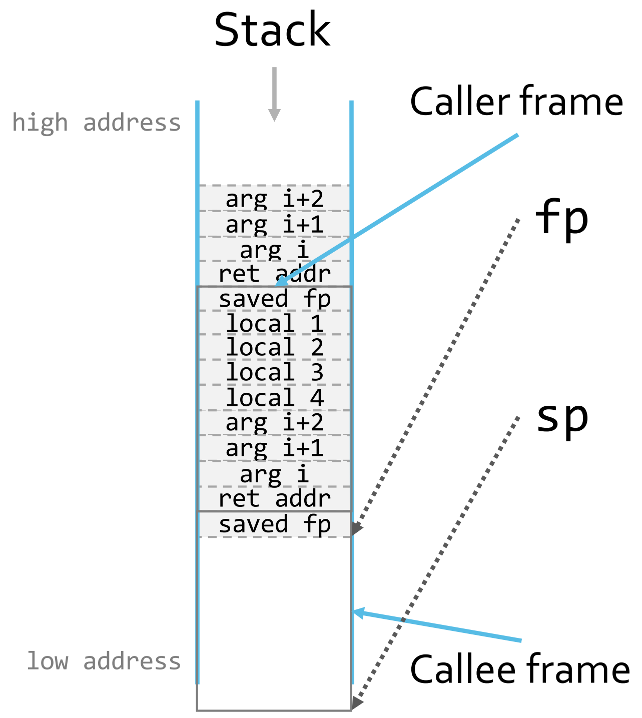
CSE 127 Computer Security

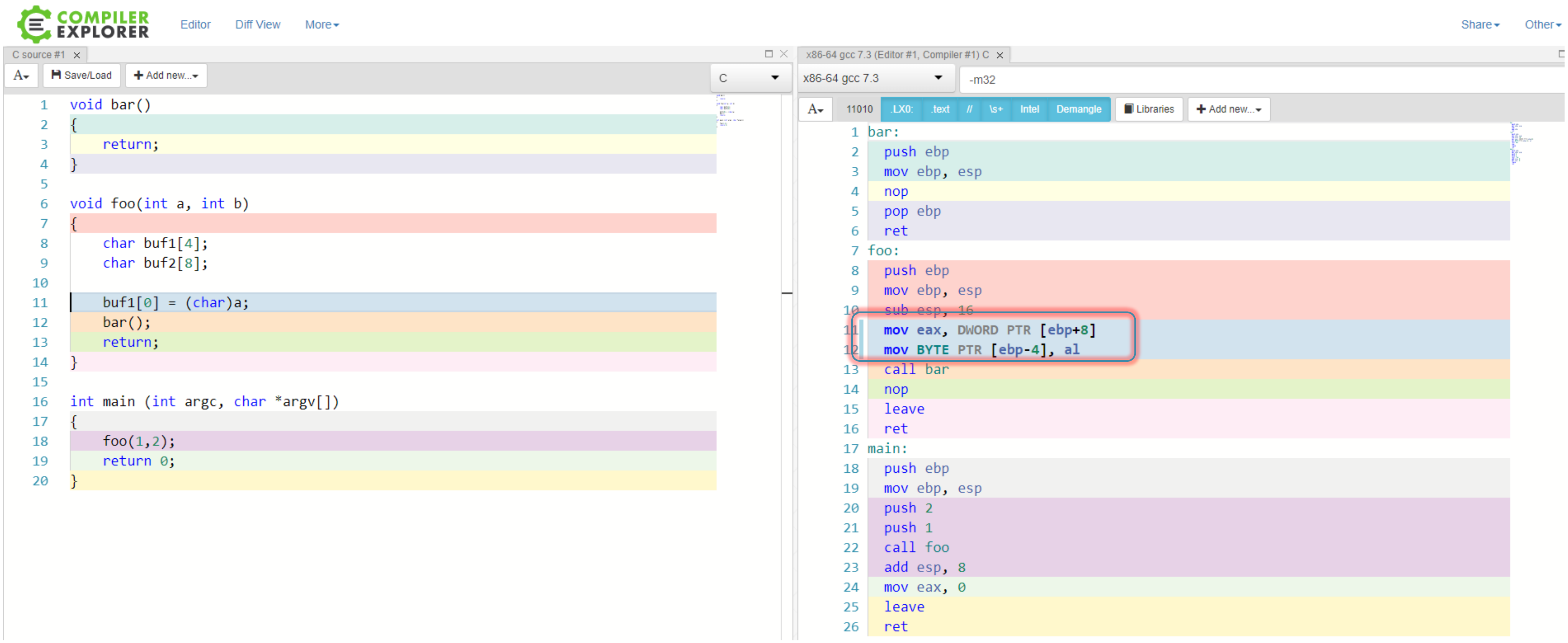
Stefan Savage, Fall 2020, Lecture 4

Control Flow Vulnerabilities II: Format Strings, Integers and the Heap

Review

- Function arguments and local variables are stored on the stack
  - Next to control flow data like the return address and saved frame pointer
- Function arguments and local variables are accessed by providing offsets relative to the frame pointer





Language: C

Compiler: x86-64 gcc 7.3

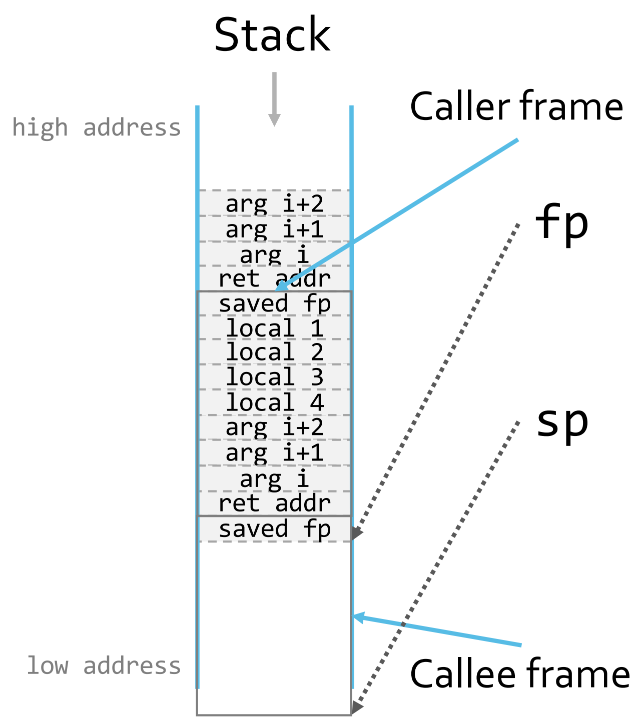
Options: -m32

```c
void bar()
{
    return;
}

void foo(int a, int b)
{
    char buf1[4];
    char buf2[8];

    buf1[0] = (char)a;
    bar();
    return;
}

int main (int argc, char *argv[])
{
    foo(1,2);
    return 0;
}
```

```
bar:
  push ebp
  mov ebp, esp
  nop
  pop ebp
  ret
foo:
  push ebp
  mov ebp, esp
  sub esp, 16
  mov eax, DWORD PTR [ebp+8]
  mov BYTE PTR [ebp-4], al
  call bar
  nop
  leave
  ret
main:
  push ebp
  mov ebp, esp
  push 2
  push 1
  call foo
  add esp, 8
  mov eax, 0
  leave
  ret
```

Related issue…

- Implicit agreement between the caller and the callee about the number, size, and ordering of function arguments.
  - #include function declaration
- What happens if function declaration differs from actual implementation?
- Hmmm…. Hold that thought



