Overview of GLIBC heap exploitation techniques

Overview of current GLIBC heap exploitation techniques up to GLIBC 2.34, including their ideas and introduced mitigations along the way

Published 13 Feb 2022 - 43 min read

By ⍰⍰⍰⍰⍰

```
pwndbg> ptype /o mchunkptr
type = struct malloc_chunk {
/*    0      |     8 */    size_t mchunk_prev_size;
/*    8      |     8 */    size_t mchunk_size;
/*   16      |     8 */    struct malloc_chunk *fd;
/*   24      |     8 */    struct malloc_chunk *bk;
/*   32      |     8 */    struct malloc_chunk *fd_nextsize;
/*   40      |     8 */    struct malloc_chunk *bk_nextsize;

                            /* total size (bytes):   48 */
                          } *
pwndbg> ptype /o heap_info
type = struct _heap_info {
/*    0      |     8 */    mstate ar_ptr;
/*    8      |     8 */    struct _heap_info *prev;
/*   16      |     8 */    size_t size;
/*   24      |     8 */    size_t mprotect_size;
/*   32      |     0 */    char pad[];

                            /* total size (bytes):   32 */
                          }
pwndbg> ptype /o mstate
type = struct malloc_state {
/*    0      |     4 */    __libc_lock_t mutex;
/*    4      |     4 */    int flags;
/*    8      |     4 */    int have_fastchunks;
/* XXX  4-byte hole  */
/*   16      |    80 */    mfastbinptr fastbinsY[10];
/*   96      |     8 */    mchunkptr top;
/*  104      |     8 */    mchunkptr last_remainder;
/*  112      |  2032 */    mchunkptr bins[254];
/* 2144      |    16 */    unsigned int binmap[4];
/* 2160      |     8 */    struct malloc_state *next;
/* 2168      |     8 */    struct malloc_state *next_free;
/* 2176      |     8 */    size_t attached_threads;
/* 2184      |     8 */    size_t system_mem;
/* 2192      |     8 */    size_t max_system_mem;

                            /* total size (bytes): 2200 */
                          } *
```

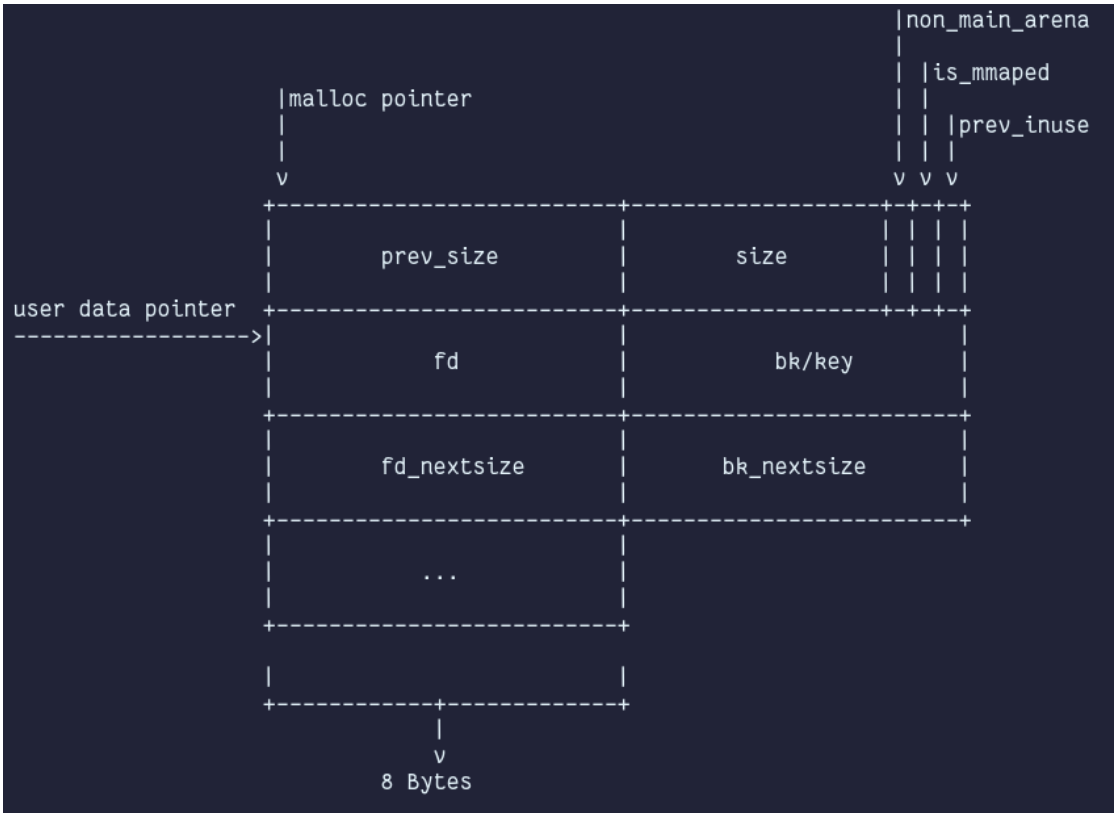
pwndbg>

This post will aim at giving a general overview of publicly found GLIBC heap exploitation techniques. Actual exploitation will be left as an exercise for the reader. The remainder of this post will be divided in 2 parts: Patched and unpatched techniques. The latter category is to the best of my knowledge. I will provide links to patches as good as I can when a technique has been rendered unusable. Unusable should be interpreted with caution, as I'm only talking original attack vector here. Someone, somewhere, might find a new way to exploit a dead technique again!

Basics

To get started, let's get on the same page with some terminology

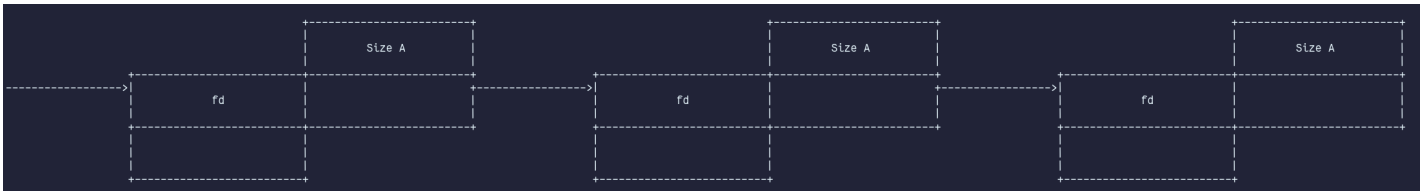
Chunks



Fastbin

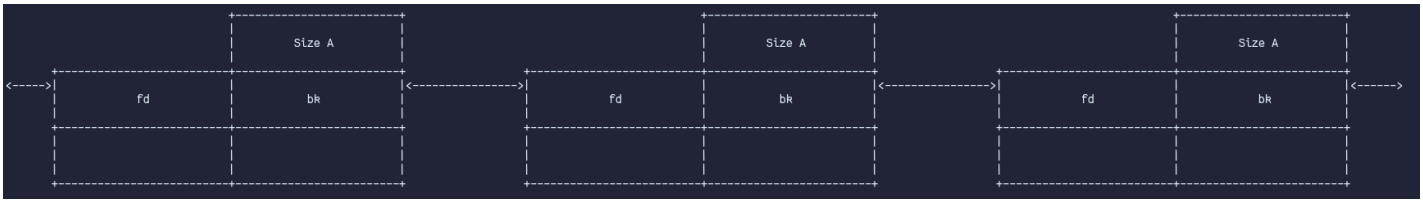
The fastbins are a collection of singly linked, non-circular lists, with each bin in the fastbins holding chunks of a fixed size ranging from **0x20** to **0xb0**. The head of the fastbin is stored in its arena, while the 1st qword of a fastbin chunk's userdata is

fastbin is stored in its arena, while the 1st qword of a fastbin chunk's userdata is repurposed as a forward pointer (fd) that points to the next chunk in a specific fastbin. A fd of NULL indicates the end of the list. The fastbins are LIFO structures. Chunks are directly linked into the corresponding fastbin if the tcache is present and the appropriately sized tcachebin is already full. When requesting a chunk in fastbin range, the search is prioritized as follows: tcachebin→fastbin→unsortedbin→top_chunk.



Smallbin

Smallbins, contrary to the fastbins are doubly linked, circular lists (62 in total) that each hold chunks of a specific size ranging from **0x20** to **0x3f0**, overlapping the fastbins. The head of each smallbin is located in the arena. Linking into a smallbin only happens via the unsortedbin, when sorting occurs. The list metadata (forward and backward pointers) are stored inline in the freed chunks. The smallbins are unlike the fastbins FIFO structures.



Unsortedbin

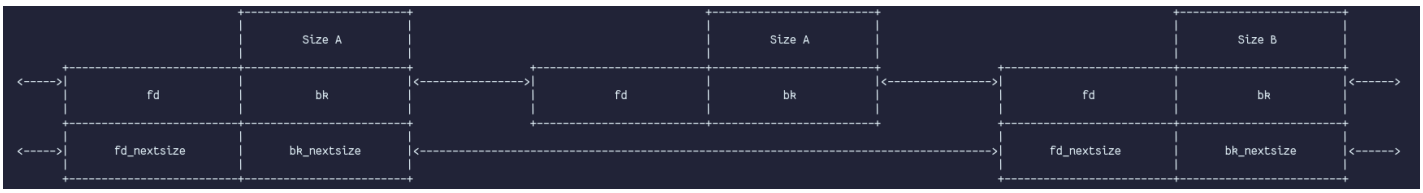
An unsortedbin is a doubly linked, circular list. It holds free chunks of any size. The head and tail of the unsortedbin are located within the arena. Inline metadata in the form of a forward and backward pointer are used to link chunks. Chunks are directly linked into the head of the unsortedbin if they are either not in fastbin range (>= 0x90) or in case of the tcache being present if the tcache is full and if the chunk is outside tcache size range (>= 0x420). The unsortedbin is only searched (from tail to head) after the tcache, fastbins and smallbins, but before the largebins. If during the search an exact fit is encountered is allocated, if it's not an exact fit the free chunk is sorted into the appropriate small- or largebin. If a chunk is bigger than the requested chunk, the remaindering process will take place and the remainder will be linked into the head of the unsortedbin again.

take place and the remainder will be linked into the head of the unsortedbin again.

