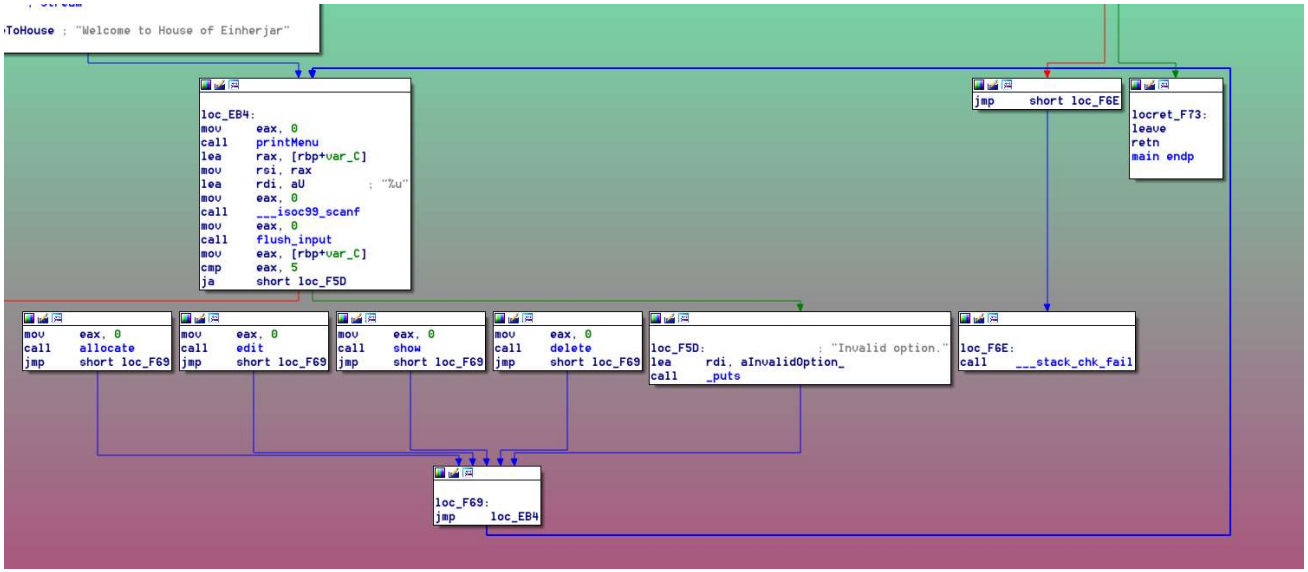
# House of Einherjar

Glibc-2.27

Attack Demonstration

# Analyzing the Binary

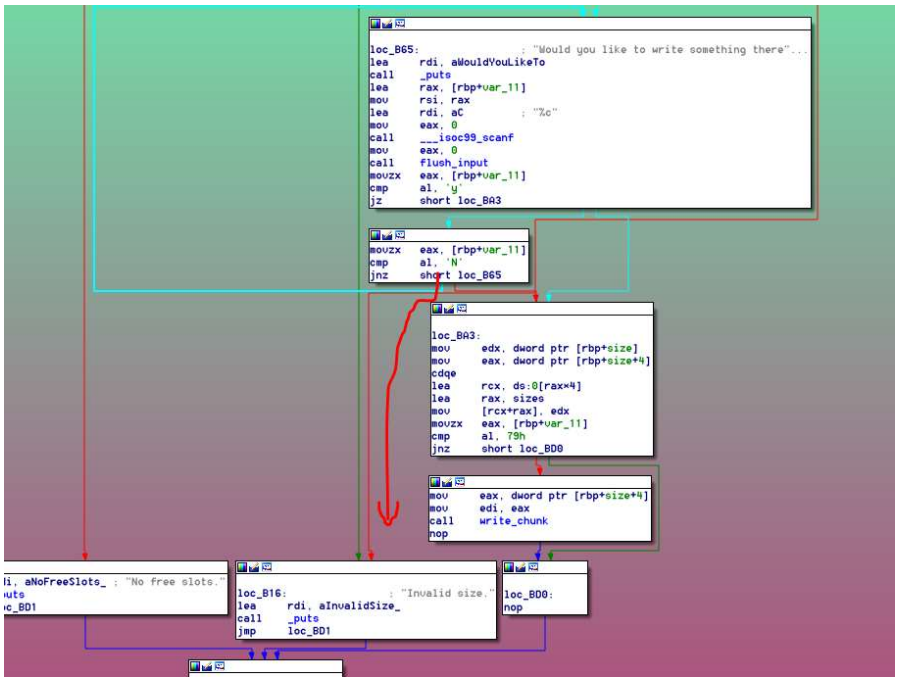



Classic Heap Note Challenge

We can:

