House of Einherjar

Yet Another Heap Exploitation Technique on GLIBC

Hiroki Matsukuma

@st4g3r

CODE BLUE 2016.10.20

$whoami

Hiroki MATSUKUMA (@st4g3r)

- Newbie Pentester
  - Cyber Defense Institute, Inc.
- CTF Player
  - TokyoWesterns
- My interests include
  - Exploitation
  - Glibc malloc (currently)

tl;dr

- Heap Exploitation(x64 Linux/Glibc malloc)
- What's "House of Einherjar" ?
  - This is a new heap exploitation technique that forces glibc malloc() to return a nearly-arbitrary address.
    - User is ordinarily able to read/write to the address returned by malloc().
  - The concept is abusing consolidating chunks to prevent from the fragmentation.
    - Off-by-one Overflow on well-sized chunk leads both control of prev_size and PREV_INUSE bit of the next chunk.
- Proof of Concept
  - http://ux.nu/6Rv6h

Overview

- Glibc malloc
  - Chunk
  - Bin
  - Consolidating Chunks
- House of Einherjar
  - Flaw / Flow
  - Demo
  - Evaluation
  - Countermeasures

Glibc malloc Chunk

- "struct malloc_chunk"
  - A memory block joins free list after being free()'ed.
  - free()'ed block is treated as "struct malloc_chunk".
    - The size of a chunk is aligned on SIZE_SZ*2 bytes.

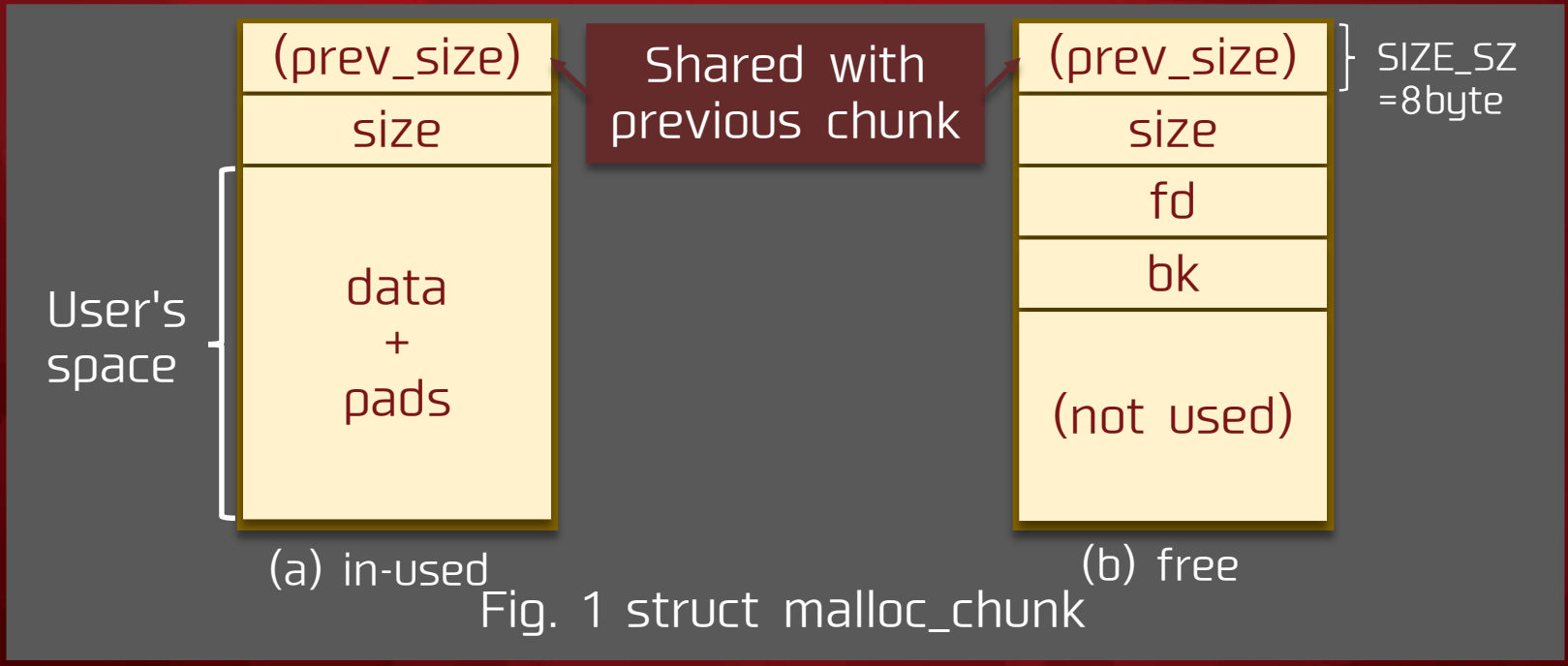


Fig. 1 struct malloc_chunk

Glibc malloc Chunk

Table 1: struct malloc_chunk

TYPE	NAME	DESCRIPTION
INTERNAL_SIZE_T	prev_size	Size of previously contigous chunk (shared)
INTERNAL_SIZE_T	size	Size of itself and its current status
struct malloc_chunk	*fd	Pointer to forwardly linked chunk (free list).
struct malloc_chunk	*bk	Pointer to backwardly linked chunk (free list).

