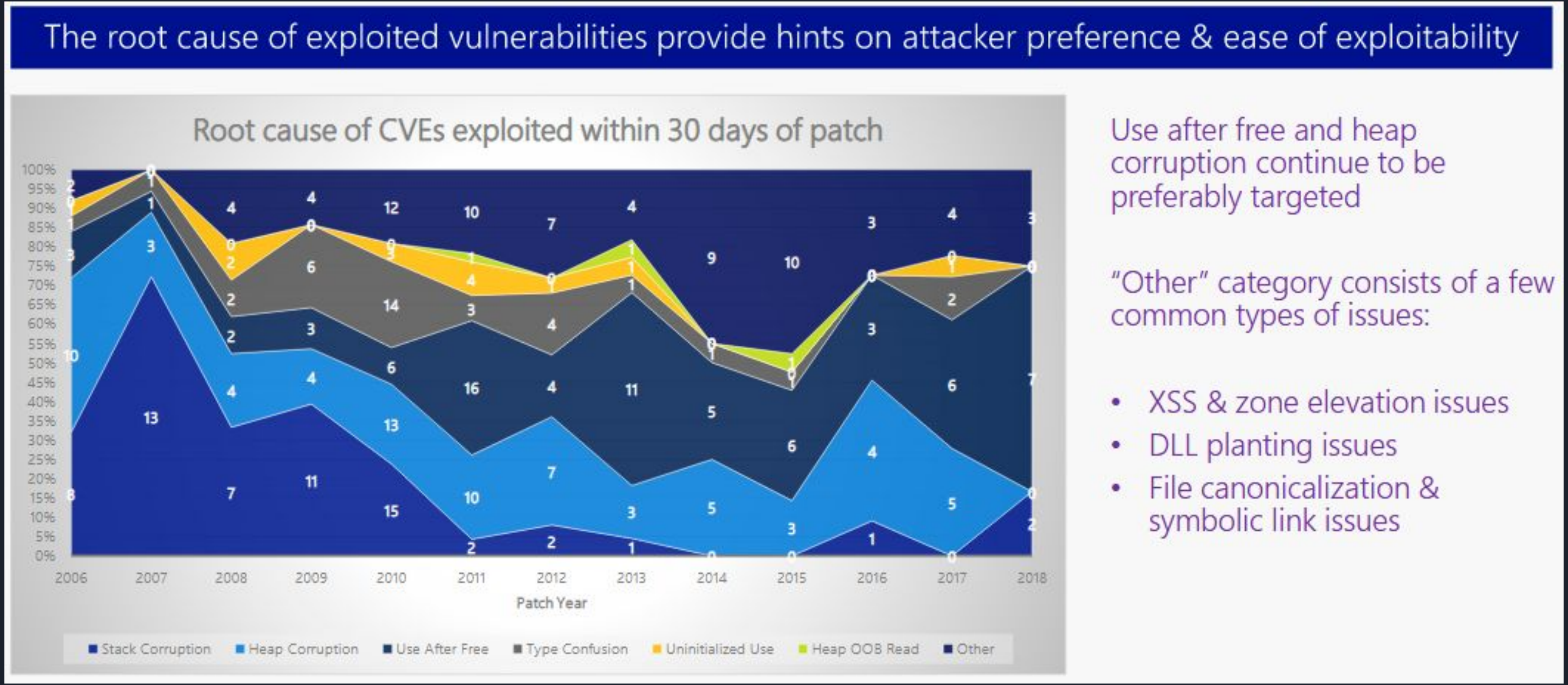
Attacking the Heap

By Joshua Wong

Why should you learn heap exploitation?



