have described in section 2.3.2. This is the buffer susceptible to buffer overflow. We will talk about it in detail in the following chapter.

We now iterate five times. In every iteration we keep a pointer to the first available position in `huffman_tables`, called `huffman_table`. In each iteration, we call the `ReadHuffmanCode` function, which constructs one table. We pass it its alphabet size, the `code_lengths` buffer, and the `huffman_table` pointer that says where to construct the table. If the function succeeds, it returns how much memory it used in the `huffman_tables` buffer, and we increment `huffman_table` accordingly. If the function fails, it returns zero. In such a case, we do not decode any further tables. The error is then propagated all the way to the user calling the library's API (application programming interface) function.

2.4.2 ReadHuffmanCode

The function `ReadHuffmanCode` (Code listing 2.3) encapsulates reading one Huffman table from the bitstream and constructing it as a table. It gets constructed at the position pointed to by the `table` pointer somewhere in the `huffman_tables` buffer.

The Huffman table is in the canonical representation. We store this representation in the `code_lengths` buffer. However, there are two variants of how the canonical representation itself is stored in the bitstream. First, the *simple code length code* is a special case if only one or two symbols are used, and we do not discuss it further. The main variant is called *normal code length code*. [45]

If the normal code length code variant is used, the code lengths (the canonical representation) are also encoded by the Huffman codes for additional bitstream size savings. We read the canonical representation of the code lengths and store it in the `code_length_code_lengths` array. The order of the symbols is determined by the constant table `kCodeLengthcodeOrder`. We then call `ReadHuffmanCodeLengths`, which decodes the canonical representation and stores it in `code_lengths`.

Once the canonical representation is decoded, we can construct the Huffman table itself by calling `BuildHuffmanTable`.

2.4.3 ReadHuffmanCodeLengths

The next function is `ReadHuffmanCodeLengths`. Its purpose is to decode the encoded canonical representation. We call a function that reads bits from the bitstream and uses them to decode one symbol from the Huffman code table `ReadSymbol(table)`.

`code_length_code_lengths` contains the canonical representation of the code lengths. The variable `num_symbols` is the alphabet size of the Huffman tree we will build eventually. The buffer we write the decoded canonical representation to is pointed to by `code_lengths`.

Code listing 2.3 Function `ReadHuffmanCode` [49]

```
int ReadHuffmanCode(int alphabet_size,
                    int* const code_lengths,
                    HuffmanCode* const table) {
  int ok = 0, size = 0;
  memset(code_lengths, // reset code_lengths
         0,
         alphabet_size * sizeof(int));
  if(ReadBits(1)) { ... } // simple code length code
  else { // normal code length code
    const uint8_t kCodeLengthCodeOrder[19] = {
      17, 18, 0, 1, 2, 3, 4, 5, 16, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
    };
    int code_length_code_lengths[19] = { 0 };
    const int num_codes = ReadBits(4) + 4; // number in range 4..19
    for(int i = 0; i < num_codes; i++)
      code_length_code_lengths[kCodeLengthCodeOrder[i]] = ReadBits(3); // 0..7
    ok = ReadHuffmanCodeLengths(code_length_code_lengths,
                                alphabet_size,
                                code_lengths);
  }
  if(ok) size = BuildHuffmanTable(table, 8, code_lengths, alphabet_size);
  return size;
}
```

We first create a table with 128 entries. Then, we call the `BuildHuffmanTable` function, which uses `code_length_code_lengths` to build a Huffman table for decoding the individual lengths of the symbols. Next, we iterate over all symbols, determined by `max_symbol`. In each iteration, we decode the length of one symbol and store its entry to the variable `p`. If `p.value` is less than 16, then it represents the length of that symbol and we store it in the `code_lengths` array. If it is 16 or more, it specifies how to set `code_lengths` for multiple symbols at once. A detailed description is available in [45].

2.4.4 BuildHuffmanTable

The `BuildHuffmanTable` function reconstructs one Huffman table based on its canonical representation. It builds the two-level tables in the way we have described in section 2.3.2 and pictured in Figure 2.4. To better understand the function, we first explain the helper functions `ReplicateValue`, `GetNextKey`, and `NextTableBitSize` (Code listing 2.5):

`ReplicateValue` This function copies the `code` entry to all the entries in the table where it should go. That is, the entries with all possible combinations of the unused MSB bits. The first entry position is specified by the `table` pointer. `step` is equal to 2^l, where l is the number of LSB bits that belong to the symbol, and `end` is the table size.

`GetNextKey` We can describe each leaf in the tree with a number, stored in the `key` variable. If we have a leaf at layer l, it is described by l bits. We store the bits in the l LSB bits of `key`, as we have introduced the coding in section 2.3.2. This function "increments" `key` to point to the next available leaf in the same layer; the layer is the `len` variable. The incrementation happens as follows:

Code listing 2.4 Function `ReadHuffmanCodeLengths` [49, 48]

```
#define LTB 7 // lengths table bits
int ReadHuffmanCodeLengths(const int* const code_length_code_lengths,
                           int num_symbols,
                           int* const code_lengths) {
  HuffmanCode table[1 << LTB];
  if(!BuildHuffmanTable(table, LTB, code_length_code_lengths, 19))
    return 0;
  int max_symbol = num_symbols;
  if(ReadBits(1)) // do not use all symbols
    max_symbol = /* read from bitstream, must not
                    be greater than num_symbols */
  for(int symbol = 0; symbol < max_symbol; symbol++) { // fill code_lengths
    HuffmanCode p = ReadSymbol(table);
    int code_len = p.value;
    if(code_len < 16)
      code_lengths[symbol] = code_len;
    else { // 16 <= code_len <= 18
      ... read additional bits and fill multiple symbols at once ...
    }
  }
  return 1;
}
```

1. Take `len` LSB bits from `key`.
2. Calculate a bitwise "mirror", i.e., LSB bit swaps places with MSB bit, second LSB bit swaps places with second MSB bit, …
3. Increment the number by one.
4. Mirror the bits back to the original order.

If the `len` LSB bits in `key` were already all ones, the function does nothing and returns the original `key`.

`NextTableBitSize` When creating a second-level table, we need to know how big it needs to be. The function `NextTableBitSize` calculates this size. It uses the variable `left` to store the number of open leaves in the current layer in the second-level table it simulates. The array `count` keeps track of how many symbols of each length we have yet to process. If all unprocessed symbols of the current length (`count[len]`) can fit in the table, it continues to the next layer and repeats the process. Once the table cannot hold any more symbols, it returns its final size.

The `root_table` pointer says where to store the table. As it is a two-level table, `root_bits` determines how many bits we use for the root table. The maximum possible length of a symbol is 15 bits. `code_lengths` stores the canonical representation, where we use a symbol as an index and the value is its length. If the length is zero, then the symbol is not present in the tree. `code_lengths_size` contains the size of the `code_lengths` buffer.

We start by iterating over the lengths from one to `root_bits`, as these symbols fit in the root table and do not require a second-level table. If the tree cannot contain all symbols with the current length, we return an error. Inside each iteration, we loop over all the symbols with the current length `len`. For each of the symbols, we create an entry with the value field set to

Code listing 2.5 Helper functions for `BuildHuffmanTable` [47, 50]

```
void ReplicateValue(HuffmanCode* table,
                    int step,
                    int end,
                    HuffmanCode code) {
  do {
    end -= step;
    table[end] = code;
  } while(end > 0);
}

uint32_t GetNextKey(uint32_t key,
                    int len) {
  uint32_t step = 1 << (len - 1);
  while(key & step) {
    step >>= 1;
  }
  return step ? (key & (step - 1)) + step : key;
}

#define MAX_ALLOWED_CODE_LENGTH 15
int NextTableBitSize(const int* const count,
                     int len,
                     int root_bits) {
  int left = 1 << (len - root_bits); // number of possible leaves in this layer
  while(len < MAX_ALLOWED_CODE_LENGTH) {
    left -= count[len]; // check if all symbols of this length can fit
    if (left <= 0) break; // table is full, finish
    ++len; // move to the next layer
    left <<= 1; // there is twice as many possible leaves in the next layer
  }
  return len - root_bits;
}
```

the symbol and the bits field set to the symbol's length. We then copy the entry (`code`) to the correct indexes by calling `ReplicateValue`. Then we generate the next key with the `GetNextKey` function. Finally, after we have processed a symbol, we decrement the counter of yet unprocessed symbols of this length (the array `count`).

After processing all symbols that fit in the root table, we move on to the remaining symbols (which require a second-level table), i.e., those with lengths from `root_bits` plus one to 15. The algorithm is the same up to the iterations over the symbols. Description of one iteration follows. We compare `root_bits` LSB bits of `key` to the variable `low`. If the two values are not equal, we need to create a new second-level table. `low` is initially set to a value that ensures the values are not equal on the first iteration, resulting in the creation of the first second-level table. We create the second-level table with the following steps:

1. Add the size `table_size` of the last table (initially the size of the root table) to the `table` pointer. This ensures the pointer now points to the start of the new second-level table.
2. Calculate the number of bits used for indexing the new table by calling `NextTableBitSize` and store it in `table_bits`. Use `table_bits` to derive `table_size`.