Glibc malloc Chunk

- "struct malloc_chunk"
  - A memory block joins free list after being free()'ed.
  - It is ordinarily treated as "struct malloc_chunk".
    - The size of a chunk is aligned on SIZE_SZ*2 bytes.

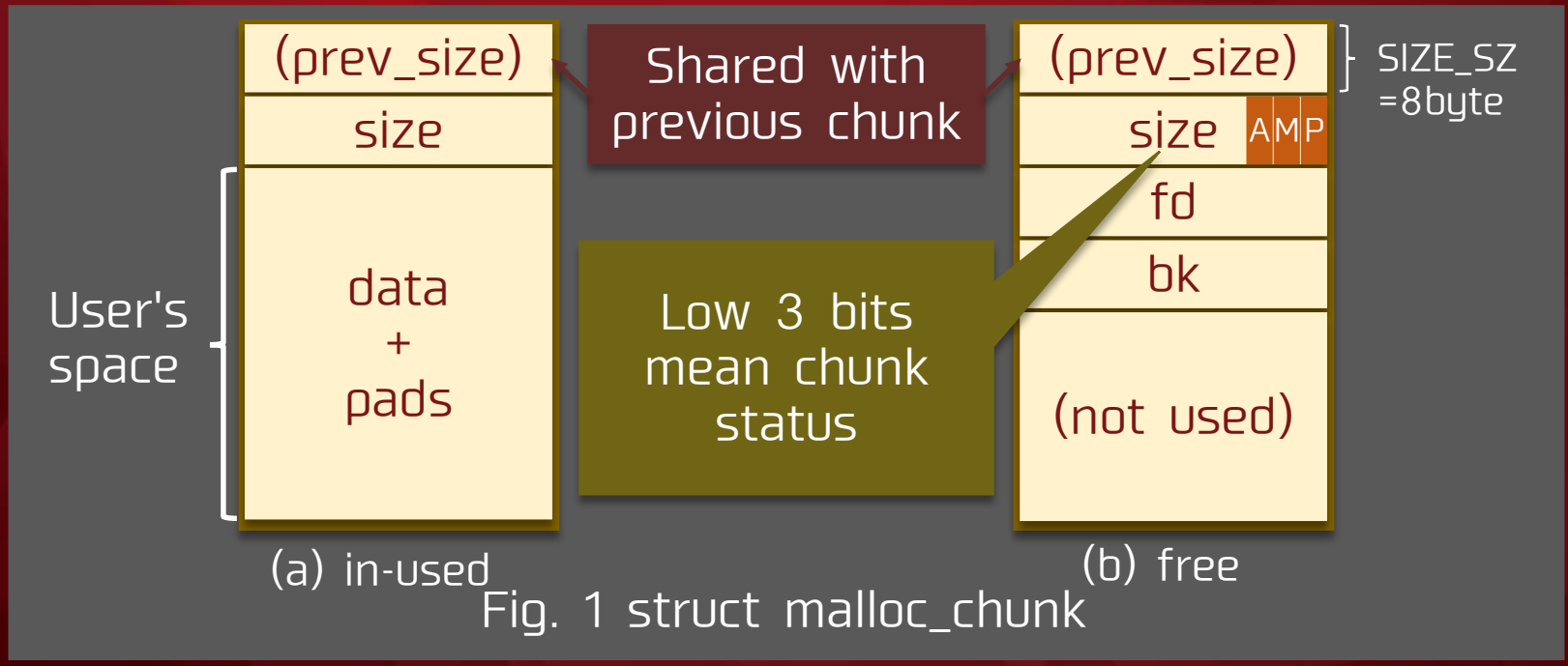


Fig. 1 struct malloc_chunk

Glibc malloc Chunk

- "struct malloc_chunk"
  - A memory block joins free list after being free()'ed.
  - It is ordinarily treated as "struct malloc_chunk".
    - The size of a chunk is aligned on SIZE_SZ*2 bytes.

