Essential C Security 102

Offensive Computer Security
Florida State University
Spring 2014

Outline of Talk

- Continuation of Heap / Dynamic Memory discussion
- Integer Security
- Formatted Output
- Concurrency and Race Conditions

Tool we will be using

http://gcc.godbolt.org/

A project that visualizes C/C++ to Assembly for you. *(use g++ compiler, intel syntax, and no options)*

Quite useful for learning this stuff

(also interesting: https://github.com/ynh/cpp-to-assembly)

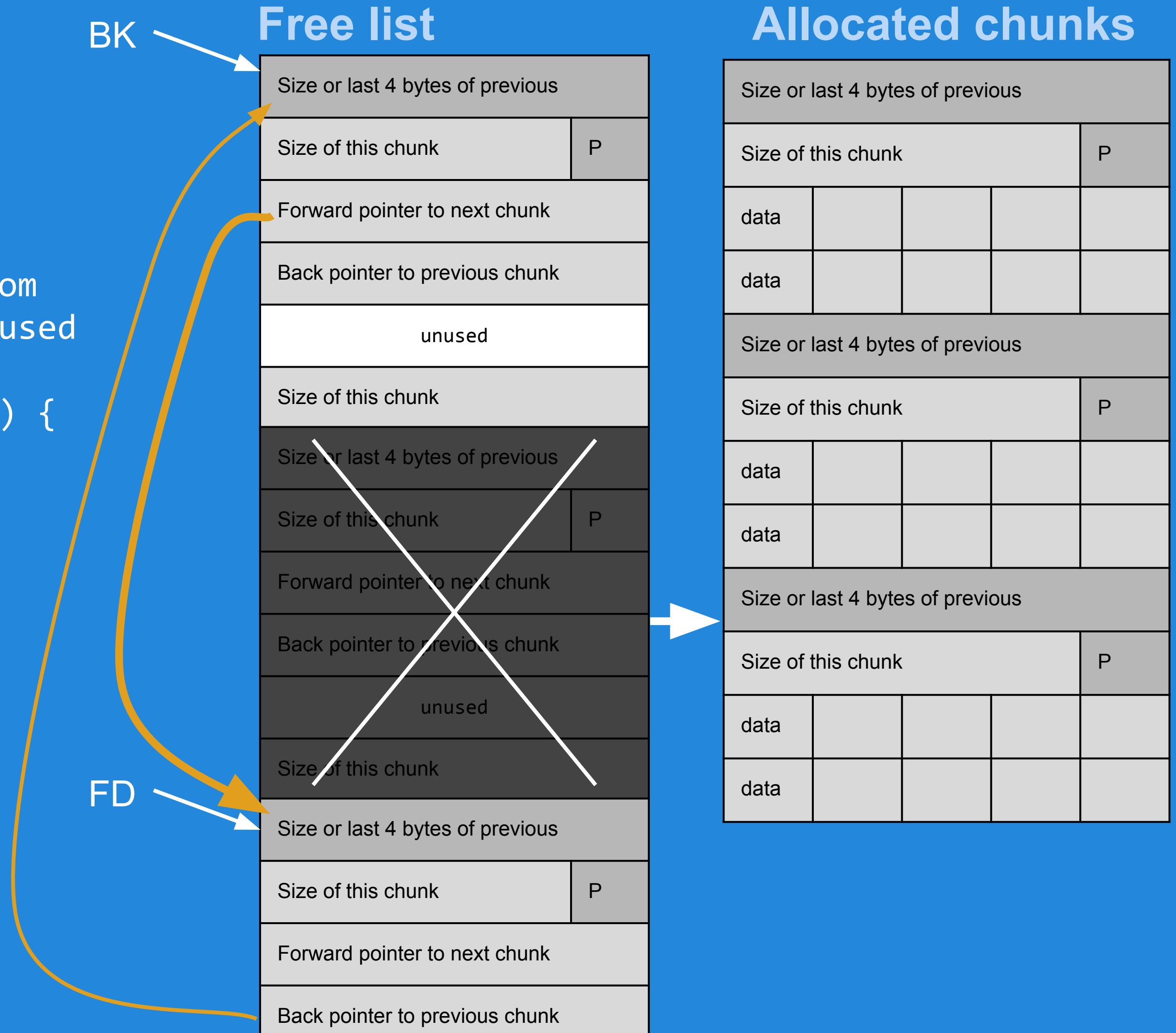
Memory Allocator

The memory manager on most systems runs as part of the process

- linker adds in code to do this
    - usually provided to linker via OS
        - OS's have default memory managers
            - compilers can override or provide alternatives
- Can be statically linked in or dynamically

Done! unlinked!

```
// This moves a chunk from
// the free list, to be used

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



Memory Allocator

In general requires:

- A maintained list of free, available memory
- algorithm to allocate a contiguous chunk of n bytes
  - Best fit method
    - chunk of size m >= n such that m is smallest available
  - First fit method
- algorithm to deallocate said chunks (free)
  - return chunk to list, consolidate adjacent used ones.

Memory Allocator

Common optimizations:

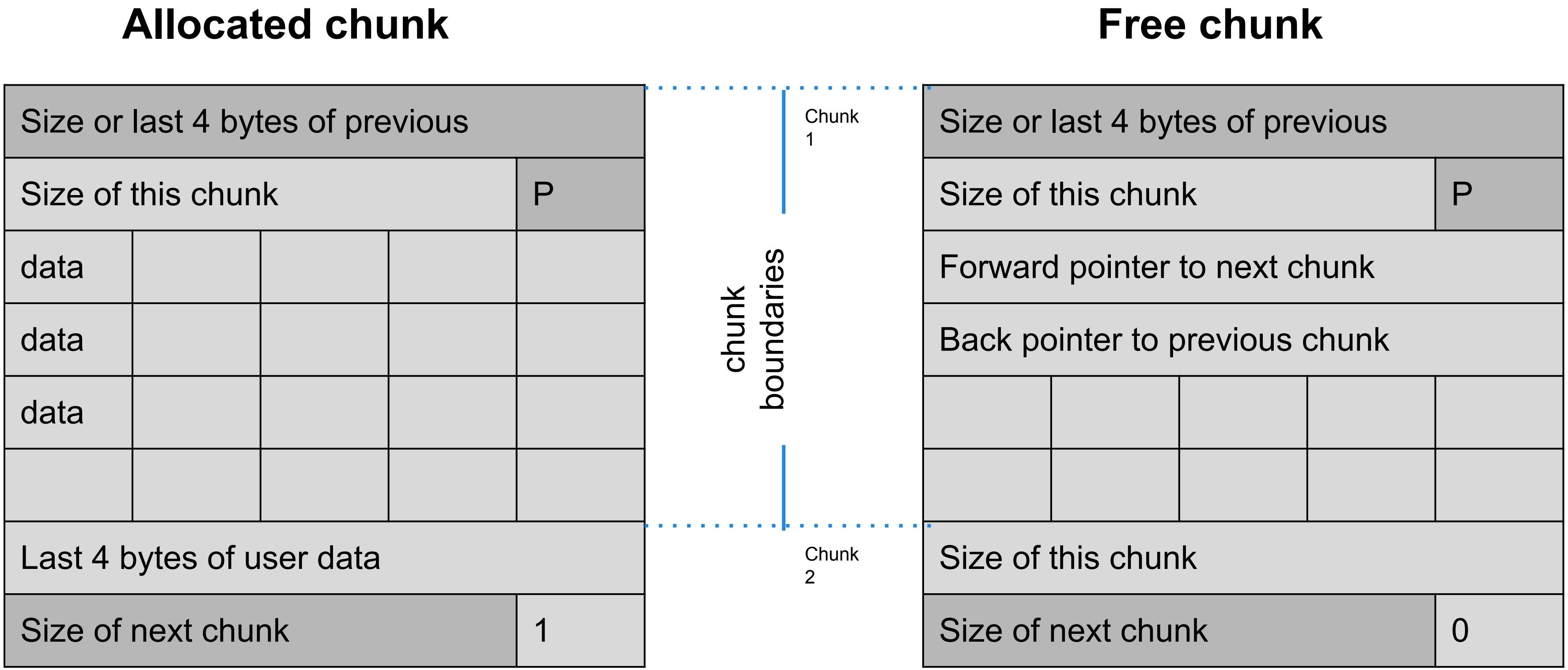
- Chunk boundary tags
  - [tag][------------chunk ---------][tag]
    - tag contains metadata:
      - size of chunk
      - next chunk
      - previous chunk (like a linked list sometimes)

Doug Lea’s dlmalloc allocator

Basis of Linux mem allocators

- Free chunks are arranged in a doubly-linked circular lists (**bins**)
- Each chunk (used and free) has:
  - next chunk and previous chunk pointers
  - size of previous chunk (if free) / last 4 bytes of the previous used chunk (if not free)
    - *Last 4 bytes is a mystery to me, don’t ask me*
  - flag for if previous chunk is used / free

Doug Lea’s dlmalloc allocator

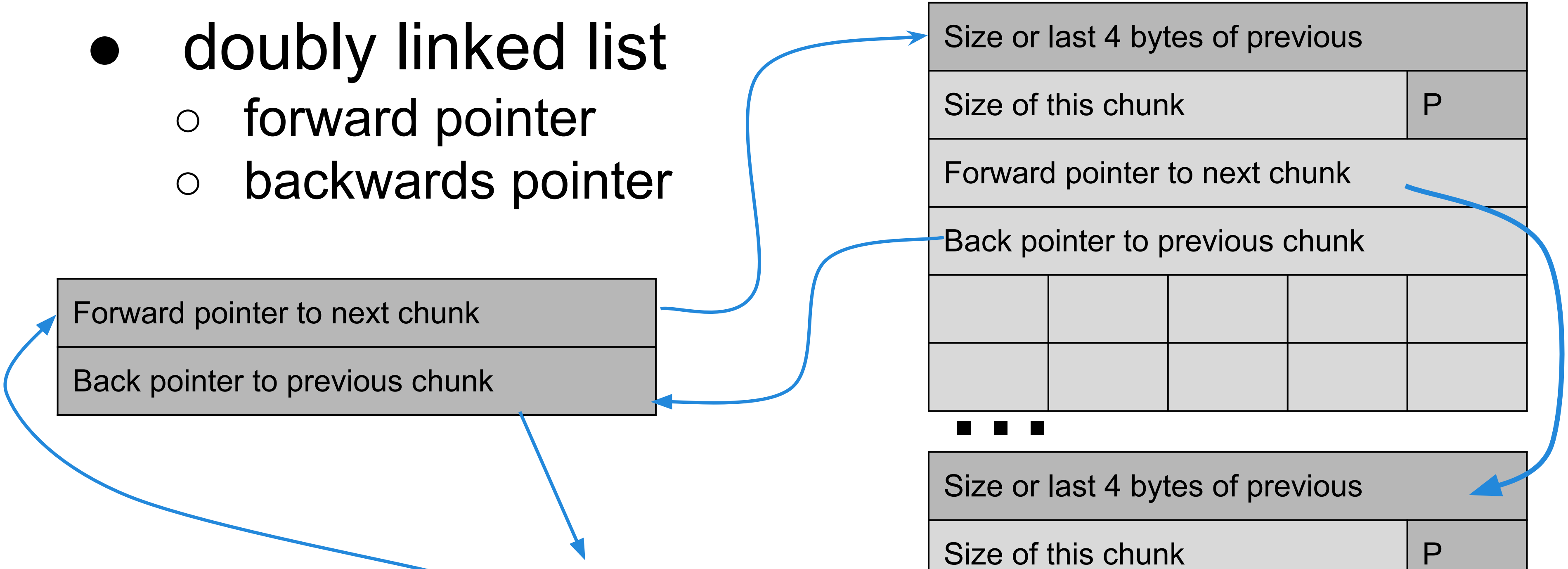


Doug Lea's dlmalloc allocator

Each list of free bin has a head:

- doubly linked list
  - forward pointer
  - backwards pointer

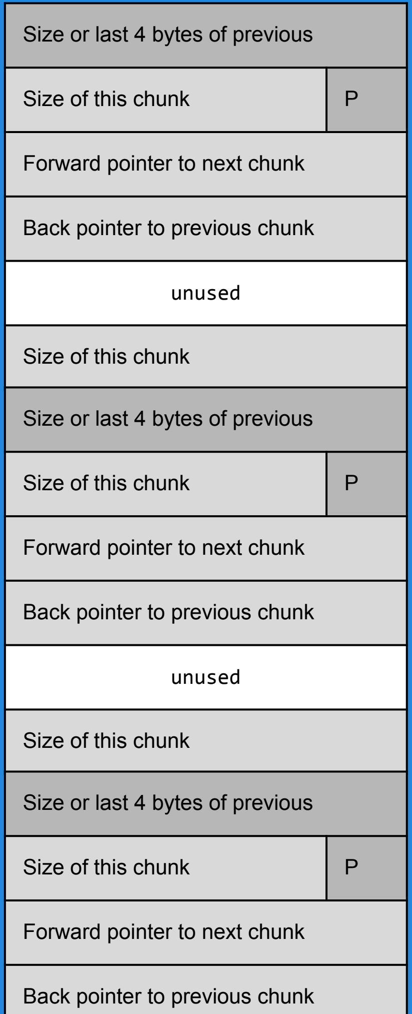
Free chunk



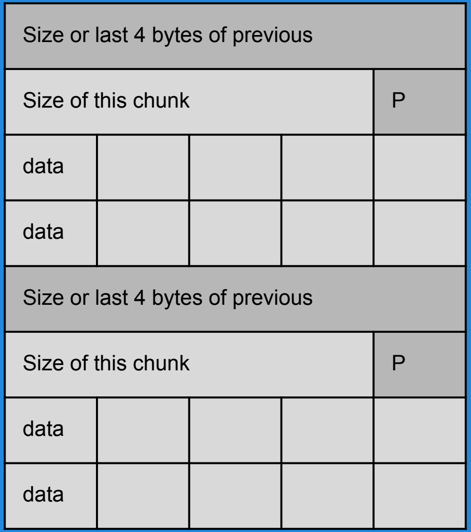
How unlinking works

