Attacking the WebKit Heap

[Or how to write Safari exploits]

Agustin Gianni + Sean Heelan
Immunity Inc.

Introduction

- The use of Webkit has been increasing steadily
  - According to Wikipedia[1], “WebKit powers Google Chrome and Apple's Safari, which in January 2011 had around 13% and 6% of browser market share respectively.”
- Webkit Heap is based on TCMalloc
  - Source is available
- A custom heap allocator eases the development of cross platform exploits.
  - They use the same one for every architecture/os
  - A single exploit to rule them all.

[1] http://en.wikipedia.org/wiki/WebKit

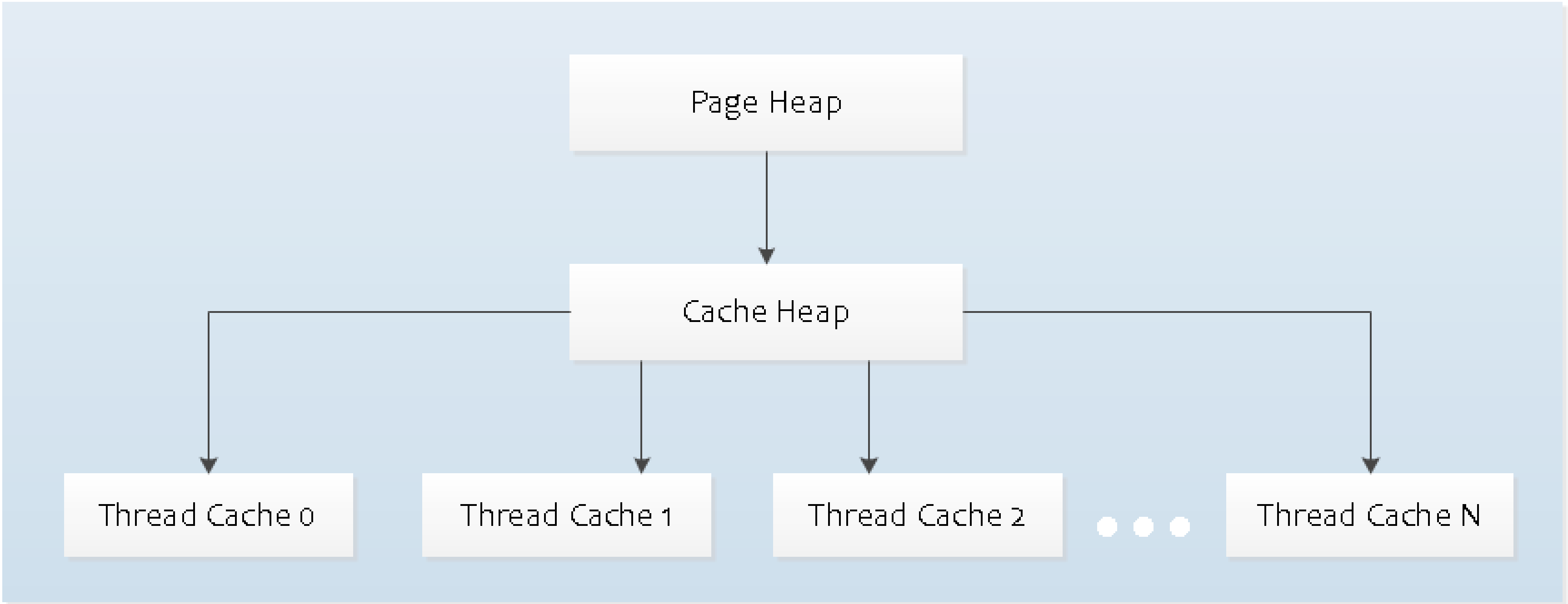
Motivation

- Huge surface of attack
  - Chrome
  - Safari
  - Android
  - Kindle
  - BlackBerry
- Security teletubbies use MacOSX
  - Finally we can pwn Charlie Miller.

Basics

- Custom allocator designed with speed in mind
  - Generally, speed = less “security checks”
- Thread oriented design
  - Each thread gets a thread-local cache.
  - Each thread manages its own allocations.
- TCMalloc consists of three different allocators
  - Hierarchically arranged
    - Higher up allocators serve the lower allocators with memory

Allocator Hierarchy



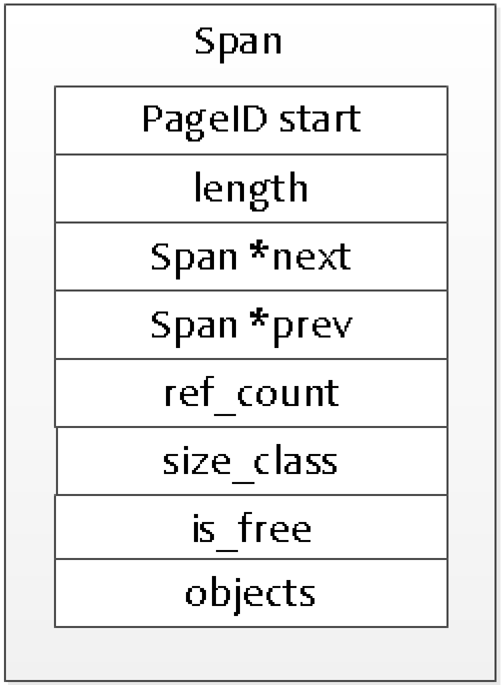
The PageHeap is closest to the system (highest), the ThreadCache is closest to the application (lowest)

Allocation Sizes

- Chunk sizes are divided into kNumClasses
  - WebKit has 68 Size Classes
- Allocations are rounded to the nearest size class
- Small chunks < 8 * PAGE_SIZE
  - Allocated from the ThreadCache
- Large chunks > 8 * PAGE_SIZE
  - Allocated from the PageHeap

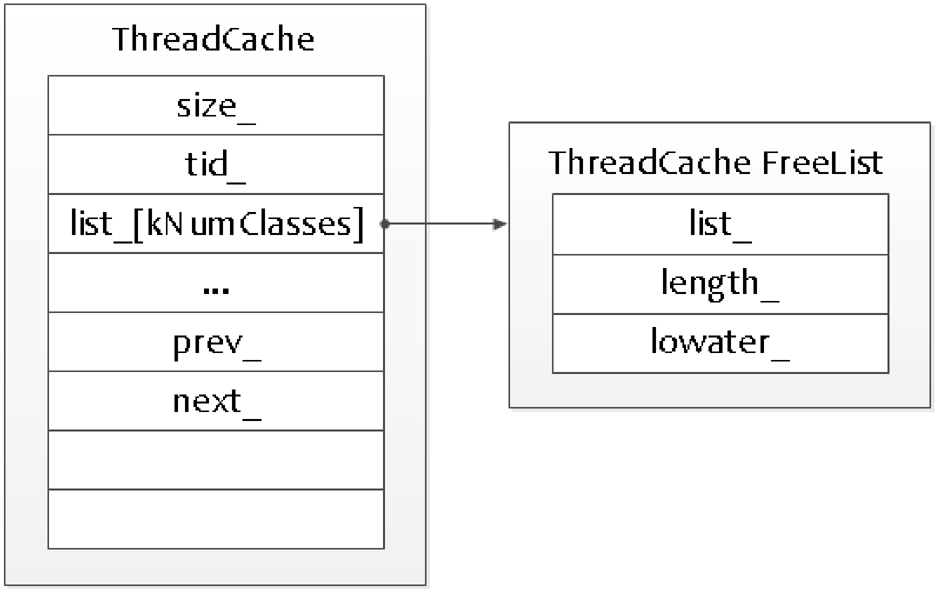
Spans

- Memory managed by TCMalloc is backed by a Span structure
- Sets of contiguous pages
- Can contain:
    - Set of Small Chunks
    - Large Chunk
- Span metadata is stored in the Span header which is allocated independently of the Span data



ThreadCache

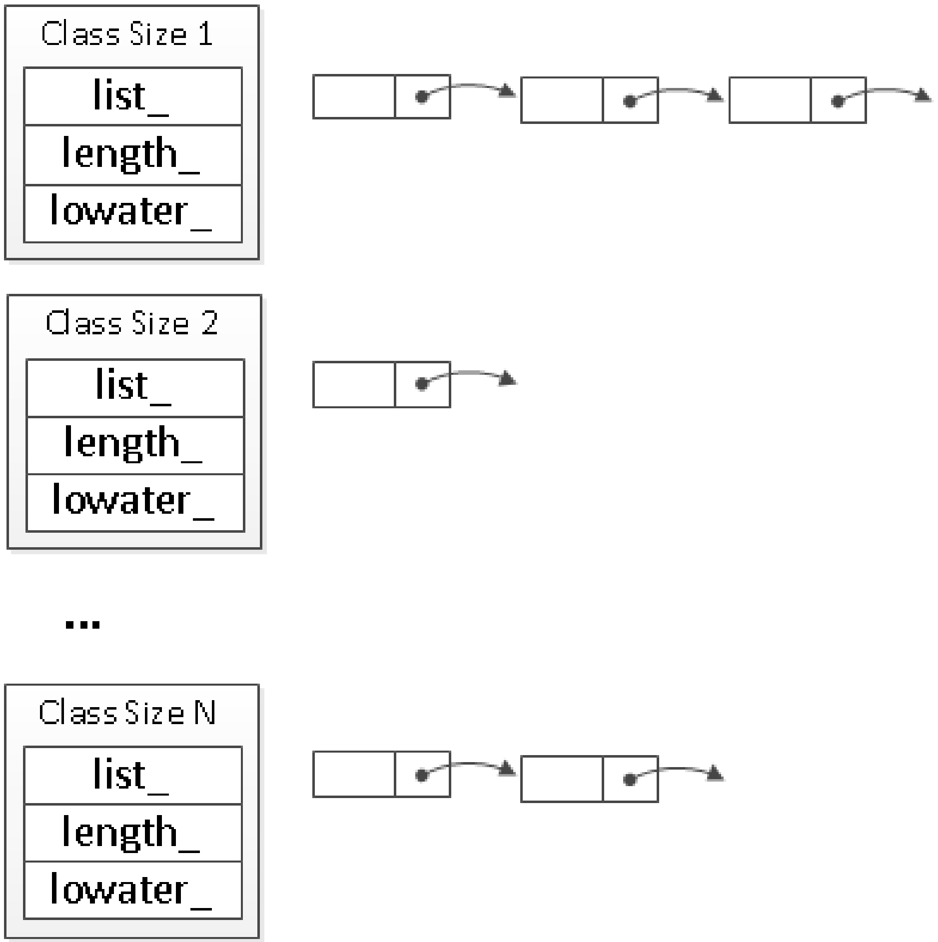
- Front-end allocator for each thread
- Contains an array of 68* free-lists
- Max. elements per free-list is 256*
- Allocates/Deallocates its memory from/to the CentralCache



* May differ between applications

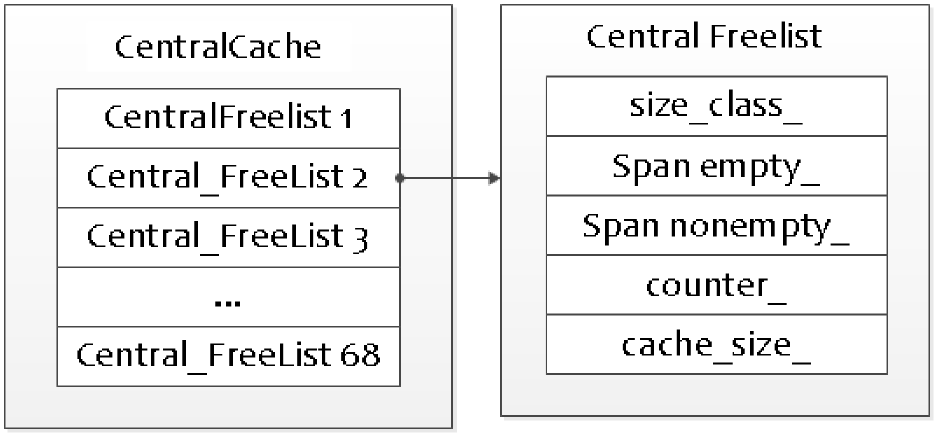
ThreadCache Freelist

- Each size class has its own ThreadCache FreeList
- The *list_* attribute points to a singly linked list of free chunks

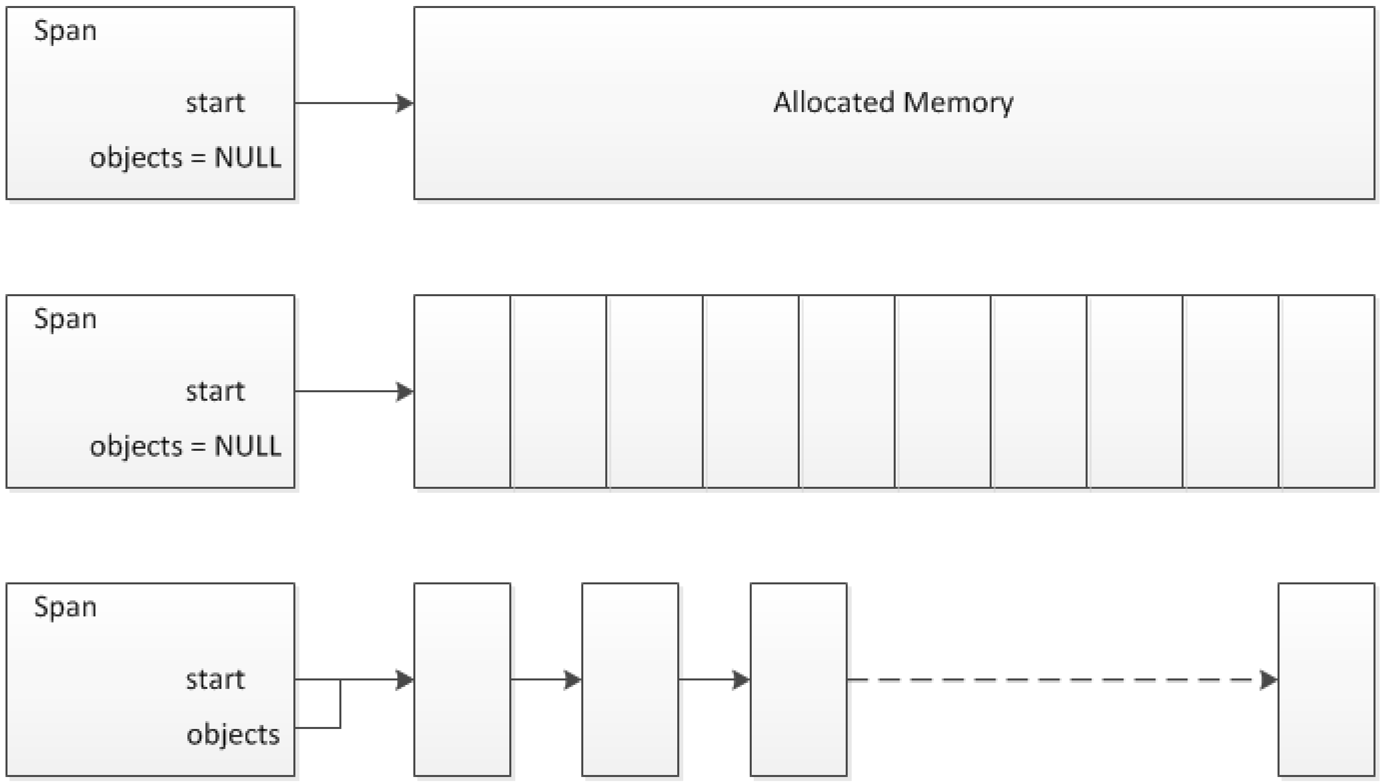


Central-Cache

- Is an array of CentralFreeList's
  - One per size class
- Obtains/Frees its memory from/to the PageHeap
  - Splits a Span into smaller chunks
- Provides chunks to Thread-Caches
  - Populates a given freelist
- Shared by all threads
  - Locking is required



