CS EE World
https://cseeworld.wixsite.com/home
May 2021
28/34
A
Submitter Info:
Name: Rajvardhan Agarwal
Email: me [at] r4j [dot] dev

IB Extended Essay

Title: Binary exploitation with modern exploit mitigations

Research question: To what extent is the GNU C library heap implementation on glibc version 2.31 more secure than version 2.15 and how has this affected its performance?

Subject: **Computer Science**
Word Count: **3811**
Session: **May 2021**

Table of Contents

1: Introduction 1
1.1: Scope and significance 1
1.2: Methodology 2
2: Processes on linux 2
2.1: The ELF executable 2
2.2: Memory Management 3
2.2.1: Stack 3
2.2.2: Heap 4
2.2.3: Mapped pages 4
2.2.4: Registers 4
2.3 Heap Freelists 5
3: Exploitation and Mitigations 6
3.1.1: Stack Buffer Overflow 6
3.1.2: Mitigations against stack buffer overflow 8
3.2: Heap 10
3.2.1: Use After Free 12
3.2.2: Double Free 12
3.2.3: Unsafe unlink 12
3.2.4: Heap buffer overflow 12
3.3: glibc heap exploitation techniques 13
3.3.1: Tcache poisoning 13
3.3.2: Tcache/Fastbin Dup 14
3.3.3: House of spirit 15
3.3.4: House of force 15
3.3.5: House of Einherjar 15
3.4: Comparison based on security 16
3.5: Performance benchmarking comparison 21
4: Conclusion 23
4.1: Propositions 23
4.2: Closing statements 23
4.3: Limitations and future scope 24
5: Works Cited 25

6: Appendix 25

Terminology Explained in simplified language with respect to the paper	
Mitigations	These are common techniques which are used to protect software applications against well-known exploits and vulnerabilities.
Executable	These files are programs which consist of encoded instructions to perform specific tasks.
Shared Library	A file containing object code which cannot be run directly but is usually linked to an executable.
Core Dump	A file containing a process's memory dump when it terminates unexpectedly.
Memory Page	A memory page is a contiguous block of virtual memory which is described by a single entry in the page table.
Physical Memory	The memory mapped to RAM.
Virtual Memory	A feature of the operating system which can be used to compensate for shortages in physical memory.
Userspace	The code that runs outside the context of the operating system's kernel.
System Call	A way for userspace programs to interact with the operating system's kernel.
Dereference	Access or manipulate data contained in a memory location pointed by a pointer.
Code Execution	An attacker's ability to execute arbitrary code on a target computer.
Buffer	A region of physical memory used to temporarily store data while it is being moved from one place to another.
Arbitrary Write	It is a condition where an attacker has the ability to write an arbitrary value at an arbitrary location in the memory.
Chunk	A fixed region of memory used to store user data. It also consists of meta-data which includes the size of the chunk.
Glibc	The GNU C library. It defines standard functions and system API which are used by programmers while developing software.

1: Introduction

Binary exploitation is the process which is used to exploit compiled applications. There are multiple ways to carry out such an attack. This generally deals with memory corruption bugs. It is well known that most of the low level/compiled languages give a lot of control to the programmer. Unlike high level programming languages, where the programmer has to only implement the code logic, low level languages require the programmer to implement everything. For example, the programmer is responsible for managing memory which is used by the program. Such low-level programming languages are very efficient because these can be directly converted into assembly language and run by the computer. This is one of the reasons why softwares such as operating systems and browsers still make use of low-level languages. Thus, logic errors in low level code are mis utilized to perform binary exploitation.

Such vulnerabilities have been exploited in the wild for decades. Several techniques are utilized to hijack the control flow of the programs. Exploitation often includes injection of shellcode into the running program or even make use of the machine instructions which are already present in the program. Developers have come up with several mitigation strategies to prevent memory corruption vulnerabilities. This makes the exploitation harder up to a certain extent but doesn't eradicate it completely.

1.1: Scope and significance

The aim of this essay is to analyze and model various stack and heap exploitation techniques on Linux. Furthermore, test its applicability with respect to the modern GNU C library. This essay also aims to showcase various mitigations that have been applied over the time to protect systems

from common attacks. It will be determined which techniques hold more relevance and are tough to mitigate.

1.2: Methodology

I will start by looking at the ELF (Executable and Linkable Format) executables, which is the format used by Linux to run executable files. Then, explain how memory management is done on Linux processes. I'll then explore various mitigations present on stack and heap implementations in the GNU C library. Various exploitation techniques used to bypass these mitigations will be discussed. The mitigations introduced in the two different glibc versions as mentioned in the research question, will be compared.

2: Processes on Linux

Processes are fundamental to any multitasking operating system. Various processes can run the same program separately. On the Linux kernel, a separate thread is created for each process. These processes might have different priorities and are handled respectively by the task scheduler.

2.1: The ELF executable

ELF is a cross-platform file format which is used for executables, shared libraries, object files and core dumps. This is the standard file format for the Linux operating system. This format consists of various segments which include the ELF header, the section header table and the program header table (Linux manual page).

2.2: Memory Management

Just like any other operating system, processes on Linux can allocate memory pages. This allocated memory might be mapped to RAM (as physical memory) or the on the disk (as virtual memory). As shown in figure 1 memory pages can generally be divided into three categories -

