Heap Analysis and Exploitation

Evergreen Hacker Club

Stack vs. Heap Structures: Overview

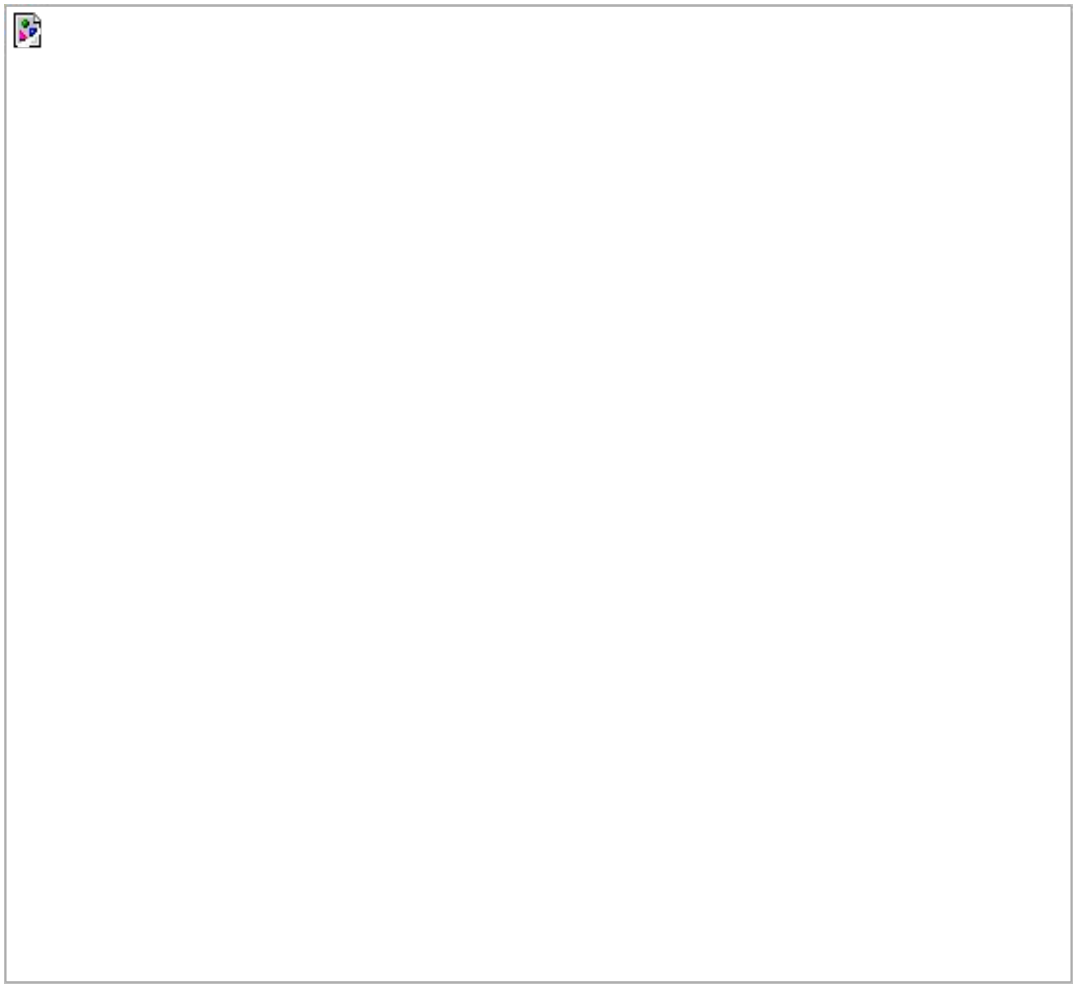
Remember the **stack**:

- A data structure used at runtime to keep track of *arguments passed to functions, local variables & return addresses*.
- Memory allocations are known at *compile time*.
- Allocations and deallocations *handled automatically*.

Now, introducing the **heap**:

- Once again, a runtime data structure, but used to keep track of *structs, objects, big buffers, larger things in general*.
- Memory allocations are dynamic and _not_ fixed, only known during *runtime*.
- Allocations and deallocations are *handled by the programmer*.

Stack vs. Heap Structures: Segment growth



Heap grows from *lower to higher* memory addresses.

While the stack grows from *higher to lower* memory addresses.