Largebin

Largebins, are also doubly linked, circular lists. Unlike the fast- or smallbins, each largebin does not just hold a fixed size but a range of sizes (e.g. 0x400 to 0x430 sized chunks are linked into the same largebin). The head of the largebin also resides in the arena. Inline metadata in the form of forward and backward pointers is also still present here. A major difference is the occurrence of two additional pointers (fd_nextsize and bk_nextsize), which are only present in the first element of each largebin. These `nextsize` pointers form another doubly linked, circular list and point to the next/previous largebin. Linking into a largebin only occurs via the arena's unsortedbin when sorting occurs, similar to

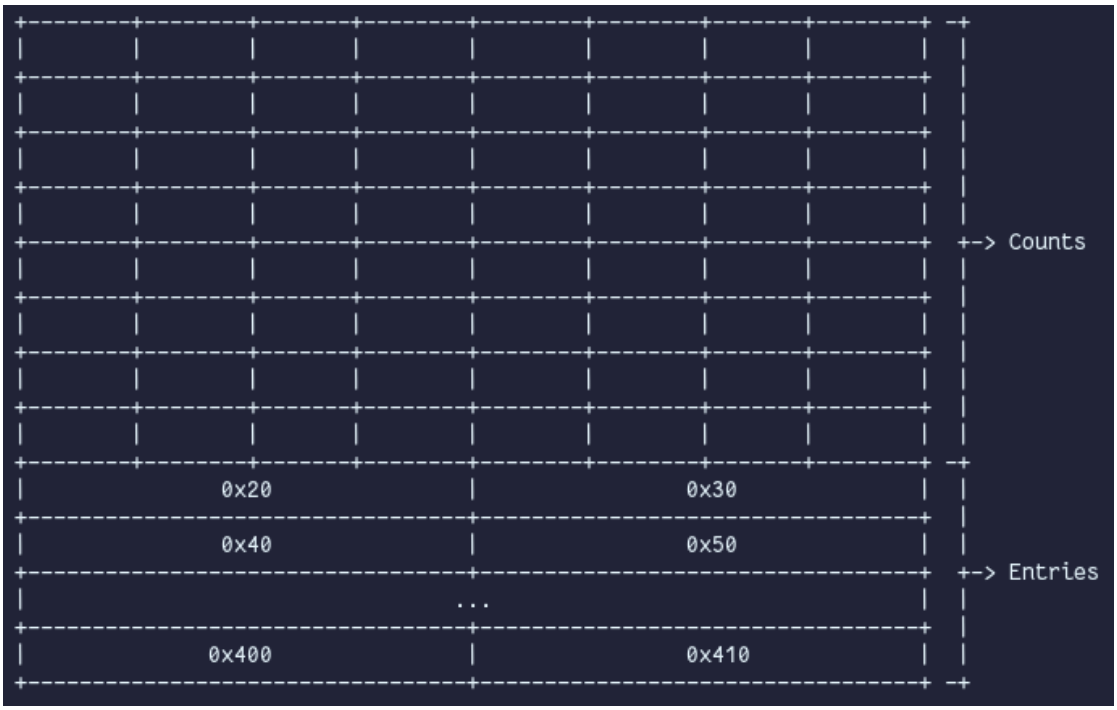
how it's handled in the smallbins. During scanning for an exact fit or larger sized chunk to serve, the appropriately sized largebin is searched from back to front. Furthermore, malloc only allocates a chunk with the `nextsize` pointers (skip list pointers) when it's the last chunk in a largebin to avoid having to change multiple pointers. Non-exact fit chunks are exhausted/remaindered when allocated from a largebin. The `last_remainder` field is not set!



Tcache

Since GLIBC >= 2.26 each thread has its own tcache which sits at the very beginning of the heap. It kind of behaves like an arena, just that a tcache is thread-specific. There are 64 tcachebins with fixed sizes, with a preceding array that keeps count about how many entries each tcachebin has. Since GLIBC >= 2.30 each count is the size of a word, before that, it was a char*. The tcachebins behave similarly to fastbins, with each acting as the head of a singly linked, non-circular list of chunks of a specific size. By default, each tcachebin can hold 7 free chunks (which can be tweaked with the `tcache count` variable

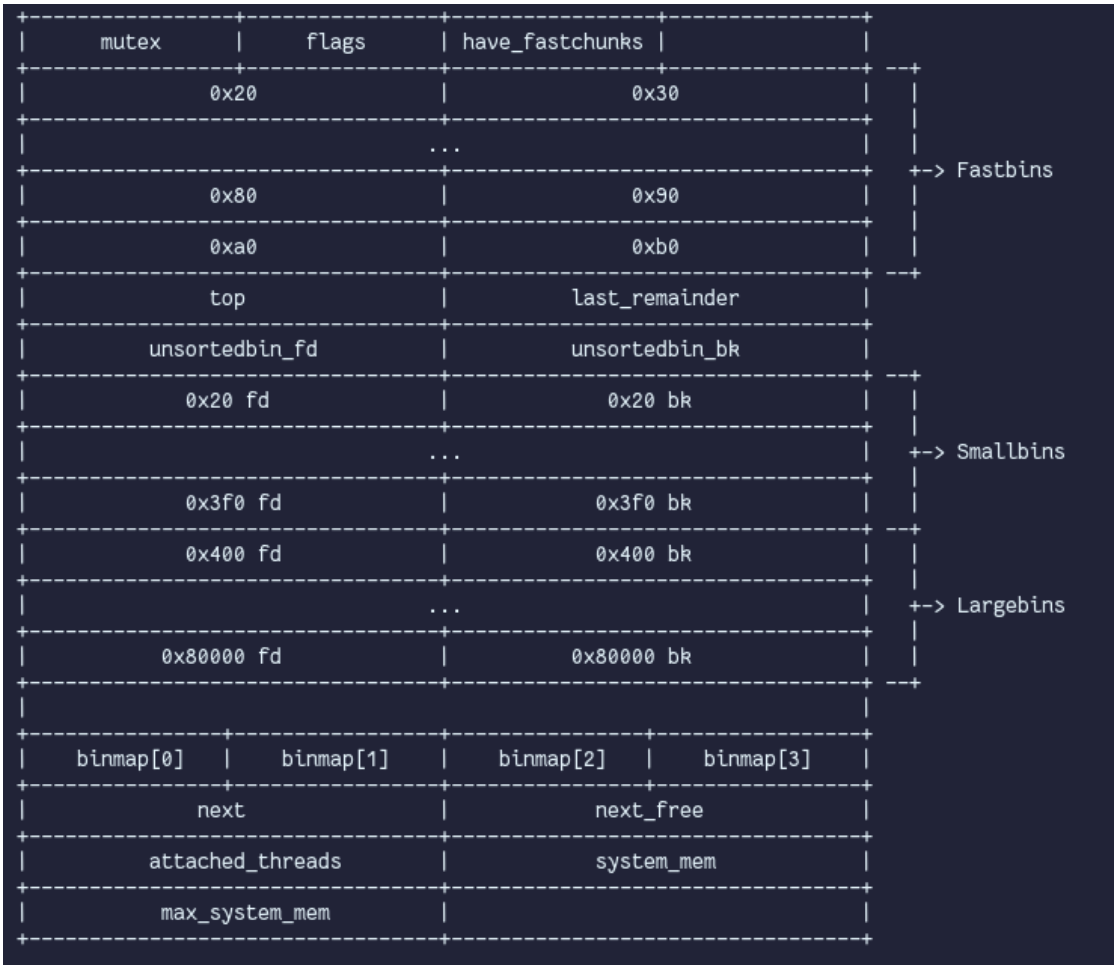
in the malloc_par struct). When a tcachebin is full, a newly freed chunk is treated as if there is no tcache around. Allocating from a tcachebin takes priority over every other bin. Phenomena with tcaches around is tcache dumping, which occurs when a thread allocates a chunk from its arena. When a chunk is allocated from the fast-/smallbins, malloc dumps any remaining free chunk in that bin into their corresponding tcachebin until it is full. Similarly, when an unsortedbin scan occurs, any encountered chunk that is an exact fit will also be dumped into the tcachebin. If the tcachebin is full and malloc finds another exact-fitting chunk in the unsortedbin, that chunk is allocated. If the unsortedbin scan is completed and >=1 chunk(s) were dumped into the tcachebin, a chunk is allocated from that tcachebin.



Arena

An arena is not more than a state struct for malloc to use. An arena primarily consists of the bins, among a few other noteworthy fields. The mutex field serializes access to an arena. The flag field holds information about whether an arena's heap memory is contiguous. The `have_fastchunks` boolean field indicates the fastbins may not be empty.

The `binmap` is a bitvector that loosely represents which of an arena's smallbins & largebins are occupied. It's used by malloc to quickly find the next largest, occupied bin when a request could not be serviced. Binmap searches occur after an unsuccessful unsortedbin or largebin search. The next field is a pointer to a singly linked, circular lost of all arenas belonging to this arena. Next_free is also a singly linked but non-circular list of free arenas (arenas with no threads attached). Attached_threads is just the number of threads concurrently using this arena. System_mem holds the value of the total writable memory currently mapped by this arena. Max_system_mem is the largest amount of writable memory this arena had mapped at any point.



Unlinking

During allocation/free operations, chunks may be unlinked from any of the free lists

During allocation/free operations, chunks may be unlinked from any of the free lists (bins) they were stored in. The unlinked chunk is often referred to as "victim" in the malloc source. Unlinking in fastbins/tcache is straightforward, as they're singly linked LIFO lists. The process only involves copying the victim's fd into the head of the list, which then points to the next free chunk following our victim, effectively removing the victim from the list. There's a partial unlinking process for bins allocated from the small-/unsortedbin. The victim's chunk bk is followed to the previous chunk. There, the address of the head of the bin is copied into the chunk's fd. Finally, the victim chunk's bk is copied over the bk of the head of the bin. Lastly, there's also the notion of a full unlink that occurs when a chunk is consolidated into another free chunk or is allocated from the largebin. In the process, the victim chunk's fd is followed and the victim bk is copied over

to the destination bk. Next, the victim chunk's bk is followed and the victim fd is copied over the destination fd.

Remaindering

Remaindering is simply the term referring to splitting a chunk into two parts, the requested size one and the remainder. The remainder is linked into the unsortedbin. Remaindering can occur at three phases: During allocation from the largebins, during binmap search, and from a last remainder during unsortedbin scanning.

Exhausting

Simply the term for when an N sized chunk is requested and only an N+0x10 sized chunk is found, e.g. in the unsortedbin. A remainder of 0x10 is an invalid chunk size, so malloc will just "exhaust" the whole N+0x10 chunk and allocate that one as is.

Consolidation

Consolidation is the process of merging at least two free chunks on the heap into a larger free chunk to avoid fragmentation. Consolidation with the top chunk is also worth noting.

Malloc hooks

These hooks have been a hell of helpful in past exploitation techniques, but due to them enabling numerous techniques showcased below, they have been removed in GLIBC >=

2 .34.

