



Lecture 4: Integer Overflows & Targeted Overwrites

presented by

Li Yi

Assistant Professor
SCSE

N4-02b-64

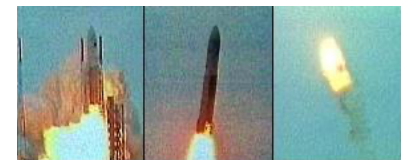
yi_li@ntu.edu.sg

Agenda

- Security issues with integers
 - Background on integers
 - Integer overflows
 - Integer mismatches
 - Units of measurement
- Format string attacks
- Memory allocation on the heap
 - Malloc and free
 - Double linked lists for memory management
- Double-free vulnerabilities
 - Exploiting memory management macros

The Ariane 5 Coding Error

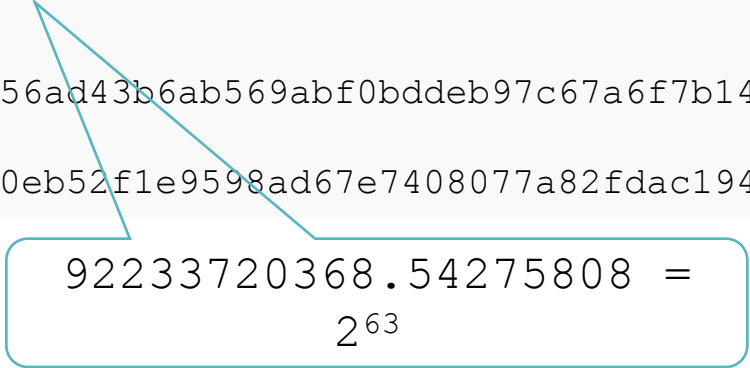
- June 4, 1996: the rocket on its first voyage, after a decade of development costing \$7 billion
- Cost of the destroyed rocket and cargo were around **\$500 million**
- The cause of the failure was a software error in the inertial reference system:
 - 64 bit floating point converted to a 16 bit signed integer
- Watch the video: <https://youtu.be/5tJPXYA0Nec>



From the Bitcoin Blockchain

```
CBlock(hash=0000000000790ab3, ver=1, hashPrevBlock=0000000000606865,
hashMerkleRoot=618eba,
nTime=1281891957, nBits=1c00800e, nNonce=28192719, vtx=2)
  CTransaction(hash=012cd8, ver=1, vin.size=1, vout.size=1, nLockTime=0)
    CTxIn(COutPoint(000000, -1), coinbase 040e80001c028f00)
    CTxOut(nValue=50.51000000, scriptPubKey=0x4F4BA55D1580F8C3A8A2C7)
  CTransaction(hash=1d5e51, ver=1, vin.size=1, vout.size=2, nLockTime=0)
    CTxIn(COutPoint(237fe8, 0), scriptSig=0xA87C02384E1F184B79C6AC)
    CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0xB7A7)
    CTxOut(nValue=92233720368.54275808, scriptPubKey=OP_DUP OP_HASH160 0x1512)
vMerkleTree: 012cd8 1d5e51 618eba
```

Block hash: 0000000000790ab3f22ec756ad43b6ab569abf0bddeb97c67a6f7b1470a7ec1c
Transaction hash:
1d5e512a9723cbef373b970eb52f1e9598ad67e7408077a82fdac194b65333c9


$$92233720368.54275808 = 2^{63}$$



CVE-2010-5139 (Bitcoin)

- On August 15, 2010, it was discovered that block 74638 contained a transaction that created over 184 billion bitcoins for two different addresses
- This was possible because the code used for checking transactions ($in \geq \sum outs$) before including them in a block didn't account for the case of outputs so large that they overflowed when summed
- A new version was published within a few hours of the discovery. The block chain had to be forked
- Although many unpatched nodes continued to build on the “bad” block chain, the “good” block chain overtook it at a block height of 74691
- The bad transaction no longer exists for people using the longest chain

On Integers

A number with no fractional part

Natural Numbers

- The natural numbers (counting numbers) \mathbb{N} are defined by the Peano postulates:
 - 1 is a member of the set \mathbb{N}
 - If n is a member of \mathbb{N} , then the **successor** $s(n)$ belongs to \mathbb{N}
 - 1 is not the successor of any element in \mathbb{N}
 - If $s(n) = s(m)$, then $n = m$
 - If $m \in \mathbb{N}$ is not 1, then there exists a $n \in \mathbb{N}$ such that $s(n) = m$
 - A subset of \mathbb{N} which contains 1, and which contains $n + 1$ whenever it contains n , must equal \mathbb{N}
- You can write $n + 1$ instead of $s(n)$
- This is the high-level specification mathematicians would use

Integers

- Recursive definition of addition: let $a, b \in \mathbb{N}$
 - If $b = 1$, then define $a + b = s(a)$
 - If $b \neq 1$, then take $c \in \mathbb{N}$ so that $s(c) = b$,
and define $a + b = s(a + c)$
- Integers: natural numbers together with their additive inverses (negative numbers) and zero
- Some simple calculations ...

$$127 + 1 = 128$$

$$255 + 1 = 256$$

$$0 - 1 = -1$$

$$16 * 17 = 272$$

$$-128 / -1 = 128$$

$$2^{63} + 2^{63} = 2^{64}$$

What will happen here?

```
int i = 1;
while (i > 0)
{
    i = i * 2;
}
```

Programming with Integers

- In mathematics integers form an infinite set
- On a computer systems, integers are represented in binary
- The representation of an integer is a binary string of fixed length (**precision**), so there is only a finite number of ‘integers’
- Programming languages: signed & unsigned integers, short and long (and long long) integers, ...

Two's Complement

- Signed integers often represented as **2's complement numbers**
 - Positive numbers are given in normal binary representation
 - Negative numbers are represented as the binary number that when added to a positive number of the same magnitude equals 2^n
- The most significant (leftmost) bit indicates the sign of the integer; it is also called the **sign bit**
 - Sign bit is zero: the number is positive
 - Sign bit is one: the number is negative

Two's Complement

- Calculating 2's complement representation of $-a$:
- Invert all bits in the binary representation of a
 - For n -bit integers, this step computes $2^n - 1 - a$
- Then add one to the intermediate result:
 - For n -bit integers, this step computes $2^n - a$
 - 2^n corresponds to the carry bit
- We have:
 - Positive numbers: $[0, 2^{n-1} - 1]$
 - Negative numbers: $[2^{n-1}, 2^n - 1]$
 - For $0 < a < 2^n$, $-a = 2^n - a$

Eight-Bit Signed Integers

Decimal Value	Binary (2's Complement)
0	0000 0000
1	0000 0001
2	0000 0010
126	0111 1110
127	0111 1111
-128	1000 0000
-127	1000 0001
-126	1000 0010
-2	1111 1110
-1	1111 1111

Computing with Integers

- Our simple calculations revisited:

- Unsigned 8-bit integers

$$255 + 1 = 0 \qquad 16 \times 17 = 16 \qquad 0 - 1 = 255$$

- Unsigned 64-bit integers

$$2^{63} + 2^{63} = 0$$

- Signed 8-bit integers

$$127 + 1 = -128 \qquad -128 \div -1 = -128$$

- In mathematics: $a + b \geq a$ for $b \geq 0$; as you can see, such obvious “facts” are no longer true