The root cause of exploited vulnerabilities provide hints on attacker preference & ease of exploitability

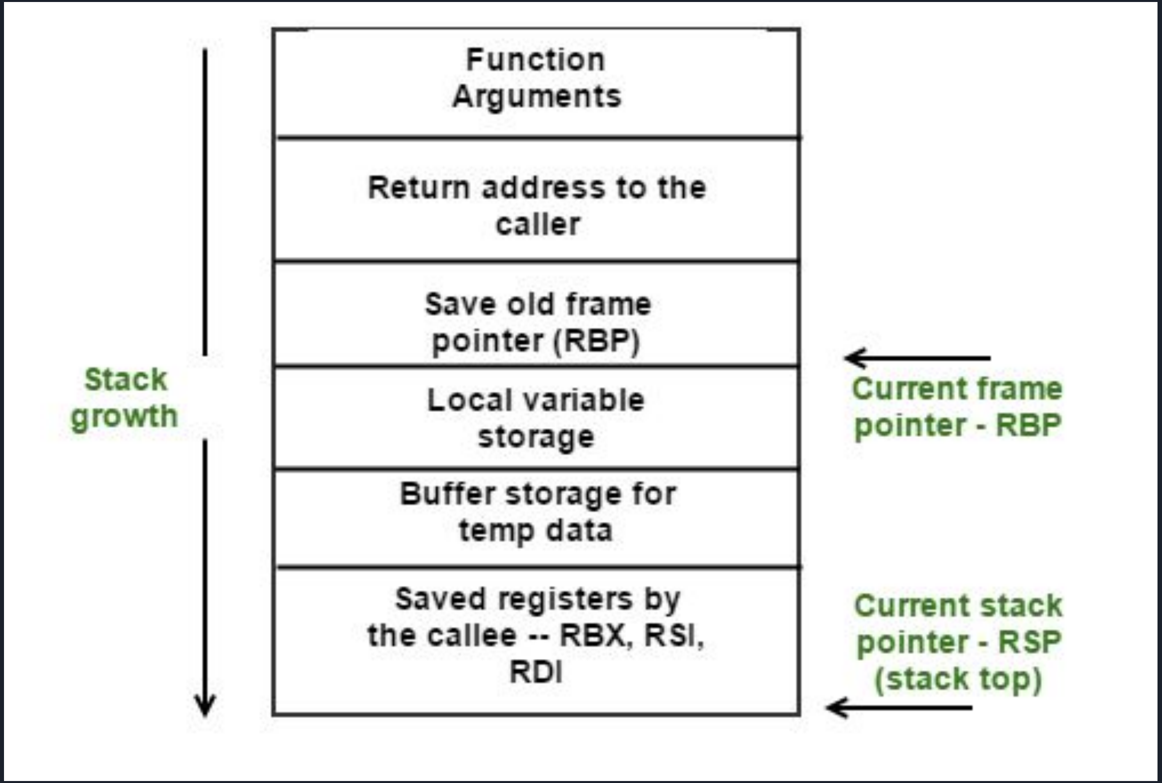
Root cause of CVEs exploited within 30 days of patch

Use after free and heap corruption continue to be preferably targeted

"Other" category consists of a few common types of issues:

- XSS & zone elevation issues
- DLL planting issues
- File canonicalization & symbolic link issues

alright, so we have this stack thing...



Function Arguments

Return address to the caller

Save old frame pointer (RBP)

Local variable storage

Buffer storage for temp data

Saved registers by the callee -- RBX, RSI, RDI

Stack growth
Current frame pointer - RBP
Current stack pointer - RSP (stack top)

but what if we...