1. `__malloc_hook` / `__free_hook` / `__realloc_hook` : Located in a writable segment within the GLIBC. These pointers, defaulting to 0 but when set, result in instead of the default GLIBC malloc/realloc/free functionality being called the function pointed to the value being set within these hooks being called on an allocation/free.
2. `__after_morecore_hook` : The variable `__after_morecore_hook` points at a function that is called each time after `sbrk()` was asked for more memory.
3. `__malloc_initialize_hook` : The variable `__malloc_initialize_hook` points to a function that is called once when the malloc implementation is initialized. So

   overwriting it with an attacker controlled value would just be useful when malloc has never been called before.
4. `__memalign_hook` : When set and `aligned_alloc()` , `memalign()` , `posix_memalign()` , or `valloc()` are being invoked, these function pointed to by the address stored in this hook is being called instead
5. `_dl_open_hook` : When triggering an abort message in GLIBC, typically `backtrace_and_maps()` to print the stderr trace is called. When this happens, `__backtrace()` and within that code `init()` is called, where `__libc_dlopen()` and `__libc_dlsym()` are invoked. The gist is that IFF `_dl_open_hook` is not NULL, `_dl_open_hook` → `dlopen_mode` and `_dl_open_hook` → `dlsym` will be called. So, the idea is to just overwrite _dl_open_hook with an address an attacker controls where a fake vtable function table can be crafted

Patched techniques

Let's dive into the known patched techniques first, as this is where it all started.

1. House of Prime

Gist: Corrupt the fastbin maximum size variable (holds the size of the largest allowable fastbin for free to create), which under certain circumstances allows an attacker to hijack the arena structure, which in turn allows either to return an arbitrary memory chunk or

direct modification of execution control data (as shown in the malloc maleficarium).

Applicable until: < 2.4

Root cause: Overflow

GLIBC VERSION	PATCHES
2.3.4	Added safe unlink
2.3.4	Check the next chunk on free is not beyond the bounds of the heap
2.3.4	Check next chunk's size sanity on free()
2.3.4	Check chunk about to be returned from fastbin is the correct size, same for the
2.3.4	Ensure a chunk is aligned on free()

Idea: The technique requires an attacker to have full control over two chunks to tamper with, 2 calls to `free()` and one call to `malloc()`. Let's assume we have 4 chunks A, B, C, and D (guard). First, an overflow from chunk A into the size field of Chunk B is used to set chunk B's size to 8. When freeing this chunk, internally this results in an index underflow returning the address of `max_fast` instead of the address of the fastbin to free this chunk into. Ultimately, this results in an overwrite of `max_fast` with a large address. The next step in this technique involves overwriting the `arena_key` variable when calling free on chunk C. A precondition here is that the arena_key is stored at a higher address than the address of the thread's arena. Otherwise, the attempted overwrite won't work. Before freeing chunk C, we need to overwrite its size field (with a set prev_inuse bit). The size we need is `(((dist_arena_key_to_fastbins0 / sizeof(mfastbinptr) + 2 ) << 3) + 1)`. When setting the size of chunk C to that value and freeing it after, `arena_key` will hold the address of chunk C. Overwriting `arena_key` is a necessity as it will point the

programs' arena to the set value, letting us hijack the whole arena itself as we control the heap space.

Notes: In these old versions of GLIBC, `fastbins[-1]` was pointing to `max_fast` as they were both part of the arena and right next to each other in memory. Nowadays, `max_fast` is called `global_max_fast`, which is a global variable in recent GLIBC implementations. For this reason, this technique cannot be leveraged as is anymore (besides the patches). However, the idea of overwriting `max_fast_global` is still applicable with the right preconditions! Overwriting the (global_)max_fast variable qualifies large chunks for fastbin insertion when freed. Freeing a large chunk and linking it into the fastbin for its size will write the address of that chunk into an offset from the

first fastbin. This allows an attacker to overwrite an 8 byte aligned qword with a heap address. The formula to calculate the size of a chunk needed to overwrite a target with its address when freed is *chunk_size = (delta * 2) + 0x20,* with delta being the distance between the first fastbin and the target in bytes.

2. Unsafe unlink

Gist: Force the unlink macro during consolidation to process attacker controlled fd/bk pointers.

Applicable until: < 2.3.4

Root cause: Overflow

GLIBC VERSION	PATCHES
2.3.4	Added safe unlink

Idea: The basic idea is to force the unlink macro, which kicks in during chunk consolidation, to process attacker-controlled fd/bk pointers. This could be forced when being able to allocate two chunks N and N+1, where N+1 is out of fastbin range (as it would just get thrown into one of the fastbins in the next step). Then we need to write to chunk Ns user data, providing some forged bk and fd pointers in the 1st and 2nd qword.

In the same write we would want to provide a fake `prev_size` field in the last qword of chunk N that matches chunk Ns size followed by an overflow into chunk N+1 to clear its `prev_inuse` flag indicating that chunk N (located before chunk N+1) is not in use anymore, even if it still is. When we free chunk N+1 now the chunk consolidation kicks in. When chunk N+1 is cleared, the `prev_inuse` flag is checked and backward consolidation is attempted, as the flag is not present. For that, the `prev_size` field is followed to locate the preceding chunk N on the heap. Then a reflected write occurs as the victims bk (chunk N) is copied over the bk of the chunk pointed to by the victims (chunk N) fd and the victims fd is written over the fd of the chunk pointed to by the victims bk.

3. House of Mind (Original)

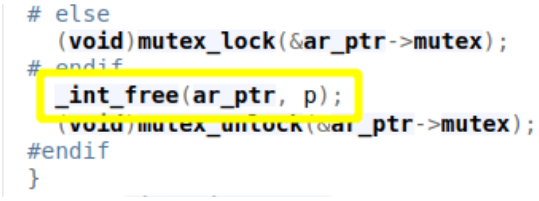
Gist: Craft a non-main arena within the main arena and trick free into processing a chunk with the `NON_MAIN_ARENA` bit set, resulting in a target overwrite of sensitive values.
Applicable until: < 2.11

Root cause: Overflow

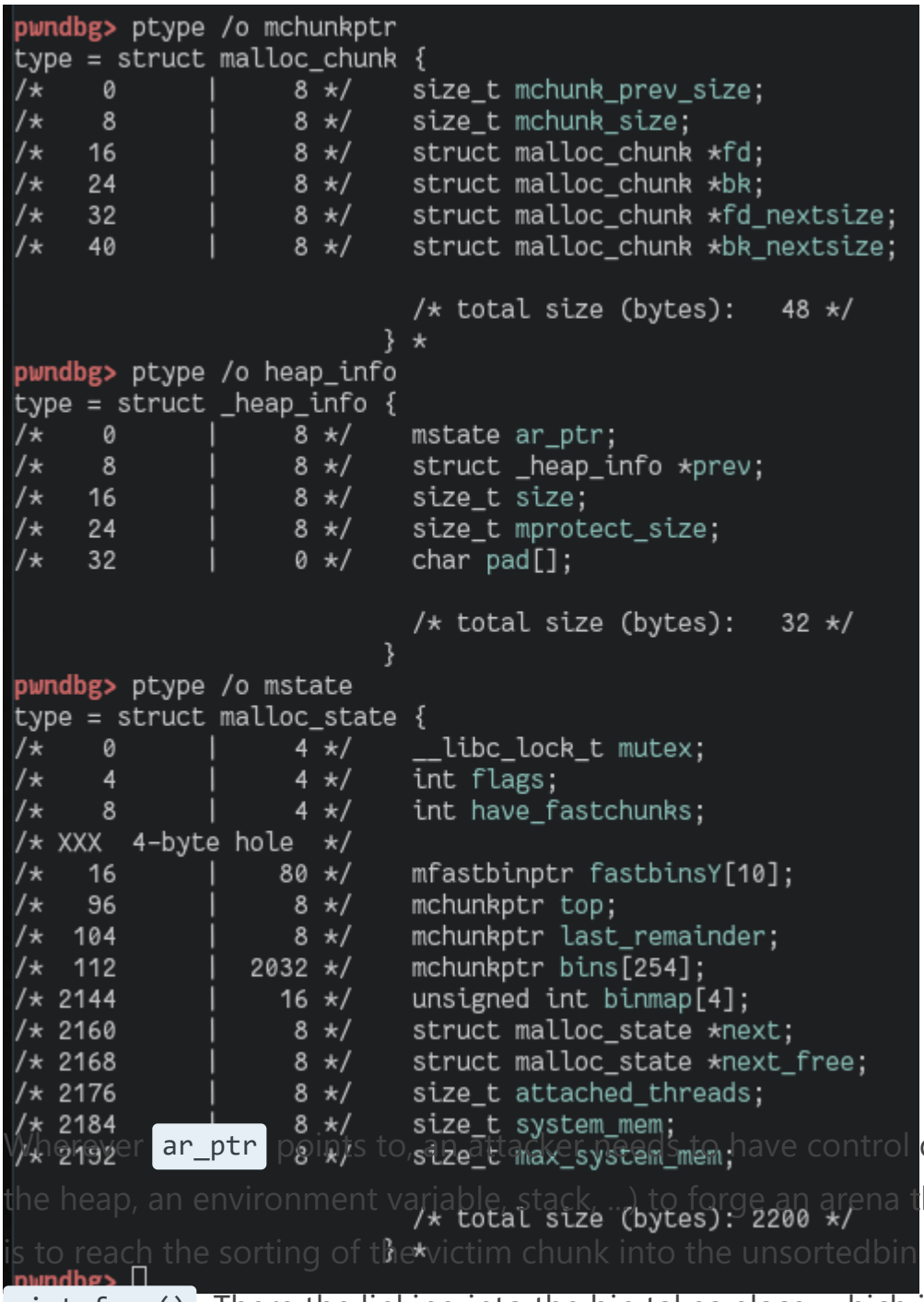
GLIBC VERSION	PATCHES
2.11	Multiple integrity checks for allocations from different bins as well as sorting fr

Idea: The idea of the House of Mind technique is to make `free` believe that the supplied chunk does *not* belong to the main arena (by setting the `NON_MAIN_ARENA` flag, e.g. due to an overflow bug). In these older GLIBC versions, a call to `free` triggers a wrapper function called `public_fREe()`:

```
void
public_fREe(Void_t* mem)
{
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (__malloc_ptr_t, __const __malloc_ptr_t)
    = force_reg (__free_hook);
  if (__builtin_expect (hook != NULL, 0)) {
    (*hook)(mem, RETURN_ADDRESS (0));
    return;
  }

  if (mem == 0)                              /* free(0) has no effect */
    return;

  p = mem2chunk(mem);

#if HAVE_MMAP
  if (chunk_is_mmapped(p))                       /* release mmapped memory. */
  {
    /* see if the dynamic brk/mmap threshold needs adjusting */
    if (!mp_.no_dyn_threshold
        && p->size > mp_.mmap_threshold
        && p->size <= DEFAULT_MMAP_THRESHOLD_MAX)
      {
        mp_.mmap_threshold = chunksize (p);
        mp_.trim_threshold = 2 * mp_.mmap_threshold;
      }
    munmap_chunk(p);
    return;
  }
#endif

  ar_ptr = arena_for_chunk(p);
#ifdef ATOMIC_FASTBINS
  _int_free(ar_ptr, p, 0);
#else
# if THREAD_STATS
  if(!mutex_trylock(&ar_ptr->mutex))
    ++(ar_ptr->stat_lock_direct);
  else {
    (void)mutex_lock(&ar_ptr->mutex);
    ++(ar_ptr->stat_lock_wait);
  }
```