Small Chunk Span Creation



- The Span objects list is used as the backend for FreeList creation for the CentralCache

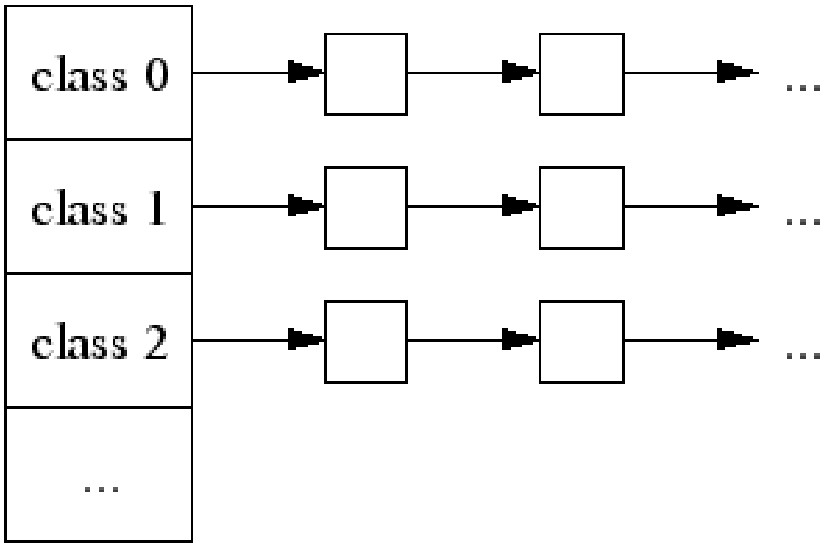
PageHeap

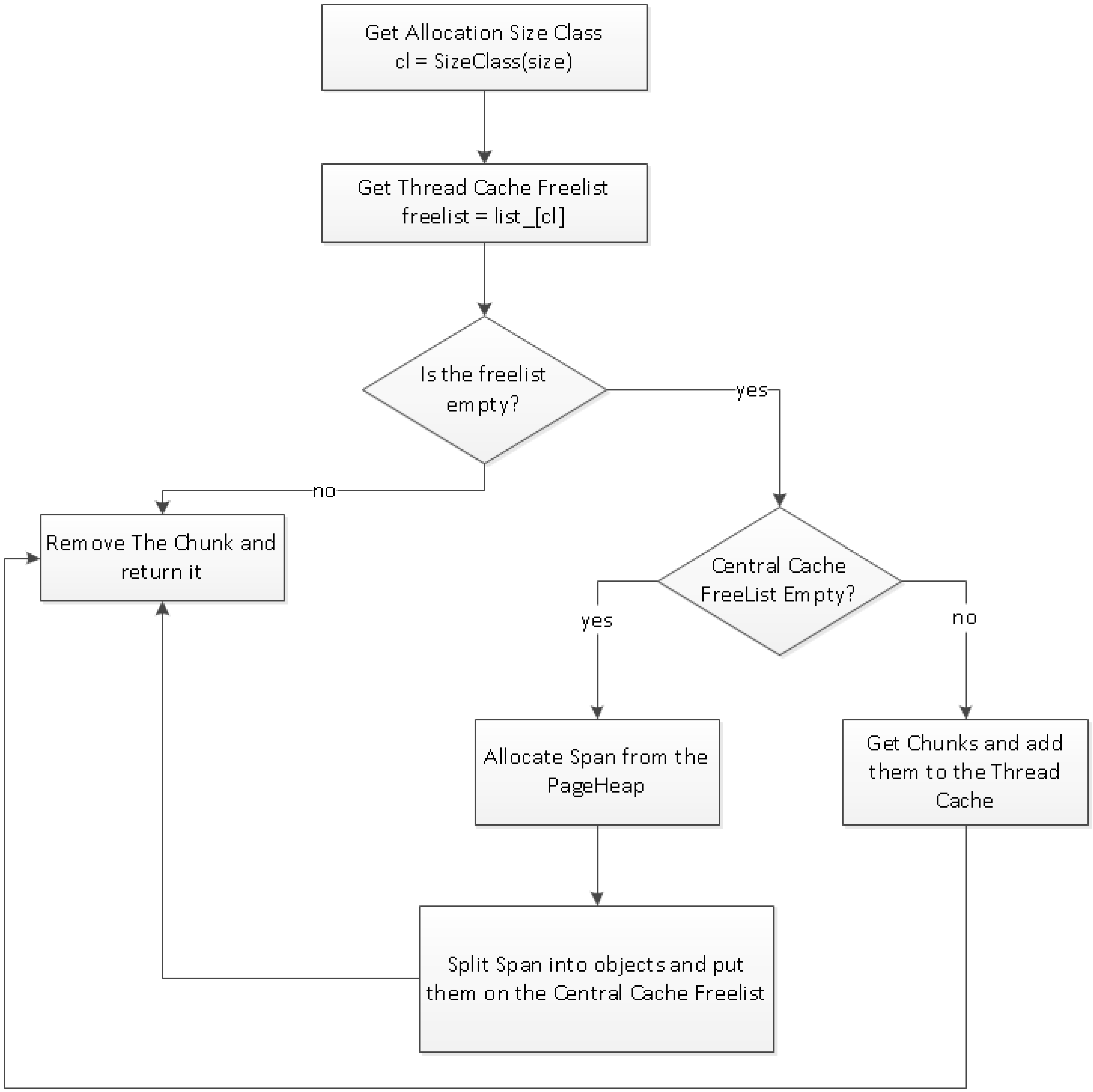
- Manages chunks of memory allocated from the system allocators
- Populated on the first call to PageHeap::New (and as required after this)
  - This will trigger an allocation from the System allocator e.g. VirtualAlloc, mmap, sbrk
- Contains two free lists
  - SpanList large
    - For chunks bigger than kMaxPages (256 in WebKit)
  - SpanList free_[kMaxPages]
    - First entry is 1 page, second entry 2 pages, and so on

Small Chunk Allocation

Small Chunk Allocation

- 68 different Size Classes *
- Sizes are rounded to the next size class
  - 8 bytes is the minimum chunk size
  - Allocations of size 0 are valid and rounded up to 8
- Chunks will be obtained from one of the Thread-Cache free lists

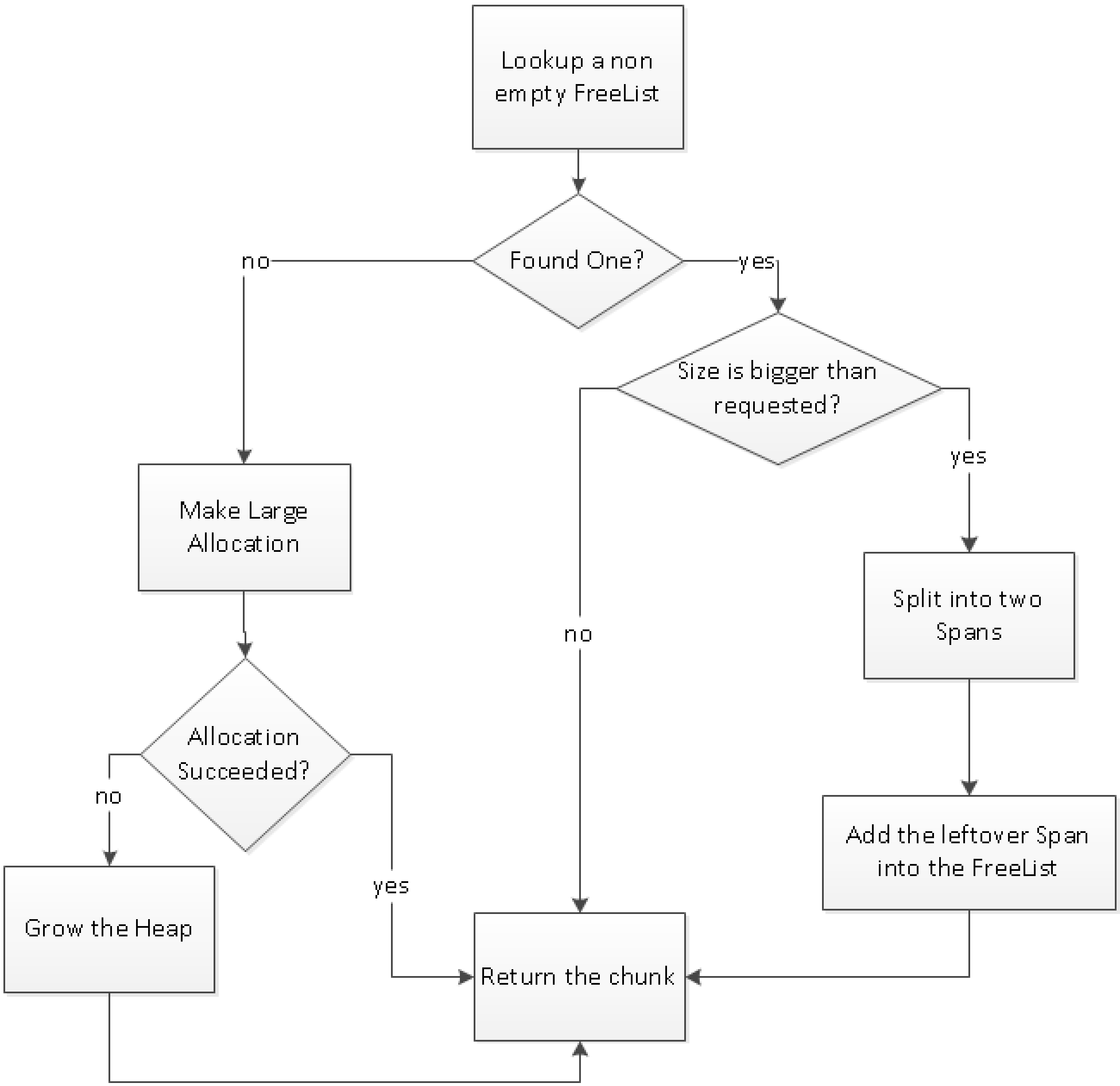



Get Allocation Size Class
cl = SizeClass(size)
Get Thread Cache Freelist
freelist = list_[cl]
Is the freelist empty?
yes
no
Remove The Chunk and return it
Central Cache FreeList Empty?
yes
no
Allocate Span from the PageHeap
Get Chunks and add them to the Thread Cache
Split Span into objects and put them on the Central Cache Freelist

Large Chunk Allocation

Large Chunk Allocation

- Handled by the PageHeap
- Allocations of more than 8*PAGE_SIZE are considered large allocations
- Page aligned
- Can trigger heap growth

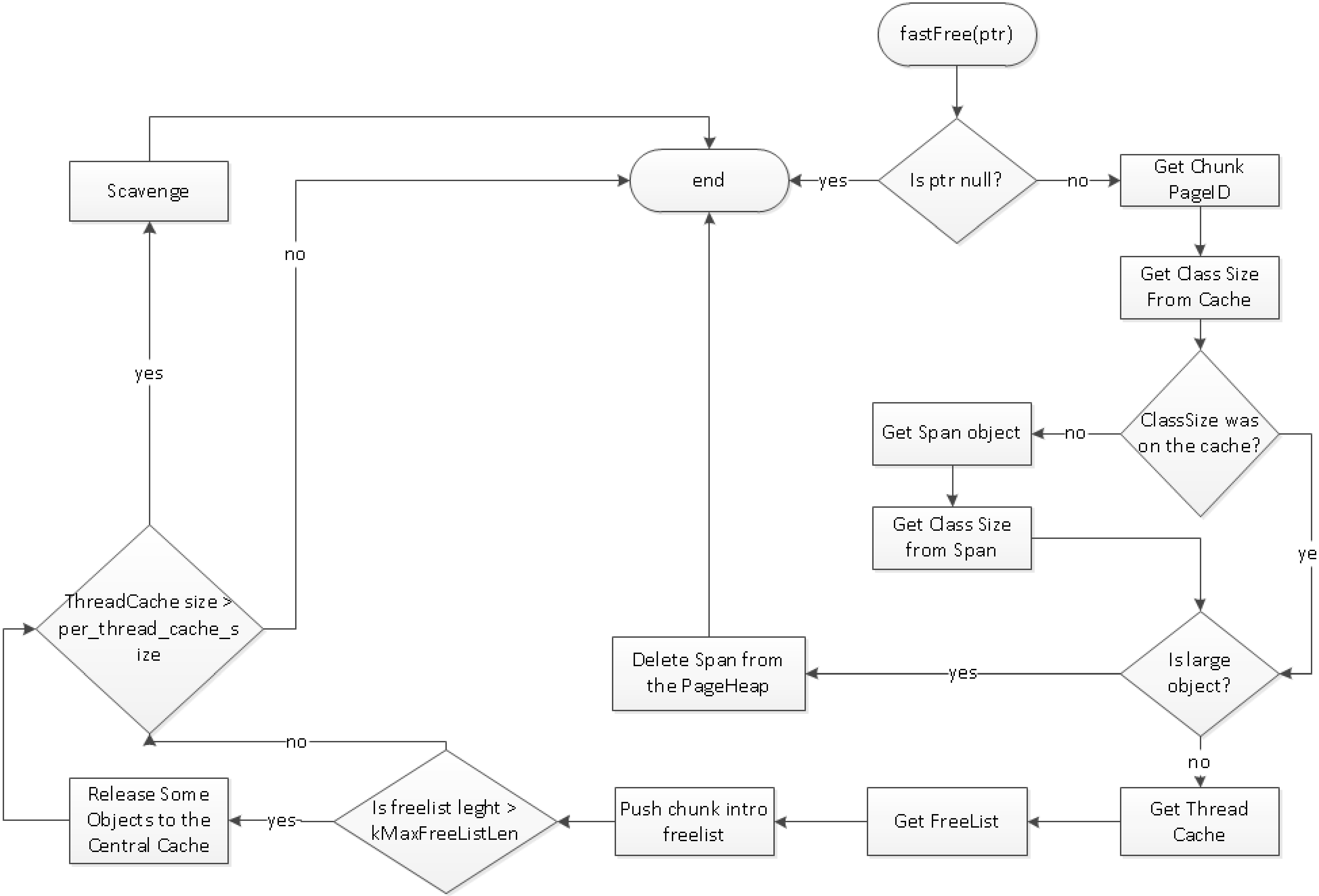


Lookup a non empty FreeList
Found One?
no
yes
Size is bigger than requested?
no
yes
Make Large Allocation
Split into two Spans
Allocation Succeeded?
no
yes
Add the leftover Span into the FreeList
Grow the Heap
Return the chunk

Chunk Deallocation

- Small chunks go directly to the *ThreadCache* free list.
  - If the free list size exceeds *'kMaxFreeListLength' (256)* some of the chunks are moved to the *central-cache* free list.
  - If the combined size of the chunks exceeds *'kMaxThreadCacheSize'* (2<<20) → GC
- Large chunks are inserted on the *PageHeap*
  - Coalescing is triggered if neighboring chunks are also free.