```
// This moves a chunk from
// the free list, to be used

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

//This is from  [1]p 184
```

Free list

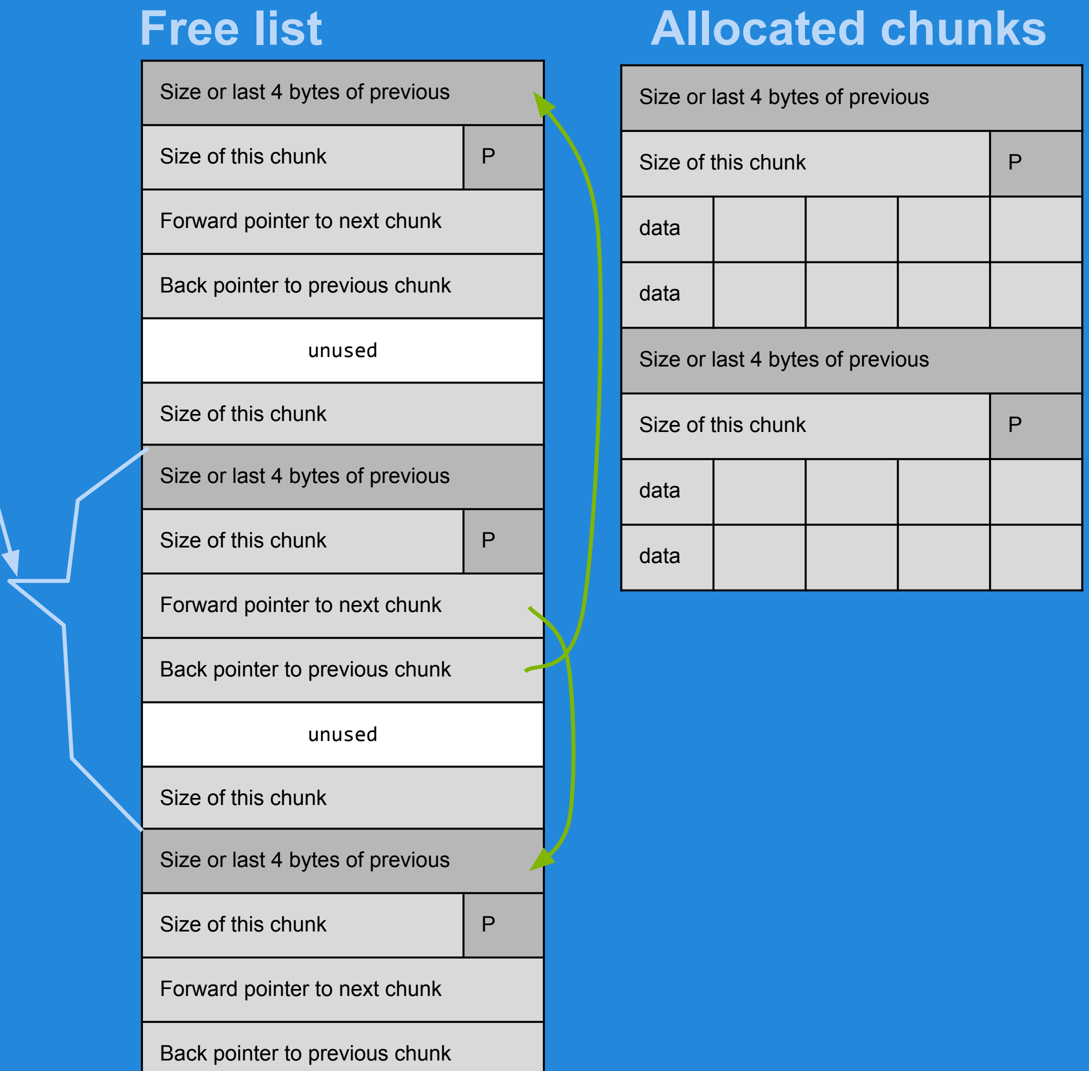


Allocated chunks



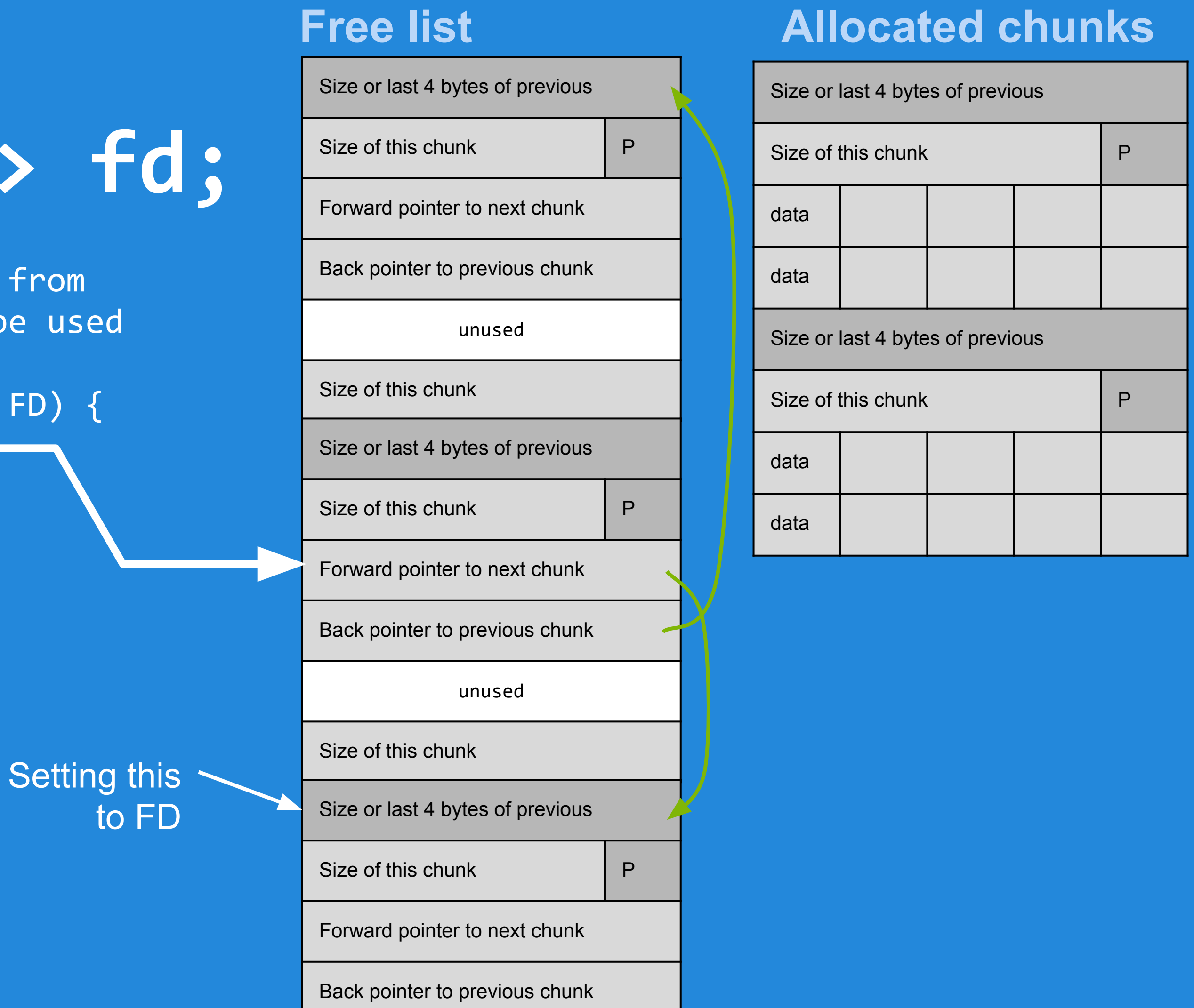
Say this is P