(1) Allocate and optionally initialize the data for a new chunk
(2) Edit the data of an existing chunk
(3) Print the string at an existing chunk
(4) Delete an existing chunk

# Vulnerabilities




allocate(): does not force new chunks to be initialized, allowing for easy data leaks

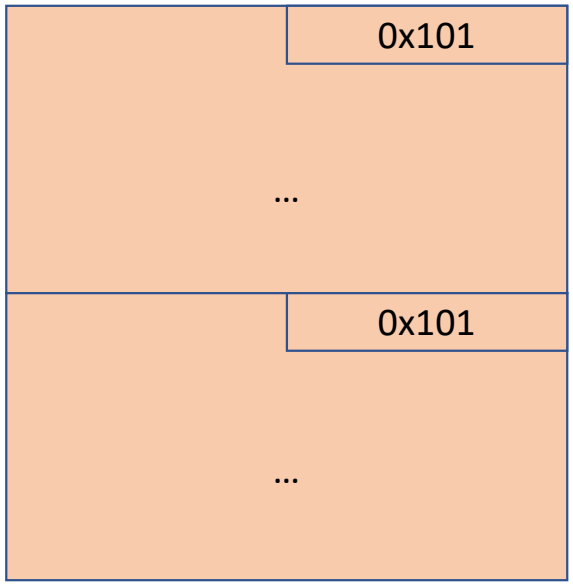
```
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp+var_4], edi
lea     rdi, aWhatWouldYouLi ; "What would you like to write there?"
call    _puts
mov     rdx, cs:stdin@@GLIBC_2_2_5 ; stream
mov     eax, [rbp+var_4]
lea     rcx, ds:0[rax*4]
lea     rax, sizes
mov     eax, [rcx+rax]
add     eax, 1
mov     esi, eax        ; n
mov     eax, [rbp+var_4]
lea     rcx, ds:0[rax*8]
lea     rax, chunks
mov     rax, [rcx+rax]
mov     rdi, rax        ; s
call    _fgets
lea     rdi, aYourDataWasSav ; "Your data was saved."
call    _puts
nop
leave
retn
```

write_chunk(): reads in one too many bytes, resulting in potential single null-byte overflow

# How To Leverage a Single Null-Byte Heap Overflow?

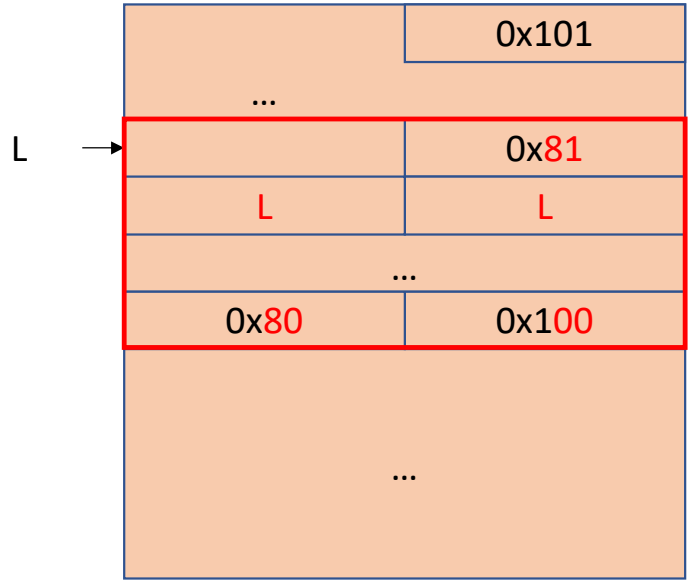
- Recall that glibc heap chunks record the status of the previous (memory-adjacent) chunk in the size field of the chunk
- By overflowing a single null byte, we
  - declare that the previous chunk is not in use, and thus can be unlinked and consolidated if needed
  - reduce the size field of the previous chunk to the highest multiple of 0x100 below the uncorrupted value (fails if chunk is already <0x100)