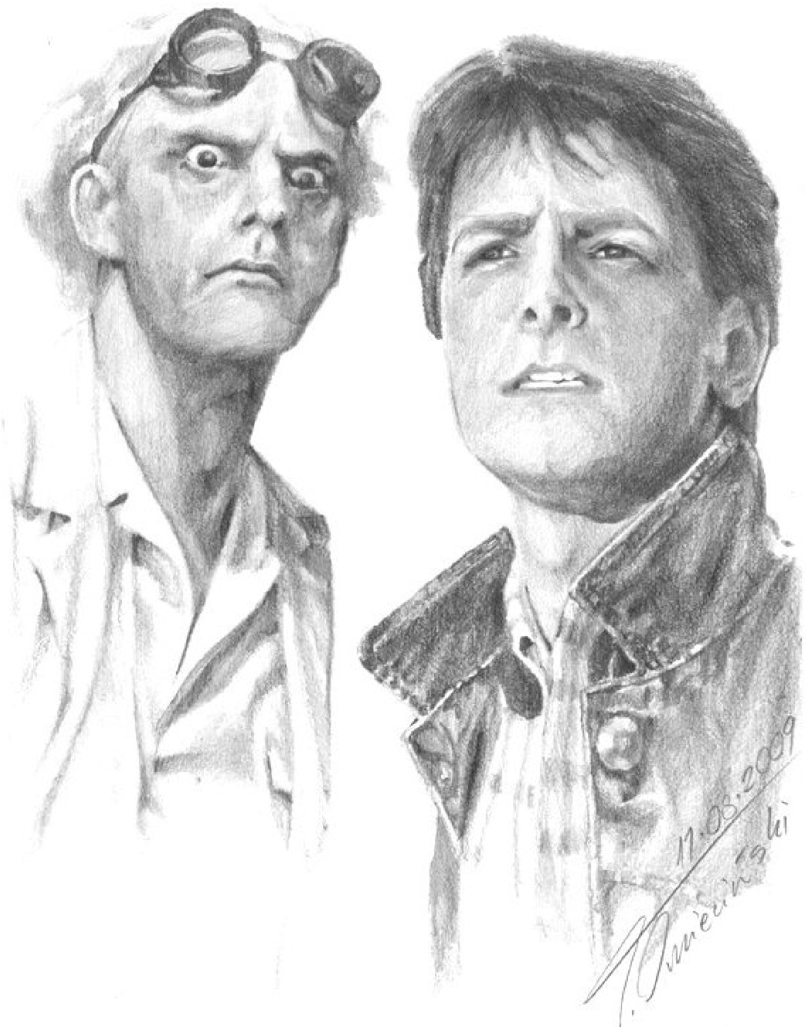
Object Deallocation Flow Graph!



End of Introduction

Exploiting Memory Corruption Bugs in TCMalloc

Security Mechanisms



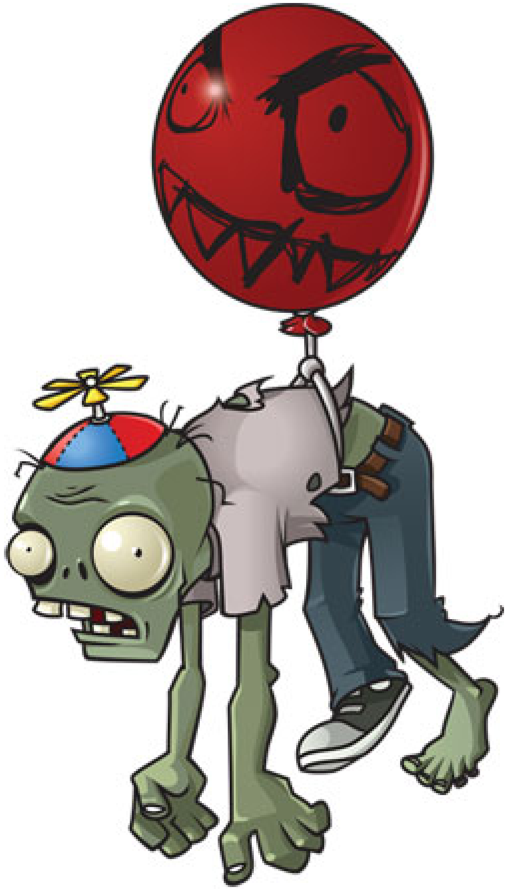
- Back to 1995
- All in all, there are no protections
- Lulz level increased

  comex: “The best way to avoid fighting with the heap is to find vulnerabilities that aren't heap overflows”

- Vendors must have felt sorry for us =D

Resurrecting the Dead

- Freeing Invalid Pointers
- ThreadCache FreeList Overflow
  - Insert to FreeList[X]
- Span Objects List Overflow
  - Insert to Span Objects list
- Double Free
- Span Metadata Overflow
  - Unlink
- Strawberry Pudding



Freeing Invalid Pointers

Freeing Invalid Pointers: The Bad

- If the pointer has a PageID (`ptr >> kPageShift`) that has not previously recorded as a Span then Bad Things may occur
  - There is a PageMap object that maps from pointers to the correct Span containing information about the page they are within
  - At the start of `do_free` TCMalloc attempts to retrieve the Span for the pointer using `GetDescriptor(ptr >> kPageShift)`

Freeing Invalid Pointers: The Bad

```
void* get(Number k) const {
    ASSERT(k >> BITS == 0);
    const Number i1 = k >> LEAF_BITS;
    const Number i2 = k & (LEAF_LENGTH-1);
    return root_[i1]->values[i2];
  }
```

- `root` is an array of 32 pointers, initialized to 0
- `values` is an array of 32768 pointers, initialized to 0
- So `ptr >> kPageShift` must therefore have been inserted into the PageMap at some point or ...

Freeing Invalid Pointers: The Bad

- .... Bad Things
  - TCMalloc in WebKit will segfault on the NULL ptr dereference if `root[i1]` has not been alloc'd
  - TCMalloc in Chrome detects the above condition and returns NULL
  - In both, the `values` array is initialized to 0 so `root_[i1]->values[i2]` will return NULL if it has never been set previously
  - Chrome again detects the NULL return value and will raise a SIGABRT or similar
  - WebKit again will kamikaze on a NULL ptr soon after

Freeing Invalid Pointers: The Bad

- Can be an inconvenience for other techniques
  - Prevents us from free'ing Span header objects as they are allocated from a separate pool of memory
  - We have to be careful not to trigger free calls on pointers we insert into free lists after they are handed to the application if they are not in a valid Span
- As a side note, the above two level array is used for 32-bit Linux/OS X, 32-bit Windows uses a flat array and 64-bit * uses a radix tree
  - For our purposes the result is effectively the same

Freeing Invalid Pointers: The Good

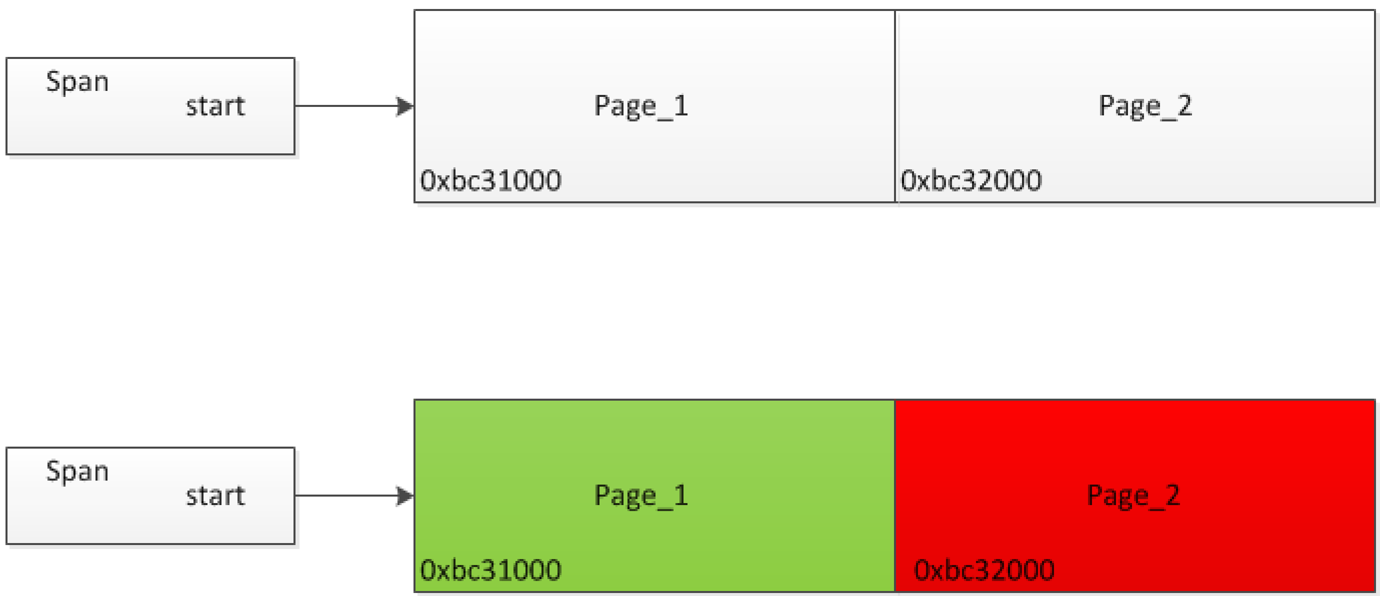
- We can however free any pointer that maps back to a valid Span
  - Free'ing large objects (> kMaxSize [32768])
    - Free of any address within the first page of the object free's the object
    - The other pages are not linked to any Span
  - Free'ing small objects
    - Any address that falls within a span recorded as containing small objects can be free'd
    - If the Span contains multiple pages, each page is linked back to the correct Span header in the PageMap
    - The pointer will be added to the free list for size class of the Span it falls within

Freeing Invalid Pointers: Large Objects

PageMap	
0xbc31	SPAN_0xbc31

Freeing Invalid Pointers: Large Objects

PageMap	
0xbc31	SPAN_0xbc31

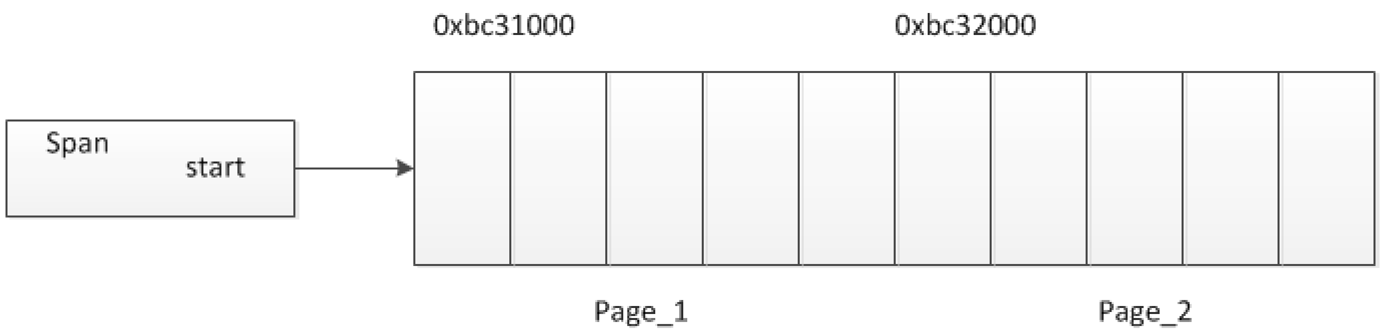


free(0xbc31ffff) :)

free(0xbc32000) :(

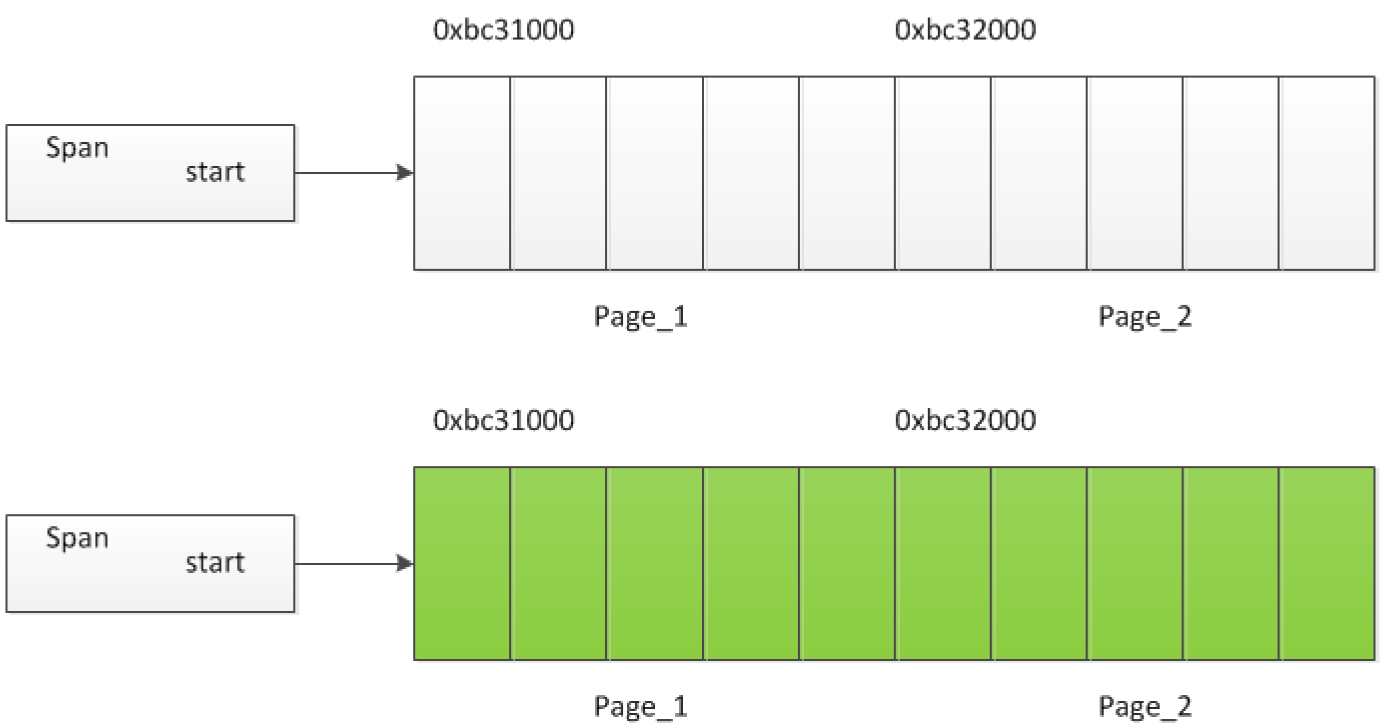
Freeing Invalid Pointers: Small Objects

PageMap	
0xbc31	SPAN_0xbc31
0xbc32	SPAN_0xbc31



Freeing Invalid Pointers: Small Objects

PageMap	
0xbc31	SPAN_0xbc31
0xbc32	SPAN_0xbc31



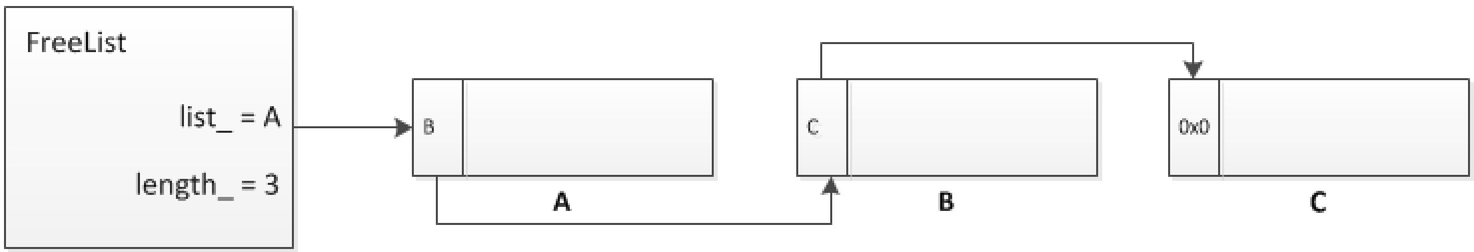
Summary

- For large objects free'ing any pointer within the first page free's the entire object
- For small objects free'ing any pointer within the Span free's that pointer
  - This pointer does *NOT* have to be correctly aligned with the small chunks in that Span
  - Therefore we can free part of an in-use chunk if we want
- Interesting vector when considering partial pointer overflows that are later free'd
- Free'ing anything else will end in Bad Things