Goals for today

- Looking at other kinds of control flow vulnerabilities beyond pure stack smashing
  - **Format strings**
    - How printf and its ilk are really interpreters of weird little programming languages
  - **Integers and integer overflow**
    - What are integers? On computers. Why are they so dangerous?
  - **Heap vulnerabilities**
    - Overflowing heap-allocated objects and vulnerabilities in the use of free()

- Note: this is not an exhaustive list, but demonstrate some of the range of problems in this space
- We may not finish this all today…

Format String Vulnerabilities

printf()

- `printf(“Diagnostic number: %d, message: %s\n”, j, buf);`
  - *“If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.”*
  - Also, `sprintf ( char * str, const char * format, ... );` and `fprintf ( FILE * stream, const char * format, ... );`

- Format specifier: `%[flags][width][.precision][length]specifier`
- Specifier
  - Type and interpretation of the corresponding argument
  - Examples:
    - `%s`: string
    - `%d`: signed decimal integer
    - `%x`: unsigned hexadecimal integer
- Flags
  - Sign, padding, justification, …

`printf()`

- Format specifier:
  `%[flags][width][.precision][length]specifier`
- Width
  - Minimum width of printed argument in characters
- Length
  - Size of argument
  - Examples:
    - `h: char`
    - `hh: short`
    - `l: long`
    - `ll: long long`

Variadic Functions

- So, how many arguments does `printf` take?
  - `int printf ( const char * format, ... );`
- C supports functions with a **variable number** of arguments.
- If the number of arguments is not pre-determined, **how does the called function know how many were passed in**?
  - Options
    - Another argument explicitly specifies count?
    - The last argument is a reserved terminator value?
    - Another argument implicitly encodes count?
      - This is what printf does.

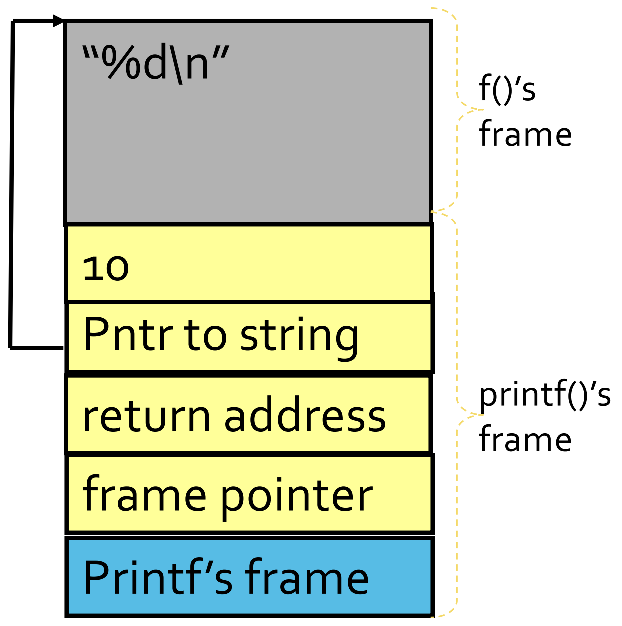
How is this implemented?

- Caller
  - Pushes arguments onto stack
  - Pushes pointer to format string onto stack

- Callee
  - Reads the format string off stack
  - *Uses* the format string to read corresponding arguments off of stack
    - Reads one value off stack for each “%” parameter
    - To be clear: printf runtime is looking up the stack, **controlled by % parameters**

Printf on the stack

```
f() {
   char fmtstrng = "%d\n";
   …

   printf(fmtstrng,10);
}
```



Key problem

- User is responsible for enforcing one-to-one mapping between format specifiers and arguments
- What if there are too many arguments?
- What if there are too few arguments?

Format String Vulnerabilities

- You can think of printf (and similar functions) as implementing a little language of commands specified in the format string.
- Consequently, these two lines are very different:
  - `printf(“%s”, buf);`
  - `printf(buf);`
- Not a good idea to let an attacker feed arbitrary commands to your command interpreter.

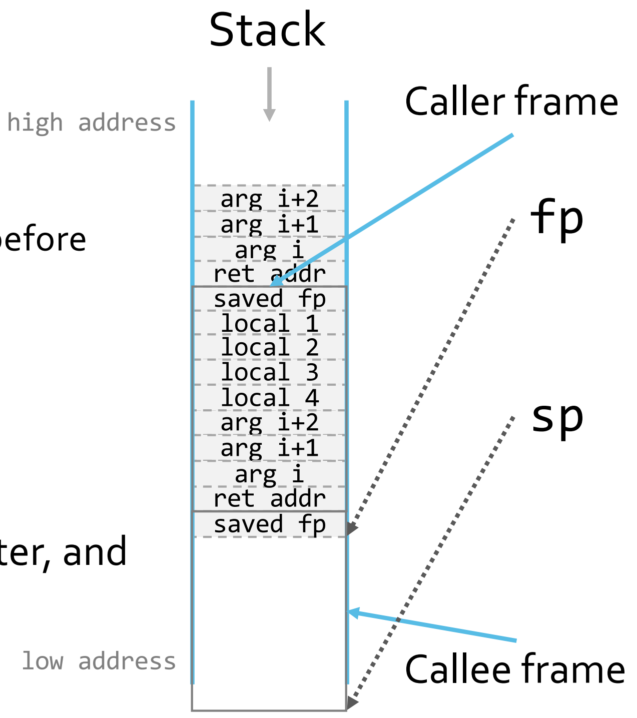
Format String Vulnerabilities

- Still, how bad could it be?
- What can an attacker do with a well-crafted format string?
  - Read arbitrary memory
  - Write arbitrary memory

Format String Vulnerabilities: Reading

`%[flags][width][.precision][length]specifier`

- Reading from the stack
  - `printf("%08x.%08x.%08x.%08x\n");`
    - What will this do?
    - Read and print the previous four words up the stack (before the format string argument pointer)
- Reading via a pointer
  - `printf("%s\n");`
  - What will this do?
  - Take the previous stack word, interpret it as a pointer, and print the memory at that address as a string



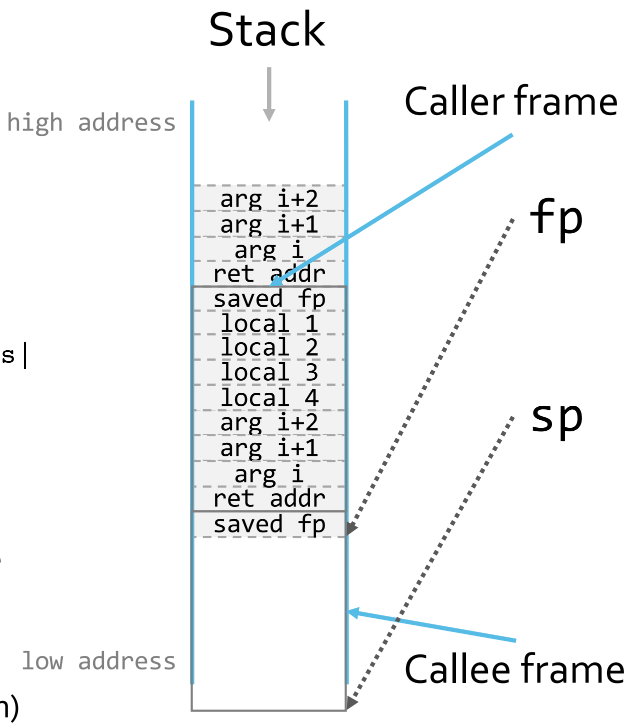
Format String Vulnerabilities: Reading

`%[flags][width][.precision][length]specifier`

- What if we want to read **arbitrary** memory?
  - **Recall: the format string *itself* is frequently stored on the stack**
- Consider:

```
f() {
  char localstring[80] = “x10\x01\x48\x08_%08x.%08x.%08x.%08x.|%s|
“);
  printf (localstring);
}
```

- Uses ”%08x” specifiers to move printf’s internal argument pointer up the stack…. back into the format string itself in the callers stack frame!
  - Mind blown…
  - Note: this assumes no intervening locals and four words of control data on stack before string pointer (can vary depending on function)
- Will print whatever string is stored at address at 0x08480110
  - Note: “x10\x01\x48\x08” = 0x08480110



Format String Vulnerabilities: Writing memory

- Ok, but that's just reading… can you use format string vulnerabilities to write memory (i.e., change program state)?
- Simple buffer overflow
  - `if (strlen(src) < sizeof(dst)) sprintf(dst, src);`
  - What if `src` contains format specifiers?

