# CSE 127 Computer Security

Rob Turner, Spring 2018, Lecture 6

## Low Level Software Security IV: Heap Corruption

# Resources

Smashing The Stack For Fun And Profit

Getting Around Non-Executable Stack (and Fix)

The Advanced Return-into-lib(c) Exploits

msfrop / ROPgadget / ropper

Q: Exploit Hardening Made Easy

# Resources

Return-Oriented Programming

Jump-Oriented Programming

Call-Oriented Programming

Blind Return-Oriented Programming

Sigreturn-Oriented Programming

Counterfeit-Object-Oriented Programming

# Recap

- Integer Overflow and  Underflow
- When do type conversions happen? (Implicit and Explicit)
- Truncation, Sign-Extension, and Zero-Extension
- Width, Rank, and Real Common Type
- Type Conversion Rules
- Return-to-lib(c)
- Return-to-text
- Return-Oriented Programming

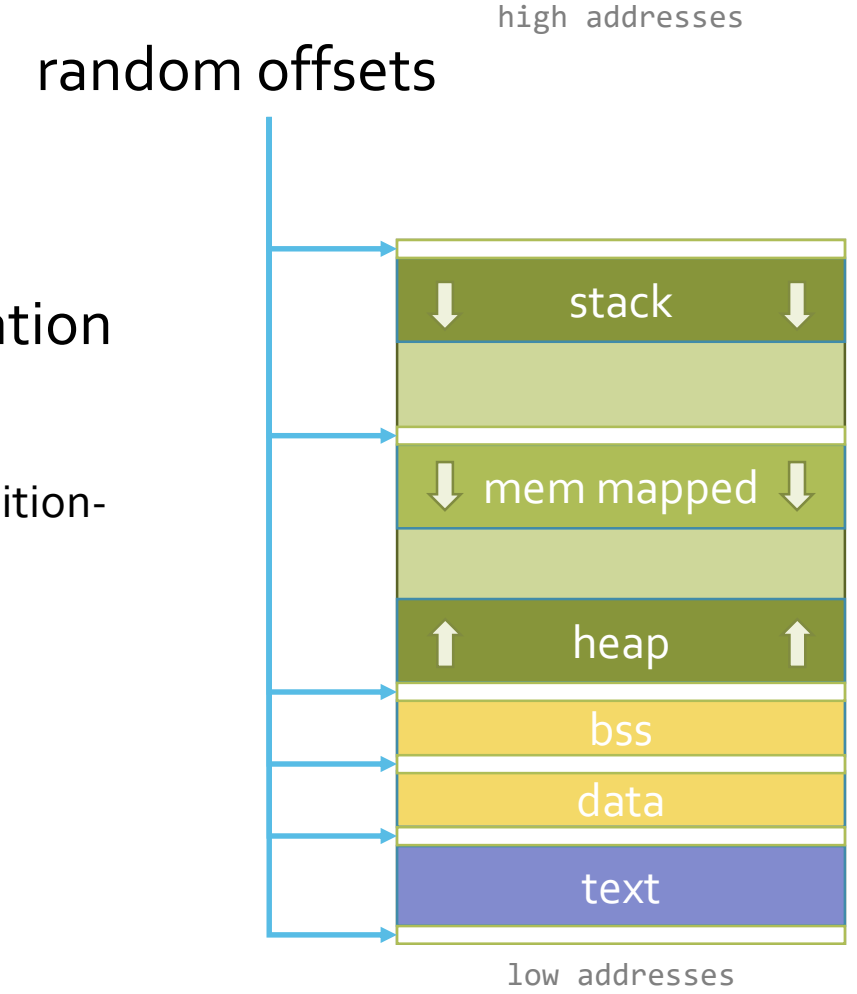
# Randomization and Diversity

- Stack Gap

random offset {

# Randomization and Diversity

- Stack Gap
- Address Space Layout Randomization (ASLR)
  - Position-Independent Code (PIC) and Position-Independent Executables (PIE)
- Fine-Grained Diversity



# Randomization and Diversity

- Bypasses?
  - Information Leaks
  - Entropy
  - NOP Sleds
  - Partial Overwrites

# Control-Flow Integrity

# Control-Flow Transfers

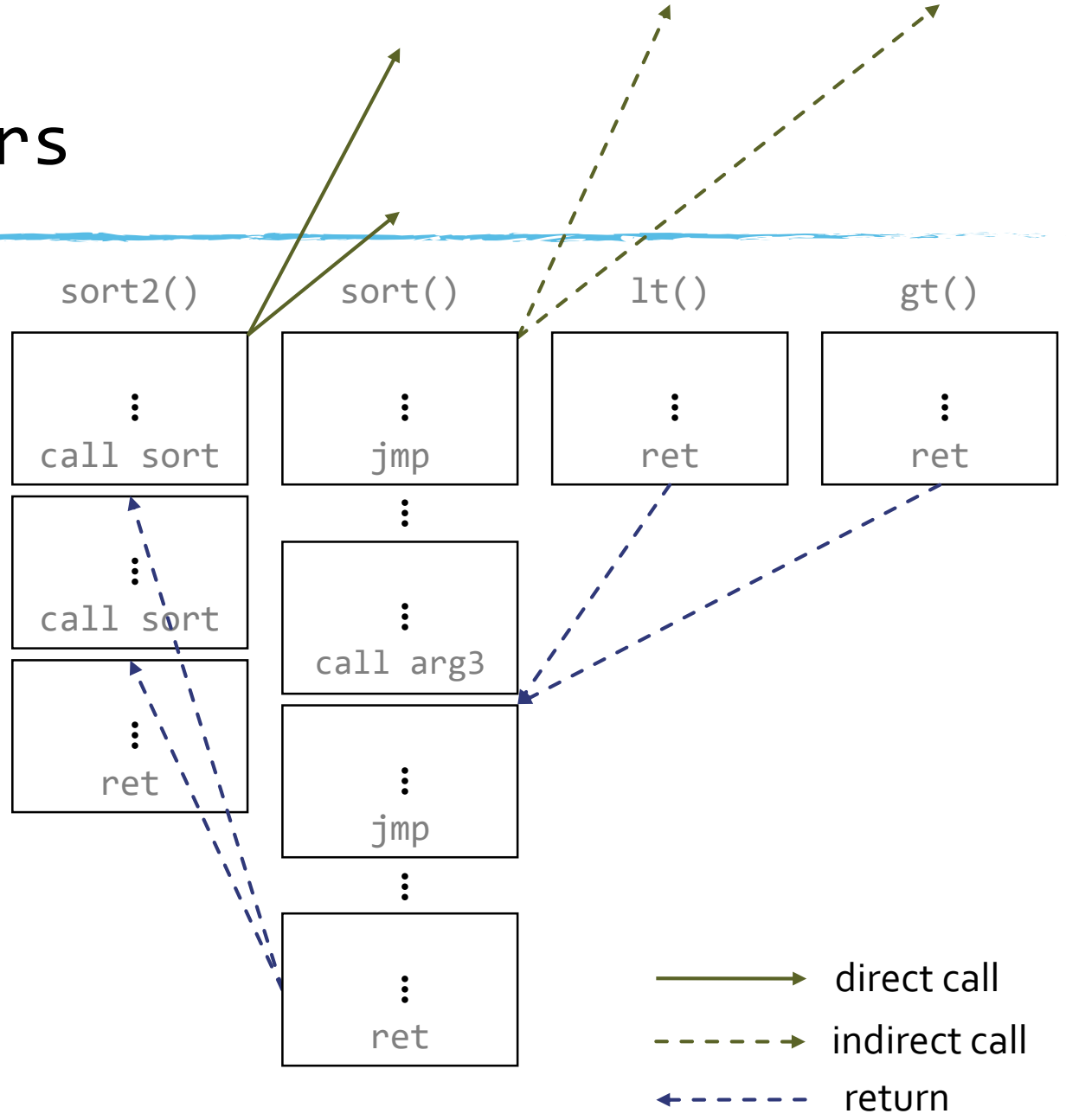
- Direct
  - Fixed Calls and Branches
- Indirect, Computed, or Free
  - Indirect Calls and Branches, Returns