ThreadCache FreeList Corruption

FreeList[X] Allocation

- ThreadCache FreeList[X]

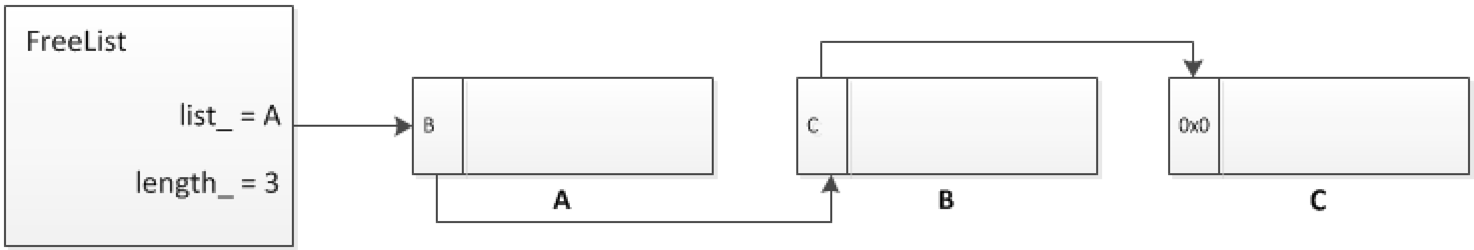


- ThreadCache::Allocate (non empty freelist)

SLL_Pop(void **list_)

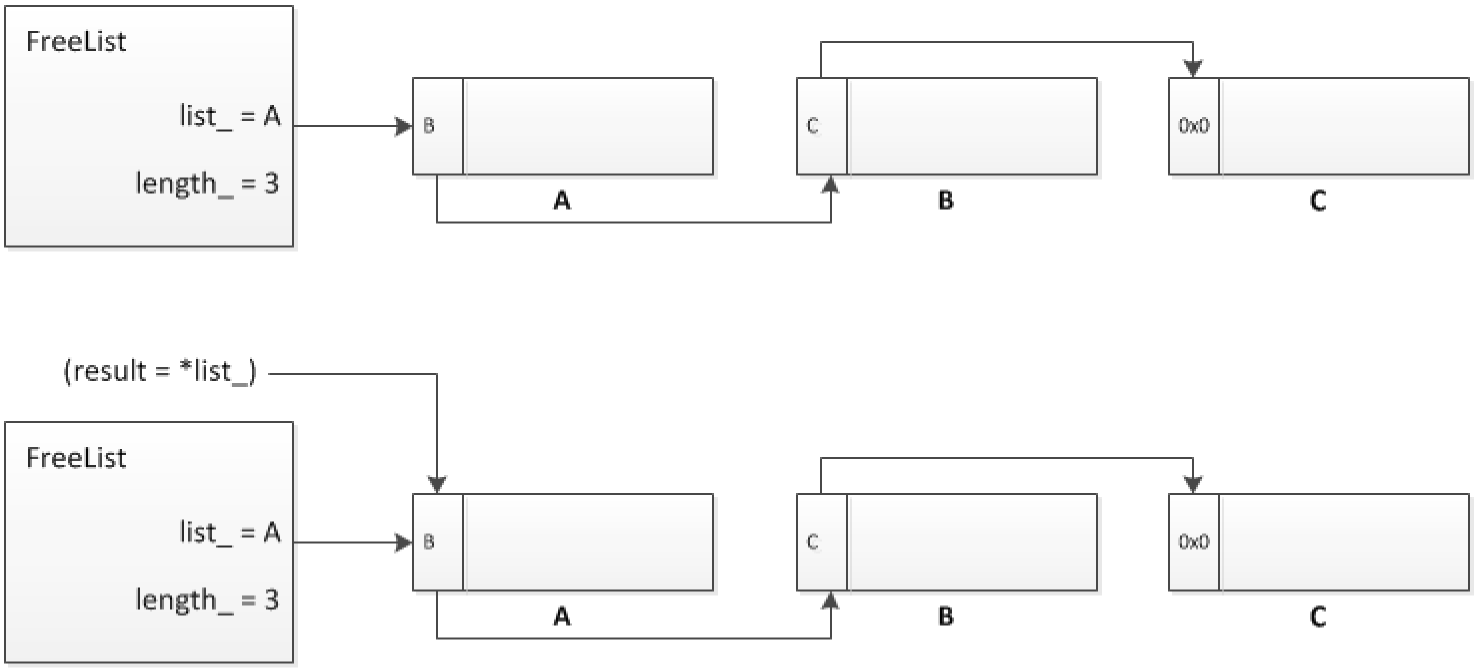
```
result = *list_;
*list_ = **list_;
return result;
```

FreeList[X] Allocation



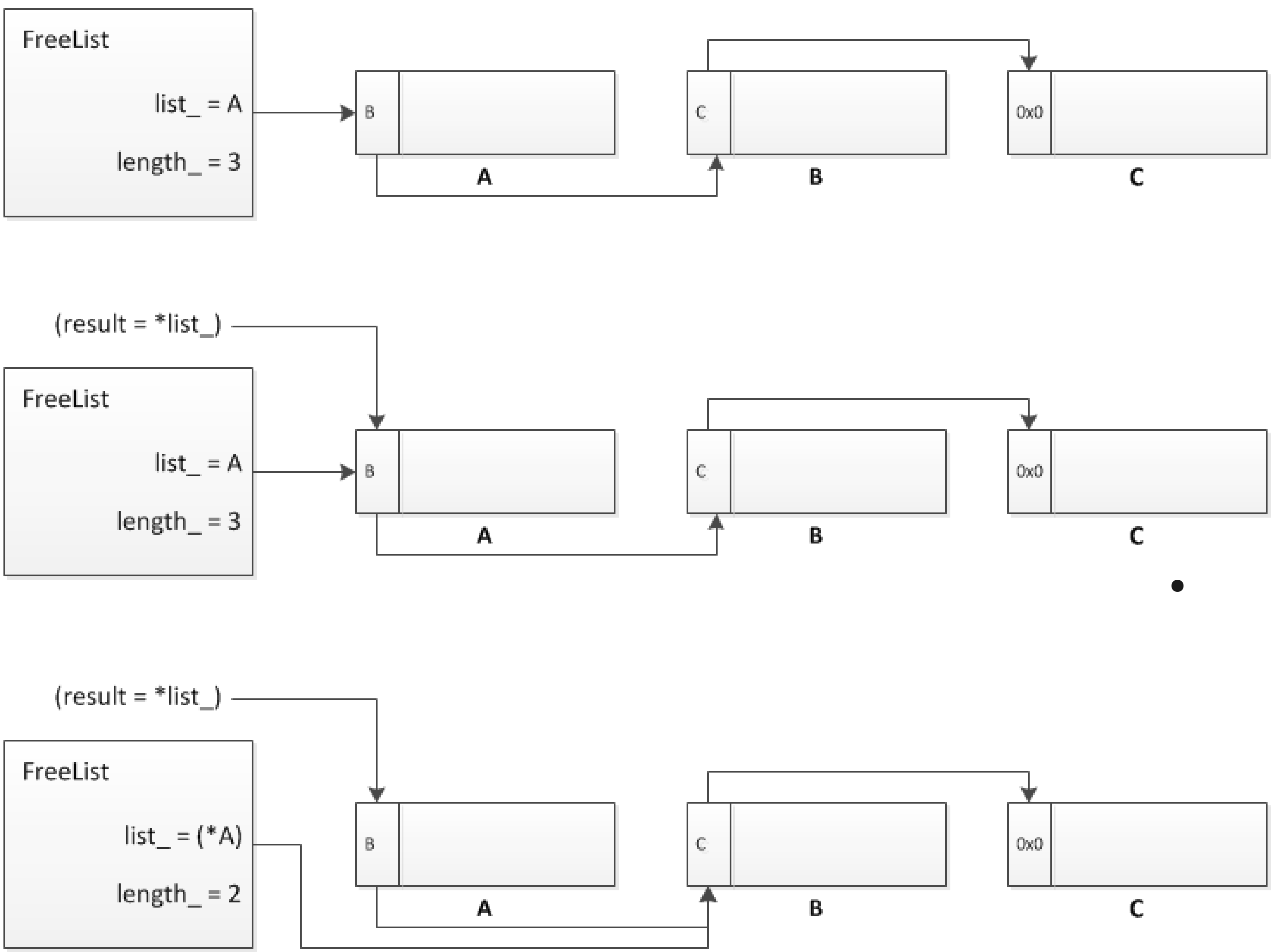
SLL_Pop(void **list_)
result = *list_;

FreeList[X] Allocation



SLL_Pop(void **list_)
result = *list_;

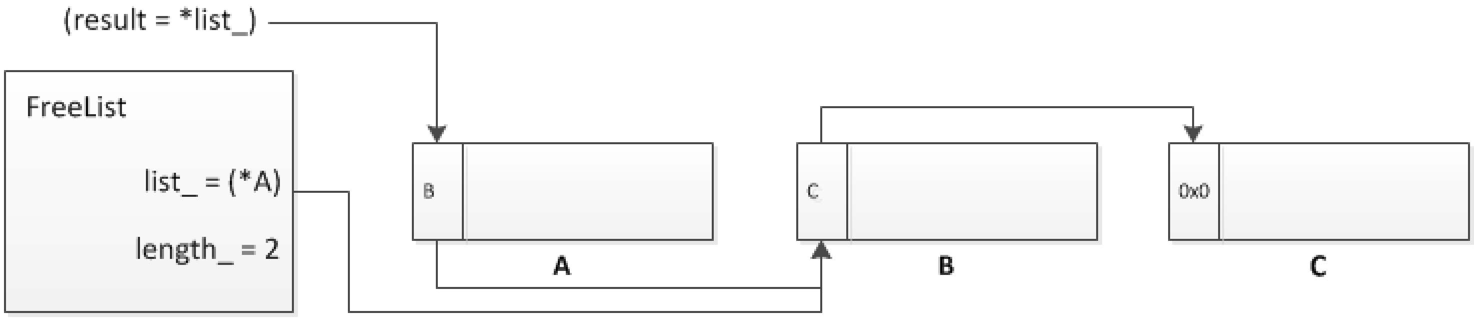
FreeList[X] Allocation



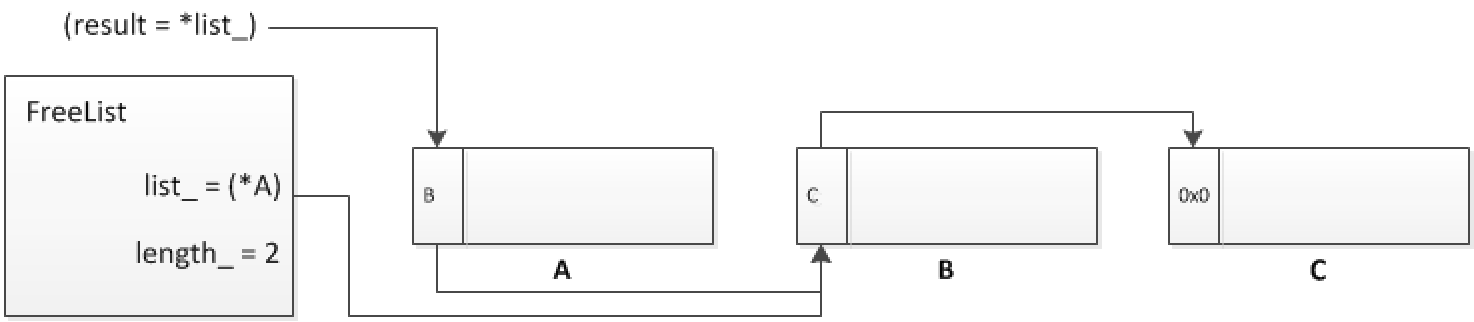
SLL_Pop(void **list_)
result = *list_;

- *list_ = **list_;
return result;

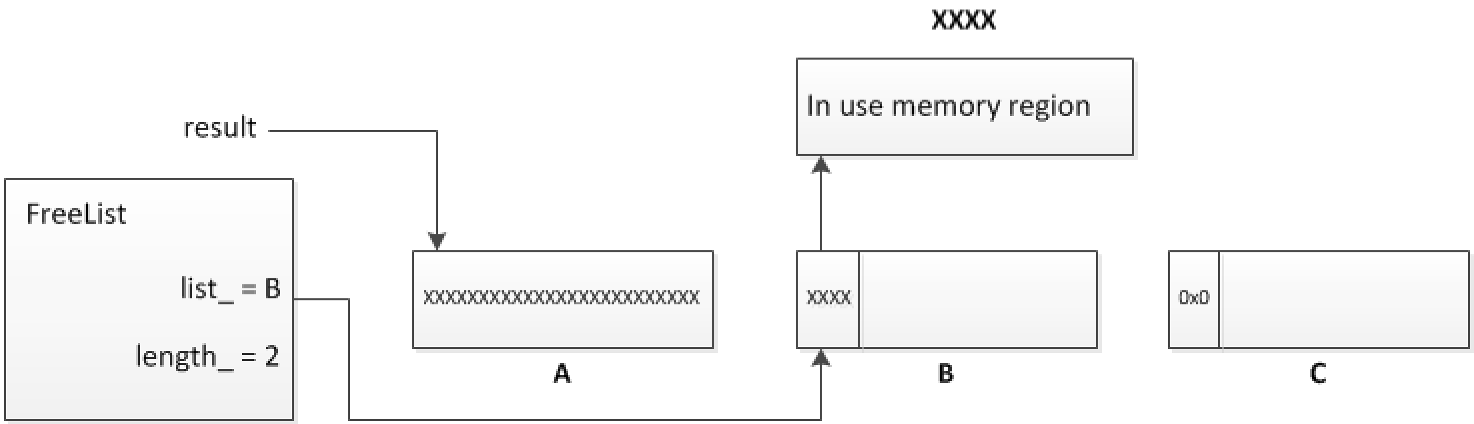
Insert to FreeList[X]