Free the Malloc

The two functions that make up the foundation of heap manipulation are:

```
void *malloc(size_t size);
```

When **malloc**() is called for the *first* time in a thread it will:

- allocate a reasonable amount of memory.
- create a heap segment or equivalent.
- return to the caller a pointer to a memory region (chunk) of size_t size

If it is not the first time it will:

- simply return a pointer to a region (chunk) within the current heap segment (or equivalent) of suitable size.

```
void free(void *ptr);
```

Sets a memory region previously returned via **malloc**() as not in use.

What is a “Chunk”?

Chunks are the smallest unit of memory administration in dynamic allocation.

Their implementation as per glibc2.19:

```
struct malloc_chunk {
     INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if free). */
     INTERNAL_SIZE_T size;      /* Size in bytes, including overhead. */
     struct malloc_chunk* fd;   /* double links -- used only if free. */
     struct malloc_chunk* bk;

     /* Only used for large blocks: pointer to next larger size. */
     struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
     struct malloc_chunk* bk_nextsize;
};
```

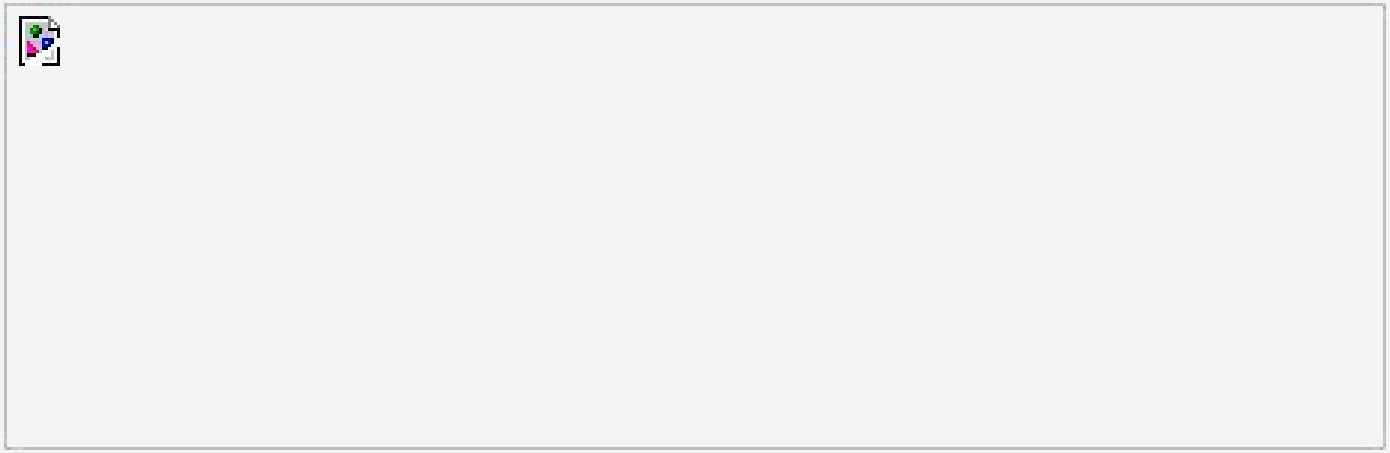
Chunks of malloc()

When free’d or malloc’d, the internal structure of a chunk will change accordingly.