- Stack
- Heap
- Mapped pages

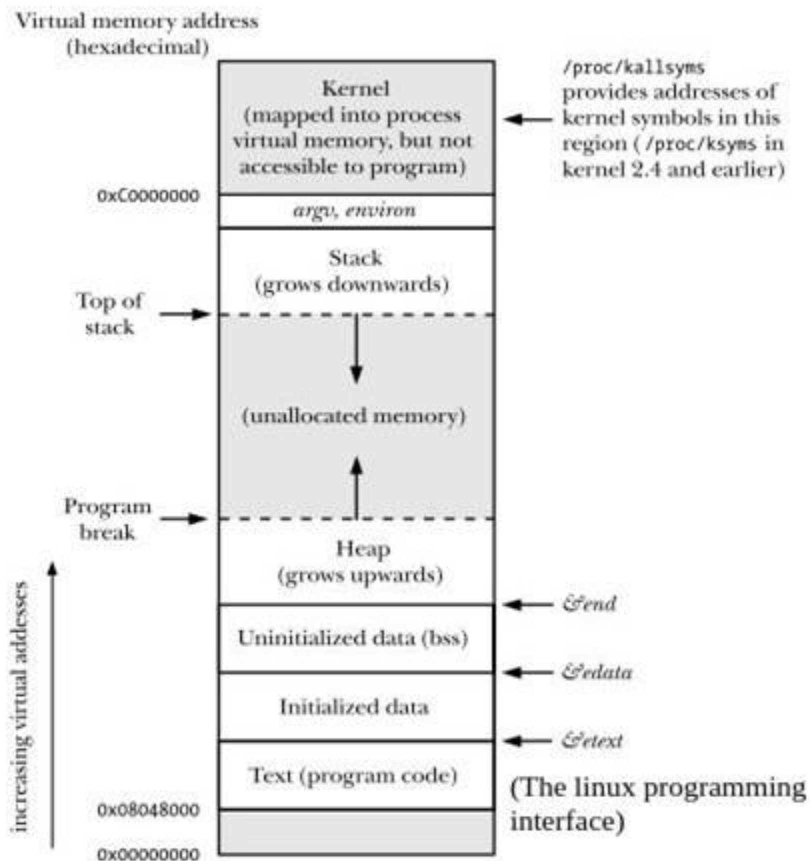

Figure 1: Memory layout in a linux process (The linux programming interface)

2.2.1: Stack

A memory page with a fixed small size is allocated for the stack. The stack follows last in first out order. The push instruction can be used to push data on the top of the stack. The pop instruction

can be used to pop data from the bottom of the stack. The stack is used to store data in a running process.

2.2.2: Heap

Heap is dynamically allocated memory which can be used by the program if a larger storage size is required. On userspace programs, heap memory is allocated through *brk* and *sbrk* system-calls. These system-calls are used by the *malloc()* function which is defined under the GNU C library. The data allocated through *malloc()* can be deallocated with the *free()* function. Usually, heap memory is directly stored on RAM.

2.2.3: Mapped pages

As it is known that, RAM has limited space which sometimes may not be enough. In such a scenario, mapped pages are allocated. This region of memory is backed by the disk storage and can be transferred to RAM when it is dereferenced. This memory management technique is also called virtual memory.

2.2.4: Registers

Registers store data which can be accessed by the processor very quickly. In general, there are various types of registers:

- **Memory Address Registers (MAR)**: These hold the address of the memory region to read data from or to write data.
- **Memory Data Registers (MDR)**: These hold the data to be written to the memory regions pointed by memory address registers.

- **General purpose registers**: There are multiple such registers present in the processor. These can be used to temporarily store data in the execution of a process.
- **Program Counter (PC)**: This register holds the address of the memory region which points to the next instruction to be executed. The value of this register is incremented after the execution of each instruction. The increment size depends on the instruction itself.
- **Instruction Register (IR)**: This register holds the instruction which is being executed currently. This is fetched from the program counter (Khushal).

2.3 Heap Freelists

Once freed, heap chunks are stored in singly/doubly linked lists. There are various types of free-lists depending on the properties of an allocated chunk.

- **Per-thread-cache list**

  This is a singly linked list which acts as a per-thread cache. It can store a maximum of 7 freed chunks at once. This free-list has the highest priority in the allocation of chunks. Addition and deletion happen in LIFO manner. There are 64 tcache bins for different sizes.
- **Fastbin**

  Just like tcahe, fastbins also use singly linked lists. Although, there is no limit on the maximum number of chunks a fastbin can store. Addition and deletion happen in LIFO manner. There is a total of 8 fastbins.
- **Unsorted Bin**

  Only one unsorted bin exists. Its main purpose is to act as a cache for allocation requests. Unlike other bins, this does not have a fixed size. Before being inserted to small/large bins, chunks are stored in the unsorted bin.

- **Small bin**

  Unlike tcache bins and fastbins, each smallbin maintains a doubly linked list. There is a total of 64 fastbins. Also, addition and deletion happen in FIFO (First in first out) manner. If an unsorted bin is present and an allocation request cannot be met by it, then it is transferred to a largebin or unsorted bin depending on the size.

- **Large bin**

  Large bins are stored in a doubly linked list. Unlike the other bins, this might have a range of sizes in the same bin which are sorted in a decreasing order. There are 65 fastbins in total. Insertion and deletion can happen at any position in the linked list for each large bin, depending on the size of the allocation request (Delorie).

3: Exploitation and Mitigations

The aim for exploitation will be to analyze and exploit common vulnerabilities and use them to gain code execution.

3.1.1: Stack Buffer Overflow

A stack-based buffer overflow is a very common vulnerability. This occurs when a stack buffer can be overwritten. This is generally due to a lack of bounds checking in the program. The following C program demonstrates a simple buffer overflow:

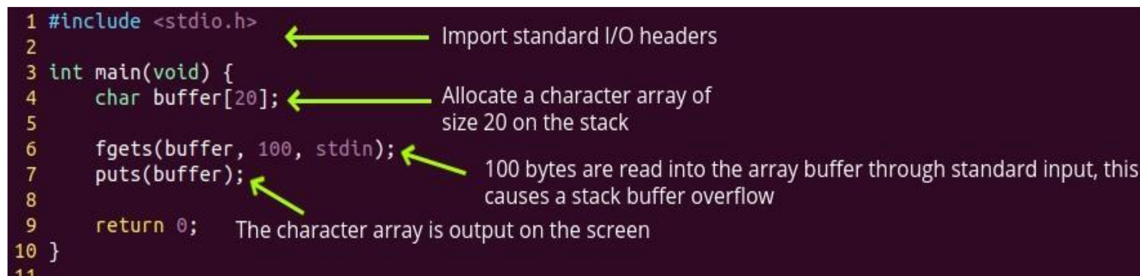

Figure 2: Sample C program demonstrating a buffer overflow

The program as shown in figure 2, allocates an array buffer of size 20 on the stack. The call to *fgets()* allows the user to input data on that buffer through standard input. As the input size is greater than 20, this can result in a stack overflow. This can be used to corrupt the metadata on the stack frame.