# Control-Flow Transfers

```
void sort2( int a[],
            int b[],
            int len )
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y)
{ return x < y; }

bool gt(int x, int y)
{ return x > y; }
```



# Control-Flow Integrity

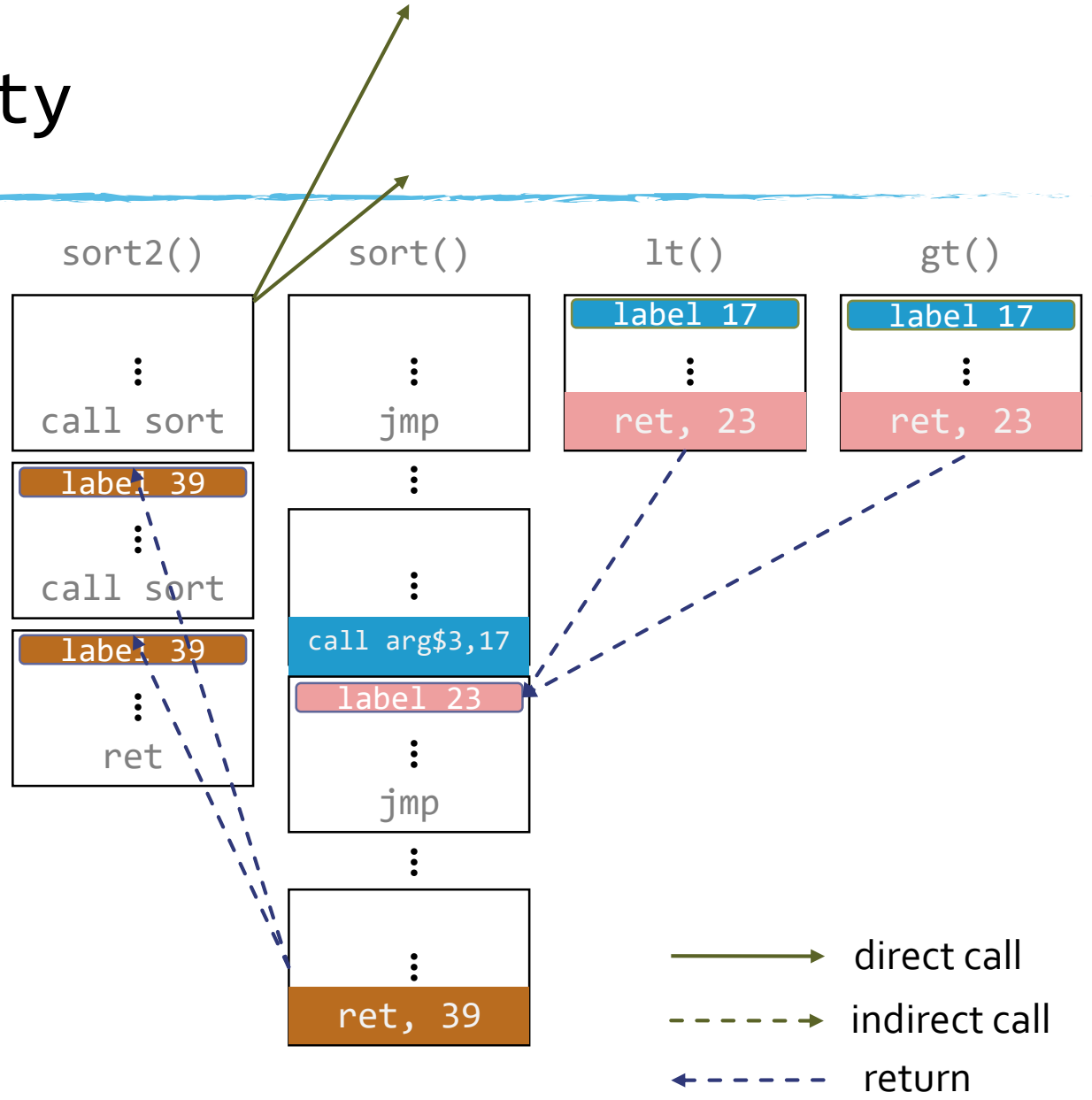
- Statically compute a control flow graph. Dynamically ensure that the program never deviates from this graph.
- For free branches, assign a label to each possible destination.
- Assume DEP is present.
- Instrument free branches with checks that compare the label at a destination with a constant value to ensure the destination is a valid address.
- *Control-Flow Integrity: Principles, Implementations, and Applications*

# Control Flow Integrity

```
void sort2( int a[],
            int b[],
            int len )
{
    sort(a, len, lt);
    sort(b, len, gt);
}

bool lt(int x, int y)
{ return x < y; }

bool gt(int x, int y)
{ return x > y; }
```



# CFI Limitations

- Performance Overhead
- Implementation Challenges
  - Creating/Maintaining a Control-Flow Graph
- Precision
- A Few Things Outside the CFI Attacker Model:
  - Data-Only Attacks
  - Interpreters
  - Operating System Kernels – Why?