Format String Vulnerabilities: Writing Memory

- Arbitrary write
  - %n
  - *"Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location."*

    -
      ```
      int x = 0;
      printf("Hello %n ", &x);  // after call x == 6
      ```
- Combine with the reading trick to write to arbitrary addresses…
  - Can build up a full word write one byte at a time
- While it has uses, "%n" is considered extremely dangerous and is frequently removed from libraries

Additional Resources

- *Exploiting Format String Vulnerabilities* by scut / team teso
  - https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

Format string summary

- Functions that take format strings act like little command interpreters
  - Anything that can step outside the semantics of your runtime system like this is potentially dangerous
- Don't let attackers decide which commands to pass to your command interpreters.

Heap vulnerabilities: Why should the stack have all the fun?

Memory management in C

- The C programming language uses **explicit memory management**
  - Data is allocated and deallocated **dynamically** from the heap **by the program**
    - **i.e., malloc and free**
  - Dynamic memory is accessed via **pointers**
  - The programmer is responsible for the details
  - The system does not track memory liveness (no garbage collection)
  - The system does not ensure that pointers are live or valid (no smart pointers)
- C++ has the same issues

Heap Vulnerabilities

- What if the attacker is able to **corrupt** data on the heap? (e.g., via a buffer overflow of a heap-allocated buffer)
  - Program data stored on the heap
  - Heap metadata (i.e., for organizing the heap itself)
- What if the attacker can cause the program to use `malloc()` and `free()` in unexpected combinations?
  - Remember, the input controls which path your program takes, and the attacker controls the input

Heap Corruption

- **Overwriting program data**
  - As is the case with stack buffer overflows, effect depends on variable semantics and usage.
  - Generally anything that influences future execution path is a promising target.
  - Typical problem cases:
    - Variables that store result of a security check
      - Eg. isAutheticated, isValid, isAdmin, etc.
    - Variables used in security checks
      - Eg. buffer_size, etc.
    - Data pointers
      - Potential for further memory corruption
    - **Function pointers** (as good as a *return address*)
      - Direct transfer of control when function is called through overwritten pointer
      - **vTables**

vTables

- How do virtual function calls work in Object-Oriented languages? (e.g., C++)
- What's the abstraction?
- How is it implemented in reality?
  - When `obj->foo()` is called from within `bar()`, how is control transferred to the correct implementation of `foo()`?

```
class Base
{ public: virtual void foo()
    {cout << "Hi\n";} };

class Derived: public Base
{ public: void foo()
    {cout << "Bye\n";} };

void bar(Base* obj)
{ obj->foo(); }

int main(int argc, char* argv[])
{
         Base *b = new Base();
         Derived *d = new Derived();

         bar(b);
         bar(d);
 }
```

vTables

- Each object contains a pointer to its *virtual function table* (aka ***vtable***)
- Vtable is an array of function pointers
  - One entry per virtual function of the object's class
- Based on the class and function, the compiler knows which offset in which vtable to use
  - Interfaces and multiple inheritance complicates this quite a bit, but the issue of relevance to us remains the same: in an OO language the heap will be full of function pointers
  - bar() compiles to something like: *(obj->vtable[0])(obj)

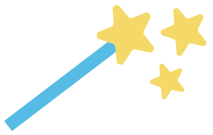
```
class Base
{ public: virtual void foo()
    {cout << "Hi\n";} };

class Derived: public Base
{ public: void foo()
    {cout << "Bye\n";} };

void bar(Base* obj)
{ obj->foo(); }

int main(int argc, char* argv[])
{
        Base *b = new Base();
        Derived *d = new Derived();

        bar(b);
        bar(d);
}
```

Heap Corruption

- **Overwriting heap metadata**
  - The heap has its own data structures that it manipulates to keep track of free and allocated memory
  - If you overwrite one of those data structures, the heap's memory management code will still use them as though they were valid

Heap Basics

- Abstraction vs Reality
  - `malloc()` and `free()`
- Abstraction
  - Dynamically allocate and release memory buffers as needed. Magic!
    - `ptr=malloc(20)`: Give me 20 bytes.
    - `free(ptr)`: I don't need my 20 bytes any more. Send them back.



- Reality
  - Where does this memory come from?
  - How does "the system" know how much memory to reclaim when `free()` is called?
  - Details vary by heap implementation, but follow a common pattern

Heap Basics

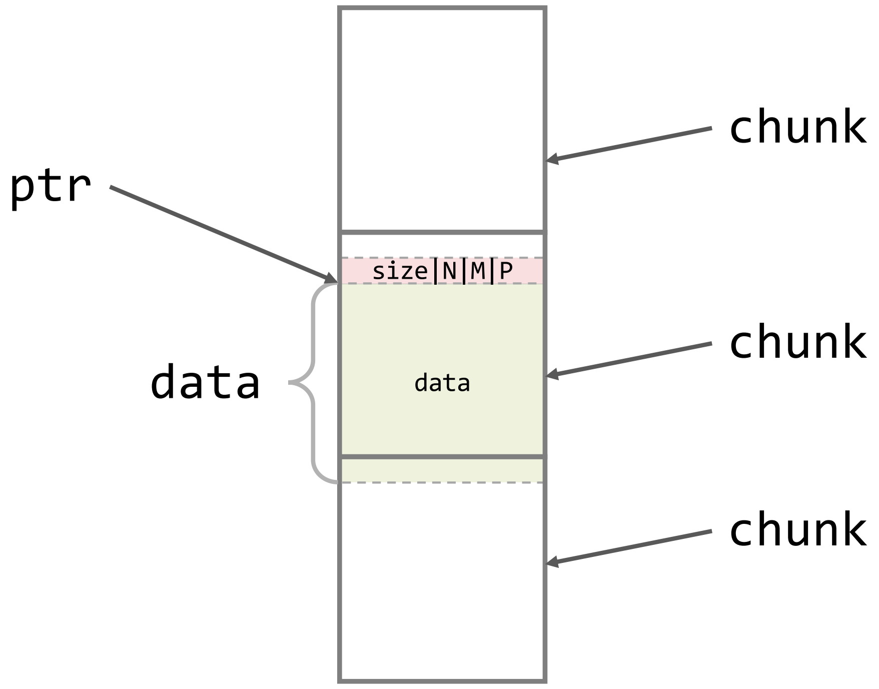
- The heap is managed by the aptly-named heap manager
- There are many different heap managers, even within a single system
  - Some are optimized for allocations of a certain size, for speed, for space efficiency, etc.
  - We will focus on Doug Lea's heap, aka glibc dlmalloc

Heap Management (glibc dlmalloc)

- The heap manager maintains contiguous “chunks” of available memory
- The heap layout evolves when `malloc()` and `free()` functions are called
  - Chunks may get allocated, freed, split, or coalesced.
- These chunks are stored in doubly linked lists (called bins)
  - Grouped (roughly) by chunk size
- When a new chunk becomes free, it is inserted into one of these lists.
- When a chunk gets allocated, it is removed from the list.

