

Bugs to Detect in Source Code Analysis

- Some examples

- **Crash Causing Defects**
- **Null pointer dereference**
- **Use after free**
- **Double free**
- **Array indexing errors**
- **Mismatched array new/delete**
- **Potential stack overrun**
- **Potential heap overrun**
- **Return pointers to local variables**
- **Logically inconsistent code**
- **Uninitialized variables**
- **Invalid use of negative values**
- **Passing large parameters by value**
- **Under allocations of dynamic data**
- **Memory leaks**
- **File handle leaks**
- **Network resource leaks**
- **Unused values**
- **Unhandled return codes**
- **Use of invalid iterators**

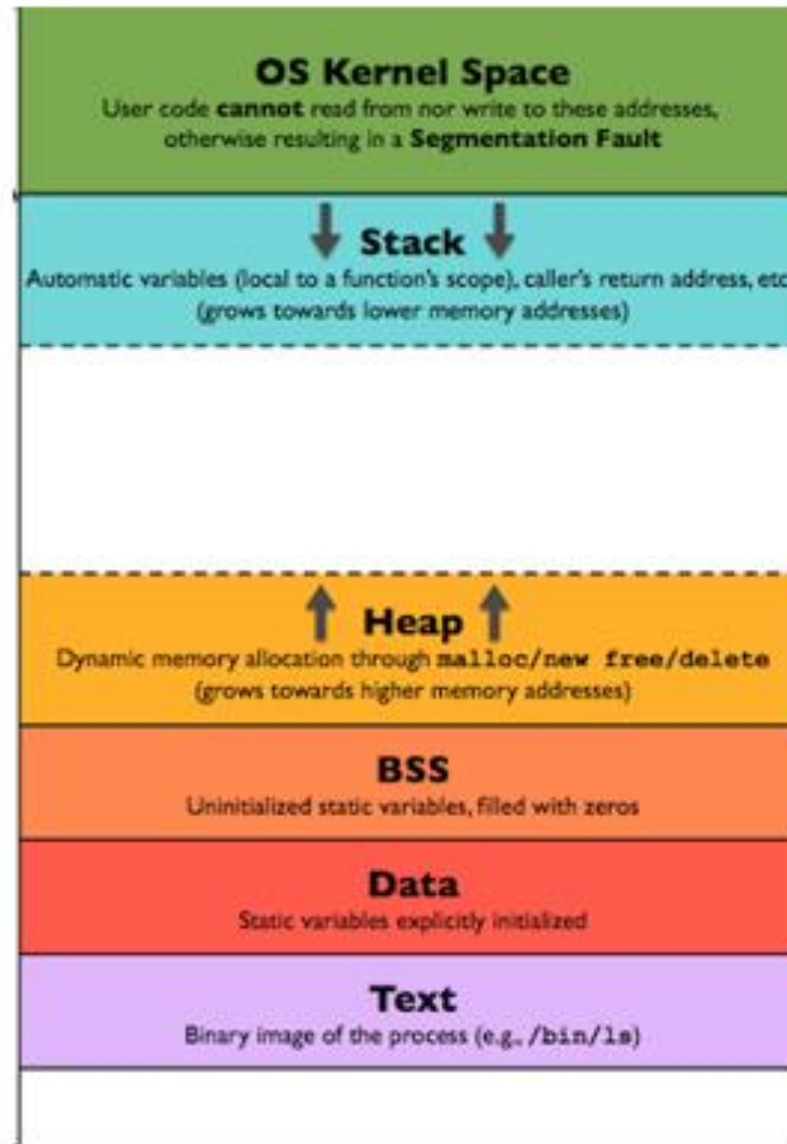
A bug is an unexpected problem that causes software to behave unpredictably or incorrectly. A vulnerability is a mistake in software that can be used by a hacker to gain access to a system or network.

Buffer overflow / overrun

Outline

- Stack Overruns
- Heap Overruns
- Array Indexing Errors
- Format String Bugs
- Unicode and ANSI Buffer size Mismatches
- Preventing Buffer Overruns
- Summary

Program's memory layout



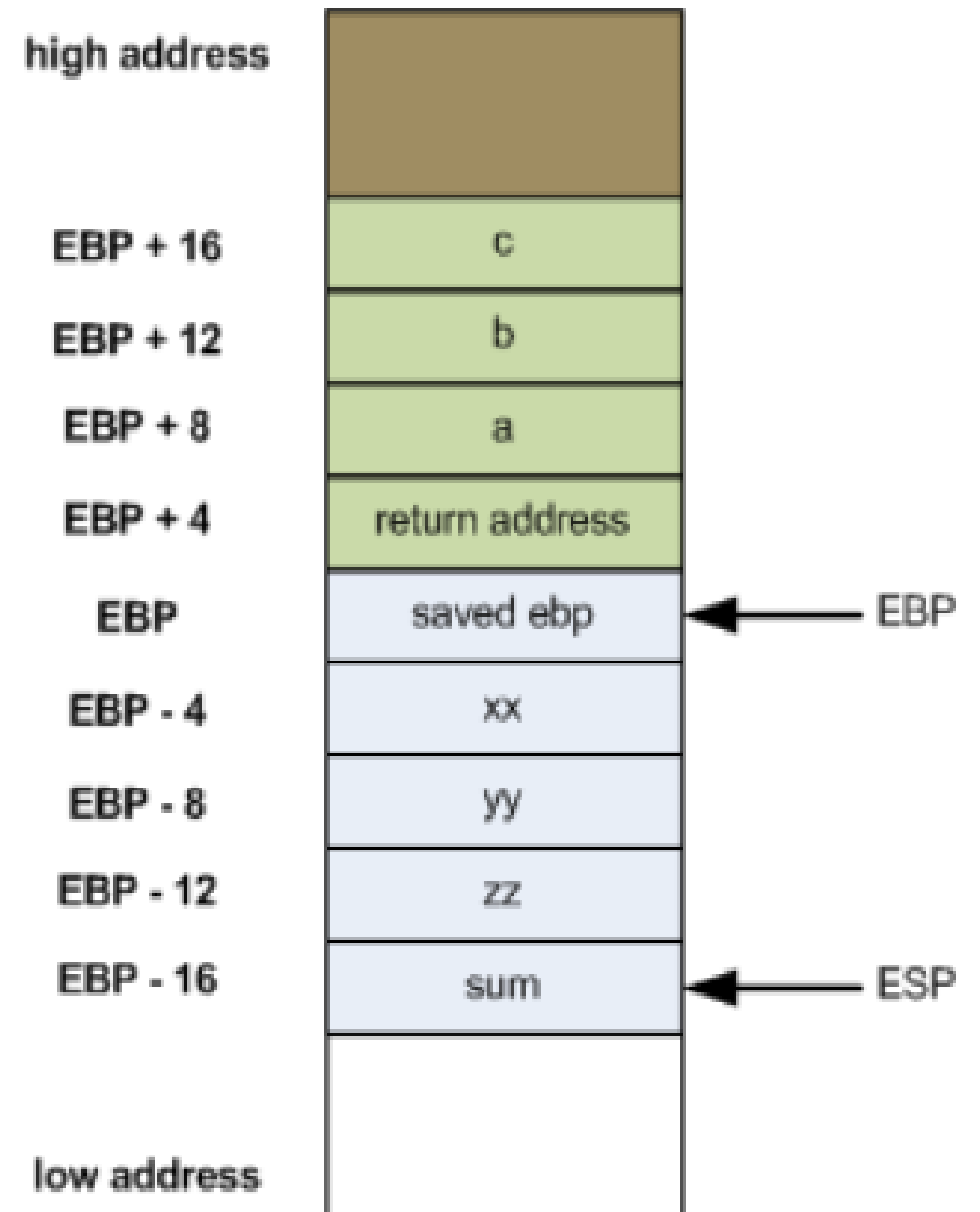
Memory Layout

```
int foobar(int a, int b, int c)
{
    int xx = a + 2;
    int yy = b + 3;
    int zz = c + 4;
    int sum = xx + yy + zz;
    return xx * yy * zz + sum;
}

int main()
{
    return foobar(77, 88, 99);
}
```

ESP (Extended Stack Pointer)

EBP - base pointer, also known as frame pointer



What is Buffer Overflow?

- A **buffer overflow**, or **buffer overrun**, is an anomalous condition where a process attempts to store data beyond the boundaries of a fixed-length buffer.
- The result is that the extra data overwrites adjacent memory locations. The overwritten data may include **other buffers**, **variables** and **program flow data**, and may result in erratic program behavior, **a memory access exception**, program termination (a crash), **incorrect results** or — especially if deliberately caused by a malicious user — a possible breach of system security.
- Most common with C/C++ programs

History

- Used in 1988's Morris Internet Worm
- Alphe One's "Smashing The Stack For Fun And Profit" in Phrack Issue 49 in 1996 popularizes stack buffer overflows
- Still extremely common today

<https://www.bleepingcomputer.com/news/security/android-october-security-update-fixes-zero-days-exploited-in-attacks/>

Libwebp is a popular library used to render webp images. This library is part of almost all modern operating systems and software platforms, including Apple iOS and Chromium based products like Google Chrome browser, Electron Software Framework, Debian (Ubuntu, Alpine), CentOS, Gentoo, SUSE and MacOS.

What Happens in a Function Call?

```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    ...  
}  
  
int main() {  
    func("abc");  
}
```

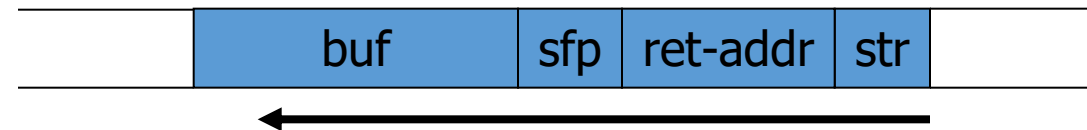
- Before main() calls func()
 - Push pointer to “abc” onto stack
 - Use “call func” assembly, which pushes current IP on stack
- Upon entering func()
 - Push stack frame pointer register (bp) on stack
 - Update sp to leave space for local variable.
- Upon leaving func()
 - Update sp to just below saved bp
 - Pop stack to bp, restore bp
 - Use “ret” assembly, which pop stack to IP

What are buffer overflows?