Code listing 2.6 Function `BuildHuffmanTable` [47, 50]

```
#define MAX_ALLOWED_CODE_LENGTH 15
int BuildHuffmanTable(HuffmanCode* const root_table,
                      int root_bits,
                      const int code_lengths[],
                      int code_lengths_size) {
  int count[MAX_ALLOWED_CODE_LENGTH + 1] = /* maps length to number of
                                              symbols with that length */
  HuffmanCode* table = root_table; // pointer to the current table
  int total_size = 1 << root_bits; // size of root and all 2nd level tables
  int table_bits = root_bits; // key length of current table (root initally)
  int table_size = 1 << table_bits; // size of current table (root initally)
  uint32_t key = 0, low = 0xffffffffu, mask = total_size - 1;
  // build root level table
  for(int len = 1; len <= root_bits; len++) {
    int step = 1 << len;
    int symbols[] = /* symbols with length len, sorted by symbol; array is
                       constructed from code_lengths and code_lengths_size*/
    if(/* not enough space in tree for symbols with length len */) return 0;
    for(int symbol in symbols) {
      HuffmanCode code = {.bits = len, .value = symbol};
      ReplicateValue(&table[key], step, table_size, code);
      key = GetNextKey(key, len);
      count[len]--;
    }
  }
  // build 2nd level tables
  for(int len = root_bits + 1; len <= MAX_ALLOWED_CODE_LENGTH; len++) {
    int step = 1 << len;
    int symbols[] = /* as above */
    if(/* as above */) return 0;
    for(int symbol in symbols) {
      if((key & mask) != low) { // need new 2nd level table
        table += table_size; // update table to point to this 2nd level table
        table_bits = NextTableBitSize(count, len, root_bits);
        table_size = 1 << table_bits; // update table bits and size
        total_size += table_size; // add the new table to the total size
        low = key & mask; // index to point the root table entry to this table
        root_table[low] = {.bits = table_bits + root_bits,
                           .value = (table - root_table) - low};
      }
      HuffmanCode code = {.bits = len - root_bits, value = symbol};
      ReplicateValue(&table[key >> root_bits], step, table_size, code);
      key = GetNextKey(key, len);
      count[len]--;
    }
  }
  if(/* tree is not full */) return 0;
  return total_size;
}
```

3. `total_size` represents the total size of the root table plus all second-level tables. Update it by adding `table_size` to it.

4. The lower bits of `key` represent the index of the entry in the root table that will point to the new table. Store them in `low`.

5. Add an entry to the root table that points to the new table. The bits field stores the number of the root table bits plus the number of the bits of the new table. The value field stores the offset of the new table from the entry in the root table, as we have shown in Figure 2.4.

Once the second-level table is ready, we create an entry for the symbol. We then store it in the appropriate positions, get the next key, and decrement the length counter, similarly to before. The first parameter in the `ReplicateValue(&table[key >> root_bits], ...)` function call points to the first position in the second-level table we will copy the entry to, as `table` points to the second-level table and the expression `key >> root_bits` gives us the bits for addressing it.

Finally, we check if the tree is *full*, i.e., each inner vertex has both children. If it is not full, we return an error.

The edge case where only one symbol is present is managed separately by copying it to every entry in the root table. We did not include this part in the code listing.

Chapter 3

Analyzing the vulnerable component

The overflow occurs when we are calling `BuildHuffmanTable`. The actual OOB write happens in the `ReplicateValue` function. `BuildHuffmanTable` relies on the fact that the constructed Huffman table will fit into the provided buffer. In the code we provided in section 2.4, we can see that we call the function from two different places.

First, we call it from `ReadHuffmanCodeLengths`. For this call, we provide a buffer that can hold 128 entries and we set root bits to 7. The buffer `code_lengths` points to is the `code_length_code_lengths` created in the `ReadHuffmanCode` function. We obtain the values in `code_length_code_lengths` by repeatedly reading three bits. Thus, every value is at most 7. However, this means that all the symbols fit in the root table whose size is correct ($2^7 = 128$), and `BuildHuffmanTable` will not create any second-level tables. We conclude that no overflow can happen by this call.

The second time we call it is directly from `ReadHuffmanCode`. This time we use it to construct one of the five tables. We place all the tables in the `huffman_tables` buffer, one after another. We have allocated this buffer in the `ReadHuffmanCodes` function. The size of this buffer comes from the `kTableSize` array. There is a comment next to this array in the source code explaining the origin of the sizes. Part of the comment says, "*All values computed for 8-bit first level lookup with Mark Adler's tool*", followed by a link to the source code of the tool. [49]

The first line of the tool's source code explains its purpose: "*enough.c – determine the maximum size of inflate's Huffman code tables over all possible valid and complete Huffman codes, subject to a length limit.*". Analyzing the tool is outside the scope of this thesis, and as we have not come across any evidence indicating it works incorrectly, we assume the sizes it generates are valid. [51]

The critical detail is that the tool only counts valid and *complete* Huffman codes. Complete Huffman trees (or full Huffman trees) are those where we cannot add any more leaves, i.e., each vertex is either a leaf or an inner vertex with two children.

If we look back at the code of the `BuildHuffmanTable` function, we can see that it checks whether the tree is full or not. If it is not, libwebp propagates the error and finalizes all of its structures and exits. The problem is that this check is done *after* we have already written the incomplete tree's tables to the buffer. If we can construct an incomplete tree requiring more space than the buffer provides, we will achieve buffer overflow.

3.1 Overflow reach

In this section, we determine the furthest position where we can overwrite the memory after the buffer's end. We refer to this value as the *reach*. We first analyze the total size of the tables of a single tree without taking into consideration the other four. By *second-level tree*, we understand the part of a tree that is represented by a second-level table. We consider only trees from the library, i.e., 15 layers and second-level trees have their roots at layer 8).

Each tree has its alphabet. The alphabet size is equal to the number of leaves. Placement of these leaves determines the shape of the tree, and thus the number of second-level tables and their sizes. For us to maximize the reach, we need to maximize the sum of the root table and all the second-level tables for a fixed number of leaves. The root table always has 256 entries, so we can analyze just the size of the second-level tables and add 256 afterward.

If we place a leaf at layer 8 or higher, it gets placed directly in the root table. This means placing leaves in these layers does not increase the total size, so placing them there is a "waste". As a result, we will be placing all leaves at layer 9 or lower. To simplify the layer numbering and the equations we provide later, we will be counting layers from the root of a second-level tree from now on.

The size of each second-level table is determined by the layer of the lowest leaf in its tree. If we call this layer l, then the table size is 2^l. The first idea we have is to place leaves in a way such that for each second-level tree, we place exactly one leaf at layer 7. This would increase the total size by 128 for every leaf. Unfortunately, we cannot achieve such a shape. This is because the tree is described in the canonical representation.

When building a tree from its canonical representation, we go from the highest layer to the lowest, and inside each layer we place leaves from left to right, always at the first available position. This implies that if the nth second-level tree has its last leaf at layer l, the $(n+1)$th second-level tree can place its leaves only at layer l or lower.

We can observe that the only second-level tree that can be incomplete is the last one (the rightmost one). The remaining second-level trees must be complete. We split the analysis into two parts—first, we look into the complete second-level trees and then into the last incomplete one.

3.1.1 Complete second-level trees

If there is no danger of confusion, we will call second-level trees simply as trees in this section.

Because of the canonical representation, each tree cannot start at a higher layer than where the previous tree has placed its last leaf. We name this layer the tree's *start layer* and mark it as s. Similarly, the layer where the tree's lowest leaf is is the *end layer*, which we mark as e. As we noted previously, the size of this tree's table is 2^e. However, the number of leaves it requires depends on the tree's shape. We want to use as few leaves as possible. If we imagine a complete tree with all of its leaves in some layer l, we need some number of leaves n. But if we place all the leaves at layer $l+1$ instead, it requires $2n$ leaves. This means we need to place each leaf in the highest layer possible. The optimal strategy is as follows. We place $2^s - 1$ leaves at layer s, then place one leaf in each layer between s and e and finish with two leaves at layer e. Such a tree uses the fewest leaves for specified s and e. We call this tree *optimal*. The number of leaves in an optimal tree is as follows:

$$\text{leaves}\,(s,e) = (2^s - 1) + (e - s - 1) + 2 = 2^s + e - s$$

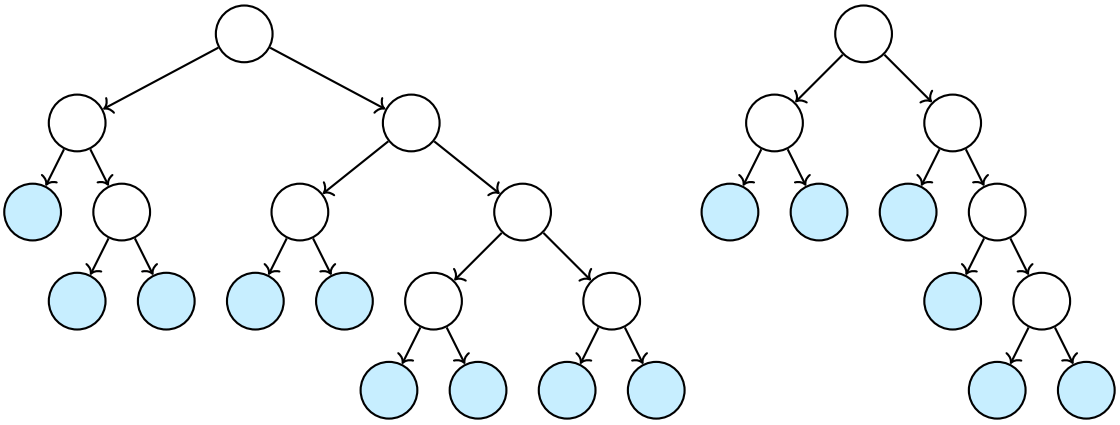
We can see a comparison of an arbitrary and optimal tree in Figure 3.1. As we want to maximize the reach, each of the second-level trees must be optimal, and in this section we will talk only about such trees.