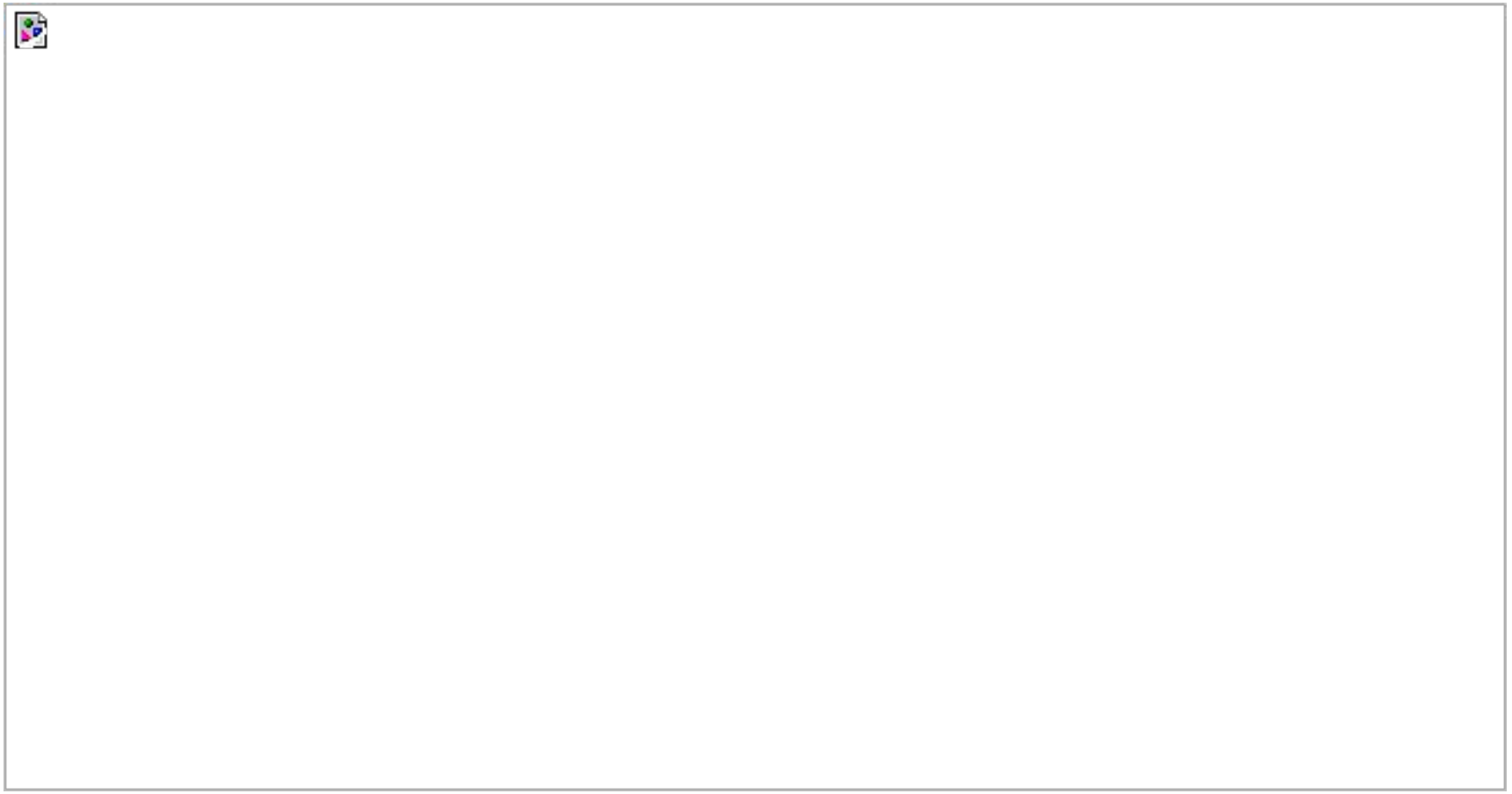
- prev_size is only initialized if the chunk is free.
- When a chunk is freed, they are inserted into a linked-list of free chunks.
  - fd and bk are then initialized
  - Then linked to a particular chunk to others within the same list.

The 3 least-significant-bits (LSB) represent the chunk flags:

- PREV_INUSE (P) *bit set when previous chunk is allocated*
- IS_MMAPPED (M) *bit set when chunk is being mmapp’d*
- NON_MAIN_ARENA (N) *bit set when chunk does not belong to heap segment*



Chunk States



On Allocator Implementations

The heap must be created at runtime by some algorithm. The various implementations of this algorithm are called **allocators** & there are quite a few:

- dlmalloc
- ptmalloc
- tcmalloc
- jemalloc
- nedmalloc
- Hoard

The allocators allocate memory dynamically and administrate that memory.

Heaps go way deep & much can depend on implementation! (Arenas, binning, chunk coalescing, fragmentation…)

In this demonstration we will be mainly concerned with glibc 2.19 which uses a heap implementation based on **ptmalloc2**.

It is said to be very fast, with low fragmentation, & thread-safe.

Heap Exploitation: Overflows

Overflows on the heap are not so much different than the classical stack smash.

Simply, one corrupts data by overflowing from one chunk into another (or what-have-you).