- Suppose a web server contains a function:

```
void func(char *str) {  
    char buf[128];  
    strcpy(buf, str);  
    do-something(buf);  
}
```

- When the function is invoked the stack looks like:

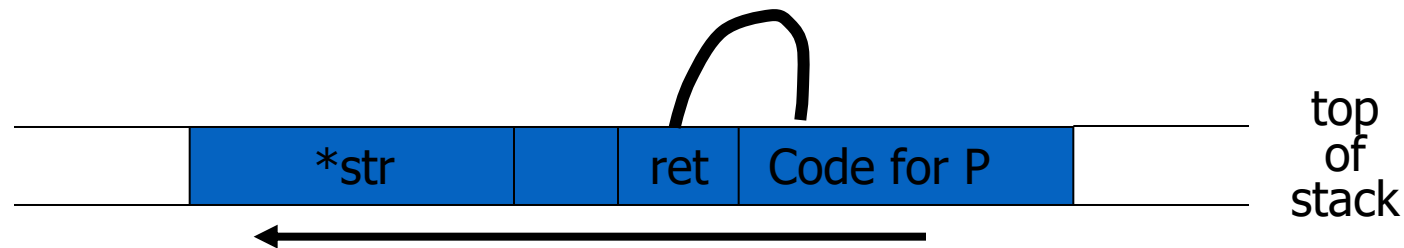


- What if ***str** is 136 bytes long? After **strcpy**:



Basic stack exploit

- Main problem: no range checking in `strcpy()`.
- Suppose `*str` is such that after `strcpy` stack looks like:



Program P: `exec("/bin/sh")`

- When `func()` exits, the user will be given a shell !!
- Note: attack code runs *in stack*.

Carrying out this attack requires

- Determine the location of injected code position on stack when func() is called.
 - So as to change stored return address on stack to point to it
 - Location of injected code is fixed relative to the location of the stack frame
- Program P should not contain the '\0' character.
 - Easy to achieve
- Overflow should not crash program before func() exits.

Some unsafe C lib functions

`strcpy (char *dest, const char *src)`

`strcat (char *dest, const char *src)`

`gets (char *s)`

`scanf (const char *format, ...)`

`sprintf (const char *format, ...)`

▪
▪
▪
▪
▪
▪

The Problem

```
void foo(char *s) {  
    char buf[10];  
    strcpy(buf, s);  
    printf("buf is %s\n", s);  
}  
...  
foo("thisstringistolongforfoo");
```

Exploitation

- The general idea is to give servers very large strings that will overflow a buffer.
- For a server with sloppy code – it's easy to crash the server by overflowing a buffer.
- It's sometimes possible to actually make the server do whatever you want (instead of crashing).

Necessary Background

- C functions and the stack.
- A little knowledge of assembly/machine language.
- How system calls are made (at the level of machine code level).
- **exec()** system calls
 - How to “guess” some key parameters.

What is a Buffer Overflow?

- Intent
 - Arbitrary code execution
 - Spawn a remote shell or infect with worm/virus
 - Denial of service
 - Cause software to crash
 - E.g., ping of death attack
- Steps
 - Inject attack code into buffer
 - Overflow return address
 - Redirect control flow to attack code
 - Execute attack code

Attack Possibilities

- Targets
 - Stack, heap, static area
 - Parameter modification (non-pointer data)
 - Change parameters for existing call to `exec()`
 - Change privilege control variable
- Injected code vs. existing code
- Absolute vs. relative address dependence
- Related Attacks
 - Integer overflows
 - Format-string attacks

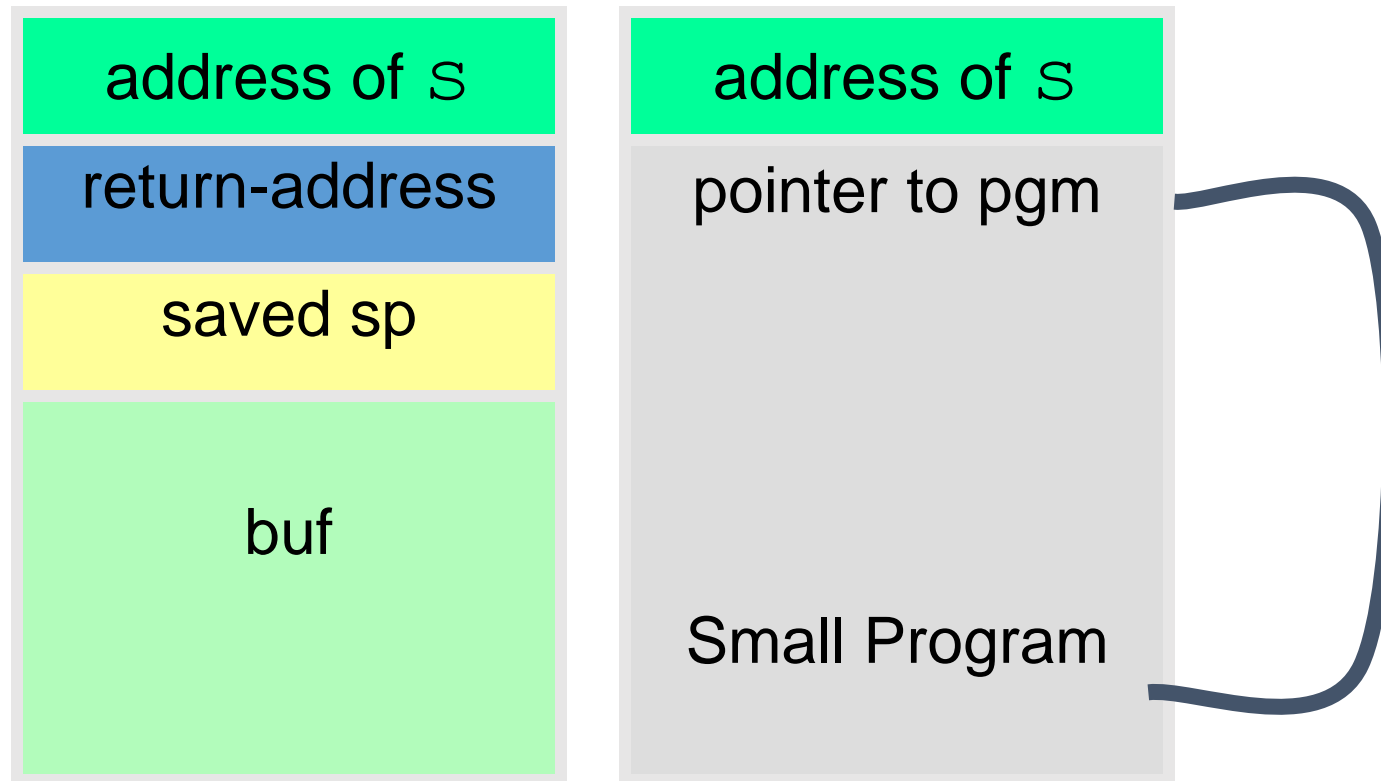
“Smashing the Stack”*

- The general idea is to overflow a buffer so that it overwrites the return address.
- When the function is done it will jump to whatever address is on the stack.
- We put some code in the buffer and set the return address to point to it!

*taken from the title of an article in Phrack 49-7

Before and After

```
void foo(char *s) {  
    char buf[100];  
    strcpy(buf, s);  
    ...  
}
```



What causes buffer overflow?

Example: gets()

```
char buf[20];  
gets(buf); // read user input until  
           // first EoL or EoF character
```

- Never use **gets**
- Use **fgets(buf, size, stdout)** instead

Example: strcpy()

```
char dest[20];
```

```
strcpy(dest, src); // copies string src to dest
```


- **strcpy** assumes **dest** is long enough , and assumes **src** is null-terminated
- Use **strncpy(dest, src, size)** instead

Spot the defect! (1)

```
char buf[20];  
char prefix[] = "http://";  
...  
strcpy(buf, prefix);  
    // copies the string prefix to buf  
strncat(buf, path, sizeof(buf));  
    // concatenates path to the string buf
```

Spot the defect! (1)

```
char buf[20];  
char prefix[] = "http://";  
...  
strcpy(buf, prefix);  
    // copies the string prefix to buf  
strncat(buf, path, sizeof(buf));  
    // concatenates path to the string buf
```




strncat's 3rd parameter is number of
chars to copy, not the buffer size

Another common mistake is giving sizeof(path) as 3rd argument...


Spot the defect! (2)

```
char src[9];  
char dest[9];
```

base_url is 10 chars long, incl. its
null terminator, so src won't be
not null-terminated



```
char base_url = "www.ru.nl";  
strncpy(src, base_url, 9);  
    // copies base_url to src  
strcpy(dest, src);  
    // copies src to dest
```



so strcpy will overrun the buffer dest

Example: strcpy and strncpy

- Don't replace **strcpy(dest, src)** by
 - **strncpy(dest, src, sizeof(dest))**
- but by
 - **strncpy(dest, src, sizeof(dest)-1)**
 - **dest[sizeof(dest)-1] = '\0';**
 - if **dest** should be null-terminated!
- A strongly typed programming language could of course enforce that strings are always null-terminated...

Spot the defect! (3)

```
char *buf;  
int i, len;  
read(fd, &len, sizeof(len));  
buf = malloc(len);  
read(fd,buf,len);
```

Spot the defect! (3)

```
char *buf;  
int i, len;  
read(fd, &len, sizeof(len));  
buf = malloc(len);  
read(fd, buf, len);
```

Didn't check if negative



len cast to unsigned and negative
length overflows



- **Mmemcpy() prototype:**
 - `void *memcpy(void *dest, const void *src, size_t n);`
- **Definition of size_t:** `typedef unsigned int size_t;`

Implicit Casting Bug

- Implicit casting bugs can occur when inequalities are evaluated incorrectly due to one type being implicitly cast to another.

- **A signed/unsigned or an implicit casting bug**
 - Very nasty – hard to spot
- **C compiler doesn't warn about type mismatch between signed int and unsigned int**
 - Silently inserts an implicit cast

Spot the defect! (4)

```
char *buf;  
int i, len;  
read(fd, &len, sizeof(len));  
if (len < 0)  
    {error ("negative length"); return; }  
buf = malloc(len+5);  
read(fd,buf,len);  
buf[len] = '\0'; // null terminate buf
```

May results in integer overflow



Spot the defect! (5)

```
#define MAX_BUF = 256
```

```
void BadCode (char* input)
{
    short len;
    char buf[MAX_BUF];
    len = strlen(input);
    if (len < MAX_BUF) strcpy(buf,input);
}
```

What if input is longer than 32K ?
len will be a negative number,
due to integer overflow
hence: potential buffer overflow

Spot the defect! (6)

```
char buff1[MAX_SIZE], buff2[MAX_SIZE];  
// make sure it's a valid URL and will fit  
if (! isValid(url)) return;  
if (strlen(url) > MAX_SIZE - 1) return;  
// copy url up to first separator, ie. first '/', to buff1  
out = buff1;  
do {  
    // skip spaces  
    if (*url != ' ') *out++ = *url;  
} while (*url++ != '/');  
strcpy(buff2, buff1);  
...
```

' data-bbox="330 660 465 755"/>

what if there is no '/' in the URL?

Loop termination (exploited by Blaster worm)

Spot the defect! (7)

```
#include <stdio.h>

int main(int argc, char* argv[])
{ if (argc > 1)
  printf(argv[1]);
  return 0;
}
```

This program is vulnerable to **format string** attacks, where calling the program with strings containing special characters can result in a buffer overflow attack.

Heap Overflow Attacks

<https://www.mathyvanhoef.com/2013/02/understanding-heap-exploiting-heap.html>

What is a heap?