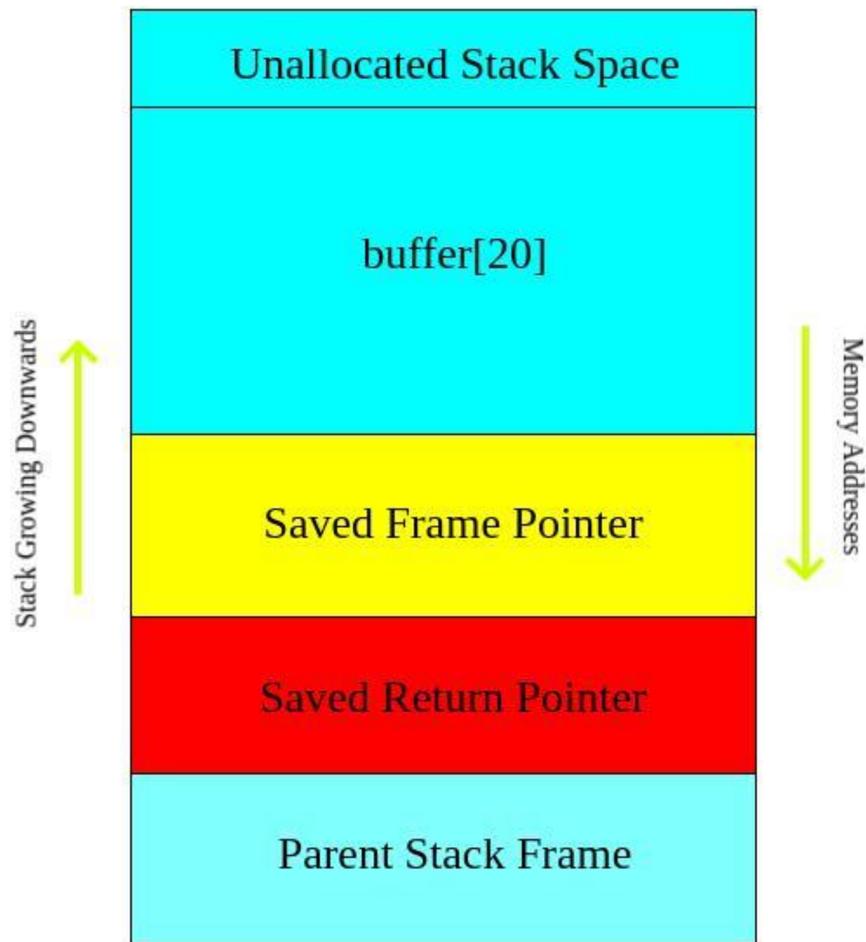

Figure 3: Stack frame for the main() function

Figure 3 visualizes the stack frame for the main function. As it can be deduced from the figure, the saved frame pointer and saved return pointer are stored contiguous to the array buffer. An attacker can use the stack buffer overflow to overwrite the saved return pointer (also known as program counter) and hijacking the control flow of the program. The saved return pointer can be overwritten

with an address to a controlled region in memory and arbitrary code execution can be achieved. This can be seen in figure 4.

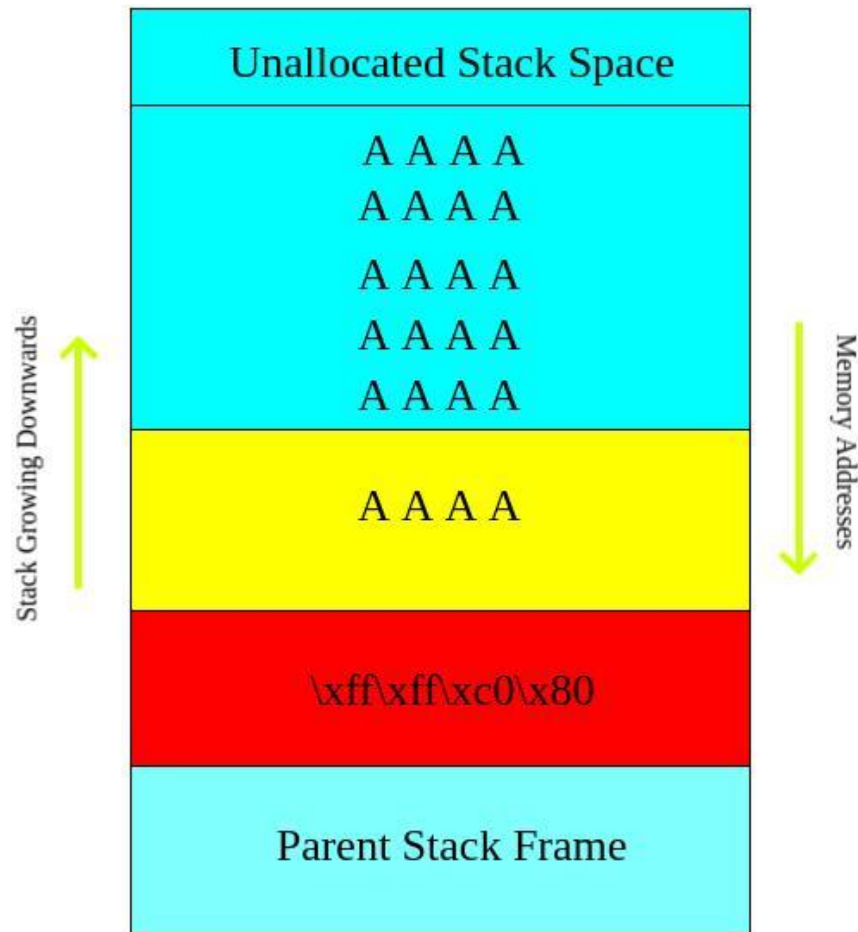

Figure 4: Stack frame after buffer overflow

3.1.2: Mitigations against stack buffer overflow

Due to wild exploitation of buffer overflows various mitigations were introduced against it. These include Stack Canary, NX, PIE (position independent executable), ASLR etc.

Stack Canary

Stack canary as seen in figure 5 is a mitigation which was introduced to protect against buffer overflow attacks. This mitigation stores a randomly generated value between the allocated buffer and the saved frame pointer. Before returning to the saved return pointer this value is checked

using XOR operation. The value on the stack is XORed with the original value of the stack canary. If the result of the XOR operation is not zero as it should be, an exception is raised and the program crashes immediately preventing an attacker from exploiting the program (Hawkins).

