IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

Understanding the heap by breaking it

A case study of the heap as a persistent data structure through non-traditional exploitation techniques

Justin N. Ferguson // BH2007

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

The heap, what is it?

- **Globally scoped data structure**
- **Dynamically allocated memory**
- **‘exists-until-free’ life expectancy**
- **Compliment to the stack segment**

Glibc implementation

- **Original implementation was by Doug Lea (dlmalloc)**
- **Current implementation by Wolfram Gloger (ptmalloc2)**
- **ptmalloc2 is a variant of dlmalloc**
- **Ptmalloc2 supports multiple heaps/arenas**
- **Ptmalloc2 supports multi-threaded applications**
- **Talk uses Glibc 2.4**
- **When research on the subject matter started Glibc 2.4 was current**
- **Glibc 2.6 seems to be by and large the same to us**

Glibc implementation

- 'The heap' is a misnomer – multiple heaps possible
- Heap is allocated via either sbrk(2) or mmap(2)
- Allocation requests are filled from either the 'top' chunk or free lists
- Allocated blocks of memory are navigated by size
- Free blocks of memory are navigated via linked list
- Adjacent free blocks are potentially coalesced into one
- Implies: no two free blocks of memory can border each other

Heap data structures

- Each heap has:
  - heap_info structure
  - malloc_state structure
  - any number of malloc_chunk structures

- heap_info structure contains/defines:
  - size of heap
  - pointer to arena for heap
  - pointer to previous heap_info structure

- malloc_state structure contains/defines:
  - mutual exclusion variable
  - flags indicating status/et cetera of the arena
  - arrays of pointers to malloc_chunks (fastbin & normal)
  - pointer to next malloc_state structure
  - other less important (to us) variables

Heap data structures

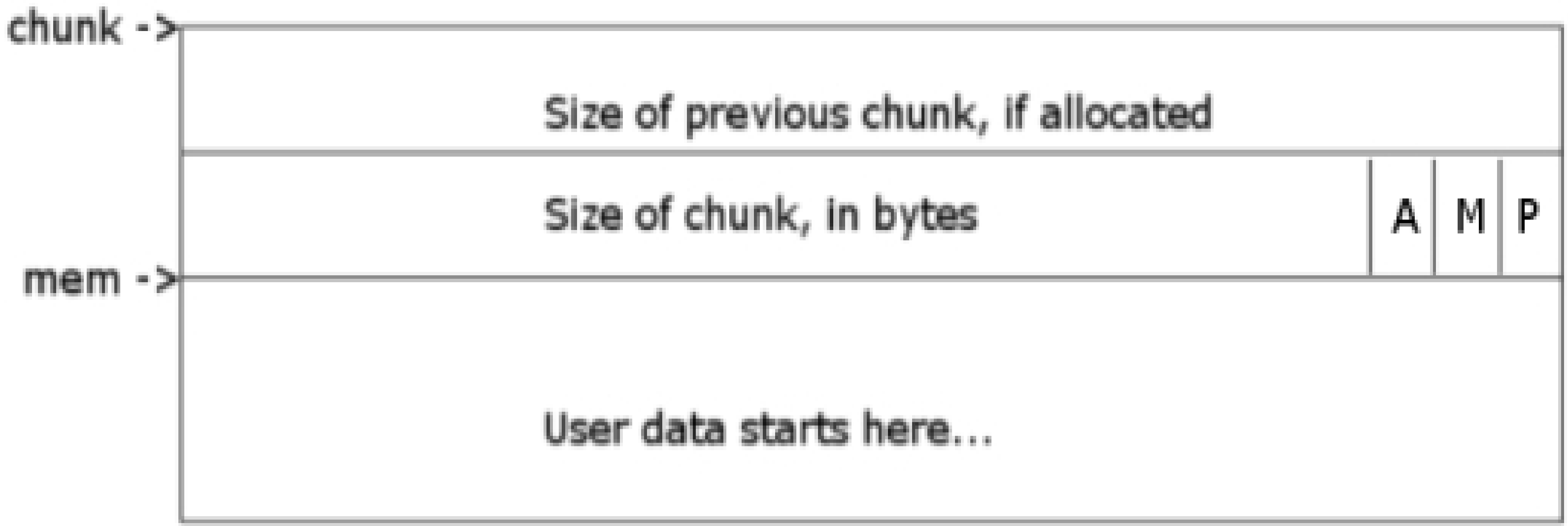
- **malloc_chunk structure contains:**
  - **size of previous adjacent chunk**
  - **size of current (this) chunk**
  - **if free, pointer to the next malloc_chunk**
  - **if free, pointer to the previous malloc_chunk**

- **most commonly known heap data structure**
- **Interpretation of chunk changes varying on state (important!)**
- **malloc_chunk C structure:**

```
struct malloc_chunk {
        INTERNAL_SIZE_T         prev_size;
        INTERNAL_SIZE_T         size;
        struct malloc_chunk *   fd;
        struct malloc_chunk *   bk;
}
```

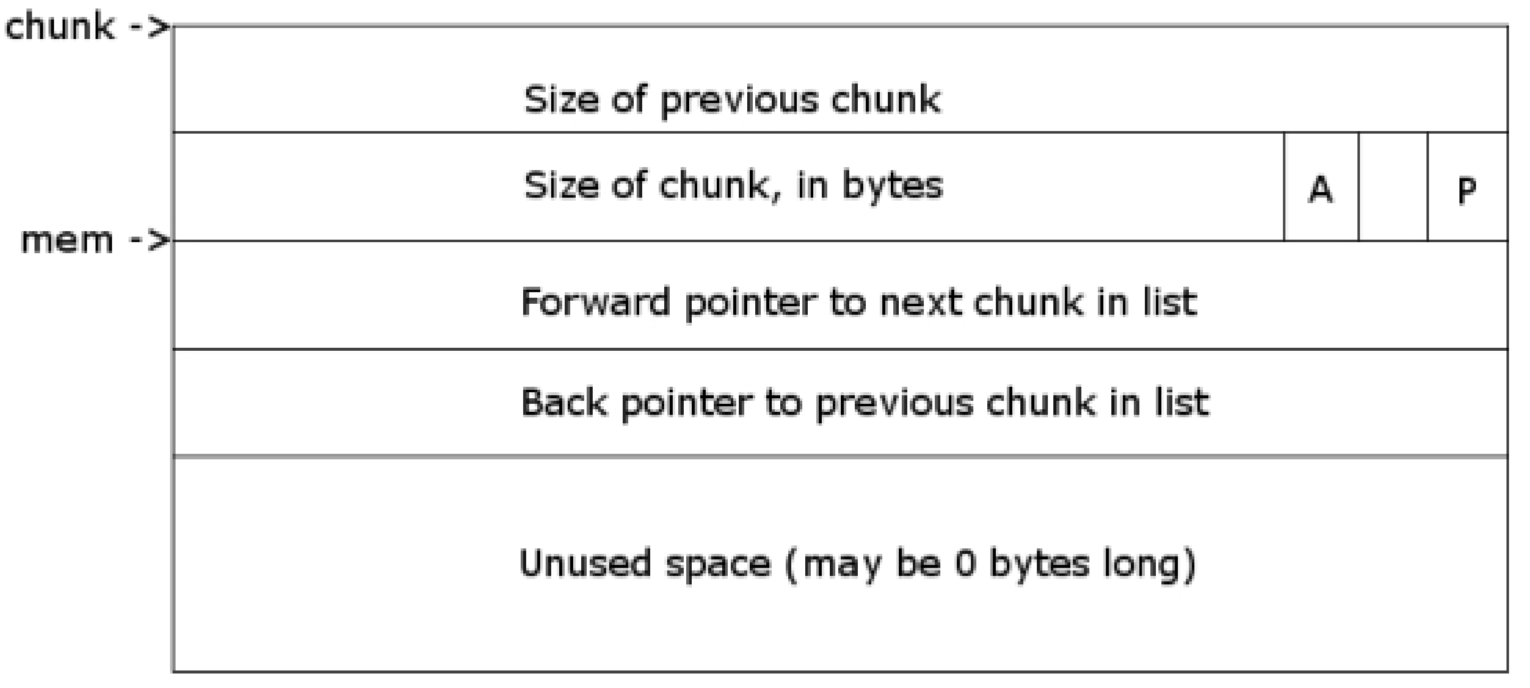
Heap data structures

- malloc_chunks have different interpretations dependent upon chunk state
- Despite physical structure not changing
- Allocated block of memory is viewed in the following way