Lots of interesting things are stored on the stack, functions pointers as struct fields are a good example.

```
struct toystr {
  void (* message)(char *);
  char buffer[20];
}
```

Anything that handles data you've corrupted is a viable attack surface.

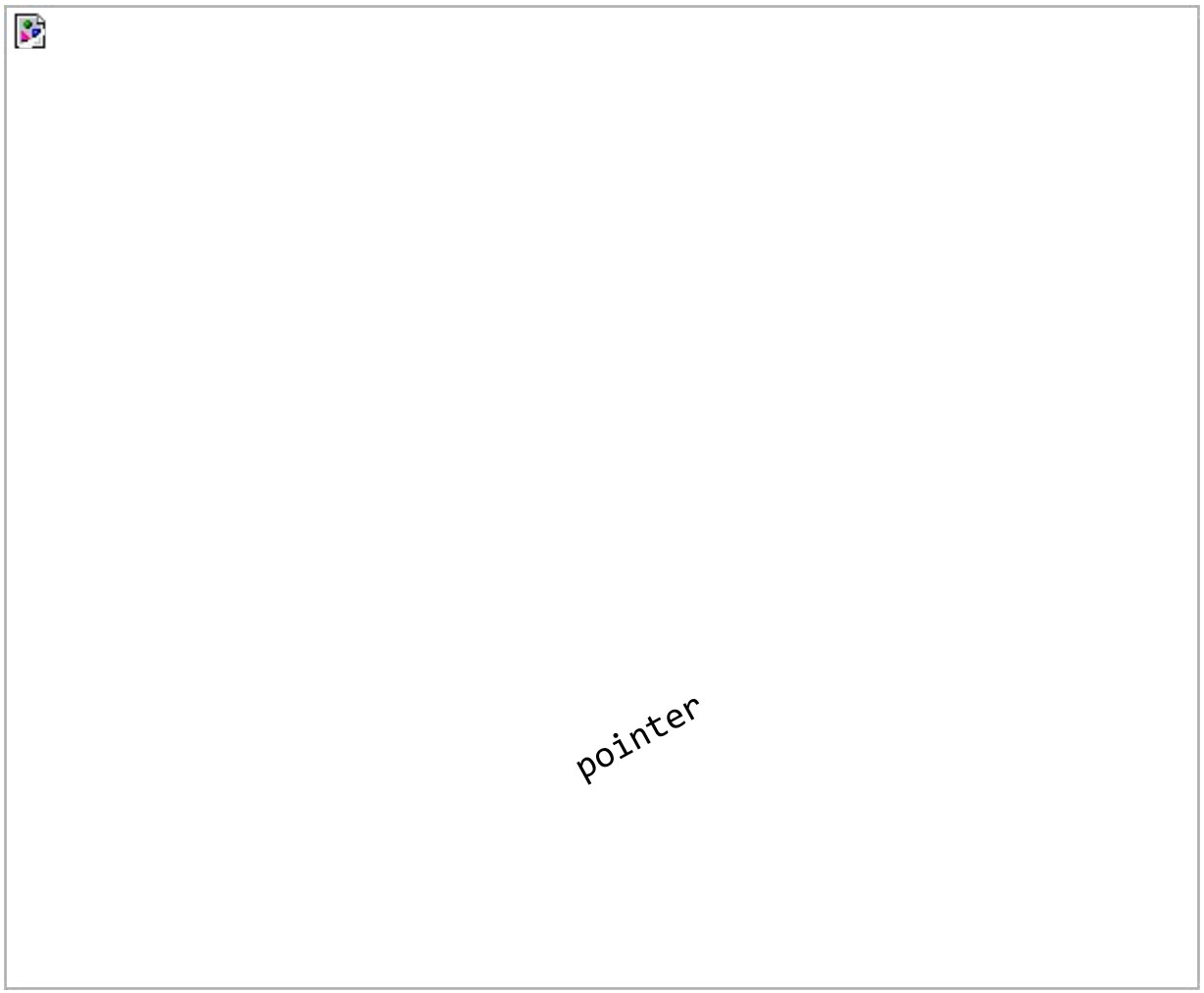
Heap Exploitation: UAF

Use-After-Free (a.k.a: UAF) is a class of vulnerability that occurs when some memory is referenced after it has been freed which can cause a program to crash, use unexpect values, or execute code.