# Heap Layout




Healthy heap layout

Single null-byte
overflow in first chunk




Overflowed heap layout

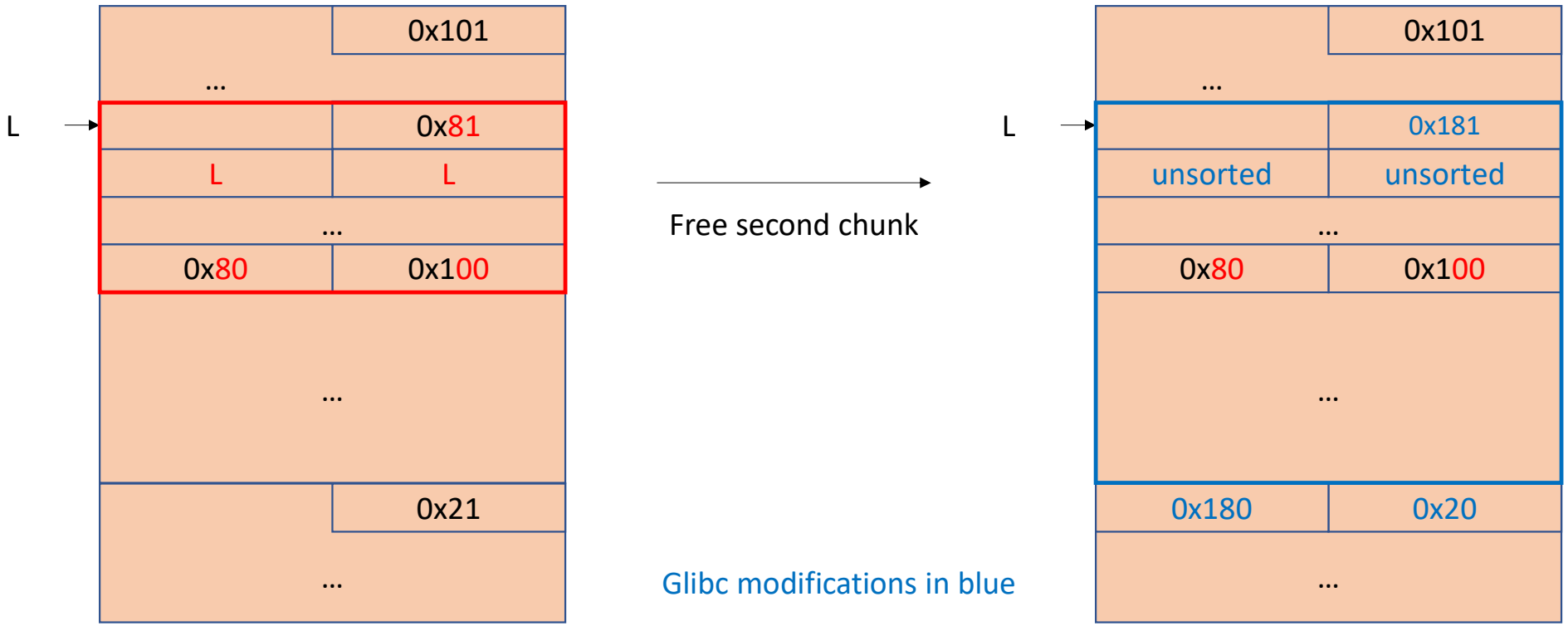
Attacker-controlled data in red

Assume tcache[0x100] is filled.
0x100 is not a fastbins size.
Assume third chunk is in use or top.
What happens when we free() the second chunk?