Heap data structures

- **Blocks of free memory have the same physical structure**
- **Parts of memory are reused for metadata**
- **Free chunk has the following representation**



Binning of free blocks

- Free chunks are placed in bins
- Bin’s are just an array of pointers to linked lists
- Bin’s could be called a free list
- Two basic different types of bin
  - fastbins
  - ‘normal’ bins
- Fastbins are for frequently used chunks
- Not directly consolidated
- Not sorted – every bin contains chunks of the same size
- Only make use of the forward pointer
- Use same physical structure as ‘normal’ bins
- ‘Normal’ bins split into three categories
  - 1st bin index is the ‘unsorted’ bin
  - then small ‘normal’ chunk bins
  - large ‘normal’ chunk bins
- Larger requests serviced via mmap(2) and thus not placed in bins

More about fastbins

- Blocks are removed from the list in a LIFO manner
- Allocations ranging from 0 to 80 fall into the fastbin range
- Default maximum fastbin size is 64 bytes
- Chunks binned by size as follows:

fastbin #	holds chunk sizes	real chunk size
0	00 - 12	16
1	13 - 20	24
2	21 - 28	32
3	29 - 36	40
4	37 - 44	48
5	45 - 52	56
6	53 - 60	64
7	61 - 68	72
8	69 - 76	80
9	77 - 80	88

‘normal’ bins

- Same physical structure (array of pointers to malloc_chunk)
- Blocks of memory less than 512 bytes fall into this range
- Small ‘normal’ chunks are not sorted
- Chunks of the same size stored in the same bin
- Fastbin chunk sizes and small ‘normal’ bin chunk sizes overlap
- Fastbin consolidation can create a small ‘normal’ bin chunk (or any other type of chunk)
- Chunks largers than 512 bytes and less than 128KB are large ‘normal’ chunks
- Bins sorted in the smallest descending order
- Chunks allocated back out of the bin’s in the least recently used fashion (FIFO)

top and last_remainder

- Special chunks
- Neither ever exist in any bin
- Top chunk borders end of available memory
- Top chunk is used for allocation (if possible) when free lists can't service request
- Chunks bordering the top are folded into the top block upon free()
- Top can grow and shrink
- Top always exists
- last_remainder can be allocated out and then upon free() placed in a bin
- last_remainder is the result of an allocation request that causes the chunk to be split

heap operations

- **Heap creation notes**
  - created implicitly
  - New arena/heap can be created mutexes

- **Block allocation notes**
  - fastbin allocations cannot cause consolidation
  - small ‘normal’ block allocation can (sometimes)
  - large ‘normal’ block allocation always calls malloc_consolidate()

- **Chunk resize notes**
  - Original chunk can be free()’d

- **Free()’ing chunk notes**
  - can trigger consolidation
  - Can cause heap to be resized

Double free()'s

- Instance of dangling pointers / use-after-free
  - (nothing new or extra-ordinary and certainly not a new bug class)

- Interesting due to insight into heap it provides
- Result of a valid instruction being used at invalid times
- In below example the free() labeled 'a' is valid
- However free() labeled 'b' is not

```
void *ptr = malloc(siz);

if (NULL != ptr) {
        free(ptr);        /* a */
        free(ptr);        /* b */
}
```

Double free()'s

- **Surprisingly undocumented**
- **Neither Vudo Malloc Tricks nor Once Upon a Free() mentions them**
- **Advanced Doug Lea's malloc exploits mentions them, kinda sorta not really**
- **The Malloc Maleficarum doesn't mention them**
- **Shellcoders handbook has a paragraph (!!) in chapter 16 that tells you they're not really exploitable**
- **Only two decent references found by author thus far**
  - **The Art of Software Security Assessment (good book)**
  - **A post to a mailing list**
- **The Art of Software Security Assessment says:**
- "There is also a threat if memory isn't reused between successive calls to free() **because the memory block could be entered into free-block list twice**"

Traditional double free() exploitation

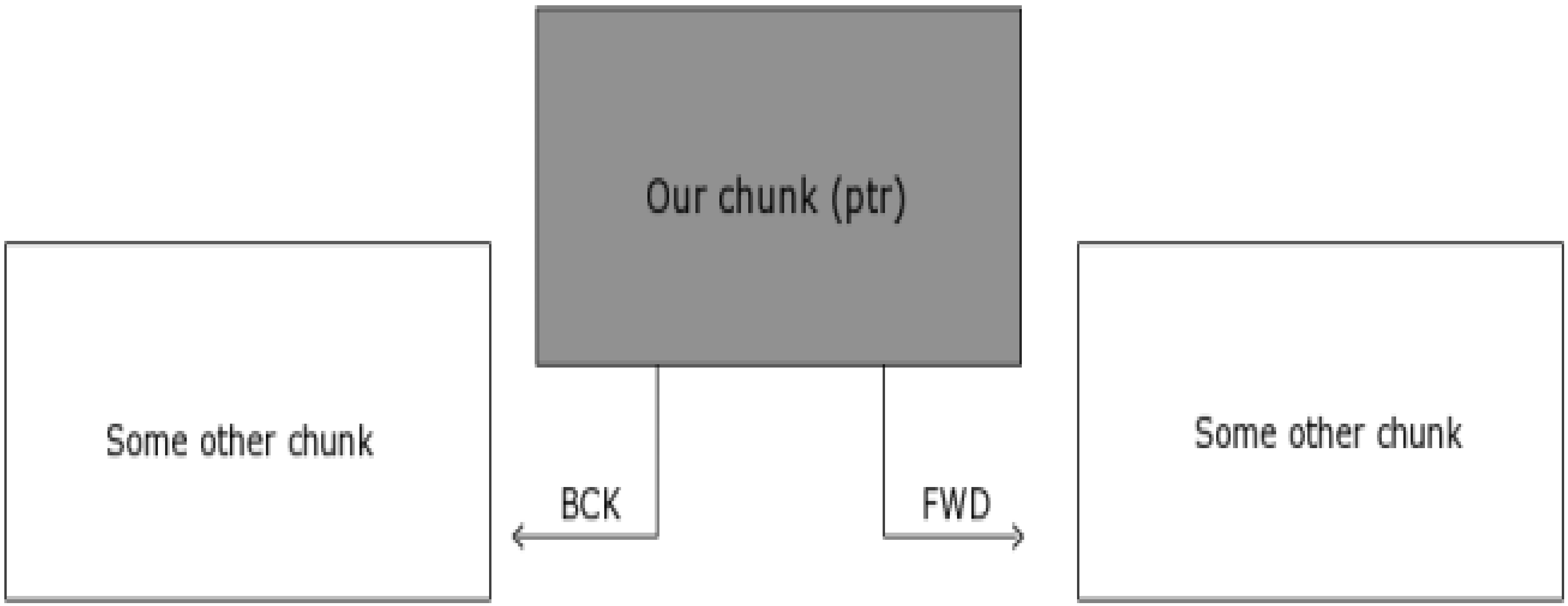
- Only in-depth talk publicly about double free() exploitation from Igor Dobrovitski in 2003
- Mailing list post detailing exploit for CVS server
- Details included most of this section
- Thanks Igor!