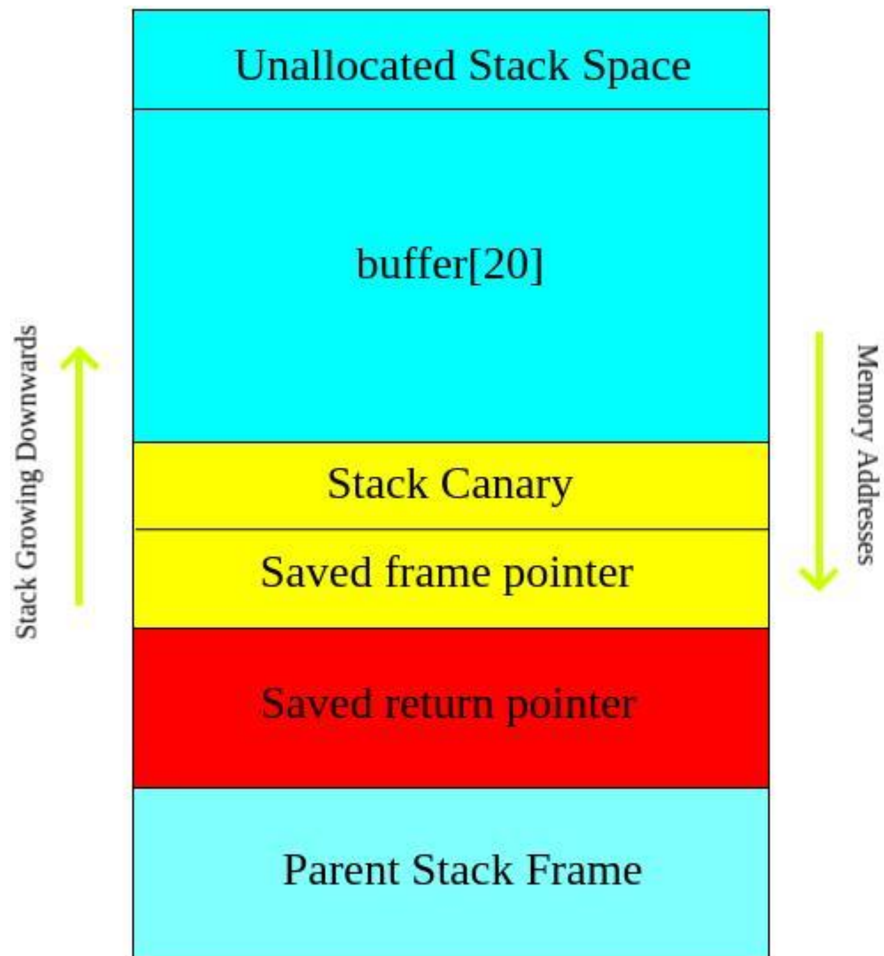

Figure 5: Demonstrating a stack canary

Weakness

This mitigation is ineffective when the value of the stack canary can be found. This can be the case if there are any memory leaks in the program.

No-Executable Bit

This NX bit is a mitigation that marks the user controlled regions (such as stack and heap) in the memory as non executable. It was added in various processors as a page table entry. This allows

the segregation of memory to be used either as code or as data storage. It ensures that even if the return pointer is hijacked, code execution is not possible. If the return pointer points to such a region, the program crashes.

Weakness

This mitigation prevents an attacker to completely take control over the execution flow of the program. But control over the program counter is still possible. This allows an attacker to reuse existing snippets of code from a program and chain them together to gain code execution. This attack method is known as Return-oriented programming, ROP (Shapiro).

ASLR

ASLR (Address Space Layout Randomization) is a mitigation technique which randomizes the address space of memory regions in a process. Due to this, addresses of memory regions are unknown. Thus, an attacker cannot trivially utilize techniques such as ROP (Marco-Gisbert).

Weakness

Just like in the case of stack canary, this mitigation technique is ineffective when memory leaks are present in the program. A memory leak may be used to calculate the addresses of memory regions.

3.2: Heap

As mentioned in 2.1.2, heap memory is dynamically allocated. Thus, a program can request or release memory from the heap region when required. The glibc malloc implementation is chunk oriented. This means that it allocates a huge region of memory at once and then it’s further divided into smaller chunks as per to the allocation requests. Each of these chunks contain meta-data which stores its properties. In general, there are two types of chunks: Allocated and freed.

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if
free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including
overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

Malloc chunk data structure definition (GNU libc source code)

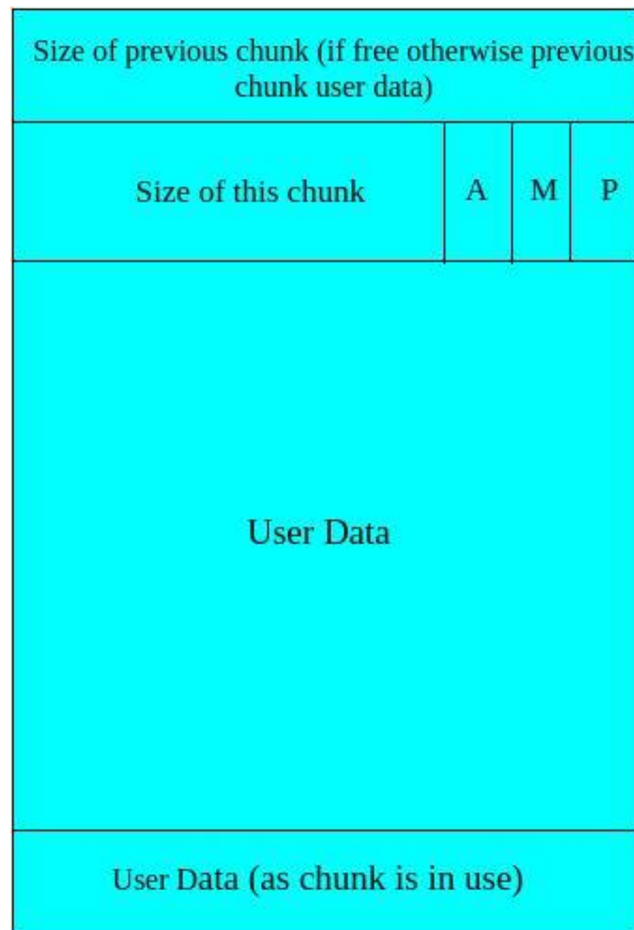

Figure 6: heap chunk in use

As shown in figure 6, the header of each used chunk consists of the size of the previous chunk and the current chunk. The size field stores three other properties which are bit-masked. When a chunk gets freed, it is stored in a free-list of its respective size. For smaller sizes it is stored in a singly linked list, while for larger sizes it is stored in a doubly linked list. The forward pointer and backward pointers in this case are called *fd* and *bk* respectively. This can be seen in figure 7.