This function gets a chunks' user data address. `mem2chunk()` will calculate the start of the supplied chunk. The resulting address is supplied to the `arena_for_chunk()` function. Depending on whether the `NON_MAIN_ARENA` bit is set in the chunks' size field, this either returns the location of the main arena or a macro called `heap_for_ptr(chunk)->ar_ptr` is invoked. `heap_for_ptr()` ultimately just does `ptr & 0xFFF00000` and returns the result. In short, when a non-main arena heap is created, it is always aligned to a multiple of `HEAP_MAX_SIZE` (1 MB). Just setting the `NON_MAIN_ARENA` bit would make the result point back to the main_arena, where a `heap_info` structure is then expected, as `ar_ptr` is the first member of such a struct. This would typically fail, as there won't be any such data at this address. The idea is now that we force so many allocations on the heap that eventually we can free a victim chunk with a `NON_MAIN_ARENA` bit set so that the returned address to the `heap_info` struct still actually resides in the main heap but overlapping attacker controlled data. When that happens, an attacker can provide an arbitrary value to e.g. `ar_ptr` (where `malloc` expects to find an arena struct, (compare `mstate` below)).



```
pwndbg> ptype /o mchunkptr
type = struct malloc_chunk {
/*    0      |     8 */    size_t mchunk_prev_size;
/*    8      |     8 */    size_t mchunk_size;
/*   16      |     8 */    struct malloc_chunk *fd;
/*   24      |     8 */    struct malloc_chunk *bk;
/*   32      |     8 */    struct malloc_chunk *fd_nextsize;
/*   40      |     8 */    struct malloc_chunk *bk_nextsize;

                           /* total size (bytes):   48 */
                         } *
pwndbg> ptype /o heap_info
type = struct _heap_info {
/*    0      |     8 */    mstate ar_ptr;
/*    8      |     8 */    struct _heap_info *prev;
/*   16      |     8 */    size_t size;
/*   24      |     8 */    size_t mprotect_size;
/*   32      |     0 */    char pad[];

                           /* total size (bytes):   32 */
                         }
pwndbg> ptype /o mstate
type = struct malloc_state {
/*    0      |     4 */    __libc_lock_t mutex;
/*    4      |     4 */    int flags;
/*    8      |     4 */    int have_fastchunks;
/* XXX  4-byte hole  */
/*   16      |    80 */    mfastbinptr fastbinsY[10];
/*   96      |     8 */    mchunkptr top;
/*  104      |     8 */    mchunkptr last_remainder;
/*  112      |  2032 */    mchunkptr bins[254];
/* 2144      |    16 */    unsigned int binmap[4];
/* 2160      |     8 */    struct malloc_state *next;
/* 2168      |     8 */    struct malloc_state *next_free;
/* 2176      |     8 */    size_t attached_threads;
/* 2184      |     8 */    size_t system_mem;
/* 2192      |     8 */    size_t max_system_mem;

                           /* total size (bytes): 2200 */
                         } *
pwndbg>
```

Whenever `ar_ptr` points to, an attacker needs to have control over the memory (e.g. on the heap, an environment variable, stack, ...) to forge an arena there as well. The scheme is to reach the sorting of the victim chunk into the unsortedbin portion within `_int_free()`. There the linking into the bin takes place, which in turn is based on the value of `ar_ptr` as it is used to find the unsortedbin! The goal is to forge the necessary fields to the fake arena. When sorting a victim chunk into the unsortedbin takes place we force a controlled overwrite as we seem to control parts of the data being used to write (e.g. `prev_size field`) and all the locations being written to as we control the whole arena.

4. House of Orange

Gist: Overflow into the main arena's top chunk, reducing its size and forcing a top chunk

extension. An unsortedbin attack on the old top chunks bk aiming at the `_IO_list_all` pointer to set up for file stream exploitation with an attacker crafted fake file stream on the heap.

Applicable until: < 2.26

Root cause: Overflow

GLIBC VERSION	PATCHES
2.24	libio: Implement vtable verification
2.26	Do not flush stdio streams, which changes the behavior of malloc_printerr that
2.27	Abort on heap corruption without a backtrace

Idea: The House of Orange consists (typically) of a 3-step approach: Extending the top chunk, an unsortedbin attack, followed by file stream exploitation. *Phase 1* consists of utilizing an overflow into the `top_chunk` and changing its value to a small, paged aligned value with a `prev_inuse` flag set. When we now request a large chunk, which the top chunk cannot serve anymore, more memory from the kernel is requested via `brk()`. In case of the memory between the old top chunk and the returned memory from `brk()` not being contiguous (which is the case as we tampered with the top chunk), the old top chunk is freed. When this is done, the old top chunk is sorted into the unsortedbin. This enables *Phase 2*, which utilizes the same overflow bug again to tamper with the old top chunk's bk pointer to point at `_IO_list_all`.

```
pwndbg> ptype /o _IO_list_all
type = struct _IO_FILE_plus {
/*    0      |   216 */    FILE file;
/*  216      |     8 */    const struct _IO_jump_t *vtable;

                           /* total size (bytes):  224 */
                         } *
pwndbg> ptype /o FILE
type = struct _IO_FILE {
/*    0      |     4 */    int _flags;
/* XXX  4-byte hole  */
/*    8      |     8 */    char *_IO_read_ptr;
/*   16      |     8 */    char *_IO_read_end;
/*   24      |     8 */    char *_IO_read_base;
/*   32      |     8 */    char *_IO_write_base;
/*   40      |     8 */    char *_IO_write_ptr;
/*   48      |     8 */    char *_IO_write_end;
/*   56      |     8 */    char *_IO_buf_base;
/*   64      |     8 */    char *_IO_buf_end;
/*   72      |     8 */    char *_IO_save_base;
/*   80      |     8 */    char *_IO_backup_base;
/*   88      |     8 */    char *_IO_save_end;
/*   96      |     8 */    struct _IO_marker *_markers;
/*  104      |     8 */    struct _IO_FILE *_chain;
/*  112      |     4 */    int _fileno;
/*  116      |     4 */    int _flags2;
/*  120      |     8 */    __off_t _old_offset;
/*  128      |     2 */    unsigned short _cur_column;
/*  130      |     1 */    signed char _vtable_offset;
/*  131      |     1 */    char _shortbuf[1];
/* XXX  4-byte hole  */
/*  136      |     8 */    _IO_lock_t *_lock;
/*  144      |     8 */    __off64_t _offset;
/*  152      |     8 */    struct _IO_codecvt *_codecvt;
/*  160      |     8 */    struct _IO_wide_data *_wide_data;
/*  168      |     8 */    struct _IO_FILE *_freeres_list;
/*  176      |     8 */    void *_freeres_buf;
/*  184      |     8 */    size_t __pad5;
/*  192      |     4 */    int _mode;
/*  196      |    20 */    char _unused2[20];

                           /* total size (bytes):  216 */
                         }
pwndbg> p *_IO_list_all.file._chain
$8 = {
  _flags = -72537977,
  _IO_read_ptr = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_read_end = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_read_base = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_write_base = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_write_ptr = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_write_end = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_buf_base = 0x7ffff7dd1703 <_IO_2_1_stdout_+131> "\n",
  _IO_buf_end = 0x7ffff7dd1704 <_IO_2_1_stdout_+132> "",
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
```

While doing so, we're also crafting a more or less complete fake file stream. Next we're requesting a chunk with a size other than 0x60, which will trigger our unsortedbin attack, resulting in the old top chunk being sorted into the 0x60 smallbin. The result is the `_IO_list_all` pointer getting overwritten with the address of the unsortedbin. This lets the unsortedbin scan continue at the "chunk" overlapping the `_IO_list_all` pointer. It naturally fails a size sanity check, triggering the `abort()` function. This results in flushing all file streams via `_IO_flist_all_lockp()`, which dereferences `_IO_list_all` to find the first file stream. Our fake file stream overlapping the main arena is not flushed, and its `_chain` pointer overlapping the 0x60 smallbin bk is followed to the fake file stream on the heap. Having our fake file stream set up correctly, the `__overflow` entry in the fake file stream's *vtable* is called, with the address of the fake file stream as the first argument.

Notes: When overwriting the old top chunk's bk and crafting a fake file stream on the heap, the `_flags` field overlaps the old top chunk's `prev_size` field. The fake file stream's `_mode` field has to be ≤ 0. The value of `_IO_write_ptr` has to be larger than the value of `_IO_write_base`. The *vtable* pointer has to be populated with the address of a fake *vtable*, at any location, with the address of `system()` overlapping the vtable's `__overflow` entry. Finally, we can skip Phase 1 if we can set up an unsortedbin attack in any other way.

5. House of Rabbit

Gist: Link a fake chunk into the largest bin and set its size so large that it wraps around

the VA space, just stopping shy of the target we want to write.

Applicable until: < 2.28

Root cause: UAF / Fastbin Dup / Overflow / ...

GLIBC VERSION	PATCHES
2.26	Size vs prev_size check in unlink macro that only checks if the value at the fake
2.27	Ensure that a consolidated fast chunk (malloc_consolidate) has a sane size

Idea: Forge a House of Force like primitive by linking a fake chunk into a largebin and setting its size field to a huge value. Our goal is to link a fake chunk with 2 size fields into the fastbin, then shuffle it into unsortedbin, and finally into the largest largebin (bin 126). One of these size fields belong to the fake chunk, the other one to the succeeding chunk, with the succeeding chunk's size field being placed 0x10 bytes before the fake chunk's size field. Once the fake chunk is linked into the fastbin it is consolidated into the unsortedbin via `malloc_consolidate()`, which cannot be triggered via `malloc()` because this results in the fake chunk being sorted which triggers an `abort()` call when it fails a size sanity check. Instead, the fake chunk is sorted by freeing a chunk that exceeds the `FASTBIN_CONSOLIDATION_THRESHOLD` (0x10000). This can be achieved by freeing a normal chunk that borders the top chunk, because `_int_free()` considers the entire consolidated space to be the size of the freed chunk! Next we modify the fake chunks size (> 0x80001) so it can be sorted into the largest large bin. We sort our fake chunk in the largebin by requesting an even larger chunk. Depending on the circumstances we may need to increase the value of `system_mem` first by allocating, freeing, then allocating a chunk larger than the current `system_mem` value. When our fake chunk got sorted into bin 126 we want to modify its size field a final time to an arbitrary large value from which our next request should be served when trying to bridge the distance between the current heap allocation and the target we want to overwrite.