Integer Overflows

- Integer overflows can lead to buffer overflows
- Example (OS kernel system-call handler); string lengths checked to defend against buffer overflows:

```
char buf[128];
combine(char *s1, size_t len1,
        char *s2, size_t len2)
{
    if (len1 + len2 + 1 <= sizeof(buf)) {
        strncpy(buf, s1, len1);
        strncat(buf, s2, len2);
    }
}
```

len1 < sizeof(buf)

len2 = 0xffffffff

len2 + 1 = 2³² - 1 + 1 = 0 mod 2³²

strncat will be executed

“Safe” Signed Addition

```
if(!((rhs ^ lhs) < 0)) //test for +/- combo
{ //either two negatives, or two positives
    if(rhs < 0)
    { //two negatives
        if(lhs < MinInt() - rhs) // rhs is negative
        {
            throw ERROR_ARITHMETIC_OVERFLOW;
        }
    }
    else
    { //two positives
        if(MaxInt() - lhs < rhs) // lhs is positive
        {
            throw ERROR_ARITHMETIC_OVERFLOW;
        }
    }
}
//else overflow not possible
return lhs + rhs;
```

Do not add arguments which have the same sign

Integer Mismatches

```
unsigned int readdata() {  
    int amount = 0;  
    ...  
    if (result == ERROR)  
        amount = -1;  
    ...  
    return amount;  
}
```

- Note: `readdata()` returns an unsigned integer
- Variable `amount` can hold a negative value
- If the error condition is met, the return will be `4,294,967,295` on a system which uses 32-bit integers

Integer Mismatches

- Lesson: declare all integers as unsigned integers unless you really need negative numbers
- Compiler usually issues a warning if a signed-unsigned mismatch occurs
- Do not ignore these warnings lightly
- **Truncation** (Unix): input UID as signed integer, check value $\neq 0$, truncate to unsigned short integer:
`0x10000 → 0x0000 (root!)`

Units of Measurement

- Engineers should be familiar with the importance of using correct units of measurement
- Loss of Mars Climate Orbiter (1999):
 - *The peer review preliminary findings indicate that one team used imperial units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation*



Remember the Mars Climate Orbiter incident from 1999?

Miscalculating the size of data structures can lead to buffer overflows

Question

Hint: to get the number of elements in the array,
`sizeof(wszUserName) / sizeof(wszUserName[0])`

- What is the size in bytes of the buffer passed to **MultiByteToWideChar**?

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

    // do something
}
```

512: WCHAR
(wide character)
has 2 bytes

if a program only allocates a 256
byte buffer, an overrun will occur

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

Conclusions

- Dangers of abstraction:
 - In mathematics, integers are an infinite set
 - On computers, integers have finite precision
- Integer overflows happen when the behaviours of abstraction and implementation diverge
- Solution: place checks in the code or use libraries that **detect** such situations and flag an error; the caller has to handle those error messages correctly

A Final Word

Ashcraft & Engler [IEEE S&P 2002]:

“Many programmers appear to view integers as having arbitrary precision, rather than being fixed-sized quantities operated on with modulo arithmetic.”

Format String Attacks

Overwriting without overrunning

Widening the Target

- The story so far: modify **return address** with **buffer overflow** on stack
 - Assumption: attacker can fairly easily guess the location of this pointer relative to a vulnerable buffer
 - Defender knows which target to protect
- More powerful attack: **overwrite arbitrary pointer** with an **arbitrary value**
- More targets, so more difficult to defend against

Targeted Overwrites

- Buffer overflows: crude way of overwriting a variable
 - There has to be a vulnerable buffer below the target
 - All positions in between are also overwritten, so a program may crash before returning
- Can we surgically overwrite a target variable?
- Attack pattern: let a function with permission to write to the target do the job for the attacker
 - Format string attacks: `printf()` does the attacker's job
 - Double-free attacks: `malloc()` does the attacker's job

printf()

- “Print formatted”, library function for C programs:

```
printf format [ argument ... ]
```

- *format* is a **format string** containing **format tokens**
- Format tokens are **control data** specifying how the arguments should be displayed (see next slide)
- *argument* is a variable length list of arguments (**user data**)
- Ideally there is one format token for each argument, **but this is not required**

printf() Family

<code>printf(char *, ...);</code>	creates a formatted string and writes it to standard out I/O stream
<code>fprintf(FILE *, char *, ...);</code>	creates a formatted string and writes it to a libc FILE I/O stream
<code>sprintf(char *, char *, ...);</code>	creates a formatted string and writes it to a location in memory
<code>snprintf(char *, size_t, char *, ...);</code>	creates a formatted string and writes it to a location in memory, with a maximum string size

printf() – Format Tokens

<code>%i,</code> <code>%d</code>	<code>int, short,</code> <code>char</code>	Integer value of argument in decimal notation
<code>%u</code>	<code>unsigned int,</code> <code>short, char</code>	Value of argument as unsigned integer in decimal notation
<code>%x</code>	<code>unsigned int,</code> <code>short, char</code>	Value of argument as unsigned integer in hexadecimal notation
<code>%s</code>	<code>char *,</code> <code>char []</code>	Character string pointed to by the argument
<code>%p</code>	<code>(void *)</code>	Value of the pointer in hexadecimal notation
<code>%n</code>	<code>(int *)</code>	Number of bytes output so far, stored in an argument that is a pointer to an integer

printf() – Precision Field

```
char buf[10];  
int ctr, x = 0;  
snprintf(buf, sizeof buf,  
          "%.100d\n", x, &ctr);  
printf("counter: %d\n", ctr);
```

.precision: here 100
digits for rendering **x**

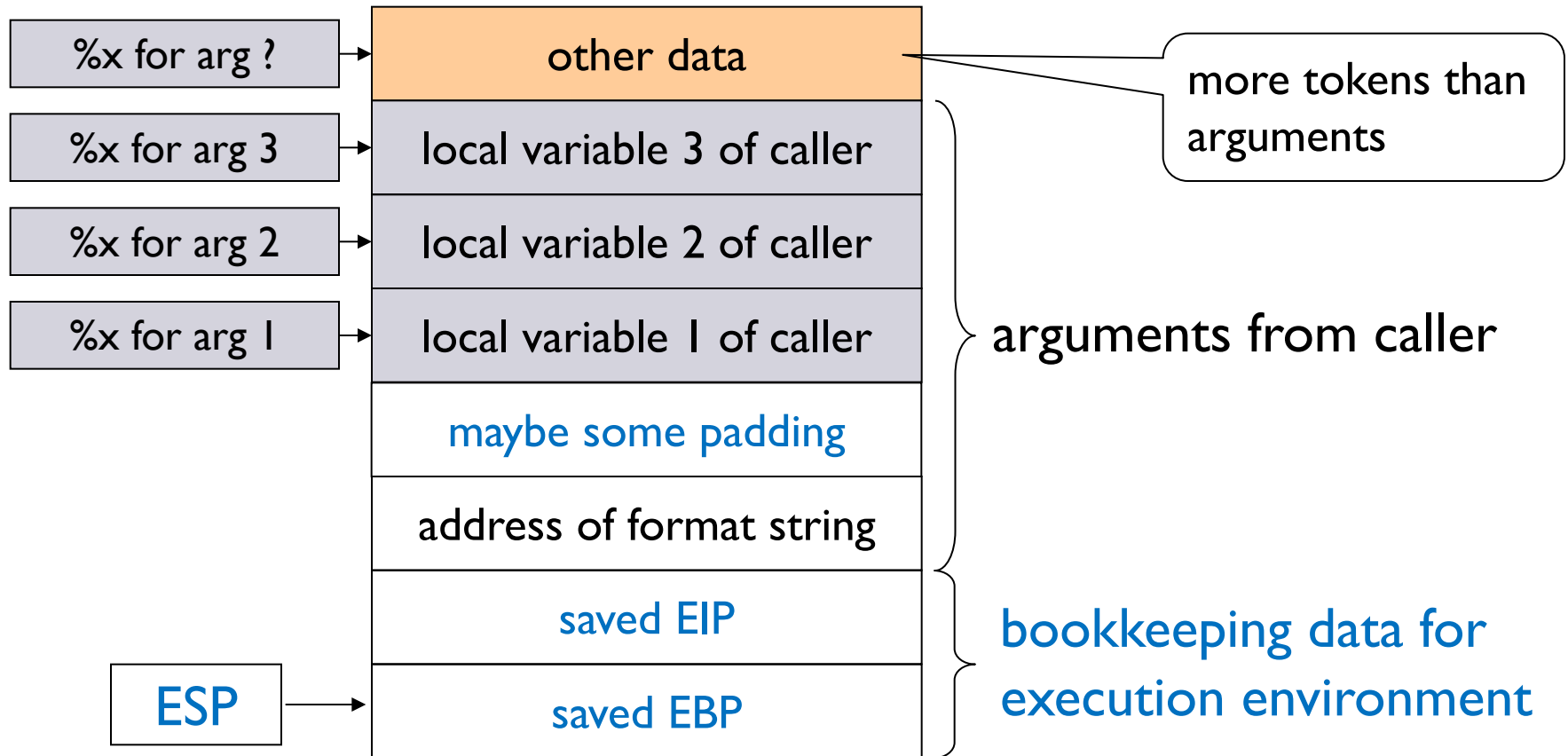
- **Counter** is incremented to 100, although only 10 characters are actually written to the buffer

Executing `printf()`

- We take a simplified look at the way `printf()` is executed; details depend on the compiler
- **Format string** & **arguments** allocated on the **stack**
 - **Stack frame** for `printf()` contains arguments passed to the print function
 - Assume **address of format string** is passed as an argument
- Number of arguments is calculated by counting the format tokens in the format string
- If there are **more tokens than arguments**, memory positions next in the stack frame will be printed

printf() – Stack Frame