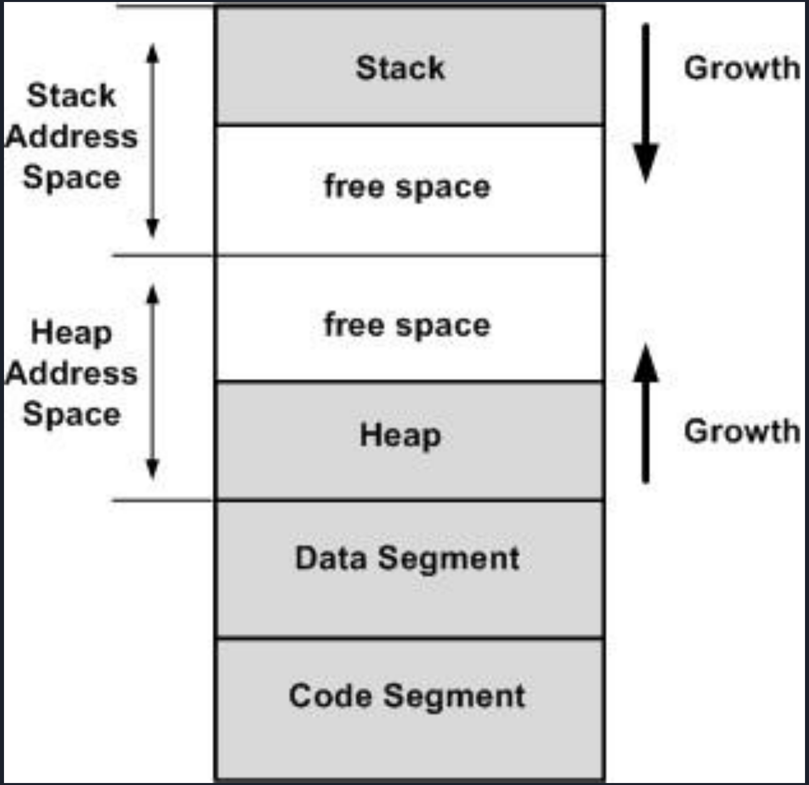
- want variables to stick around after a function call?

```
string a;
void doSomething() {
     a = "hello";
}
// A should still exist outside of doSomething()
```

- want to pick how much memory a variable uses at runtime?

```
int[] array;
void doSomething(int size) {
     array = new int[size]; // don't know size at runtime
}
```

solution: keep some memory aside



what should a heap do?

Should allow a program to:

- Ask for a chunk of memory
- Give the memory back when it's done
    - So we can give it to something else

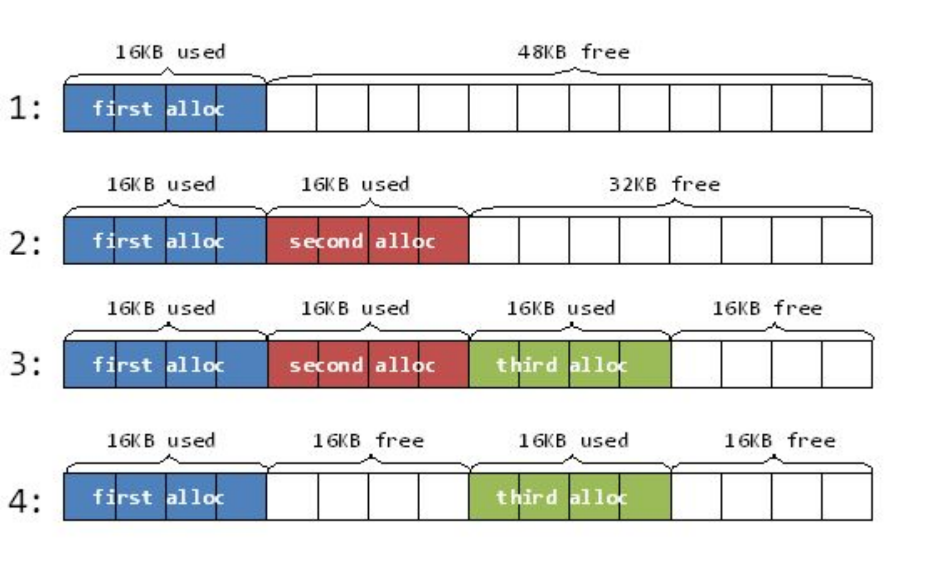
These operations correspond to *malloc* and *free*

```
int * arr = (int *) malloc(10 * sizeof(int));
int * arr = new int[10];
```

```
free(arr);
```

designing a good heap is tricky

- Don't know the memory sizes program needs ahead of time
  - Need to support both small and large chunks
- Need to quickly find free memory
- Should optimally lay out memory - avoiding "fragmentation"



there is no perfect heap

Several different ways of doing it, with different tradeoffs

Heap exploitation is taking advantage of how a heap implementation works under-the-hood.