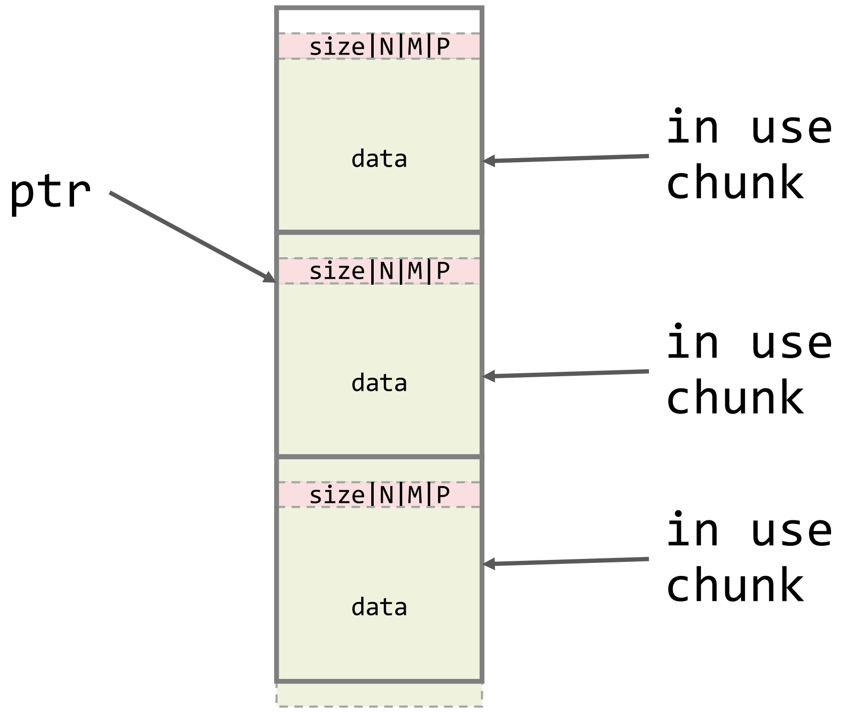
Heap Management (glibc dlmalloc)

- Chunks
  - Basic units of memory on the heap
  - Either Free or In Use
  - User data + heap metadata
    - Size: chunk size
    - Flags:
      - N: Non main arena (not relevant for us)
      - M: is Mmaped (not relevant for us)
      - P: Previous chunk is in use
    - Note that the last word of the data block is the first word of the next chunk



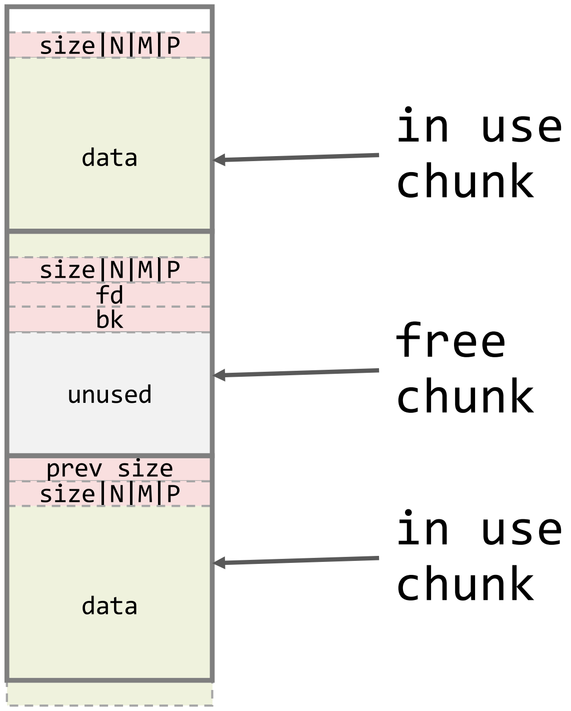
In Use Chunk

- Malloc returns a pointer to the start of the data block
- Free can release the chunk by looking at the metadata in the word before the data block



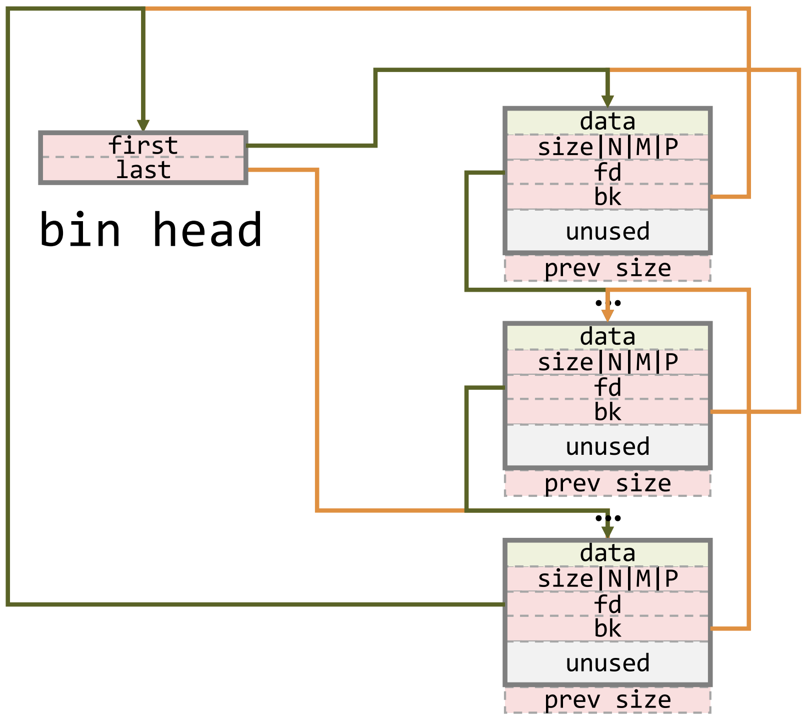
Free Chunk

- Free chunks are kept in doubly-linked lists (bins)
  - Some of the unused data area of each free chunk is used to store forward and back pointers
- Consecutive free chunks are coalesced
  - No two free chunks can be adjacent to each other
- Additionally, the last word of unused data (first word of next chunk) contains a copy of the size of the free chunk.