For an optimal tree with start layer s and end layer e, we define a function $\text{gain}(s,e)$. This function expresses how much bigger the tree's table size is compared to the number of leaves it

Figure 3.1 Comparison of arbitrary and optimal second-level trees with $s = 2$ and $e = 4$, both creating table of size 16

(a) Arbitrary tree, 9 leaves

(b) Optimal tree, 6 leaves



uses. If we want to maximize the reach, we need to maximize the sum of the gains across all the complete second-level trees. We calculate the gain of one tree by subtracting the number of leaves from the table size:

$$\text{gain}\,(s,e) = 2^e - \text{leaves}\,(s,e) = 2^e - (2^s + e - s) = 2^e - 2^s + s - e$$

A special case of an optimal tree is a tree where $s = e$, i.e., all the leaves are in the same layer. The gain of such a tree is zero, meaning for each leaf we use, we increase the table size by one.

Let us consider two neighboring trees and three values: s, m, e. The first tree starts at layer s and ends at layer m, while the second one goes from layer m to layer e. We fix the values of s and e and look at what happens when we set m to different values $(s, s+1, \ldots, e)$. The bigger m is, the bigger the total tables' size, but also the number of used leaves. The important thing is that the combined gain remains constant, which we can also see in Figure 3.2:

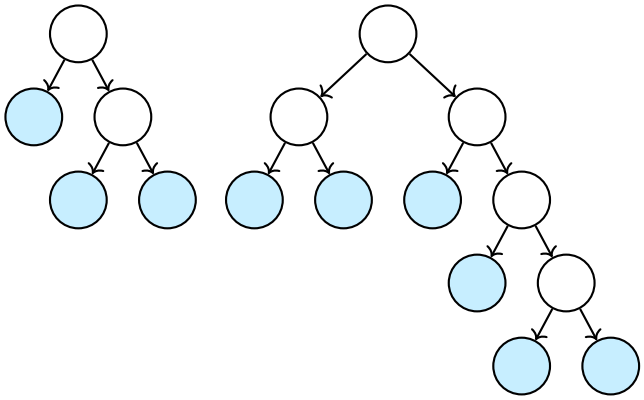
$$\begin{aligned}\text{gain}\,(s,m) + \text{gain}\,(m,e) &= (2^m - 2^s + s - m) + (2^e - 2^m + m - e)\\ &= 2^e - 2^s + s - e + (2^m - 2^m + m - m)\\ &= 2^e - 2^s + s - e\\ &= \text{gain}\,(s,e)\end{aligned}$$

We now extend this statement for n trees. If the ith tree starts at layer s_i and ends at layer e_i, then the condition $s_1 \le e_1 = s_2 \le e_2 = s_3 \le \ldots \le e_{n-1} = s_n \le e_n$ holds. We calculate the combined gain of all the trees:

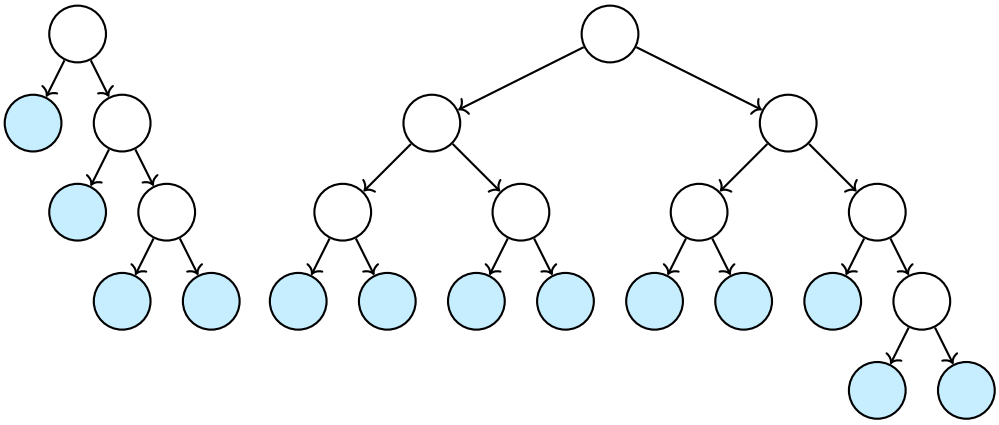
$$\begin{aligned}\sum_{i=1}^{n} \text{gain}\,(s_i,e_i) &= \sum_{i=1}^{n} (2^{e_i} - 2^{s_i} + s_i - e_i)\\ &= -2^{s_1} + s_1 + \sum_{i=1}^{n-1} ((2^{e_i} - e_i) + (-2^{s_{i+1}} + s_{i+1})) + 2^{e_n} - e_n\\ &= 2^{e_n} - 2^{s_1} + s_1 - e_n + \sum_{i=1}^{n-1} (2^{e_i} - e_i - 2^{e_i} + e_i)\\ &= 2^{e_n} - 2^{s_1} + s_1 - e_n\\ &= \text{gain}\,(s_1,e_n)\end{aligned}$$

Figure 3.2 Constant gain demonstrated on two pairs of second-level trees, both starting at layer 1 and ending at layer 4

(a) $m = 1$, table size $4+16 = 20$, leaves $3+6 = 9$, gain $20-9 = 11$



(b) $m = 2$, table size $8 + 16 = 24$, leaves $4 + 9 = 13$, gain $24 - 13 = 11$



This means that as long as we only use optimal trees, the total gain depends solely on the start layer of the first tree and the end layer of the last tree. The gain function gives the best results when s and e are as far apart as possible, i.e., when we span as many layers as possible. For the maximum span, the gain$(1, 7)$ is 120.

We now explore how to construct trees with this gain. We can use 8 leaves to construct a single tree with $s = 1$ and $e = 7$. As we cannot get any more gain, the best we can do is to exchange each of the remaining leaves for a plus one total table size increase. For this, we can use optimal trees with $s = e = 1$, which we lay on the left of the first tree. Each such tree will use two leaves and increase the total size by two. Depending on the number of leaves and s and e, we could be left with one leaf. Because the smallest optimal tree requires two leaves, no matter where we place this leaf, the total size will not increase. We can place it in the initial tree by rearranging it slightly (or somewhere in the incomplete tree later).

While this approach works in theory, we are limited to 256 second-level trees (the root table has 256 entries). We cannot just create, for example, trees with 8 leaves. That is because such a tree would have all leaves at layer 3, but the initial tree we built spans layers 1 to 7. Thanks to the canonical representation, those two trees cannot exist simultaneously, and all other trees must be either at layer 1 or layer 7, containing 2 and 128 leaves, respectively. This does not allow for much granularity, although creating as many trees as possible at layer 7 and then using the remaining leaves to construct trees at layer 1 would probably be sufficient.

Still, we present a different approach. It builds on the following idea. Let us have an optimal tree starting at some layer s and ending at some layer e. We then "split" it into two trees (by creating a new tree), such that the left one starts at s and ends at some m, and the right one starts at m and ends at e. As we observed previously, this modification does not influence the total gain. It, however, influences the number of leaves it needs. The constant gain implies that each additional leaf is converted to a plus one table size increase, as we want. We can calculate how many more leaves we used by this modification. We get the formula by subtracting the number of leaves in the original tree from the number of leaves in the two new trees:

$$\begin{aligned}\text{leaves}(s,m) + \text{leaves}(m,e) - \text{leaves}(s,e) &= (2^s + m - s) + (2^m + e - m) - (2^s + e - s) \\ &= 2^m + (2^s - 2^s) + (m - m) + (s - s) + (e - e) \\ &= 2^m\end{aligned}$$

This means we can split any tree into two in a way that it consumes 2^m more leaves for some m in the range from s to e. Building on this idea, we present an algorithm for building optimal trees that have the biggest possible gain, the number of the trees is minimal, and they use all the leaves (perhaps except one, assuming $s = 1$). We show the algorithm[1] implemented in Python in Code listing 3.1. We start again with one tree starting at layer `s` and ending at layer `e` and decrease the number of leaves left `leaves`. Next, we repeat the following as long as we have at least two unplaced leaves. We split the leftmost tree and select the biggest possible m, such that we still have at least 2^m leaves. We subtract 2^m from the number of remaining leaves `leaves`. Once we cannot split anymore, we return the constructed trees.