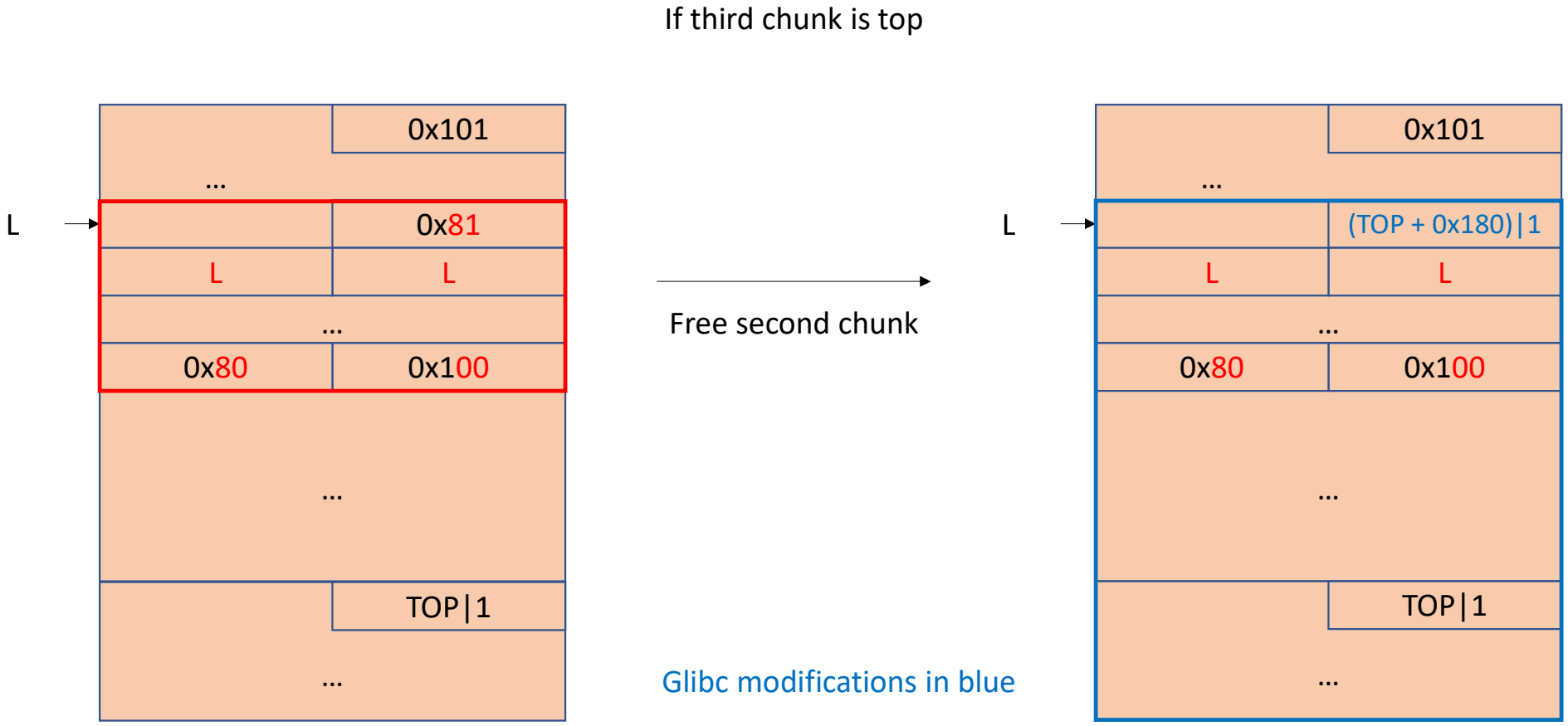
```
4259  else if (!chunk_is_mmapped(p)) {
4260
4261    /* If we're single-threaded, don't lock the arena.  */
4262    if (SINGLE_THREAD_P)
4263      have_lock = true;
4264
4265    if (!have_lock)
4266      __libc_lock_lock (av->mutex);
4267
4268    nextchunk = chunk_at_offset(p, size);
4269
4270    /* Lightweight tests: check whether the block is already the
4271       top block.  */
4272    if (__glibc_unlikely (p == av->top))
4273      malloc_printerr ("double free or corruption (top)");
4274    /* Or whether the next chunk is beyond the boundaries of the arena.  */
4275    if (__builtin_expect (contiguous (av)
4276                          && (char *) nextchunk
4277                          >= ((char *) av->top + chunksize(av->top)), 0))
4278        malloc_printerr ("double free or corruption (out)");
4279    /* Or whether the block is actually not marked used.  */
4280    if (__glibc_unlikely (!prev_inuse(nextchunk)))
4281      malloc_printerr ("double free or corruption (!prev)");
4282
4283    nextsize = chunksize(nextchunk);
4284    if (__builtin_expect (chunksize_nomask (nextchunk) <= 2 * SIZE_SZ, 0)
4285        || __builtin_expect (nextsize >= av->system_mem, 0))
4286      malloc_printerr ("free(): invalid next size (normal)");
4287
4288    free_perturb (chunk2mem(p), size - 2 * SIZE_SZ);
4289
4290    /* consolidate backward */
4291    if (!prev_inuse(p)) {
4292      prevsize = prev_size (p);
4293      size += prevsize;
4294      p = chunk_at_offset(p, -((long) prevsize));
4295      unlink(av, p, bck, fwd);
4296    }
```

```
1403 /* Take a chunk off a bin list */
1404 #define unlink(AV, P, BK, FD) {                                            \
1405     if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
1406       malloc_printerr ("corrupted size vs. prev_size");                       \
1407     FD = P->fd;                                                             \
1408     BK = P->bk;                                                             \
1409     if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                   \
1410       malloc_printerr ("corrupted double-linked list");                     \
1411     else {                                                                  \
1412         FD->bk = BK;                                                        \
1413         BK->fd = FD;                                                        \
1414         if (!in_smallbin_range (chunksize_nomask (P))                       \
1415             && __builtin_expect (P->fd_nextsize != NULL, 0)) {              \
1416             if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)      \
1417                 || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))  \
1418               malloc_printerr ("corrupted double-linked list (not small)"); \
1419             if (FD->fd_nextsize == NULL) {                                  \
1420                 if (P->fd_nextsize == P)                                    \
1421                   FD->fd_nextsize = FD->bk_nextsize = FD;                   \
1422                 else {                                                      \
1423                     FD->fd_nextsize = P->fd_nextsize;                       \
1424                     FD->bk_nextsize = P->bk_nextsize;                       \
1425                     P->fd_nextsize->bk_nextsize = FD;                       \
1426                     P->bk_nextsize->fd_nextsize = FD;                       \
1427                   }                                                        \
1428               } else {                                                     \
1429                 P->fd_nextsize->bk_nextsize = P->bk_nextsize;              \
1430                 P->bk_nextsize->fd_nextsize = P->fd_nextsize;              \
1431               }                                                            \
1432           }                                                                \
1433       }                                                                    \
1434 }
```