```
printf("%x\n%x\n%x\n%x", arg1, arg2, arg3);
```



Reading Arbitrary Memory Locations

- Identify a format string function where you get to specify the format string
- When `printf()` is called without format tokens, an attack can pass its own format tokens to create a “ghost” stack frame
- To read from a chosen memory address, put this address as a constant in the format string
- Construct format string so that this constant becomes the argument for a `%s` format token in the ghost frame
 - Format string has to include format tokens for traversing the memory locations from the location the first argument is expected to be at to the address of the format string

Reading Arbitrary Memory Locations

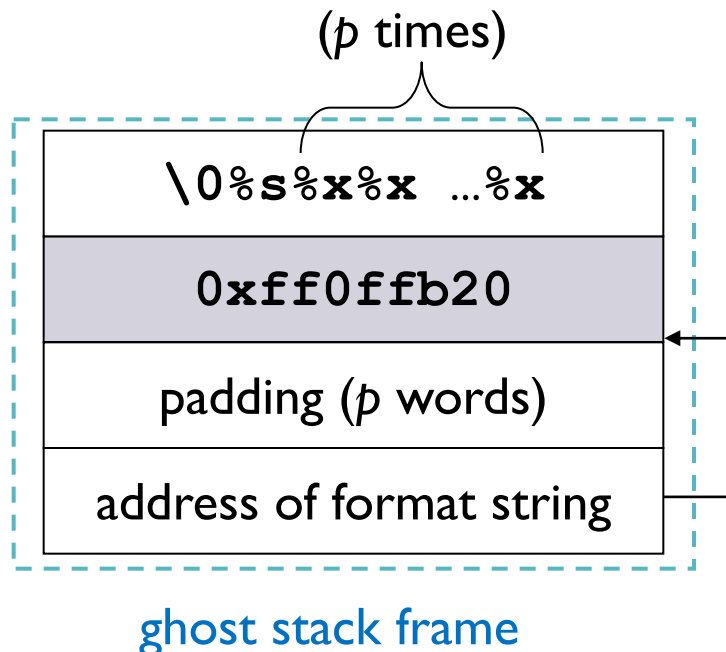
Attacker
controlled
buffer

```
printf("\x20\xfb\x0f\xff%x%x...%x%s");
```

address as constant

traverse
padding

print value at
address pointed to



- Target address given in format string in reverse byte order
- Compiler dependent padding
- Token %s applied to location containing target address
- (Padding and) value of location pointed to by target address are printed

Format String Attacks

- Format string exploit for WU-FTPD (Washington University FTP daemon), published June 2000
- Malign input passed instead of a format string argument to a function from the `printf()` family
 - `printf("%s", str)` prints argument as string
 - `printf(str)` provides unchecked input to `printf()` ; attacker can introduce new format tokens
 - Don't trust your inputs!
- Problem had been known but not thought to be security critical

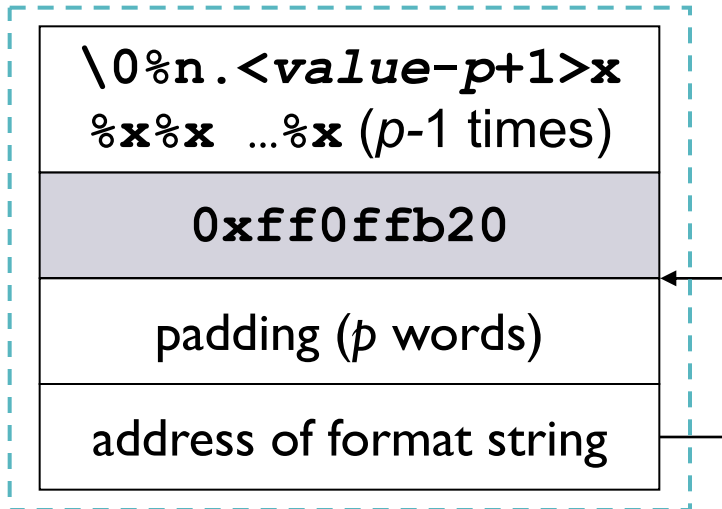
Writing to Arbitrary Memory Locations

- When `printf()` is called without format tokens, an attack can pass its own format tokens to create a “ghost” stack frame
- To write to a chosen memory address, put this address as a constant in the format string
- Construct format string so that this constant becomes the argument for a `%n` format token in the ghost frame
 - Include format tokens to traverse memory locations from the location the first argument is expected to be at to the address of the format string
 - Use `.precision` to set counter to value of your choice

Writing to Arbitrary Memory Locations

```
printf("\x20\xfb\xff%x%x...%x%.<value-p+1>x%n");
```

address as constant traverse padding increment counter write value to address pointed to



ghost stack frame

- Target address given in format string in reverse byte order
- Compiler dependent padding
- `%.<value-p+1>x` increments counter to desired value
- Token `%n` applied to location containing target address
- **value** is written to the location pointed to by the target address

Comments

- Frequent target: “bad” calls to `syslog()`, i.e., calls without a constant format string
 - `void syslog(int priority, const char *format, ...);`
 - Reference: <https://man7.org/linux/man-pages/man3/syslog.3.html>
 - **Bad call:** `syslog(LOG_AUTH, errmsg);`
- Defence (easy!):
 - Always include a constant format string so that user input is processed in the way expected

Code Example

- Format string vulnerability in the Unix screen utility (up to version 3.9.5) allowing attacker to overwrite UID of the process
- Format string bug in `screen.c` in function

