Lecture 14:
Heap Security:
Principles and Techniques

Sanchuan Chen

schen@auburn.edu

11/9/2023

Why Heap Security

> “**Heap exploitation** remains the most common and critical security problems in system software.” [1]

Heap related vulnerabilities took nearly 53% of security problems in Microsoft products in 2017 [2].

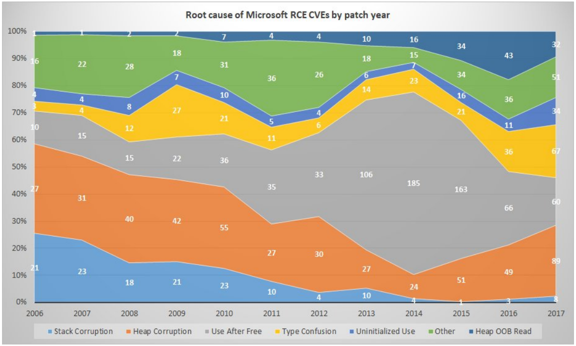


Figure: Root cause of Microsoft RCE CVEs by patch year

Why Heap Security

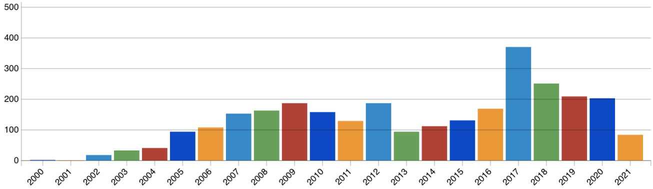


Figure: Heap-related CVEs by year

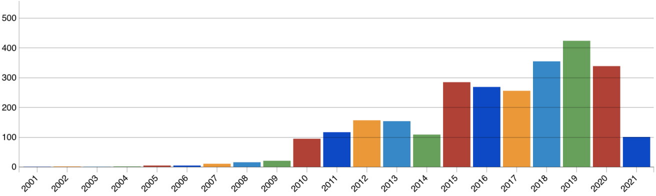


Figure: use-after-free CVEs by year

Why Heap Security

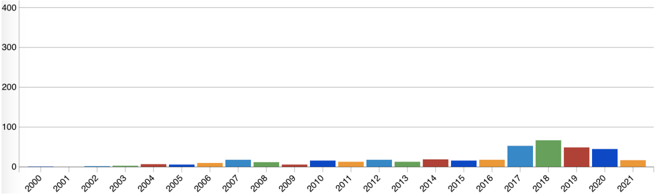


Figure: double-free CVEs by year

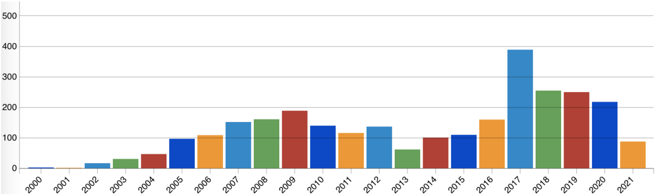


Figure: Heap-overflow CVEs by year

Why Heap Security

Why heap vulnerabilities being a popular target:

- Heap exploitation **tends to be application-independent**
- Heap vulnerabilities are powerful to **bypass modern mitigation schemes** (such as ASLR and DEP).

Why Heap Security

Target programs	Before ASLR		
	02-04	05-07	Total
Scriptable	0	12	12
Non-scriptable	9	7	16
(via heap exploit techs)	12	12	**24**

Table: The number of exploitations that lead to code execution from heap vulnerabilities in exploit-db (before ASLR). Source [1]

Target programs	After ASLR				
	08-10	11-13	14-16	17-19	Total
Scriptable	13	29	11	4	57
Non-scriptable	5	1	3	2	11
(via heap exploit techs)	3	4	1	2	**10**

Table: The number of exploitations that lead to code execution from heap vulnerabilities in exploit-db (after ASLR). Source [1]