Notes: The quickest way to achieve linking our fake chunk into bin 126 is via the *House*

of Lore technique. This attack might still be feasible in later versions of GLIBC. However, an attacker has to manually tamper with the `prev_size` field of the fake fence post chunk to bypass the introduced mitigations. Additionally, GLIBC 2.29 introduced further size sanity checks for allocations from the top chunk, making this technique even more difficult to pull off.

6. Unsortedbin Attack

Gist: Perform a reflective write that results in writing a GLIBC address to an attacker controlled location.

Applicable until: < 2.29

Root cause: Overflow / WAF

GLIBC VERSION	PATCHES
2.29	Various unsortedbin integrity checks, which for example ensure that the chunk

Idea: This attack allows us to write the address of an arena's unsortedbin to an arbitrary memory location. This is enabled by the so-called *partial unlinking* process that occurs when allocating/sorting an unsortedbin. In this process, a chunk is removed from the tail end of a doubly linked, circular list and when doing so, it involves writing the address of the unsortedbin over the fd pointer of the chunk pointed to by the victim chunk's bk. When we're able to tamper with the bk of an already freed chunk that was put in the unsortedbin, we can get the address of the unsortedbin at the address we supplied in the bk + 0x10. Being able to leak the unsortedbin means leaking the GLIBC address. Other use cases are to bypass the libio vtable integrity check by overwriting the `_dl_open_hook`, corrupt the `global_max_fast variable` (*House of Prime*), or target `_IO_list_all` (*House of Orange*).

7. House of Force

Gist: Overwrite the `top_chunk` with a huge value to span the VA space, so the next

allocation overwrites the target.
Applicable until: < 2.29

Root cause: Overflow

GLIBC VERSION	PATCHES
2.29	Top chunk size integrity check
2.30	Make malloc fail with requests larger than PTRDIFF_MAX

Idea: Leverage an overflow into the top chunk and change its size to a large value to bridge the gap between the top chunk and the target (can span and wrap around the whole VA space). This allows us to request a new chunk overlapping the target and overwriting it with user controlled data. The technique worked as the top chunk size was not subject to any size integrity checks, nor was malloc checked for arbitrarily large allocation request that would exhaust the whole VA space.

8. House of Corrosion

Gist: House of Orange style attack that also utilize file stream exploitation (stderr) that is preceded by an unsortedbin attack against the `global_max_fast` variable leveraging partial overwrites.

Applicable until: > 2.26 && < 2.30

Root cause: WAF

GLIBC VERSION	PATCHES
2.24	libio vtable hardening
2.27	Further libio vtable hardening
2.27	abort() no longer flushes file stream buffers

2.28	_allocate_buffer and _free_buffer function pointers were replaced with explicit c
2.28	Check if bck->fd != victim when removing a chunk from the unsortedbin durin
2.29	Various unsortedbin integrity checks, which for example ensure that the chunk

Idea: This technique advances the *House of Orange* and requires an unsortedbin attack against the `global_max_fas`t variable (check *House of Prime*). Once `global_max_fast` has been overwritten with the address of the unsortedbin, large chunks qualify for fastbin insertion when freed. With the WAF bug, this yields 3 primitives. First, freeing a large chunk will link it into the fastbin for its size, writing the address of that chunk into an offset from the first fastbin. This allows an attacker to overwrite an 8 byte aligned qword with a heap address. Second, using the first primitive, freeing a large chunk to write its address to an offset from the first fastbin. The value at that offset is treated as a fastbin entry; hence, it is copied into the freed chunk's fd. This fd again can be tampered with the WAF bug. Requesting the same chunk back will write the tampered value back into its original location in memory. This allows an attacker to modify the least significant bytes of an address in the libc writable segment or replace a value entirely. Third, we can make one further improvement to transplant values between writable memory locations. Changing the size of a chunk between freeing and allocation it allows a value to be read onto the heap from one address, then written to a second address after being changed. This requires an attacker to emulate a double-free bug, using the WAF, which is achieved by requesting 2 nearby chunks, freeing them, then modifying the LSB of the chunk's fd that points to the other chunk to point back to itself instead! When this victim chunk is allocated, a pointer to it remains in the fastbin slot overlapping the transplant destination. Now the victim chunk size can be changed via a WAF aligned with its size field, then it is freed again to copy the transplant source data into its fd on the heap. At this point, an attacker can modify the data with the WAF. Lastly, the victim chunk size is changed again, and the chunk allocated, writing the target data to its destination fastbin slot. This third primitive requires an attacker to write "safety" values on the heap where the fastbin next size checks are applied against the victim chunk (e.g. by requesting large chunks to write the top chunk size field into). The *House of Corrrosion* combines these 3 primitives into a file stream exploit by tampering with various stderr fields and then

primitives into a file stream exploit by tampering with various stdin fields and then

triggering a failed assert. The libio vtable integrity check is bypassed by modifying the stderr vtable ptr such that a function that uses a function pointer is called when the assert fails. The location of this function pointer is overwritten with the address of a call RAX gadget.

Notes: This technique requires good heap control and quite a few allocations on top of guessing 4 bits of entropy. Furthermore, this technique only works when your thread is attached to the main arena. The strength of this technique, however, lies in it being able to drop a shell without a PIE binary having to leak any address. Finally, this technique in all its glory seems difficult to pull of, it has been demonstrated to work with the mitigations introduced in GLBIC 2.29, but personally, I think it requires too many prerequisites to pull of reliably; hence, it's in the patched section.

9. House of Roman

Gist: The idea of the House of Roman is a leakless heap exploitation technique that mainly leverages relative overwrites to get pointers in the proper locations without knowing the exact value of them.

Applicable until: < 2.29

Root cause: Overflow / Off-by-one / UAF

GLIBC VERSION	PATCHES
2.29	Various unsortedbin integrity checks, which for example ensure that the chunk

Idea: First, we try to make a fastbin point near the `__malloc_hook` using a UAF. Since we don't know the addresses of literally anything, the idea here is to make good use of heap feng shui and partial overwrites. We allocate 4 chunks A (0x60), B (0x80), C (0x80), and D (0x60). We then directly free chunk C, which gets put in the unsortedbin and also populated with a fd and bk. Then we allocate a new chunk E (0x60) which takes the spot of the old chunk C. Chunk E is not cleared and still holds the old fd and bk. Next we free

chunk D and A and so the fastbins are getting populated and chunk A's fd now points to chunk D (offset 0x190). Now we use our first partial overwrite and overwrite the LSB of chunk A's fd with a null byte. This results in the fastbin pointer in chunk A now pointing to chunk E, which is still allocated and also holds the old unsortedbin's fd and bk. Our next goal is to make the old unsortedbin pointers to point to something useful instead of the unsortedbin. We want to leverage another partial overwrite to make the fd point to the common fake fast chunk near the `__malloc_hook`. As we do NOT have a leak, we have to brute force 4 bits of entropy. The lower 3 nibbles of a GLIBC addresses are not subject to ASLR, so we can just overwrite the 12 lower bits of the old fd in chunk E without any further knowledge. However, the remaining 4 bits are subject to ASLR. When we successfully brute forced the fake chunks address and linked it into the fastbin by partially overwriting the old fd, we have to clear the fastbins first by doing two 0x60 allocations. The next chunk we allocate will be our fake chunk, overlapping the `__malloc_hook`. Step 2 now involves directing an unsortedbin attack against the `__malloc_hook` to write the address of the unsortedbin into the `__malloc_hook`. To accomplish this, we allocate another at least 0x80 sized chunk and a small guard chunk. Next, we free the larger chunk right away and link it into the unsortedbin. The chunk will receive a fd and bk again, where we target the bk with `__malloc_hook - 0x10` with another UAF attack. The next chunk falling into the unsortedbin size will trigger the unsortedbin attack. Now we already have the `__malloc_hook` populated with a GLIBC address. As we still don't have a leak, we need yet another partial overwrite in the `__malloc_hook` with either the address of `system()` or a one gadget. In The first case we need the same brute forcing idea as before but this time with 8 bits. As for a one gadget, we would need 12 bits. This allows us to overwrite the `__malloc_hook` with either of these and pop a shell eventually.

10. House of Storm

Gist: The House of Storm technique refers to a combination of a largebin + unsortedbin attack to write a valid size to a user chosen address in memory to forge a chunk.

Applicable until: < 2.29

Root cause: UAF

GLIBC VERSION	PATCHES
2.29	Multiple unsortedbin integrity checks
2.30	2.30: Check for largebin list corruption when sorting into a largebin.

Idea: The idea is to allocate two large chunks, each followed by a guard chunk to avoid consolidation and with the first chunk (A) being larger than the second (B). Next, we free both chunks in the order B, A, dropping them both into the unsortedbin. Allocating the larger chunk (A) again pushes the smaller one (B) into the largebin. We free A right away after. At this point, there is a single chunk in the largebin and a single chunk in the unsortedbin, with the chunk in the unsortedbin being the larger one. Now the idea is to overwrite chunk A's bk pointing to the target. We also have to set chunk B's bk to a valid address (e.g. `target + 0x8`). Next, we want to tamper with chunk B's `bk_nextsize` field and corrupt it with the address where our fake chunk size field should be placed. At this point, we already corrupted everything we need. Now the final step is to find the appropriate size for the last chunk to allocate. This will trigger the chain, once in the unsortedbin, the largebin chunk will be used to write a fake but valid size field value to the location of our target. Next, the unsortedbin will see that the chunk pointed to by its bk has a valid size field and is removed from the bin (this is our forged chunk that overlaps the target).