```
serv_select_fn()
...
else if (visual && !D_VB && (!D_status ||
!D_status_bell))
{
    D_status_delayed = -1;
    Msg(0, VisualBellString);
    if (D_status)
    { ...
```

- `Msg()` feeds second argument to `sprintf()` but `VisualBellString` is user defineable

Resources

- Chapter 7 in J.C. Foster et al.: [Buffer Overflow Attacks](#), Syngress, 2005
- Tim Newsham: [Format String Attacks](#), Guardent Inc., Sept. 2000, <http://julianor.tripod.com/bc/tn-usfs.pdf>

Memory Management

Chunks and bins

Managing Memory in C

- Allocating memory:

- `void * malloc (size_t size)`

- Returns pointer to newly allocated block of *size* bytes

- Contents of the block are not initialized

- Returns null pointer if block cannot be allocated

Managing Memory in C

- Deallocating memory:
 - `void free (void *ptr)`
 - `*ptr` must have been returned by a previous call to `malloc()`, `calloc()` or `realloc()`
 - If `ptr` is `NULL`, no operation is performed
 - Behaviour undefined if `free(ptr)` has already been called
- De-allocating memory is not an atomic operation
 1. Mark memory as free
 2. Take pointer variable out of service

Memory Organization

Case study: Doug Lea malloc for 32-bit architecture

- Memory divided into **chunks**
- A chunk contains **user data** and **control data**
- Chunks allocated by **malloc** contain **boundary tags**
- Free chunks are placed in **bins**
 - A bin is a double linked lists
 - Several bins, for chunks of different sizes
- Boundary tag of a free chunk contains a forward and a backward pointer to its neighbours in the bin

Memory Organization

```
struct chunk {  
    int prev_size;  
    int size;  
    /* used of if free */  
    struct chunk *fd;  
    struct chunk *bk;  
}
```

Forward pointer to first chunk in list

Backward pointer to last chunk in list

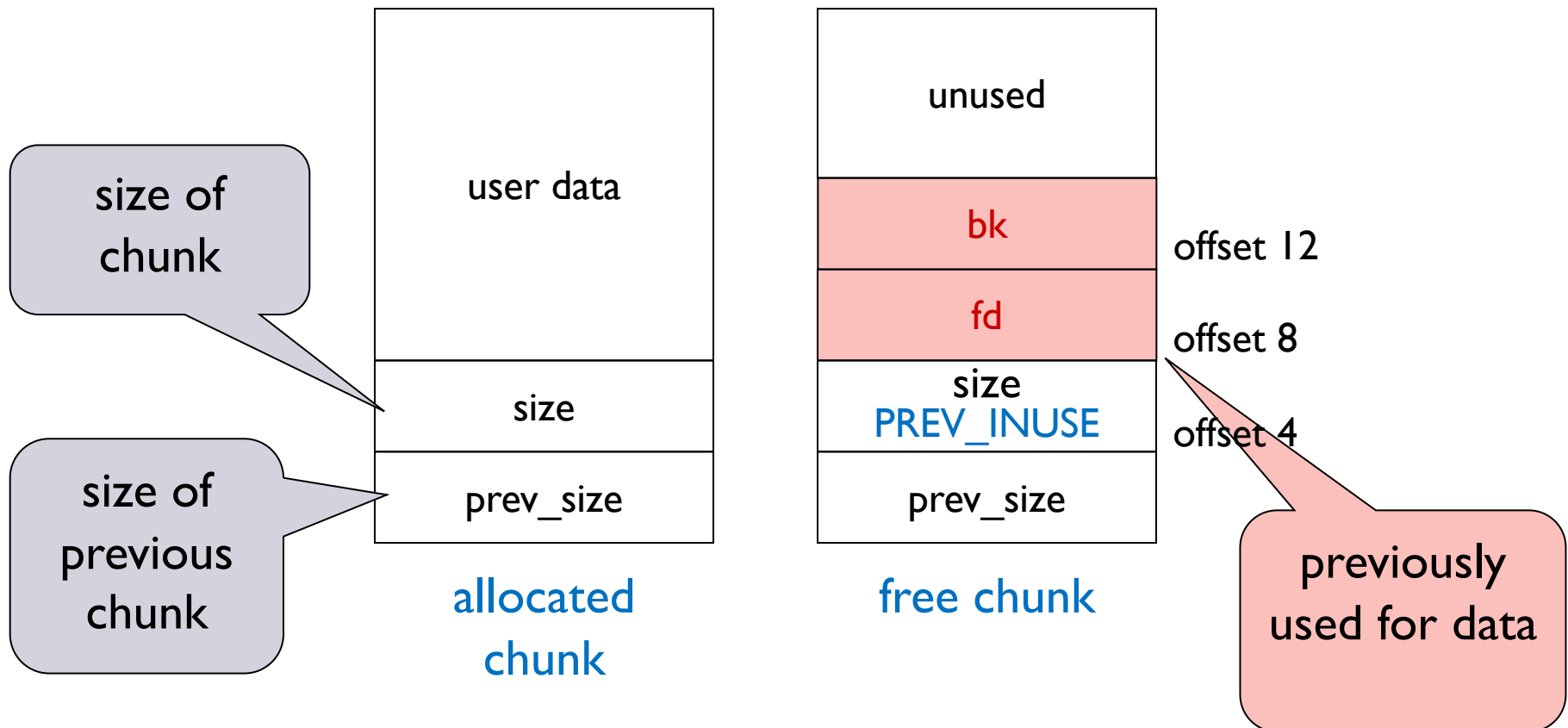


free

allocated

Allocated and Free Chunks