Code listing 3.1 Building optimal trees

```
def leaves_in_tree(s, e):
  return 2**s + e - s

def build_optimal_trees(s, e, leaves):
  trees = [(s, e)] # set the initial tree spanning all the available layers
  leaves -= leaves_in_tree(s, e) # subtract leaves in the initial tree
  while True:
    s, e = trees[0] # select the layers of the leftmost tree
    for m in range(e, s - 1, -1): # m = e, e - 1, ..., s + 1, s
      leaves_delta = 2**m
      if leaves_delta > leaves: # not enough leaves for this transform
        continue
      trees.insert(0, (s, m)) # add a new tree at the start
      trees[1] = (m, e) # modify the second tree accordingly
      leaves -= leaves_delta # update remaining leaves
      break
    else: # not enough leaves to create a new tree
      break
  return trees
```

[1] In Python, the code in an `else` block after a `for` loop executes when the loop finished "naturally", i.e., when it was not terminated by `break`.

3.1.2 Incomplete second-level tree

If we do not use all the leaves for the complete second-level trees, we can create one incomplete tree at the end, whose table is also the final one. This means the entry at the highest address in this table determines the final reach of the whole tree. Unlike the complete second-level tables, if present, the size of this table is always 128. The cause of this is the implementation of the `NextTableBitSize` function. As long as there are non-occupied positions left in the tree, this function keeps subtracting `count[len]` from `left`. However, as this tree is not complete, the `left` variable will always be greater than zero. This means the function finishes only after the condition in the `while` is not true. This gives us a constant bit size of `MAX_ALLOWED_CODE_LENGTH - root_bits`, so 7 in our case. The size of the table is therefore $2^7 = 128$.

The problem we face is that consecutive leaves do not create consecutive entries in the table. The order of the entries is determined by the `GetNextKey` function instead, which does not increment the key itself, but it increments the key's bitwise "mirror."

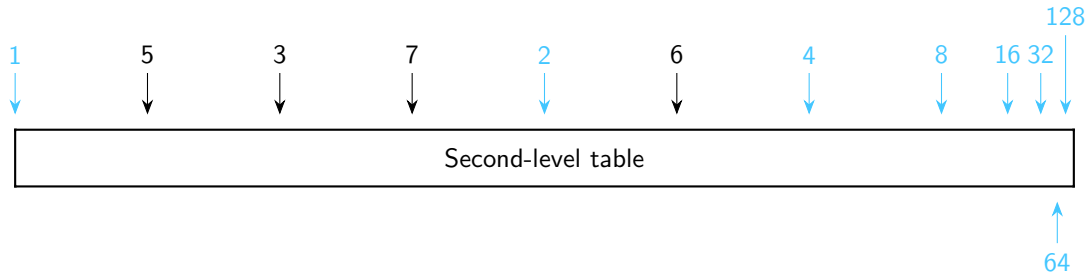
First, we look at what happens when we start placing leaves at layer 7, as if the end layer of the last complete second-level tree was 7. We refer to Table 3.1 and Figure 3.3. As the key corresponds to the position in the table, the first leaf we place goes to the beginning of the table. This means we used one leaf and increased the reach by one as well. Next, we place the second leaf, which goes to the middle of the table. This time, we used one leaf and increased the reach by 64. However, the third leaf goes between the first and the second—it does not increase the reach at all. The fourth leaf again increases the reach, this time by 32. But the cost of this reach increase is two leaves because we have to place both the third and the fourth leaves. The next reach increase is at the eighth leaf. It increases by 16, costing us an additional four leaves. This process continues as we pictured.

Table 3.1 Positions where leaves at layer 7 are placed, and the reach increases

Used leaves	Key (binary)	Reach increase	Additional leaves required
1	0000000	1	1
2	1000000	64	1
3	0100000		
4	1100000	32	2
5	0010000		
6	1010000		
7	0110000		
8	1110000	16	4
⋮	⋮		
16	1111000	8	8
⋮	⋮		
32	1111100	4	16
⋮	⋮		
64	1111110	2	32
⋮	⋮		
128	1111111	1	64

In the table, we show all 128 leaves to emphasize the pattern. However, we must use fewer than 128. This is because if we used all of them, the tree would be complete and be part of the complete second-level trees from the previous section.

Figure 3.3 Positions where leaves at layer 7 are placed



From the previous section, we know that, after maximizing the gain, we increase the reach by one for each of the remaining leaves. In the table, we see that we can achieve a better reach than one for each leaf (up to a certain point). For leaves at layer 7, we see we should use 16 leaves, which increase the reach by $1 + 64 + 32 + 16 + 8 = 121$. Whether we use the last eight leaves in this tree or leave them in the complete trees does not matter; in both cases, they increase the reach by eight.

We limited ourselves only to layer 7. We now generalize this calculation to take the highest available layer into consideration. If we place a leaf in some layer l, the effect is the same as placing two leaves at layer $l + 1$. For the "Reach increase" column in Table 3.1, each leaf at layer 7 corresponds to one row, each leaf at layer 6 corresponds to two rows, each leaf at layer 5 corresponds to four rows, …, up to 64 rows for each leaf at layer 1. In general, a leaf at layer l corresponds to 2^{7-l} rows. This means that to minimize the number of leaves used, we want to place them at the highest layer possible.

We might find ourselves in a situation where we filled up layer l with $2^l - 1$ leaves. We cannot add the last leaf, as that would complete the tree. However, we can still place one leaf at layer $l + 1$, then one at layer $l + 2$, and so on, resembling the shape of an optimal tree (without the last leaf). The question is whether placing leaves in the lower layers results in any further range increase. If layer l is filled with all leaves except one, we can get the number of rows used by multiplying the number of rows per leaf by the number of leaves. If we subtract this value from the total number of rows (128), we get the number of rows we did not use:

$$128 - \left(\left(2^{7-l}\right) \cdot \left(2^l - 1\right)\right) = 128 - \left(2^{7-l+l} - 2^{7-l}\right) = 128 - 128 + 2^{7-l} = 2^{7-l}$$

We get the maximum possible number of unused rows in layer 1, and that is $2^{7-1} = 64$; it is less for all other layers. However, we see that, out of the 64 last rows, only one increases the reach, i.e., row 128. But as we said previously, we cannot use this row. We conclude that to maximize the reach for the amount of leaves spent, placing leaves only in the highest available layer is optimal.

To calculate the optimal number of leaves to use for a specified minimum layer, we provide an algorithm in Code listing 3.2. We use the `prefix_sum` array to get the total reach for a selected number of used rows. As indexing in Python starts at zero, the array contains values $0, 1, 65, 65, 97, 97, 97, 97, 113, \ldots, 127, 128$. The function `incomplete_tree` calculates the optimal reach and the number of leaves used for the specified layer. In each iteration, we add one leaf and corresponding rows. We then check if the newly included rows contained an increase in reach. If they did, we test whether it is "worth" it to use all these rows, i.e., if the increase in reach is not smaller than its cost in leaves. If we decide to accept it, we update the `result_reach` and `result_leaves` variables with the new best values. Otherwise, if it is not "worth" it, we return the previous best result.

Since the layer can only take seven values, we present Table 3.2 containing the best reaches and numbers of used leaves for each of the layers.

Code listing 3.2 Building incomplete tree

```
increses_at = {1:1, 2:64, 4:32, 8:16, 16:8, 32:4, 64:2, 128:1}

prefix_sum = [0]
for leaves in range(1, 129):
  prefix_sum.append(prefix_sum[leaves - 1] + increses_at.get(leaves, 0))

def incomplete_tree(layer):
  result_reach = 0 # return value
  result_leaves = 0 # return value
  step = 2**(7-layer) # number of rows equivalent to one leaf
  rows = 0 # used rows so far
  prev_reach = 0 # total reach in previous iteration
  leaves = 0 # number of used leaves
  while rows + step < 128: # must not consume all entries
    rows += step
    leaves += 1
    this_reach = prefix_sum[rows] # get reach up to this row
    reach_delta = this_reach - prev_reach
    if reach_delta > 0: # check if reach increased from previous iteration
      leaves_delta = leaves - result_leaves
      if reach_delta < leaves_delta: # if reach increased by less than the
        break                        # number of leaves it consumed, finish
      result_reach = this_reach # update best reach
      result_leaves = leaves # update best leaves
    prev_reach = this_reach
  return result_reach, result_leaves
```

Table 3.2 Best reach and number of used leaves of an incomplete tree for each of the layers

Layer	Reach	Leaves used
1	127	1
2	127	2
3	127	4
4	125	4
5	125	8
6	121	8
7	121	16

3.1.3 Calculating the reach

The implementation places the five trees in one buffer. If one of the trees is not complete, the following trees will not be constructed. The best chance we have at achieving buffer overflow is to build four valid trees, each of which occupies the maximum amount of memory it can, and then overflow the fifth tree. Before we focus on this scenario, we first explore the possibility of overflowing one of the other trees.