  (if you're here find me and I'll buy you a beer)

- Remember that an allocated chunk is represented differently than a free chunk

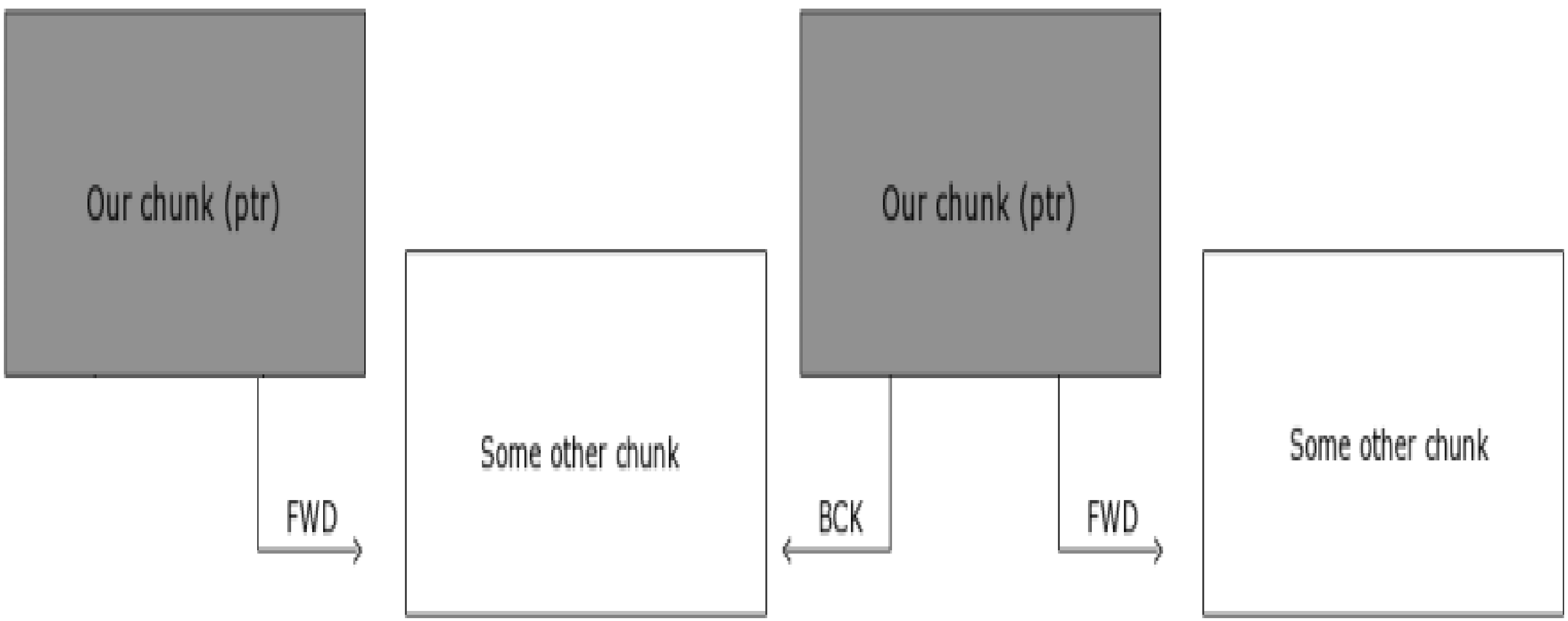
Traditional double free() exploitation

- Free blocks end up in a bin
- Bins are linked lists
- After first free list would look something like this:

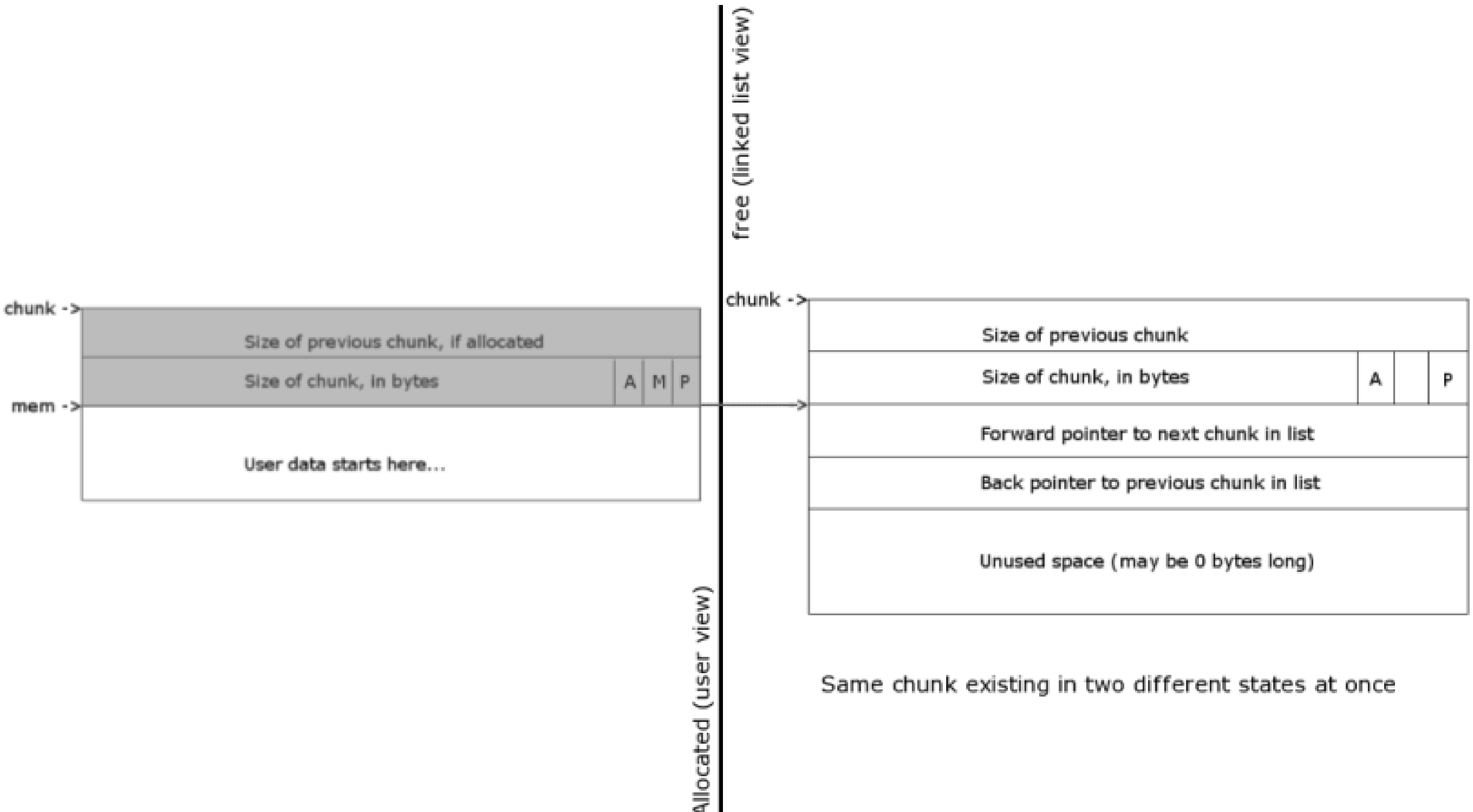


Traditional double free() exploitation

- **What happens when you free() the same chunk twice ? ;]**



Traditional double free() exploitation



Traditional double free() exploitation

- Traditional exploitation depended on the unlink() macro
- Thanks Solar Designer!

  (If you're here find me and I'll buy you a beer)
- unlink() macro back then looked like this:

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

Traditional double free() exploitation

- **Steps to traditional exploitation:**
  1. **Get the same block of memory passed to free() twice**
  2. **Get one of the chunks allocated back to you**
  3. **Overwrite the ‘fd’ and ‘bk’ pointers**
  4. **Allocate the second instance of the block on the free-list**
  5. **??**
  6. **Profit**

- **Reliable, ‘just worked’**
- **Of course, like all good things …**

Oops! It’s not 1996!