- Heap is a collection of variable-size memory chunks allocated by the program
 - e.g., malloc(), free() in C,
creating a new object in Java
creating a new object in Java script
- Heap management system controls the allocation, de-allocation, and reclamation of memory chunks.
 - To do this some meta data is necessary for book-keeping

Static Linking

In static linking, external code (libraries) is integrated into the executable at compile time. This means that the code from the library becomes part of the executable file.

- Pros:

- Simplicity: The executable contains all necessary code and doesn't depend on external libraries at runtime.

- Performance: Might be slightly faster at startup as all code is readily available.

- Reliability: No dependency on external library versions at runtime.

- Cons:

- Size: Executables are larger as they contain all linked libraries.

- Updates: Updating a library requires recompiling the entire application.

- Resource Usage: Duplicate library code in multiple executables takes more disk space.

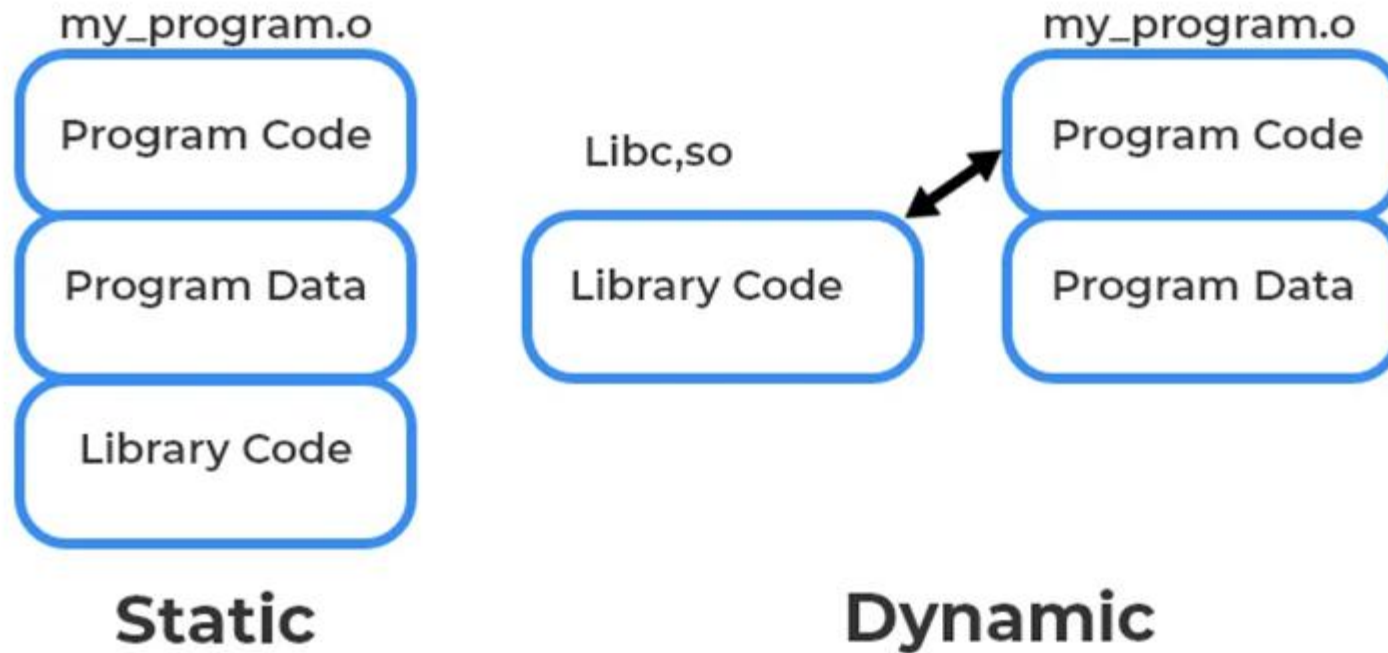
Dynamic Linking

In dynamic linking, libraries remain separate and the executable contains references to these external libraries. These libraries are loaded into memory at runtime.

- Pros:

- Size: Smaller executables, as they don't contain the actual library code.
 - Memory Efficiency: Shared libraries are loaded into memory once and used by multiple programs.
 - Updates: Libraries can be updated independently of the executable.
- Cons:
- Complexity: Requires proper management of library versions and paths.
 - Dependency: If a required library is missing or incompatible, the program won't run.
 - Startup Time: May increase slightly due to the need to locate and load libraries at runtime.

Static Vs Dynamic Linking



Heap overflow attacks

- What if heap memory is corrupted?
 - If a buffer is allocated on the heap and overflowed, we could overwrite the heap meta data
 - This can allow us to modify any memory location with any value of our chosen
 - This could lead to running arbitrary code

The Global Offset Table (GOT) and the Procedure Linkage Table (PLT) are crucial components in dynamic linking, particularly in the context of shared libraries in Unix-like operating systems.

Procedure Linkage Table (PLT):

- The PLT is used in conjunction with the GOT to manage dynamic function calls.
- When a function in a shared library is called for the first time, the call goes through the PLT, which contains a stub for each dynamic function.
- Initially, this stub points to code that invokes the dynamic linker to resolve the function's address. Once resolved, the stub is updated to directly point to the function, speeding up subsequent calls.

Global Offset Table (GOT):

- It's a table of addresses used to manage global variables and static functions in shared libraries.
- When a program starts, the actual addresses of these elements are unknown. The GOT maintains placeholders that get updated with correct addresses once the shared library is loaded and linked.
- This allows code in the shared library to reference its global variables and static functions regardless of the actual memory location.

Note: PLT is located in the code segment of an executable program and is readable but not writable, and the GOT is located in the data segment of an executable program and is readable and writable.

Global Offset Table (GOT) & Procedure Linkage Table (PLT)

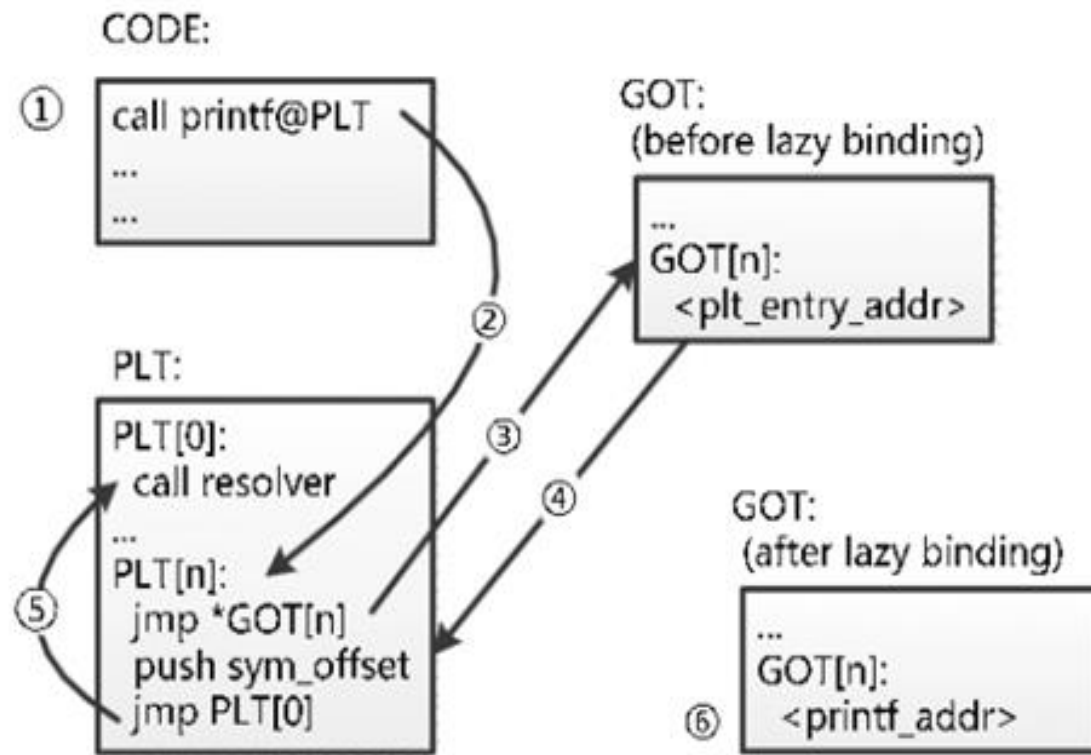
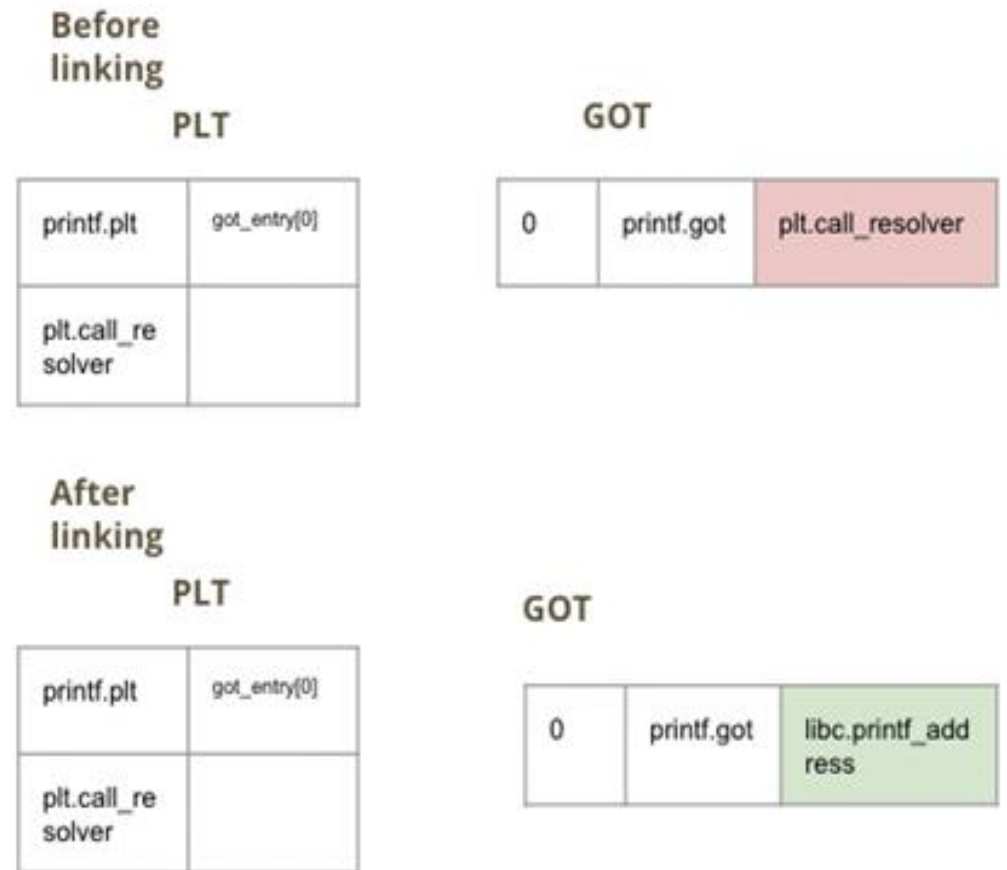
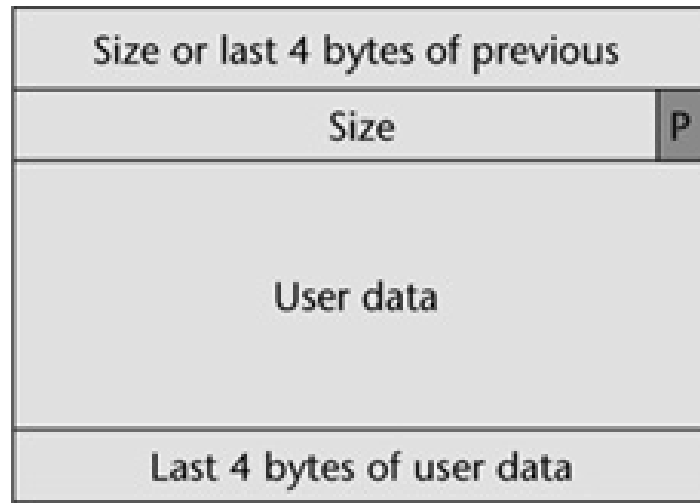