```
// This moves a chunk from
// the free list, to be used

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



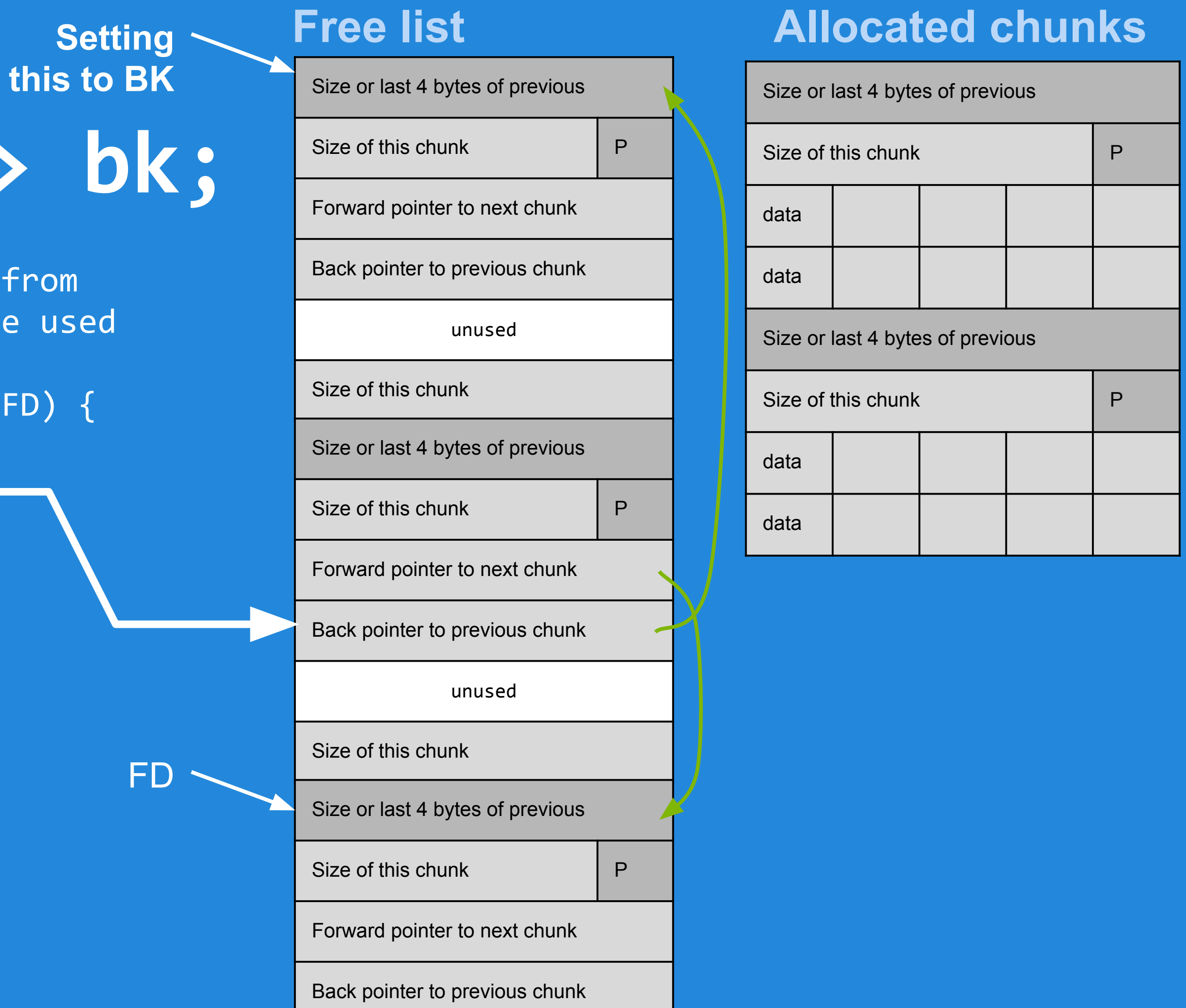
1)FD = P-> fd;

```
// This moves a chunk from
// the free list, to be used

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



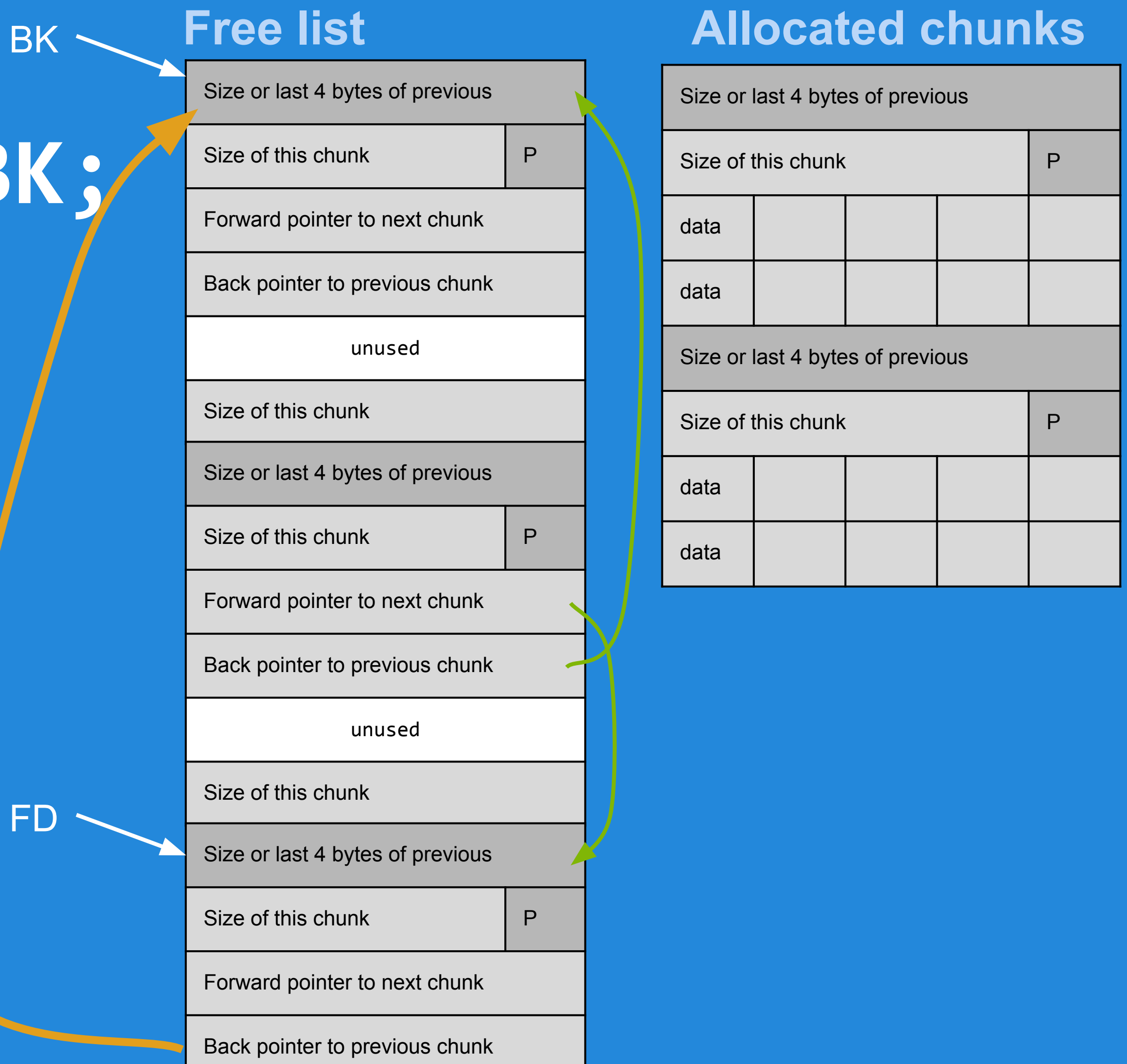
2)BK = P-> bk;

```
// This moves a chunk from
// the free list, to be used

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



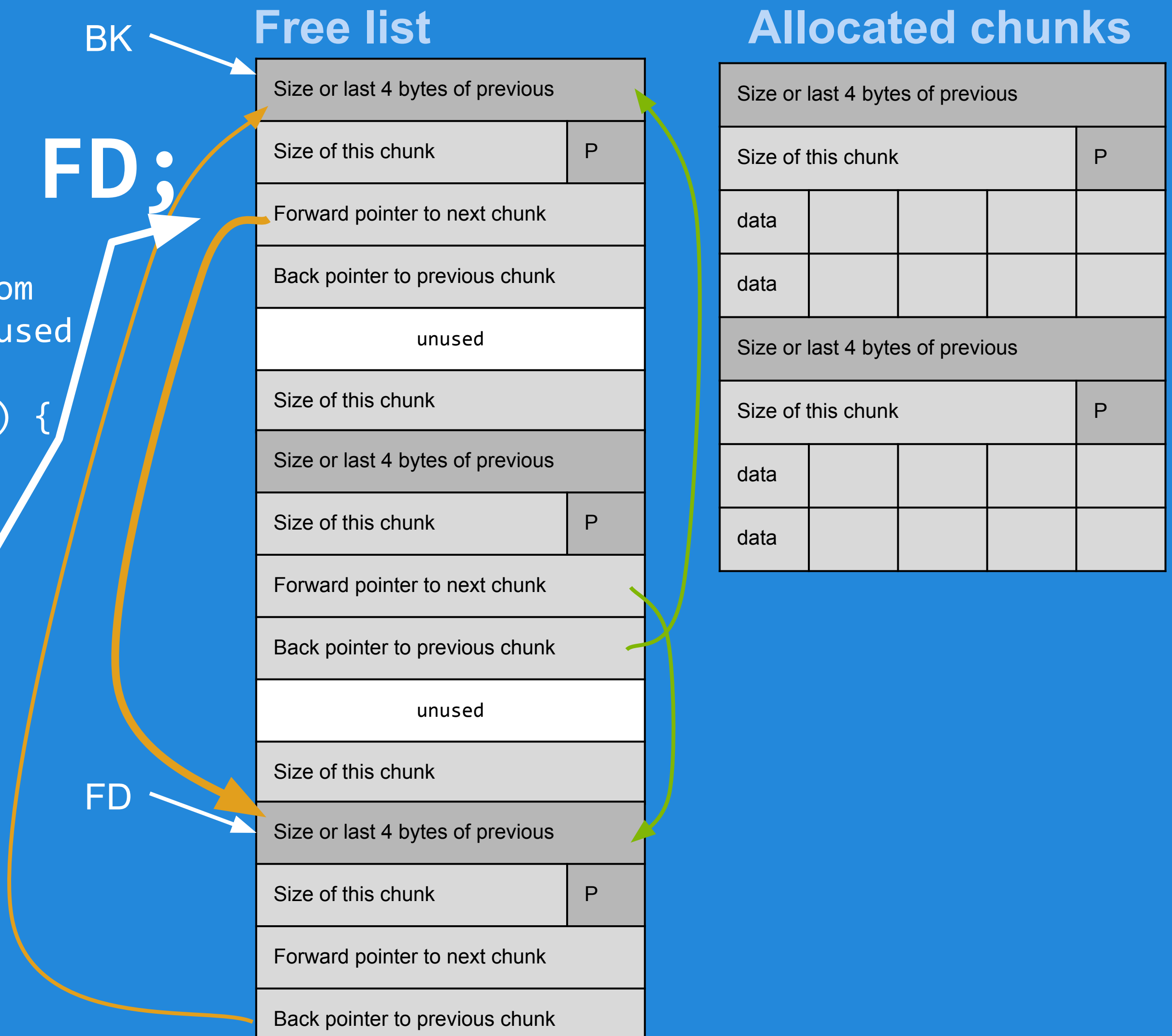
3)FD->bk = BK;

```
// This moves a chunk from
// the free list, to be used

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



4)BK->fd = FD;