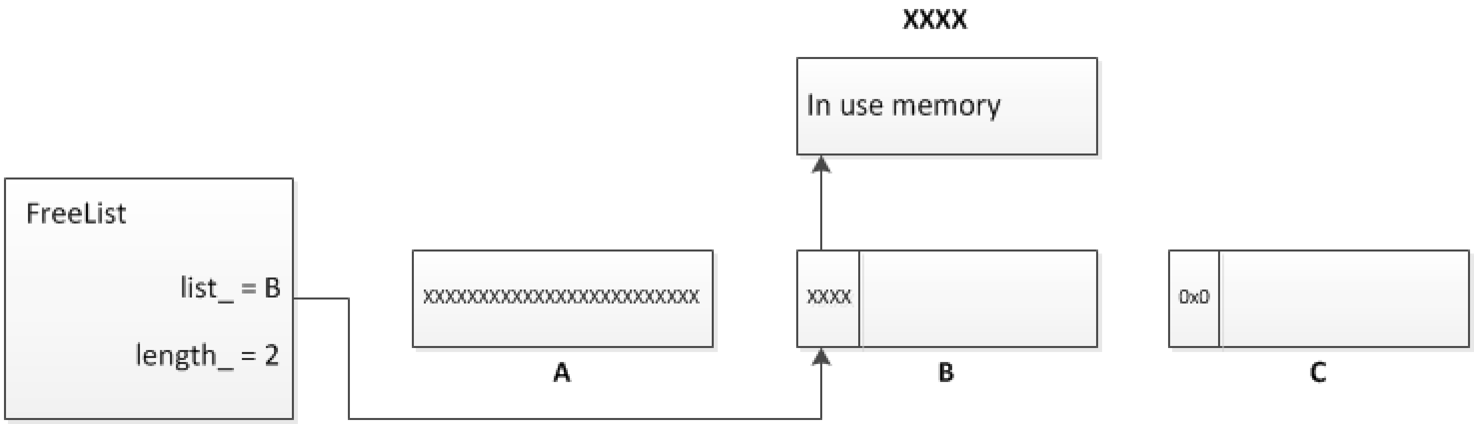
Insert to FreeList[X]



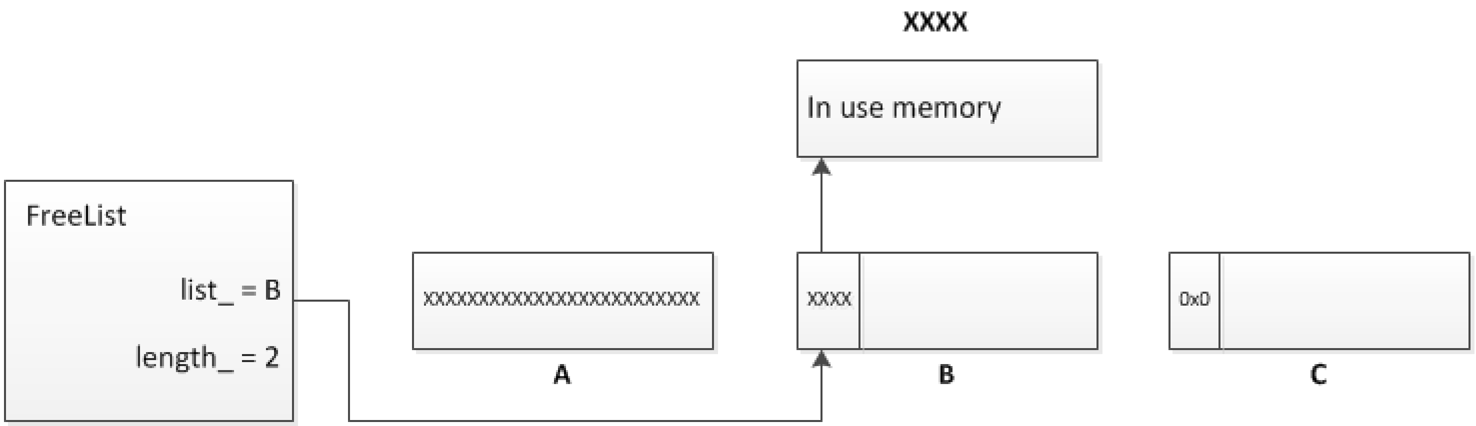
overflow_func(result)



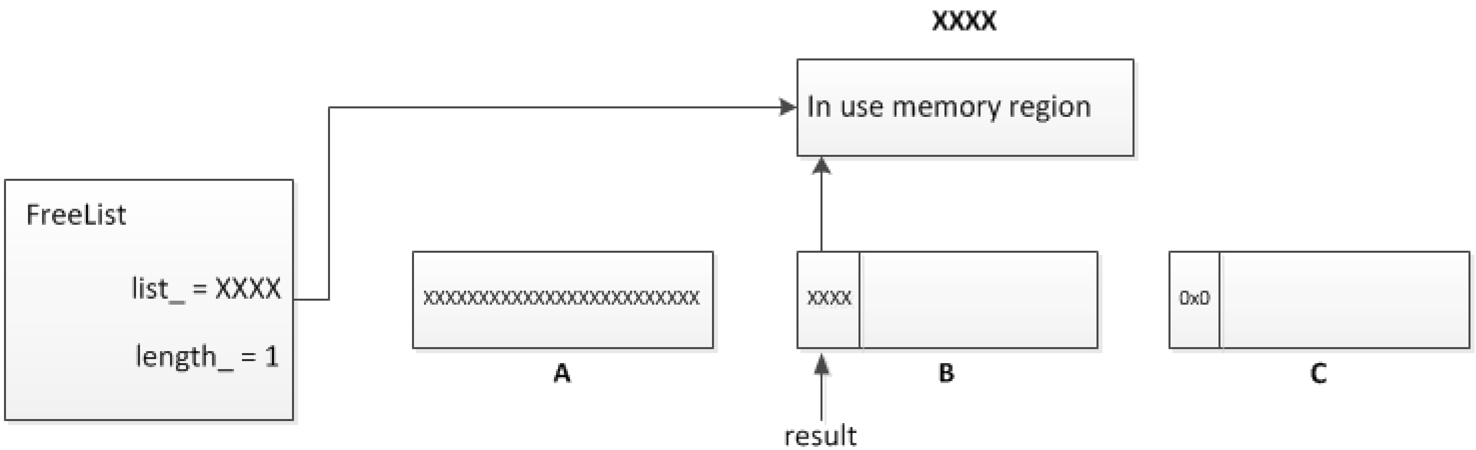
Insert to FreeList[X]



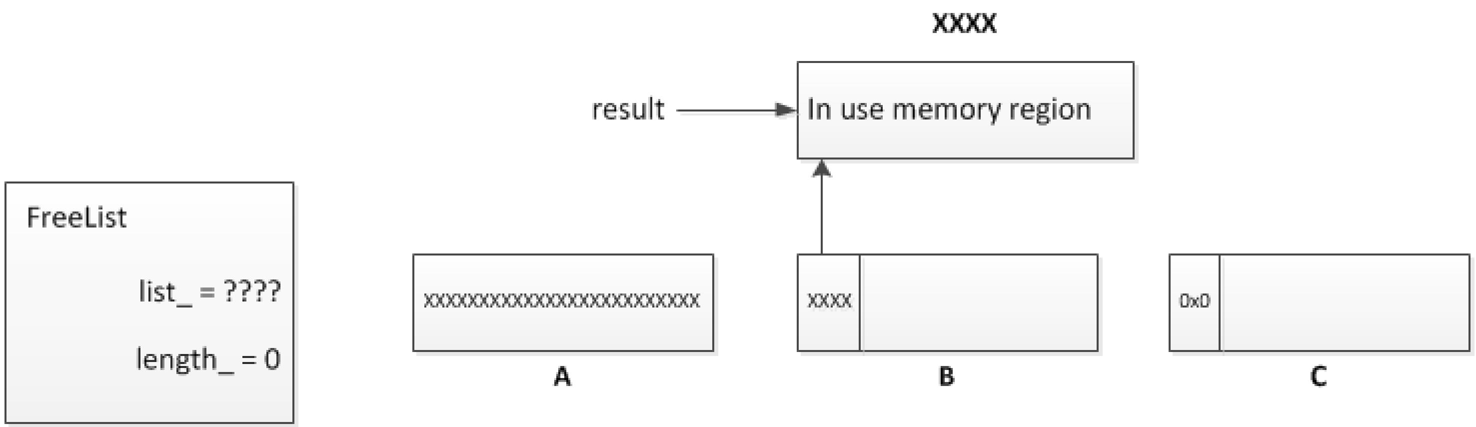
Insert to FreeList[X]



Allocate(FreeList[X])



Insert to FreeList[X]



Allocate(FreeList[X])

- This allocation returns an address we control to the application as valid heap memory
  - Similar to the *Insert to Lookaside* technique in effect
- The new list head pointer is equal to the first DWORD of this region
- Caveat: Ensuring our overflow chunk is behind a chunk in a free list may require some trickery...

FreeList Creation

- Initial FreeList creation

  FetchFromCentralCache

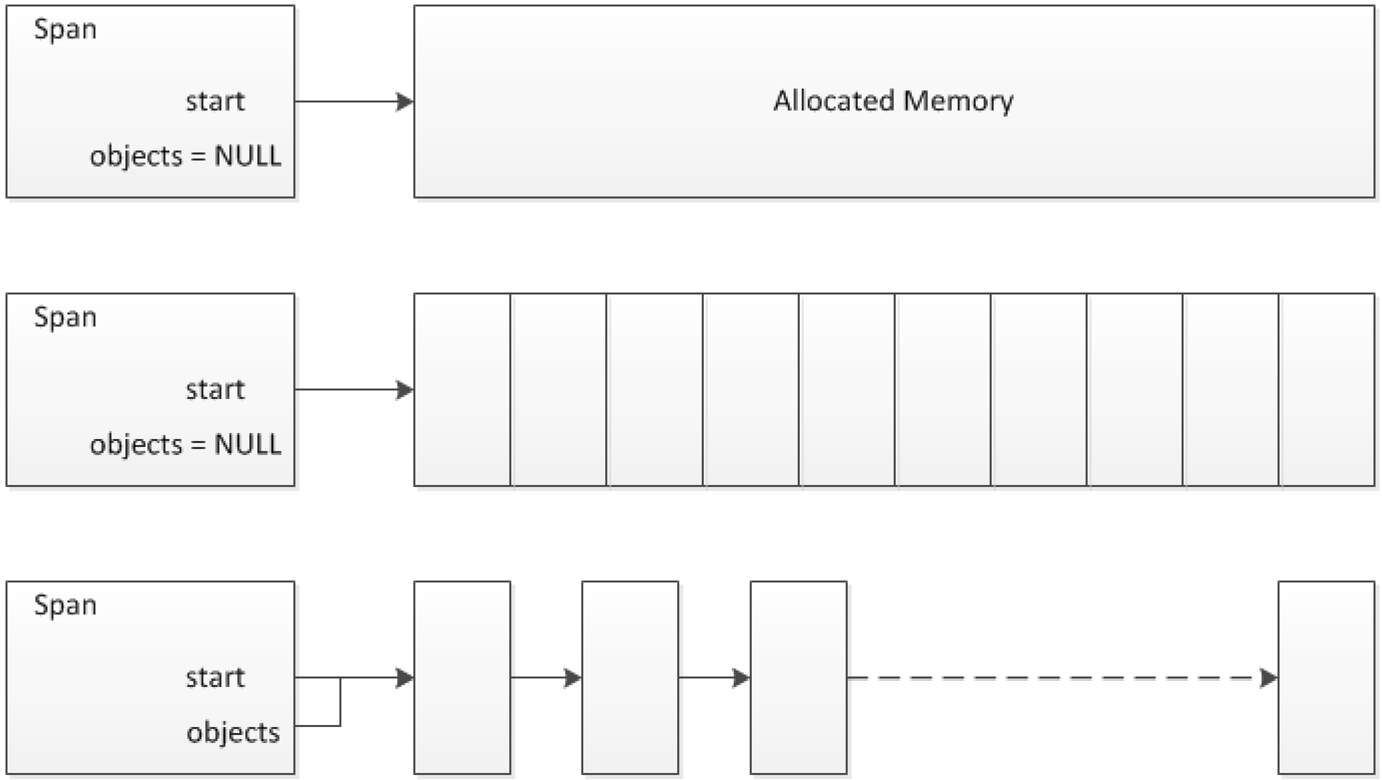
  -> CentralFreeList::RemoveRange

    -> CentralFreeList::FetchFromSpansSafe

      ->CentralFreeList::Populate

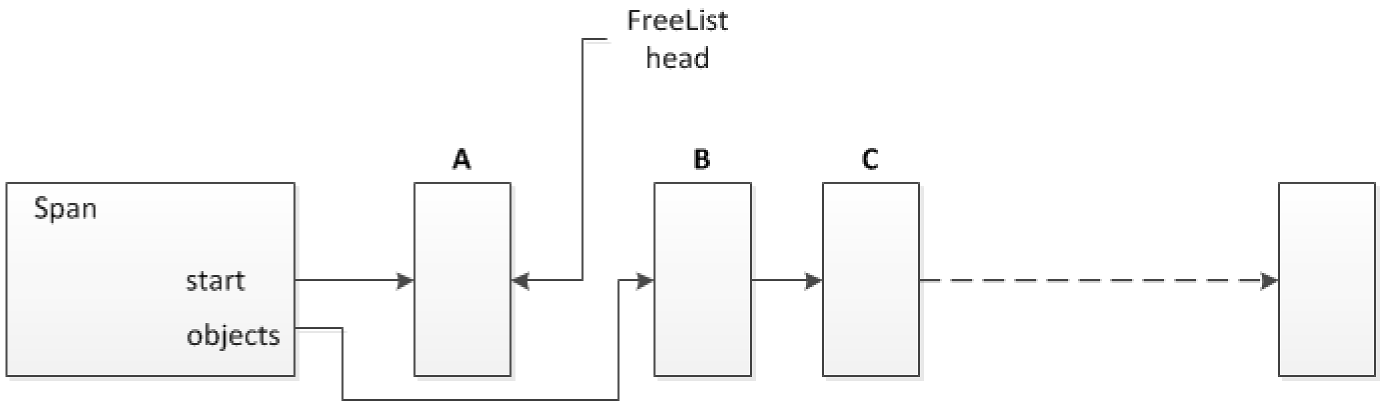
- FetchFromSpans returns the chunks in address order but RemoveRange creates its list by prepending the chunks to the head
- The result – Chunks in a new FreeList are *behind* a newly allocated chunk

FreeList Creation: Populate()