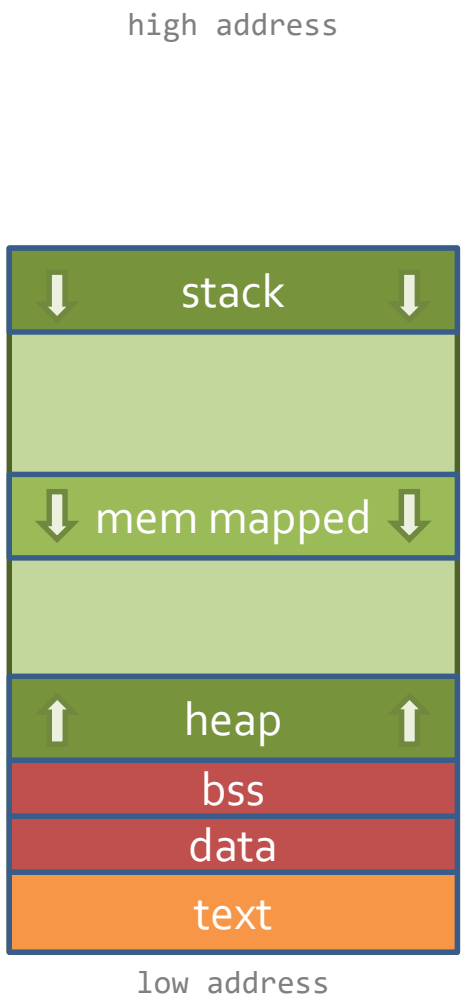
# CFI Limitations

- *Out of Control: Overcoming Control-Flow Integrity*
- *Stitching the Gadgets: On the Ineffectiveness of Coarse Grained CFI*

# Heap Vulnerabilities

# Heap Corruption

- Additional dynamically-allocated data is stored on the "heap"
- Heap manager provides ability to allocate and deallocate memory buffers
  - `malloc()` and `free()`
  - 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"?



# Heap Corruption

- Heap-based buffer overflows
  - Missing Checks
  - Flawed Checks
  - Integer Overflow
  - Etc.
- Pointer types, like other types in C, can also have casting and arithmetic problems.
  - Be very careful when manipulating pointers.
    - Keep in mind that pointer arithmetic is scaled by the size of the pointed-to type

# Heap Corruption

- What if the attacker is able to compromise data on the heap?
- What can be overwritten?
  - Program data on the heap
  - Heap metadata
- 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
      - Direct transfer of control when function is called through overwritten pointer
      - vTables