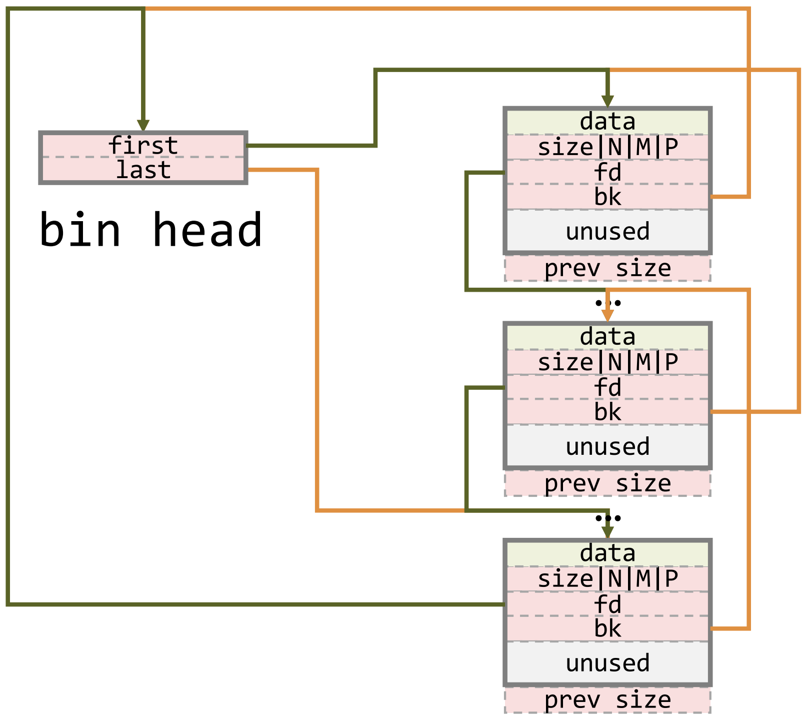
Free List

- Free chunks are kept in circular doubly-linked lists (bins)



Free List

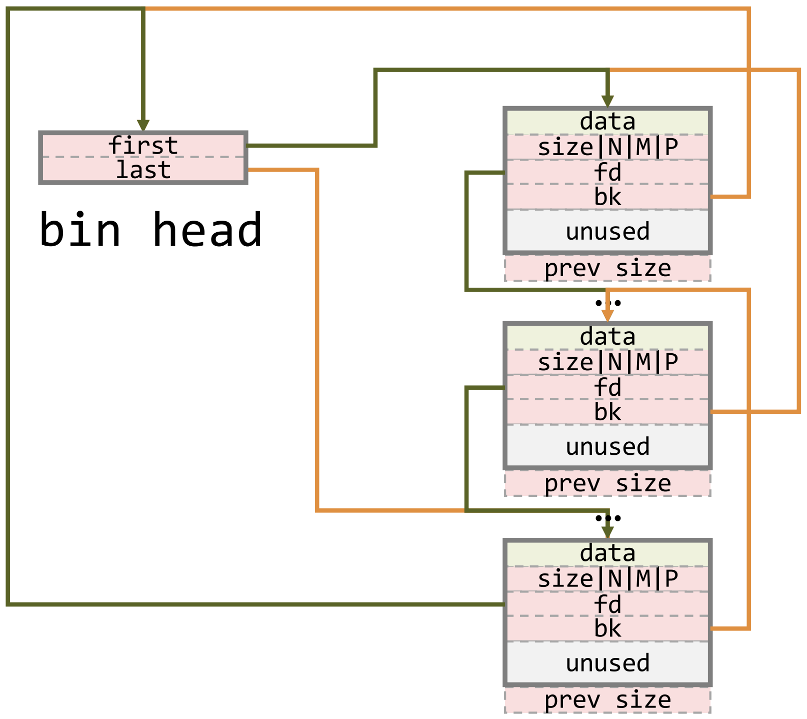
- Chunks are inserted into the list when they are freed.
- Chunks are removed from the list if they get allocated, or if they need to be combined with a newly-freed adjacent chunk



Free List

- Unlink operation to remove a chunk from the free list:

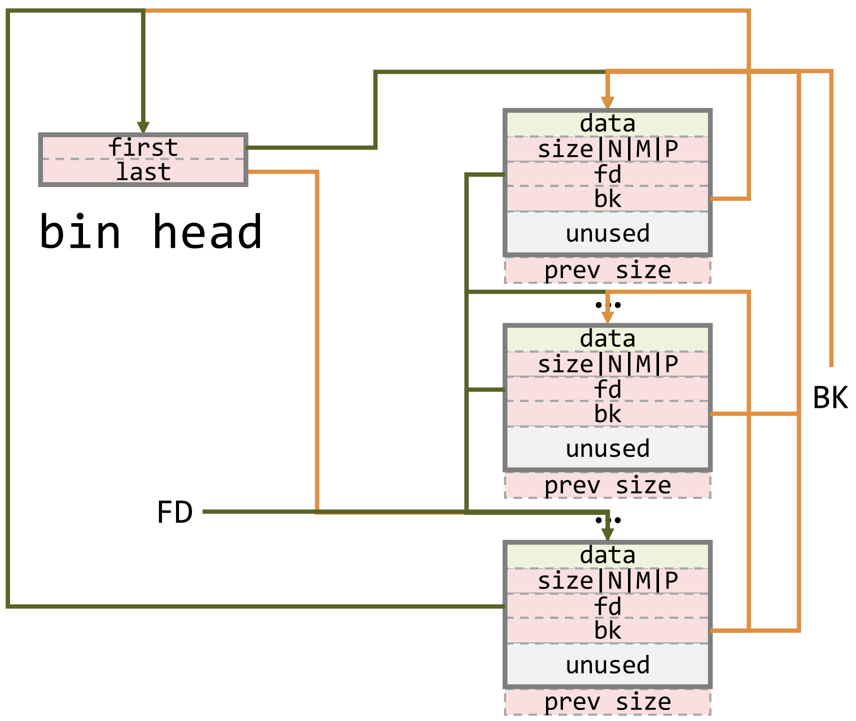
```
#define unlink(P, BK, FD)
{
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```



Free List

- Unlink operation to remove a chunk from the free list:

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



Heap Corruption

- What can we do if we manage to get `free()` to act on data we control?
  - What if we can cause the heap manager to act on fake chunks?

Heap Corruption

- Look at the unlink macro again
- What can an attacker do if she manages to control what's in the `fd` and `bk` fields of a free chunk?
- Write an attacker chosen value to an attacker-chosen address
  - ***Write-what-where*** primitive
  - Generally enough to compromise system (overwrite well-known control data somewhere -- ret addr, vtable, function pointers, etc)

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

Heap vulnerabilities via memory management errors

- An attacker may also attempt to get the victim to invoke `malloc()` and `free()` in unintended sequences or with invalid arguments.
  - In principal, every block of memory that is allocated by malloc() has to be released by a corresponding call to free()
    - As an attacker, what is your reaction when you see "has to be"?
  - Programming the heap *weird machine*

- What sequences of events can trigger unintended functionality
  - Use after free
    - `free(p); p->foo();`
    - `free(p); q = malloc(n); memcpy(p, buf, k);`
  - Double free
    - `free(p); free(p); q = malloc(n); r = malloc(n);`
    - `free(p); q = malloc(n); free(p);`

Use-After-Free and Double Free

- Use-after-free
    - free(p); p→foo();
    - free(p); q = malloc(n); memcpy(p, buf, k);
- Double free
    - free(p); free(p); q = malloc(n); r = malloc(n);
    - free(p); q = malloc(n); free(p);

Use After Free Vulnerabilities

- Take a look at the this code
  - Ignore the missing checks for result of `malloc()`
    - This is how NULL pointer errors happen
- What will it print?
- What if p referenced a structure with function pointers?
  - Or a C++ object?