```
struct chunk {  
    int prev_size;  
    int size;  
    /* used of if free */  
    struct chunk *fd;  
    struct chunk *bk;  
}
```

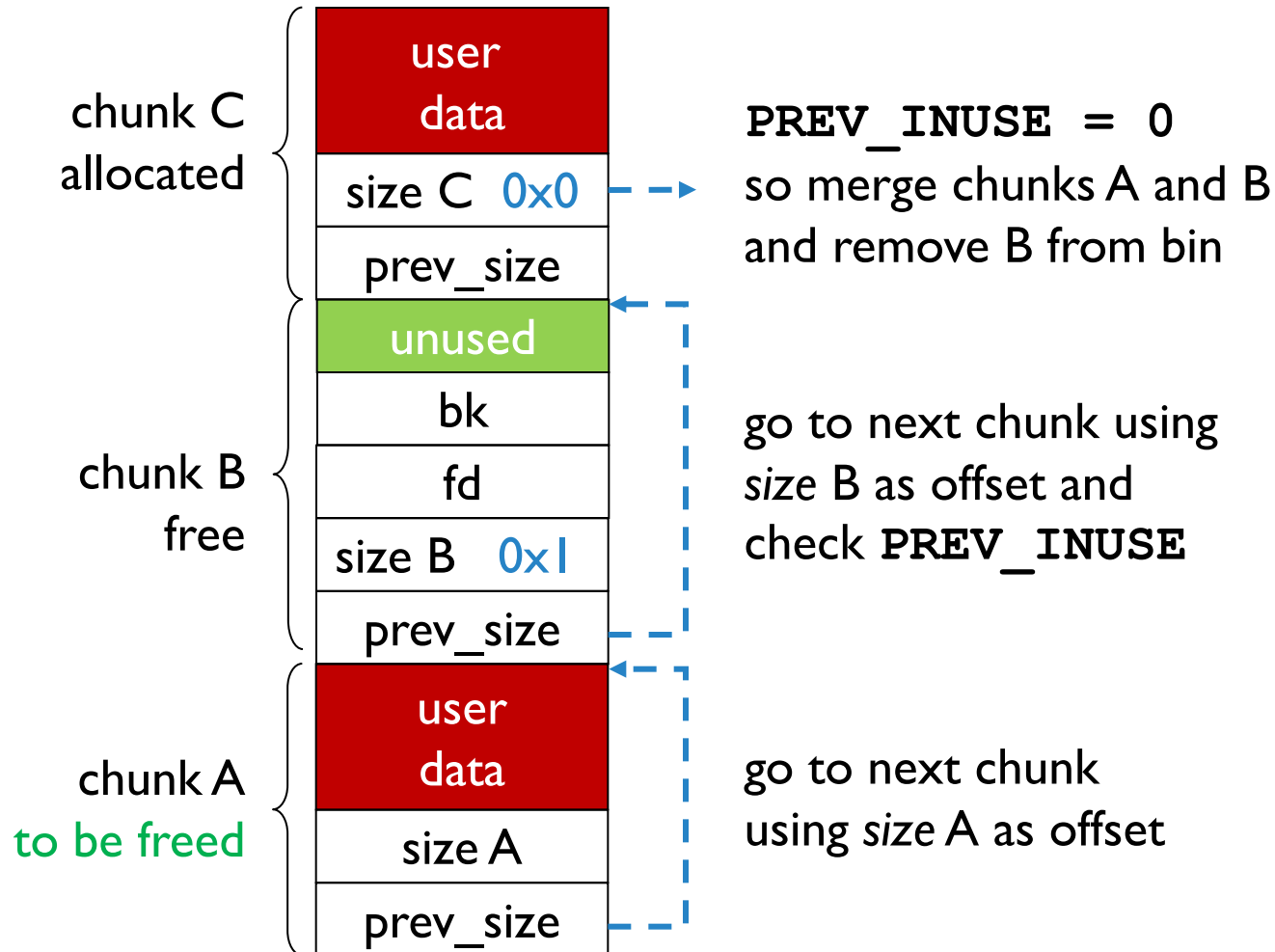


Control Flags

- Values for **size** always given in multiples of 8 (convention)
- Three least significant bits of **size** used as **control flags**
 - **PREV_INUSE** **0x1**
 - **IS_MAPPED** **0x2**
 - Some libraries also use the third bit
- When a chunk is freed it is coalesced into a single chunk with neighbouring free chunks
 - **No adjacent free chunks in memory**
- Technicality: **prev_size** is not used when the previous chunk is allocated

Coalescing Chunks

Hint: forward consolidation is shown here

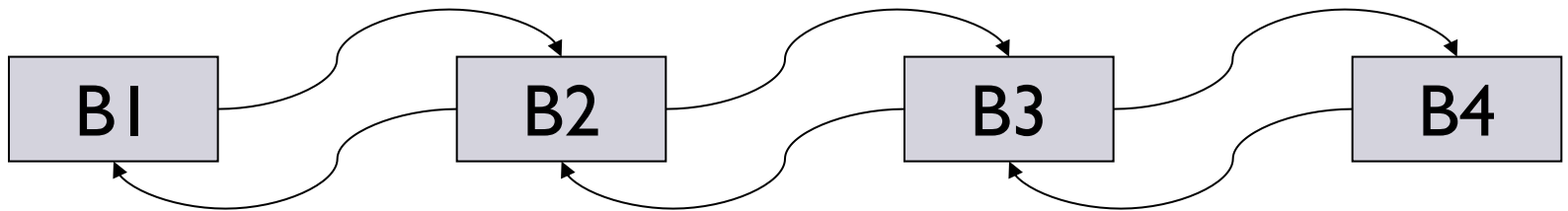


Two Views on Memory

- “Topological” (physical) view: location of chunks in memory
 - Chunks ordered by their addresses
 - We can talk about the “chunk above” and the “chunk below”
 - This view matters when chunks are coalesced
- “Logical” view: location of chunks in a bin
 - Chunks ordered by their position in a list
 - We can talk about “previous chunk” and “next chunk”
 - This view matters when chunks are allocated and freed

Managing a Bin

- **Bin**: double-linked lists of **free chunks**, ordered by **increasing size** to facilitate fast smallest-first search
- To allocate a buffer, take a suitable chunk from the list using **unlink()**
- When a chunk is freed, insert it in the right position in the list using **frontlink()**



Frontlink() – Simplified

```
#define frontlink(A, P, S, IDX, BK, FD) ...
[1] FD = start_of_bin(IDX);
[2] while ( FD != BK && S < chunksize(FD) )
    {
[3]     FD = FD->fd;
    }
[4] BK = FD->bk;
[5] P->bk = BK;
[6] P->fd = FD;
[7] FD->bk = BK->fd = P;
}
```

Store chunk of size **S**, pointed to by **P**, at appropriate position in the double linked list of bin with index **IDX**

Frontlink () Macro

[1] **FD** initialized with a pointer to the start of the list of the given bin

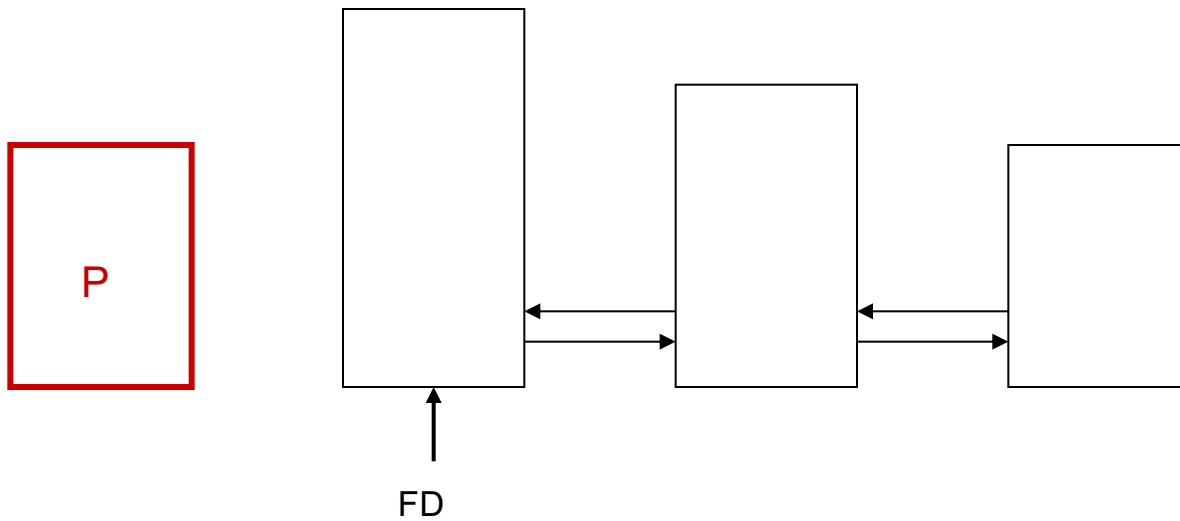
[2] Loop searches the double linked list to find first chunk not larger than **P** or the end of the list by following consecutive forward pointers (line 3)

[4] Follow back pointer **BK** to previous element in list

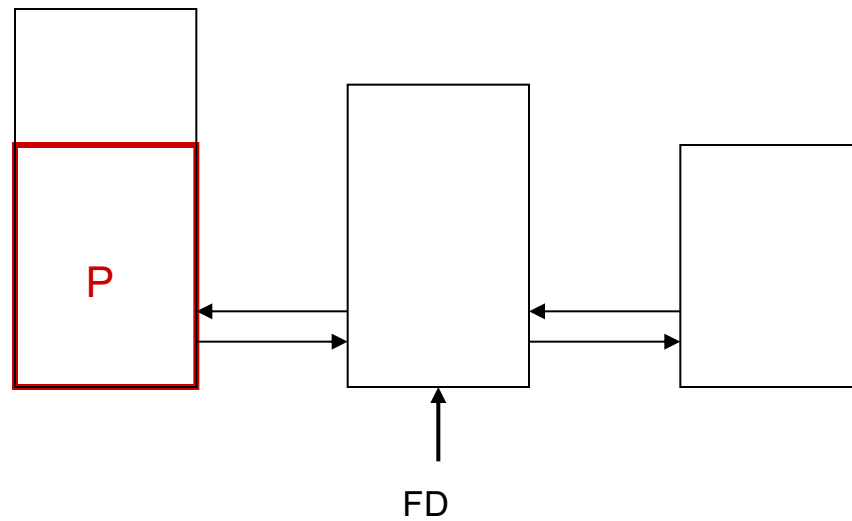
[5]+[6] Set backward and forward pointers for chunk **P**

[7] Update backward pointer of next chunk & forward pointer of previous chunk to address of chunk **P** (field **fd** at 8 byte offset within a boundary tag)

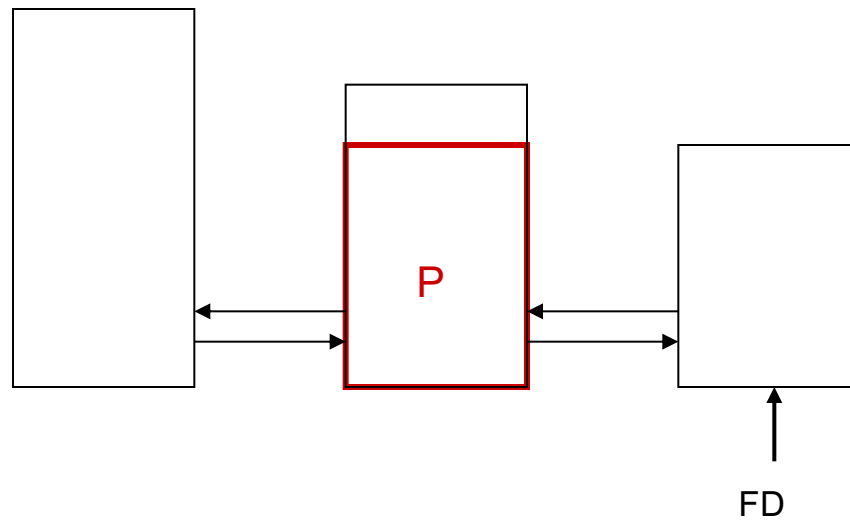
Frontlink()



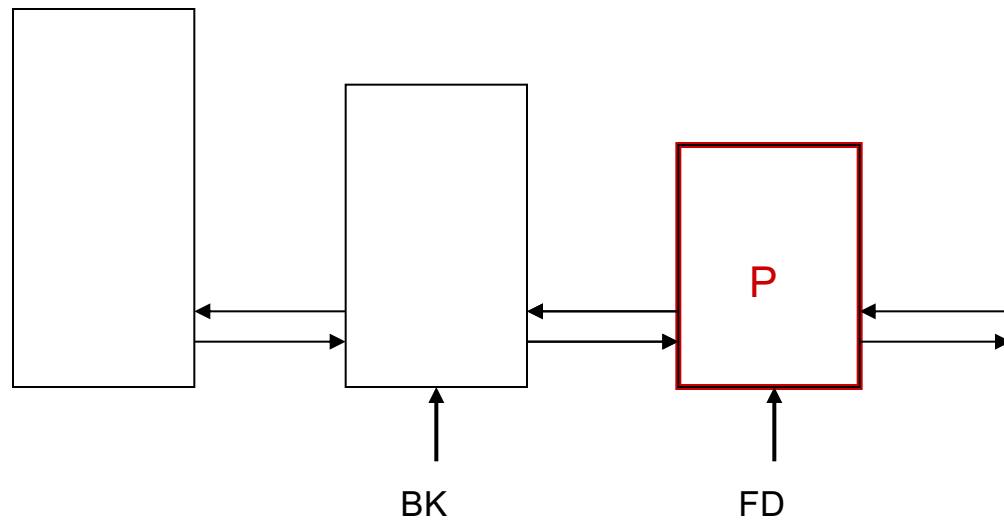
Frontlink()



Frontlink()



Frontlink()



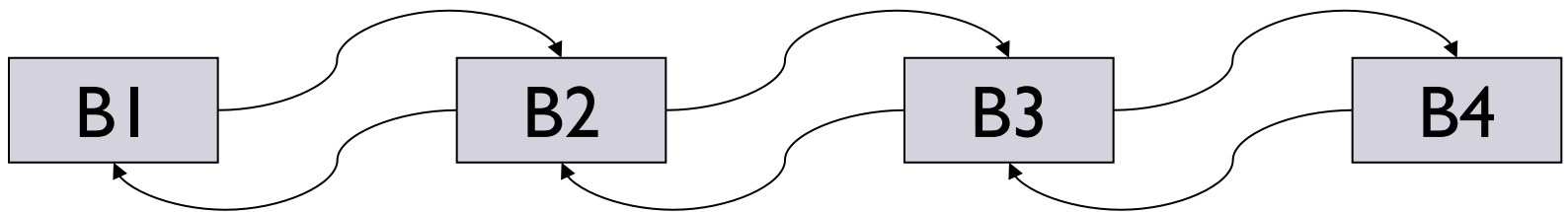
Unlink