# vTables

- How are virtual function calls implemented in Object-Oriented languages?
- 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

```
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 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

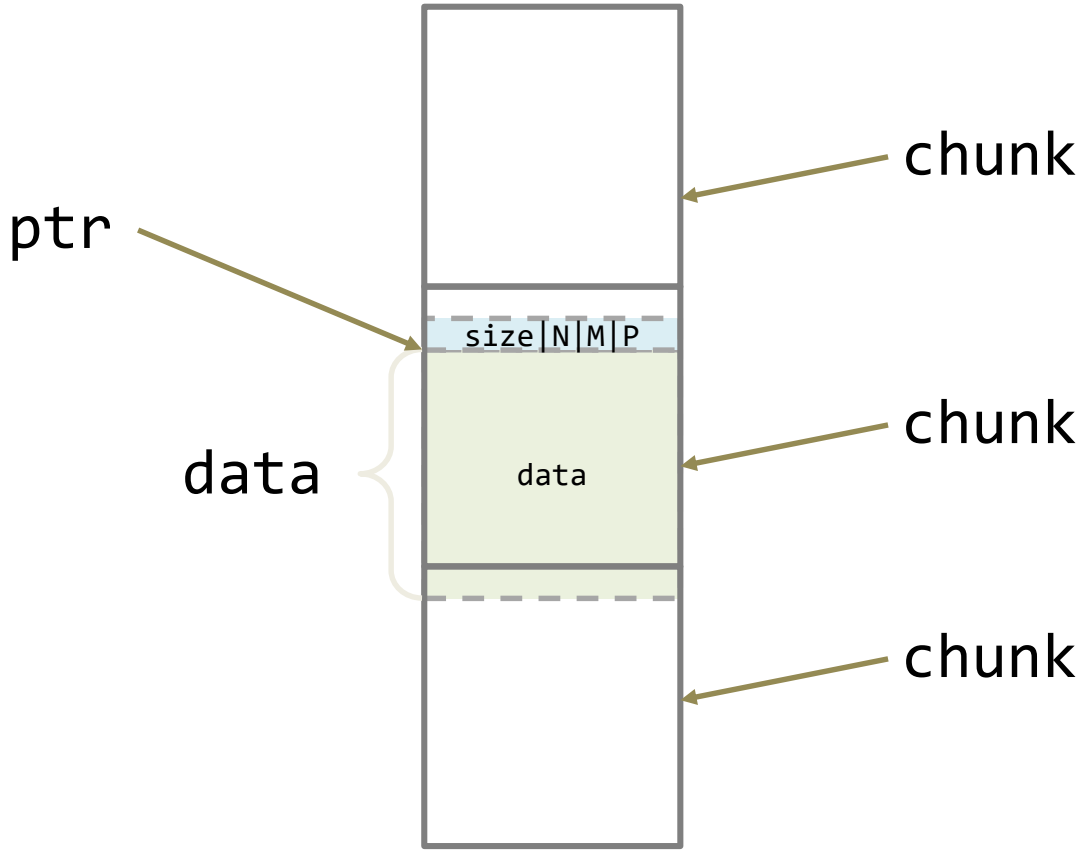
- The heap is managed by the aptly-named heap manager
- There are many different heap managers, even within a single system
  - Some are optimized for allocations of a certain size, for speed, for space efficiency, etc.
  - We will focus on Doug Lea's heap, aka glibc dlmalloc