11. House of Husk

Gist: This technique builds upon a UAF bug to trigger an unsortedbin attack to overwrite the `global_max_fast` variable while also targeting the `__printf_arginfo_table` and `__printf_function_table` with relative overwrites to call a custom function/one gadget

Applicable until: < 2.29

Root cause: UAF

GLIBC VERSION	PATCHES

2.29	Various unsortedbin integrity checks, which for example ensure that the chur

Idea: In GLIBC, there exists a function called `register_printf_function`. Its sole purpose is to register a new format string for `printf()`. Internally, this function calls `__register_printf_specifier` and it sets up the `__printf_arginfo_table` when invoked for the first time.

```
/* Register FUNC to be called to format SPEC specifiers.  */
int
__register_printf_specifier (int spec, printf_function converter,
                             printf_arginfo_size_function arginfo)
{
  if (spec < 0 || spec > (int) UCHAR_MAX)
    {
      __set_errno (EINVAL);
      return -1;
    }

  int result = 0;
  __libc_lock_lock (lock);

  if (__printf_function_table == NULL)
    {
      __printf_arginfo_table = (printf_arginfo_size_function **)
        calloc (UCHAR_MAX + 1, sizeof (void *) * 2);
      if (__printf_arginfo_table == NULL)
        {
          result = -1;
          goto out;
        }
```

Snippet from __register_printf_specifier that populates the __printf_arginfo_table

When a function from the *printf* family is invoked, the `__printf_function_table` is checked for whether it is populated or not.

```
/* Use the slow path in case any printf handler is registered.  */
if (__glibc_unlikely (__printf_function_table != NULL
                      || __printf_modifier_table != NULL
                      || __printf_va_arg_table != NULL))
  goto do_positional;
```

Snippet from the vfprintf handling in GLIBC < 2.29

If one of the 3 tables is not NULL, execution is continued in the custom format string specification at the `do_positional label`. There the `__printf_arginfo_table` is used, in

specification at the `do_positional label`. There the `__printf_arginfo_table` is used, in which each index has to be a pointer to a function that is executed for a format string.

```
/* Fill in the types of all the arguments.  */
for (cnt = 0; cnt < nspecs; ++cnt)
  {
    /* If the width is determined by an argument this is an int.  */
    if (specs[cnt].width_arg != -1)
      args_type[specs[cnt].width_arg] = PA_INT;

    /* If the precision is determined by an argument this is an int.  
    if (specs[cnt].prec_arg != -1)
      args_type[specs[cnt].prec_arg] = PA_INT;

    switch (specs[cnt].ndata_args)
      {
      case 0:         /* No arguments.  */
        break;
      case 1:         /* One argument; we already have the
                         type and size.  */
        args_type[specs[cnt].data_arg] = specs[cnt].data_arg_type;
        args_size[specs[cnt].data_arg] = specs[cnt].size;
        break;
      default:
        /* We have more than one argument for this format spec.
           We must call the arginfo function again to determine
           all the types.  */
        (void) (*__printf_arginfo_table[specs[cnt].info.spec])
          (&specs[cnt].info,
           specs[cnt].ndata_args, &args_type[specs[cnt].data_arg],
           &args_size[specs[cnt].data_arg]);
        break;
      }
  }
```

Snippet from printf_positional function that is being invoked at do_positional

With that out of the way, the technique consists of 5 steps. Step 1 is to allocate 3 chunks A, B, and C. Chunk A needs to qualify for the unsortedbin and can also be used for a GLIBC address leak with a UAF. The size of chunk B has to be as calculated according to the `chunk_size` formula for offset `__printf_function_table`. The formula being `chunk_size = (delta * 2 ) + 0x20`, with delta being the number of bytes after the fastbin location. Chunk C is analogous to chunk B, just for the offset `__printf_arginfo_table`. In step 2 we free chunk A and overwrite its unsortedbin bk with our UAF with the value of `global_max_fast - 0x10`. Now we allocate chunk A again, which triggers the overwrite of `lobal_max_fast`. Step 3 involves freeing chunk B, which overwrites `__printf_function_table` with a heap address allowing the check != NULL to fail, and moving code execution to the custom *printf* format specifier path. Step 4 involves overwriting `__printf_arginfo_table` with a forged pointer to a custom function/one gadget. The `__printf_arginfo_table` is indexed as `((ASCII_val - 2 ) *`

`8)` for a specific *printf* format character. At this calculated index, a pointer to the function to call is stored when this format specifier is encountered. So to trigger our exploit we just need to make a call to `printf()` with the format specifier that we have targeted as explained above.

Notes: This technique in theory does not depend on the version of GLIBC, as long as it has fastbin and unsortedbin attacks available.

12. House of Kauri

Gist: Link a chunk into multiple tcachebins by overflowing into a freed chunk and changing it size field.

Applicable until: < 2.32

Root cause: Overflow / Double-free

GLIBC VERSION	PATCHES
2.29	Tcache double free check
2.29	Tcache validate tc_idx before checking for double frees
2.32	Safe-linking

Idea: This little technique is just another double-free mitigation bypass that when we have the possibility to overflow from one chunk into another while also being presented with a double-free, we can potentially link the same chunk in different tcache free-lists by tampering with a chunk's size field. This effectively confuses the tcache data structure by linking the same chunk into different free lists.

```
4334  #if USE_TCACHE
4335    {
4336      size_t tc_idx = csize2tidx (size);
4337      if (tcache != NULL && tc_idx < mp_.tcache_bins)
4338        {
4339          /* Check to see if it's already in the tcache.  */
4340          tcache_entry *e = (tcache_entry *) chunk2mem (p);
4341
4342          /* This test succeeds on double free.  However, we don't 100%
```

```
4343              trust it (it also matches random payload data at a 1 in
4344              2^<size_t> chance), so verify it's not an unlikely
4345              coincidence before aborting.  */
4346           if (__glibc_unlikely (e->key == tcache_key))
4347             {
4348               tcache_entry *tmp;
4349               size_t cnt = 0;
4350               LIBC_PROBE (memory_tcache_double_free, 2, e, tc_idx);
4351               for (tmp = tcache->entries[tc_idx];
4352                    tmp;
4353                    tmp = REVEAL_PTR (tmp->next), ++cnt)
4354                 {
4355                   if (cnt >= mp_.tcache_count)
4356                     malloc_printerr ("free(): too many chunks detected in tcache");
4357                   if (__glibc_unlikely (!aligned_OK (tmp)))
4358                     malloc_printerr ("free(): unaligned chunk detected in tcache 2");
4359                   if (tmp == e)
4360                     malloc_printerr ("free(): double free detected in tcache 2");
4361                   /* If we get here, it was a coincidence.  We've wasted a
4362                      few cycles, but don't abort.  */
4363                 }
4364             }
4365
4366           if (tcache->counts[tc_idx] < mp_.tcache_count)
4367             {
4368               tcache_put (p, tc_idx);
4369               return;
4370             }
4371         }
4372     }
4373 #endif
```

13. House of Fun

Gist: The House of Fun technique is based on the front link attack targeting freed largebin chunks to make malloc return an arbitrary chunk overlapping a target.

Applicable until: < 2.30

Root cause: UAF

GLIBC VERSION	PATCHES
2.30	Check for largebin list corruption when sorting into a largebin.

Idea: The easiest way to pull this technique of is by leveraging the largebin mechanics. We want to allocate 2 large chunks A and B that fit in the same largebin but are not the same size (e.g. 0x400, 0x420). Each largebin should be separated by a guard chunk to avoid chunk consolidation. Next, we free the first smaller largebin and shuffle it from the unsortedbin into the largebin. Next, we use our UAF vulnerability to overwrite the freed chunk's bk with the area we want to later tamper with (At this point, this chunk's bk still pointed to the head of the largebin). Afterwards, we free the second larger chunk and

shuffle it into the largebin as well. As both chunks are sorted into the same largebin and the second chunk is larger, the largebin is sorted. When this occurs our forged bk from chunk A now points to chunk B. The goal with that setup is to get a largebin chunk under

attacker control and overwrite its bk and `bk_nextsize` to point before the `_dl_open_hook` so that a future allocated chunk has its address written over the `_dl_open_hook` pointing then to an attacker controlled chunk on the heap, where we can forge a function vtable that executes a one gadget for example.

14. Tcache Dup

Gist: Bypassing tcache double-free mitigations by freeing our chunk into the tcache as well as the fastbin.

Applicable until: < 2.29

Root cause: Double-free

GLIBC VERSION	PATCHES
2.29	Tcache double free check
2.29	Tcache validate tc_idx before checking for double frees

Idea: The strategy is the same as with the fastbin dup. Until GLIBC 2.29 there was no double free mitigation whatsoever. Starting with version 2.29 we need to adjust our strategy a little because when a chunk is linked into a tcachebin, the address of that thread's tcache is written into the slot usually reserved for a free chunk's bk pointer, which is relabelled as a *key* field. When chunks are freed, their key field is checked and if it matches the address of the tcache, then the appropriate tcachebin is searched for the freed chunk. If its found `abort()` is called. We can bypass this by filling the tcachebin, freeing our chunk into the same sized fastbin, emptying the tcachebin and freeing our victim another time. Next, we allocate the victim chunk from the tcachebin, at which point we can tamper with the fastbin fd pointer. When the victim chunk is allocated from its fastbin, the remaining chunks in the same fastbin are dumped into the tcache,

including the fake chunk, tcache dumping does not include a double-free check.

Unpatched Techniques

Techniques detailed here look like they're still useable at the time of writing (GLIBC 2.34). Hence, the *applicable until* only contains a question mark. Patches might be present, when specific patches (known to me) have made this technique more difficult to exploit.

1. House of Lore

Gist: Link a fake chunk into the unsortedbin, smallbin or largebin by tampering with inline malloc metadata.

Applicable until: ?

Root cause: UAF / Overflow

Idea: Depending on which bin we're targeting, we need to make sure all requirements are satisfied for such a chunk. When we're linking a fake chunk into the smallbins, it requires overwriting the bk of a chunk linked into a smallbin with the address of our fake chunk. We also need to ensure that `victim.bk->fd == victim`. We get there by writing the address of the victim small chunk into the fake chunk's fd before the small chunk is allocated. Once the small chunk is allocated, the fake chunk must pass the `victim.bk->fd == victim` check as well, which we can achieve by pointing both its fd and bk at itself (at least in GLIBC 2.27). For the largebins, the easiest way to link a fake chunk in there is overwriting a skip chunk's fd with the address of a fake chunk and preparing the fake chunk's fd and bk to satisfy the safe unlinking checks. The fake chunk must have the same size field as the skip chunk and the skip chunk must have another same-sized or smaller chunk in the same bin, as malloc will not check the skip chunk's fd for a viable chunk if the skip chunk is the last in the bin. The fake chunk's fd and bk can be prepared to satisfy the safe unlinking checks by pointing them both at the fake chunk. As for the unsortedbin, it behaves similar to an unsortedbin attack and must meet all those requirements.

Notes: This technique is more of a general idea instead of a specific technique. It outlines how to trick malloc in linking fake chunks into different bins. It has been shown

that it can still be pulled of in GLIBC 2.34 at the time of writing. One only has to find a way around the introduced mitigations along the way.

2. Safe Unlink

Gist: Analogous to the unsafe unlink technique. Force the unlink macro to process attacker controlled fd/bk pointers, leading to a reflective write. We bypass the safe unlinking checks by aiming the reflected write at a pointer to a chunk that is actually in use.

Applicable until: ?