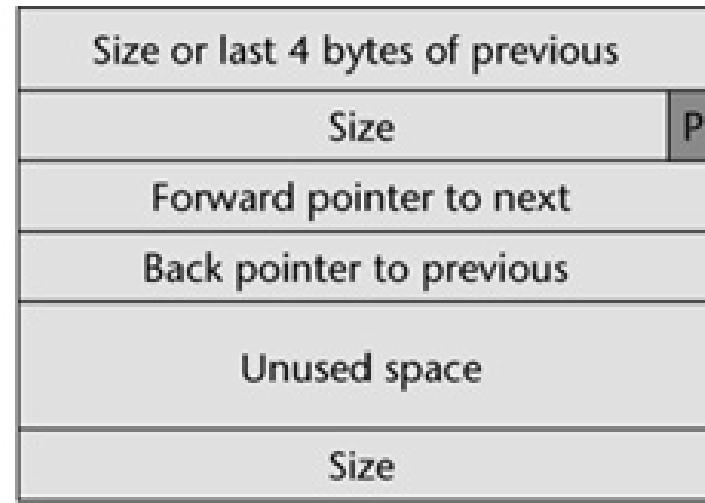
Figure 1. Usage of PLT and GOT



Doug Lea's malloc and free



Allocated chunk



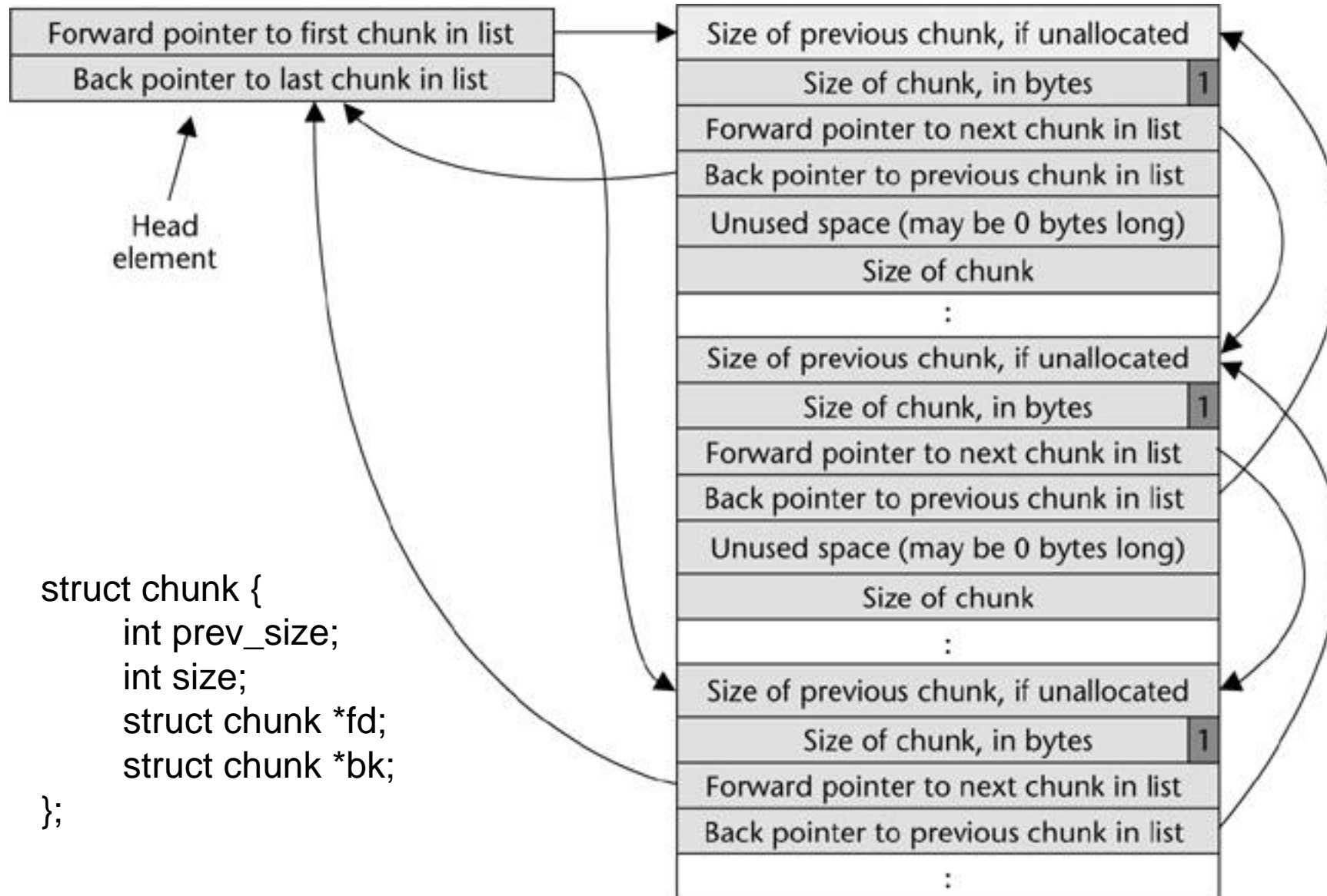
Free chunk

P == 1: previous chunk is in use

P == 0: previous chunk is free

Size includes the first two control words

Double-linked free chunk list



heap.c

```
#define BUFSIZE 128

int main(int argc, char *argv[])
{
    char *a, *b, *c;

    if(argc < 2) {
        printf("Usage: %s <buffer>\n", argv[0]);
        exit(-1);
    }

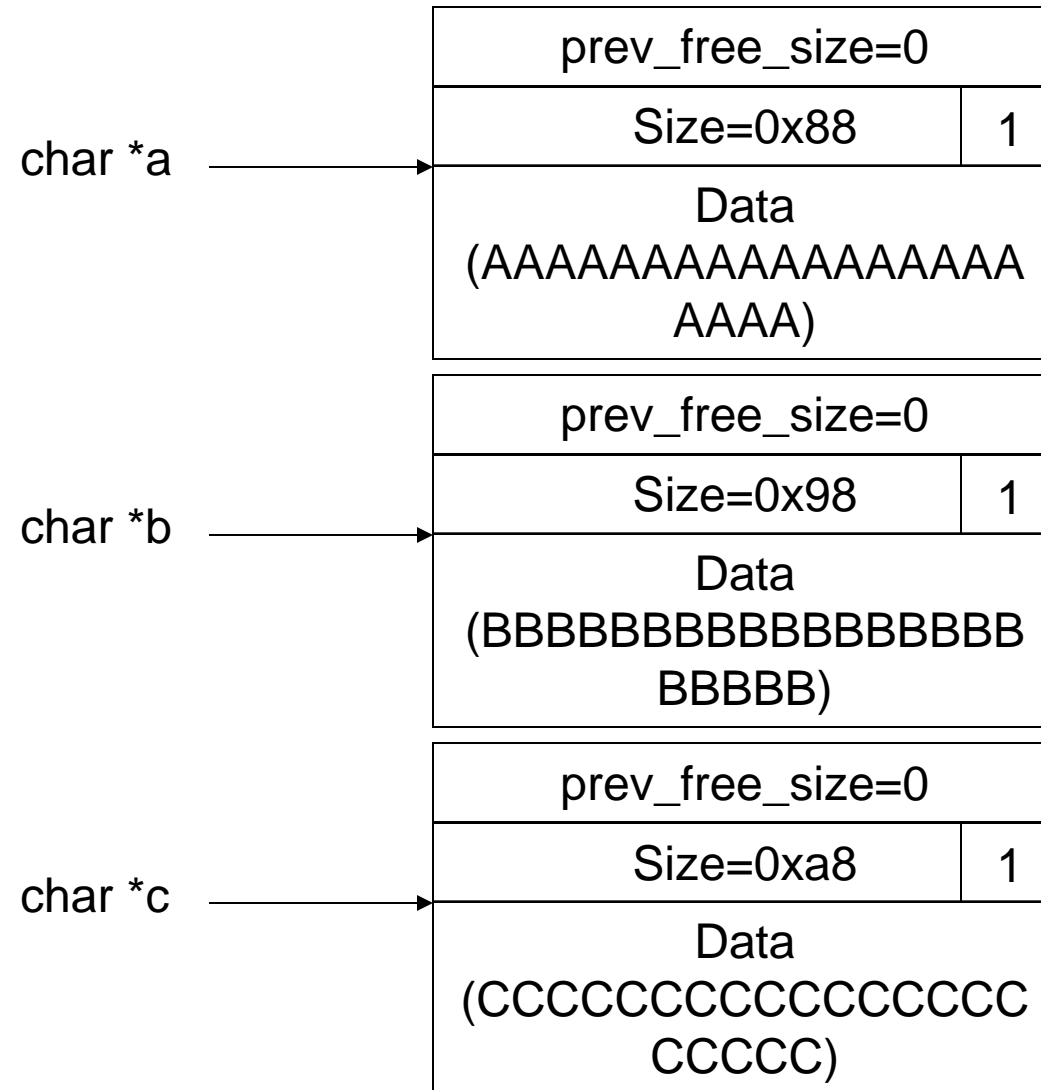
    a = (char *) malloc(BUFSIZE);
    b = (char *) malloc(BUFSIZE+16);
    c = (char *) malloc(BUFSIZE+32);

    printf("address of a: %p\n", a);
    printf("address of b: %p\n", b);
    printf("address of c: %p\n", c);

    strcpy(b, "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB");
    strcpy(c, "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC");
    strcpy(a, argv[1]);

    free(a);
    free(b);
    free(c);
}
```