# 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.

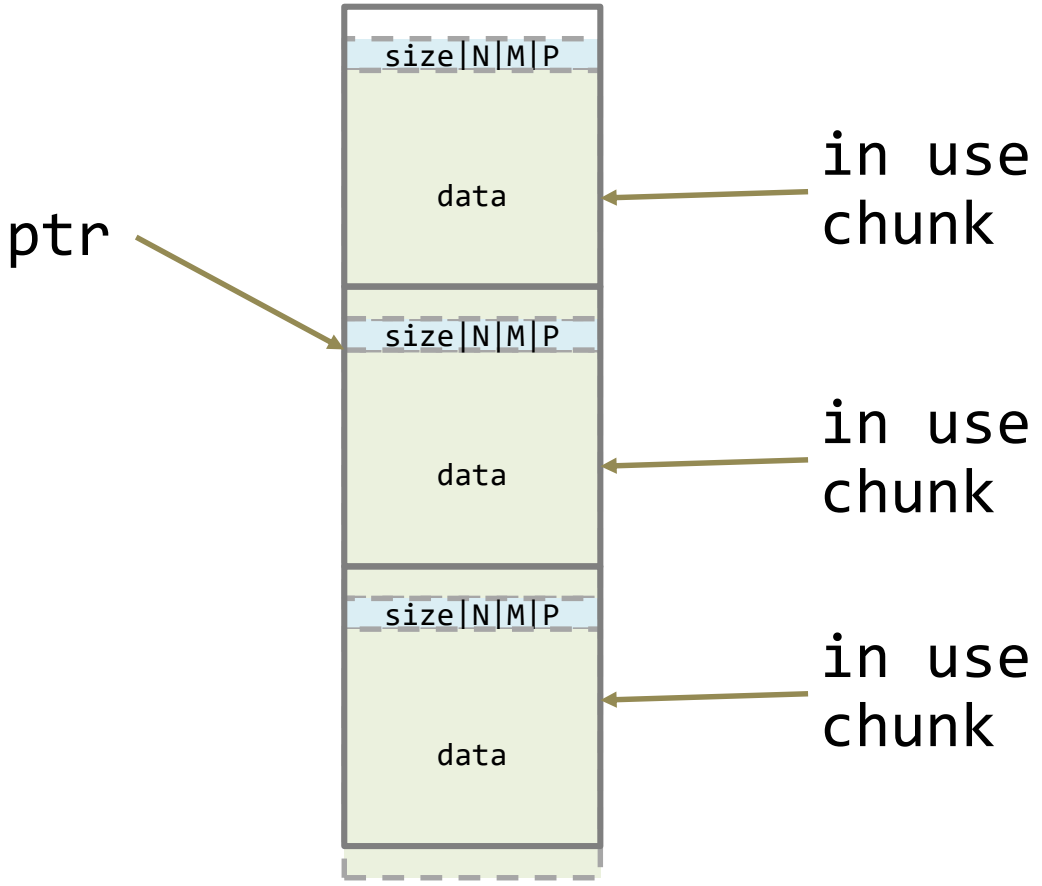
# Heap Management (glibc dlmalloc)

- 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



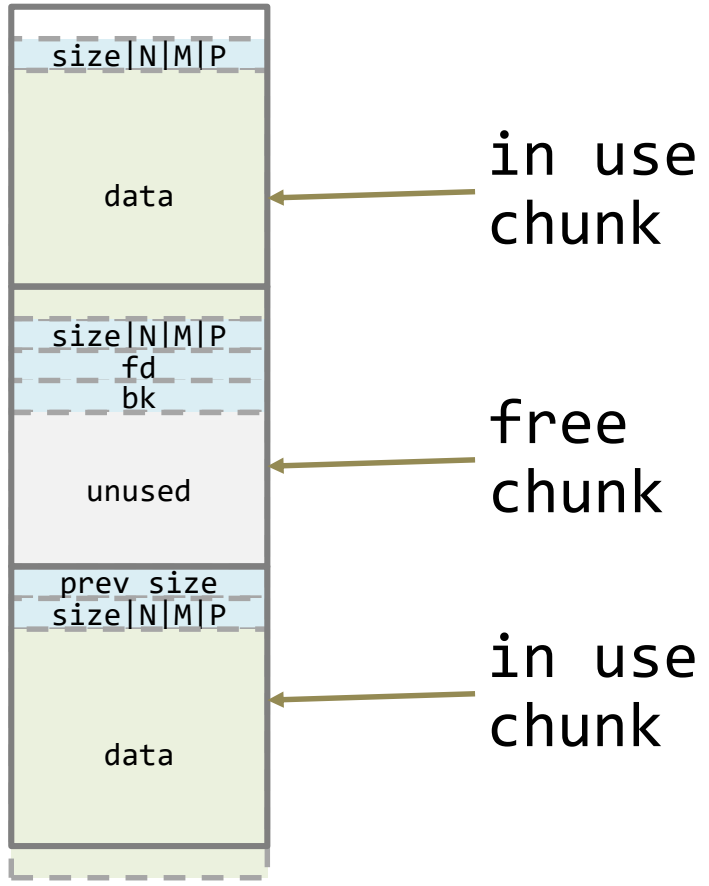
# In Use Chunk

- 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
- Do you see where this is going?



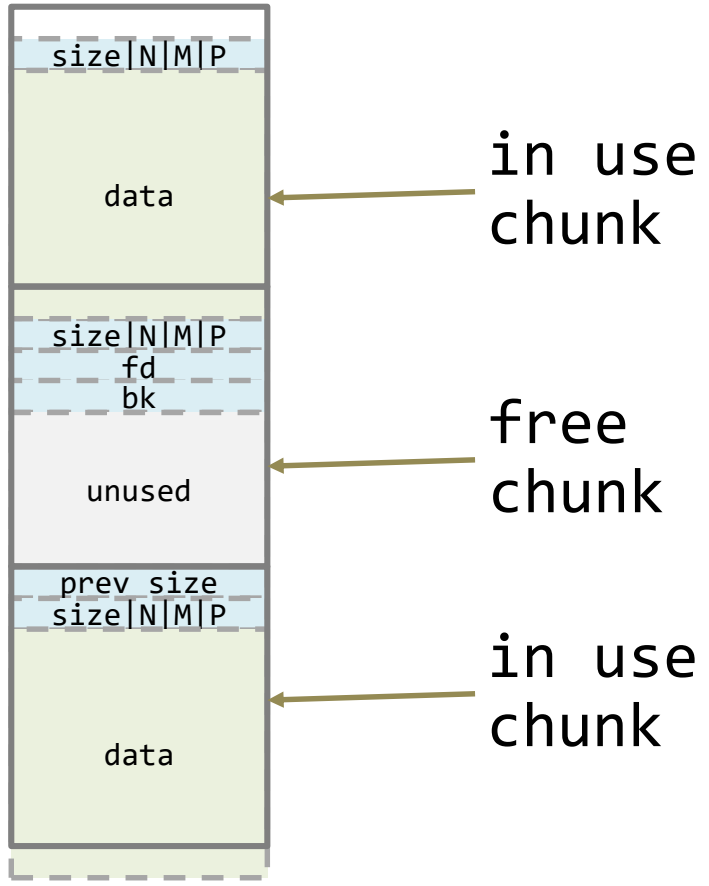
# Free Chunk

- 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



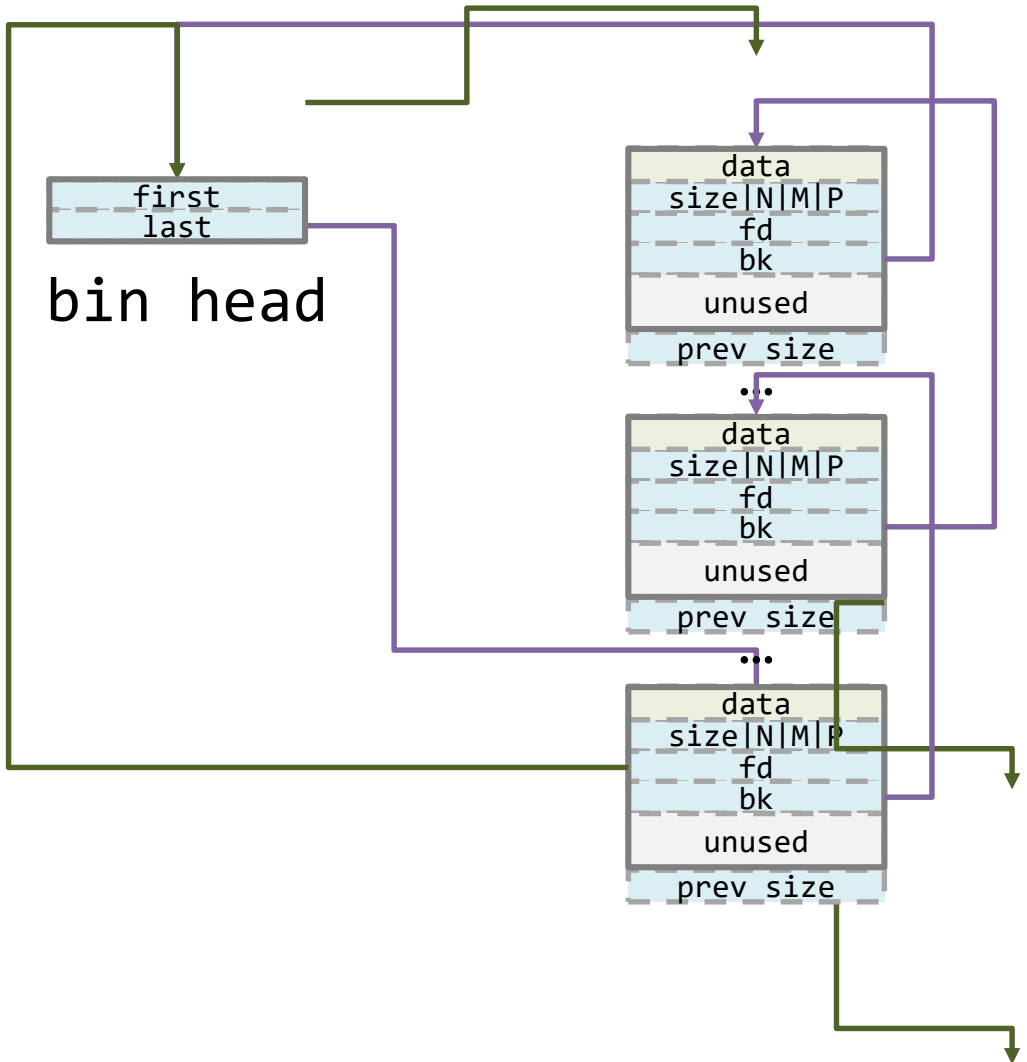
# Free Chunk

- Additionally, the last word of unused data (first word of next chunk) contains a copy of the size of the free chunk.
- Why?