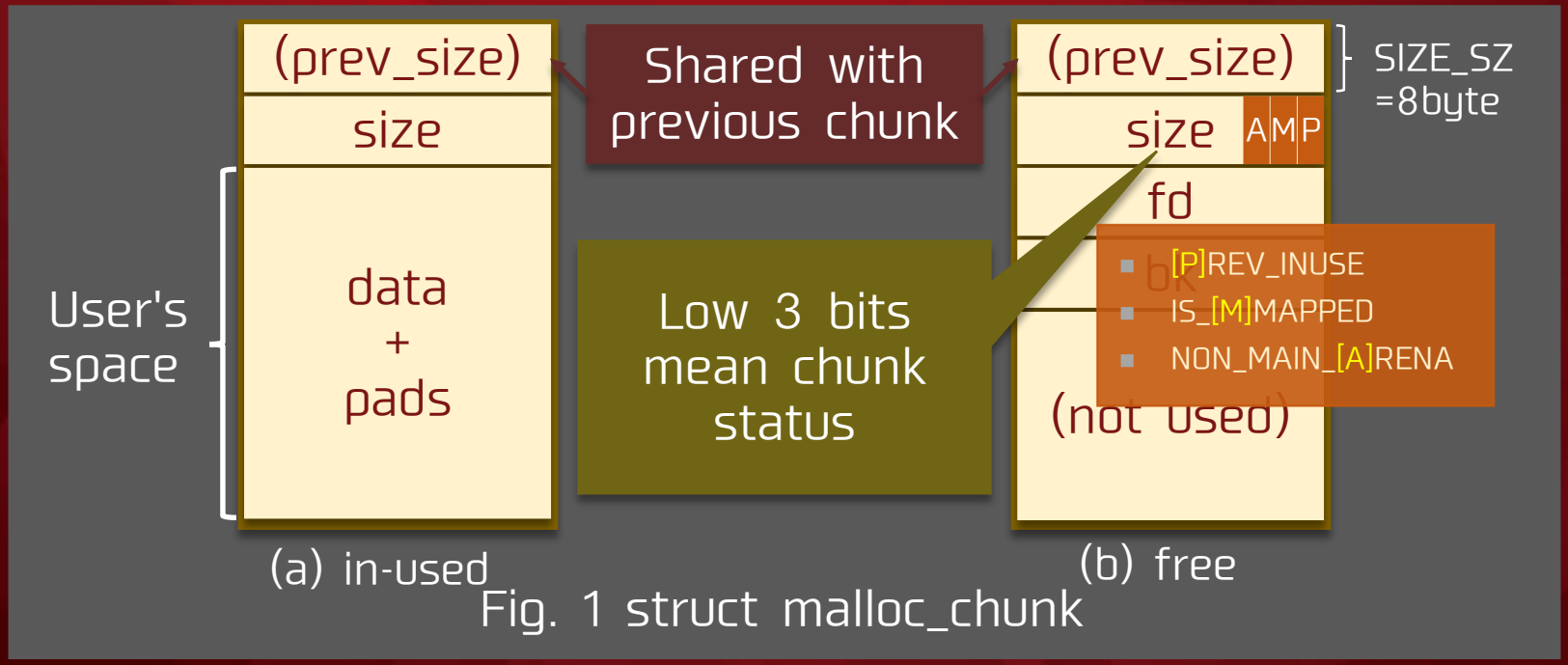


Fig. 1 struct malloc_chunk

Glibc malloc Bin

- A free chunk belongs to free list(bin).
  - Small bins
    - MAX_FAST_SIZE < size < MIN_LARGE_SIZE
      - MAX_FAST_SIZE: 0xa0
      - MIN_LARGE_SIZE: 0x400
  - Unsorted bins
    - The chunk which has just free()'ed temporarily belongs to this list.
    - There is no size restriction.