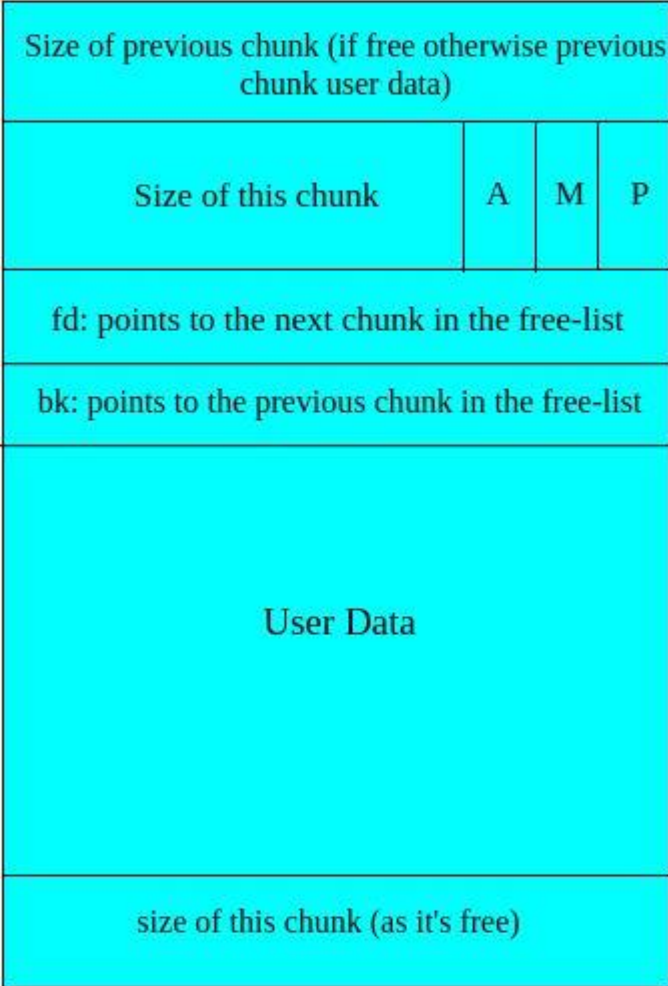

Figure 7: freed heap chunk

3.2.1: Use After Free

A use-after-free vulnerability is caused when the program frees an allocated chunk but it is still accessible by the user. Thereby, the term: use-after-free. This vulnerability can be used to corrupt

the chunk metadata or can be used for memory leaks. It might leave an attacker with powerful primitives, required for exploitation (Younan).

3.2.2: Double free

This vulnerability occurs when an allocated pointer is freed twice. This means that two consecutive calls to *malloc()* will return the same address. This has a similar effect to that of a use after free vulnerability. An attacker can use this to corrupt the chunk metadata.

3.2.3: Unsafe unlink

The unlink method is used for removing a node from a doubly linked list. In this case, if an attacker controls the *fd* and *bk* pointers for a freed chunk it can be used to gain an arbitrary write primitive. This can be further used to gain code execution (Kapil).

3.2.4: Heap buffer overflow

The causes of this vulnerability are the same as a stack buffer overflow as mentioned in 3.1.1. This allows an attacker to overflow into contiguous chunks and corrupt their metadata. There are near to no mitigations that fix this issue. It might help an attacker to gain various primitives which can lead to code execution.

3.3: glibc heap exploitation techniques

There are various known heap exploitation techniques specific to the glibc heap implementation. These utilize the vulnerabilities discussed in 3.2.

3.3.1: Tcache poisoning

Tcache as shown in figure 8 is a per-thread cache which can store a small number of chunks. These are stored in singly linked lists. There are different bins for each size depending on the alignment.

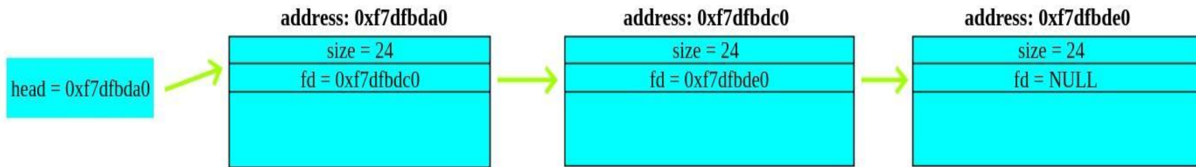

Figure 8: Sample tcache bin for size 24

As tcache is supposed to be fast, a lot of security checks were skipped. This gives rise to many attacks, one of those being *tcache poisoning*. This attack can be used when a use-after-free vulnerability is present. As shown in figure 9, it can be performed by corrupting the *fd* pointer in a freed chunk and can force malloc() to return a pointer to an arbitrary region (Itkin).

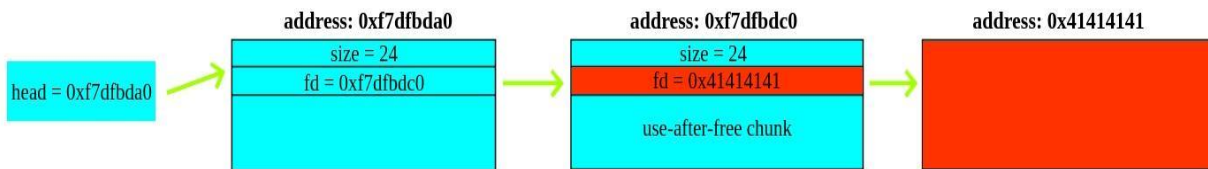

Figure 9: Tcache poisoning through use-after-free

3.3.2: Tcache/Fastbin Dup

Similar to tcache, fastbin is a singly linked list which contains freed chunks. There are a few differences between these two bin lists. Unlike tcache, fastbin does not have a limit in the number of chunks it can store. Fastbin also has extra security checks to ensure the validity of a chunk. Duping is a method to free the same chunk twice. This results in having a duplicate of the chunk in the freelist. This can be seen in figure 10.

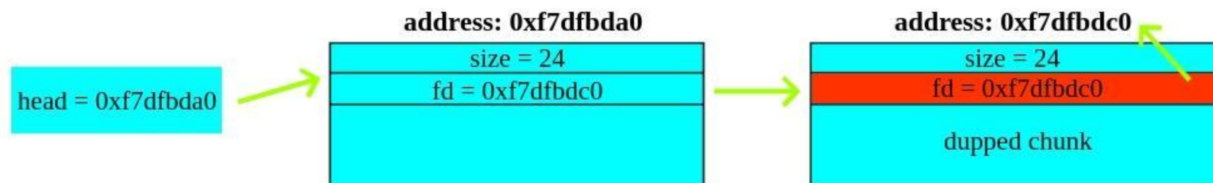

Figure 10: Tcache/fastbin dup

3.3.3: House of spirit

Unlike other heap vulnerabilities which require an overflow or a use-after-free primitive, this attack functions differently. It requires an attacker to have control over the pointer pointing to the heap region. This pointer can be overwritten to point to a non-heap region, such as stack or mapped memory. Once this pointer is freed with the *free()* function, it gets inserted into the free-list. For this to happen, the region must meet certain security checks. However, this is not difficult due to the structure of a glibc heap chunk, as only the size field of the current and the continuous chunk must be valid (Rørvik).