- What happens if the chunk adjacent to the free chunk is also freed?



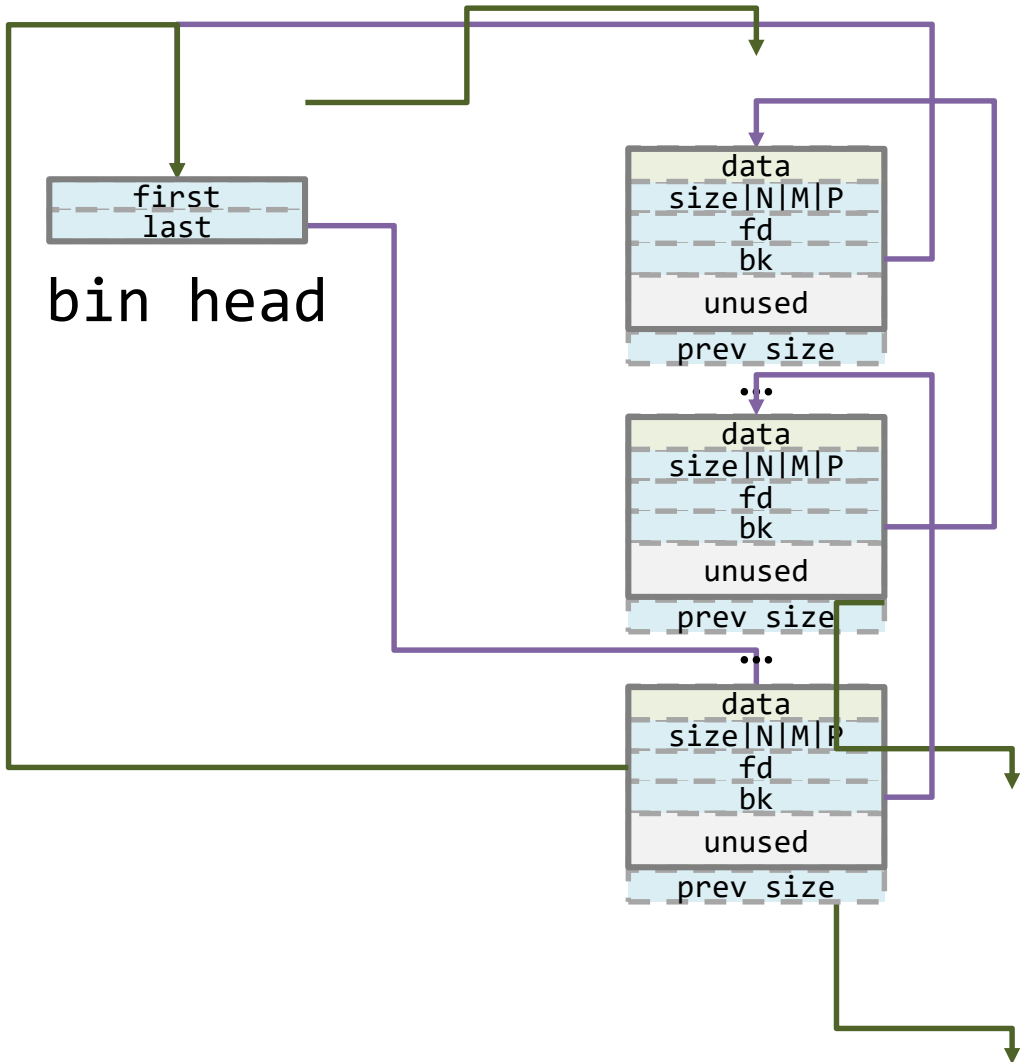
# Free List

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



# Free List

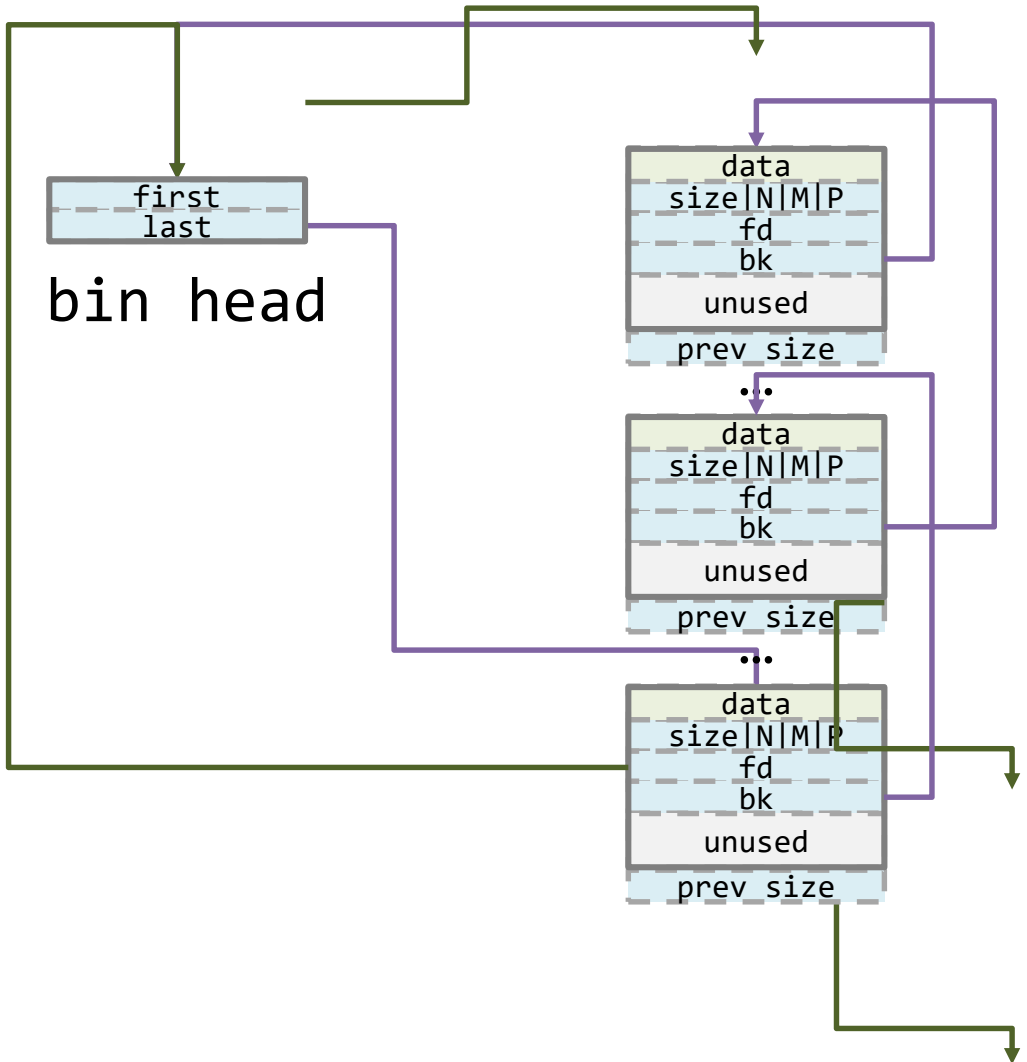
- Chunks are inserted into the list when they are freed.
- Chunks are removed from the list if they get allocated, or if they need to be combined with a newly-freed adjacent chunk



# Free List

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

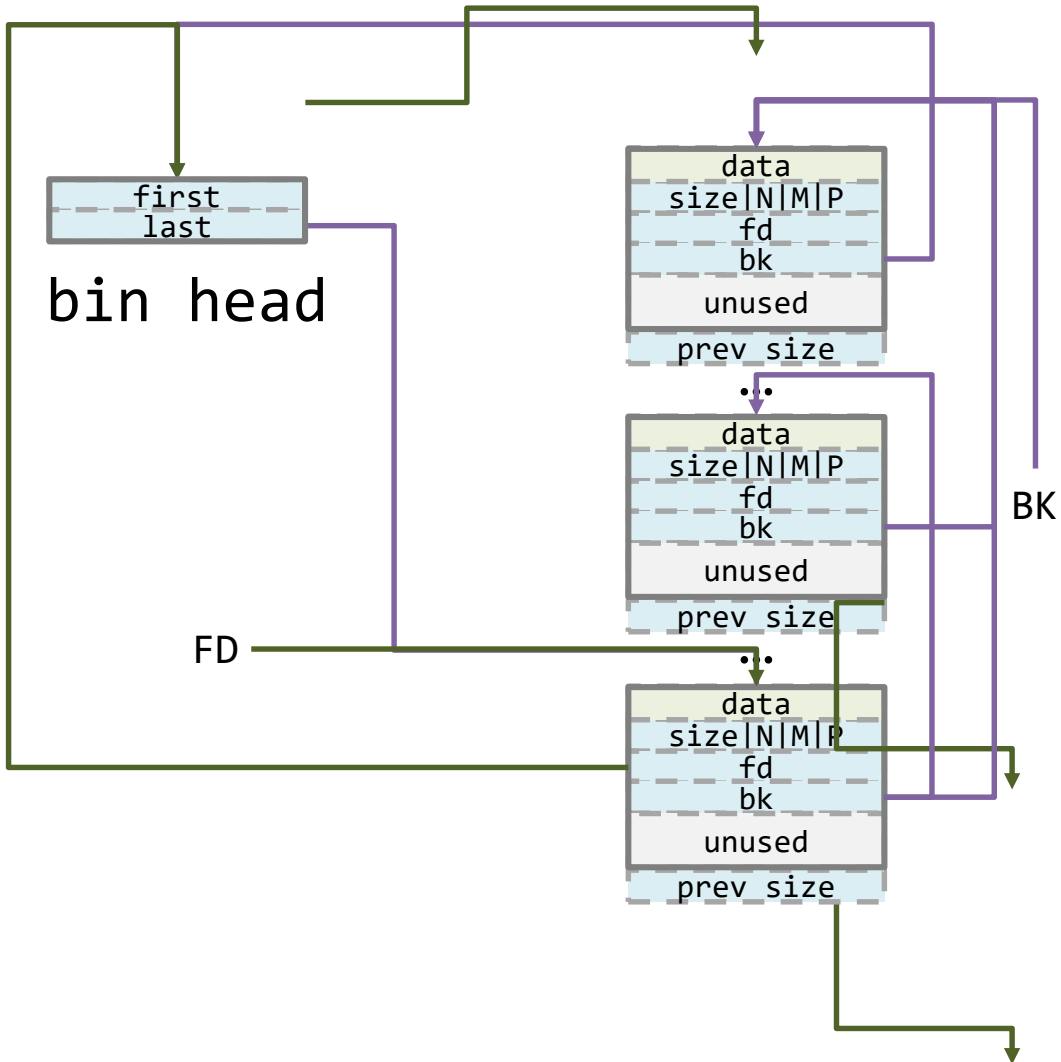
```
#define unlink(P, BK, FD)
{
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```