Glibc malloc Bin

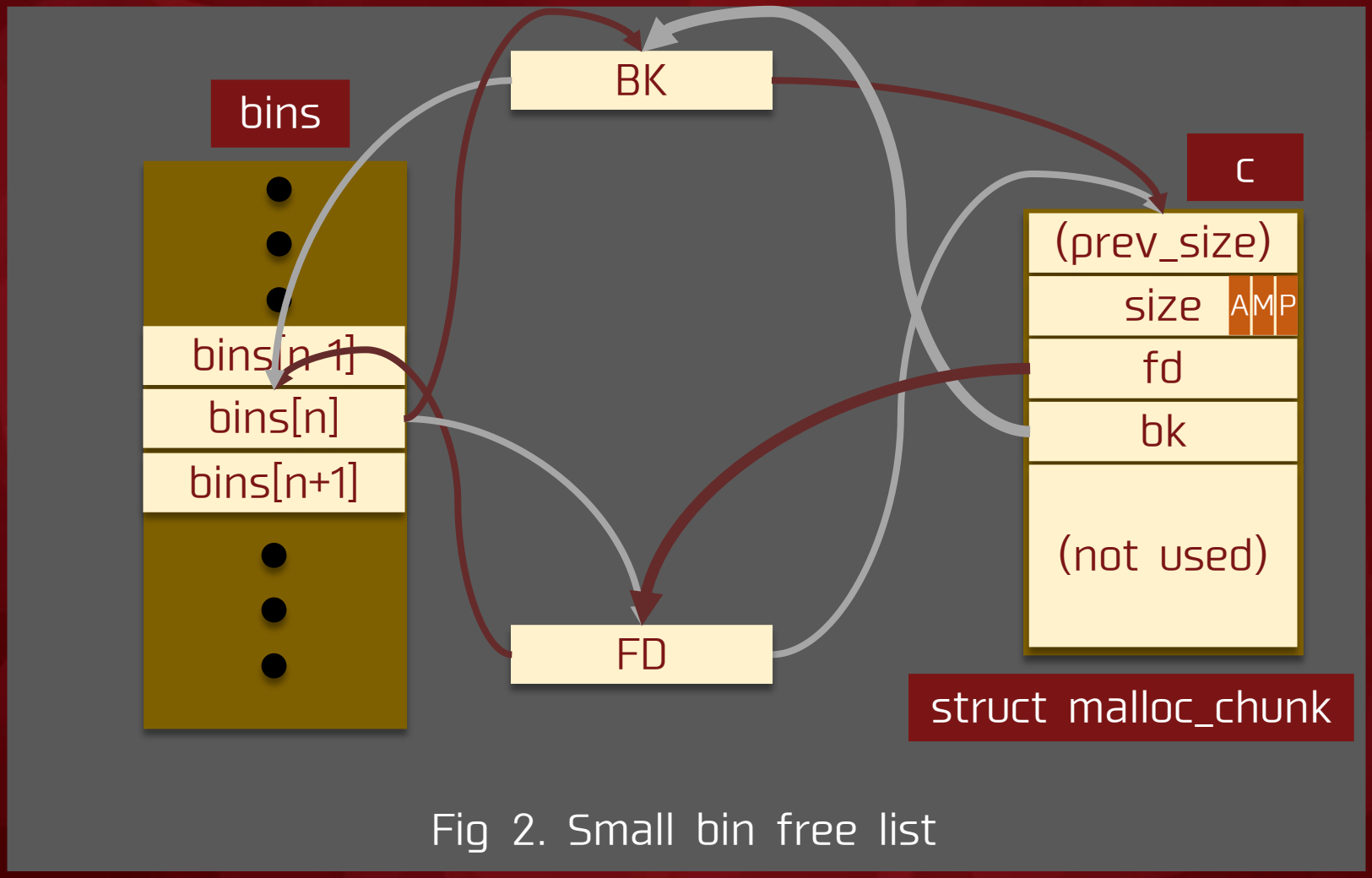


Fig 2. Small bin free list

Glibc malloc Consolidating Chunks

- It can easily cause fragmentation owing to frequent allocations and vice versa.
  - Let's consider consolidating the chunk being free()'ed and already free()'ed contiguous chunk.
    - Previous contiguous.
    - Next contiguous.
- PREV_INUSE bit
  - The flag for distinguishing whether the previous contiguous chunk is in used or not.
  - This is the sole criterion for the consolidating.

Glibc malloc Consolidating Chunks

- Where is the flow of chunk consolidating?
  - Let's read glibc!
  - free(p)
    - __libc_free(p)
      - _int_free(av, p, have_lock) <- THIS!

```
/*
   ------------------------------ free ------------------------------
 */

static void
_int_free (mstate av, mchunkptr p, int have_lock)
{
  INTERNAL_SIZE_T size;        /* its size */
  mfastbinptr *fb;             /* associated fastbin */
  mchunkptr nextchunk;         /* next contiguous chunk */
  INTERNAL_SIZE_T nextsize;    /* its size */
  int nextinuse;               /* true if nextchunk is used */
  INTERNAL_SIZE_T prevsize;    /* size of previous contiguous chunk */
  mchunkptr bck;               /* misc temp for linking */
  mchunkptr fwd;               /* misc temp for linking */

  const char *errstr = NULL;
  int locked = 0;

  size = chunksize (p);
```

(a) Entry point

Fig. 3 _int_free()

```
/* consolidate backward */
if (!prev_inuse(p)) {
  prevsize = p->prev_size;
  size += prevsize;
  p = chunk_at_offset(p, -((long) prevsize));
  unlink(av, p, bck, fwd);
}
```

(b) Consolidating point

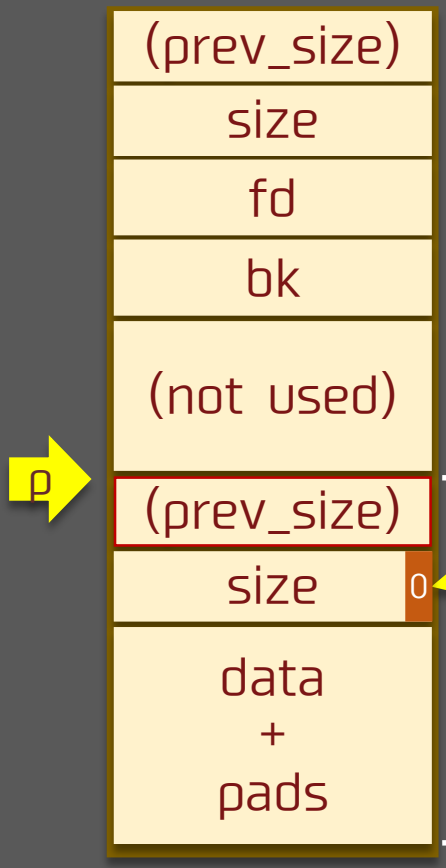
Fig. 3 _int_free()

```
  /*
    Place the chunk in unsorted chunk list. Chunks are
    not placed into regular bins until after they have
    been given one chance to be used in malloc.
  */

  bck = unsorted_chunks(av);
  fwd = bck->fd;
  if (__glibc_unlikely (fwd->bk != bck))
    {
      errstr = "free(): corrupted unsorted chunks";
      goto errout;
    }
  p->fd = fwd;
  p->bk = bck;
  if (!in_smallbin_range(size))
    {
      p->fd_nextsize = NULL;
      p->bk_nextsize = NULL;
    }
  bck->fd = p;
  fwd->bk = p;

  set_head(p, size | PREV_INUSE);
  set_foot(p, size);

  check_free_chunk(av, p);
}
```

(c) End point

Fig. 3 _int_free()