- **unlink() macro has been hardened .. Most everywhere**
- **Double free() protections have been implemented .. Most everywhere**
- **New unlink() macro:**

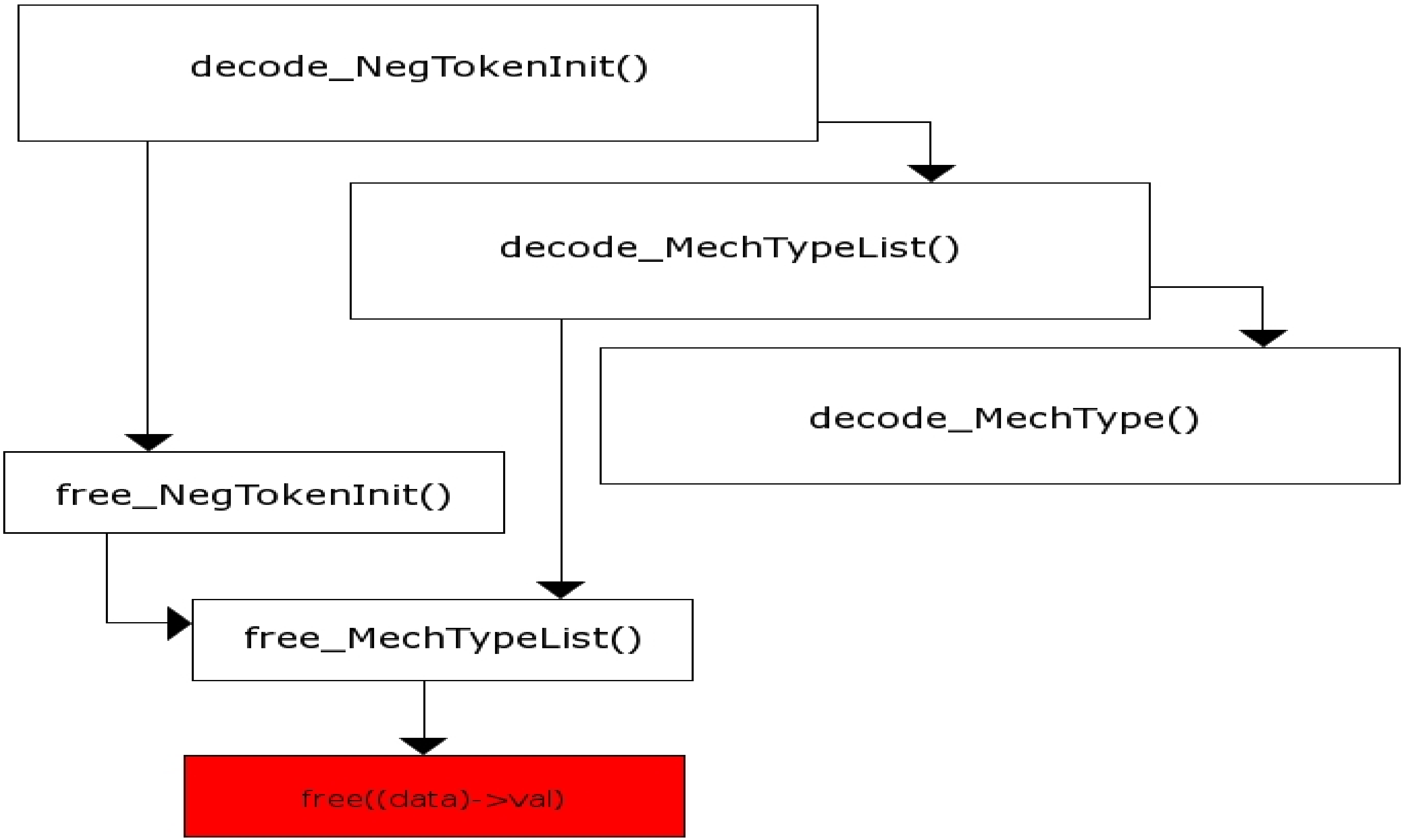
```
#define unlink(P, BK, FD) {                                              \
      FD = P->fd;                                                        \
      BK = P->bk;                                                        \
      if (__builtin_expect (FD->bk != P || BK->fd != P, 0))             \
            malloc_printerr (check_action, "corrupted double-linked list", P);   \
        else {                                                           \
              FD->bk = BK;                                               \
              BK->fd = FD;                                               \
        }                                                                \
}
```

- **Result of hackers abusing the macro**
- **Thanks hackers!**

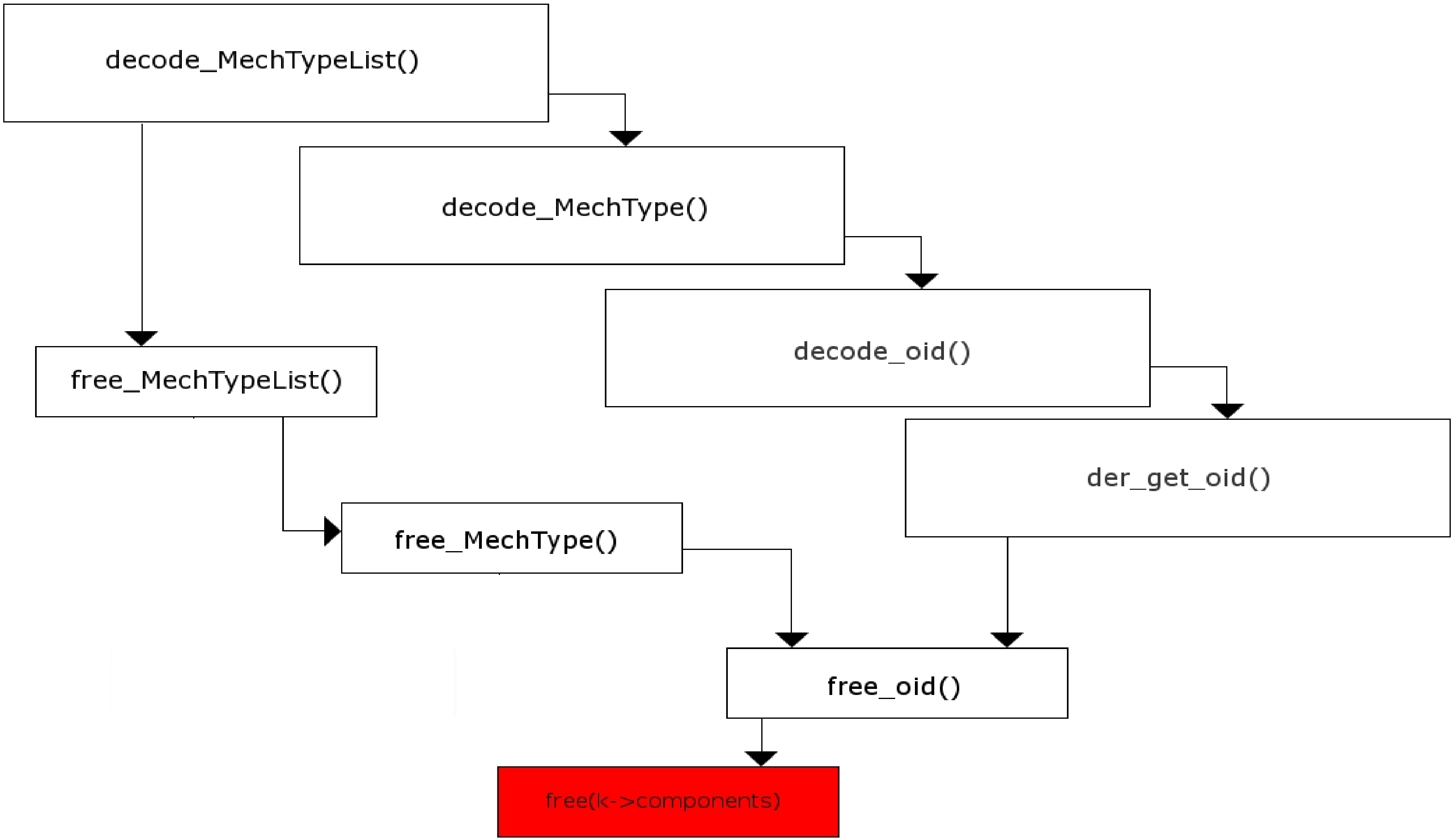
The example

- Result of multiple error handling checks being performed on functions that call each other that on error will cause a multiple free condition
- mod_auth_kerb versions 5.3, 5.2, …
- Result of using an old asn.1 compiler from Heimdal
- New ones don’t have the same problem, but have other problems
  - (yes if you’ve used it you should audit your code)
- Thanks Heimdal!