# 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

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

# Heap Corruption

- How can attacker corrupt the control data in a free chunk?
- Simple overflow
- Indirect overwrite
- Use after free
- Fake chunk

# 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

- 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

- Note that in a multi-threaded application, the second malloc() may not be obvious.
- Just because usage 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);
```

# Recap

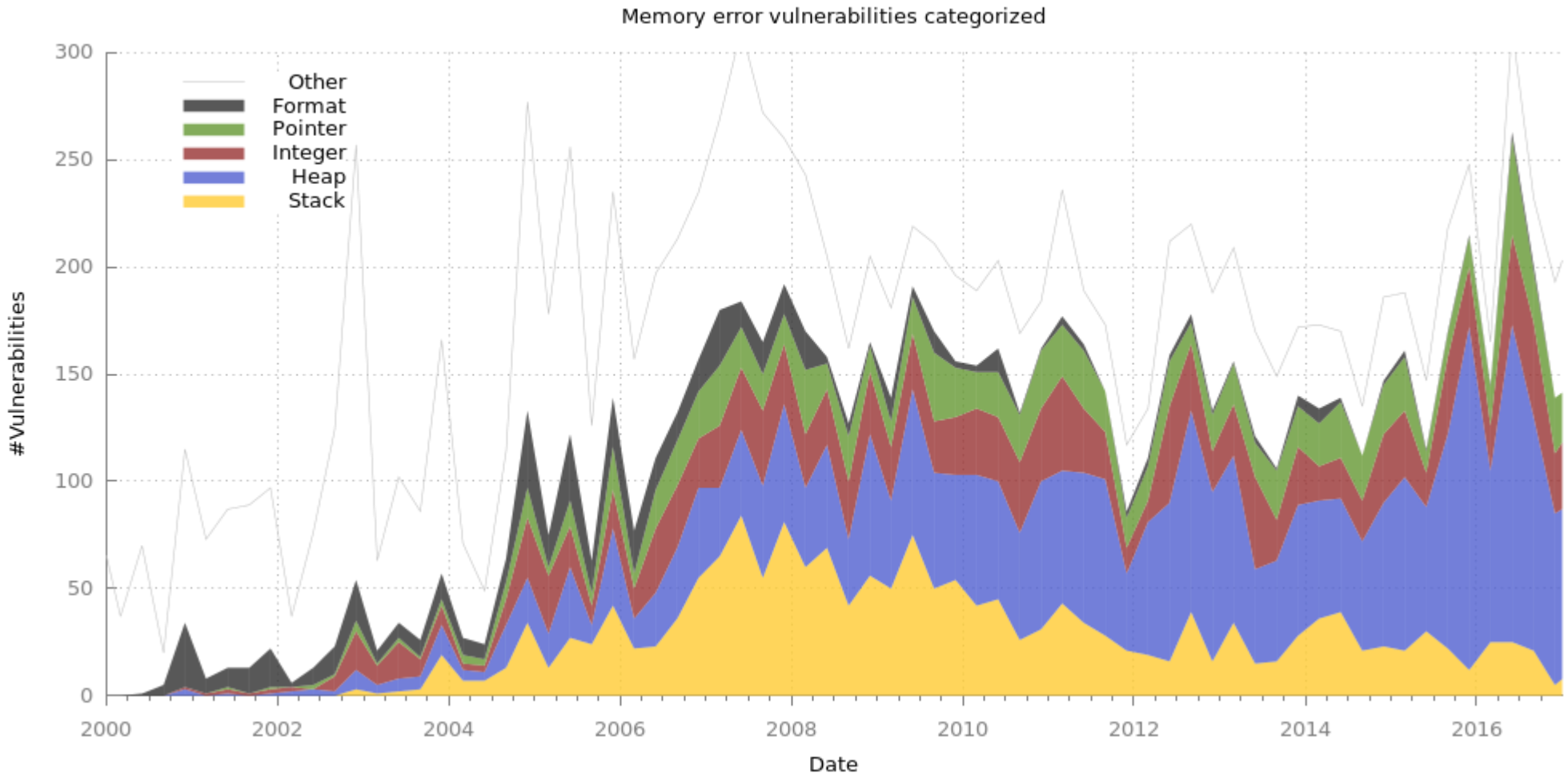
- Randomization and Diversity
- Control-Flow Integrity
- Heap Exploitation