```
// This moves a chunk from
// the free list, to be used

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



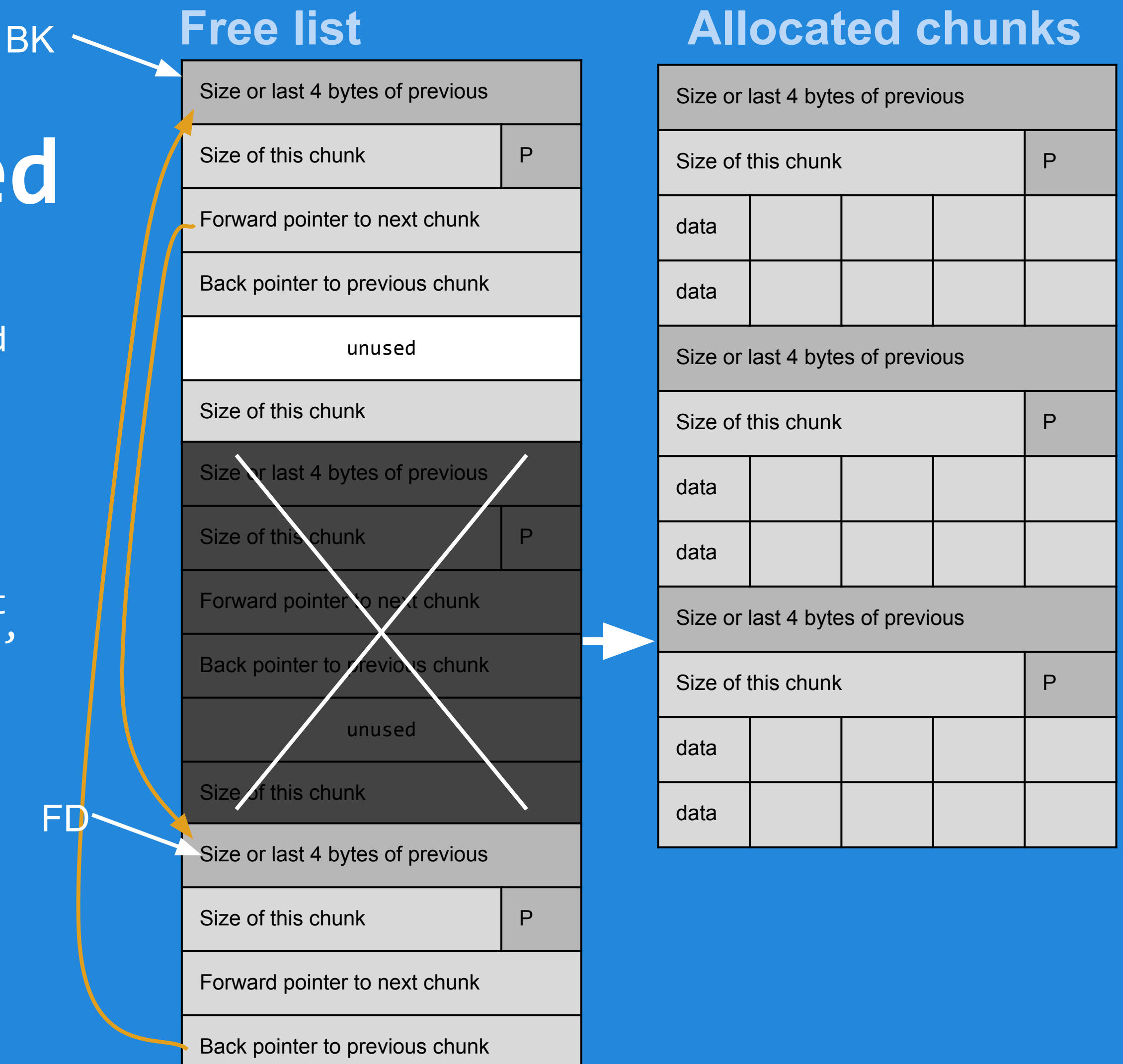
Chunk is now Allocated

Things to note:

- Two pointers are changed
  - BK->fd
  - FD->bk
    - Keep this in mind
- This trusts the data in the system to work right
  - double malloc doesn't mess this up
    - not a bug

free() is the reverse of this process

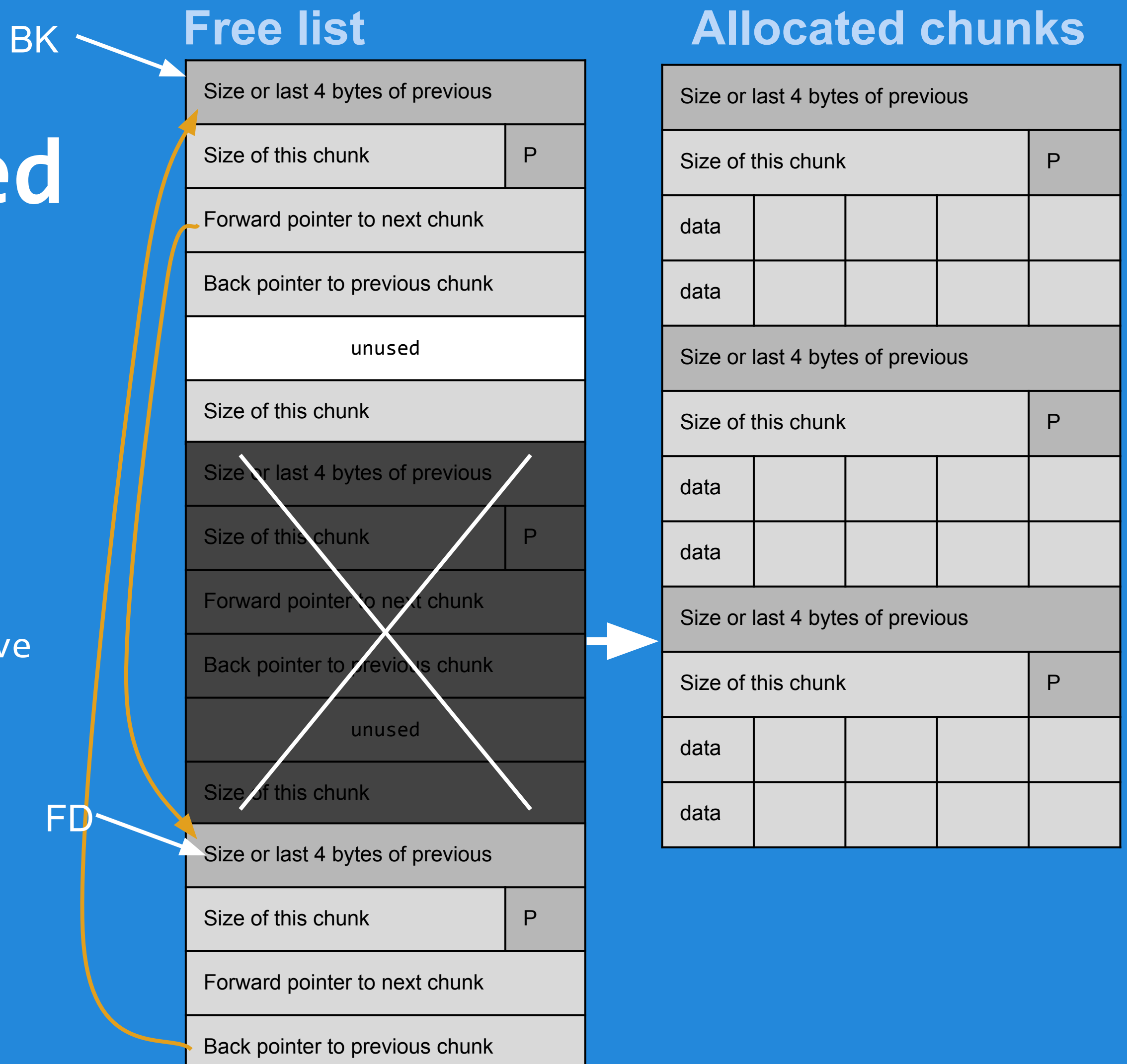
- involves changing pointers
  - double free messes this up!