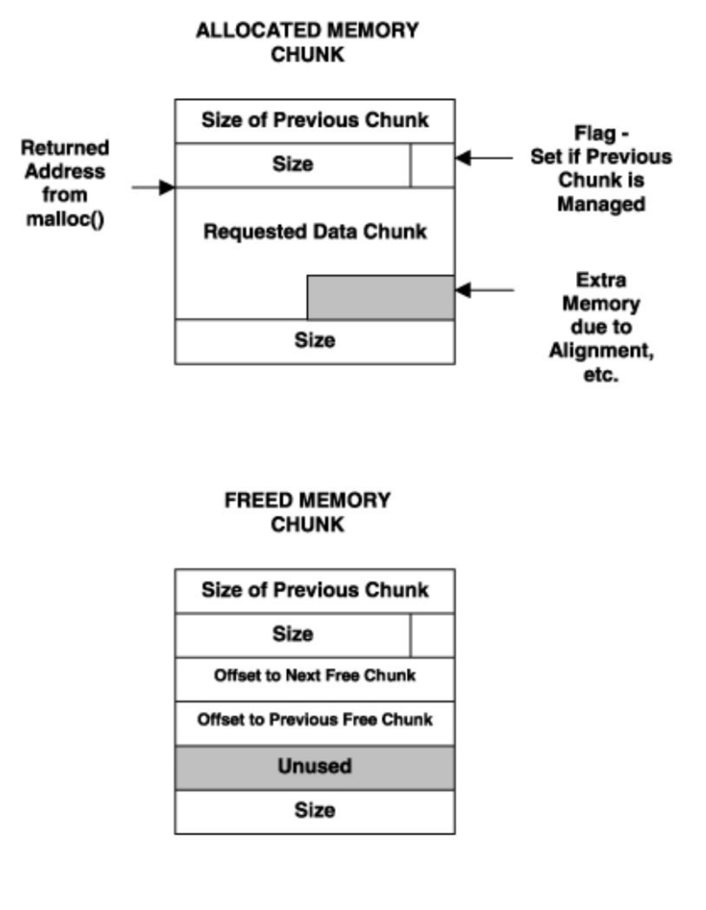
We will focus on the heap implementation in glibc