3.1.3.1 Overflowing the first four trees

The reason we might want to do that is that the fifth tree has the smallest alphabet and the symbol influences the value of the bytes that are overwritten during overflow. As we have shown in Table 2.2, the last tree has the smallest alphabet, the tree before that has about six times more symbols, and the first tree has even more. This could be useful later, during exploitation. If we want to achieve a buffer overflow in this way, we need to overflow far enough to get past the memory reserved for all the trees that will not be constructed.

The memory used by a tree consists of three parts. First, there is a 256 entry root table. Second, the complete second-level tables. Third, the optional incomplete second-level table. We do not have to compute the sizes exactly, making an upper estimate of the size is good enough for now. If we cannot achieve overflow even with this size, we know we cannot achieve overflow with the actual maximum size. We dedicate all the leaves to maximizing the complete tables and simply say the incomplete table will add an additional 128. When constructing the complete tables, we can use the maximum span of layers, that is, $s = 1$ and $e = 7$. This gives us 120 gain (119 if we cannot utilize the last leaf, but 120 still serves as an upper estimate) plus a size increase equal to the alphabet size. By using a as the alphabet size, the total size is $256 + 120 + a + 128 = 504 + a$.

As the red, blue, and alpha trees have identical alphabets, we only have to analyze the one closest to the end of the buffer, i.e., the alpha tree. The green table has 12 possible alphabet sizes, depending on the color cache. For all of these alphabets, we compare the upper estimate with the size required to overflow the buffer. The sizes are measured in number of entries; the size in bytes is four times that. We show this comparison in Table 3.3. Based on the results, we conclude that none of the first four trees is able to overflow the buffer.

Table 3.3 Analysis of table sizes of the first four tables required to achieve buffer overflow

Alphabet size	Upper estimate of size	Size to reach the end of the buffer
256	760	1040
280	784	2954
282	786	2956
284	788	2958
288	792	2962
296	800	2970
312	816	2986
344	848	3018
408	912	3082
536	1040	3212
792	1296	3468
1304	1808	3980
2328	2832	5004

3.1.3.2 Overflowing the fifth tree

For us to maximize the reach, we first have to make the first four tables as big as possible. For simplicity, we will not use color cache. Fortunately, mistymntncop has already found the maximum tables. We use his canonical representations and show them in Table 3.4, as they are presented in [52].

We now construct the distance tree with the maximum reach. The alphabet size of this tree is 40. For the complete second-level tables, we always start at layer 1 as it maximizes the gain. The only thing we have to choose is the layer where the complete tables end and the incomplete table

Table 3.4 Number of leaves in layers leading to maximum complete table sizes [52]

Layer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Green	1	1	0	0	0	0	0	0	3	229	41	1	1	1	2
Red, blue, alpha	1	1	0	0	0	0	0	0	7	241	1	1	1	1	2

begins. We call it the *transition layer*. For each of the transition layers, we give the incomplete tree the number of leaves it requires, as we calculated in Table 3.2. We give the remaining leaves to the complete second-level trees. The reach of the complete tables is the number of leaves we gave to them plus the gain (computed with the formula we presented earlier, with start layer 1 and end layer equal to the transition layer; minus one if one leaf remains unpaired). The total reach is the reach from the complete tables, incomplete table, and an additional 256 for the root table. The size allocated for this tree is 410, so we need to exceed that to achieve buffer overflow. In Table 3.5, we present the final reaches and overflows.

Table 3.5 Maximum overflows for each transition layer, U. L. stands for unpaired leaf

Transition layer	Incomplete tree		Complete trees				Total	
	Leaves	Reach	Leaves	U. L.	Gain	Reach	Reach	Overflown by
1	1	127	39	Yes	-1	38	421	11
2	2	127	38	Yes	0	38	421	11
3	4	127	36	No	4	40	423	13
4	4	125	36	Yes	10	46	427	17
5	8	125	32	No	26	58	439	29
6	8	121	32	Yes	56	88	465	55
7	16	121	24	No	120	144	521	111

We can see that the maximum overflow we can achieve is when layer 7 is used as the transition layer. It overflows the buffer by 111 entries. As one entry (structure `HuffmanCode`) takes four bytes, we conclude the furthest we can overflow is 444 bytes.

Sometimes, we do not want to use the maximum reach. Instead, we might need to position specific memory overwrite closer to the end of the buffer. In that case, we have several options. We can rearrange the complete tables to "waste" two leaves at a time, which gives us a granularity of 8 bytes. This way, we can decrease the gain all the way to zero. If this is not enough, we can add or remove leaves from the incomplete tree. Finally, we can start placing leaves at a lower layer (instead of layer 1).

3.2 Overflow value

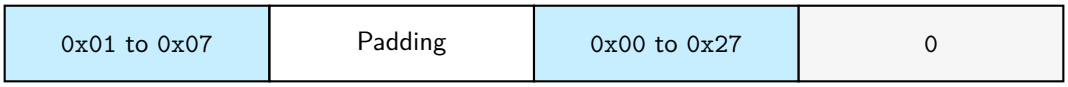
Each entry in the tables we overflow is the `HuffmanCode` structure. The bits field occupies one byte, and the value field occupies two bytes. Because of compiler optimizations, one padding byte is inserted between them. When copying this structure, the compiled code copies the byte for the bits field and the two bytes for the value field. This means the padding byte is not overwritten and remains unchanged. The overflown entries are the ones representing actual leaves, as the other type—those pointing to the second-level tables—are present only in the root table, which cannot overflow.

The bits field contains the length of the symbol, which corresponds to the layer of the leaf (counting from the eighth layer). As we saw in Table 3.5, overflow is possible for all bit lengths, i.e., 1 to 7. However, the higher the bits field is, the bigger reach we have.

The value field contains the symbol. The distance tree has 40 symbols, from 0 to 39. We can distribute them in the leaves almost arbitrarily. The only limitation we face is the canonical representation, which forces the symbols in one layer to be in ascending order. As the symbol is less than 255, the upper byte is always zero.

Taking the little-endian ordering into account, we present the memory structure of this `HuffmanCode` in Figure 3.4. In the attached files, we provide scripts for visualizing the trees and visualizing the memory writes they cause.

Figure 3.4 Four bytes of `HuffmanCode` corresponding to a leaf in the distance tree



Chapter 4

Exploiting the vulnerable component

Once we know what values we can overflow and how far after the buffer they are placed, we can start constructing the exploit. The goal of this chapter is to create a proof-of-concept application that we can exploit to arbitrary code execution. We use Ubuntu 24.04 with glibc version 2.39. and the libwebp library on the last vulnerable commit.

Based on the analysis of the vulnerability and the analysis of the heap in glibc, we are confident we can arrange the heap in a way that places the vulnerable buffer and our structures at the desired offset so that we can overwrite them. We will leave this part to the end once we know what structures we need.

4.1 Planning the exploit

For us to achieve arbitrary code execution, we need two things. First, we need to store our malicious code somewhere in memory, and second, we need to make the application execute this code.

Executing the code is easy—we overwrite the address of a function pointer stored on the heap. Once this function is called, our code will be executed. However, we need to determine what address to use. As we are on a 64bit system, the size of a pointer is 8 bytes. The `HuffmanCode` structure uses 4 bytes. This gives us three options:

- Overwrite the first 4 bytes. These bytes correspond to the 32 LSB bits of the value (little-endian). Overwriting this part is the easiest because of `malloc`'s alignment of chunks and the address parity to which most of the entries in the vulnerability are written.
- Overwrite the last 4 bytes. These bytes correspond to the 32 MSB bits of the value. Making the vulnerability overwrite this part is harder, but possible.
- Overwrite both the first and the last 4 bytes. We can achieve this by using two `HuffmanCodes` that get written next to each other. Achieving this is even more difficult, but still possible.

First, we look at overwriting the last 4 bytes, as it would enable us to set the pointer to values that span a much greater area of memory than modifying the first 4 bytes. The lowest theoretical address we can create in this way is `0x0000__01________`, and the highest address is `0x0027__07________` (underscores represent parts that remain unchanged). If the symbol is greater than zero, then the address falls in the non-canonical part of the address space. But even if we set the symbol to zero, it still does not represent any memory accessible under "normal"

circumstances. According to our testing, the heap is placed at much lower addresses and the shared libraries and user stack are too far as well. Overwriting the first 4 bytes in addition to the last 4 bytes does not solve this issue.