3.3.4: House of force

Just as in house of spirit, the end goal for this technique is to make *malloc()* return an arbitrary pointer. This requires a heap overflow and also the ability to fully control the size for allocation requests. The size of the top chunk is modified to a very large value such as -1. -1 being an unsigned integer, translates to 0xffffffffffffffff on 64 bit systems. A value this big can cover the entire address space if the program. Thus, if the address of the top chunk is known, this technique can be used to return an arbitrary pointer with *malloc()* (Kevin).

3.3.5: House of Einherjar

House of Einherjar is a very powerful glibc heap exploitation technique, which can be exploited in the very old glibc releases as well as the latest ones with slight modification. Just like other techniques, the end goal is to make *malloc()* return an arbitrary pointer in the memory. This technique requires only a single null byte overflow vulnerability, which is a common mistake made by programmers. As shown in figure 7, the PREV_IN_USE flag is bit masked in the size field of

a chunk. This technique uses the overflow to clear this flag. As the flag is now cleared, the prev_size field is used and can be controlled by the attacker. When the next chunk gets freed, the heap allocator finds that the previous chunk is not in use. Thus, it consolidates the previous chunk with the current chunk to prevent fragmentation on the heap. The size of the previous chunk is defined in the prev_size field, which is controlled by the attacker. The attacker can use this primitive to free arbitrary heap chunks which are contiguous to the chunk being freed and cause a use-after-free condition. This can be further exploited trivially to gain code execution. This technique is very powerful as it works on almost all versions of the glibc and can be used to gain powerful primitives.

3.4: Comparison based on security

One of the major differences between both the glibc versions is the introduction of tcache freelist. Although tcache makes the allocations faster, it does this at the cost of security. While in some cases the tcache implementation presents better mitigations, it makes exploitation easier otherwise. The following structure presents a freed tcache chunk.

```
typedef struct tcache_entry
{
  struct tcache_entry *next;
  struct tcache_perthread_struct *key;
} tcache_entry;
```

Double free detection in tcache

As the tcache list is based on a singly linked list, the bk field is not used. This is thus used as a *key*. This points to the tcache_perthread_struct for the tcache list of the respective chunk size. This value is set when the chunk is already in the tcache free-list. Due to ASLR, it is not possible for an attacker to know the value of tcache_perthread_struct as it changes on each run. When an

allocation request is made and the tcache_entry is deleted from the list, the key value is cleared. Due to this, if a chunk is already in the tcache list, and it is freed again through a dangling pointer, the key value will be already present in the chunk. This will trigger a further check and the tcache list will be iterated to check if this chunk already exists in the list. If it does, the program will abort to prevent exploitation.

Weakness

This mitigation solely depends on whether the key value points to the tcache_perthread_struct. Thus, it can be overcome with the help of a use-after-free vulnerability where the value of the key is overwritten. As a result, the check is never triggered. However, these two vulnerabilities rarely co-exist.

Figure 11 and 12 show the implementation of the described technique (glibc 2.31):

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#define TCACHE_MAX_BINS 7

typedef struct tcache_entry
{
  struct tcache_entry *next;
  /* This field exists to detect double frees.  */
  struct tcache_perthread_struct *key;
} tcache_entry; // definition of tcache_entry struct

typedef struct tcache_perthread_struct
{
  char counts[TCACHE_MAX_BINS];
  tcache_entry *entries[TCACHE_MAX_BINS];
} tcache_perthread_struct; // definition of tcache_perthread_struct

int main(void) {
    void * ptr = malloc(0x18);
    free(ptr); // Inserted into tcache list

    tcache_entry * e = (tcache_entry *)ptr; // cast void pointer to tcache_entry pointer
    e->key = NULL; // clear tcache key
    free(ptr); // Double free

    printf("First malloc(): %p\n", malloc(0x18)); // returns the address of the first chunk that was freed into tcache list
    printf("Second malloc(): %p\n", malloc(0x18)); // returns the same address as the first malloc due to double free
}
```

Figure 11: Tcache double free check bypass

```
→ /tmp ./double_free
First malloc(): 0x55f7258512a0
Second malloc(): 0x55f7258512a0
→ /tmp
```

Both calls return the same address

Figure 12: Tcache double free demonstration

Double free detection in fastbin

Unlike the double free check in the tcache list, fastbin doesn't use a key to detect double free. Instead, it just compares the pointer being freed to the head of the linked list. Thus, if the same chunk is freed right after the previous free, it will trigger the double free abort.

Weakness

Only the head of the fastbin linked list is checked for the double free. As the whole linked list isn't traversed, a double free attack can be performed by freeing another chunk before the double free. This can be seen in figure 13 and 14.

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

int main(void) {
    void * ptr = malloc(0x18);
    void * ptr2 = malloc(0x18);

    free(ptr); // ptr gets freed into fastbin
    free(ptr2); // gets freed into fastbin
    free(ptr); // double free as only fastbin top is checked

    printf("First malloc(): %p\n", malloc(0x18)); // returns address of the chunk that was first put into fastbin
    printf("Second malloc(): %p\n", malloc(0x18)); // returns second chunk from fastbin
    printf("Third malloc(): %p\n", malloc(0x18)); // return address of first chunk again
}
```

Figure 13: Fastbin double free check bypass

```
root@8870a7681035:/# cd /tmp/
root@8870a7681035:/tmp# ./double_free
First malloc(): 0x556608ebf010
Second malloc(): 0x556608ebf030
Third malloc(): 0x556608ebf010
```

First and third malloc calls return the same address

Figure 14: Fastbin double free example

Freelist poisoning in tcache

```
/* Caller must ensure that we know tc_idx is valid and there's
   available chunks to remove.  */
static __always_inline void *
tcache_get (size_t tc_idx)
{
  tcache_entry *e = tcache->entries[tc_idx];
  tcache->entries[tc_idx] = e->next;
  --(tcache->counts[tc_idx]);
  e->key = NULL;
  return (void *) e;
}
```

Figure 15: tcache_get method

As it can be seen from the code in figure 15, *malloc()* uses the *tcache_get()* method internally to get chunks from the tcache list. Here, no checks are being made. Thus, if an attacker controls the next field with a use-after-free they will be able to control the pointer returned by *tcache_get()* and hence *malloc()*.