Heap Exploitation History

2001	Once upon a free() [3]
2003	Advanced Doug lea's malloc exploits [4]
2004	Exploiting the wilderness [5]
2007	The use of set_head to defeat the wilderness [6]
2007	Understanding the heap by breaking it [7]
2009	Yet another free() exploitation technique [8]
2009	Malloc Des-Maleficarum [9]
2010	The house of lore: Reloaded [10]
2014	The poisoned NUL byte, 2014 edition [11]
2015	Glibc adventures: The forgotten chunk [12]
2016	Ptmalloc fanzine [13]
2016	New exploit methods against Ptmalloc of Glibc [14]
2016	House of Einherjar [15]
2020	Automatically Discover Heap Exploitation Primitives [16]

Table: Timeline for new heap exploitation technique discovered and their count in parentheses

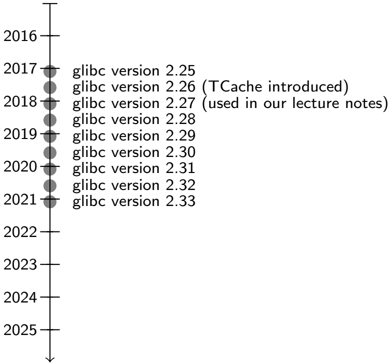
Memory Allocators

The examples of existing allocators are shown in Table 4.

Allocators	Description
ptmalloc2	A default allocator in Linux.
dlmalloc	An allocator that ptmalloc2 is based on.
jemalloc	A default allocator in FreeBSD.
tcmalloc	A high-performance allocator from Google.
mimalloc	An allocator from Microsoft.
musl	An allocator for embedded systems.
PartitionAlloc	A default allocator in Chromium.
libumem	A default allocator in Solaris.
Diehard [17]	A probabilistic allocator for unsafe languages
Dieharder [18]	A secure heap allocator
FreeGuard [19]	A faster secure heap allocator
Guarder [20]	A tunable secure allocator

Table: 4: Memory allocators.

Memory Allocators



Key Principles in Heap Design

Two Key Objectives

- Higher performance
- Less fragmentation

Key Principles in Heap Design

Two Key Design Principles

- Using binning to avoid being too fragmented
- Using in-place metadata (IPM) for fast lookup

Allocators	Binning	IPM	Description (applications)
ptmalloc2	✓	✓	A default allocator in Linux.
dlmalloc	✓	✓	An allocator that ptmalloc2 is based on.
jemalloc	✓		A default allocator in FreeBSD.
tcmalloc	✓	✓	A high-performance allocator from Google.
PartitionAlloc	✓		A default allocator in Chromium.
libumem	✓		A default allocator in Solaris.

Key Principles in Heap Design

Binning

Many memory allocators use size-basd classification, called *binning*.

- A whole size range is divided into multiple groups to manage memory blocks according to their size groups.
- Small size blocks focus on performance.
- Large size blocks focus on memory usage of the allocators.
- The allocator finds the smallest but sufficient block for given request as best-fit block.
- The allocator only scans blocks in proper size group instead of scanning all memory blocks.

Key Principles in Heap Design

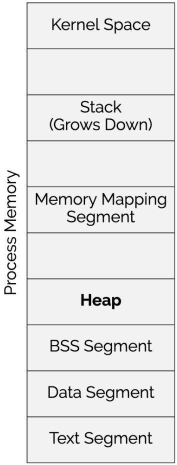
In-place Metadata

Many memory allocators place metadata near the payload, called in-place metadata. The meta-data, which is mostly pointers or size-related values used for allocator's data structures, is essential for fast look up and memory usage.

- Shortcoming: security problems from corrupted metadata in the presence of memory corruption bugs.
- Benefit: increased locality so that allocator can get benefit from the cache resulting in performance improvement.

Heap Management

Heap is a region of memory created at runtime to allocate variables whose size cannot be determined at compile time.