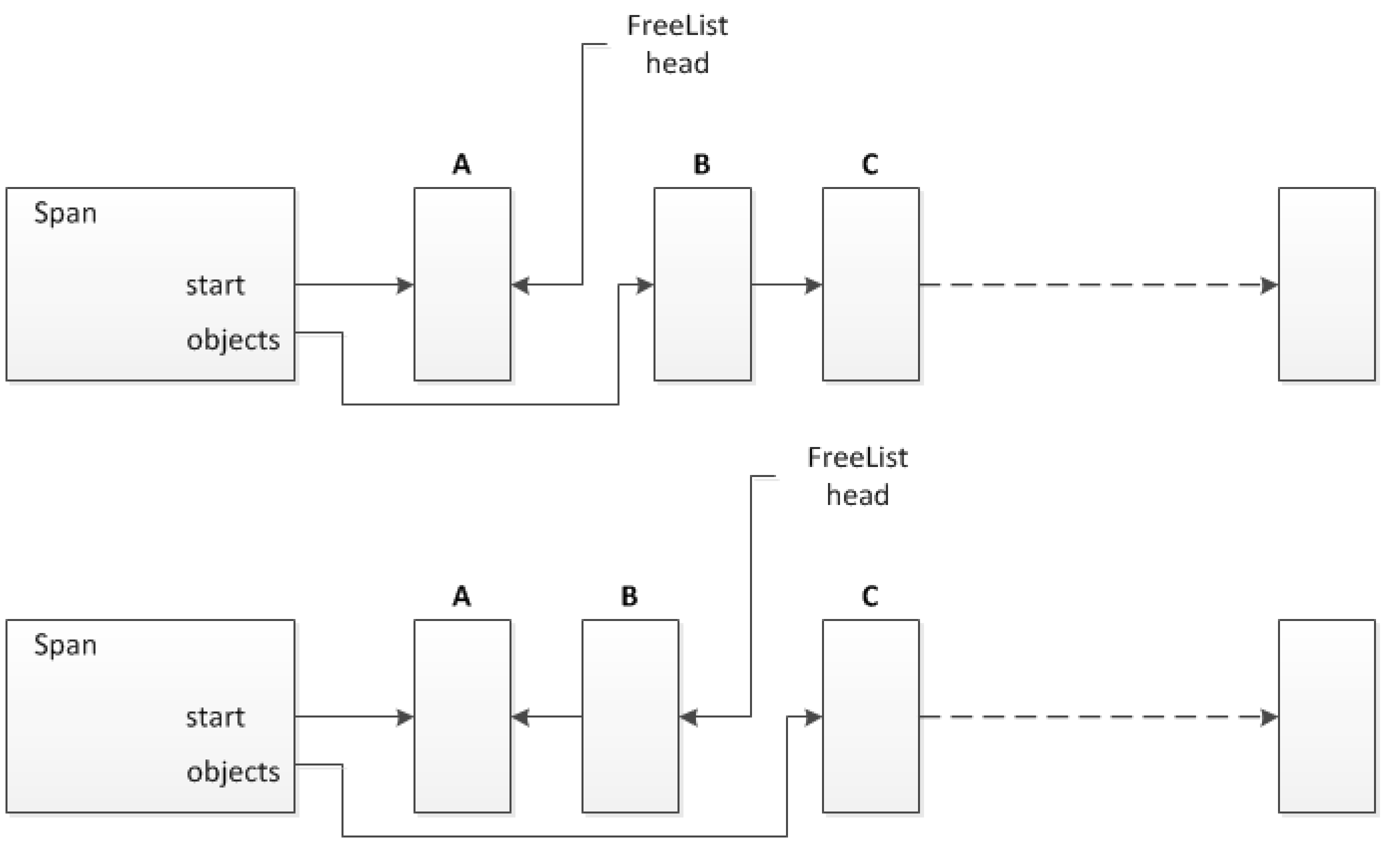
- Populate (re)sets the objects list of a span into address order

FreeList Creation: RemoveRange()




```
while (count < num) {
    void *t = FetchFromSpans();
    if (!t) break;
    SLL_Push(&head, t);
    count++;
}
```

FreeList Creation: RemoveRange()




```
while (count < num) {
        void *t = FetchFromSpans();
        if (!t) break;
        SLL_Push(&head, t);
        count++;
}
```

- FetchFromSpans pops from the head of the objects list and SLL_Push sets each as the new free list head

FreeList Creation

- Chunks are in reverse address order
- Created from the Span starting at `0xdcaa000`

```
(gdb) dump_free_list 0xd42f364
FreeList @ 0xd42f364     Chunk size: 0x50       Length: 31
        0xdcaa960
        0xdcaa910
        0xdcaa8c0
        0xdcaa870
        0xdcaa820
        0xdcaa7d0
        0xdcaa780
        0xdcaa730
        0xdcaa6e0
        0xdcaa690
        0xdcaa640
        0xdcaa5f0
        0xdcaa5a0
        0xdcaa550
        0xdcaa500
        0xdcaa4b0
        0xdcaa460
        0xdcaa410
        0xdcaa3c0
        0xdcaa370
        0xdcaa320
        0xdcaa2d0
        0xdcaa280
        0xdcaa230
        0xdcaa1e0
        0xdcaa190
        0xdcaa140
        0xdcaa0f0
        0xdcaa0a0
        0xdcaa050
        0xdcaa000
```

Crafting the FreeList Layout

- Solution.
  - Empty the FreeList and the first Span in the nonempty list
    - The next allocation will retrieve an ordered set of chunks via Populate etc. from one or more spans and create a FreeList.
    - Maximum FreeList lengths differ between browsers
      - Safari – 256
      - Chrome – 8192
  - Rearrange the FreeList via malloc/free calls

FreeList Corruption Notes

- ThreadCache::Allocate
  - Gives the Insert to FreeList[X] technique – revives the 4-to-N byte overflow primitive
  - Requires an overflow and at least two allocations
    - The first allocation to set the list head pointer from our corrupted chunk and the second to hand back this pointer to the application
  - On allocation of the target pointer the [D|Q]WORD at this address becomes the FreeList head.
    - Need to be wary of further allocations if we cannot set this to 0x0 (End of List) or ensure that the allocation that returns the target pointer is the last pointer in the free list (length_ == 0 afterwards)

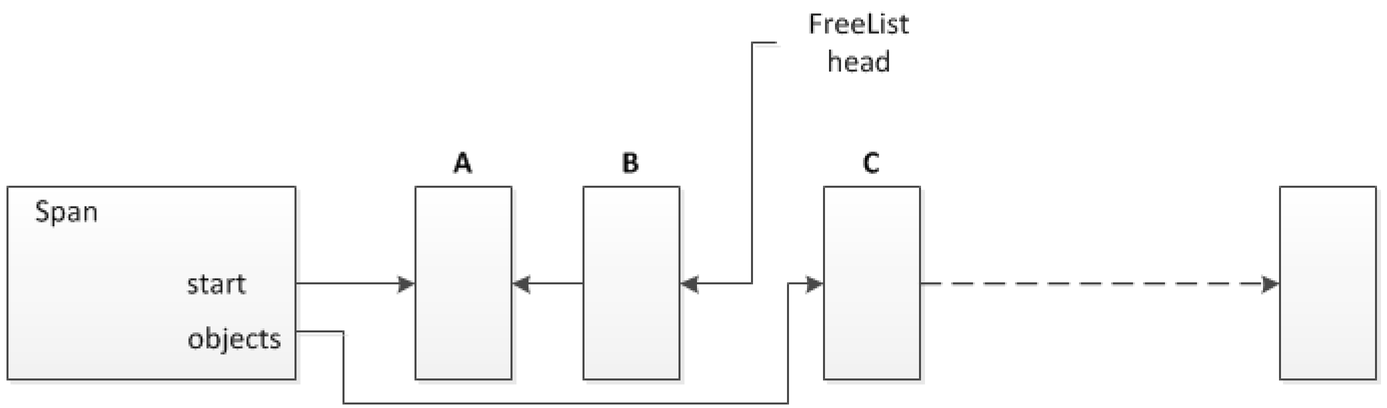
FreeList Corruption Notes

- ThreadCache::Deallocate
  - Generally functions correctly as chunks are prepended to the FreeList without walking it
  - May trigger a call to ReleaseToCentralCache if FreeList[X]->length() > kMaxFreeListLength
    - This in turn causes the FreeList to be walked through PopRange. If our corrupted chunk is within *batch_size* elements from the head of the list the corrupted next pointer will be followed as will the DWORD at that address and so on up to *batch_size* times.
  - If the memory we inserted into the FreeList is not within a page allocated by TCMalloc and gets free'd then Very Bad Things™ happen (Process death)

Span Objects List Corruption

Span Objects List Corruption

- Hang on a sec...

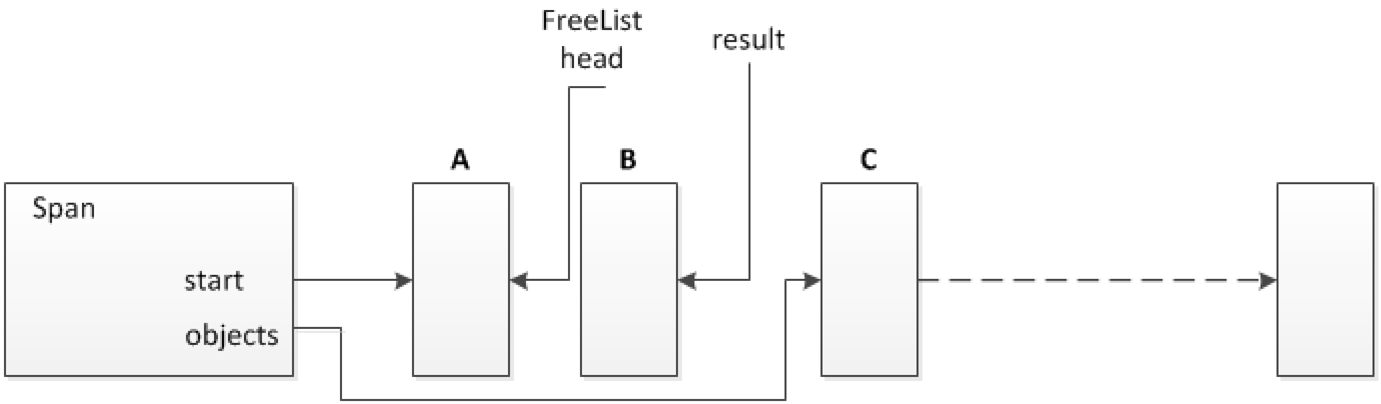


- What about the span objects list?
  - Also a singly linked list
  - The head resides directly after the FreeList head when a new Span is created and partially returned as a FreeList

Span Objects List Corruption

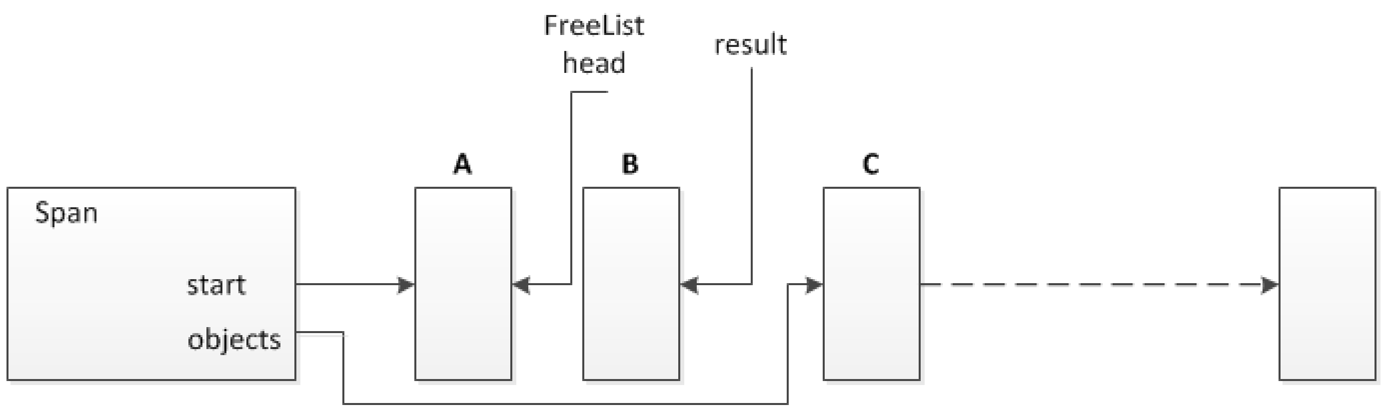
- What if we force this situation as before (empty free lists, trigger call to Populate() to reset Span object list) and overflow the first chunk returned instead of re-ordering for a FreeList overwrite

Span Objects List Corruption

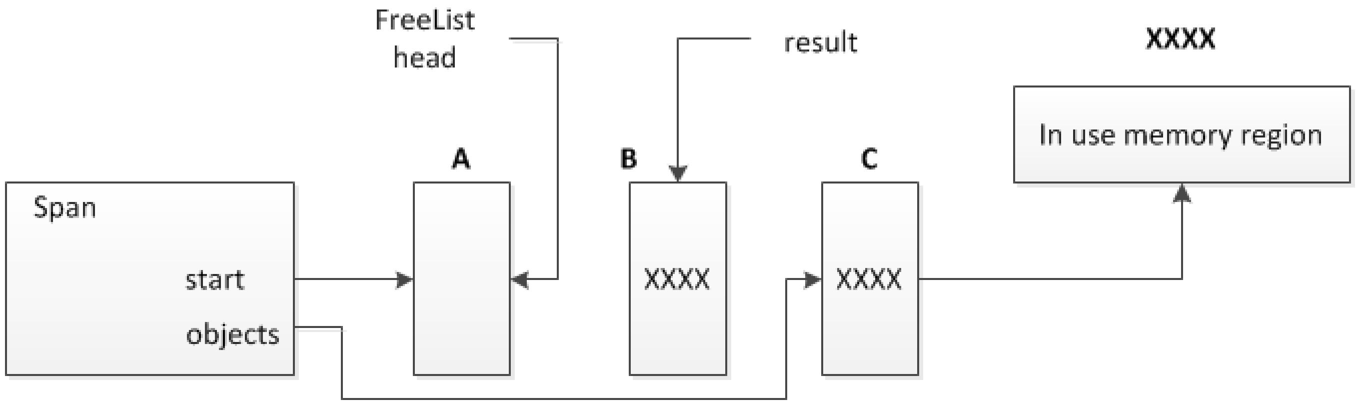


result = malloc(size)

Span Objects List Corruption



result = malloc(size)

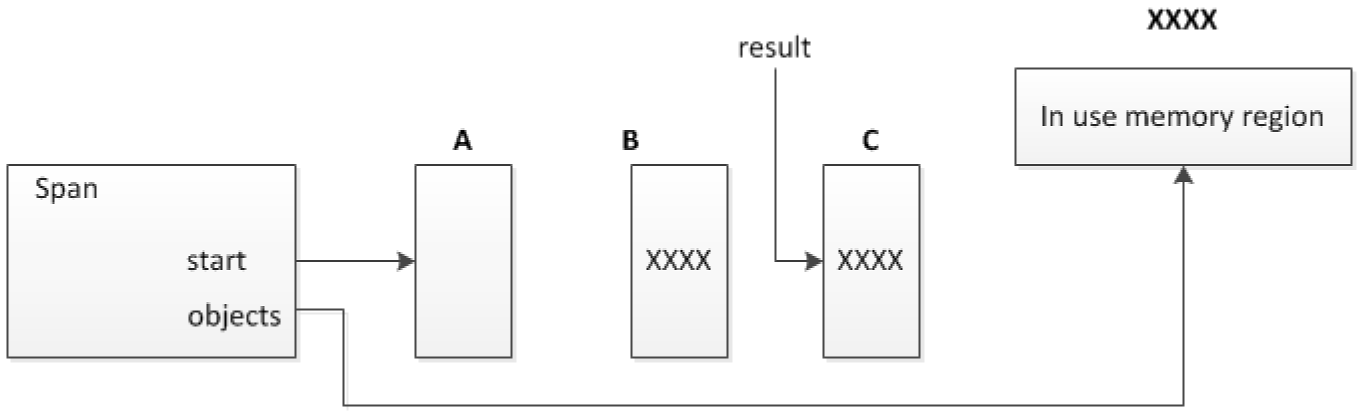


overflow(result)

Faking a FreeList

- At this point the old FreeList is untouched
- But... we have trashed the next pointer for the first chunk in the span objects list,
- The next time TCMalloc tries to build a new FreeList from this Span it will add the pointer we control to the list