Freelist poisoning in fastbin

```
for (i = 0; i < NFASTBINS; ++i)
  {
    p = fastbin (av, i);

    /* The following test can only be performed for the main arena.
       While mallopt calls malloc_consolidate to get rid of all fast
       bins (especially those larger than the new maximum) this does
       only happen for the main arena.  Trying to do this for any
       other arena would mean those arenas have to be locked and
       malloc_consolidate be called for them.  This is excessive.  And
       even if this is acceptable to somebody it still cannot solve
       the problem completely since if the arena is locked a
       concurrent malloc call might create a new arena which then
       could use the newly invalid fast bins.  */

    /* all bins past max_fast are empty */
    if (av == &main_arena && i > max_fast_bin)
      assert (p == 0);

    while (p != 0)
      {
        /* each chunk claims to be inuse */
        do_check_inuse_chunk (av, p);
        total += chunksize (p);
        /* chunk belongs in this bin */
        assert (fastbin_index (chunksize (p)) == i);
        p = p->fd;
      }
```

Check if the inuse bit is set in the size field of next chunk

Size check to ensure chunk is from the same bin

Figure 16: fastbin_get method

The general strategy for fastbin poisoning is similar to that of tcache. It requires control over the fd pointer after a chunk is freed. Fastbin, just like tcache, is based on a singly linked list. If the fd pointer of a freed chunk is overwritten, a subsequent call to *malloc()* will return a pointer to this address. However, unlike tcache there are extra checks on the chunk metadata before the chunk is returned, as seen in figure 16. These metadata checks significantly reduce the impact of a use-after-free bug. While, tcache poisoning provides an attacker with a primitive to make *malloc()* return a nearly arbitrary pointer, fastbin limits this ability by only returning chunks which follow the same structure as a fastbin chunk for the specified size. This forces an attacker to either find or create “fake chunks” in the memory such that the fd pointer points to it.

3.5: Performance benchmarking comparison

Performance is an important ground for the comparison between the efficiency of the two implementations. This benchmarking will be performed on various devices. Various allocation sizes will be tested for each device.

Hypothesis

In glibc version 2.26 a freelist known as *tcache* was introduced. It is a per-thread cache with a limited number of entries which serves allocation sizes less than 1024. This has the highest priority when an allocation is served. Thus, the performance is expected to be better in glibc 2.31 while there should not be much of a difference in larger allocation sizes.

Results

#1) Processor: Intel i7-10750H
Device name: Lenovo legion 5 laptop
CPU count: 6 (12 threads)
Processor Speed: 3200MHz
Operating System: Ubuntu 20.04 LTS (based on linux kernel version 5.4.0)

Runtime for single process 10,000,000 requests between 0 to 1024 bytes

Test no.	Glibc 2.15 time in seconds	Glibc 2.31 time in seconds
1	0.382785	0.020909
2	0.388431	0.052439
3	0.378810	0.017607
4	0.383213	0.018688
5	0.379097	0.051745
Average	0.3824672	0.0322776

Runtime for 12 processes 10,000,000 requests between 0 to 1024 bytes

Test no.	Glibc 2.15 time in seconds (average of all processes)	Glibc 2.31 time in seconds (average of all processes)

1	0.5984496	0.1309238
2	0.6059243	0.12632841
3	0.606181	0.11436658
4	0.592927	0.1234309
5	0.606477	0.1244845
Average	0.60199178	0.12390683

Runtime for single process 10,000,000 requests between 0 to 8192 bytes

Test no.	Glibc 2.15 time in seconds	Glibc 2.31 time in seconds
1	0.309495	0.296383
2	0.319182	0.292292
3	0.316961	0.302637
4	0.314434	0.301698
5	0.317795	0.301023
Average	0.315573	0.2988066

Runtime for 12 processes 10,000,000 requests between 0 to 8192 bytes

Test no.	Glibc 2.15 time in seconds (average of all processes)	Glibc 2.31 time in seconds (average of all processes)
1	0.534113	0.52165375
2	0.5106381	0.51852641
3	0.51503525	0.5148141
4	0.5121525	0.521318
5	0.4939454	0.52281108
Average	0.51317685	0.519824668

Result Analysis

As it can be seen in the results, glibc 2.31 performs much better than 2.15 for smaller allocations. This provides evidence that the hypothesis holds true. There are very small time differences for larger allocations. However, these might have been caused due to the random errors that were present during the testing.

The introduction of tcache brings in significant improvement in the performance of the heap allocator.

4: Conclusion

4.1: Propositions

The glibc 2.31 release brings with itself a significant increase in the performance of the heap allocator. This also includes various security fixtures and mitigations. However, the introduction of tcache makes exploitation much easier in various conditions. It has been seen that a lot of exploits have utilized tcache for successful exploitation. Thus, the implementation of tcache must be reviewed under future releases and the required security checks should be introduced.

4.2: Closing statements

After studying the implementation and comparing the benchmarks of the heap allocator in both the glibc releases, it is evident that version 2.31 shows a better performance. It also introduces various mitigations that were absent in the prior release. It fixes trivial double free attacks which were widely exploited previously. However, in the implementation for tcache it makes a trade-off between performance and security. It choses performance over security by removing most security checks which were present in the prior fastbin implementation. This issue needs to be resolved in the future releases for improved security. This essay utilized various research papers,

documentations and security-related websites to limit the chances of citing inaccurate information. Hence to answer my research question, “To what extent is the GNU C library heap implementation on glibc version 2.31 more secure than version 2.15 and how has this affected its performance?”, the new glibc version 2.31 has made improvements in performance but has barely added any mitigating measures for common security vulnerabilities. A lot of these vulnerabilities can be addressed by introducing freelist hardening measures while having almost no impact on the performance (Itkin). These might be introduced in future releases of the glibc.

4.3: Limitations and future scope

The techniques discussed in this essay require a good control over the heap allocation and free requests. This might not always be possible in real life scenarios. There are various heap exploitation techniques based on smallbins and largebins, which were exploitable in glibc 2.15 but are no longer exploitable in glibc 2.31. However, these go beyond the scope of this essay and hence were not discussed. Applications such as browsers, sometimes use their own heap implementations (Ex. chrome uses tcmalloc). Thus, glibc heap exploitation techniques can be completely disregarded in such scenarios. The GNU C library is undergoing constant development. Thus, tcache being an experimental feature might be removed in future releases.