Root cause: Overflow

GLIBC VERSION	
2.26	Size vs prev_size check in unlink macro that only checks if the value at the fake
2.29	Proper size vs. prev_size check before unlink() in backwards consolidation via fr

Idea: The idea of the unsafe unlink remains the same. Generally, this technique works best if we have access to a global pointer to an array in which heap chunks are stored. This time we need to make sure that our chunk that we tampered with (fd, bk) is part of a doubly linked list before unlinking it. The check consists of `victim.fd->bk == victim` && `victim.bk->fd == victim`. It essentially says that the bk of the chunk pointed to by the victim chunk's fd points back to the victim chunk that the applies when following the victim chunk's bk and the fd of the chunk at that position needs also to point back to our victim. We can achieve this if we can forge a fake chunk starting at the 1st qword of a legit chunk's user data. Point its fd & bk 0x18 and 0x10 bytes respectively before a user data pointer to the chunk in which they reside. We also need to craft a 0x10 size smaller `prev_size` field to account for our forged chunk within the legitimate chunk. Finally, we

need to leverage an overflow to clear the `prev_inuse` flag of the succeeding chunk. The adjusted `prev_size` field will trigger the unlink and consolidation with our fake chunk instead of the legitimate one. In the end, we want that the fd and bk of the of chunk that our fake chunk points to point both back to our fake chunk. Once our fake chunk passes the safe unlinking checks, malloc unlinks it as before. Our fake chunks fd is followed and our fake chunks bk is copied over the bk at the destination. Then our bk is followed and copied and our fd is copied to the fd in the destination.

3. Fastbin Dup

Gist: Link a fake chunk into the fastbin by tricking malloc into returning the same chunk twice from the fastbins due to a double-free bug.

Applicable until: ?

Root cause: Double-free

GLIBC VERSION	PATC
2.3.4	Disallow freeing the same fast chunk twice in a row by checking if the victim is
2.3.4	Check during free for a chunk in fastbin range if the size of the next chunk (top
2.3.4	Check that a returned chunk from a fastbin is the correct size
2.27	Ensure that a consolidated fast chunk (malloc_consolidate) has a sane size

Idea: Leverage a double-free bug to make malloc link the same fast chunk twice into the respective fastbin. We can achieve this by allocating two chunks A & B and freeing them in the order A, B, A. This results in malloc later on returning this very chunk twice as well. This allows us to corrupt fastbin metadata (fd pointer) to link a fake chunk into the fastbin. This fake chunk will be allocated eventually, allowing us to either read and/or write to it.

4. House of Spirit

Gist: Link an arbitrary chunk into e.g. the fastbin by having full control over what address is supplied to `free()`.

Applicable until: ?

Root cause: Pass an arbitrary pointer to free

Idea: This technique does not rely on any traditional heap bug. It's based on the scenario that if we're able to pass an arbitrary pointer (to a crafted fake chunk for example) to free we can later allocate said chunk again and potentially overwrite sensitive data, The fake chunk must fulfill all requirements (e.g. proper size field for a fast chunk).

5. House of Einherjar

Gist: Clear a `prev_inuse` bit and force backward consolidation (with a fake chunk) creating overlapping chunks.

Applicable until: ?

Root cause: (Null byte) Overflow / Arbitrary write

GLIBC VERSION	PATCHES
2.24	Size vs prev_size check in unlink macro that only checks if the value at the fake

Idea: Clear a legit chunks `prev_inuse` flag to consolidate it backwards with a (fake) chunk, creating overlapping chunks. When attempting to consolidate with a fake chunk, we need to provide a forged fake `prev_size` field as well when operating on GLIBC >= 2.26! Having access to overlapped chunks could allow us to read or write sensitive data.

6. House of Mind (Fastbin Edition)

Gist: The idea of this attack is largely identical to the original proposed *House of Mind*

that was mostly focused on the unsortedbins, while this one targets the fastbins. We use a fake non-main arena to write to a new location.

Applicable until: ?

Root cause: Overflow

Idea: The idea of this attack is largely identical to the original proposed *House of Mind* that was mostly focused on the unsortedbins. Due to GLIBC mitigations as well as other exploit mitigations such as NX the old attack does not work anymore. However, the idea here is still identical, we need to do some proper heap feng shui to align attacker controlled data with a forged `heap_info` struct. Create said `heap_info` as well as a fake arena. Corrupt a fastbin sized chunk afterwards (setting the `NON_MAIN_ARENA` bit) and trigger the attack via freeing our victim chunk. The first step, the heap feng shui needs us to determine how many allocations of what size we have to request from the program to reach a memory address, which we can control, and where the `heap_for_ptr()` macro would return to, so we can forge the `heap_info` struct. On recent systems `EAP_MAX_SIZE` grew from 1 MB to 4 MB. We would need at least 0x2000000 bytes to force a new heap memory alignment that would return a suitable location that we control and can place the `heap_info` struct into. Once we have control of the aligned memory, we prepare the `heap_info` struct in the 2nd step. Here we really only need to provide the `ar_ptr` again, which is the first struct member. As before, we want to point `ar_ptr` to our forged arena. For a successful overwrite, we need to consider two things: our arena location and the fastbin size we target. If we want to overwrite a specific location in memory with our heap chunk, we must put the arena close to the target. When considering which fastbin to target, we need to keep in mind that depending on the choice, a different offset within the `fastbinsY[NFASTBINS]` array will be affected. When everything is in order, we can trigger the vulnerability by freeing our victim chunk. Ultimately, the attacker fully controls the location being written to but not the value itself (the written value will be at the address of the victim chunk). As we're overwriting our target (arbitrary location) with a rather large heap memory pointer, it could be used to overwrite sensitive bounded values like maximum index vars or something like `global_max_fast` again.

Notes: This likely still needs a heap leak to make it work.

7. Poison NULL byte

Gist: Leverage a single NULL byte overflow to create overlapping chunks but unlike the

House of Einherjar we do not have to provide a fake `prev_size` field.
Applicable until: ?

Root cause: Overflow

GLIBC VERSION	
2.26	Size vs prev_size check in unlink macro that only checks if the value at the fake
2.29	Proper size vs. prev_size check before unlink() in backwards consolidation via fr

Idea: The overflow must be directed at a free chunk with a size field of at least 0x110. In this scenario, the LSB of the size field is cleared, removing >= 0x10 bytes from the chunk's size... When the victim chunk is allocated again, the succeeding prev_size field is not updated. This may require 4 chunks. Chunk 1 is used to overflow in the freed chunk 2. Chunk 3 is normal-sized. Chunk 4 is used to avoid top chunk backwards consolidation. When overflown into chunk 2 we request new chunks 2.1 (normal-sized) and 2.2. Then we free chunk 2.1 and chunk 3. As the prev_size wasn't properly updated before, chunk 3 will be backwards consolidated with chunk 2.1 overlapping the still allocated chunk 2.2.

8. House of Muney

Gist: Leakless exploitation technique that tampers with the (prev_)size field of an MMAPED chunked so that when it is freed and allocated again after it's overlapping part of the memory mapping in GLIBC, where we want to tamper with the symbol table leading to code exec.

Applicable until: ?

Root cause: Overflow

Idea: We want to tamper with a MMAPED chunk, in particular with its size and/or `prev_size` field (Note: `prev_size` + `size` must equal a page size). The size of the chunk depends an can roughly be calculated with the formula `SizeofMMAPChunk = byteToLibc + bytesToOverlapDynsym`. The goal here is to gain control over normally read-only GLIBC sections: *.gnu.hash* and *.dynsym*. There are two main points to consider in this first step: `mmap_threshold` and heap feng shui. To get a chunk outside the default heap section, it must be larger than this upper limit, or we have to find a way to tamper with that limit value somehow if it denies us a proper allocation. Second, we want to position our mmaped chunk in a proper location, so that may require some tinkering. After tampering with the chunks size field, we want to free it. This results in parts of the GLIBC code getting unmapped from the VAS. The next step after freeing the chunk is allocating it back (mmapped), which results in all values being NULLed. As we pretty much deleted parts of the LIBC at this point, we need to rewrite the parts that matter. This can be done by e.g. copying in the GLIBC sections byte for byte and making the necessary changes to the symbol table, or debug the process to see what parts the loader would actually need to only provide this minimal working solution. What it comes down to in the later technique is to find the `elf_header_ptr` location in the overlapping chunk and write the `elf_bitmask_ptr`, `elf_bucket_ptr`, `elf_chain_zero_ptr`, and `symbol_table_ptr` to finally overwrite a specific function (like `exit()`) in the symbol table with the offset to system, a one gadget, any desired function or begin a ROP chain.

Notes: This technique does **not** work in full RELRO binaries. Also, the overwritten function (in the symbol table) that is being called to trigger the exploit must *not* have been called before, as otherwise the function resolution process won't kick in.

9. House of Rust

Gist: The House of Rust allows for a direct attack into GLIBC's stdout FILE stream by abusing the tcache stashing mechanism with two tcache stashing unlink + largebin attack combos.

Applicable until: ?

Root cause: UAF

GLIBC VERSION	PATCHES
2.32	Safe-linking

Idea: The House of Rust is a 5 stage approach. Stage 1 will be completely dedicated to heap feng shui. Most, if not all, needed allocations will be made here. Stage 2 involves a tcache stashing unlink + and a largebin attack. The tcache stashing unlink is used to link the `tcache_perthread_struct` into the 0xb0 tcachebin. The largebin attack is used to fix the broken the fd pointer of the chunk that holds a pointer to the `tcache_perthread_struct` in its bk pointer. At the end of this Stage, the next 0x90 sized request with be served at the `tcache_perthread_struct`. This step requires roughly 20 allocations of varying sizes. Next up in stage 3 a similar TSU + LB attack is attempted to write a GLIBC value somewhere in the `tcache_perthread_struct`. This again requires an additional 20 allocations. Stage 4 aims at getting a GLIBC leak in stdout via file stream exploitation. For that, we want to edit the chunk holding the GLIBC address, overwriting the 2 LSBs of the GLIBC, so it points to the stdout FILE structure. This requires guessing 4 bits of the GLIBC load address. When brute forcing is successful, we can allocate from the appropriate tcachebin, overlapping with `IO_2_1_stdout`. This ends up with a huge information leak the next time there is stdout activity through the file stream. Putting it all together in stage 5 to pop a shell is just consisting of editing the `tcache_perthread_struct` chunk again, overwriting its fd with the address with e.g.: one of the malloc hooks as usual.

10. House of Crust

Gist: Leakless technique that leverages house of rust safe-linking bypass primitives, leading to a *House of Corrosion* like attack that ends up with file stream exploitation in stderr to pop a shell.

Applicable until: ?