```
#define unlink(P, BK, FD)
{
[1]  FD = P->fd;
[2]  BK = P->bk;
[3]  FD->bk = BK;
[4]  BK->fd = FD;
}
```

- [1] [2] Save pointers in chunk **P** to **FD** and **BK**
- [3] Update backward pointer of next chunk in the list: address located at **FD** plus 12 bytes (offset of *bk* field in boundary tag) overwritten with value stored in **BK**
- [4] Update forward pointer of the previous chunk

Mental Exercise

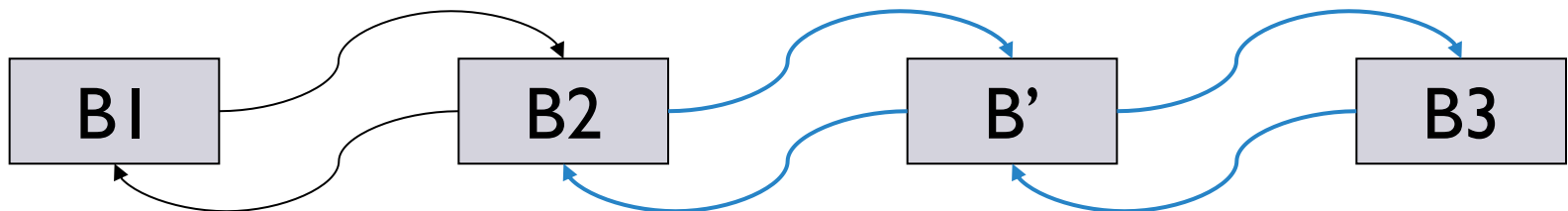
- What will happen when freeing a chunk that has already been freed?
- Add a chunk B' into the following list
- Then add it again



Insert B' before B3

Assume loop terminates with FD equal to B3

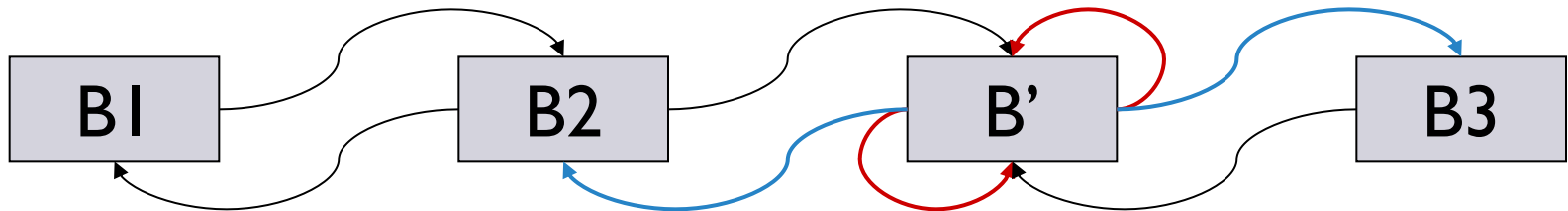
[4] BK = FD->bk;	FD = B3;	BK = B2;
[5] P->bk = BK;		B' ->bk = B2;
[6] P->fd = FD;		B' ->fd = B3;
[7] FD->bk = BK->fd = P;	B3->bk = B2->fd = B' ;	



Insert chunk B' again

Loop ends with FD equals to **B'**

[4] BK = FD->bk;	FD = B' ;	BK = B2 ;
[5] P->bk = BK;		B' ->bk = B2 ;
[6] P->fd = FD;		B' ->fd = B' ;
[7] FD->bk = BK->fd = P;	B' ->bk = B2->fd = B' ;	



Unlink double-free'd chunk B'

[1] $FD = P \rightarrow fd;$

[2] $BK = P \rightarrow bk;$

[3] $FD \rightarrow bk = BK;$

[4] $BK \rightarrow fd = FD;$

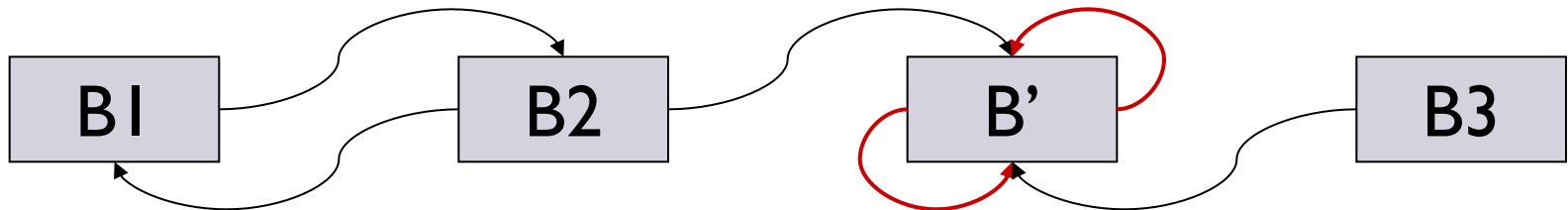
$FD = B' \rightarrow fd = B'$

$BK = B' \rightarrow bk = B'$

$FD \rightarrow bk = B' \rightarrow bk = B'$

$BK \rightarrow fd = B' \rightarrow fd = B'$

Nothing changes: the chunk to be removed from the list of free chunks is still on the list!

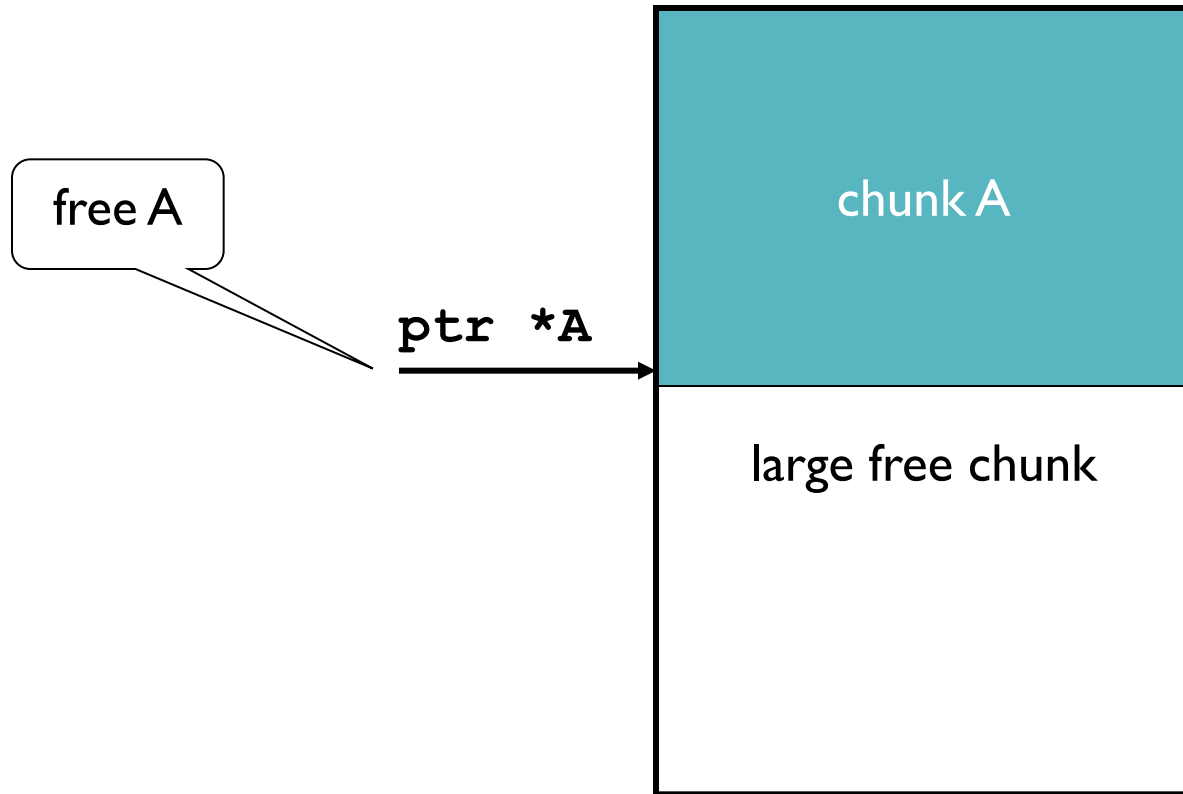


Double-Free Vulnerabilities

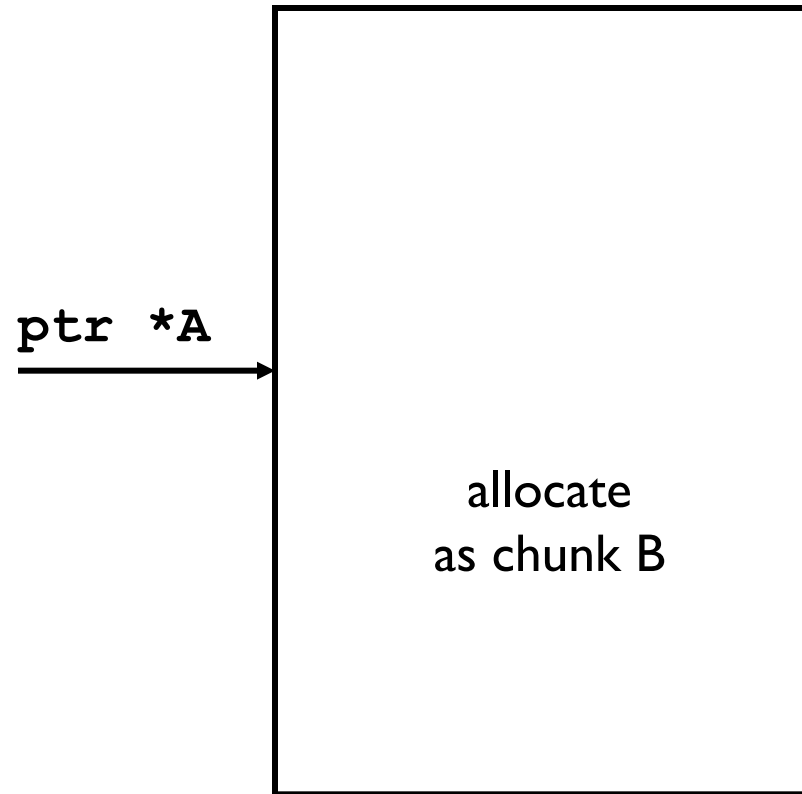
Double-Free Attack

- Allocate memory chunk **A**
- Call **free (A)** , with forward consolidation to create larger chunk
- Allocate large chunk **B**; **hope to get space just freed**
- Copy **ghost chunk** into **B** at the location of **A** and a **free ghost chunk** adjacent to the chunk at **A**
- Call **free (A)** again; coalescing the two ghost chunks will try to remove the free ghost chunk from its bin

Double-free Attack



Double-free Attack

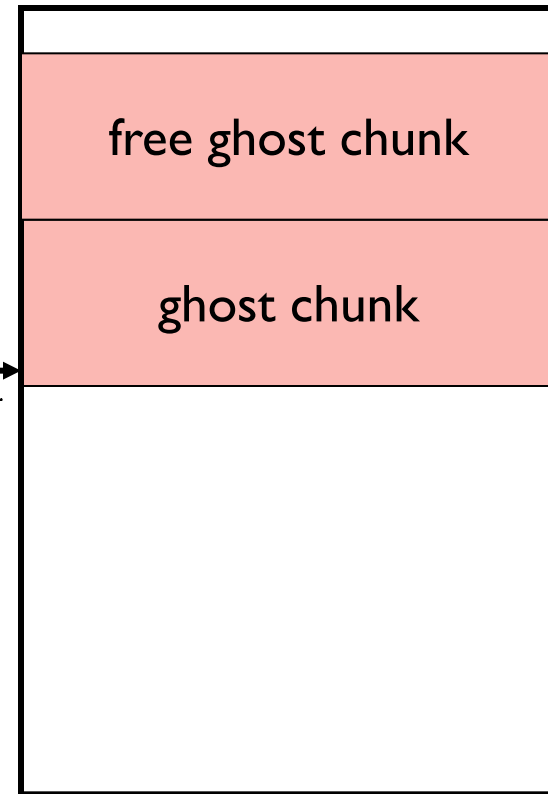


Double-free Attack

now free A again; ghost chunk at A is coalesced with neighbouring free ghost chunk; **unlink** is applied to **free ghost chunk**

ptr *A

write ghost chunks into B
(legitimate as B has been allocated)



Removing the Ghost Chunk

- Attacker writes **fake forward and backward pointers** into the first eight bytes of the free ghost chunk
 - ***fd*** : **target address** to be overwritten, **minus 12**
 - ***bk*** : **value to be written** to the target address
- Unlinking the free ghost chunk uses the fake pointers ***fd*** and ***bk***
$$FD = fd$$
$$BK = bk$$
$$fd \rightarrow bk = bk \quad (\text{field } bk \text{ is at offset } +12)$$
- The value ***bk*** is written to the target address ***fd+12***