basic heap implementation

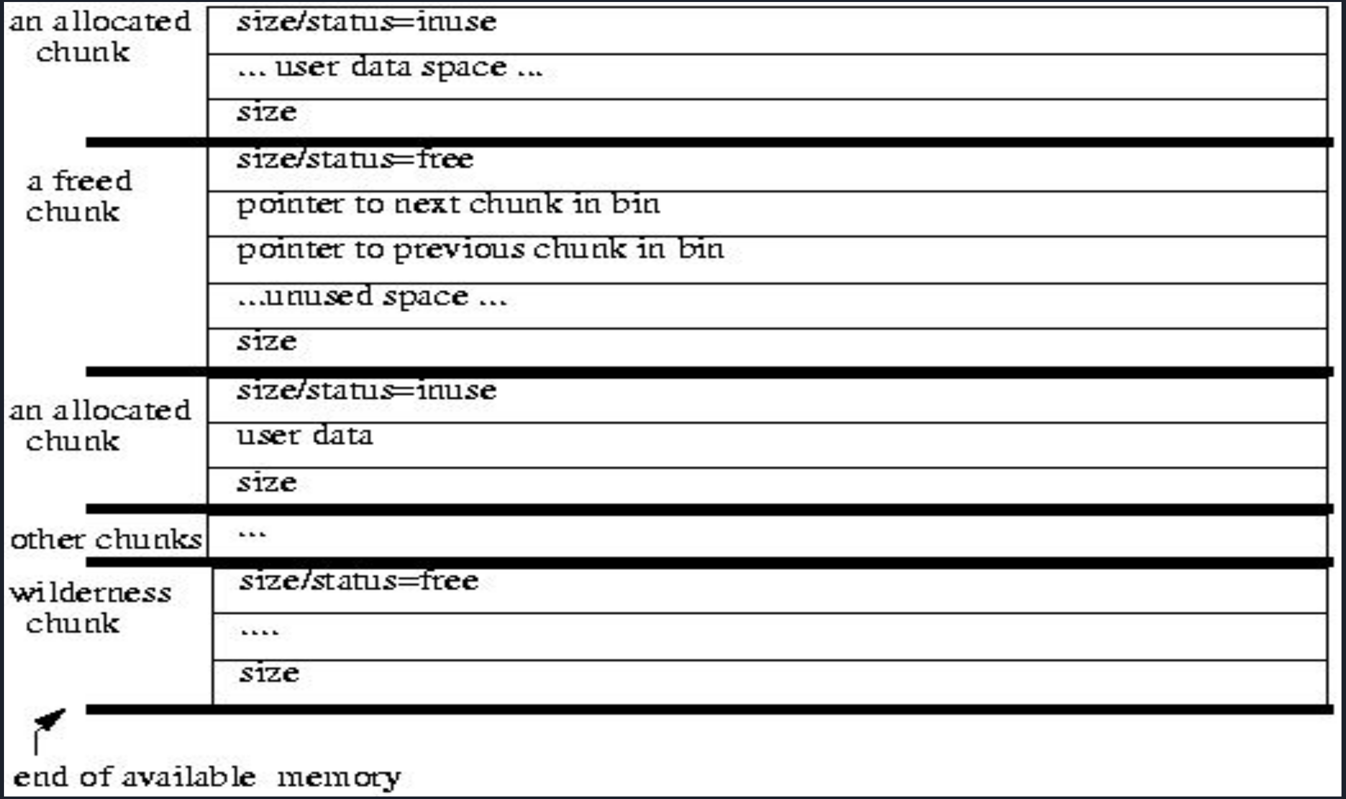
- By default, all of memory is in a "wilderness" chunk
    - New memory is taken from this wilderness chunk
- The heap also maintains a list of "free" chunks
    - Chunks that have been allocated but subsequently freed
- metadata is stored next to each chunk to help with this

how malloc works

- when we call malloc, metadata is stored nearby in headers and footers
  - int indicating size of chunk
  - flag indicating if chunk is free or in use
- extra memory padded to align the chunk, and leave room for free prev/next pointers (more on this later)


ALLOCATED MEMORY CHUNK
Size of Previous Chunk
Size
Requested Data Chunk
Size
Returned Address from malloc()
Flag - Set if Previous Chunk is Managed
Extra Memory due to Alignment, etc.
FREED MEMORY CHUNK
Size of Previous Chunk
Size
Offset to Next Free Chunk
Offset to Previous Free Chunk
Unused
Size