Root cause: UAF

GLIBC VERSION	PATCHES
2.32	Safe-linking

Idea: The first 2 steps are identical to the *House of Rust*. Step 3 is largely similar as well with the goal being to write 2 libc addresses to the `tcache_perthread_struct` with the execution having to use largebins as leaving the unsortedbin pointing to the `tcache_perthread_struct` after would cause an abort later down the road. Step 4 shares similarities with the *House of Corrosion* as it tries to corrupt the `global_max_fast` variable via tampering with the `tcache_perthread_struct`. This, again, requires 4 bits of GLIBC load address guessing. With the `global_max_fast` being overwritten, we're set for a transplanting primitive as described in the *House of Corrosion*. The last step, step 5, involves executing three transplants that were prepared instep 4 and trigger stderr activity (file stream exploitation). One possible way for FSE due to a recent bug (see below) is to overwrite the `__GI_IO_file_jumps` vtable. To trigger the exploit that ends up calling a one gadget requires a bit more tampering, allocating and freeing chunks as the goal is to trigger stderr activity by making malloc attempt to sort a fake chunk with a `NON_MAIN_ARENA` bit set from the unsortedbin into the largebin (which will fail).

Notes: This technique relies on a bug introduced in GLIBC v2.29 where the libio vtables are mapped into a writable segment. Additionally, the original write-up makes uses of a gadget that was only present in one of the custom GLIBC builds that were done; hence we basically just skimmed over this one due to these two massive constraints.

11. House of IO

Gist: Bypassing safe-linking, which protect single-linked free lists (fastbin, tcache) by abusing the unprotected pointer to the `tcache_perthread_struct`.

Applicable until: ?

Root cause: Underflow / UAF

GLIBC VERSION	PATCHES

GLIBC VERSION	PATCHES
2.32	Safe-linking

Idea: The `tcache_perthread_object` is allocated when the heap is created. Furthermore, it is stored right at the heap's beginning (at a relatively low memory address). The safe-linking mitigation aims to protect the fd/next pointer within the free lists. However, the head of each free-list is *not* protected. Additionally, freeing a chunk and placing it into the tcachebin also places a non-protected pointer to the appropriate tcache entry in the 2nd qword of a chunks' user data. The *House of IO* assumes one of three scenarios for the bypass to work. First, any attacker with a controlled linear buffer underflow over a heap buffer, or a relative arbitrary write will be able to corrupt the tcache. Secondly, a UAF bug allowing to read from a freed tcache eligible chunk leaks the tcache and with that, the heap base. Thirdly, a badly ordered set of calls to `free()`, ultimately passing the address of the tcache itself to free, would link the tcache into the 0x290 sized tcachebin. Allocating it as a new chunk would mean complete control over the tcache's values.

```
/* Safe-Linking:
   Use randomness from ASLR (mmap_base) to protect single-linked lists
   of Fast-Bins and TCache.  That is, mask the "next" pointers of the
   lists' chunks, and also perform allocation alignment checks on them.
   This mechanism reduces the risk of pointer hijacking, as was done with
   Safe-Unlinking in the double-linked lists of Small-Bins.
   It assumes a minimum page size of 4096 bytes (12 bits).  Systems with
   larger pages provide less entropy, although the pointer mangling
   still works.  */
#define PROTECT_PTR(pos, ptr) \
  ((__typeof (ptr)) ((((size_t) pos) >> 12) ^ ((size_t) ptr)))
#define REVEAL_PTR(ptr)  PROTECT_PTR (&ptr, ptr)
```

12. Largebin attack

Gist: Similar to an *unsortedbin attack,* as its based on tampering with a freed largebin's

`bk_nextsize` pointer.

Applicable until: ?

Root cause: UAF / Overflow

GLIBC VERSION	PATCHES
2.30	Two checks for large bin list corruption when sorting into a largebin

Idea: This attack is largely similar to a classic *unsortedbin attack*, which got rather difficult to pull of after multiple patches to the unsortedbin integrity checks. The unsortedbin works by manipulating the bk pointer of a freed chunk. In the largebin attack, we leverage the mechanism when the largebin is looped/sorted and tries to work on a tampered with largebin with attacker controlled `bk_nextsize` and/or bk pointers. The technique itself is often used as a gateway for other exploit techniques, e.g. by overwriting the `global_max_fast` variable enabling multiple fastbin attacks. The basic idea is to overwrite a freed large chunks `bk_nextsize` pointer with the address of `target-0x20`. When requesting a final chunk and the largebin sorting takes places, our overwriting the target is triggered as well. This setup requires some prior heap feng shui, with multiple large chunks separated by guards to avoid consolidation. A working example for GLIBC 2.31 with the mentioned patch above is as follows: Request a large chunk A (0x428) followed by a guard chunk. Request another smaller large chunk B (0x418) followed by another guard chunk. Free chunk A to link it into the unsortedbin. Request a chunk C that is larger than chunk A to trigger the largebin sorting and make chunk A be placed in there. Next we free chunk B to place it into the unsortedbin again. Now, we would need to leverage a UAF or even overflow scenario to overwrite chunk A's `bk_nextsize` pointer with the `target address - 0x20`. If we now request a final chunk D that is larger than the just freed chunk B, it places chunk B into the largebin. In this particular version of GLIBC, malloc does *not* check a freed chunks `bk_nextsize` pointer for integrity if the new inserted chunk is smaller (chunk B) than currently the smallest present one (chunk A). Upon inserting chunk B into the largebin, chunk A's `bk_nextsize->fd->nextsize` is overwritten to the address of chunk B. In our case, our target is now overwritten with the address of chunk B.

13. Tcache - House of Botcake

Gist: Bypass current tcache double-free mitigations by linking the victim chunk into the tcache as well as the unsortedbin where it has been consolidated with a neighboring

chunk.

Applicable until: > 2.25 && < ?

Root cause: Double-free

GLIBC VERSION	PATCHES
2.29	Tcache double-free checks
2.29	Tcache validate tc_idx before checking for double frees

Idea: This is a powerful Tcache poisoning attack that tricks malloc into returning a pointer to an arbitrary memory location. This technique solely relies on the tcache eligible chunks (overlapping the fast chunks). So first we need to fill the tcache bin of a specific chunk size (e.g. 0x100). For that, we need to allocate 7 0x100 sized chunks that will serve as filler material soon. Next up, we need to prepare another 0x100 sized chunk (A) that is used for consolidation later down the road. Next up with allocate yet another 0x100 sized chunk B that will be our victim. Afterwards, we still need a small guard chunk to prevent consolidation with the top chunk. Now we free the 7 prepared filler chunks to fill the 0x100 sized tcachebin. Step 2 now involves freeing our victim chunk B that is, due to its sized, sorted into the unsortedbin. Step 3 involves freeing chunk A, leading to consolidation with our victim chunk B. Next, we allocate a 0x100 sized garbage chunk to free one slot from 0x100 tcachebin. Now we trigger the double-free bug with freeing chunk victim chunk B another time. Now our victim chunk is part of a larger chunk in the unsortedbin as well as part of the 0x100 tcachebin! We're now able to do a simple tcache poisoning by using this overlapped chunk. We will allocate a new 0x120 sized chunk overlapping the victim chunk and overwrite the victim chunk's fd with the target. The 2nd next allocation we make from the 0x100 sized tcachebin overlaps the target. A major

advantage of this technique is that we can potentially free chunk A and B as many times as we want, and each time modify their fd pointers to gain as many arbitrary writes as needed.

14. Tcache - House of Spirit

Gist: Free a fake tcache chunk to trick malloc into returning a nearly arbitrary pointer.

Applicable until: ?

Root cause: Being able to pass an arbitrary pointer to `free()`

Idea: The idea is identical to the non tcache chunk *House of Spirit* technique, but easier to pull of and with one constraint. We do *not* have to create a second fake chunk after the fake chunk we want to link into the tcache free list, as there are no strict sanity checks in the tcache route in malloc. The freed fake chunk must be appropriately sized, so it is eligible for the tcache.

15. Tcache - Poisoning

Gist: Poison the tcache bins and tricking malloc into returning a pointer to an arbitrary memory location.

Applicable until: ?

Root cause: Overflow / UAF

GLIBC VERSION	PATCHES
2.28	Ensuring the tcache count is not NULL

Idea: We can achieve this by allocating two tcache eligible chunks A, B of the same size. Then we're freeing them in the same order we allocated them right after. Next, we use an overflow or UAF bug to overwrite the fd pointer in chunk B with our target location. Now

the second allocation we do from this tcachebin will allocate a chunk overlapping our target

16. Tcache - Stashing unlink

Gist: Link a fake chunk into the tcache by using a quirk of calloc, which does not allocate from the tcachebin as its top priority.

Applicable until: ?

Root cause: Overflow / UAF

Idea: In step 1 of this technique, we will allocate 9 [A to I] chunks that are within tcache size and not fastbin size. They will fall within smallbin size (e.g. 0x90). Next we will free chunks D to I, followed by chunk B to fill the 0x90 tcachebin. Next, we will free chunk A and C, which are now put into the unsortedbin. Step 2 starts with allocating a chunk J that's just above the 0x90 smallbin size (e.g. 0xa0). This will shuffle the two 0x90 chunks from the unsortedbin into the smallbin. Next, we allocate 2 more 0x90 sized chunks K and L, which are taken from the tcachebin, which concludes step 2. Step 3 leverages our overflow / UAF bug, in which we want to overwrite the bk pointer of freed chunk C that's located inside the smallbin with an address to a fake chunk. Now we want to allocate another 0x90 chunk M. This allocation has to take place with **calloc**, as only then, the allocation won't be taken from the tcachebin. This will return the previously freed chunk from the smallbin to the user, while the remaining smallbin as well as our fake chunk will be dumped into the tcachebins as there are two free slots in there. Now the next 0x90 sized allocation we're doing with malloc will be taken from and will be our fake chunk

Notes: This attack requires at least one allocation with calloc!

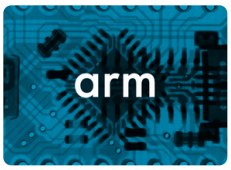
And that's a wrap! I'll be eventually coming back to this post, adding more links to GLIBC patches or even newly discovered techniques. If you found some mistakes or have found a cool new technique, feel free to ping me, as I'd love to know where I went wrong or extend this post with more knowledge :)!

I'm a (Vulnerability-) Researcher based in Europe. In general, anything low-level excites me. Focussing on bug hunting, exploitation, hardware, and fuzzing

Prev article

MISC study notes about ARM AArch64 Assembly and the ARM Trusted Execution...

LATEST POSTS



MISC study notes about ARM AArch64 Assembly and the ARM Trusted Execution Environment (TEE)

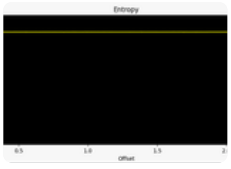
12 February 2022

LinkSys EA6100 AC1200 - Part 2 - A serial connection FTW!

5 November 2021

The devil entered the stage!

3 February 2021



LinkSys EA6100 AC1200 - Part 1 - PCB reversing

11 January 2021

Newsletter

Stay up to date! Get all the latest & greatest posts delivered straight to your inbox

youremail@example.com

Subscribe

TAG CLOUD

RE (8) Hardware (6) Exploitation (5) General (2) Fuzzing (1)

Design with ❤ by @GodoFredoNinja