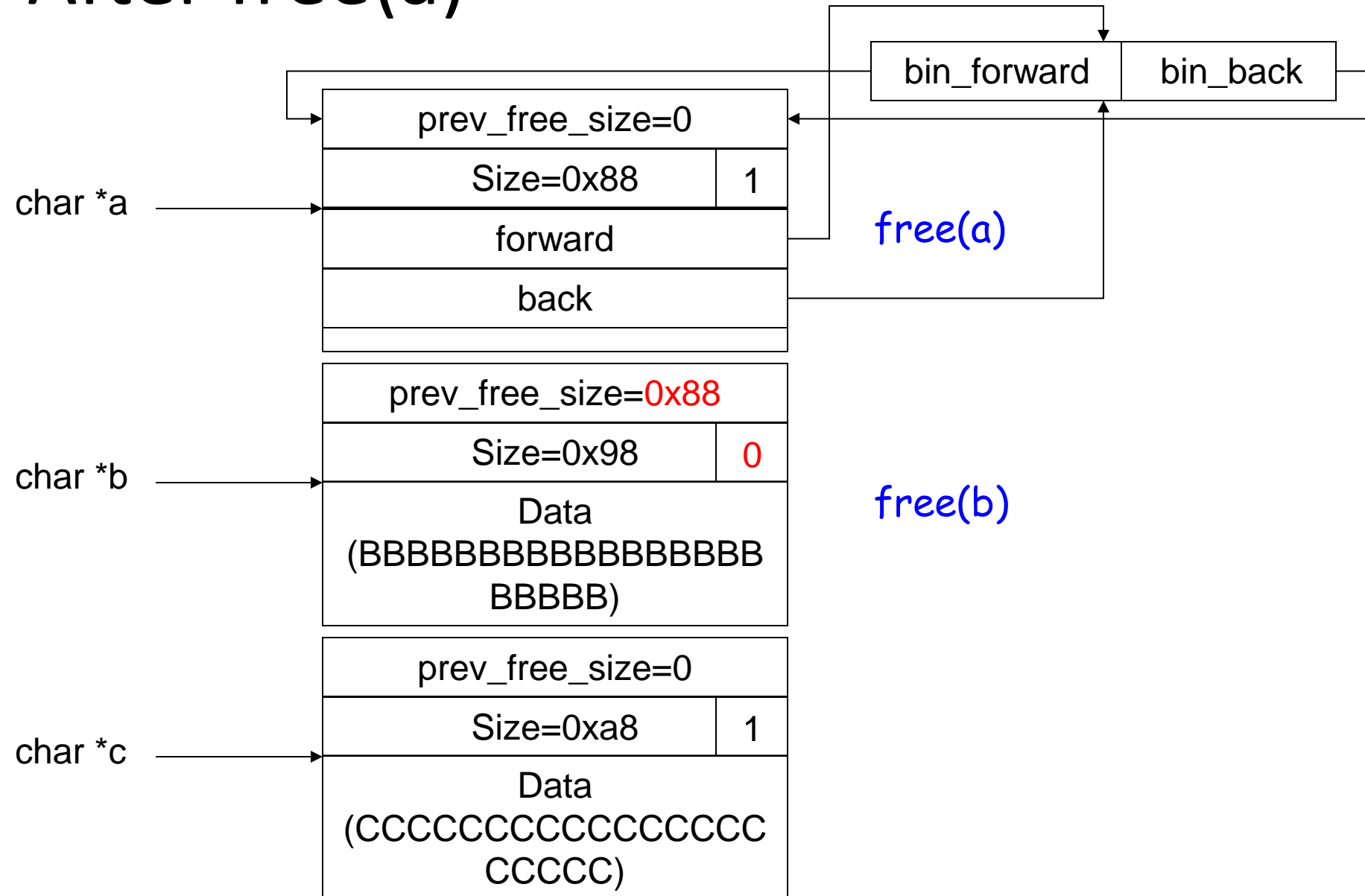
The heap in memory



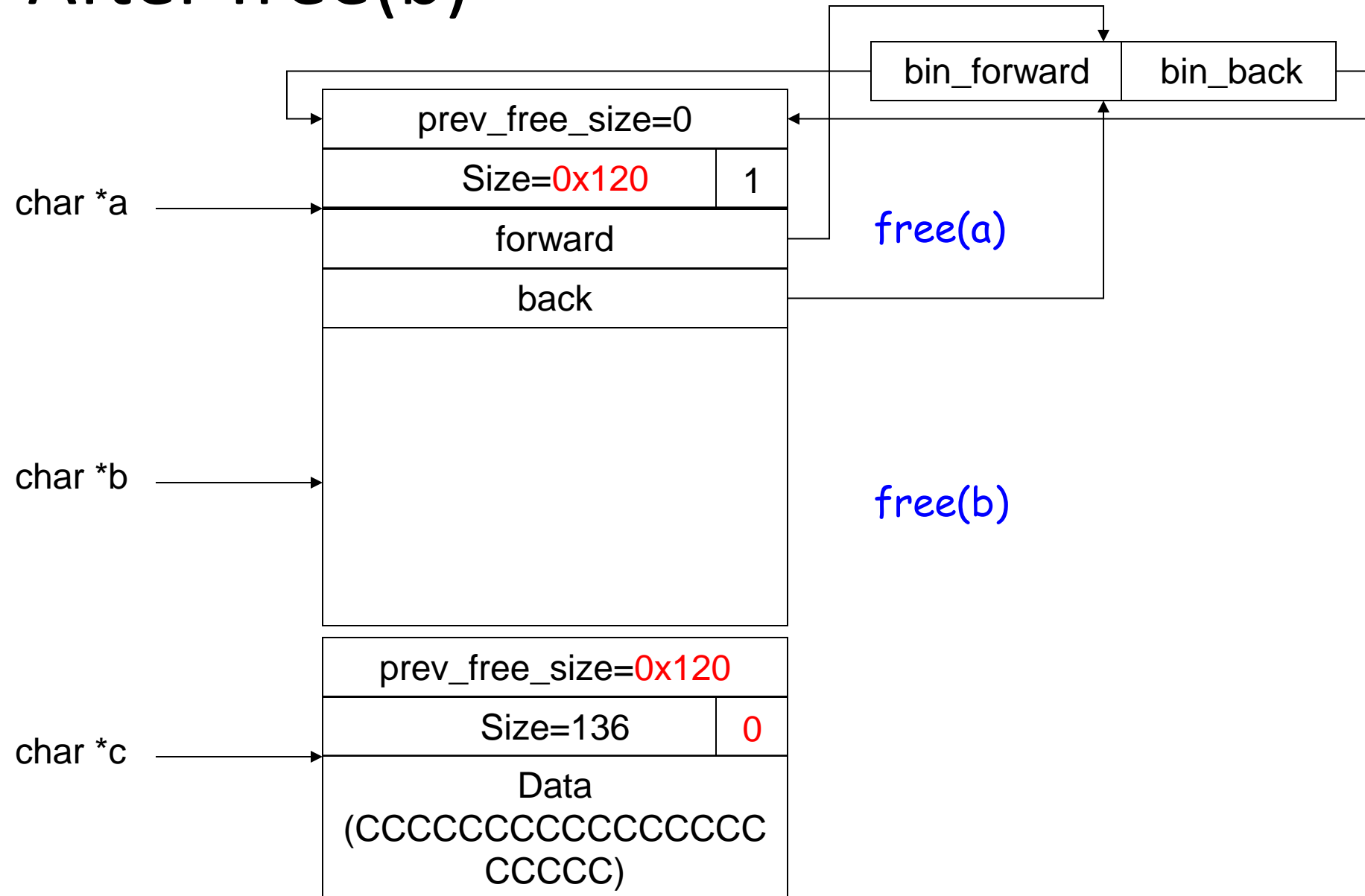
bin_forward	bin_back
-------------	----------

`free(a)`

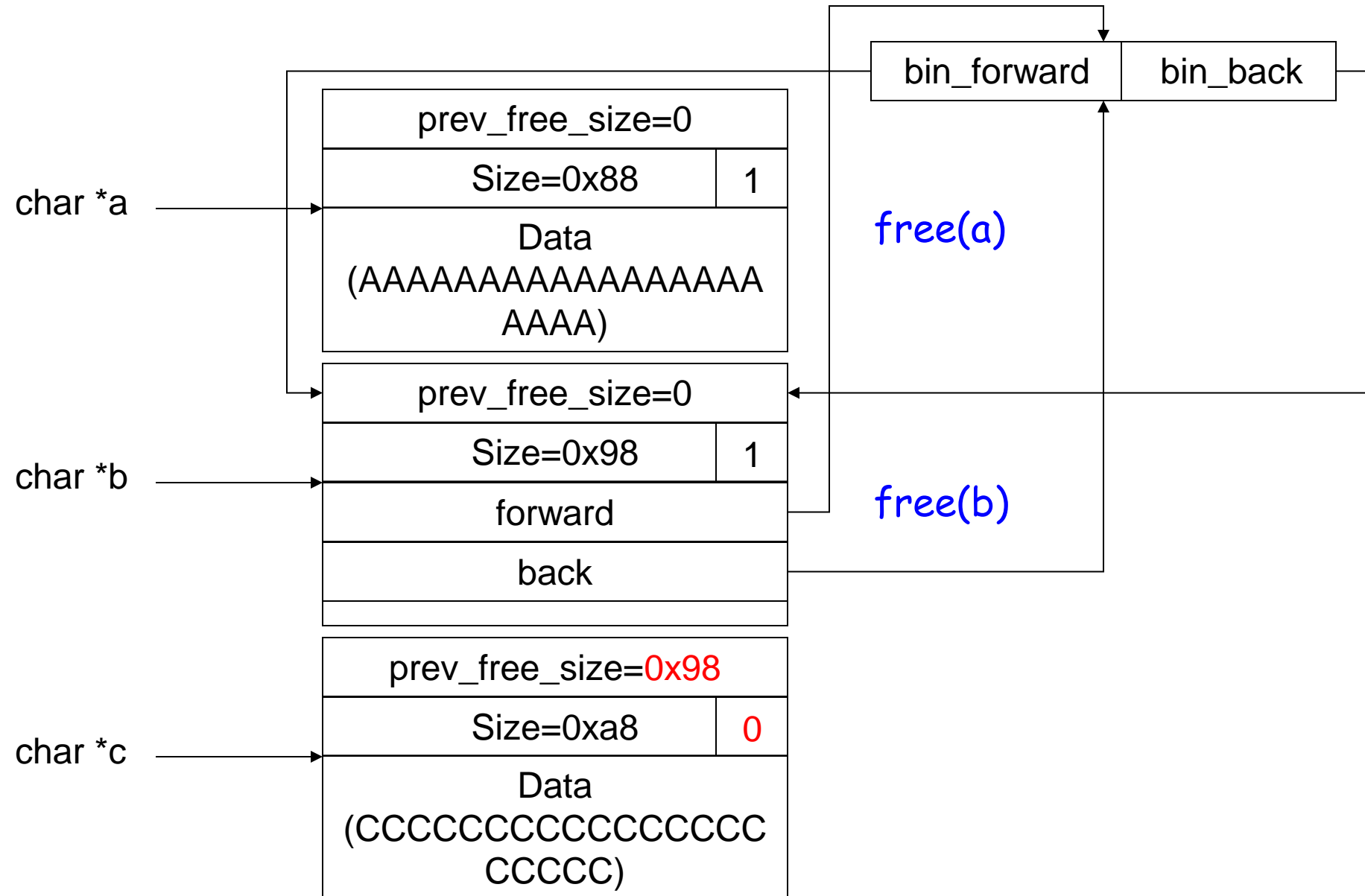
After free(a)