Chunk is now Allocated

free() is the reverse of this process

- involves changing pointers
  - double free messes this up!
- Majority of buffer overflows since 2000 have been on the heap [1]
  - b/c devs don’t understand it well



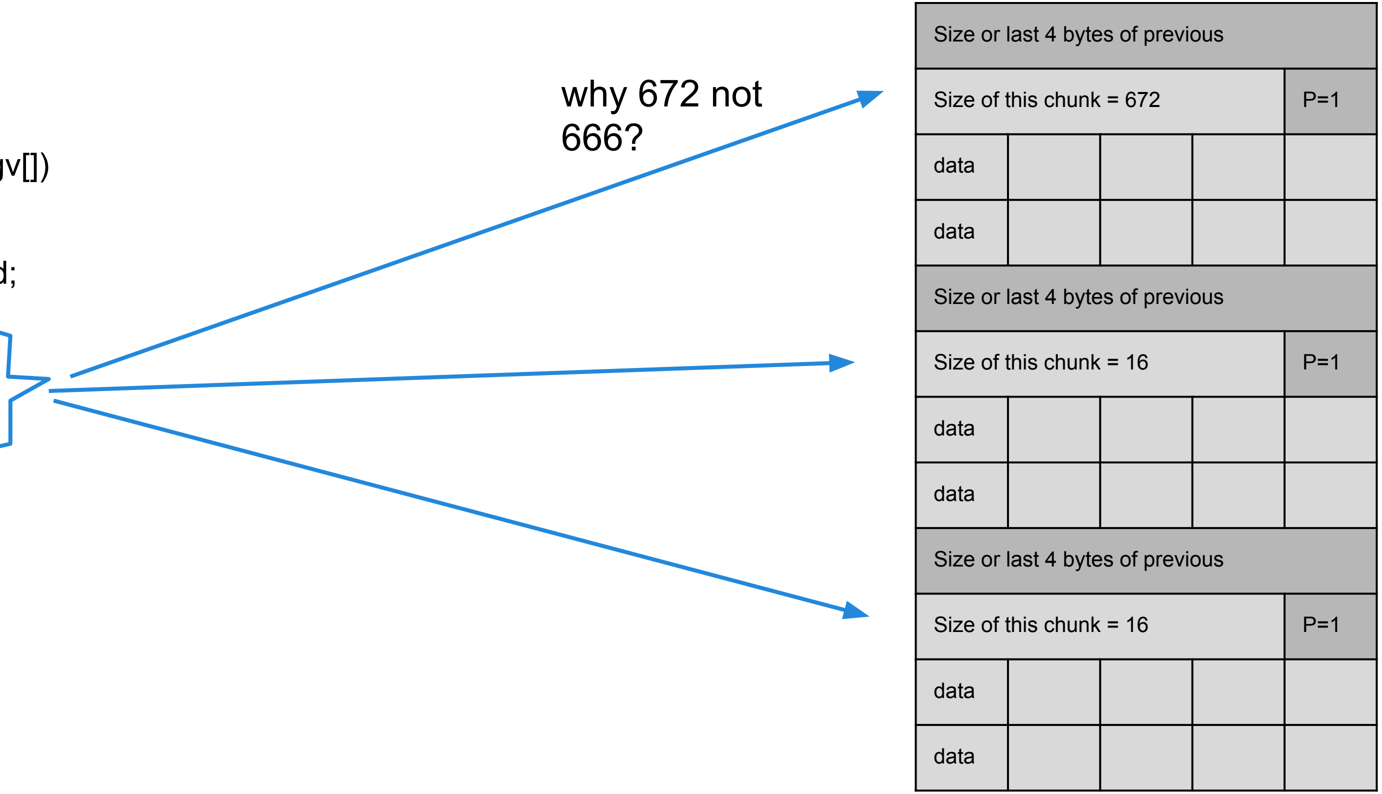
Exploring Heap Vulnerabilities

For these examples we'll use this guy as our friendly guide

- likes free()[dom]
- likes heaps [of british skulls]

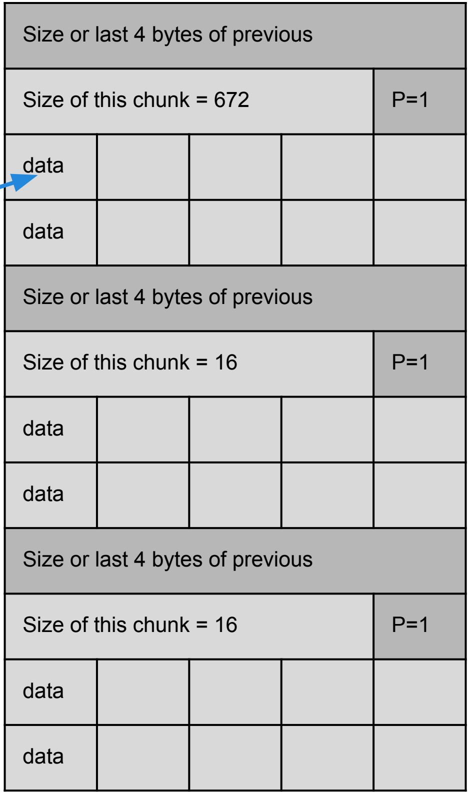
How free works (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```



How free works (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```



How free works (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

```
//pseudo code for free()
define free() {

  if (next not in use)
        consilidate with next;
        //(merges with existing chunk
        on free list)
 else
         link chunk to free list;

}
```

Size or last 4 bytes of previous				
Size of this chunk = 672				P=1
data				
data				
Size or last 4 bytes of previous				
Size of this chunk = 16				P=1
data				
data				
Size or last 4 bytes of previous				
Size of this chunk = 16				P=1
data				
data				

How free works (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

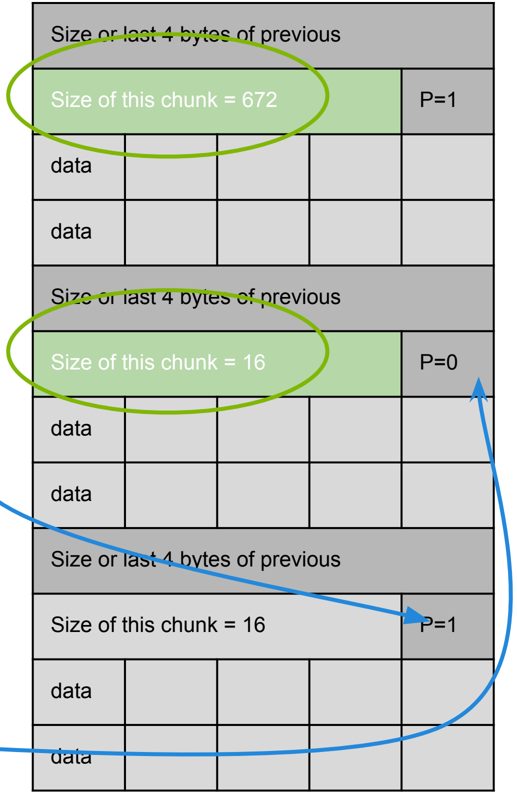
Checks to see if next chunk is also free

- checks P (PREV_IN_USE) flag on next, next chunk
  - it finds this via the size metadata in the current chunk and next chunk

In this case it is in use

So first is just freed up and linked to the free list

The P flag on the next bin (second) is then set to 0



How free works (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

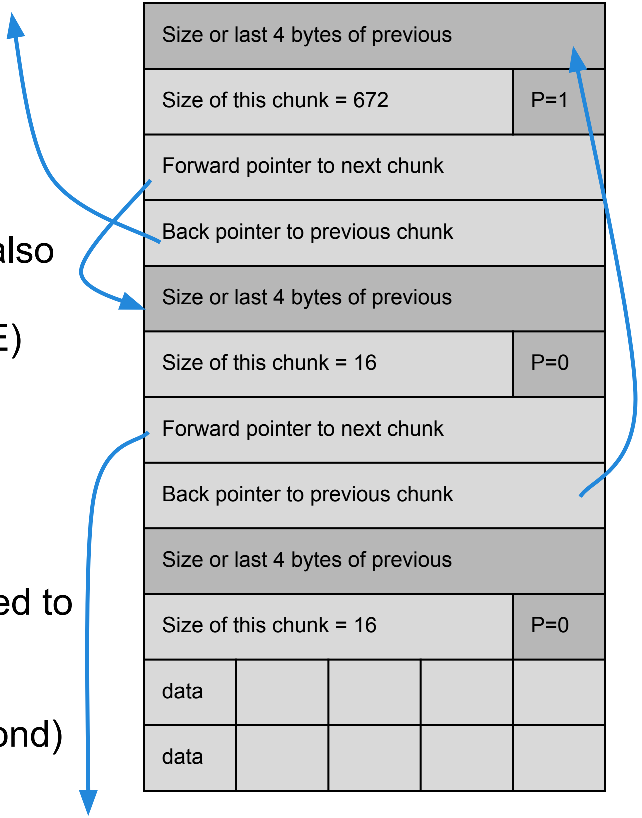
Checks to see if next chunk is also free

- checks P (PREV_IN_USE) flag on next, next chunk (not shown)

In this case it is in use

So first is just freed up and linked to the free list

The P flag on the next bin (second) is then set to 0



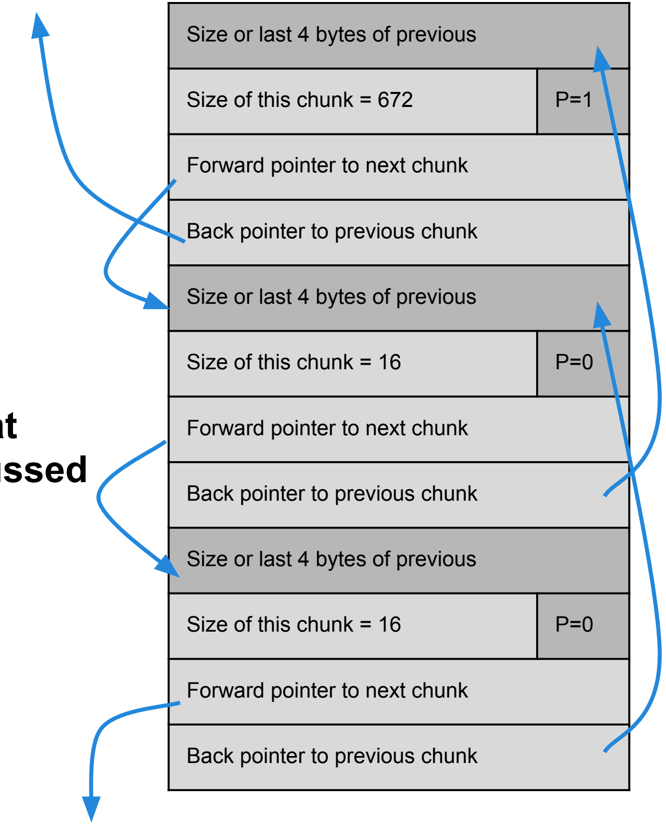
How free works (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

and so on

Note that consolidation may happen, and this is not shown

- **consolidation calls that unlink macro we discussed earlier**

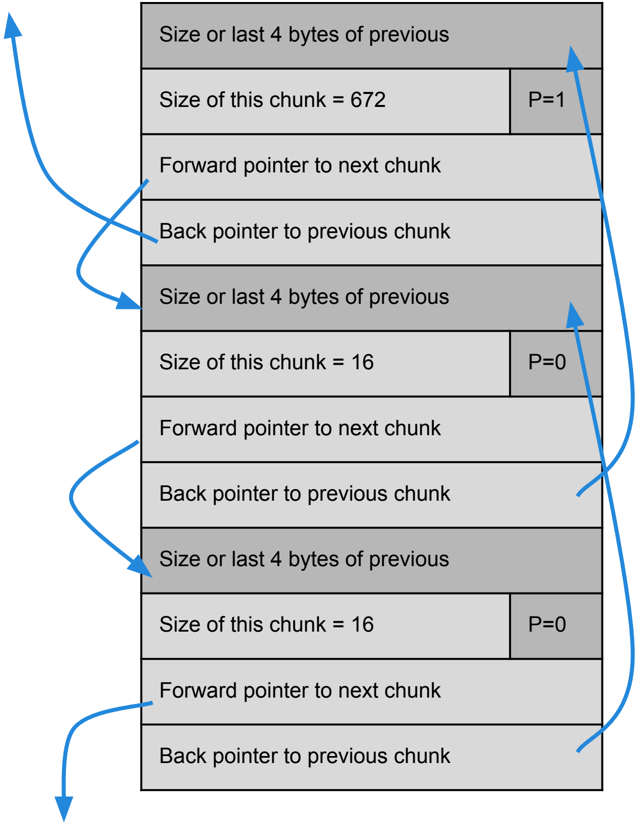


How free works (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

What to note:

- Pointers changed
  - in the chunk freed
  - and in OTHER chunks!
    - relies on meta data being correct
    - lets explore how this can be subverted maliciously
      - (arbitrary memory write vuln)

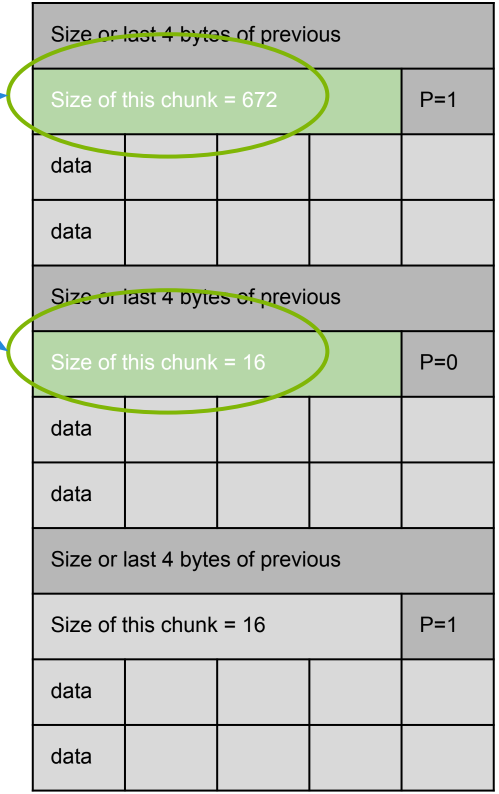


How free works (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

This metadata is the target

but we can only hit the 2nd one here



Heap Buffer Overflow (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Size or last 4 bytes of previous				
Size of this chunk = 672				P=1
data				
data				
Size or last 4 bytes of previous				
Size of this chunk = 16				P=1
data				
data				
Size or last 4 bytes of previous				
Size of this chunk = 16				P=1
data				
data				

Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

“FREEEEEEEEEEEDOOM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM
MMMMMMMMMMMMMM

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
“FREEEEEEEEEEEEEDOOMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM	

Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Will cause free
(second)
to segfault

“FREEEEEEEEEEEEEDOOM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
“FREEEEEEEEEEE EEDOOMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM MMMMMMMMMM	

Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

will alter the behavior of free()

“FREEEEEEEEEEEEEDOOM MMMMMMMMMMMMMMMM MMMMMMM…<dummy even integer(to have P=0)><new size (-4)><a fd pointer><a bk pointer>”

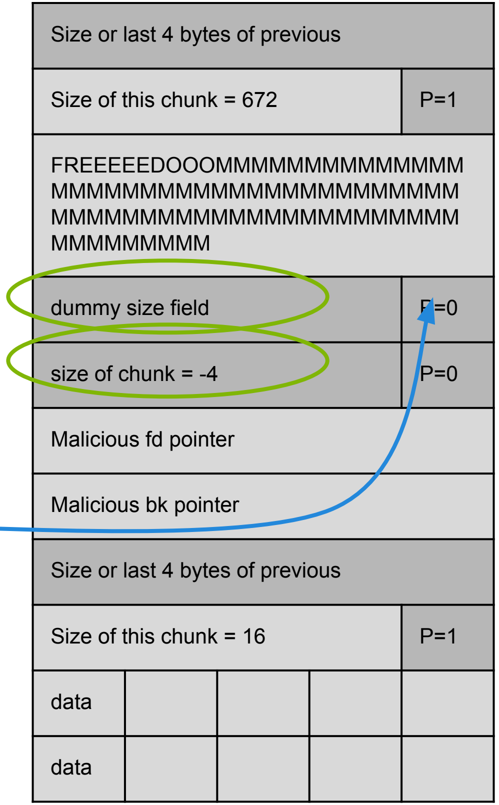
Size or last 4 bytes of previous				
Size of this chunk = 672				P=1
FREEEEEDOOOMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMM				
dummy size field				P=0
size of chunk = -4				P=0
Malicious fd pointer				
Malicious bk pointer				
Size or last 4 bytes of previous				
Size of this chunk = 16				P=1
data				
data				

Heap Buffer Overflow (from [1] p186)

```
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Size field in second chunk overwritten with a negative number

- when free() attempts to find the third chunk it will go here:



Heap Buffer Overflow (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

Size field in second chunk overwritten with a negative number

- when free() attempts to find the third chunk it will go here:
  - it sees the 2nd chunk is listed as free
    - unlink time

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
FREEEEEDOOOMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
Malicious fd pointer	
Malicious bk pointer	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1
data	
data	

Heap Buffer Overflow (from [1] p186)

```c
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[])
{
  char *first, *second, *third;
  first = malloc(666);
  second = malloc(12);
  third = malloc(12);
  strcpy(first, argv[1]);

  free(first);
  free(second);
  free(third);
}
```

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

need not point to the heap or to the free list!

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
FREEEEEDOOOMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
Malicious fd pointer	
Malicious bk pointer	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1
data	
data	

Heap Buffer Overflow (from [1] p186)

When this command runs:

- writes attacker supplied data to an attacker supplied address
    - to (fd + 12)
        - why?

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

The destination of the arbitrary write

The value which to write

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
FREEEEEDOOOMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
Malicious fd pointer	
Malicious bk pointer	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1
data	
data	

Double free() bug (kinda) does this

Take this guy free() him

free() him again and it produces some messed up zombie state of the former heap

Double free() Vulnerability

- Another exploitable bug
- Conditions to be vulnerable:
  - chunk to be free()'d must be isolated (no free adjacent chunks, they must be in use).
  - the free list bin in which the chunk is going must be empty (all those size-chunks must be in use)

Double free() Vulnerability

- much more complicated than the last bug
  - sadly we don’t have time for it
    - *See “Secure Coding in C and C++” by Robert Seacord for a great discussion*
- affects dlmalloc and old versions of RtlHeap
  - most modern allocator alternatives do safe unlinking
    - prevents most double frees

Use after free() Vulnerability

- involves using a pointer to a heap chunk that has been freed
    - when used as a function pointer == vulnerability
        - To exploit: need to overwrite that free'd portion of memory with malicious substitute

Integer Security

- Signed vs Unsigned
- Integer Truncation
- Overflow
- Underflow
- Nuances
- Conversion / Promotion
- Casting

Signed vs Unsigned (char == 1 byte)

signed char y;

SIGN	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	=1

SIGN	64	32	16	8	4	2	1	
0	1	1	1	1	1	1	1	=127

SIGN	64	32	16	8	4	2	1	
1	0	0	0	0	0	0	0	=-128

SIGN	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	1	=-1

unsigned char y;

128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	0	=0

128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	=1

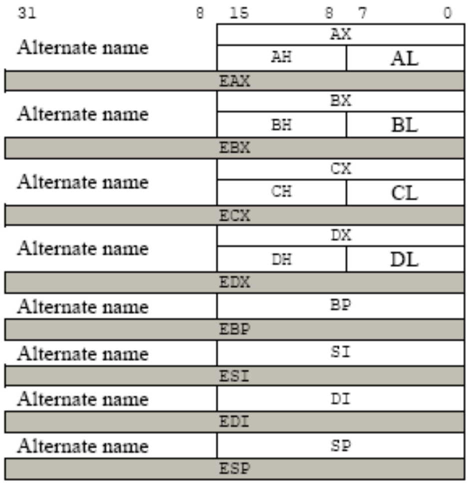
128	64	32	16	8	4	2	1	
0	0	1	0	1	0	0	0	=40

128	64	32	16	8	4	2	1	
1	1	1	1	1	1	1	1	=255

Truncation Example 1

```c
#include <stdio.h>
void foo()
{
  int i = -1;
  short x;
  x = i;
}
```

Lets see how this compiles and exactly what happens



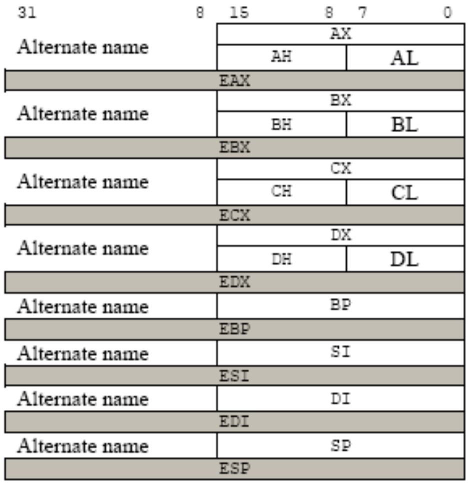
Truncation Example 1

Code editor

```c
#include <stdio.h>
void foo()
{
  int i = -1;
  short x;
  x = i;
}

```

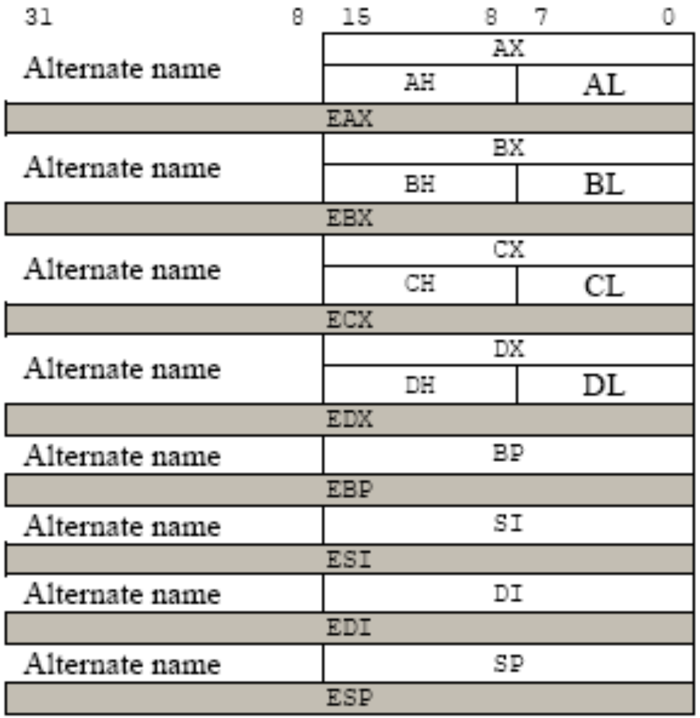
Assembly output

```asm
foo():
    push    rbp
    mov rbp, rsp
    mov DWORD PTR [rbp-4], -1
    mov eax, DWORD PTR [rbp-4]
    mov WORD PTR [rbp-6], ax
    pop rbp
    ret

```



x86_64 vs x86_32

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl
r8	r8d	r8w	r8b
r9	r9d	r9w	r9b
r10	r10d	r10w	r10b
r11	r11d	r11w	r11b
r12	r12d	r12w	r12b
r13	r13d	r13w	r13b
r14	r14d	r14w	r14b
r15	r15d	r15w	r15b



Integer Truncation continued

Do not code your applications with the native C/C++ data types that change size on a 64-bit operating system

- use type definitions or macros that explicitly call out the size and type of data contained in a variable

The 64-bit return value from sizeof in the following statement is truncated to 32-bits when assigned to bufferSize.