Vulnerability Zero



Vulnerability One



Threads & ptmalloc2

- Earlier versions of Glibc had no thread safety for its allocators
- Demonstrated publicly by Michal Zalewski in Delivering Signals for Fun & Profit (underappreciated)
- Thread safety is a key difference between dlmalloc and ptmalloc
- Thread safety is provided by two mutual exclusions
  - list_lock: used during heap/arena creation
  - Per-arena mutex: locked prior to entry into internal routines
- Cannot enter critical sections without a lock
- Provides thread safety, mostly

Bad logic is bad logic, mutex or not.

- **Don’t get me wrong- the mutexes are great**
- **Don’t protect against assumptions in the code base**
- **Some of those assumptions can be found in the double free() protections**
- **<blink>Glibc developers are not really at fault</blink>**
  - If you allow someone to arbitrarily corrupt metadata the game is over, they’re just trying to protect you from yourself

Double free() protections

- **Normal chunks:**

```
if (__builtin_expect (p == av->top, 0)) {
                errstr = "double free or corruption (top)";
                goto errout;
}
```

- **Checks to ensure the arena's top chunk is not the pointer being free()'d**

- **Not typically a problem outside of lab conditions**

- **Several other chunks will likely exist and will border top by the time we multiple free()**

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

Double free() protections

```
if (__builtin_expect (contiguous (av) && (char *) nextchunk >=
                      ((char *) av->top + chunksize(av->top)), 0))
{
        errstr = "double free or corruption (out)";
                goto errout;
}
```

- **Can be bypassed by making the heap non-contiguous**
  - almost always happens when new arena is created
- **Probably don't want to create another arena**
- **Takes some work to cause another arena to be created**
- **Second check is to ensure that the next chunk is outside of the arena**
  - Pretty rare condition for multiple free() bugs
  - Could happen if the heap shrank
  - Problem just isn't common enough

Double free() protections

```
if (__builtin_expect(!prev_inuse(nextchunk), 0)) {
        errstr = "double free or corruption (!prev)";
        goto errout;
}
```

- **Ouch!**
- **Can’t be bypassed through heap layout manipulation**
- **Can be bypassed when using threads**
- **Thanks Pthreads!**

IOActive™
COMPREHENSIVE COMPUTER SECURITY SERVICES

Double free() protections

- **Fastbin chunks:**

```
if (__builtin_expect (*fb == p, 0)) {
        errstr = "double free or corruption (fasttop)";
        goto errout;
}
```

- **Checks that the currently being free()'d chunk is not the last chunk that was free()'d**
- **Only 'real' check for fastbin chunks**
- **Provides reliable method for causing a multiple free() condition**
- **Thanks Wolfram!**

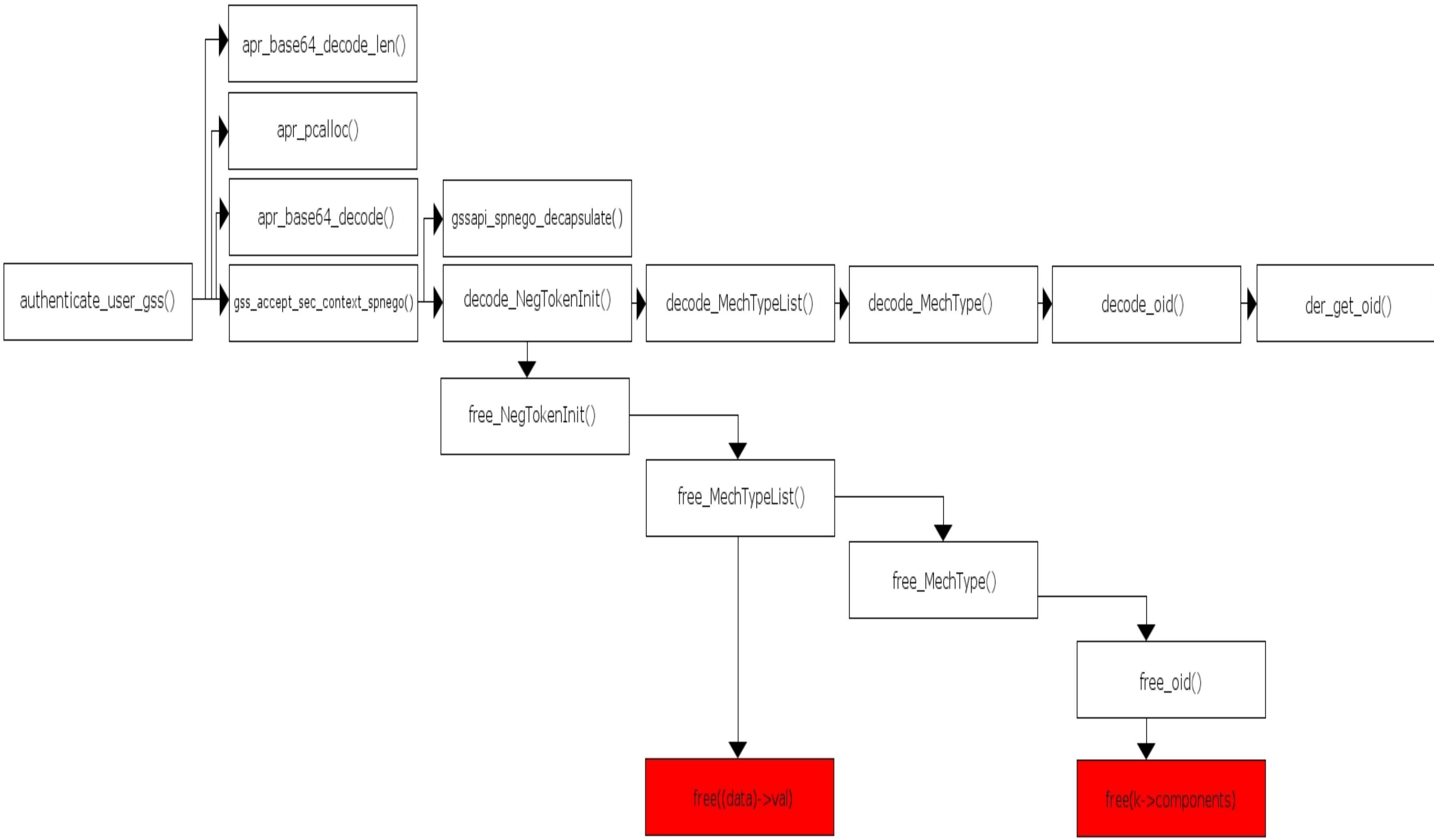
Using this knowledge in nefarious ways

- **‘Normal’ chunks**
  - Our chunk cannot get coalesced with top
  - Our chunk summed with its size cannot be outside of the bounds of the heap OR the heap needs to be non-contiguous
  - In the next chunk the previous in use bit must be set