Double-free Attack

- **Deallocation** of heap memory is not atomic!
- Second call of **free (A)** only has the desired effect if the pointer to **A** had not been set to null when **A** was freed the first time
 - *free(ptr)* : If *ptr* is null, no operation is performed
- **Fault: a function releases a memory chunk but does not set the respective pointer to null**
- Attack is possible when a calling application happens to free the same chunk again

Double-free Vulnerabilities

- Double Free Bug in zlib Compression Library (v1.1.3)
 - [CERT® Advisory CA-2002-07](#)
- MySQL double free()
 - CAN-2003-0073
- Microsoft IE invalid GIF double-free vulnerability
 - Discovered: September 2, 2003
- Linux: CVS double free vulnerability
 - Advisory number: CSSA-2003-006.0
- Vulnerabilities in MIT Kerberos 5
 - TA04-247A, September 3, 2004
 - kadmind daemon, CVE-2007-1216, April 3, 2007
- PHP session_regenerate_id(): MOPB-22-2007

Double-free – Defences

- Stay true to abstraction: do not apply **unlink** to a chunk that is not part of a double link list
 - E.g., do not unlink chunk **p** if
$$!(p \rightarrow fd \rightarrow bk == p \rightarrow bk \rightarrow fd == p)$$
- Protect boundary tag with **canaries**
 - When a chunk is allocated, initialize canary (checksum over memory location, field sizes, etc.)
 - When a chunk is freed, recalculate checksum and compare with canary
- **Randomize** memory allocation so that attacker cannot predict the next chunk that will be allocated
- Split user data from control data
 - Put control information about chunks into a memory region not accessible from user space

Conclusions

The Wider Picture

- System V malloc differs from Doug Lea malloc, but also stores control information with user data
- Memory allocation systems other than Doug Lea malloc have similar problems
- Problem: no clean abstraction; overwriting user data gives an opportunity for changing control data
- Versions of malloc that separate user data from control information exist
- Similar issues in Windows memory management
 - <http://www.symantec.com/connect/blogs/double-free-vulnerabilities-part-2>

Conclusions

- Attack pattern: take a function that does some bookkeeping for the runtime system and thus is permitted to write to memory
 - Can be as trivial as counting characters in `printf()`
 - Can be more sophisticated as managing double-linked lists
- Present malign “ghost” inputs to this function
 - Either directly as inputs as in `printf()`
 - Leave inputs behind to be picked up later, as in `unlink()`
- Defence: ensure that function will only process the inputs it has been designed for

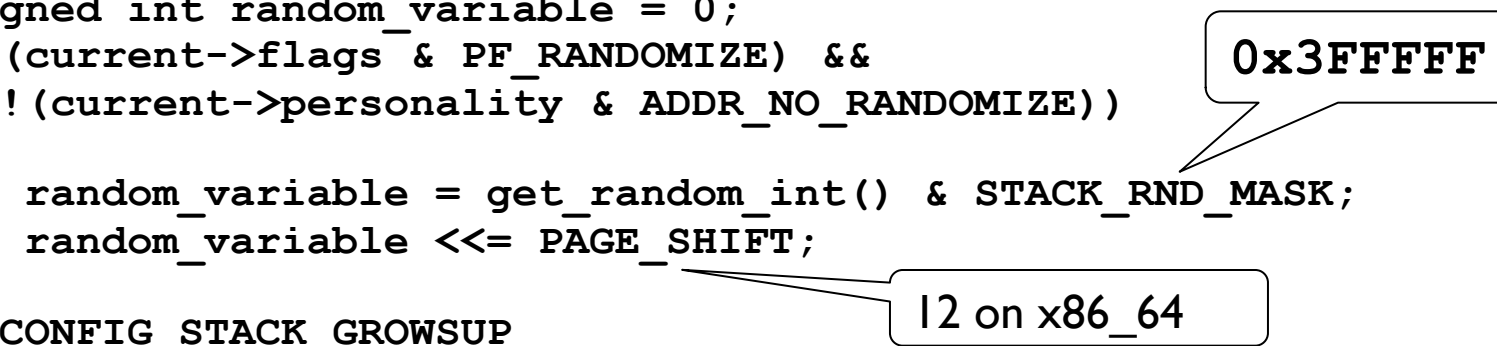
Tutorial

Issues with integers and heap feng shui

Working with Integers

- Spot the problem in this ASLR routine, and its impact