# Heap Layout


If third chunk is in use
L
0x101
...
0x81
L
L
...
0x80
0x100
...
0x21
...
Free second chunk
Glibc modifications in blue
L
0x101
...
0x181
unsorted
unsorted
...
0x80
0x100
...
0x180
0x20
...

# Heap Layout


If third chunk is top
L
0x101
...
0x81
L
L
...
0x80
0x100
...
TOP|1
...
Free second chunk
Glibc modifications in blue
L
0x101
...
(TOP + 0x180)|1
L
L
...
0x80
0x100
...
TOP|1
...

# Exploit

- Either way, we get overlapping chunks, so we can control heap metadata of a future allocated chunk through the first chunk
- To complete the exploit in 2.27, poison tcache to write to free hook
  - Allocate chunk (X) from overlapped region
  - Free X to tcache
  - Edit first chunk to change X's next pointer to free hook
  - Allocate chunk (X returned); initialize with "/bin/sh"
  - Allocate chunk (free hook returned); initialize with system()
  - Free X

# Exploit

- What about leaks? Use show()
  - We require knowledge of the address of the fake chunk (L), which requires a heap leak
    - Free two chunks to tcache
    - Reallocate both without initialization
  - We also need to know the address of free hook and system(), which requires a libc leak
    - Free non-fastbin small chunk to unsorted bin
    - Request slightly (0x10) smaller chunk, which first moves the freed chunk to smallbin, then returns the chunk with smallbin address
    - Requesting the same size doesn't work, because it will be first stashed to tcache, zeroing out the bin address

```
3781          /* Take now instead of binning if exact fit */
3782
3783          if (size == nb)
3784            {
3785              set_inuse_bit_at_offset (victim, size);
3786              if (av != &main_arena)
3787                set_non_main_arena (victim);
3788 #if USE_TCACHE
3789              /* Fill cache first, return to user only if cache fills.
3790                 We may return one of these chunks later.  */
3791              if (tcache_nb
3792                  && tcache->counts[tc_idx] < mp_.tcache_count)
3793                {
3794                  tcache_put (victim, tc_idx);
3795                  return_cached = 1;
3796                  continue;
3797                }
3798              else
3799                {
3800 #endif
3801              check_malloced_chunk (av, victim, nb);
3802              void *p = chunk2mem (victim);
3803              alloc_perturb (p, bytes);
3804              return p;
3805 #if USE_TCACHE
3806                }
3807 #endif
3808            }
```