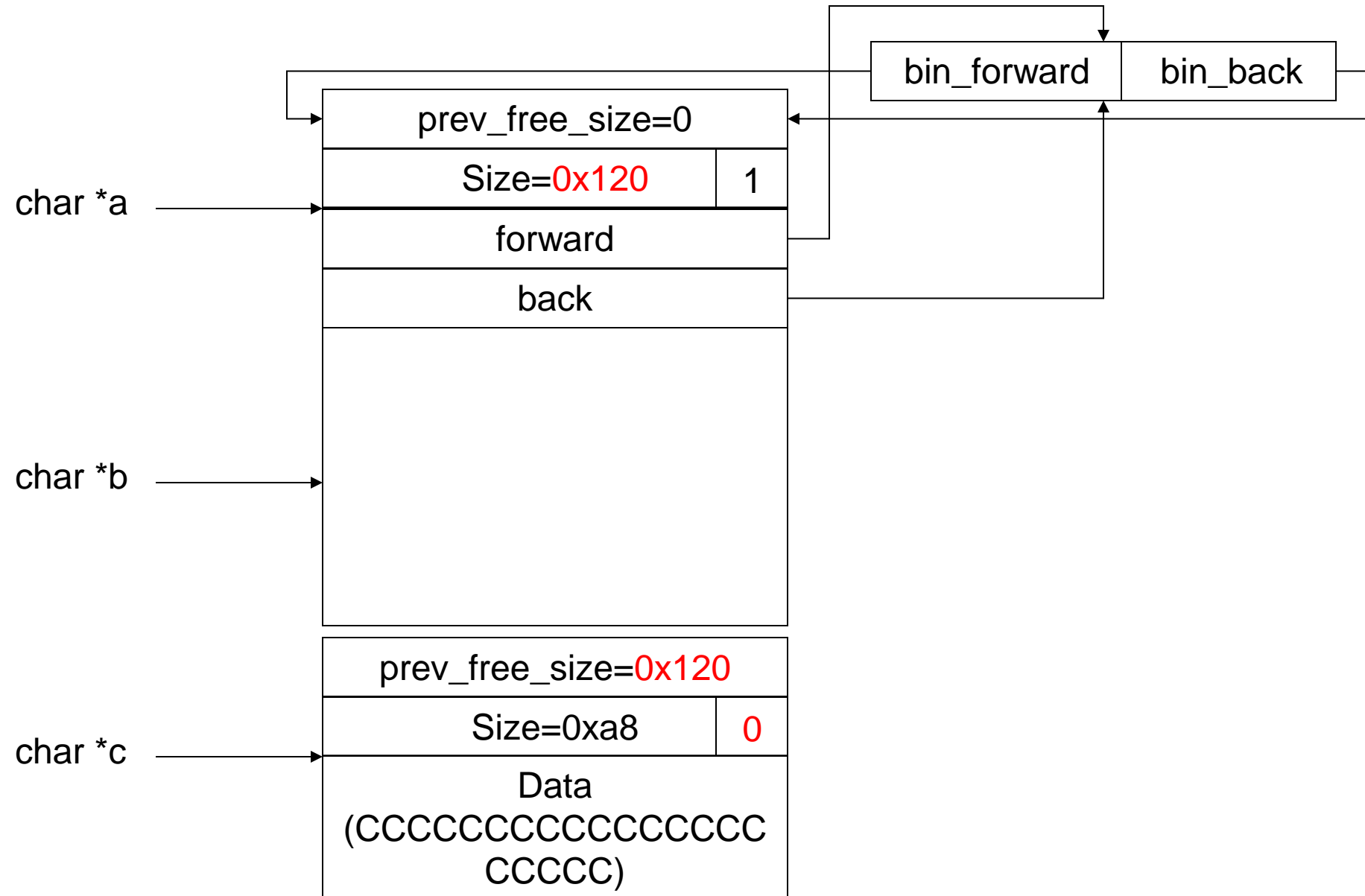
After free(b)



What if “b” was freed first?



Then “a” was freed



Key Observation

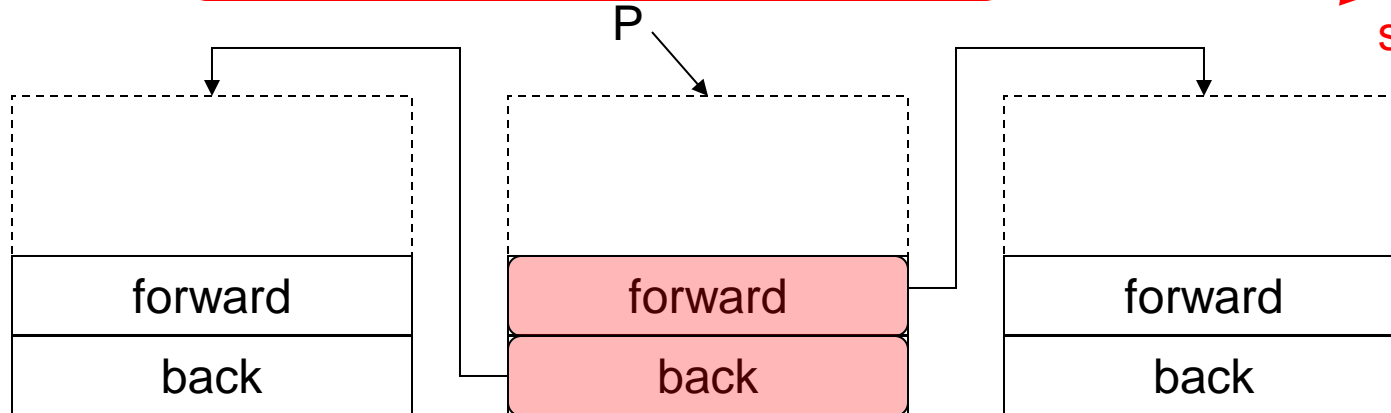
- When the chunk (A) to be free'd is followed by a free chunk (B), chunk B will be unlinked from the free list, and A will be inserted into the free list.
- If the heap meta data in B has been corrupted, the unlink operation will be dangerous.
 - Why?

Remove a chunk from the double-linked free list

- unlink(P)

```
P->fd->bk = P->bk;
```

```
P->bk->fd = P->fd;
```



Will create
some trouble
for us

If the “fd” pointer is corrupted,
we can control which memory
cell to be modified

If the “bk” pointer is corrupted,
we can control what value to
put into the memory cell

Overwrite the “forward” field with “where-12”. Overwrite the “back” field with “what”.

A heap overflow attack

- Overwrite a heap memory chunk by running past the allocated space.
- Create a fake heap structure that pretends to be a free chunk.
- When the current chunk is free()'d, an arbitrary memory overwrite will occur.
- We can control the “where” and “what” of the memory overwrite.

The “where”

- By modifying certain memory locations, we can modify the EIP and make it point to injected malicious code.
- The Global Offset Table (GOT) contains the entry addresses of dynamically linked library functions (e.g. printf, malloc, free, exit, strcpy, etc.)
 - We can overwrite a GOT entry that will likely be used by the program

```
$ objdump -R heap
heap:   file format elf32-i386
DYNAMIC RELOCATION RECORDS
OFFSET TYPE           VALUE
0804975c R_386_GLOB_DAT
__gmon_start__
08049744 R_386_JUMP_SLOT  malloc
08049748 R_386_JUMP_SLOT
__libc_start_main
0804974c R_386_JUMP_SLOT  printf
08049750 R_386_JUMP_SLOT  exit
08049754 R_386_JUMP_SLOT  free
08049758 R_386_JUMP_SLOT  strcpy
```

The “what”

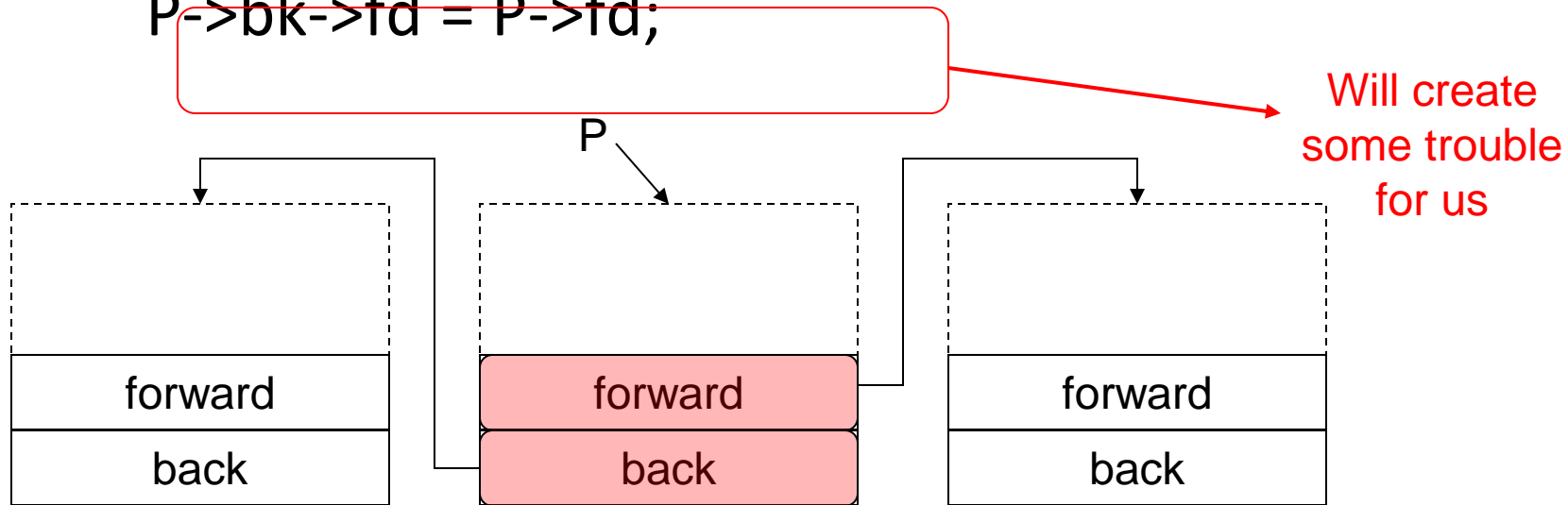
- The “what” will be the address of the malicious code we injected into the buffer.
- But, `P->bk->fd=P->fd` will “puncture” our shellcode from bytes 8-12.
- Thus the shell code must “jump over” this hole.