Heap Management

Primary public functions include malloc function and free function. The following is the description in source code.

malloc/malloc.c

```
/*
  malloc(size_t n)
  Returns a pointer to a newly allocated chunk of at least n bytes, or
  null if no space is available. Additionally, on failure, errno is
  set to ENOMEM on ANSI C systems.

  If n is zero, malloc returns a minumum-sized chunk. (The minimum
  size is 16 bytes on most 32bit systems, and 24 or 32 bytes on 64bit
  systems.)  On most systems, size_t is an unsigned type, so calls
  with negative arguments are interpreted as requests for huge amounts
  of space, which will often fail. The maximum supported value of n
  differs across systems, but is in all cases less than the maximum
  representable value of a size_t.
*/
```

Heap Management

malloc/malloc.c

```
/*
  free(void* p)
  Releases the chunk of memory pointed to by p, that had been previously
  allocated using malloc or a related routine such as realloc.
  It has no effect if p is null. It can have arbitrary (i.e., bad!)
  effects if p has already been freed.

  Unless disabled (using mallopt), freeing very large spaces will
  when possible, automatically trigger operations that give
  back unused memory to the system, thus reducing program footprint.
*/
```

Heap Management

How to use malloc and free functions:

```
/* Dynamically allocate 16 bytes */
char *buf = (char *)malloc(10);
/* Copy string to dynamically allocated memory */
strcpy(buf, "string");
/* Print content of dynamically allocated memory */
printf("%s\n", buf);
/* Free dynamically allocated memory */
free(buf);
```

Heap Memory Allocation

Glibc `malloc` function uses system calls to obtain memory from the Operating System. Particularly, `malloc` function invokes either `brk(2)` or `mmap(2)` system call to obtain memory [21].

`brk(2)` system call obtains memory from kernel by increasing program break (`brk(2)`). Initially, the start (`start_brk`) of heap segment and end (`brk(2)`) of heap segment point to same memory address.

`mmap(2)` system call create a private anonymous mapping segment, which is used as newly allocated heap memory.

Heap Memory Allocation

Based these two system calls, there are quite a few memory allocator available for heap memory allocation [22].

- dlmalloc (General purpose allocator)
- ptmalloc2 (glibc)
- jemalloc (FreeBSD and Firefox)
- tcmalloc (Google)
- libumem (Solaris)

Particularly, ptmalloc2 is a fork of dlmalloc, with threading support and released in 2006. ptmalloc2 has become the default memory allocator for Linux and we will focus on ptmalloc2 in our lectures.

Arena, Bin, and Chunk

Arena is the area where dynamic runtime memory is stored, containing both used and unused memory.

In ptmalloc2, arena is a per-thread data structure. The arena for the main thread is called main arena.

In glibc source code, malloc_state is the arena header.

Arena, Bin, and Chunk

malloc/malloc.c

```
struct malloc_state
{
  /* Serialize access.  */
  __libc_lock_define (, mutex);

  /* Flags (formerly in max_fast).  */
  int flags;

  /* Set if the fastbin chunks contain recently inserted free blocks.  */
  /* Note this is a bool but not all targets support atomics on booleans.  */
  int have_fastchunks;

  /* Fastbins */
  mfastbinptr fastbinsY[NFASTBINS];

  /* Base of the topmost chunk -- not otherwise kept in a bin */
  mchunkptr top;

  /* The remainder from the most recent split of a small request */
  mchunkptr last_remainder;
```

Arena, Bin, and Chunk

```

  /* Normal bins packed as described above */
  mchunkptr bins[NBINS * 2 - 2];

  /* Bitmap of bins */
  unsigned int binmap[BINMAPSIZE];

  /* Linked list */
  struct malloc_state *next;

  /* Linked list for free arenas.  Access to this field is serialized
     by free_list_lock in arena.c.  */
  struct malloc_state *next_free;

  /* Number of threads attached to this arena.  0 if the arena is on
     the free list.  Access to this field is serialized by
     free_list_lock in arena.c.  */
  INTERNAL_SIZE_T attached_threads;

  /* Memory allocated from the system in this arena.  */
  INTERNAL_SIZE_T system_mem;
  INTERNAL_SIZE_T max_system_mem;
};
```