We instead look at modifying the first 4 bytes. As they correspond to the LSB bits of the value, the overall position of the pointer is mostly influenced by the other 4 bytes, which are outside our control. We could create an application where the last 4 bytes are set to our desired values, and we overwrite the rest. However, this would not be a very realistic proof-of-concept. Instead, we can work with a `NULL` pointer, which could reflect a real-life scenario. This makes the lowest theoretical address `0x00000001`, and the highest address `0x00270007` (we display only the 4 LSB bytes, as the remaining ones are zeros). However, looking at Figure 1.1, we see that there is nothing in the address space up to the address `0x00400000`.

At this moment, we tried creating the exploit on Windows instead. According to our testing, the stack on Windows is somewhere around the address `0x00190000`, which falls in the range we can create. While this seems like the solution, we failed at the other step, aligning the buffer and our structures on the heap. The author of [53] faced a similar obstacle. He solved it by using the low-level function `VirtualAlloc` instead of `malloc`. However, even when using this function, we were not able to arrange the memory.

We return to Linux. Even though we did not succeed on Windows, it gave us the idea to explore low-level memory management functions on Linux. We come across the `mmap` function. This function can be used when we want to access a file's content without manually reading it and storing it in memory. We pass it the file descriptor, and it returns a pointer. We can then access the file's content as if it was stored in memory, starting at the pointer. [54]

While this may be one of the function's purposes, it can do much more. Firstly, the content of the memory it returns does not have to be backed by a file. Actually, if the memory manager requires additional memory, it gets it by calling `mmap`. Secondly, the function can accept multiple flags, which modify the function behavior. Thirdly, it lets us specify the protections of the "created" memory. We can set whether the memory may be read, written, and executed. Lastly, we usually pass `NULL` as the first argument. However, if we pass some address instead, `mmap` takes it as a hint as to where it should create the memory at. [54, 14].

We can take advantage of this and get our malicious code to the desired address. Again, we create a structure on the heap, which will serve as a wrapper for parameters for `mmap`. One of the parameters is the pointer, initially set to `NULL`, as this resembles the common usage of the function. We then overwrite the pointer by overflowing the buffer. Finally, we call `mmap` and pass it the parameters from the structure. We use the fact that the `mmap` can map the content of a file to the memory and we save our malicious code in that file. The only "atypical" thing we have to do is to make the memory executable by passing the `PROT_EXEC` option to `mmap`. However, other protections, such as ASLR, can remain active.

The structure of the vulnerable application is the following:

1. Arrange the memory. While doing so, allocate two specific structures on the heap. One is the wrapper for parameters for `mmap`, one of which is the pointer. The other is the function pointer. Both pointers are set to `NULL`.

2. Call the libwebp API function to decode the provided image. Decoding this image will cause an overflow and overwrite *both* the function pointer and the pointer for `mmap`.

3. Execute `mmap` with the parameters from the structure. This loads our malicious code at the desired address.

4. Call the function in the wrapper, which executes our malicious code.

4.2 Arranging the heap

For the overflow, we chose a tree whose canonical representation is shown in Table 4.1. Only the two last leaves overwrite memory outside the buffer, and as they belong to the last layer, each does only one write. The first write is at offset 152 bytes from the end of the buffer, and the second write is at offset 24. The canonical representation requires that the first symbol is smaller than the second symbol. It is sufficient to use the last two symbols (38 and 39), although the exploit can be achieved with other symbol choices.

Table 4.1 Canonical representation of the tree that enables the exploit

Layer	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Distance tree	0	0	0	0	0	0	0	0	0	0	0	7	14	0	19

Therefore, we create addresses `0x00260007` at offset 152 and `0x00270007` at offset 28. We use these values to overwrite the two pointers. Out of the two addresses, we want to use the lower address for `mmap`, as it can create a big enough memory block such that it contains even the higher address. This way, the function pointer points to a memory block that we have entirely under control.

The last thing we need to do is to place the two structures after the vulnerable buffer. As we saw in chapter one, consecutive calls to `malloc` create consecutive chunks in memory. This, however, is under the condition that no chunks were freed before. If we have free chunks, the memory manager will try to reuse them again.

Ideally, we would allocate the buffer for the trees and then immediately allocate our two structures (which would get placed right after). Next, we call the libwebp function that triggers the overflow, overwriting our pointers. When the function finishes, we create the memory by calling `mmap`. Finally, we call the function pointer, which executes our code.

Unfortunately, we face two problems. The first is that the allocation of the buffer for the trees and the subsequent overflow all happen in one call of the libwebp function, meaning we cannot "insert" the allocation of our structures between these two actions. The second problem is that the function allocates and then frees other buffer as well, not only the one for the trees.

We show the relevant function calls during the decoding process in Code listing 4.1. The listing represents the function calls that occur when decoding our image, up to the allocation of the buffer for the trees. However, if our image used some other features, such as the meta prefix codes, there would be additional allocations.

Code listing 4.1 Function calls affecting the heap during the decoding of our WebP image

```
void* a = malloc(0x1d8);
void* b = malloc(0x1000);
void* c = malloc(0x262);
free(b);
free(a);
void* d = malloc(0x170);
void* e = malloc(0x460);
HuffmanCode* huffman_tables = malloc(0x2e28);
```

It turns out that arranging the heap to our needs is trivial. Described in Code listing 4.2, we only have to allocate one big buffer, then allocate our structures (separated by padding), and finally free the big buffer. Then we call the libwebp function.

Code listing 4.2 Arranging the heap; `mmap` and function pointer wrappers

```
typedef struct
{
  void (*fun)();
} FunctionWrapper;

typedef struct
{
  void* addr;
  size_t length;
  uint8_t* mapped_region;
  int fd;
} MMapWrapper;


void* tmp = malloc(0x1d8    // a
                   + 0x1000 // b
                   + 0x262  // c
                   + 0x2e28 // huffman_tables
                   + 0x2e); // additional alignment
FunctionWrapper* fw = malloc(sizeof(FunctionWrapper)); // 8
void* pad = malloc(0x50);
MMapWrapper* mmw = malloc(sizeof(MMapWrapper)); // 32
free(tmp);
```

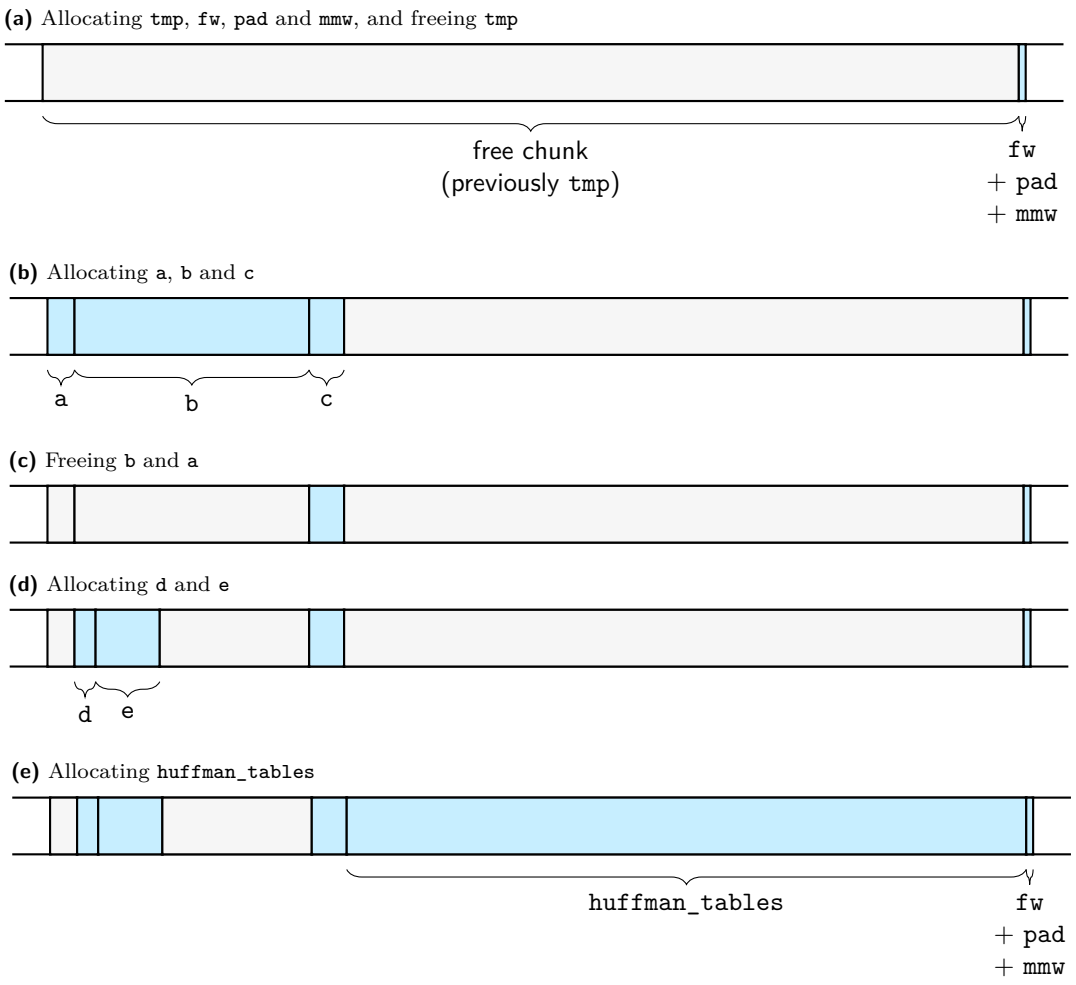
We show the effect of all these calls in Figure 4.1. We allocate `tmp`, `fw`, `pad` and `mmw`, all of which are placed next to each other. Then we free `tmp`. Allocating `a`, `b` and `c` places them in the free chunk. Next, we free `b` and `a`. As `a` is small enough, it gets placed into a special bin type called the *fast bin*. Free chunks in these bins are used only if the requested size by the allocation is very similar to the size of the free chunk, and they are not merged with neighboring free chunks. This is why `d` and `e` are placed in the free chunk `b`, even though `d` would fit in the free chunk `a`. Finally, we allocate the `huffman_tables` buffer. As it does not fit in the chunk `a` nor in the remaining space of the chunk `b`, it gets placed in the remaining space of the chunk `tmp`. This way, it is placed directly before our structures.