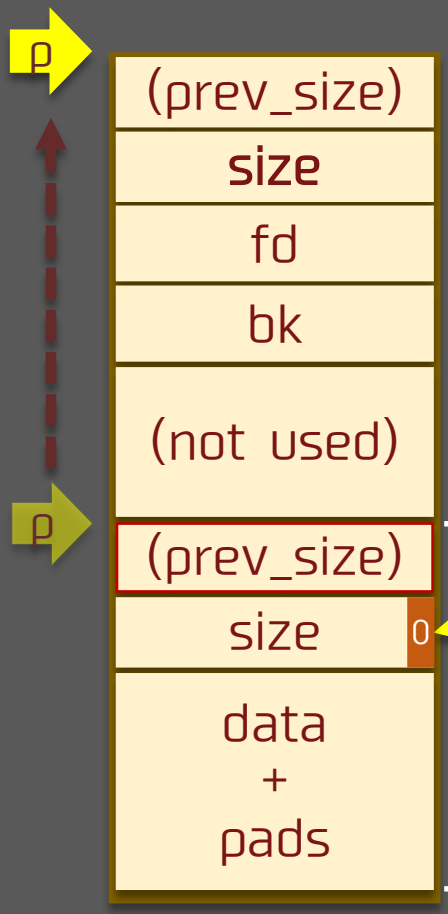
Glibc malloc Consolidating Chunks




```
size = p->size
If not prev_inuse(p):
    prevsize = p->prev_size
    size += prevsize
    p += -(long)(prevsize)
```

(a) Test prev_inuse
Fig. 4 Consolidating

Glibc malloc Consolidating Chunks

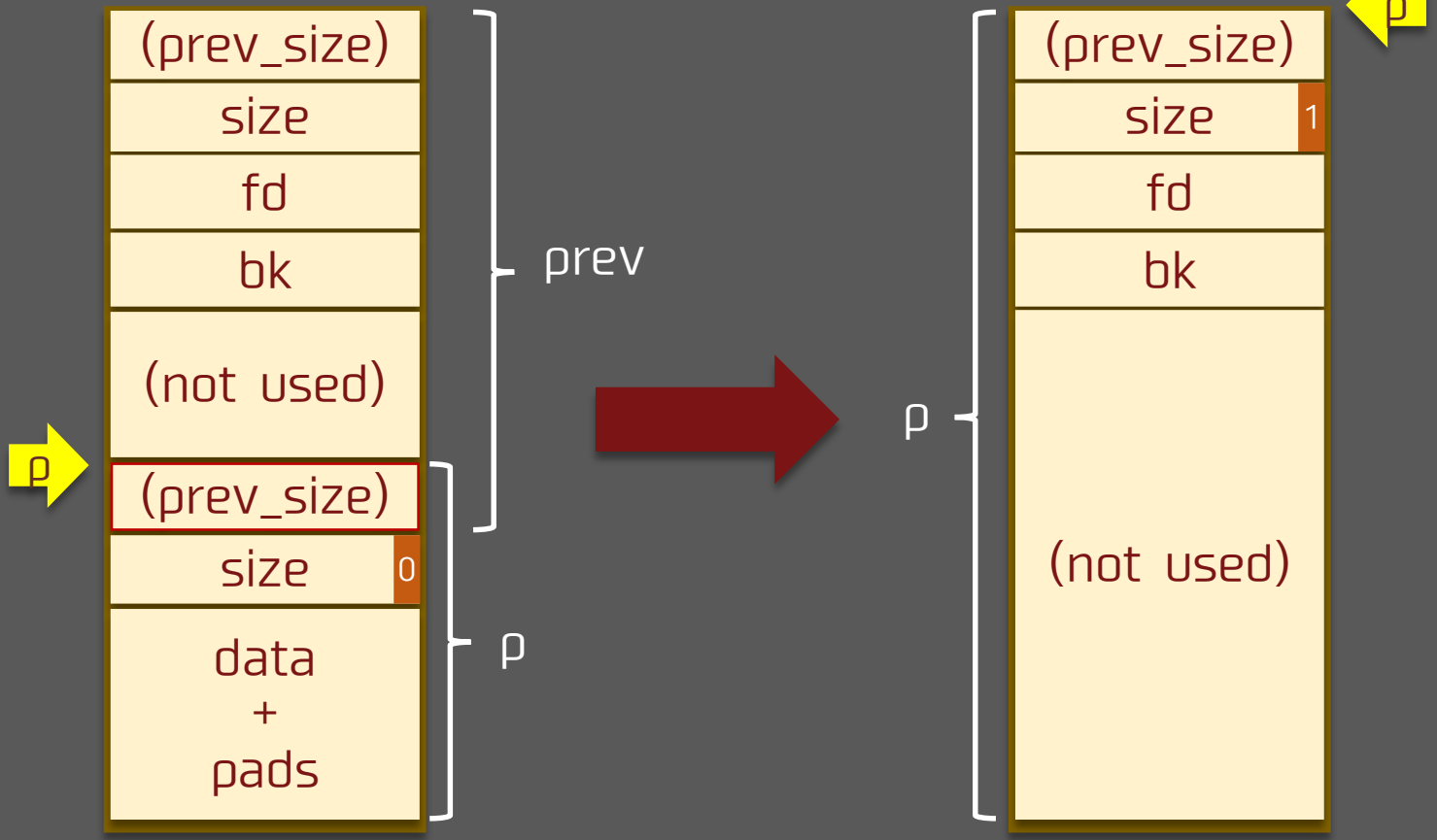



```
size = p->size
If not prev_inuse(p):
    prevsize = p->prev_size
    size += prevsize
    p += -(long)(prevsize)
```