```
char *p, *q;

p = malloc(20);
snprintf(p, 20, "Hi");
printf("%s\n", p);
free(p);

q = malloc(20);
snprintf(q, 20, "Bye");

printf("%s\n", p);
free(q);
```

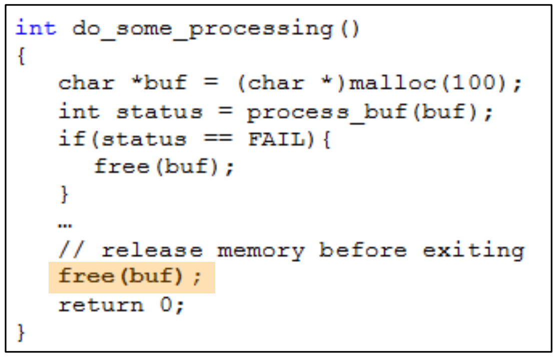
Use After Free Vulnerabilities

- Note that in a multi-threaded application, the second malloc() may not be obvious.
- Just because the use of previously freed memory appear right after the free, does not mean it is safe.

```
char *p;

p = malloc(20);
snprintf(p, 20, "Hi");
printf("%s\n", p);
free(p);
printf("%s\n", p);
```

Double Free Vulnerabilities

- Here is a simple example containing a double-free vulnerability:
- The highlighted line in the example above will be freeing buf for a second time if status == FAIL.
- In real-world code typically these issues are spread across many lines of code, functions, or even files because typically buffers are allocated in one function, then used in other functions, and released in yet another function. Therefore, very careful code review is needed to catch double-free issues.



```
int do_some_processing()
{
   char *buf = (char *)malloc(100);
   int status = process_buf(buf);
   if(status == FAIL){
     free(buf);
   }
   …
   // release memory before exiting
   free(buf);
   return 0;
}
```

Double Free Vulnerabilities

- What happens when you free something the first time?
  - Check for opportunity to coalesce with adjacent chunks
    - If so, unlink adjacent chunk(s) from free links
  - Add chunk to free list

- What happens when you free something the second time?
  - Same thing… but…
    - Hmmm… you can’t shouldn’t on the contents of the meta data being correct (could have been reallocated and used since it was freed the first time)
    - But free() will happily add same chunk onto free list multiple times…
      - What happens in subsequent allocations?

Double free vulnerabilities

```
a = malloc(10);
b = malloc(10);
c = malloc(10);
free(a);
free(b);
free(a);
d = malloc(10); // uses memory returned from free(a)
e = malloc(10); // uses memory returned from free (b)
f = malloc(10); // uses memory returned from free (a)… now d points to f
```

What to do?

- Safe heap implementations
  - Safe unlinking – validate that p->next-prev == p->prev-next == p
  - Cookies on heap allocations (checked on free)
  - Heap integrity check on malloc on free
  - Encrypt key heap function pointers (decrypt on use and then re-encrypt)
  - Randomize cheap chunk meta-data (encoded with random key like ptrs)
  - Etc
- Use Garbage Collection instead
- Centralize memory allocation for objects (make it easier to review)

Integers and Integer overflow

Integer Arithmetic in C

- Quiz time!
- What does this code produce?
  - `100 200 300`
  - `100 200 44`
  - `100 -56 44`
- Depends on how a, b, and c are defined

```
a = 100;
b = 200;
c = a+b;
printf("%d %d %d\n",
       (int)a,
       (int)b,
       (int)c);
```

Integer Overflow/Underflow

- C defines fixed-width integer types (`short`, `int`, `long`, etc.) that do not always behave like Platonic integers from elementary school.
- Because of the fixed width, it is possible to overflow or wrap maximum expressible number for the type used
  - Or underflow in case of negative numbers

Integer Overflow/Underflow

- How can this be a problem?
- What if n is too large?
- What if n is negative?

```
my_type* foo(int n)
{
    my_type *ptr = malloc(n * sizeof(my_type));
    for(int i = 0; i < n; i++)
    {
        memset(&ptr[i], i, sizeof(my_type));
    }
    return ptr;
}
```

Integer Overflow/Underflow

- When a value of an unsigned integer type overflows (or underflows), it simply wraps.
  - As if arithmetic operation was performed modulo $2^{\text{(size of the type)}}$.
- Overflow (and underflow) of signed integer types is <u>undefined</u> in C.
  - Though most implementations wrap.

Integer Overflow/Underflow

- A common first attempt to check for unsigned overflow looks like this:
  - `if (a+b > UINT32_MAX) …`
- What’s wrong with this check?
  - Assume `a` and `b` are of type `uint32`
- Expression `a+b` cannot hold a value greater than `UINT32_MAX`
  - `a+b == (a+b) mod UINT32_MAX`

Checking for Overflow/Underflow

- Unsigned overflow checks need to use the complementary operation to the one being checked
  - Subtraction to check for addition overflow
    - `if (UINT32_MAX – a < b)`
  - Division to check for multiplication overflow
    - `if ((0 != a) && (UINT32_MAX / a < b))`
- More complex for signed types

```
– if(((a>0) && (b>0) && (a > (INT32_MAX-b)))
  ||  (a<0) && (b<0) && (a < (INT32_MIN-b))))
```

- Do you think people do this?

Integer Overflow/Underflow

- Quick review
- `char`
  - At least 8 bits. `sizeof(char) == 1`
- `short`
  - At least 16 bits
- `int`
  - Natural word size of the architecture, at least 16 bits
- `long`
  - At least 32 bits
- `long long`
  - At least 32 bits

Integer Type Conversion

- Integer type conversions are yet another common source of security vulnerabilities.
  - Whenever a value is changed from one type to another.
- How are values converted from one type to another? What happens to the bit pattern?
  - Truncation
  - Zero-extension
  - Sign-extension

Integer Type Conversion

- ***Truncation*** occurs when a value with a wider type is converted to a narrower type.
- When a value is truncated, its high-order bytes are removed so that it is the same width as the narrower type.

```
uint32_t j = 0xDEADBEEF;
uint16_t i = j;
// i == 0xBEEF
```

Integer Type Conversion

- ***Zero-extension*** occurs when a value with a narrower, unsigned type is converted to a wider type.
- When a value is zero-extended, it is widened so that it is the same width as the wider type.
  - The new bytes are unset (are zero).

```
uint16_t i = 0xBEEF;
uint32_t j = i;
// j == 0x0000BEEF
```

Integer Type Conversion

- ***Sign-extension*** occurs when a value with a narrower, signed type is converted to a wider type.
- When a value is sign-extended, it is widened so that it is the same width as the wider type.
  - If the sign bit of the original value is set, the new bytes are set.
  - If the sign bit of the original value is unset, the new bytes are unset.
- Sign-extension is used for signed types rather than zero-extension because it preserves the value.

```
int8_t i = -127; //           1000 0001
int8_t j = 127;  //           0111 1111
int16_t ki = i;
int16_t kj = j;
// ki == 1111 1111 1000 0001
// kj == 0000 0000 0111 1111
```

Integer Type Conversion

- The following two slides display reference charts that explain particular integer type conversions.
- You have no obligation to memorize these

Integer Type Conversion (for reference)

From	To	Method	Lost or Misinterpreted
unsigned char	signed char	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted
unsigned char	signed short int	Zero-extend	Safe
unsigned char	signed long int	Zero-extend	Safe
unsigned char	unsigned short int	Zero-extend	Safe
unsigned char	unsigned long int	Zero-extend	Safe
unsigned short int	signed char	Preserve low-order byte	Lost
unsigned short int	signed short int	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted
unsigned short int	signed long int	Zero-extend	Safe
unsigned short int	unsigned char	Preserve low-order byte	Lost
unsigned long int	signed char	Preserve low-order byte	Lost
unsigned long int	signed short int	Preserve low-order word	Lost
unsigned long int	signed long int	Preserve bit pattern; high-order bit becomes sign bit	Misinterpreted
unsigned long int	unsigned char	Preserve low-order byte	Lost
unsigned long int	unsigned short int	Preserve low-order word	Lost

Integer Type Conversion (for reference)

From	To	Method	Lost or Misinterpreted
signed char	short int	Sign-extend	Safe
signed char	long int	Sign-extend	Safe
signed char	unsigned char	Preserve bit pattern; high-order bit no longer sign bit	Misinterpreted
signed char	unsigned short int	Sign-extend to short; convert short to unsigned short	Lost
signed char	unsigned long int	Sign-extend to long; convert long to unsigned long	Lost
signed short int	char	Truncate to low-order byte	Lost
signed short int	long int	Sign-extend	Safe
signed short int	unsigned char	Truncate to low-order byte	Lost
signed short int	unsigned short int	Preserve bit pattern; high-order bit no longer sign bit	Misinterpreted
signed short int	unsigned long int	Sign extend to long; convert long to unsigned	Lost
signed long int	char	Truncate to low order byte	Lost
signed long int	short int	Truncate to low order bytes	Lost
signed long int	unsigned char	Truncate to low order byte	Lost
signed long int	unsigned short int	Truncate to low order bytes	Lost

Integer Type Conversion

- Type conversion can be **explicit** or **implicit**
- Explicit type conversions use the casting syntax:

```
– int i = (int) 4.5;
```

- Implicit type conversions:

```
– signed short i = 1;   // assignment conversion
– unsigned int j = 2; // assignment conversion
– if (i < j) { … }    // comparison conversion
– void function(int arg);
  function(5.3);      // function argument conversion
```

`Integer Type Conversion`

- Implicit conversions are common in arithmetic expressions.
- Prior to evaluation within an expression, numeric operands are converted:
  - First, they are "promoted" to be of at least `int` rank
    - If a `signed int` can represent all values of the original type, the value is converted to `signed int`; otherwise, it is converted to `unsigned int`
    - Rank order (descending):

```
– [un]signed long long    >
–   [un]signed long       >
–     [un]signed int      >
–       [un]signed short >
–         char, [un]signed char
```

  - Then they are converted to ***Common Real Type***
- Conversion rules are **complex**

Integer Type Conversion: Common Real Type

- Are both operands of the same type?
  - No further conversion is needed.
- If not, are both operand signed or both unsigned?
  - The operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank. (*I did not make you read the full rank rules, you should thank me*)
- If not, is rank of unsigned operand >= rank of signed operand?
  - The operand with signed integer type is converted to the type of the operand with unsigned integer type.
- If not, can the type of the signed operand represent all of the values of the type of the unsigned operand?
  - The operand with unsigned integer type is converted to the type of the operand with signed integer type.
- If not, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

Why did they do this thing?!?!

- Integer promotions allow operations to be performed in a natural size for an architecture.
  - The `signed int` and `unsigned int` types correspond to the natural word size of an architecture.
  - Most efficient to perform operations in the natural size of an architecture.
- Integer promotions avoid arithmetic errors caused by the overflow of intermediate values.
  - For example, consider this code.
  - What result do you expect?
  - What result would you get without integer promotion?
  - Thus, integer promotions also help operations produce the expected, intuitive result by avoiding overflow.

```
signed char a      = 100;
signed char b      = 3;
signed char c      = 4;
signed char result = a*b / c;
```

Integer Type Conversion

- The rules for type conversions are complex and not always intuitive.
- It is not always obvious where type conversions occur.
- For example: what does this code print?
  - Not what you'd like it to.
- Let's walk through conversion to common real type…

```c
signed int i = -1;
unsigned int j = 1;
if (i < j) printf("foo\n");
else printf("bar\n");
```

Integer Type Conversion

- Both operands have rank >= `int`?
  - Yes.
- Both operands of the same type?
  - No.
- Both operands signed or both unsigned?
  - No.

```c
signed int i = -1;
unsigned int j = 1;
if (i < j) printf("foo\n");
else printf("bar\n");
```

Integer Type Conversion

- Rank of unsigned operand >= rank of signed operand?
  - Yes.
  - Then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
  - `(unsigned int)-1 == UINT_MAX`

```c
signed int i = -1;
unsigned int j = 1;
if (i < j) printf("foo\n");
else printf("bar\n");
```

Integer Type Conversion

- Take a moment to review this code.
- Note the check to verify that port number is >= 1024 unless the user is root.
  - The first 1024 ports are assigned to critical network services, such as HTTPS, POP, and SMTP and are considered privileged.

```c
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root)
        // handle error
else
        sockaddr.sin_port = port;
```

Integer Type Conversion

- Where is the vulnerability?
- Is there an input that will allow a normal user to open a connection on a privileged port?

```
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root)
        // handle error
else
        sockaddr.sin_port = port;
```

Integer Type Conversion

- The field `sin_port` is declared as a 16-bit unsigned integer.
  - Range $[0, 2^{16} - 1]$.
- The variable `port` is declared as a 32-bit signed integer.
  - Range $[-2^{31}, 2^{31} - 1]$.
- When `port` is assigned to `sin_port`, the two high-order bytes of the value are truncated and the port number is changed.

```
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root)
        // handle error
else
        sockaddr.sin_port = port;
```

Integer Type Conversion

- What happens if an attacker sets the variable port to 65979?
- The comparison within the if statement is between two signed 32-bit integers and 65979 is greater than 1024.
- But, note the hexadecimal representation of 65979
  - 0x000101BB
- When port is assigned to sin_port, its value is truncated, and sin_port is set to 0x01BB, or 443.
- Port 443 is used for HTTPS traffic.