Arena, Bin, and Chunk

One arena can have multiple heaps, each heap has a heap_info data structure, which is the heap header.

malloc/arena.c

```
/* A heap is a single contiguous memory region holding (coalesceable)
   malloc_chunks.  It is allocated with mmap() and always starts at an
   address aligned to HEAP_MAX_SIZE.  */

typedef struct _heap_info
{
  mstate ar_ptr; /* Arena for this heap. */
  struct _heap_info *prev; /* Previous heap. */
  size_t size;   /* Current size in bytes. */
  size_t mprotect_size; /* Size in bytes that has been mprotected
                           PROT_READ|PROT_WRITE.  */
  /* Make sure the following data is properly aligned, particularly
     that sizeof (heap_info) + 2 * SIZE_SZ is a multiple of
     MALLOC_ALIGNMENT. */
  char pad[-6 * SIZE_SZ & MALLOC_ALIGN_MASK];
} heap_info;
```

Arena, Bin, and Chunk

Bins are lists that store free chunks for quickly finding suitable chunks to satisfy allocation requests.

In-use chunks are not tracked by the arena, and the free chunks are put in bins. Multiple bins exist:

- TCache bin
- Fast bin
- Unsorted bin
- Small bin
- Large bin

Arena, Bin, and Chunk

Chunk is the smallest memory management unit in dynamic allocation.

In the source code, malloc_chunk is the chunk header and defined as follows:

```
/*
   malloc_chunk details:

    (The following includes lightly edited explanations by Colin Plumb.)

    Chunks of memory are maintained using a ‘boundary tag’ method as
    described in e.g., Knuth or Standish.  (See the paper by Paul
    Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
    survey of such techniques.)  Sizes of free chunks are stored both
    in the front of each chunk and at the end.  This makes
    consolidating fragmented chunks into bigger chunks very fast.  The
    size fields also hold bits representing whether chunks are free or
    in use.

    An allocated chunk looks like this:


```

Arena, Bin, and Chunk

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
    user.  "Nextchunk" is the beginning of the next contiguous chunk.

    Chunks always begin on even word boundaries, so the mem portion
    (which is returned to the user) is also on an even word boundary, and
    thus at least double-word aligned.
```

Arena, Bin, and Chunk

```
    Free chunks are stored in circular doubly-linked lists, and look like this:

    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                     |A|0|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Forward pointer to next chunk in list             |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Back pointer to previous chunk in list            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Unused space (may be 0 bytes long)                .
            .                                                               .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `foot:' |             Size of chunk, in bytes                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|0|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    The P (PREV_INUSE) bit, stored in the unused low-order bit of the
    chunk size (which is always a multiple of two words), is an in-use
    bit for the *previous* chunk.  If that bit is *clear*, then the
    word before the current chunk size contains the previous chunk
    size, and can be used to find the front of the previous chunk.
    The very first chunk allocated always has this bit set,
    preventing access to non-existent (or non-owned) memory. If
    prev_inuse is set for any given chunk, then you CANNOT determine
    the size of the previous chunk, and might even get a memory
    addressing fault when trying to do so.
```

Arena, Bin, and Chunk