(b) Relocation

Fig. 4 Consolidating

Glibc malloc Consolidating Chunks



(c) New chunk

Fig. 4 Consolidating

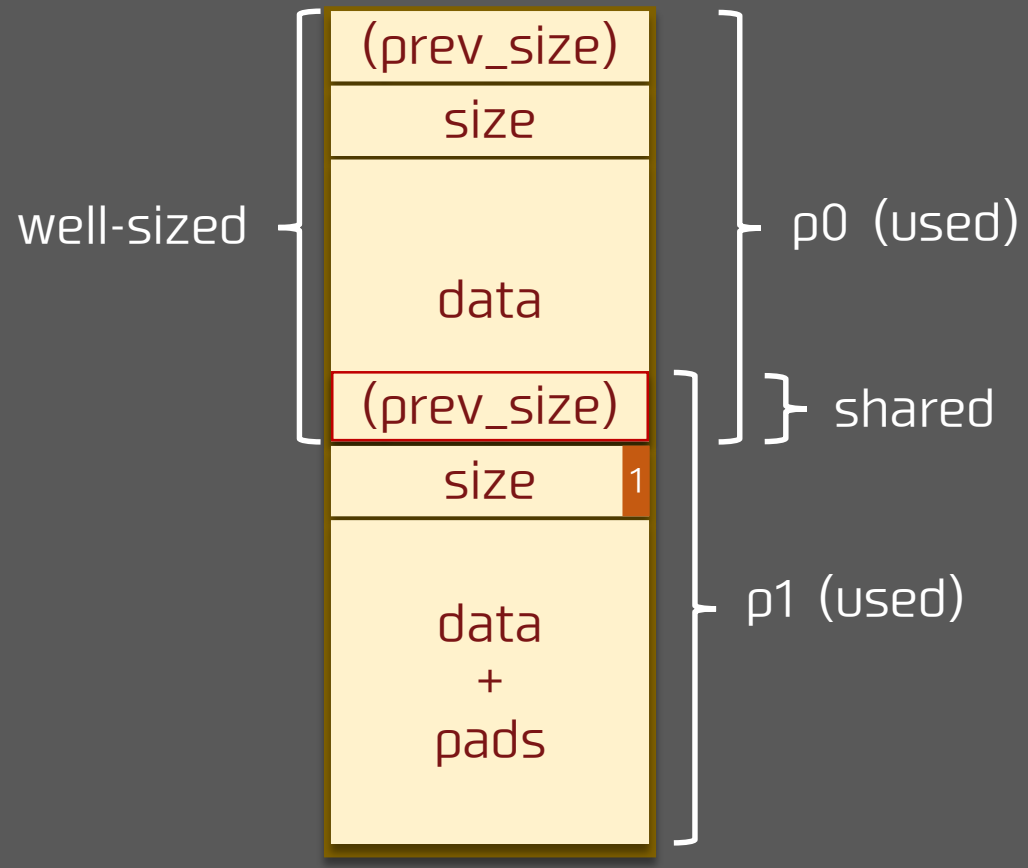
House of Einherjar Flaw / Flow

- Our current knowledge
  - "p->prev_size" can be shared with previous contiguous chunk.
  - PREV_INUSE bit of "p->size" decides whether the two contiguous chunks will be consolidated or not.
  - New location of p depends on "p->prev_size".
    - "p = chunk_at_offset(p, -((long)prevsize))"

House of Einherjar Flaw / Flow

- Our current knowledge
  - "p->prev_size" can be shared with previous contiguous chunk.
  - PREV_INUSE bit of "p->size" decides whether the two contiguous chunks will be consolidated or not.
  - New location of p depends on "p->prev_size".
    - "p = chunk_at_offset(p, -((long)prevsize))"
- Assumptions for House of Einherjar
  - Three chunks.
    - p0: the well-sized chunk(includes p1->prev_size).
    - p1: the small bin sized chunk.
    - (p2: the chunk to prevent from calling malloc_consolidate()).
  - p0 will be Off-by-one(OBO) poisoned by NUL byte('¥0').