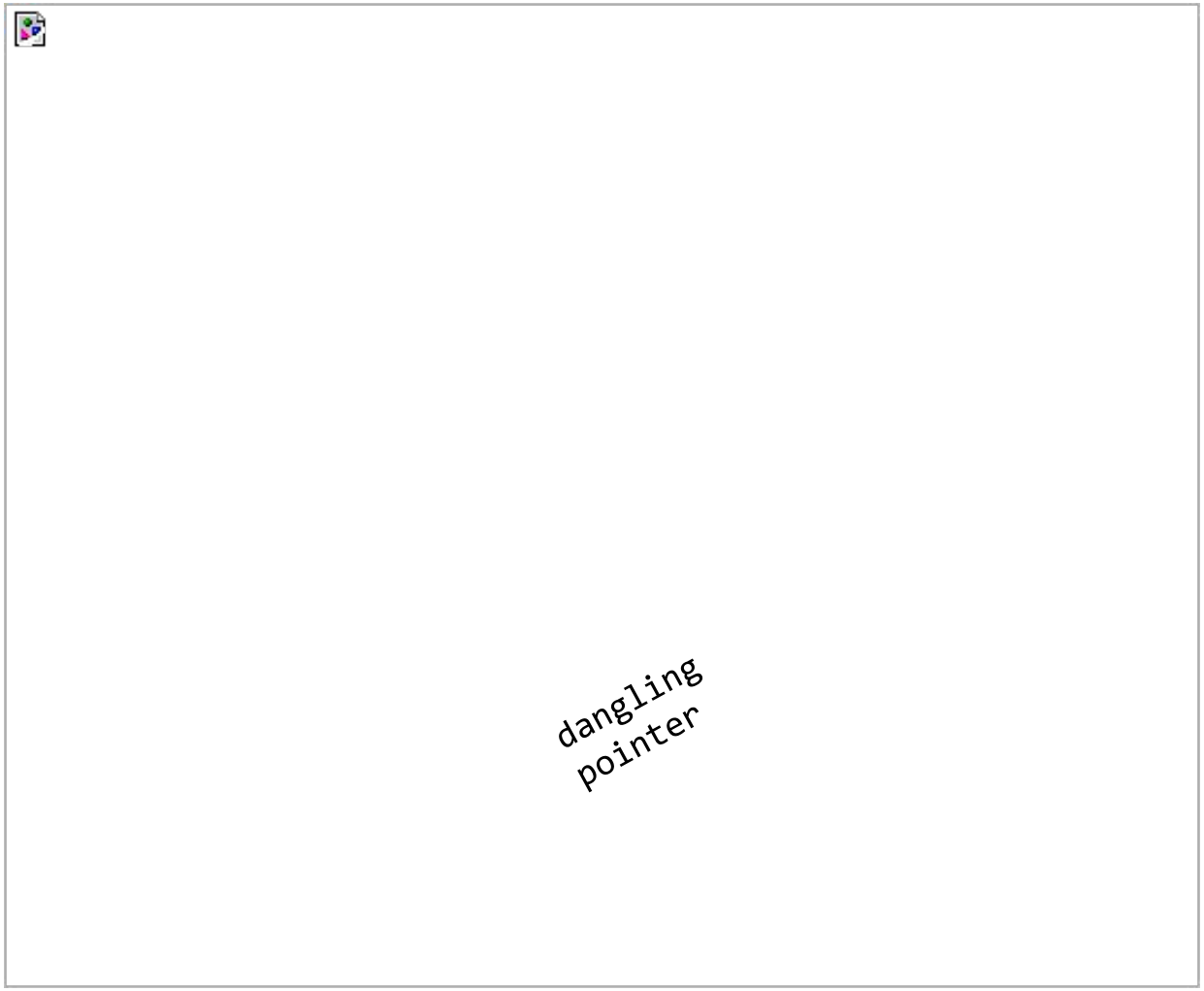
These leftover references are often called “dangling pointers”.

This kind of thing usually happens on the heap and within complex programs such as web browsers. Check out CVE-2016-1961 in firefox, for example.