```
static unsigned long randomize_stack_top(unsigned long stack_top)
{
    unsigned int random_variable = 0;
    if ((current->flags & PF_RANDOMIZE) &&
        !(current->personality & ADDR_NO_RANDOMIZE))
    {
        random_variable = get_random_int() & STACK_RND_MASK;
        random_variable <=< PAGE_SHIFT;
    }
#ifdef CONFIG_STACK_GROWSUP
    return PAGE_ALIGN(stack_top) + random_variable;
#else
    return PAGE_ALIGN(stack_top) - random_variable; #endif
}
```



The diagram illustrates the problem in the ASLR routine. A call to `get_random_int()` returns the value `0x3FFFFFF`. This value is then shifted right by 12 bits (indicated by `<=< PAGE_SHIFT` where `PAGE_SHIFT` is 12 on x86_64) to produce the final `random_variable` value. This shift operation is the source of the vulnerability, as it does not properly mask the random value to the stack's randomization range.

Integer Issues & Double-Free

- Analyze the Stagefright Android security vulnerability
 - <https://s3.amazonaws.com/zhafiles/Zimperium-Handset-Alliance/Joshua+Drake+-+Stagefright+Scary+Code+in+the+Heart+of+Android-slides.pdf> (note slides 46-49)
 - <https://www.usenix.org/conference/woot16/workshop-program/presentation/drake> (note slide 9)
- Describe the general pattern of a double-free attack that follows a free-free-malloc-malloc pattern
 - A free-free-malloc-malloc double free attack corrupts the bin; is the same true for the malloc-free-malloc-free attack pattern?
 - Which steps are deterministic, where does the attacker need some luck to proceed?