example chunk

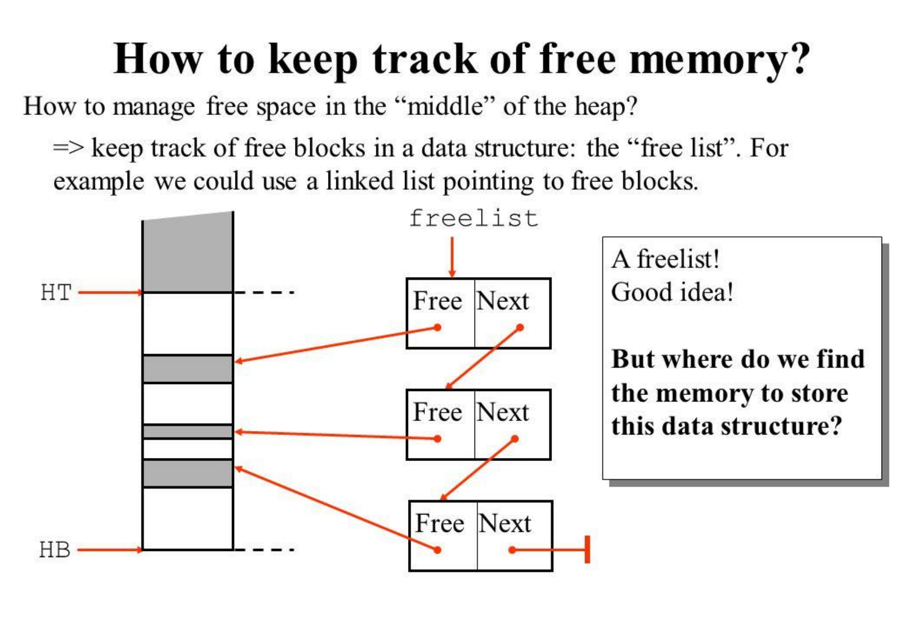


an allocated chunk	size/status=inuse
	... user data space ...
	size
a freed chunk	size/status=free
	pointer to next chunk in bin
	pointer to previous chunk in bin
	...unused space ...
	size
an allocated chunk	size/status=inuse
	user data
	size
other chunks	...
wilderness chunk	size/status=free
	....
	size

end of available memory

how free chunks/bins work

- when a chunk is freed, it is placed in a linked list of freed nodes
- naive implementation: make a giant linked list of freed nodes
    - this is slow
- solution: different linked lists for chunks of various sizes



How to keep track of free memory?

How to manage free space in the "middle" of the heap?

=> keep track of free blocks in a data structure: the "free list". For example we could use a linked list pointing to free blocks.

A freelist!
Good idea!

But where do we find the memory to store this data structure?