```
int bufferSize = (int) sizeof (something);
```

The solution is to cast the return value using size_t and assign it to bufferSize declared as size_t as shown below:

```
size_t bufferSize = (size_t) sizeof (something);
```

Safe type definitions/functions

- **ptrdiff_t**: A signed integer type that results from subtracting two pointers.
- **size_t:** An unsigned integer and the result of the sizeof operator. This is used when passing parameters to functions such as malloc (3), and returned from several functions such as fred (2).
- **int32_t**, **uint32_t etc**.: Define integer types of a predefined width.
- **intptr_t and uintptr_t:** Define integer types to which any valid pointer to void can be converted.

Platform Matters (intro)

- In many programming environments for C and C-derived languages on 64-bit machines, "int" variables are still 32 bits wide
  - <u>but long integers and pointers are 64 bits wide.</u>
  - This is described as the LP64 data model
- Alternative models:
  - ILP64 (all 3 types are 64 bits wide)
  - SILP64 (even shorts are 64 bits wide)
  - LLP64 (compatibility mode, everything is 32 bit)

Platform Matters

The difference among the three 64-bit models (LP64, LLP64, and ILP64) **lies in the non-pointer data types.**

ILP32 = Microsoft Windows & Most Unix and Unix-like systems @ 32bit

LP64 = Most Unix and Unix-likesystems, e.g. Solaris,Linux, BSD, and OS X;z/OS

LLP64 = Microsoft Windows (x86-64 and IA-64)
ILP64 = HAL Computer Systemsport of Solaris toSPARC64

Table 1. 32-bit and 64-bit data models

	ILP32	LP64	LLP64	ILP64
char	8	8	8	8
short	16	16	16	16
int	32	32	32	64
long	32	64	32	64
long long	64	64	64	64
pointer	32	64	64	64

A side note on Exploit Dev

The size of a struct may change from platform to platform!

```
struct test {
	int i1;
	double d;
	int i2;
	long l;
}
```

Why does this matter?

Can you find the bug? (60 secs)

```
__inline__ unsigned long long int rdtsc()
{
#ifdef __x86_64__
  unsigned int a, d;
  __asm__ volatile ("rdtsc" : "=a" (a), "=d" (d));
  return (unsigned long)a | ((unsigned long)d << 32);
#elif defined(__i386__)
  unsigned long long int x;
  __asm__ volatile ("rdtsc" : "=A" (x));
  return x;
#else
#define NO_CYCLE_COUNTER
  return 0;
#endif
}
```

Table 1. 32-bit and 64-bit data models

	ILP32	LP64	LLP64	ILP64
char	8	8	8	8
short	16	16	16	16
int	32	32	32	64
long	32	64	32	64
long long	64	64	64	64
pointer	32	64	64	64

Another Example

- On a 32-bit system, int and long are of the same size.
  - Due to this, some developers use them interchangeably. This can cause pointers to be assigned to int and vice-versa.
  - But on a 64-bit system, assigning a pointer to an int causes the truncation of the high-order 32-bits.

The solution is to store pointers as pointer types or the special types defined for this purpose, such as **intptr_t and uintptr_t.**

Integer “overflow”

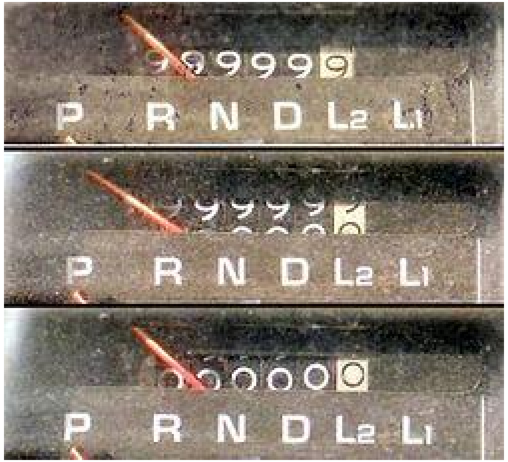


image source: wikipedia

- operation results in numeric value that is too large for storage space
- **C99** standard dictates that the result is always modulo, "computation involving <u>unsigned</u> operands can never overflow"
  - **<u>wraparound</u>**: does not overflow into other storage
    - UINT_MAX + 1 == 0

Integer “overflow”

- **C99** standard dictates that the result is always modulo, "computation involving unsigned operands can never overflow"
  - what about signed operands?
    - Overflowing a signed integer is an undefined behavior.
      - INT_MAX + 1 == ???

Integer “overflow”

- But for not **C99** settings:
  - **result saturation**:
    - occurs on GPUs and DSP
    - wrap around does not occur, instead a MAXVALUE is always returned
      - still does not overflow into adjacent memory

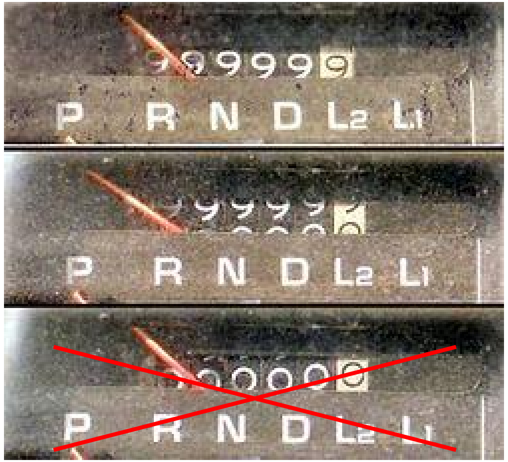


image source: wikipedia

Integer “underflow”

- occurs in subtraction
  - unsigned int x = 0 - 1
    - x == $2^{16} - 1$
- **C99** standard dictates that the result is always modulo, for unsigned operands
  - **<u>wraparound</u>**: does not overflow into other storage
- For signed operands this is undefined

Other Integer Nuances

- -INT_MIN == Undefined behavior
- Bit shifting integers:
  - Negative integers cannot be left shifted
    - -1 << x == Undefined behavior (for any x >= 0)
  - Error to shift a 1 into the sign position
    - INT_MAX << 1 == Undefined behavior
  - Error to shift by value > than bitwidth of the object
    - x86-32, int is 32 bits. Error to do (int) x << 33
      - x << 32 is ok. equivalent to x = x xor x

Other Nuances

- Starting a number with 0 designates octal
    - 1000, 2000, 0100, 0200, 0300, … 0981
    -

Integer Promotion / Conversion

- unsigned wins
  - https://www.securecoding.cert.org/confluence/display/seccode/INT02-C.+Understand+integer+conversion+rules
  - (1U > -1) == (1U > UINT_MAX) == 0
- Operands promoted (up to size) int
  - Integer types smaller than int are promoted when an operation is performed on them
    - (short) x << 17
      - promoted to integer, so this is safe

Other Integer Nuances

- (int)X - 1 + 1 == undefined IF X == INT_MIN
- INT_MIN % -1
- Does:
    - (short)x + 1 == (short)(x+1) for all values?

Integer Casting

(unsigned int) -1 becomes UINT_MAX

SIGN	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	=1
0	1	1	1	1	1	1	1	=127
1	0	0	0	0	0	0	0	=-128
1	1	1	1	1	1	1	1	=-1

128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	0	=0
0	0	0	0	0	0	0	1	=1
0	0	1	0	1	0	0	0	=40
1	1	1	1	1	1	1	1	=255

size_t vs ssize_t

size_t = an unsigned int

- rationale: Sizes should never be negative

But stupid things happen:

http://pubs.opengroup.org/onlinepubs/009604499/functions/mbstowcs.html

- People want to be able to return (size_t)-1==-1
  - thus ssize_t
    - [-1, SSIZE_MAX]
    - SSIZE_MAX = 32 767

My buddy Sean @ CMU CERT found this awesome case

Importance of Integer Bugs

- Crypto Libraries
  - http://blog.regehr.org/archives/1054
  - Probably in bitcoin / cryptocurrency libraries
- Often not understood by developers
- Can lead to vulnerabilities
- Suggested reading:
  - http://www.cs.utah.edu/~regehr/papers/overflow12.pdf

Floats

Float variable can be NaN (Not a number)

Float Nuances:

- 0.1 + 0.2 = 0.30000000000000004
- NaN == NaN is always false
- sums of many floats dont always add up right
  - precision is lost
- Suggested Readinghttp://floating-point-gui.

Bug time! (30 Seconds)

```
// Coefficients USED TO CONVERT FROM RGB TO monochrome.
const uint32 kRedCoefficient = 2125;
const uint32 kGreenCoefficient = 7154;
const uint32 kBlueCoefficient = 0721;
const uint32 kColorCoefficientDenominator = 10000;
```

This error was found in theChromium project by PVS-Studio C/C++ static code analyzer.

Integer bug resources

http://www.ibm.com/developerworks/library/l-port64/

promotion rules: https://www.securecoding.cert.org/confluence/display/seccode/INT02-C.+Understand+integer+conversion+rules

floats: http://floating-point-gui.de/

crypto libraries: http://blog.regehr.org/archives/1054

Formatted Output Security

- Section 0x352 (HAOE) covers this very well
- The problem of Format Strings
  - misuse
  - exploitation
    - Crashing
    - information leak/disclosure
- Mitigation Techniques
  - user input =/=> format string

Format Strings

- printf
- sprintf
- snprintf
- fprintf
- syslog
- ...

Looking at how functions are called

Code editor

```c
// Type your code here, or load an example.
#include <stdio.h>
void foo(){
    char buffer[256];
    sprintf(buffer, "Hello World! #%d", 12345);
}
```

Assembly output

```asm
.LC0:
    .string "Hello World! #%d"
foo():
    push    rbp
    mov rbp, rsp
    sub rsp, 256
    lea rax, [rbp-256]
    mov edx, 12345
    mov esi, OFFSET FLAT:.LC0
    mov rdi, rax
    mov eax, 0
    call    sprintf
    leave
    ret
```

- arguments passed via registers
  - we’ll cover this more next time
- then call sprintf

Looking at how functions are called

64 bit

- using registers

Code editor

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

void foo() {
 char buffer[256];
  sprintf(buffer, "Hello %d  %d  %d  %d", 1,2,3,4);

}
```