- Use-After-Free and Double Free

# Resources

- A Memory Allocator
- Once Upon a Free()…
- Protostar

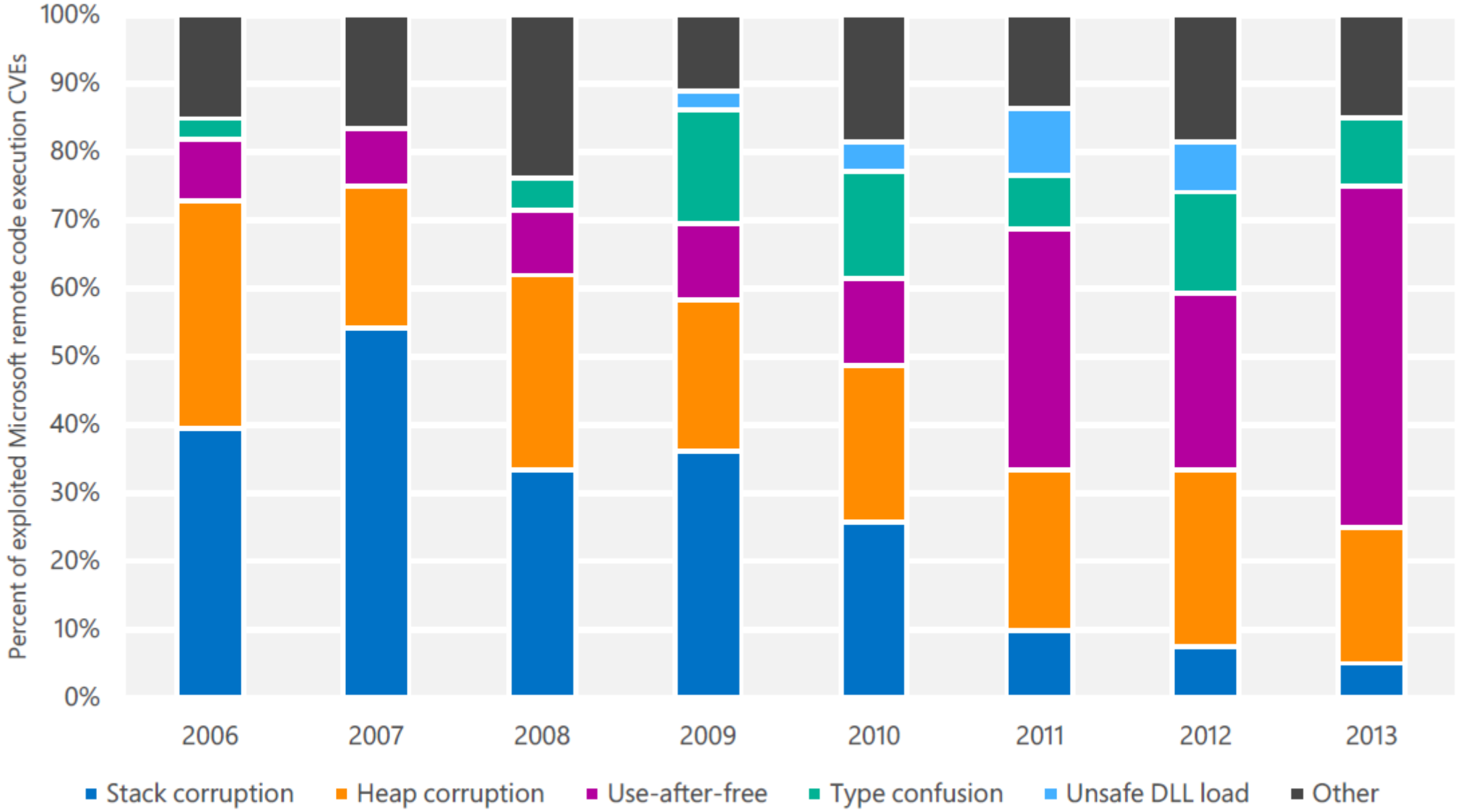
# Additional Resources

- *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.



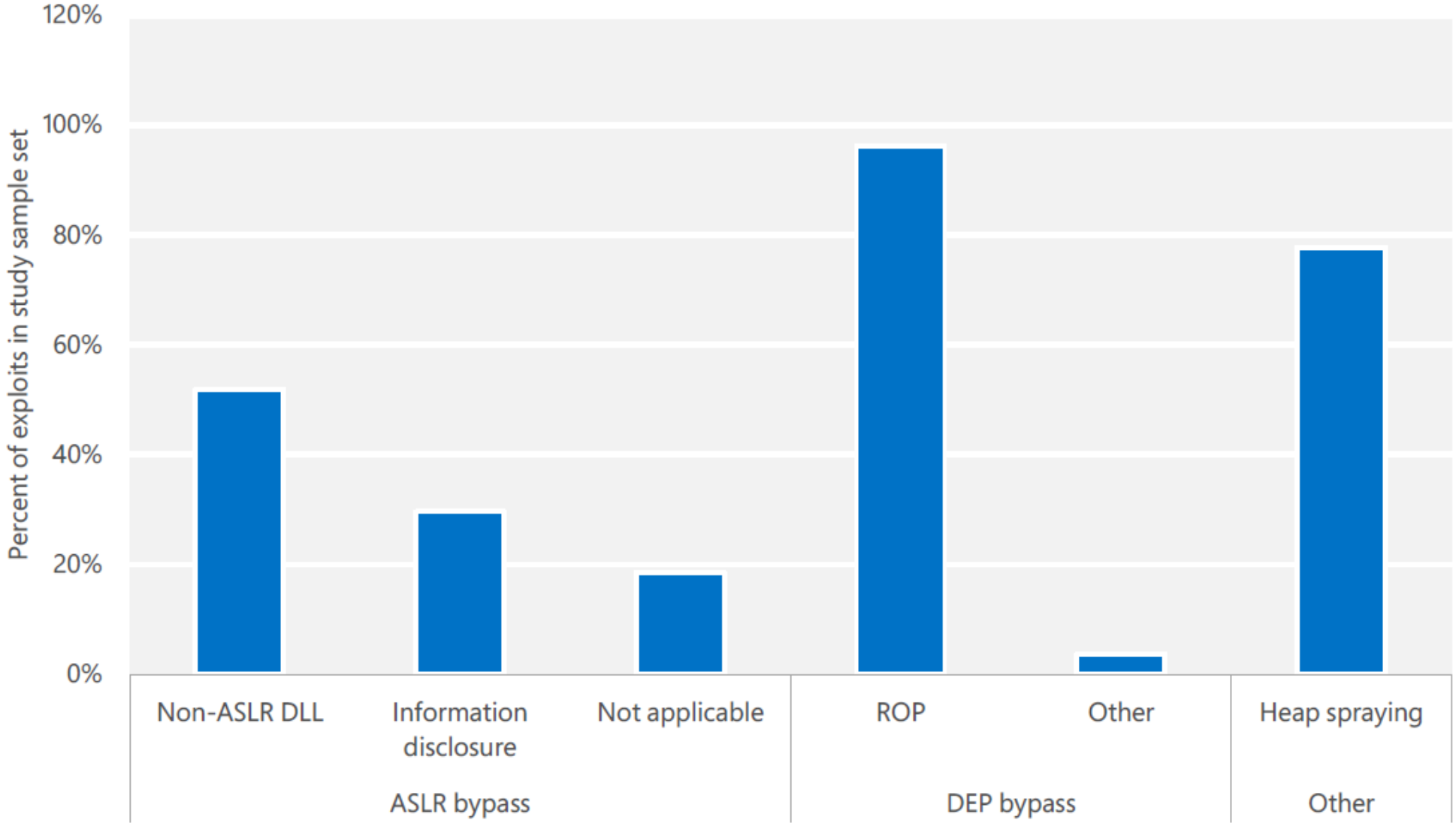

http://vvdveen.com/memory-errors/

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

Figure 4. Techniques used by exploits targeting Microsoft products, January 2012–February 2014




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

# Homework

- Read Chapter 7 from *The Craft of System Security*
- Second project is due Monday (4/23 @ 10pm)

# Next Lecture…

Cryptography I