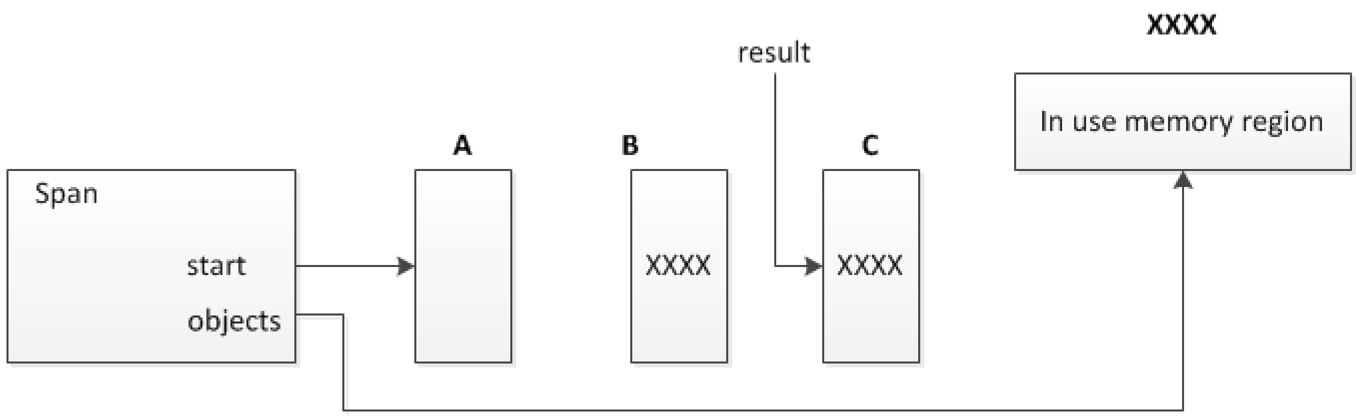
Faking a FreeList



result = FetchFromSpans();

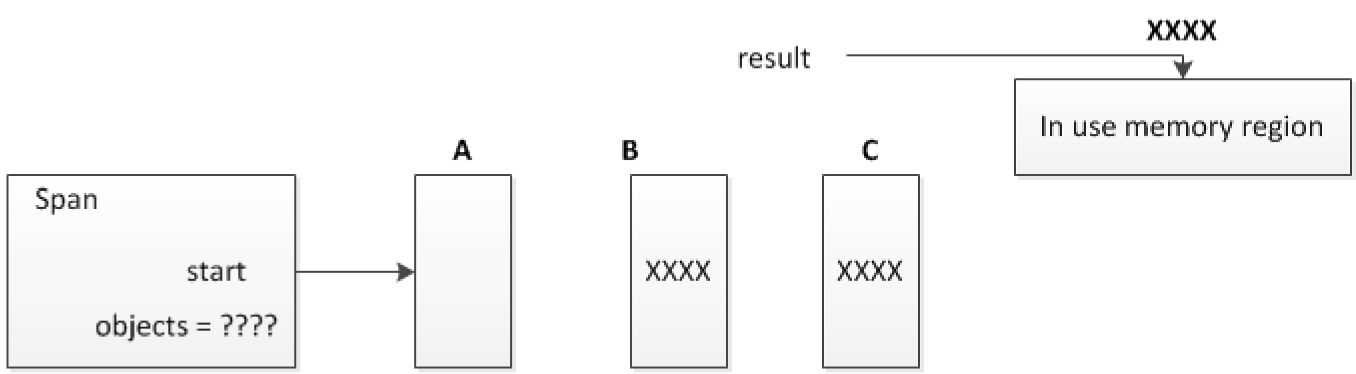
free_list.push(result)

Faking a FreeList



result = FetchFromSpans();

free_list.push(result)



result = FetchFromSpans();

free_list.push(result)

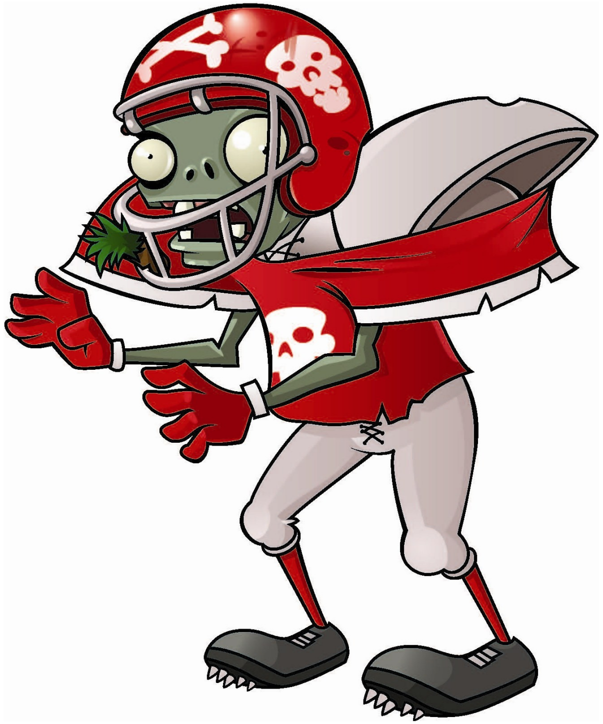
Faking a FreeList

Faking a FreeList

- What happens next depends on the first [Q|D]word of the chunk at XXXX
  - *RemoveRange()* will continue to use this Span to build the new free list until it reaches its limit or empties the Span
  - Ideally we want it to be 0x0 so the Span is considered empty
  - If not then this pointer will be followed and so on until a 0x0 is reached or enough chunks are retrieved

Summary

- Overflow the head of a new free list (or any chunk before a chunk in a Span object list) and corrupt the next pointer of a chunk in a Span object list
- Empty the FreeList for that Span object size
- Trigger another allocation of this size, causing TCMalloc to create a new free list from the non-empty spans
- This allocation will follow our controlled pointer (presuming no other spans have been added to the nonempty span list) when building the free list
- The last chunk added is directly returned to the application
- Revives the 4-to-N byte overflow primitive, again



Double Free

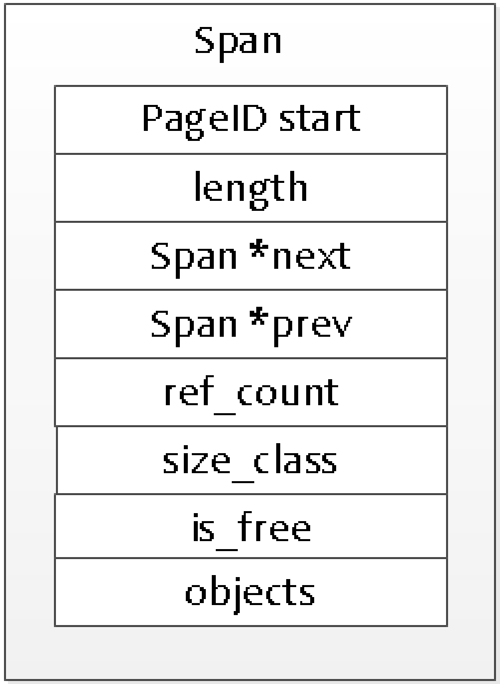
Double Free

- TCMalloc has no protection against this type of error
  - A double free of a pointer simply results in the same chunk being inserted into the FreeList twice
- A cycle in the FreeList is created if the chunk was not removed from the FreeList between frees (via allocation or returning to the CentralFreeList)
- Exploitation - obvious?
  - Allocate twice. First as an object containing function pointers then as a controllable object e.g. a string

Span Metadata Corruption: Hacking like it's 1995

Hello Darkness my old friend

```
static inline void DLL_Remove(Span* span) {
   span->prev->next = span->next;
   span->next->prev = span->prev;
   span->prev = NULL;
   span->next = NULL;
 }
```



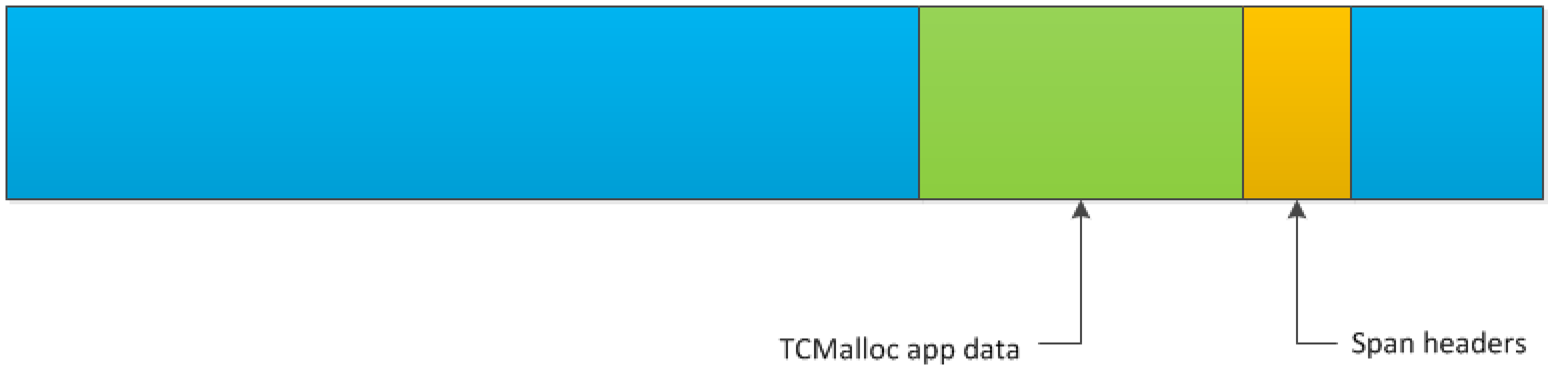
Overflowing Span Metadata

- New spans created for large allocations (>0x8000) and when the CentralFreeList runs out of chunks for smaller sizes
- Metadata for Spans is *not* stored inline with the pages representing the data
- Span headers are separate objects allocated from their own PageHeapAllocator (initially a 0x8000 byte pool created via sbrk, mmap or VirtualAlloc)

Overflowing Span Metadata

- Overflowing Span metadata is not as convenient or as common as a FreeList or Span object list overwrite
- Requires the Span pool to be after whatever chunk we overflow with no unmapped pages in between
  - May not be possible to ensure this and will depend on the OS and application embedding TCMalloc
- If we can force the required memory layout then this may allow for as many mirrored write-4s as we can overflow consecutive headers

Required Memory Layout

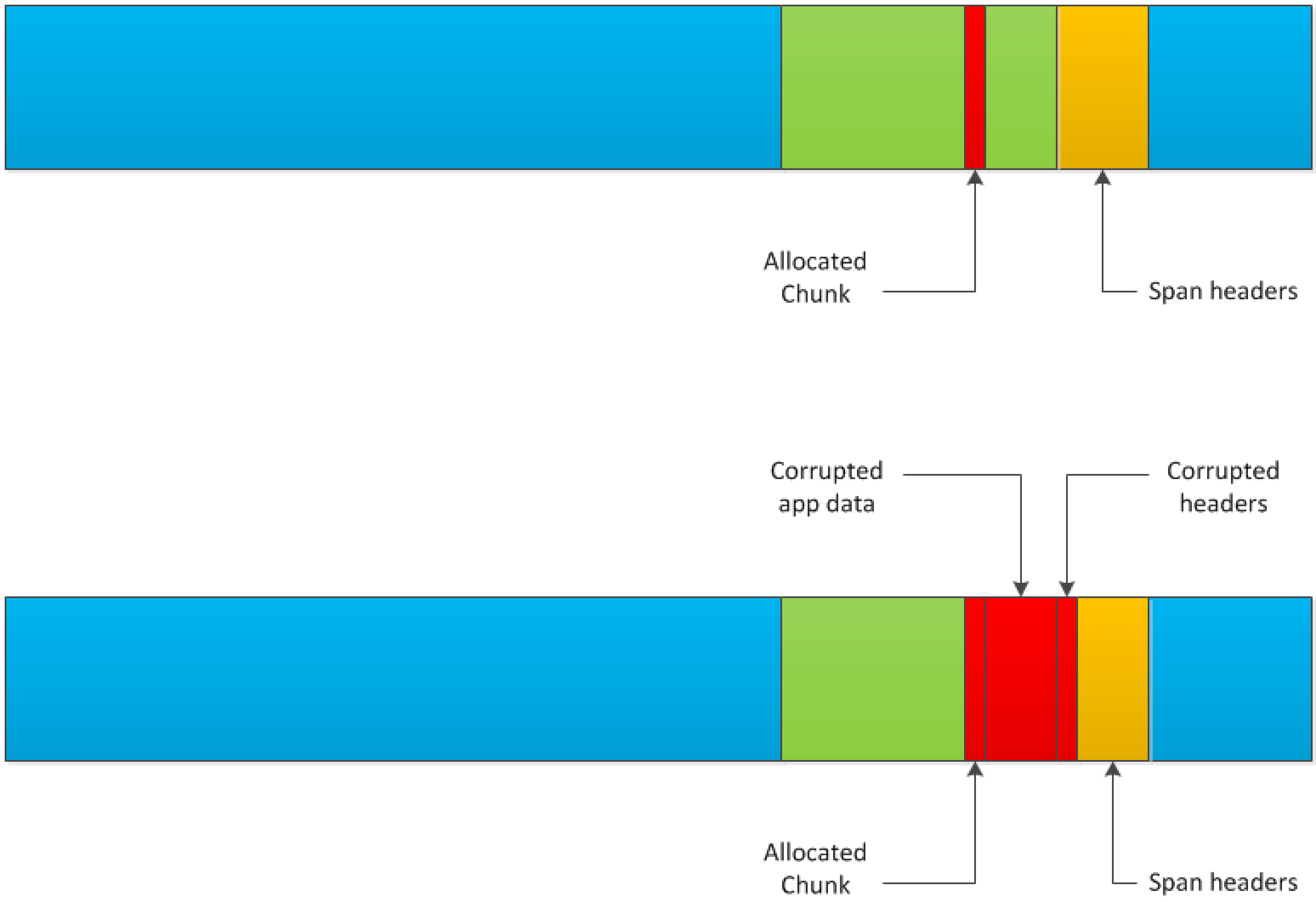


- Where is the pool of Span headers
  - If sbrk() is in use then we can force it to be after the chunks managed by TCMalloc
  - If mmap or VirtualAlloc are used then it could be in any number of locations due to randomization

Corrupting Span Metadata

- Presuming we have the correct memory layout, then what?
  - We need an overflow large enough to cover the gap between our allocated chunk and the Span metadata

Corrupting Span Metadata



Triggering the Unlink

- *DLL_Remove* is called in a number of places as part of Span management in the *PageHeap* and *CentralFreeList*
- The most straightforward path to *DLL_Remove* appears to be through *do_free* on a chunk larger than *kMaxSize*
  - This retrieves the Span header and directly calls *pageheap->Delete(span)*
  - This in turn can lead to a call of *DLL_Remove* on headers located before and after the header 'span'

Strawberry Pudding

<sinan> That's crap. I can make a strawberry pudding with so many prerequisites

(In reference to some Windows heap technique)

Strawberry Pudding

- There are countless ways to get interesting 'things' to happen in TCMalloc depending on what you can corrupt
- The Span metadata unlink is approaching 'strawberry pudding' territory
- Many other fun primitives can be found within the heap managment routines e.g. consider the refcount attribute of a Span in the context of a double free or corrupted FreeList
- Entirely unnecessary though =D No integrity checks, we can win trivially.