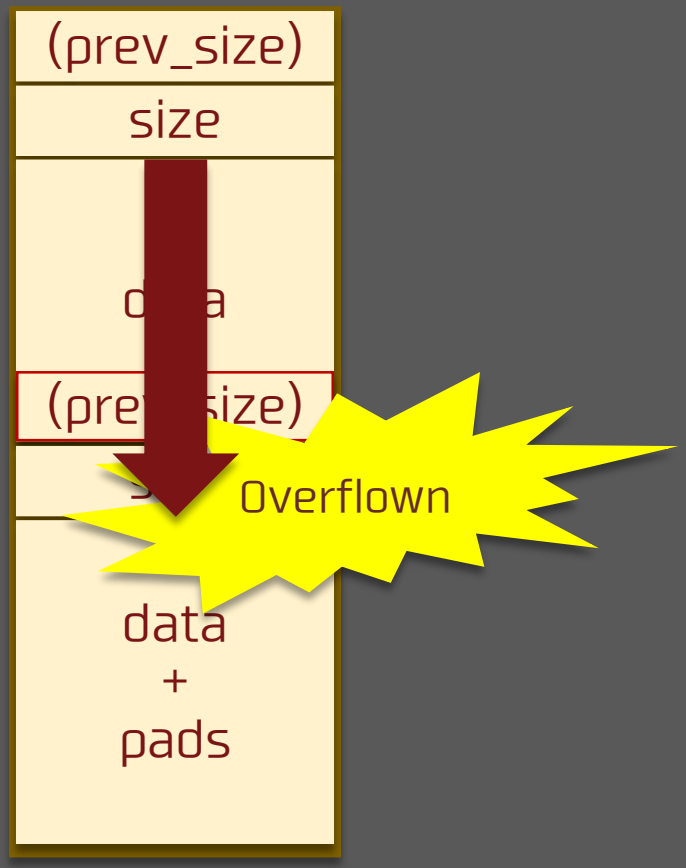
House of Einherjar Flaw / Flow



(a) Before overflowing

Fig. 5 The House of Einherjar

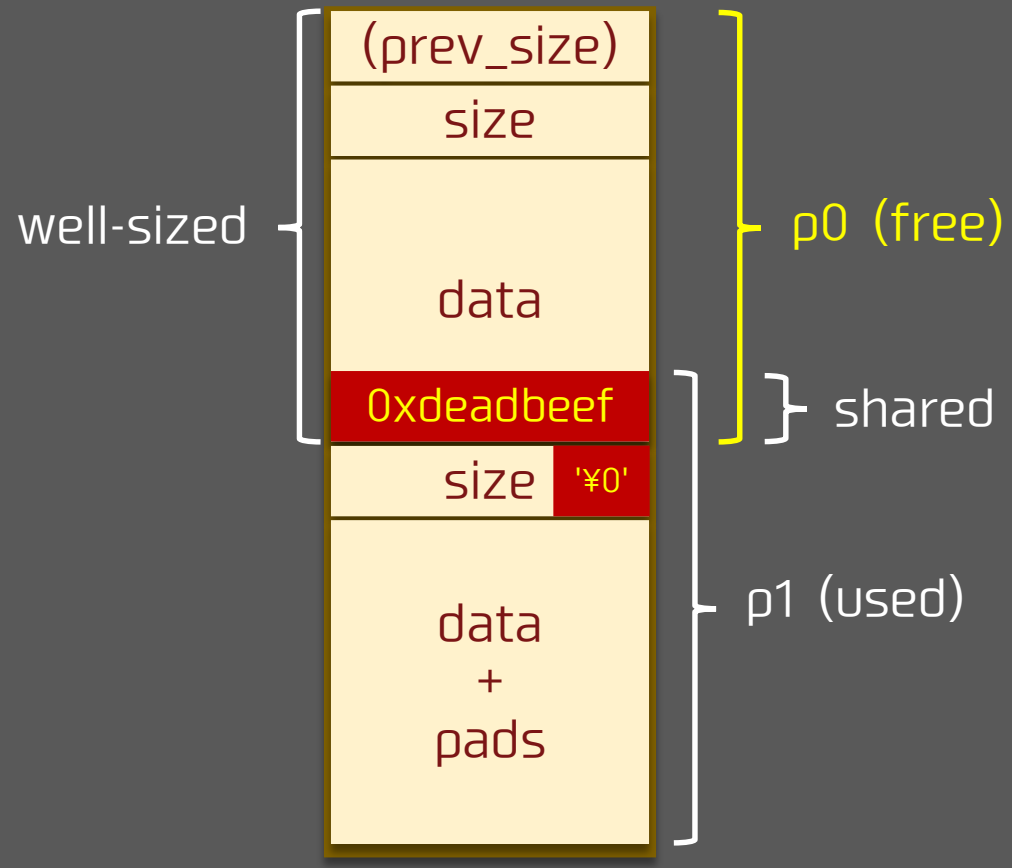
House of Einherjar Flaw / Flow



(b) Overflowing

Fig. 5 The House of Einherjar

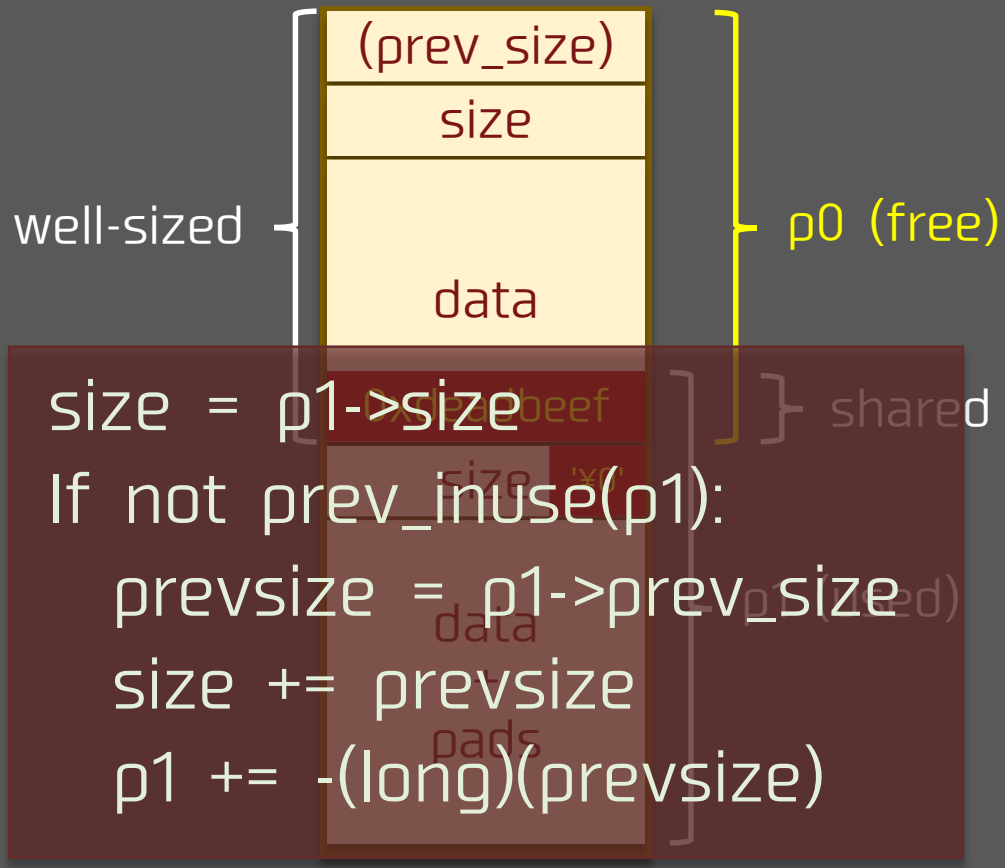
House of Einherjar Flaw / Flow



(c) After overflowing

Fig. 5 The House of Einherjar

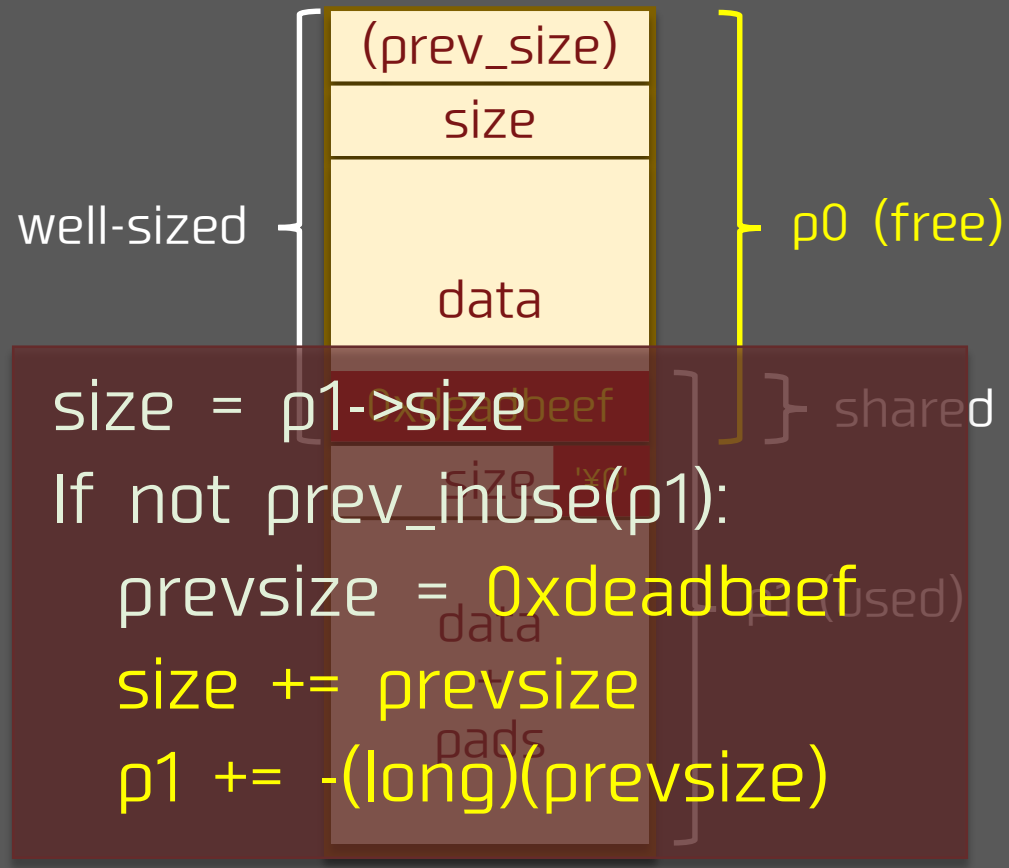
House of Einherjar Flaw / Flow



(c) After overflowing

Fig. 5 The House of Einherjar

House of Einherjar Flaw / Flow



(c) After overflowing

Fig. 5 The House of Einherjar

House of Einherjar Flaw / Flow

- How to enter into House of Einherjar
  - The well-sized chunk will occur OBO Overflow into the next chunk.
  - We can put a fake chunk near the target area.
    - For easy, we should make fd and bk members of fake chunk to point to the fake chunk's self.
  - We have to be able to calculate the diff between the target area and "p1".
    - Leaking the two addresses is required.
  - We have to be able to fix "p1->size" broken by free()'ing.
    - On the assumption that we can write to the fake chunk anytime.

Demo

http://ux.nu/6Rv6h

House of Einherjar Evaluation

- Merit
  - It depends on application's memory layout but only OBO Overflow is required
  - Huge malloc() like "House of Force" is not required.
- Demerit
  - The target area will be limited on the location of the fake chunk.
  - The leaking the two addresses is necessary.
- Evaluation: *"Not so bad"*

House of Einherjar Countermeasures

- "struct malloc_chunk" is NOT good
  - "chunk->prev_size" SHOULD NOT be overwritable by normal writes to a chunk.
  - It uses Boundary Tag Algorithm. (It is what it is!)
- Countermeasures?
  - Address checking
    - Is the consolidated chunk address valid?
    - Stack and heap address spaces are completely different.
      - It is possible to save a return address.
      - But that cannot be the solution for House of Einherjar to heap address space.

Thank You For Your Attention!
Any Questions?