4.3 Creating the malicious image

Although we already described everything required to construct the malicious image, we summarize its structure here. We also provide a script for constructing this file in the attached files.

1. (32 bits) `"RIFF"` characters.
2. (32 bits) Total file size minus 8. We can put zeros here and fill it at the end.
3. (64 bits) `"WEBPVP8L"` characters.
4. (32 bits) Total file size minus 20. We can put zeros here and fill it at the end.
5. (8 bits) Signature value `0x2f`.

Figure 4.1 Arranging the heap



6. (14 bits) Width, zeros.
7. (14 bits) Height, zeros. This results in an image containing one pixel.
8. (1 bit) Alpha, zero.
9. (3 bits) Version, zeros.
10. (1 bit) Transform, zero.
11. (1 bit) Color cache, zero.
12. (1 bit) Meta prefix codes, zero.
13. (? bits) Prefix codes, encoded as we described previously. As long as the tree decoding the code lengths is complete, its shape can be arbitrary. Here we save the four complete trees of the maximum sizes from Table 3.4 and the last tree triggering the overflow from Table 4.1.
14. (0 bits) Encoded image. As the decoder detects an incomplete tree, it stops its execution, so further bits are not even read.

4.4 Creating the malicious file

The last thing we need to discuss is the creation of the malicious code that gets executed. We store this code in the file that is later loaded to memory by `mmap`. The function pointer contains the address `0x00270007`, and the pointer we pass to `mmap` holds the value `0x00260007`. `mmap` aligns this address and maps our file at the address `0x00260000`. This means our pointer starts executing the file from the `0x00010007`th byte. As nothing depends on the first bytes of the file, we start by placing `0x00010007` zero bytes in the file.

The bytes that follow describe the instructions that will get executed, and the attacker can choose any instructions they want. As an example, we present one such code.

Each instruction is encoded in bytes. These values are referred to as the *operation codes* (*opcodes* for short). We first figure out what instructions to use, and we convert them to their opcodes and append them to the file later.

We settled on using the `syscall` instruction. This instruction calls one of system functions. What functions can be called and how to pass their arguments is shown in [55]. The function we target is `sys_execve`. This function executes our specified program by replacing the current program. We pass it the path to the program, its arguments, and its environment. [56]

For us to execute the program, we must store the parameters somewhere. The most convenient way is to store them in the file as well, before or after the instructions. The path is a normal C-style string, i.e., ASCII-encoded characters terminated by a zero byte. The arguments are passed as an array of strings. This means we create the individual arguments as strings in the same way as we created the path. We then place the addresses of these strings next to each other and terminate them by a `NULL` pointer. The environment is created analogically. We know the file is placed at address `0x00260000`, so we calculate the exact addresses accordingly.

The `syscall` instruction takes all the required information from registers. We set the parameters to the corresponding registers, as described in [55]. The complete code, therefore, consists of five instructions:

1. Set `%rax` to 59, which selects the `sys_execve` call.
2. Set `%rdi` to the address of the path.
3. Set `%rsi` to the address of the arguments.
4. Set `%rdx` to the address of the environment.
5. Invoke the `syscall`.

This way, we can execute any program we want. As we do not call any other functions, nor do we require access to data in memory other than the parameters in the file, the Address Space Layout Randomization protection can remain active. In the attached files, we provide a script for building this file.

Conclusion

This thesis aimed to analyze the CVE-2023-4863 vulnerability in greater detail than has been done before, with a particular focus on the Huffman trees leading to buffer overflow. Based on the analysis, we discovered that code execution can indeed be achieved and prepared a vulnerable proof-of-concept application.

In the first chapter, we discussed buffer overflows in general. We described the process's layout and then introduced buffer overflows on the stack and the heap. Finally, we described common buffer overflow protection mechanisms.

In the second chapter, we summarized the available information about the CVE-2023-4863 vulnerability. Next, we presented the WebP file format. As the image is compressed by using Huffman trees, we explained them with attention to how they are implemented in the libwebp library. We finished this chapter by giving an overview of the functions responsible for the overflow.

In the third chapter, we analyzed the influence of the Huffman trees on the buffer overflow. We determined what bytes and how far behind the end of the buffer we can write.

In the last chapter, we explored the possibility of a code execution exploit. We created a Linux proof-of-concept application that calls the vulnerable libwebp function. By arranging the heap, we were able to make the library overwrite the content of two other structures. The first structure then created a memory segment with our malicious code, and the second executed it. For this exploit to work, we needed to make the memory segment with the malicious code executable, but other protective measures, such as ASLR, can remain active.

We conclude that, under certain circumstances, the vulnerability can be exploited to code execution. This only signifies its importance. Based on the overflow analysis we provided, future work can analyze whether existing applications that use the library can be exploited.

Bibliography

1. COMMON VULNERABILITIES AND EXPOSURES. *CVE-2023-4863* [online]. 2024. [visited on 2024-03-29]. Available from: `https://www.cve.org/CVERecord?id=CVE-2023-4863`.
2. BROOKSHEAR, J Glenn. *Computer Science: An Overview*. 11th ed. Upper Saddle River, NJ: Pearson, 2011. ISBN 9780132569033.
3. PYTHON. *Built-in Exceptions* [online]. 2024. [visited on 2024-03-23]. Available from: `https://docs.python.org/3/library/exceptions.html`.
4. PROSSIMO. *What is memory safety and why does it matter?* [online]. 2022. [visited on 2024-03-23]. Available from: `https://www.memorysafety.org/docs/memory-safety/`.
5. CPPREFERENCE. *Undefined behavior* [online]. 2023. [visited on 2024-03-23]. Available from: `https://en.cppreference.com/w/c/language/behavior`.
6. PYTHON. *Issue 42938: [security][CVE-2021-3177] ctypes double representation BoF* [online]. 2022. [visited on 2024-03-24]. Available from: `https://bugs.python.org/issue42938`.
7. OWASP FOUNDATION. *Buffer Overflow* [online]. 2024. [visited on 2024-03-29]. Available from: `https://owasp.org/www-community/vulnerabilities/Buffer_Overflow`.
8. BRYANT, Randal E; O'HALLARON, David R. *Computer Systems: A Programmer's Perspective*. 2nd ed. Upper Saddle River, NJ: Pearson, 2010. ISBN 9780136108047.
9. LINUX KERNEL. *Memory Management* [online]. [visited on 2024-04-03]. Available from: `https://www.kernel.org/doc/html/v5.8/x86/x86_64/mm.html`.
10. ANLEY, Chris; HEASMAN, John; LINDNER, Felix; RICHARTE, Gerardo. *The shellcoder's handbook*. 2nd ed. Chichester, England: John Wiley & Sons, 2007. ISBN 9780470080238.
11. CPPREFERENCE. *system* [online]. 2022. [visited on 2024-04-07]. Available from: `https://en.cppreference.com/w/c/program/system`.
12. CPPREFERENCE.COM. *signal* [online]. 2022. [visited on 2024-04-07]. Available from: `https://en.cppreference.com/w/c/program/signal`.
13. LINUX MANUAL PAGES. *libc(7)* [online]. 2023. [visited on 2024-04-09]. Available from: `https://man7.org/linux/man-pages/man7/libc.7.html`.
14. THE GNU C LIBRARY. *MallocInternals* [online]. 2022. [visited on 2024-04-09]. Available from: `https://sourceware.org/glibc/wiki/MallocInternals`.
15. GB_MASTER. *x86 Exploitation 101: heap overflows… unlink me, would you please?* [online]. 2014. [visited on 2024-04-10]. Available from: `https://gbmaster.wordpress.com/2014/08/11/x86-exploitation-101-heap-overflows-unlink-me-would-you-please/`.