- **Fastbin chunks**
  - Chunk being free()’d cannot be the chunk in the same bin that was most recently free()’d
  - i.e.:

```
free(a);
free(b);
free(a);
```

Vulnerability control flow



Consolidated calamity

- Using vulnerability 1
- Using fastbins
- Cannot exploit this under these circumstances
  - Problem is lack of control in the first four bytes
  - Techniques still valid
- Looking aside from that, a few techniques to own with
- Not directly asserting control via linked list operations
- Abusing fastbins by causing a consolidation
- Following set of events takes place
  - Free two different chunks of the same size (fastbin)
  - Free two different chunks again
  - Allocate first back, write to it
  - Allocate block of memory larger than 512 bytes to cause consolidation

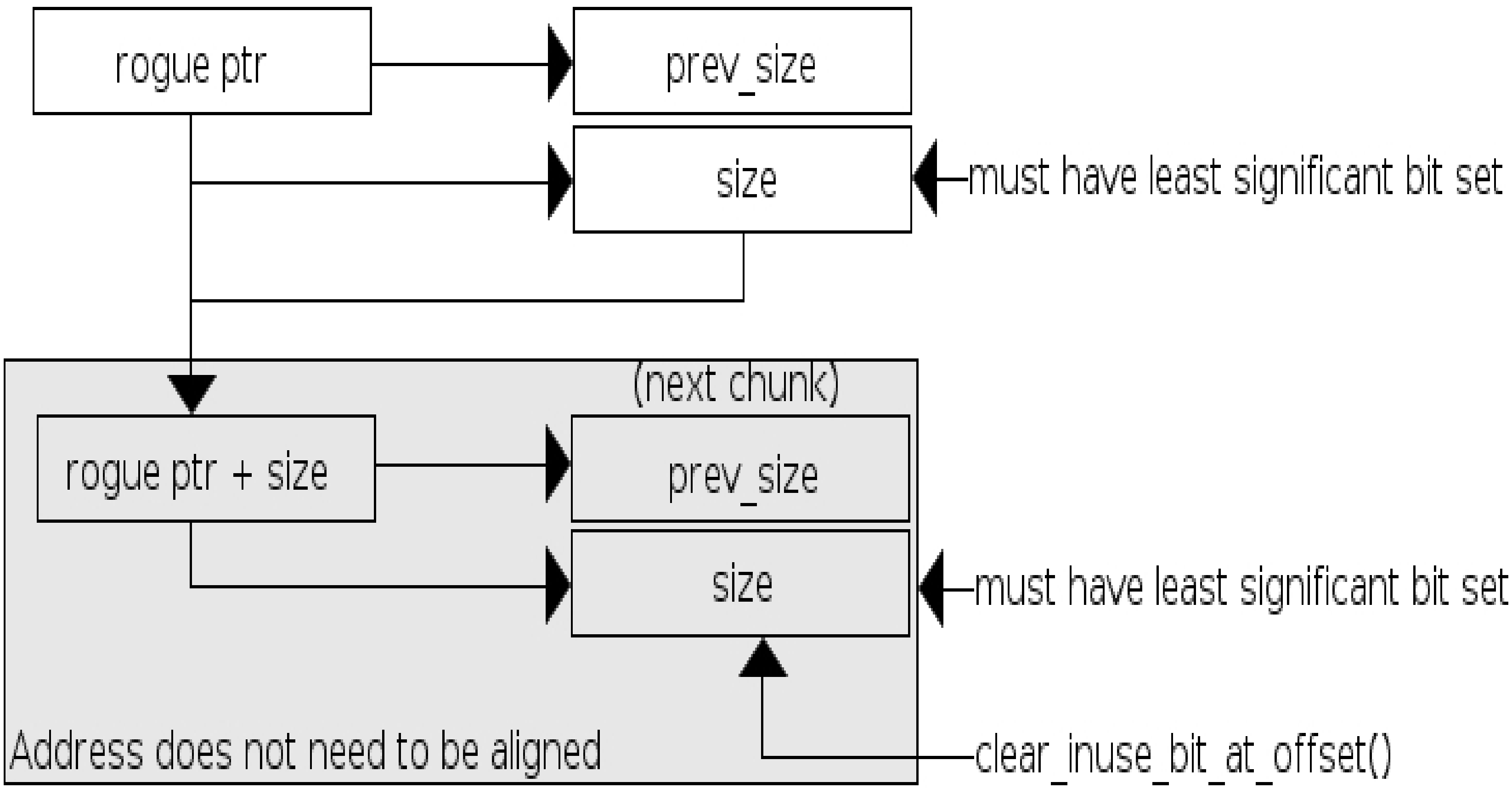
Consolidated calamity

```
size = p->size & ~(PREV_INUSE|NON_MAIN_ARENA);
nextchunk = chunk_at_offset(p, size);
nextsize = chunksize(nextchunk);

 if (!prev_inuse(p)) {
        /* backwards consolidate */
}

  if (nextchunk != av->top) {
        nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

        if (!nextinuse) {
                /* forward consolidate */
        } else
                clear_inuse_bit_at_offset(nextchunk, 0);
                /* link into unsorted bin */
                set_foot(p, size);
        }

else {
        /* modify size call set_head() */
        av->top = p;
}
```

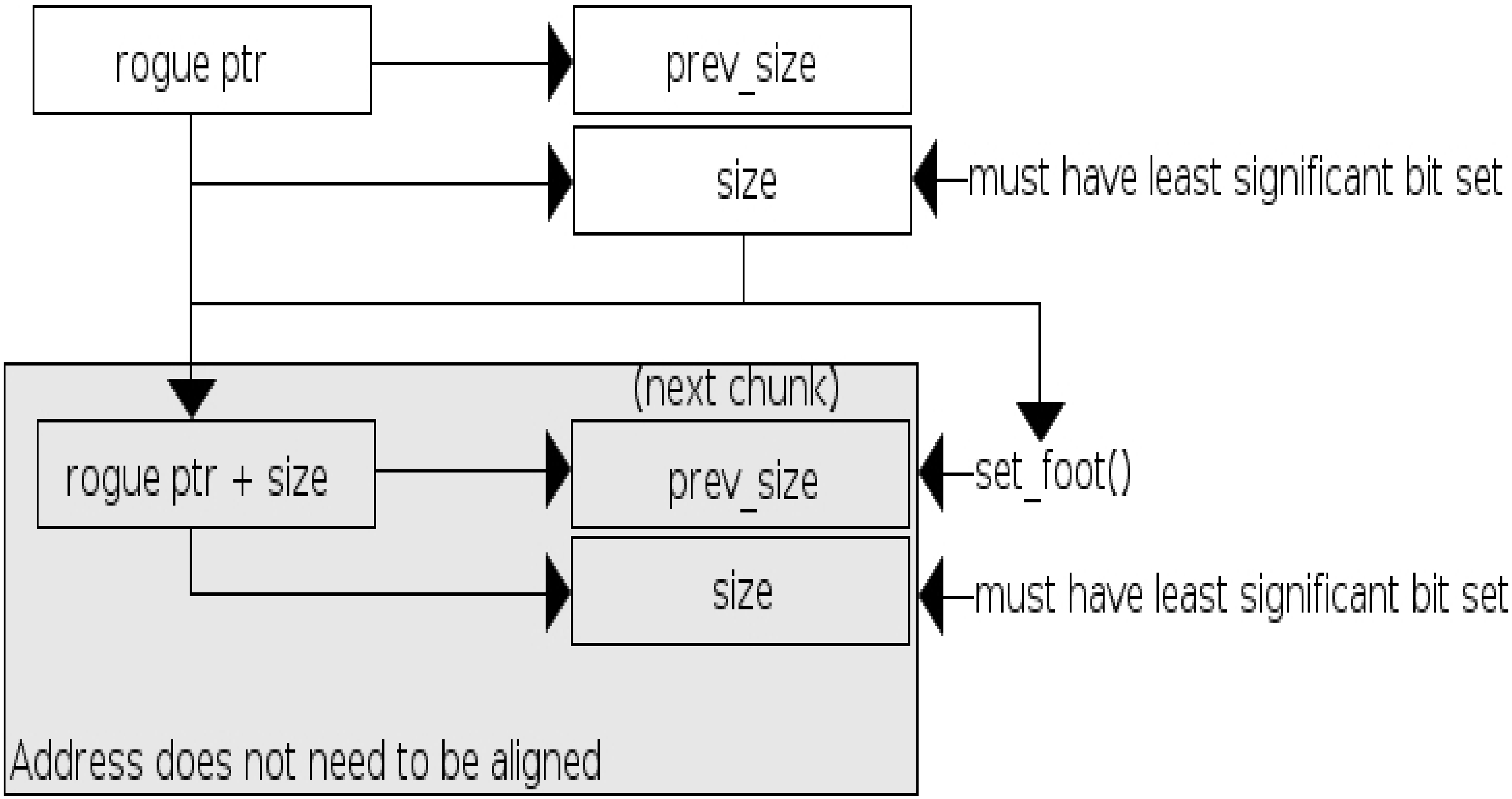
Consolidated calamity

```
size = p->size & ~(PREV_INUSE|NON_MAIN_ARENA);
nextchunk = chunk_at_offset(p, size);
nextsize = chunksize(nextchunk);

 if (!prev_inuse(p)) {
        /* backwards consolidate */
}

  if (nextchunk != av->top) {
        nextinuse = inuse_bit_at_offset(nextchunk, nextsize);

        if (!nextinuse) {
                /* forward consolidate */
        } else
                clear_inuse_bit_at_offset(nextchunk, 0);
                /* link into unsorted bin */
                set_foot(p, size);
        }

else {
        /* modify size call set_head() */
        av->top = p;
}
```