5: Works Cited

Agarwal, Khushal. “Different Classes of CPU Registers.” *GeeksforGeeks*, 2019, www.geeksforgeeks.org/different-classes-of-cpu-registers/.

Backhouse, Kevin. “Last Orders at the House of Force - GitHub Security Lab.” *Security Lab*, 18 June 2020, securitylab.github.com/research/last-orders-at-the-house-of-force.

Delorie, DJ. “Overview of Malloc.” *MallocInternals - Glibc Wiki*, 10 May 2019, sourceware.org/glibc/wiki/MallocInternals.

Di Federico, Alessandro, et al. "How the {ELF} Ruined Christmas." *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015.

Elf(5) - Linux Manual Page, Man Pages, 2020, man7.org/linux/man-pages/man5/elf.5.html.

Goichon, François. "Glibc adventures: The forgotten chunks." *Proc. Context Inf. Secur.*. Vol. 28. 2015.

Hawkins, William H et al. "Dynamic canary randomization for improved software security." *Proceedings of the 11th Annual Cyber and Information Security Research Conference*. 2016.

Itkin, Eyal. “Safe-Linking - Eliminating a 20 Year-Old Malloc() Exploit Primitive.” *Check Point Research*, 10 Aug. 2020, research.checkpoint.com/2020/safe-linking-eliminating-a-20-year-old-malloc-exploit-primitive/.

Kapil, Dhaval. “Bins and Chunks.” *Heap*, 2020, heap-exploitation.dhavalkapil.com/diving_into_glibc_heap/bins_chunks.

Lever, Chuck, and David Boreham. “Center for Information Technology Integration.” *Citi*, Nov. 1999, www.citi.umich.edu/projects/linux-scalability/reports/malloc.html.

Marco-Gisbert, Hector, and Ismael Ripoll. "On the Effectiveness of Full-ASLR on 64-bit Linux." *Proceedings of the In-Depth Security Conference*. 2014.

Markstedter, Maria. “Heap Exploitation Part 1: Understanding the Glibc Heap Implementation.” *Azeria*, 2019, azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/.

Rørvik, Mathias Frits. *Investigation of x64 GLIBC heap exploitation techniques on Linux*. MS thesis. 2019.

Shapiro, Rebecca et al. "“Weird Machines” in {ELF}: A Spotlight on the Underappreciated Metadata." *7th {USENIX} Workshop on Offensive Technologies ({WOOT} 13)*. 2013.

Younan, Yves. "FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers." *NDSS*. 2015.

Yun, Insu, et al. "Automatic techniques to systematically discover new heap exploitation primitives." *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2020.

Zhang, Chao, et al. "SecGOT: Secure global offset tables in ELF executables." *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press, 2013.

6: Appendix

Code used to benchmark malloc (Lever)

```
/*
 *  malloc-test
 *  cel - Thu Jan  7 15:49:16 EST 1999
 *
 *  Benchmark libc's malloc, and check how well it
 *  can handle malloc requests from multiple threads.
 *
 *  Syntax:
 *  malloc-test [ size [ iterations [ thread count ]]]
 *
 */

#include stdio.h
#include stdlib.h
#include sys/time.h
#include unistd.h

#include pthread.h

#define USECSPERSEC 1000000
#define pthread_attr_default NULL
#define MAX_THREADS 50

void run_test(void);
void * dummy(unsigned);

static unsigned size = 512;
static unsigned iteration_count = 1000000;

int main(int argc, char *argv[])
{
        unsigned i;
        unsigned thread_count = 1;
        pthread_t thread[MAX_THREADS];

        /*
         * Parse our arguments
         */
        switch (argc) {
        case 4:
                /* size, iteration count, and thread count were specified */
                thread_count = atoi(argv[3]);
                if (thread_count > MAX_THREADS) thread_count = MAX_THREADS;
        case 3:
                /* size and iteration count were specified; others default
*/
                iteration_count = atoi(argv[2]);
        case 2:
                /* size was specified; others default */
                size = atoi(argv[1]);
        case 1:
                /* use default values */
                break;
```

```
        default:
                printf("Unrecognized arguments.\n");
                exit(1);
        }

        /*
         * Invoke the tests
         */
        printf("Starting test...\n");
        for (i=1; i<=thread_count; i++)
                if (pthread_create(&(thread[i]), pthread_attr_default,
                                   (void *) &run_test, NULL))
                        printf("failed.\n");

        /*
         * Wait for tests to finish
         */
        for (i=1; i<=thread_count; i++)
                pthread_join(thread[i], NULL);

        exit(0);
}

void run_test(void)
{
        register unsigned int i;
        register unsigned request_size = size;
        register unsigned total_iterations = iteration_count;
        struct timeval start, end, null, elapsed, adjusted;

        /*
         * Time a null loop.  We'll subtract this from the final
         * malloc loop results to get a more accurate value.
         */
        gettimeofday(&start, NULL);
         srand(0xdeadbeef);

        for (i = 0; i < total_iterations; i++) {
                register void * buf;
                buf = dummy(i);
                buf = dummy(i);
        }

        gettimeofday(&end, NULL);

        null.tv_sec = end.tv_sec - start.tv_sec;
        null.tv_usec = end.tv_usec - start.tv_usec;
        if (null.tv_usec < 0) {
                null.tv_sec--;
                null.tv_usec += USECSPERSEC;
        }

        /*
         * Run the real malloc test
         */
        gettimeofday(&start, NULL);
```

```
        for (i = 0; i < total_iterations; i++) {
                register void * buf;
                buf = malloc(rand() % (request_size+1));
                free(buf);
        }

        gettimeofday(&end, NULL);

        elapsed.tv_sec = end.tv_sec - start.tv_sec;
        elapsed.tv_usec = end.tv_usec - start.tv_usec;
        if (elapsed.tv_usec < 0) {
                elapsed.tv_sec--;
                elapsed.tv_usec += USECSPERSEC;
        }

        /*
         * Adjust elapsed time by null loop time
         */
        adjusted.tv_sec = elapsed.tv_sec - null.tv_sec;
        adjusted.tv_usec = elapsed.tv_usec - null.tv_usec;
        if (adjusted.tv_usec < 0) {
                adjusted.tv_sec--;
                adjusted.tv_usec += USECSPERSEC;
        }
        printf("Thread %d adjusted timing: %d.%06d seconds for %d requests"
                " of %d bytes.\n", pthread_self(),
                adjusted.tv_sec, adjusted.tv_usec, total_iterations,
                request_size);

        pthread_exit(NULL);
}

void * dummy(unsigned i)
{
        return NULL;
}
```