WebKit Heap Manipulation

Tools of the trade

- Immunity Debugger + GDB
  - Immunity Debugger
  - GDB (OS X + Android)
  - Allows us to dump information about the state of the heap
    - Chunk size, etc.
- vmmap
  - Accurate view of the state of a processes memory

Heap Primitives

- We need three simple things
  - To allocate memory
  - To free memory
  - To control the contents of allocated chunks
- Bonus
  - Predict the heap layout

Current Techniques

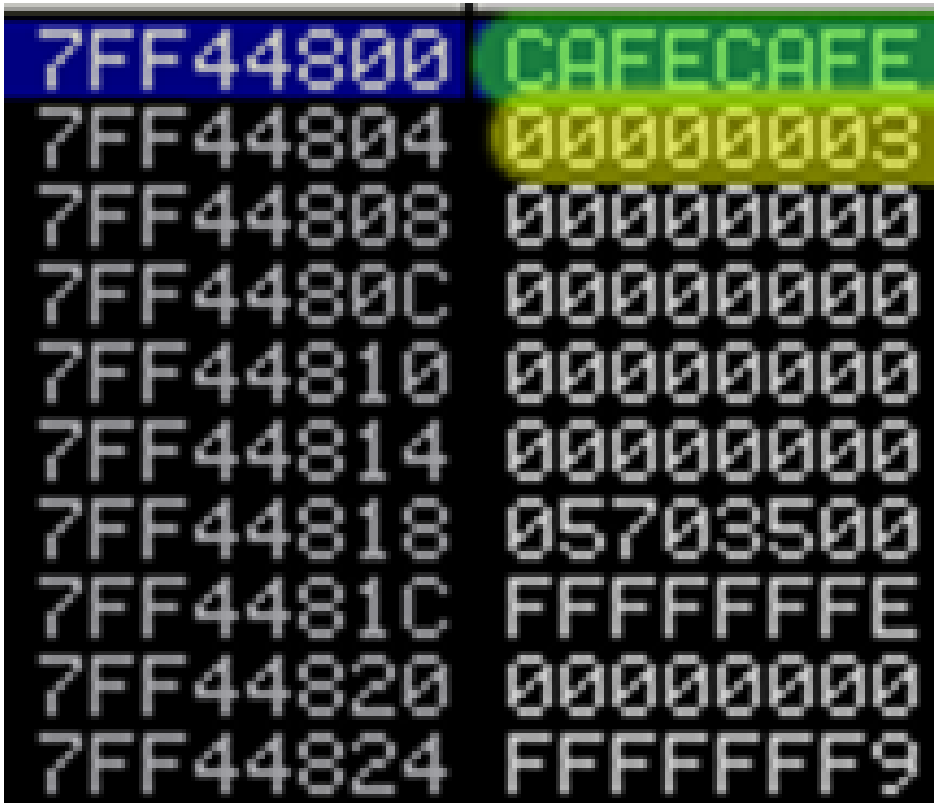
- Array Allocation
  - No deterministic chunk free
    - It relies on the behavior of the garbage collector
  - Control just the first [Q|D]WORD
    - We are screwed if our function pointer is offset+8
  - In newer releases of WebKit the array creation is deferred until the elements are assigned.
    - We need to force a reallocation by assigning each one of the elements of the array.
  - Summing up, it is inconvenient

Current Techniques

- Plain String allocation
  - Rendered useless because of Ropes
    - Ropes are a non linear representation of strings
    - A string is represented by a tree of arrays of characters
    - Each one of the nodes can be reused by others strings
      - Doing a substring on a string does not copy anything, just adds a new reference to the node/nodes
    - We need to find a way to build “linearized” strings
      - More on this later

Array Technique

- Control of the first [Q|D]word
- To allocate N bytes ...
  - S = [Q|D]WORD_SIZE
  - E = # of array elements
  - C = Constant
  - N = S * E + C
  - E = (N – C) / S
- We need an array of “E” elements
- Example allocation
  - Green: Controlled DWORD
  - Yellow: Partially controlled DWORD



Array Spray Example

- Allocate 'n' chunks
- Size 5*4 + 20
- First DWORD is 0xcafecafe
- Second DWORD is 0x00000003

```
function spray(n) {
    var h1 = [];
    for (i = 0 ; i < n; i++) {
        h1[i] = new Array(0x5);
        h1[i].length = 0xcafecafe; // first [Q|D]WORD
        h1[i][0] = 0xbadc0ded;     // second [Q|D]WORD will be 3
        h1[i][1] = 0xbadc0ded;
        h1[i][2] = 0xbadc0ded;
    }

    return h1;
}
```

Allocation Primitive

- Since there is no direct fastMalloc available
  - We need to get creative
- First approach:
  - Just build strings!
  - The catch: ropes
- Second approach:
  - Take a look at the source code
  - Realize what 'unescape' does

Unescape

- Unescape takes an encoded string and decodes it.
- To do so it needs the string in linear form (ie. No ropes)
- Appends each decoded char to a StringBuilder
- StringBuilder needs memory to hold the "unescaped" string.
- Potentially this gives us control over
  - The size of the allocation
  - The contents of the created chunks

String Builder

- Uses a reference counted storage
- Manages memory allocation automatically
- 'unescape' will append the unescaped characters to the a StringBuilder
- If more memory is needed, appendUninitialized will allocate a new buffer
- Size of the new allocation:
  - new_size = prev_size + (prev_size >> 2) + 1
- The previous buffer will be **freed** if its reference count reaches zero.
  - This will be always the case when using unescape

```
class StringBuilder
{
    unsigned m_length;
    String m_string;
    RefPtr<StringImpl> m_buffer;
    UChar* m_bufferCharacters;

    void append(...);
    String toString();
    unsigned length() const;
    void reserveCapacity(unsigned);
    void resize(unsigned);
    void allocateBuffer(const UChar*, unsigned);
    UChar* appendUninitialized(unsigned length);
};
```

Heap Spray

- String size is divided by two to take into account that each character is two bytes
- This will create 50 chunks of size 0x2c0

```
<html>
    <body onload="runTest()">
        <script>
            function spray(size, n) {
                var string_size = size / 2;
                var str = unescape("%ucafe%ucafe");
                var c   = unescape("%u1111%u1111");

                while (str.length < string_size)
                    str += c;

                var h1 = [];
                h1[0] = str.substring(0, string_size);

                for (i = 1 ; i <= n; i++)
                    h1[i] = unescape(h1[0]);

                return h1;
            }

            function runTest() {
                var pepe = spray(0x2c0, 50);
            }
        </script>
    </body>
</html>
```

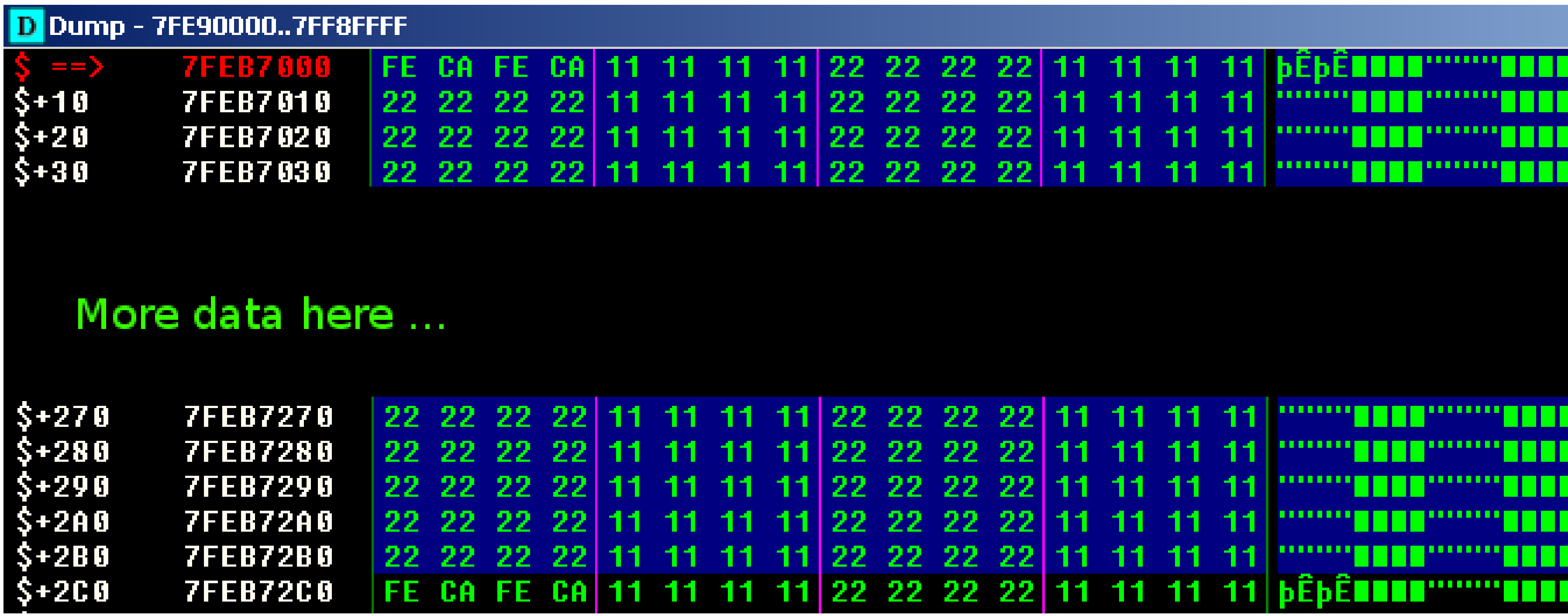
Heap Spray

- Chunks are contiguous (ie. No metadata inbetween)
  - Hence aligned to the object size
- The whole contents of the objects are controlled
  - Allows to craft really complex objects

Heap Spray

#	Adddress	What?
1	0x7feb7000	fe ca fe ca 11 11 11 11
2	0x7feb72c0	fe ca fe ca 11 11 11 11
3	0x7feb7580	fe ca fe ca 11 11 11 11
4	0x7feb7840	fe ca fe ca 11 11 11 11
5	0x7feb7b00	fe ca fe ca 11 11 11 11
6	0x7feb7dc0	fe ca fe ca 11 11 11 11
7	0x7feb8080	fe ca fe ca 11 11 11 11
8	0x7feb8340	fe ca fe ca 11 11 11 11
9	0x7feb8600	fe ca fe ca 11 11 11 11
10	0x7feb88c0	fe ca fe ca 11 11 11 11
More chunks here		
46	0x7ff7e080	fe ca fe ca 11 11 11 11
47	0x7ff7e340	fe ca fe ca 11 11 11 11
48	0x7ff7e600	fe ca fe ca 11 11 11 11
49	0x7ff7e8c0	fe ca fe ca 11 11 11 11
50	0x7ff7eb80	fe ca fe ca 11 11 11 11

Heap Spray


D Dump - 7FE90000..7FF8FFFF
$ ==> 7FEB7000 FE CA FE CA 11 11 11 11 22 22 22 22 11 11 11 11 þÊþÊ
$+10 7FEB7010 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+20 7FEB7020 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+30 7FEB7030 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
More data here …
$+270 7FEB7270 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+280 7FEB7280 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+290 7FEB7290 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+2A0 7FEB72A0 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+2B0 7FEB72B0 22 22 22 22 11 11 11 11 22 22 22 22 11 11 11 11
$+2C0 7FEB72C0 FE CA FE CA 11 11 11 11 22 22 22 22 11 11 11 11 þÊþÊ

Deallocation Primitive

- There is no direct 'fastFree' available
- Traditional approach:
  - Loop until GC kicks in
  - This is not reliable
- Our approach
  - Abuse the behavior of the StringBuilder

Deallocation Primitive

- Unescape appends decoded chars to the StringBuilder
  - This will trigger a new allocation
  - new_size = prev_size + (prev_size >> 2) + 1
    - std::vector like allocation behavior
  - The previous string will be immediately freed
- Unescape a string bigger than the one we need to free
- This will generate some heap noise
  - Must be taken into account
  - Most of the times it does not harm

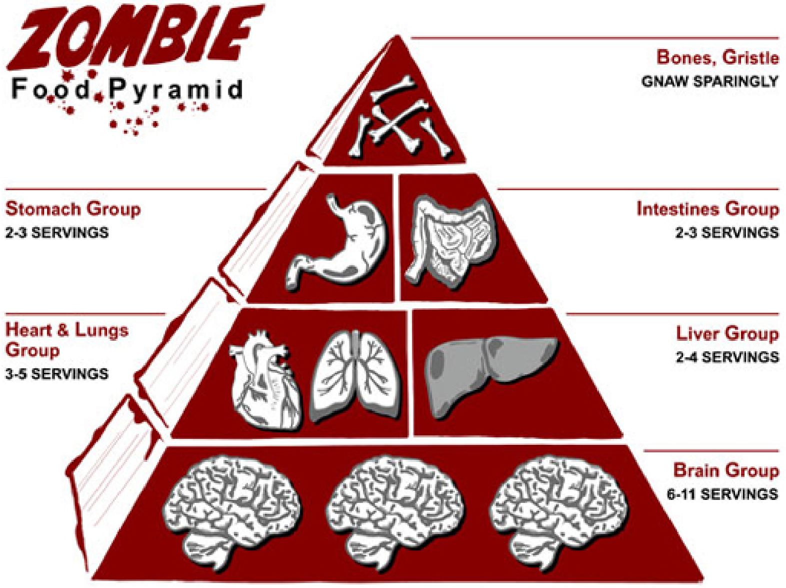
Allocation Trace

```
fastMalloc(size=0x000000a2) = 0x10c5b94d0
fastMalloc(size=0x000000cc) = 0x100b32c30
fastfree(0x10c5b94d0)
fastMalloc(size=0x00000100) = 0x10ee61b00
fastfree(0x100b32c30)
fastMalloc(size=0x00000142) = 0x10ee40dc0
fastfree(0x10ee61b00)
fastMalloc(size=0x00000194) = 0x10ee758c0
fastfree(0x10ee40dc0)
fastMalloc(size=0x000001fa) = 0x10ee57800
fastfree(0x10ee758c0)
fastMalloc(size=0x0000027a) = 0x100b46780
fastfree(0x10ee57800)
fastMalloc(size=0x0000031a) = 0x100b736c0
fastfree(0x100b46780)
fastMalloc(size=0x000003e2) = 0x10c5b7000
fastfree(0x100b736c0)
fastMalloc(size=0x000004dc) = 0x10c654000
fastfree(0x10c5b7000)
fastMalloc(size=0x00000614) = 0x10ee2e380
fastfree(0x10c654000)
fastMalloc(size=0x0000079a) = 0x10ee52800
fastfree(0x10ee2e380)
fastMalloc(size=0x00000982) = 0x10ee12400
fastfree(0x10ee52800)
fastMalloc(size=0x00000820) = 0x10f209b00
fastfree(0x10ee12400)
```

- We want to make a hole of size 0x79a
- The corresponding allocation size is 0x820

Conclusions

- A simple heap leaves us with lots opportunities to exploit vulnerabilities
- Heap layout modification is easy again by using the “unescape” technique.
- No heap protections makes our life easy.



Previous Work

- Mark Daniel
- Jake Honoroff
- Charlie Miller
- Skypher

References

- http://goog-perftools.sourceforge.net/doc/tcmalloc.html
- https://trac.webkit.org/wiki/FastMalloc%20Glossary
- http://securityevaluators.com/files/papers/isewoot08.pdf

The end

THX

AGUSTIN GIANNI
AGUSTIN@IMMUNITYINC.COM
@AGUSTINGIANNI

SEAN HEELAN
SEAN@IMMUNITYINC.COM
@SEANHN