Remove a chunk from the double-linked free list

- unlink(P)

$P \rightarrow fd \rightarrow bk = P \rightarrow bk;$

$P \rightarrow bk \rightarrow fd = P \rightarrow fd;$

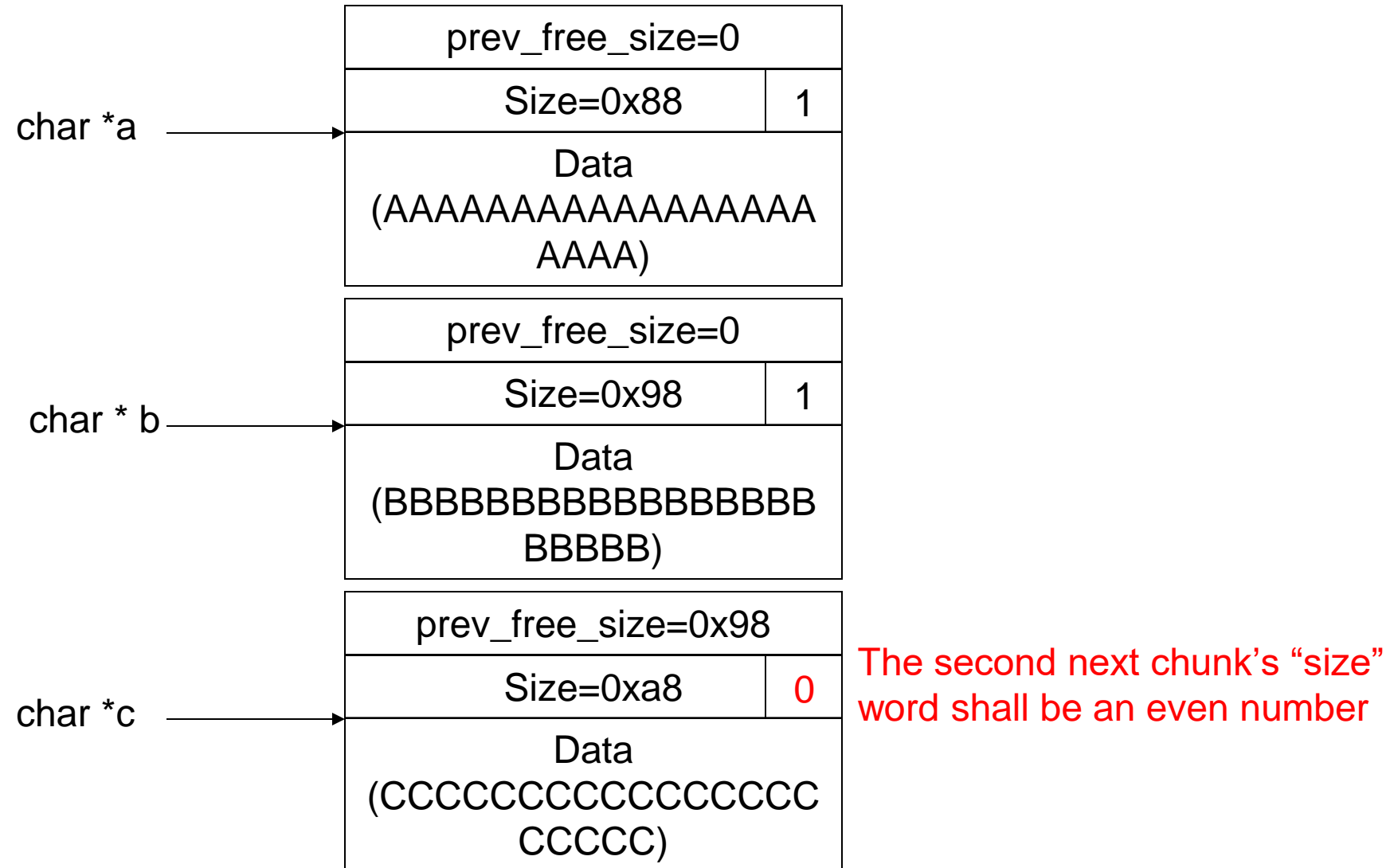


If the "fd" pointer is corrupted,
we can control which memory
cell to be modified

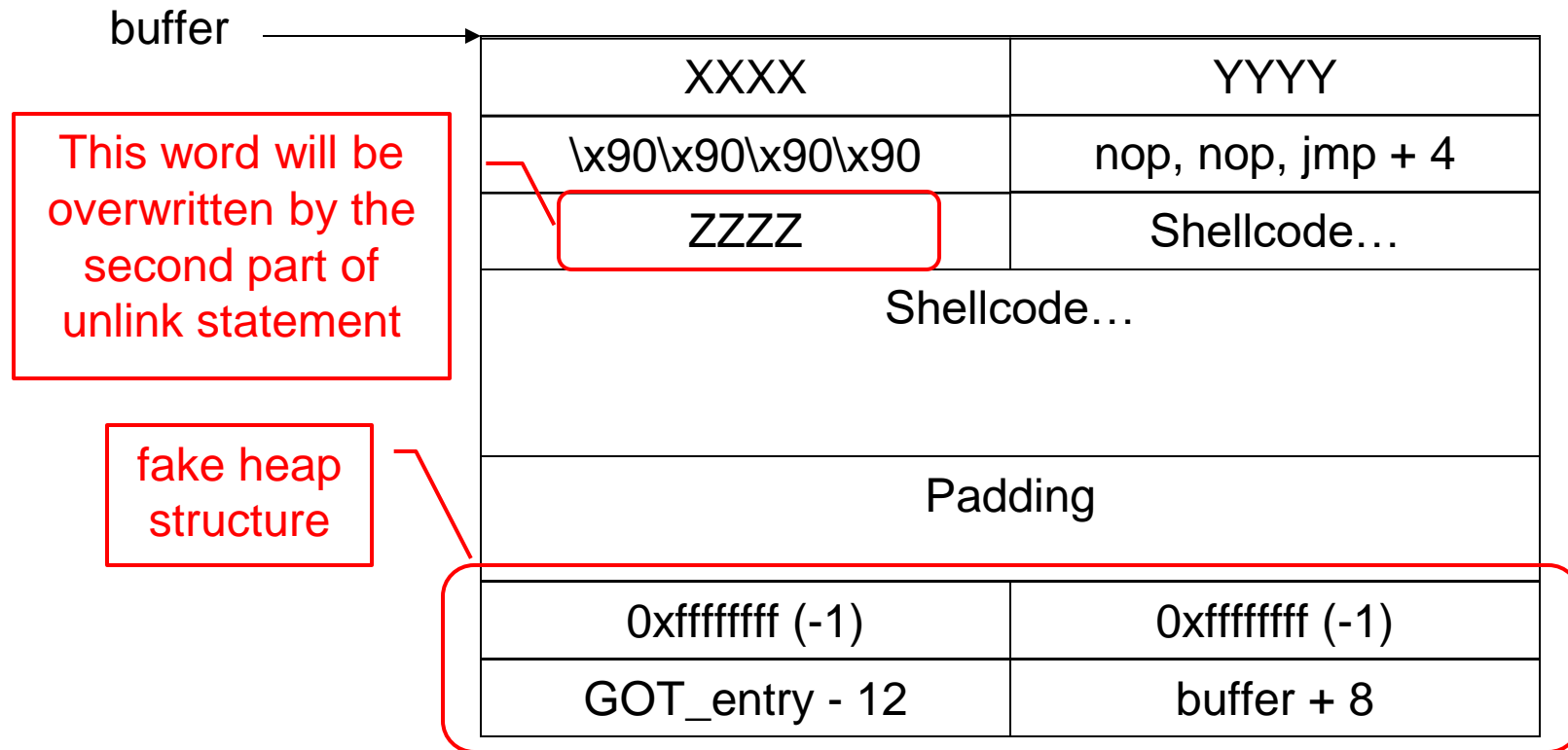
If the "bk" pointer is corrupted,
we can control what value to
put into the memory cell

Overwrite the "forward" field with "where-12". Overwrite the "back" field with "what".

How to make free() think the chunk after “a” is free



Assembled heap-overflow payload



Format String Attacks

- `int printf(const char *format [, argument]...);`
 - `snprintf`, `wsprintf` ...
- What may happen if we execute
`printf(string);`
 - Where **`string`** is user-supplied ?
 - If it contains special characters, eg `%s`, `%x`, `%n`, `%hn`?

Format String Attacks

- Why this could happen?
 - Many programs delay output message for batch display:
 - `fprintf(STDOUT, err_msg);`
 - Where the `err_msg` is composed based on user inputs
 - If a user can change `err_msg` freely, format string attack is possible

Format String Attacks

- **%x** reads and prints 4 bytes from stack
 - this may leak sensitive data
- **%n** writes the number of characters printed so far onto the stack
 - this allow stack overflow attacks...
- C format strings break the “don’t mix data & code” principle.
- “Easy” to spot & fix:
 - replace **printf(str)** by **printf(“%s”, str)**

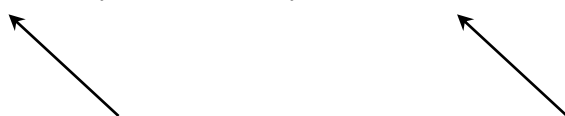
```
#include <stdio.h>

void foo(char *format){
    int a1=11; int a2=12;
    int a3=13; int a4=14;
    printf(format);
}

void main(int argc, char **argv){
    foo(argv[1]);
    printf("\n");
}

$./format-x-subfun "%x %x %x %x : %x, %x, %x "
```

80495bc **e d c : b**, bffff7e8, **80483f4**



Four variables Return address

- **What does this string (“%x:%x:%s”) do?**
 - **Prints first two words of stack memory**
 - **Treats next stack memory word as memory addr and prints everything until first '\0'**
 - **Could segment fault if goes to other program's memory**

- **Use obscure format specifier (%n) to write any value to any address in the victim's memory**
 - %n --- write 4 bytes at once
 - %hn --- write 2 bytes at once
- **Enables attackers to mount malicious code injection attacks**
 - Introduce code anywhere into victim's memory
 - Use format string bug to overwrite return address on stack (or a function pointer) with pointer to malicious code

Example of “%n”---- write data in memory

```
#include <stdio.h>
void main(int argc, char **argv){
    int bytes;
    printf(“%s%n\n”, argv[1], &bytes);
    printf(“You input %d characters\n”, bytes);
}
```

s:~\$./test hello

hello

You input 5 characters

getenvaddr.c which can give us the address of environment variables:

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

int main(int argc, char* argv[]){
    char* ptr;
    if (argc < 3) {
        printf("at least 2 arguments needed\n");
        exit(0);
    }

    ptr = getenv(argv[1]);
    ptr += (strlen(argv[0]) - strlen(argv[2]))*2;
    printf("%s is stored at the address %p\n", argv[1], ptr);
    return 0;
}
```

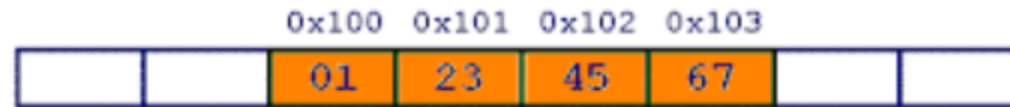
Now first, we convert the address given by the program in the little endian address format. So, it will be `"/xc0/xd0/xf0/xf0"`.