16. LINUX KERNEL. *Documentation for /proc/sys/kernel/* [online]. 2009. [visited on 2024-04-11]. Available from: https://docs.kernel.org/admin-guide/sysctl/kernel.html.

17. GOOGLE. *An image format for the Web - WebP* [online]. 2024. [visited on 2024-03-23]. Available from: https://developers.google.com/speed/webp.

18. GOOGLE. *Compression Techniques - WebP* [online]. 2024. [visited on 2024-03-23]. Available from: https://developers.google.com/speed/webp/docs/compression.

19. ADOBE. *File formats in Adobe Photoshop* [online]. 2023. [visited on 2024-03-23]. Available from: https://helpx.adobe.com/photoshop/using/file-formats.html.

20. GIMP. *Getting Images out of GIMP* [online]. [visited on 2024-03-23]. Available from: https://docs.gimp.org/2.10/en/gimp-images-out.html.

21. BLENDER. *Supported Graphics Formats* [online]. 2024. [visited on 2024-03-23]. Available from: https://docs.blender.org/manual/en/latest/files/media/image_formats.html.

22. GOOGLE. *Frequently Asked Questions - WebP* [online]. 2024. [visited on 2024-03-23]. Available from: https://developers.google.com/speed/webp/faq.

23. COMMON VULNERABILITIES AND EXPOSURES. *Glossary* [online]. 2024. [visited on 2024-04-12]. Available from: https://www.cve.org/ResourcesSupport/Glossary.

24. COMMON VULNERABILITIES AND EXPOSURES. *Overview* [online]. 2024. [visited on 2024-04-12]. Available from: https://www.cve.org/About/Overview.

25. COMMON VULNERABILITIES AND EXPOSURES. *Process* [online]. 2024. [visited on 2024-04-12]. Available from: https://www.cve.org/About/Process.

26. COMMON VULNERABILITY SCORING SYSTEM. *CVSS v3.1 Specification Document* [online]. 2024. [visited on 2024-04-12]. Available from: https://www.first.org/cvss/v3.1/specification-document.

27. THE CITIZEN LAB. *BLASTPASS: NSO Group iPhone Zero-Click, Zero-Day Exploit Captured in the Wild* [online]. 2023. [visited on 2024-04-13]. Available from: https://citizenlab.ca/2023/09/blastpass-nso-group-iphone-zero-click-zero-day-exploit-captured-in-the-wild/.

28. THE CITIZEN LAB. *About the Citizen Lab* [online]. 2022. [visited on 2024-04-13]. Available from: https://citizenlab.ca/about/.

29. CHROMIUM. *Security vulnerability in WebP [40071416]* [online]. 2024. [visited on 2024-04-13]. Available from: https://issues.chromium.org/issues/40071416.

30. THE WEBP PROJECT. *Fix OOB write in BuildHuffmanTable.* [online]. 2023. [visited on 2024-04-13]. Available from: https://github.com/webmproject/libwebp/commit/902bc9190331343b2017211debcec8d2ab87e17a.

31. THE WEBP PROJECT. *Merge tag 'v1.3.2'* [online]. 2023. [visited on 2024-04-13]. Available from: https://github.com/webmproject/libwebp/commit/5fac76cf8d5bac94b79054a804d259d5e4d0056a.

32. COMMON VULNERABILITIES AND EXPOSURES. *CVE-2023-5129* [online]. 2024. [visited on 2024-04-14]. Available from: https://www.cve.org/CVERecord?id=CVE-2023-5129.

33. HAWKES, Ben. *The WebP 0day* [online]. 2023. [visited on 2024-04-14]. Available from: https://blog.isosceles.com/the-webp-0day/.

34. MISTYMNTNCOP. *mistymntncop/CVE-2023-4863* [online]. 2023. [visited on 2024-04-14]. Available from: https://github.com/mistymntncop/CVE-2023-4863/.

35. NIST. *CVE-2023-4863* [online]. 2024. [visited on 2024-04-14]. Available from: https://nvd.nist.gov/vuln/detail/CVE-2023-4863.

36. THE WEBP PROJECT. *Speed up Huffman decoding for lossless* [online]. 2014. [visited on 2024-04-13]. Available from: https://github.com/webmproject/libwebp/commit/f75dfbf23d1df1be52350b1a6fc5cfa6c2194499.

37. GOOGLE CHROME. *Chrome Releases: Stable Channel Update for Desktop* [online]. 2023. [visited on 2024-04-14]. Available from: https://chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop_11.html.

38. MOZILLA. *Mozilla Foundation Security Advisory 2023-40* [online]. 2023. [visited on 2024-04-14]. Available from: https://www.mozilla.org/en-US/security/advisories/mfsa2023-40/.

39. MICROSOFT. *Release notes for Microsoft Edge Security Updates* [online]. 2024. [visited on 2024-04-14]. Available from: https://learn.microsoft.com/en-us/deployedge/microsoft-edge-relnotes-security.

40. BRAVE. *Release Channel 1.57.64* [online]. 2023. [visited on 2024-04-14]. Available from: https://community.brave.com/t/release-channel-1-57-64/505527.

41. BRAVE. *CVE-2023-4863 Vulnerability* [online]. 2023. [visited on 2024-04-14]. Available from: https://github.com/brave/brave-browser/issues/33032.

42. OPERA. *Opera 102.0.4880.51 Stable update* [online]. 2023. [visited on 2024-04-14]. Available from: https://blogs.opera.com/desktop/2023/09/opera-102-0-4880-51-stable-update/.

43. THE TOR PROJECT. *New Release: Tor Browser 12.5.4* [online]. 2023. [visited on 2024-04-14]. Available from: https://blog.torproject.org/new-release-tor-browser-1254/.

44. GOOGLE. *WebP Container Specification* [online]. 2024. [visited on 2024-04-17]. Available from: https://developers.google.com/speed/webp/docs/riff_container.

45. GOOGLE. *Specification for WebP Lossless Bitstream* [online]. 2024. [visited on 2024-04-18]. Available from: https://developers.google.com/speed/webp/docs/webp_lossless_bitstream_specification.

46. HUFFMAN, David A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*. 1952, vol. 40, no. 9, pp. 1098–1101. Available from DOI: 10.1109/JRPROC.1952.273898.

47. THE WEBP PROJECT. *libwebp/src/utils/huffman_utils.c at 7ba44f80f3b94fc0138db159afea770ef06532a0* [online]. 2022. [visited on 2024-04-21]. Available from: https://github.com/webmproject/libwebp/blob/7ba44f80f3b94fc0138db159afea770ef06532a0/src/utils/huffman_utils.c.

48. THE WEBP PROJECT. *libwebp/src/utils/huffman_utils.h at 7ba44f80f3b94fc0138db159afea770ef06532a0* [online]. 2019. [visited on 2024-04-22]. Available from: https://github.com/webmproject/libwebp/blob/7ba44f80f3b94fc0138db159afea770ef06532a0/src/utils/huffman_utils.h.

49. THE WEBP PROJECT. *libwebp/src/dec/vp8l_dec.c at 7ba44f80f3b94fc0138db159afea770ef06532a0* [online]. 2023. [visited on 2024-04-22]. Available from: https://github.com/webmproject/libwebp/blob/7ba44f80f3b94fc0138db159afea770ef06532a0/src/dec/vp8l_dec.c.

50. THE WEBP PROJECT. *libwebp/src/webp/format_constants.h at 7ba44f80f3b94fc0138db159afea770ef06532a0* [online]. 2022. [visited on 2024-04-23]. Available from: https://github.com/webmproject/libwebp/blob/7ba44f80f3b94fc0138db159afea770ef06532a0/src/webp/format_constants.h.

51. Z-LIBRARY PROJECT. *zlib/examples/enough.c at v1.2.5* [online]. 2011. [visited on 2024-04-23]. Available from: `https://github.com/madler/zlib/blob/v1.2.5/examples/enough.c`.

52. MISTYMNTNCOP. *CVE-2023-4863/craft.c* [online]. 2023. [visited on 2024-04-27]. Available from: `https://github.com/mistymntncop/CVE-2023-4863/blob/main/craft.c`.

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

54. LINUX MANUAL PAGES. *mmap(2)* [online]. 2023. [visited on 2024-04-27]. Available from: `https://man7.org/linux/man-pages/man2/mmap.2.html`.

55. CHAPMAN, Ryan A. *Linux System Call Table for x86 64* [online]. 2012. [visited on 2024-04-28]. Available from: `https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/`.

56. LINUX MANUAL PAGES. *execve(2)* [online]. 2023. [visited on 2024-04-28]. Available from: `https://man7.org/linux/man-pages/man2/execve.2.html`.

Contents of attached files

- `README.txt` ..... the text file with description of the attached files
- `impl` ..... the directory with the implementation
  - `src` ..... source code of the proof-of-concept application and other scripts
  - `virt` ..... virtual environment containing the application and the scripts
- `thesis` ..... the directory with the thesis
  - `BP_Holy_Pavel_2024.pdf` ..... the thesis in PDF format
  - `src` ..... the LaTeX source files of the thesis