Heap Exploitation: UAF

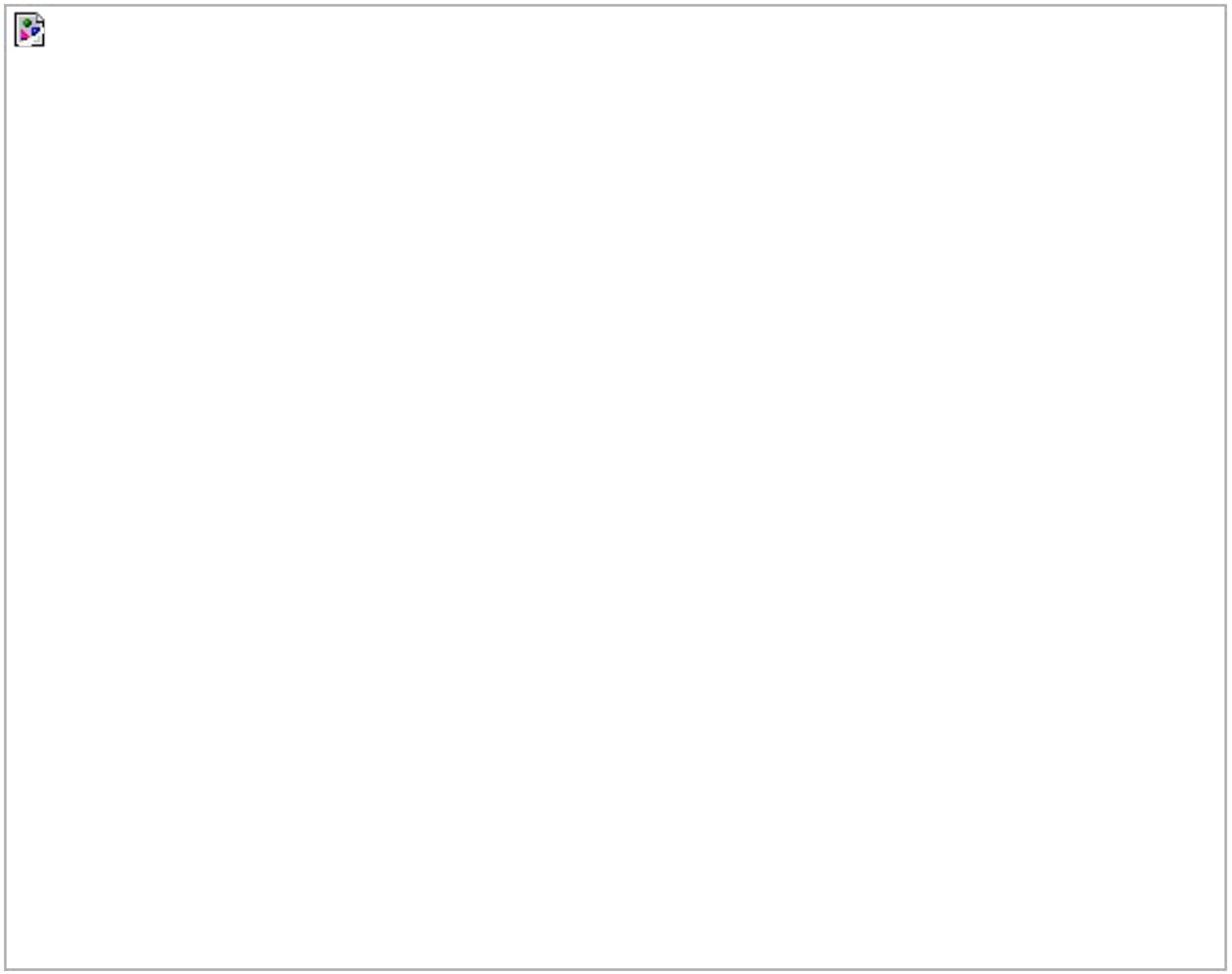


Heap Exploitation: UAF



Two free()’d chunks, leaving a dangling pointer.

Heap Exploitation: UAF



newly malloc()’d
chunk overwriting
those just free’d

Heap Exploitation: UAF

Usually, in order to exploit a use-after-free vulnerability, you'll have to malloc a different structure over the one you just freed.

Say, for example we have:

```
struct toystr {
  void (* message)(char *);
  char buffer[20];
}

struct person {
  int favorite_number;
  int age;
  char name[16];
}
```