Consolidated calamity



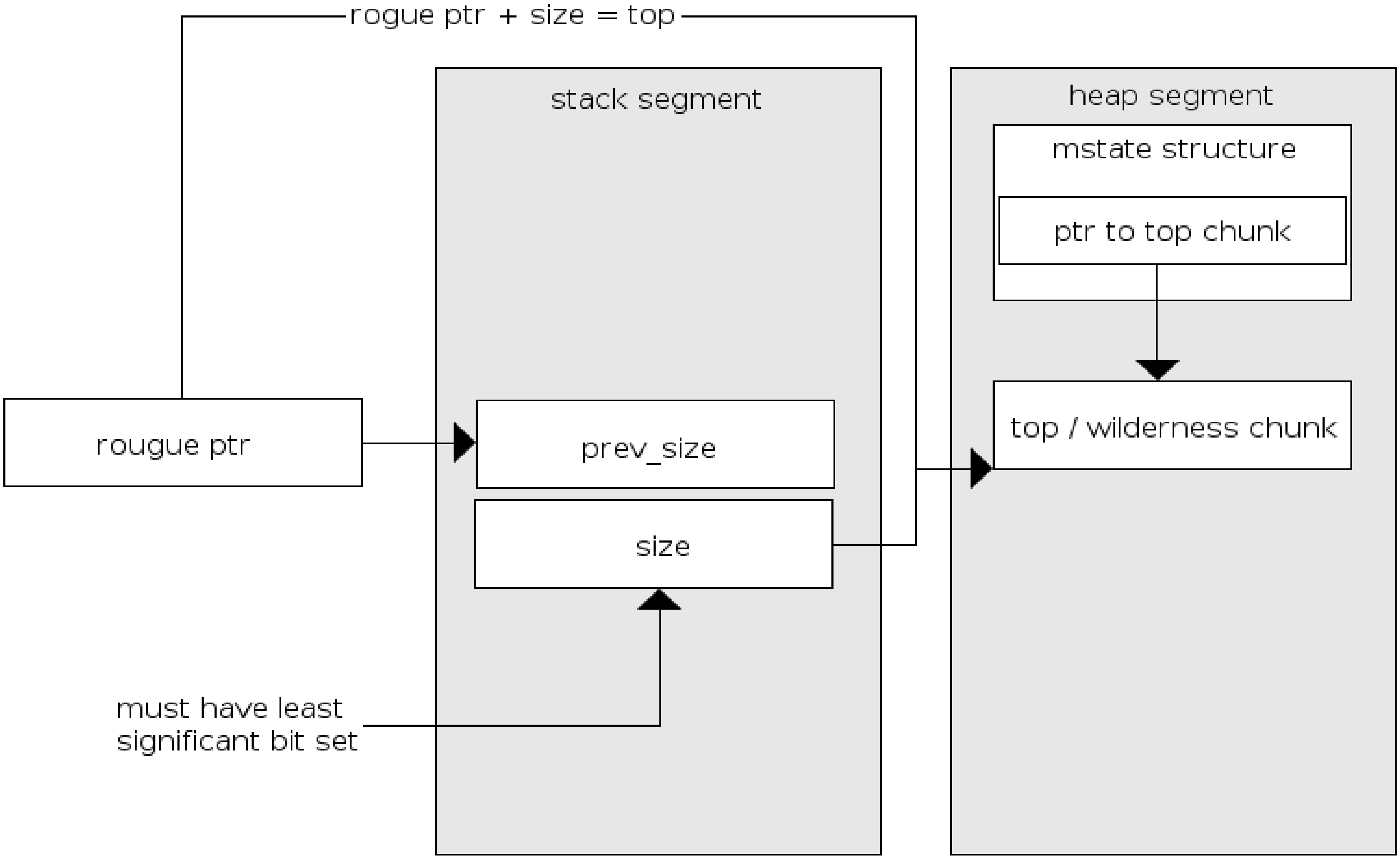
Consolidated calamity



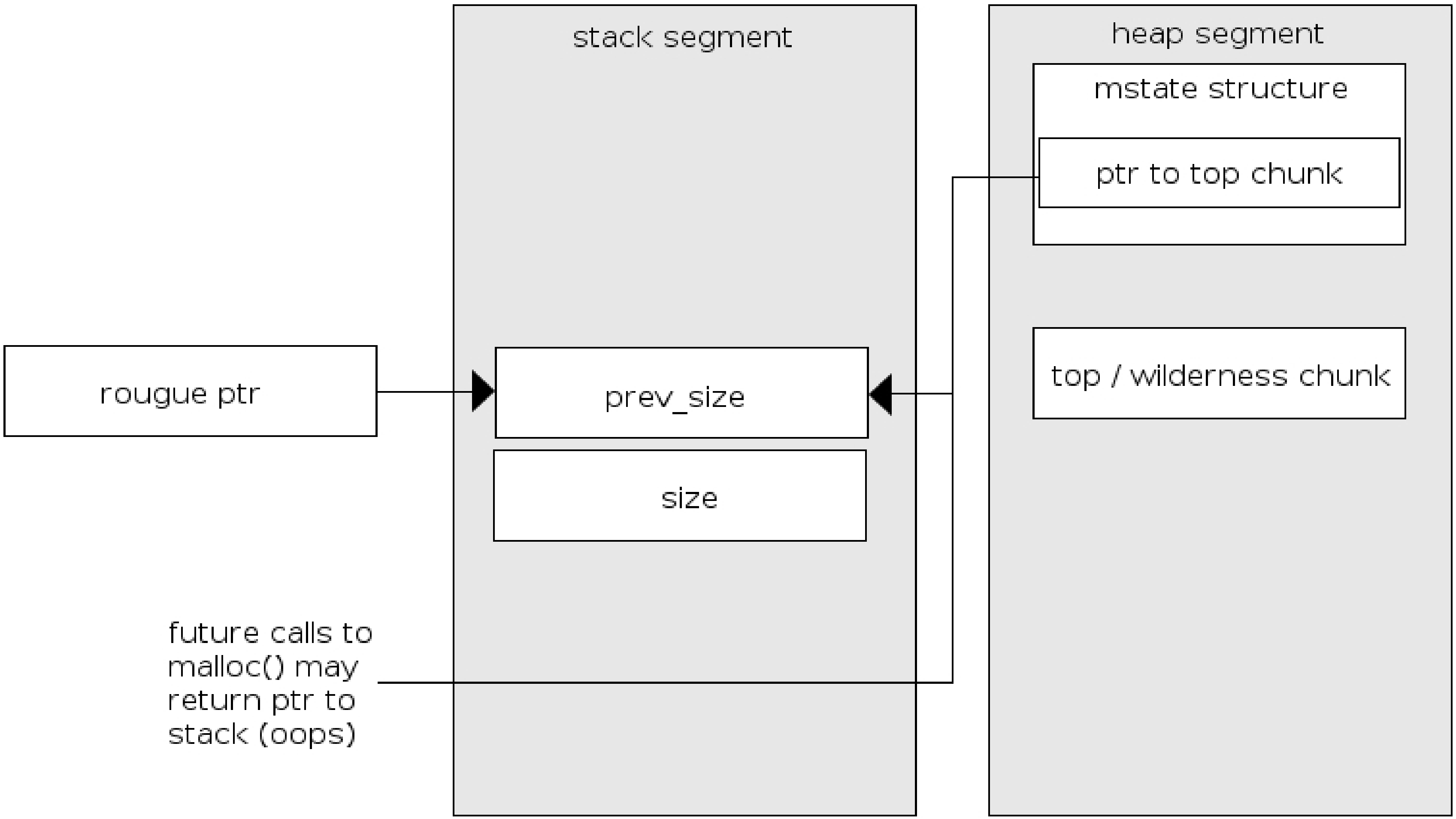
Consolidated calamity

- Can be used to turn on or off least significant bit at arbitrary address
- Addresses need not be aligned to any boundary
- Next chunk is only used in consolidation, not bin-walk loop
- Thus chaining multiple writes together is possible
- Somewhat difficult in practice however
- First technique is more useful than the second
- Second technique requires that size be dual purpose

Consolidated calamity



Consolidated calamity



Multi-threaded calamity

```
 /* der_get_oid() */
0:      data->components = malloc((len + 1) *
                              sizeof(*data>components));
        […]
        if (p[-1] & 0x80) {
1:              free_oid (
                return ASN1_OVERRUN;
        }

   /* decode_MechType() */
        fail:
2:              free_MechType(data);
                return e;
```

must have least
significant bit set

Multi-threaded calamity

- **At point 0 we have a malloc()**
- **At point 1 we have a free()**
- **At point 2 we have another free**
- **Concept is to get one thread somewhere in between point 1 and 2 before another thread is at point 0**
- **If accomplished**
  - Possible for the other thread to receive recently free()'d chunk of memory back
  - At point 2, after the chunk has been allocated again then it is double free()'d
  - However all checks are bypassed due to chunk being allocated at time of second free

Multi-threaded calamity

- **The problem:**
  - If both threads started at exactly the same time, how do you get one to lag behind the other

- **That’s not even considering potential issues server side**

- **Or delay on the network between the two connections**

- **We’re not going to consider that at the moment, the task is complex enough that we will presume a lab environment**

Multi-threaded calamity

- First idea:
- Get one thread inside first free
- Get other to wait on mutex at malloc()
- Using the mutex to help us win the race
- Not going to work :/
- malloc() calls pthread_mutex_trylock()
- pthread_mutex_trylock() won't block
- This potentially creates a new arena

Multi-threaded calamity

- First idea:
- Get one thread inside first free
- Get other to wait on mutex at malloc()
- Using the mutex to help us win the race
- Not going to work :/
- malloc() calls pthread_mutex_trylock()
- pthread_mutex_trylock() won’t block
- This potentially creates a new arena

OSPF

- Do we even really need to worry?
- Will use the following function to determine approximate clock ticks

```
/* IA-32 single processor/core */
void
get_time(struct timer_t *timer)
{
        __asm__ __volatile__(
                        "rdtsc                  \n"
                        "movl   %%edx, %0       \n"
                        "movl   %%eax, %1       \n"
                        : "=m" (timer->high), "=m" (timer->low)
                        :
                        : "%edx", "%eax"
        );
}
```

OSPF

- Tested to see how long it took to get from point 0 to point 1
- Used minimal data
- Ran tests 10,001 times
- Highest 379363 ticks
- Lowest 5668
- Average 13245.1429857014
- Rounded to 13,200
- Need to find a way to save ~13,200 ticks

OSPF

- In the code path there are multiple loops
  - apr_base64_decode_len()