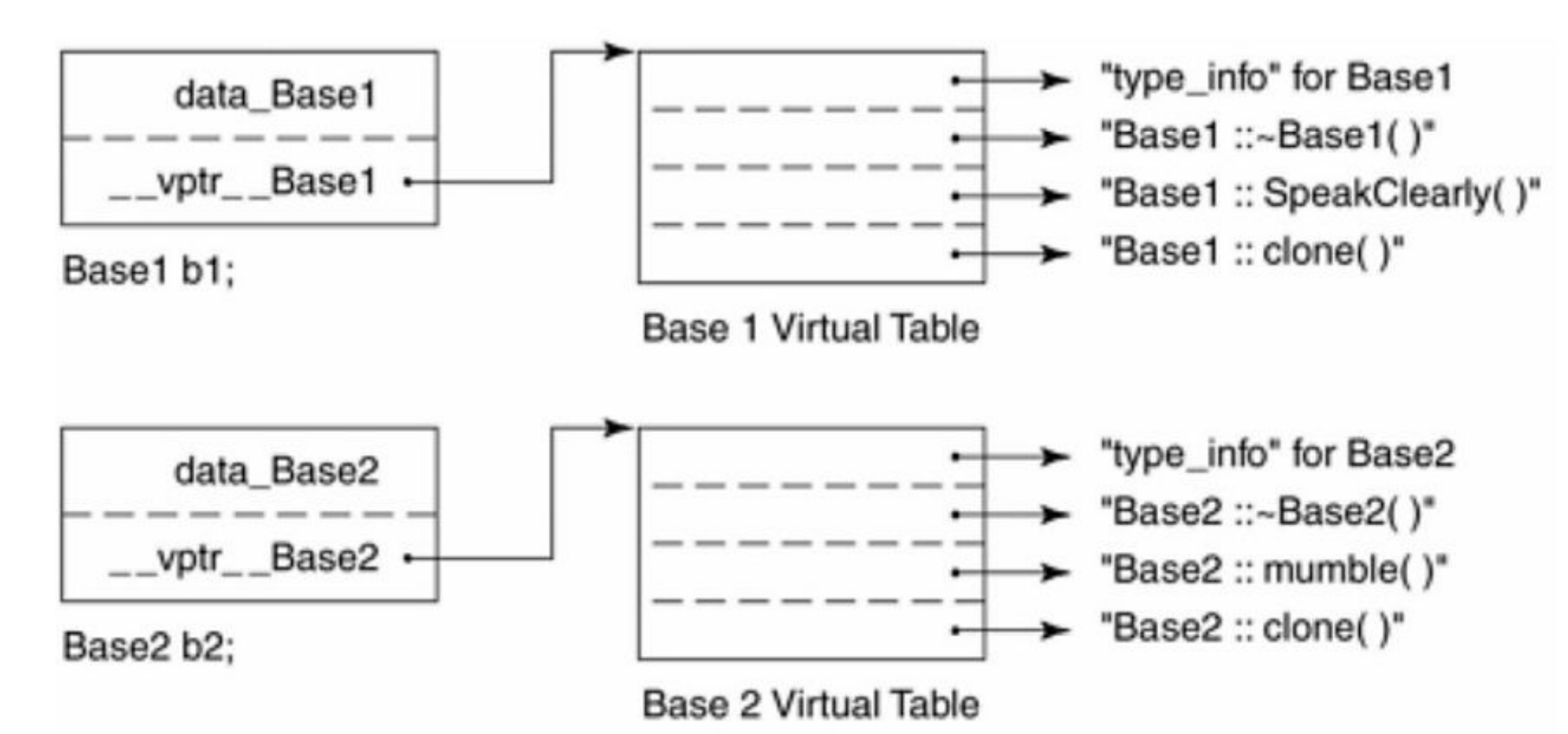
Types of Bins

- Fast Bins: small boi, no merging, limited space
- Small Bins: Double linked list, merging
- Unsorted Bins: chunks that fit in the small bin go here
- Large Bins: big boi, sorted order, chunks within certain range, merging
- Tcache: chunks cached by current thread

Heap Overflows

- Need to account for the structure of the heap chunks
- Common targets: function pointers, c++ vtables, etc
- Forge new chunks or corrupt adjacent chunk headers in order to fuck with the heap’s free() logic

C++ Vtables



data_Base1

__vptr__Base1

Base1 b1;

Base 1 Virtual Table

"type_info" for Base1

"Base1 ::~Base1( )"

"Base1 :: SpeakClearly( )"

"Base1 :: clone( )"

data_Base2

__vptr__Base2

Base2 b2;

Base 2 Virtual Table

"type_info" for Base2

"Base2 ::~Base2( )"

"Base2 :: mumble( )"

"Base2 :: clone( )"

How to exploit?

- Overwrite the vtable pointer in our class to point to a fake vtable that we control
- Overwrite instance variables in the class

More info: http://phrack.org/issues/56/8.html

Unlink Exploit

- Takes advantage of the coalescing algorithm in free() in order to write arbitrary values into any memory
- Does this by constructing fake chunks & corrupting chunks in order to mess with the free linked list pointers
- Obsolete as long as you don’t use dlmalloc. Google Project Zero was able to get this exploit to work under certain circumstances: https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html

Double Free

- Same node added to a bin twice.
- Can be used to write values into arbitrary memory spaces

Example:

free(a);   Bin -> [a] -> [b] -> [a]

free(b);

free(c);

Use-after-free

- Vulnerability where a freed chunk is still in use
- Similar to double free where one can call malloc to modify the contents of the chunk
- Can be used to leak information(defeat ASLR) or get arbitrary code execution
- Riskier in browsers/scripting engines(since you can reliably allocate memory on the heap) or C++ (because vtables)

Null byte poisoning

- Triggered by an off-by-one overwrite that involves zeroing out the PREV_IN_USE flag in the next chunk
- Will coalesce a non free chunk, which can result in a UAF (due to overlapping chunks)

House of Force

- requires overwrite and control over allocation size
- overwrite top chunk size chunk
- make next allocation align with libc hooks

What to do after you get a uaf?

- overwrite the fd pointer to point to a chunk of memory to get a (nearly) arbitrary write
- leak heap and libc addresses to bypass ASLR
- unsorted bin attack(write unsorted bin linked list address anywhere)

Challenge!

shellcode.codes