```c
struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        //...
};
//...
sockaddr_in sockaddr;
int port; // user input
//...
if(port < 1024 && !is_root)
        // handle error
else
        sockaddr.sin_port = port;
```

What to do?

- Strongly typed language
  - Most such problems go away in Java for example
- Runtime checking
  - gcc –ftrapv (trap on signed overflow on add, sub, mult)
  - Safe libraries (David LeBlanc's SafeInt class)
    - Template overrides all operators (not the fastest)
    - Use when variable can be influenced by untrusted input
- Static checking (range analysis)
  - Sarkar et al, "Flow-insensitive Static Analysis for Detecting Integer Anomalies in Programs"

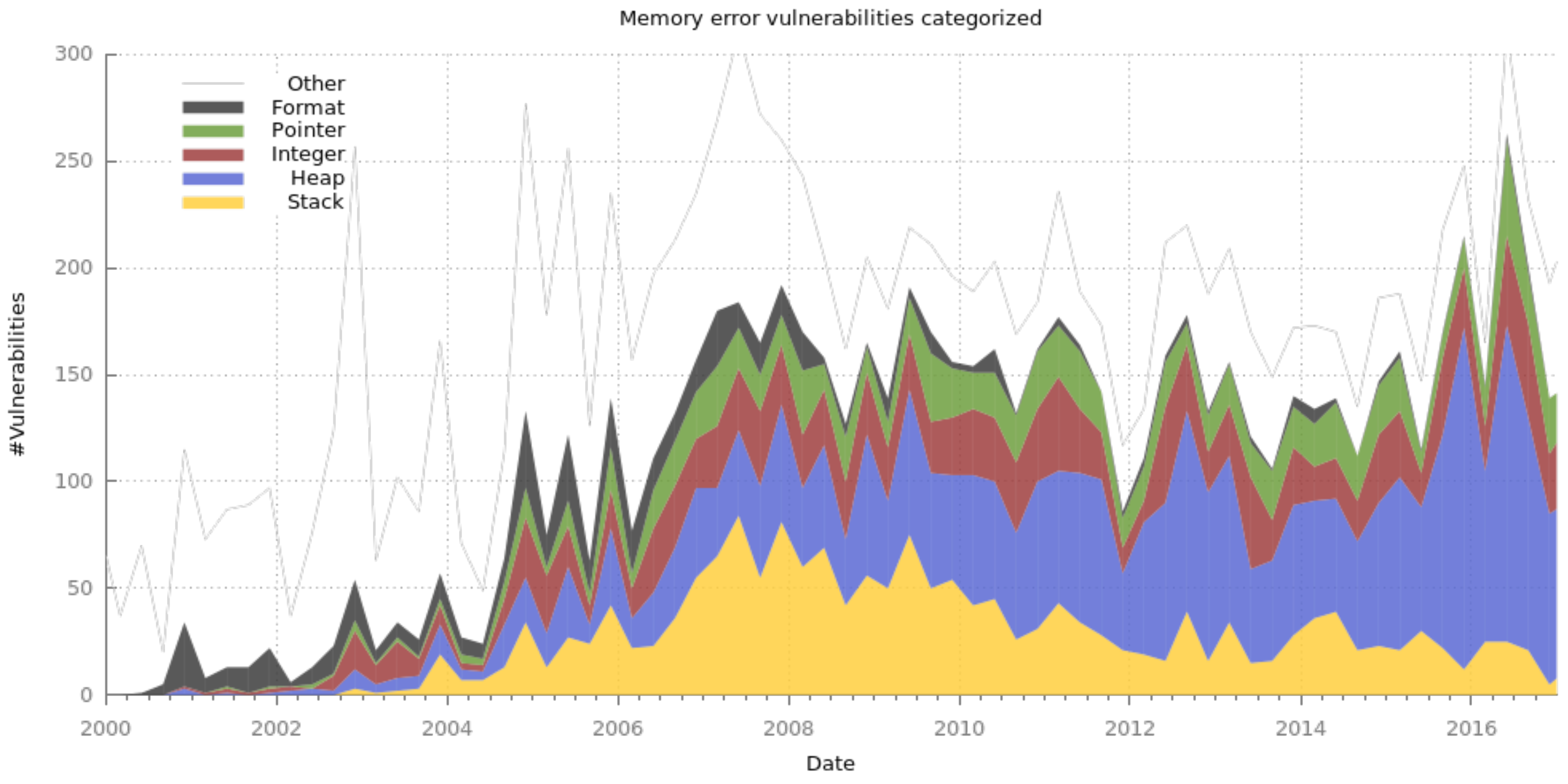
Using SafeInt

```
void *ConcatBytes(void *buf1,
                  SafeInt<int> len1,

                  char *buf2,
                  SafeInt<int> len2)

{

  void *buf = malloc(len1 + len2);

  if (buf == NULL) return;

  memcpy(buf, buf1, len1);

  memcpy(buf + len1, buf2, len2);

}
```

Overload "+"
Will throw exception on overflow

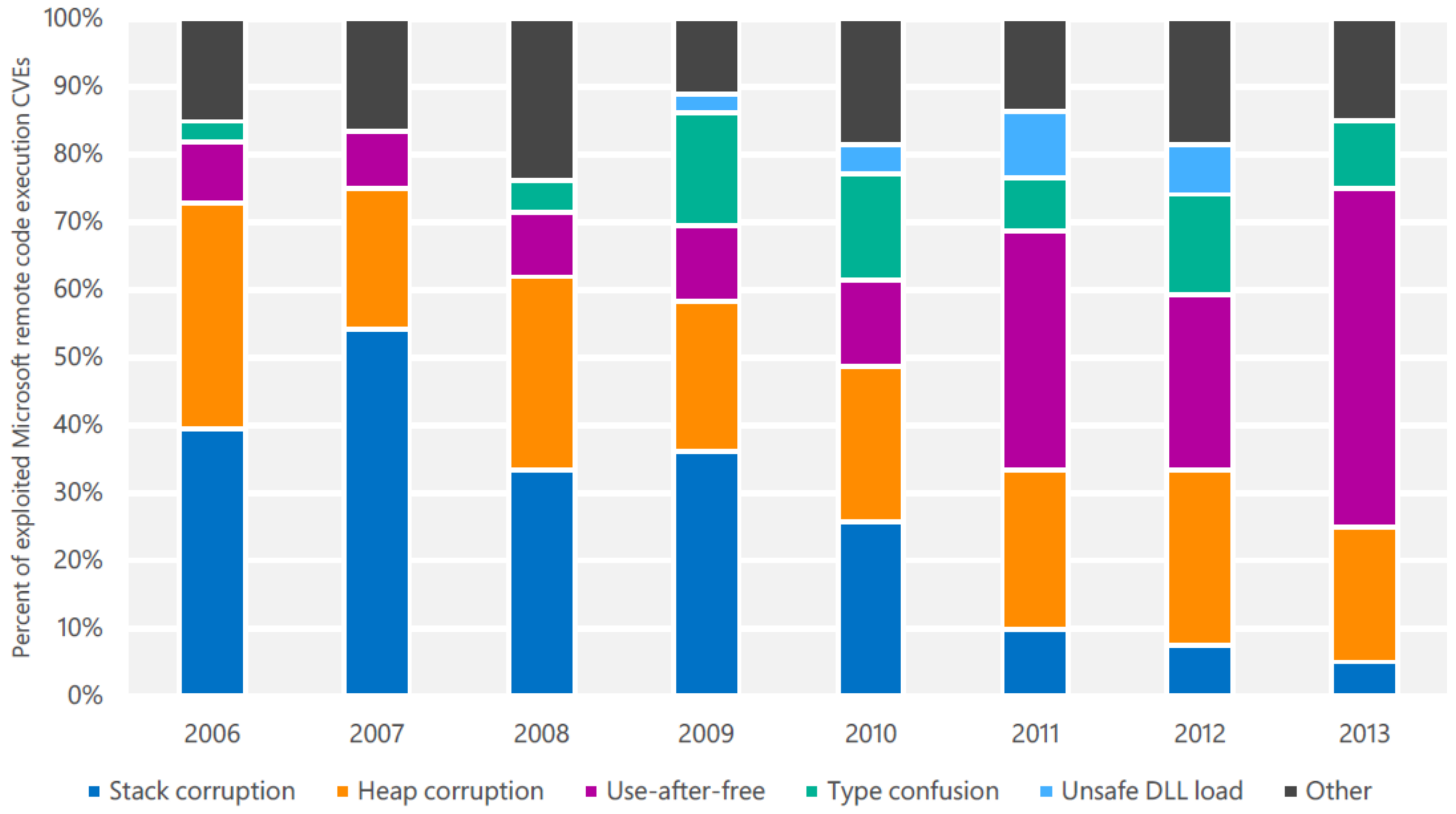
Additional Resources

- *Understanding glibc malloc* by sploitfun
  - https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- Shellphish how2heap
  - https://github.com/shellphish/how2heap
- *Vudo - An object superstitiously believed to embody magical powers* by Michel "MaXX" Kaempf
  - http://phrack.org/issues/57/8.html#article
- *Advanced Doug lea's malloc exploits* by jp
  - http://phrack.org/issues/61/6.html#article
- Plus references form Lectures 3, 4, and 5.



http://vvdveen.com/memory-errors/

Figure 3. The root causes of exploited Microsoft remote code execution CVEs, by year of security bulletin



Microsoft Security Intelligence Report Volume 16 (2013), https://www.microsoft.com/en-us/security/intelligence-report