Assembly output

```asm
.LC0:
    .string "Hello %d  %d  %d  %d"
foo():
    push    rbp
    mov rbp, rsp
    sub rsp, 256
    lea rax, [rbp-256]
    mov r9d, 4
    mov r8d, 3
    mov ecx, 2
    mov edx, 1
    mov esi, OFFSET FLAT:.LC0
    mov rdi, rax
    mov eax, 0
    call    sprintf
    leave
    ret
```

Looking at how functions are called

32 bit (-m32 compiler flag)

- arguments on the stack

Code editor

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

void foo() {
 char buffer[256];
  sprintf(buffer, "Hello %d  %d  %d  %d", 1,2,3,4);

}
```

Assembly output

```asm
.LC0:
        .string "Hello %d  %d  %d  %d"
foo():
        push    ebp
        mov ebp, esp
        sub esp, 296
        mov DWORD PTR [esp+20], 4
        mov DWORD PTR [esp+16], 3
        mov DWORD PTR [esp+12], 2
        mov DWORD PTR [esp+8], 1
        mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
        lea eax, [ebp-264]
        mov DWORD PTR [esp], eax
        call    sprintf
        leave
        ret
```

Looking at how functions are called

64 bit

- eventually will use the stack

Code editor

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

void foo() {
 char buffer[256];
  sprintf(buffer, "Hello %d  %d  %d  %d  %d  %d  %d  %d", 1,2,3,4, 5, 6 , 7, 8);

}
```

Assembly output

```asm
.LC0:
        .string "Hello %d  %d  %d  %d  %d  %d  %d  %d"
foo():
        push    rbp
        mov rbp, rsp
        sub rsp, 288
        lea rax, [rbp-256]
        mov DWORD PTR [rsp+24], 8
        mov DWORD PTR [rsp+16], 7
        mov DWORD PTR [rsp+8], 6
        mov DWORD PTR [rsp], 5
        mov r9d, 4
        mov r8d, 3
        mov ecx, 2
        mov edx, 1
        mov esi, OFFSET FLAT:.LC0
        mov rdi, rax
        mov eax, 0
        call    sprintf
        leave
        ret
```

Looking at how functions are called

- Depends on architecture
- Depends on calling standard
  - more on this later
- Depends on type of function
  - normal code vs. system call
    - more on this later

Looking at how functions are called

32 bit

- format string function parses these to determine what to use on the stack

Code editor

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

void foo() {
  char buffer[256];
   sprintf(buffer, "Hello %d   %d   %d   %d", 1,2,3,4);

}
```

Assembly output

```asm
.LC0:
        .string "Hello %d   %d   %d   %d"
foo():
        push    ebp
        mov ebp, esp
        sub esp, 296
        mov DWORD PTR [esp+20], 4
        mov DWORD PTR [esp+16], 3
        mov DWORD PTR [esp+12], 2
        mov DWORD PTR [esp+8], 1
        mov DWORD PTR [esp+4], OFFSET FLAT:.LC0
        lea eax, [ebp-264]
        mov DWORD PTR [esp], eax
        call    sprintf
        leave
        ret
```

Looking at how functions are called

32 bit

- format string **SAFE** example

Code editor

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

void foo(char *buffer) {
  printf("%s", buffer);
}
```

Assembly output

```asm
.LC0:
        .string "%s"
foo(char*):
        push    ebp
        mov ebp, esp
        sub esp, 24
        mov eax, DWORD PTR [ebp+8]
        mov DWORD PTR [esp+4], eax
        mov DWORD PTR [esp], OFFSET FLAT:.LC0
        call    printf
        leave
        ret
```

Looking at how functions are called

32 bit

- format string vulnerability example

Code editor

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

void foo(char *buffer) {
  printf(buffer);
}
```

Assembly output

```
foo(char*):
        push    ebp
        mov ebp, esp
        sub esp, 24
        mov eax, DWORD PTR [ebp+8]
        mov DWORD PTR [esp], eax
        call    printf
        leave
        ret
```

Format Strings

%[flags][width][.precision][{length-modifier}] conversion-specifier

- %d or %i= signed decimal integer
- %u = unsigned decimal integer
- %o = unsigned octal
- %x = unsigned hexadecimal integer
- %X = unsigned hexadecimal integer (uppercase)
- %f = decimal float
- %e = scientific notation
- %a = hexadecimal floating point
- %c = char
- %s = string
- %p = pointer address
- %n = nothing printed, but corresponds to a pointer. The number of characters written so far is stored in the pointed location.

	specifiers						
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Exploiting Format Strings

Back to that example:

```
gets(buffer); // buffer == “%s%s…”
printf(buffer);
```

printf(“%s%s%s%s%s%s%s%s%s%s…”);

- reads pointer values off the stack for each %s
  - until all %s specifiers are satisfied
  - or until segfault

Exploiting Format Strings

printf("%08x %08x %08x %08x %08x....");

- prints out values on the stack in hex format
  - allows viewing of stack contents by attacker
  - printed in human-friendly format
    - x86-64 / x86 values are stored little-endian in memory
      - very important to remember

Exploiting Format Strings

printf(“%08x %08x %08x %08x %08x.…”);

- Iteratively increases the argument pointer by 8 each time.
    - for variable argument functions
    - va_start
    - va_list
        - an array. va_list[i] is argument pointer. (YMMV)

Exploiting Format Strings

printf("%04x....");

- can move forward argument pointer by other values.
  - typically just by 4 or 8 bytes on x86-32. Not sure on x86-64
  - This can be exploited to view arbitrary memory locations

Exploiting Format Strings

printf("\xde\xf5\xe5\x04%x%x%x%x%s");

- viewing arbitrary memory locations (32bit)
  - move argument pointer forward enough to point within the string (the %x chain)
- %s uses a stack value as a pointer
  - prints out what it points to
    - here, will print the value at 04e5f5de (little endian)

Exploiting Format Strings

printf(“\xde\xf5\xe5\x04%x%x%x%x%s”);

- \x ← these are escape characters
    - denotes special character
        - ASCII encoding
    - used here to provide a little-endian address (32 bit example)
        - (more on this in the exploitation section)

Exploiting Format Strings

Writing to memory address (from [1] p326)

```
int i;
printf(“hello%n\n”, (int *)&i);
```

writes 5 to variable i;

Exploiting Format Strings

Writing to arbitrary memory address

printf(“\xde\xf5\xe5\x04%x%x%x%x%n”);

printf(“\xde\xf5\xe5\x04%x%x%x%150x%n”);

works well for writing small values

- but not memory addresses

Exploiting Format Strings

Writing to arbitrary memory address

printf("\xde\xf5\xe5\x04%x%x%x%x%n");

will write the number of characters before the %n printed so far to 04e5f5de.

- We need to explore length modifier:

%[flags][width][.precision][{length-modifier}] conversion-specifier

	specifiers						
length	d i	u o x X	f F e E g G a A	c	s	p	n
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

Other modifiers

%[flags][width][.precision][{length-modifier}] conversion-specifier

- width
- precision

```
int i;
printf("%50u%n", 1, &i);  // i = 50
printf("%5000u%n", 1, &i);// i = 5000
```

Exploiting Format Strings

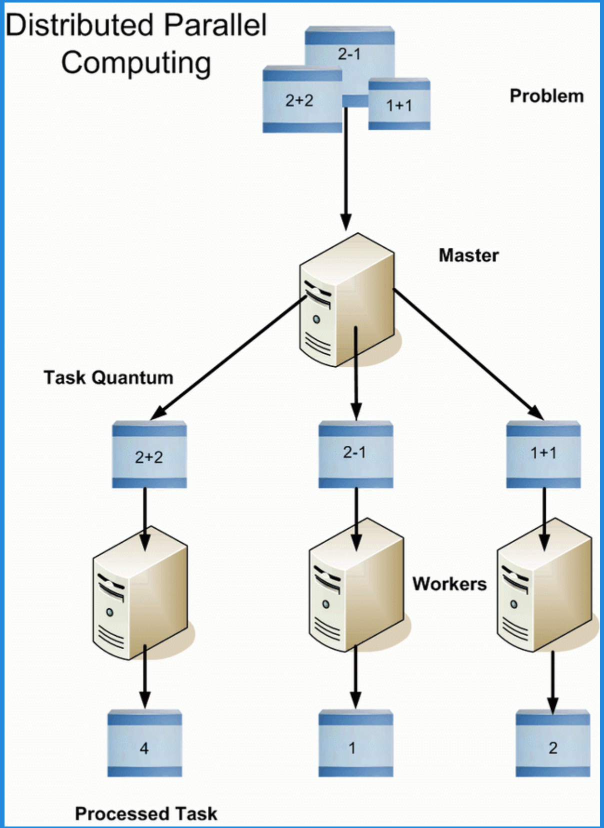
Combine these techniques to write arbitrary values to arbitrary memory location (s) byte by byte:

- pg 174 HAOE explains this best. The following writes 0xDDCCBBAA to the address at 0x08409755

We'll finish this topic later in the semester

Memory	94	95	96	97			
First write to 0x08409755	AA	00	00	00			
Second write to 0x08409756		BB	00	00	00		
Third write to 0x08409757			CC	00	00	00	
Fourth write to 0x08409758				DD	00	00	00
RESULT	AA	BB	CC	DD			

Concurrency & Race Conditions



Concurrency, Parallelism, & Multithreading

Concurrency:

- Several computations executing simultaneously and potentially interacting with each other
- Concurrency not always equal multithreading
  - *possible for multithreaded applications to not be concurrent*

Multithreading:

- program has two or more threads that **may** execute concurrently

Parallelism:

- Data parallelism vs. task parallelism
  - **data**: split data set into segments apply function in parallel
  - **task**: split job into several distinct tasks to be run in parallel

3 Properties for Race Conditions [1]

1. Concurrency Property
    - At least 2 control flows must be executing concurrently
2. Shared Object Property
    - a shared race object must be accessed by both of the concurrent flows
3. Change State Property
    - At least one of the control flows must alter the state of the race object

Race Condition explained

concurrency property:

- train
- tracks

shared object

- the tracks (junction)

change state:

- the junction

Race Condition Bughunting Strategy

1. Focus on the shared objects.
2. For each shared object, follow how it is handled through the code. Focus on any state changes.
3. For each state change, enumerate what other concurrent entities might be operating on it.
    a. Hunt for what could go wrong line by line between the two threads
        i. R & Ws

Race Condition potential results

- Corrupted Values
- ‘Volatile’ objects act in undefined ways when handled asynchronously
- Elevated permissions
  - (permission escalation)
  - CVE-2007-4303, CVE-2007-4302, ...
- Deadlocks
  - DoS

Questions?

Reading: 0x280 up to 0x300 (HAOE)
and 0x350 up to 0x400

Great Study Resource: http://q.viva64.com/