```
[thelaw-pc writing]# ./getenvaddr PATH ./a.out
PATH is stored at the address 0xffffdfc0
all tabs law-pc writing]#
```

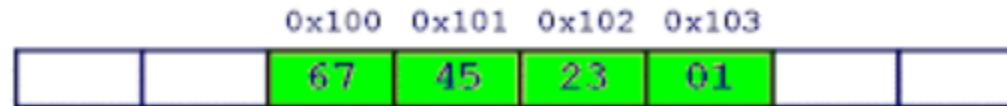
Then, we supply the address via printf to preserve the address format and a bunch of %x to empty the stack so that esp can point to the address we provided in the format string, i.e., the variable's address is later printed with %s.

```
[thelaw-pc writing]# ./a.out $(printf "\xc0\xdf\xff\xff")%X.%X.%X.%X.%X.%X.%X.%X.%X.%X.%S  
ffffd896.f7da968c.565561e7.f7db276c.f7f88000.ffffd2ec.ffffd764.3.0.f7ffcfc0.asd  
[thelaw-pc writing]#
```

Suppose integer is stored as 4 bytes (For those who are using DOS-based compilers such as C++ 3.0, integer is 2 bytes) then a variable x with value 0x01234567 will be stored as following.



Big Endian



Little Endian

Preventing

So this way, we can exploit format strings to escape the bounds of the application and perform arbitrary reads. Similar to what we did with arbitrary reads, we can use %n format strings to arbitrary write to variables.

This could lead to overwriting sensitive variables such as passwords and username of the program and lead to a complete compromise.

Attackers can also make use of this vulnerability to perform arbitrary writes in the dtors table of the program and make the program execute malicious code.

Preventing Format String Exploits

- Input Validation
- Format Guard
- Kimchi
- Address Space Layout Randomization (ASLR)

Function Pointer Overwritten

- Function pointers: (used in attack on PHP 4.0.2)



- Overflowing buf will override function pointer.
- Harder to defend than return-address overflow attacks

Unicode and ANSI Buffer Size Mismatches (windows platform)

It occurs if you mix up the number of elements with the size in bytes of a Unicode buffer. There are two reasons it's rather widespread: Windows NT and later support ANSI and Unicode strings, and most Unicode functions deal with buffer sizes in wide characters, not byte sizes.

The most commonly used function that is vulnerable to this kind of bug is **MultiByteToWideChar**.

<https://learn.microsoft.com/en-us/windows/win32/api/stringapiset/nf-stringapiset-multibytetowidechar>

Example

```
BOOL GetName(char *szName)
{
    WCHAR wszUserName[256];
    // Convert ANSI name to Unicode.
    MultiByteToWideChar(CP_ACP, 0, szName, -1, wszUserName, sizeof(wszUserName))

    ;
    // Snip
}
```

The problem is the last argument of `MultiByteToWideChar`. The documentation for this argument states: “Specifies the size, in wide characters, of the buffer pointed to by the `lpWideCharStr` parameter.” The value passed into this call is `sizeof(wszUserName)`, which is 256, right?

No, it's not. `wszUserName` is a Unicode string; it's 256 wide characters. A wide character is two bytes, so `sizeof(wszUserName)` is actually 512 bytes. Hence, the function thinks the buffer is 512 wide characters in size. Because `wszUserName` is on the stack, we have a potential exploitable buffer overrun.

```
MultiByteToWideChar(CP_ACP, 0, szName, -1, wszUserName,  
                    sizeof(wszUserName) / sizeof(wszUserName[0]));
```


Finding buffer overflows

- Hackers find buffer overflows as follows:
 - Run target server program on local machine.
 - Issue requests with long tags.
All long tags end with “\$\$\$\$\$”.
 - If web server crashes,
search core dump for “\$\$\$\$\$” to find
overflow location.
- Some automated tools exist. (eEye Retina, ISIC).
- Then use disassemblers and debuggers (eg. IDA-Pro) to construct exploit.

Preventing Buffer Overflow Attacks

- Handle the strings safely
- Avoid unsafe concatenation
- Use type safe languages (Java, Scala)
- Use safe library functions
- Use better software development process, e.g., manual code review
- Static source code analysis
- Non-executable stack
- Run time checking: StackGuard, Libsafe, SafeC, (Purify), and so on.
- Address space layout randomization
- Detection deviation of program behavior
- Access control to control aftermath of attacks...

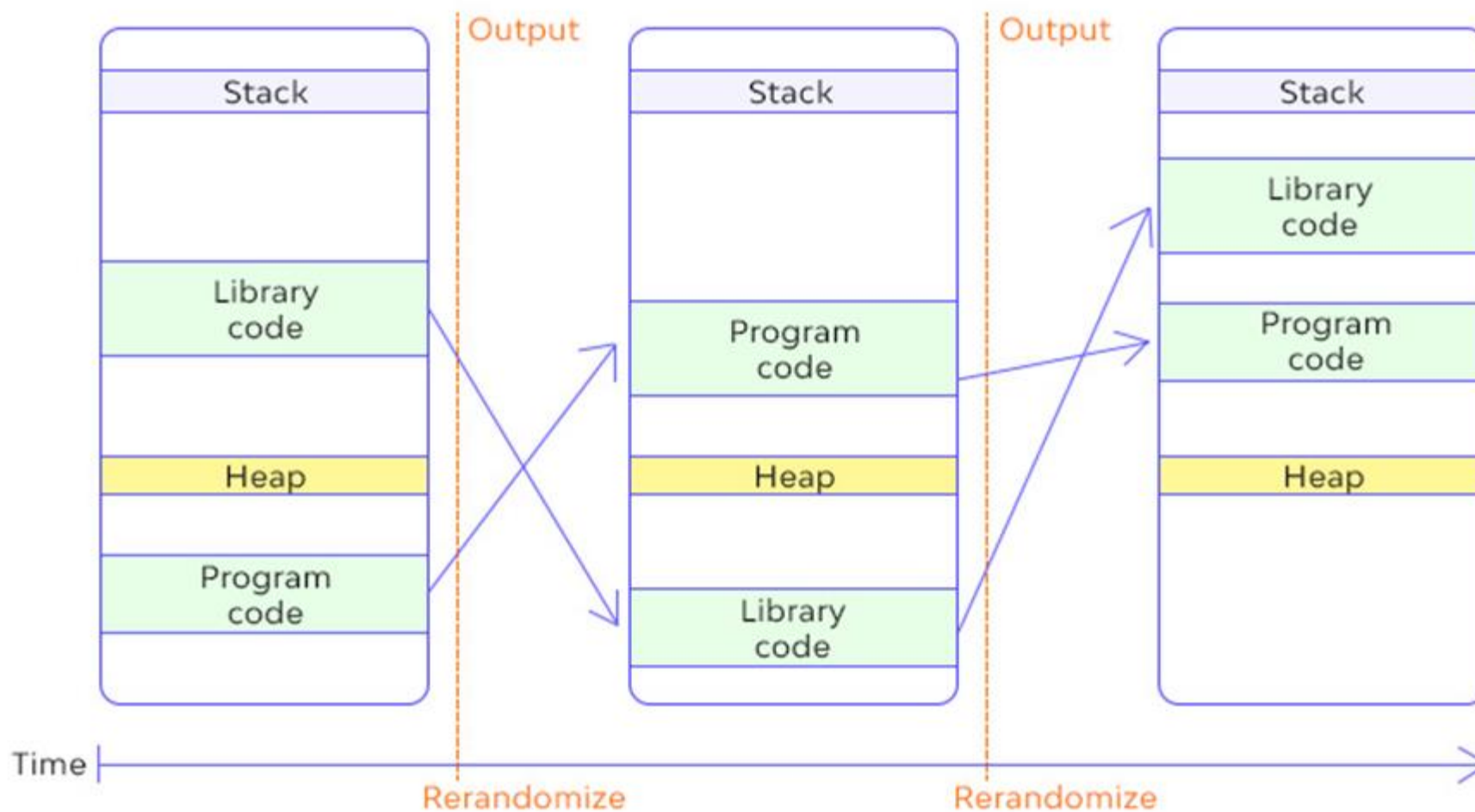
Insecure Function	Safe Alternative
<code>strcpy</code>	<code>strncpy*</code> , <code>strcpy_s*</code>
<code>strcat</code>	<code>strlcat*</code> , <code>strcat_s*</code>
<code>printf/sprintf</code>	<code>snprintf*</code> , <code>sprintf_s*</code>
<code>gets</code>	<code>fgets</code>

Protections Provided by Operating Systems

- ASLR – Address space layout randomization
- DEP – Data Execution Prevention

Data Execution Prevention (DEP) is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003. DEP enables the system to mark one or more pages of memory as non-executable. Marking memory regions as non-executable means that code cannot be run from that region of memory, which makes it harder for the exploitation of buffer overruns.

Address Space Layout Randomization example



- Structured exception handler overwrite protection (SEHOP)—helps stop malicious code from attacking Structured Exception Handling (SEH), a built-in system for managing hardware and software exceptions. It thus prevents an attacker from being able to make use of the SEH overwrite exploitation technique. At a functional level, an SEH overwrite is achieved using a stack-based buffer overflow to overwrite an exception registration record, stored on a thread's stack.

Writing secure code is the best way to prevent buffer overflow vulnerabilities.

Reference

Writing Secure Code by Michael Howard and David LeBlanc -Microsoft
<https://eli.thegreenplace.net/2011/02/04/where-the-top-of-the-stack-is-on-x86/>

additional

```
size_t bytes = n * m;  
if (bytes < n || bytes < m) { /* BAD BAD BAD */  
    ... /* allocate "bytes" space */  
}
```

Unfortunately, if `m` and `n` are signed, the C language specification allows the compiler to optimize out such tests [CWE-733, CERT VU#162289]. Even if they are unsigned, the test still fails in some cases. For example, on a 64-bit machine, if `m` and `n` are both declared `size_t`, and both set to `0x180000000`, the result of multiplying them is `0x240000000000000000`, but `bytes` will contain that result modulo 2^{**64} , or `0x4000000000000000`. This passes the test (the result is bigger than either input), despite the fact that overflow did occur.

Instead, the correct way to test for integer overflow during multiplication is to test *before* the multiplication. In particular, you divide the maximum allowable result by the multiplier and compare the result to the multiplicand or vice-versa. If the result is smaller than the multiplicand, the product of those two values would cause an integer overflow. Still, getting this right can be tricky. For example, choosing the wrong maximum allowable integer constant (e.g., `SIZE_MAX` or `INT_MAX`?) produces incorrect results.

the safest way to perform multiplication with unknown inputs is to use the [clang checked arithmetic builtins](https://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins)

<https://clang.llvm.org/docs/LanguageExtensions.html#checked-arithmetic-builtins>

```
size_t bytes;
if (__builtin_umull_overflow(m, n, &bytes)) {
    /* Overflow occurred.  Handle appropriately. */
} else {
    /* Allocate "bytes" space. */
}
```