  - apr_base64_decode ()
  - der_get_oid()
- By examining these functions we can find a shorter code path that yields the same results by slightly modifying our data

OSPF

- In the code path there are multiple loops
  - apr_base64_decode_len()
  - apr_base64_decode ()
  - der_get_oid()
- By examining these functions we can find a shorter code path that yields the same results by slightly modifying our data

OSPF

- apr_base64_decode_len()
- Simple while loop iterating through pointer to user data while it dereferences to a valid base-64 encoded character
- Ran tests 10,001 times
- Low of between 260 and 305 ticks depending on character used
- High was 688558-245953
- Average was between 602.832816718328 and 573.697930206979
- Fairly stable average of ~600 ticks per byte saved
- If we can cut up to eight character, this is approximately 4800 ticks saved

OSPF

- apr_base64_decode()
- Similar to apr_base64_decode_len(), series of simple loops
- After 10,001 tests
- High tick count of 4055
- Low of 1010
- Average per byte count being 1144.69163083692
- Rounded up to 1150, multiplied by eight (for each of the 8 bytes we can omit)
- Multiplied by 4800 ticks for the ticks saved in apr_base64_decode_len() yields 14000 ticks
- Higher than necessary savings of 13,200 ticks

OSPF

- **der_get_oid()**
- **Too different code paths in the loop depending on if the byte being processed is greater than 0x80**
- **If byte was less than 0x80**
  - 10001 tests
  - High of 4017 ticks
  - Low of 5 ticks
  - Average 150.049195080492
- **If byte was greater than 0x80**
  - 10001 tests
  - High of 1610 ticks
  - Low of 132
  - Average of 388.5800419958 (round to 390)

OSPF

- **Following averages for results:**

  apr_base64_decode_len(): 600 ticks

  apr_base64_decode(): 1150

  der_get_oid(): 150 or 390

- **Trying to save ~13200 ticks**
- **Can cut 6 characters out of input from first thread**
  - Alternate between >= 0x80 and < 0x80
  - Saves on average 12840 clock ticks
- **Can cut 7 characters**
  - All characters below 0x80
  - Saves on average 13300 clock ticks

How realistic?

- How realistic is tick counting?
- Decent- nothing entirely accurate and depends a lot on conditions
- Gives a decent idea of how long actual operations take to perform
- Provides decent metric for finding a slightly shorter path that yields same heap results
- Not incredibly reliable, is possible (and has been recreated in the lab)
- Especially useful on SMP/multi-core machines

LinuxThreads and caps

- LinuxThreads are especially something to look out for
- Didn’t properly implement POSIX
- Different threads in a given process could have different user ids
- Even while LinuxThreads is rarely in use …
- Linux capabilities are more common (anymore)
- Capabilities are also per-thread
- One thread can invalid the heap reference of another
- Can cause privilege escalation even if the code in the privileged thread is flawless

Conclusions

- **More than one way to accomplish things**
- **At least two more ways to exploit these conditions listed**
  - Time constraints kept them from being presented
- **Heap is persistent and is shared, this is something that can be exploited**
- **Threading provides an interesting method for arranging code into advantageous sequences**
- **Slides are hard to fit code onto :/**

Questions?