```

    The A (NON_MAIN_ARENA) bit is cleared for chunks on the initial,
    main arena, described by the main_arena variable.  When additional
    threads are spawned, each thread receives its own arena (up to a
    configurable limit, after which arenas are reused for multiple
    threads), and the chunks in these arenas have the A bit set.  To
    find the arena for a chunk on such a non-main arena, heap_for_ptr
    performs a bit mask operation and indirection through the ar_ptr
    member of the per-heap header heap_info (see arena.c).
```

Note that in practice not all fields contain the corresponding data, for instance, for different bins, the layout of chunks are different, which we will see later.

Key Principles in Heap Exploitation

Heap exploitation relies on a key **writing to arbitrary places with arbitrary values** primitive, which can often be achieved through the following steps:

1. Corrupt heap metadata first, using the combinations of **heap transactions** [23]
2. Corrupt security critical data (e.g., GOT, return addresses, function pointers) using corrupted meta-data (which enables the primitive of *writing to arbitrary places with arbitrary values*)
3. Bypass condition checks if necessary or possible, with crafted metadata

Heap Exploitation Techniques

2001	(1) Once upon a free() [3]
2003	(1) Advanced Doug lea's malloc exploits [4]
2004	(2) Exploiting the wilderness [5]
2007	(2) The use of set_head to defeat the wilderness [6]
2007	(3) Understanding the heap by breaking it [7]
2009	(1) Yet another free() exploitation technique [8]
2009	(6) Malloc Des-Maleficarum [9]
2010	(2) The house of lore: Reloaded [10]
2014	(1) The poisoned NUL byte, 2014 edition [11]
2015	(2) Glibc adventures: The forgotten chunk [12]
2016	(2) Ptmalloc fanzine [13]
2016	(3) New exploit methods against Ptmalloc of Glibc [14]
2016	(1) House of Einherjar [15]
2020	(5) Automatically Discover Heap Exploitation Primitives [16]

Table: Timeline for new heap exploitation technique discovered and their count in parentheses

Heap Exploitation Techniques

Name	Description
Fast bin dup	Corrupting a fast bin freelist (e.g., by double free or write-after free) to return an arbitrary location
Unsafe unlink	Abusing unlinking in a freelist to get arbitrary write
House of spirit	Freeing a fake chunk of fast bin to return arbitrary location
Poison null byte	Corrupting heap chunk size to consolidate chunks even in the presence of allocated heap
House of lore	Abusing the small bin freelist to return an arbitrary location
Overlapping chunks	Corrupting a chunk size in the unsorted bin to overlap with an allocated heap
House of force	Corrupting the top chunk to return an arbitrary location
Unsorted bin attack	Corrupting a freed chunk in unsorted bin to write a uncontrollable value to arbitrary location
House of einherjar	Corrupting PREV_IN_USE to consolidate chunks to return an arbitrary location that requires a heap address
Unsorted bin into stack	Abusing the unsorted freelist to return an arbitrary location
House of unsorted einherjar	A variant of house of einherjar that does not require a heap address
Unaligned double free	Corrupting a small bin freelist to return already allocated heap
Overlapping small chunks	Corrupting a chunk size in a small bin to overlap chunks
Fast bin into other bin	Corrupting a fast bin freelist and usemalloc_consolidate() to return an arbitrary non-fast-bin chunk

Table: Modern heap exploitation techniques

Heap Transactions

A set of operations modifies the heap, and we can define a *heap transaction* [23] as an operation that modifies the heap directly or indirectly. There could be up to 6 heap transactions.

Heap Transactions

- **malloc (M)**, which is used to allocate memory.
- **free (F)**, which is used to deallocate memory.
- **overflow (O)**. An overflow transaction is a transaction that has an out-of-bounds write into a buffer.
  - Mostly, the memory overwritten is another chunk adjacent in memory.
  - Common cause 1: missing boundary check.
  - Common cause 2: a bug in the determination of the allocation size, usually due to an integer overflow.
- **use-after-free (UAF)** The use-after-free transaction is an access to memory that has been freed.
  - Read: resulting in an information leak.
  - Write: data stored inside the freed chunk manipulated by an attacker.

Heap Transactions

- **double-free (DF)** The double-free transaction is that a memory chunk is freed twice.
  - The chunk is stored inside allocator's internal structures for freed chunks twice.
  - Leading to further corruption of the heap structure.
- **fake-free (FF)** The fake-free transaction is that an attacker controls the parameter passed to free, making it point to a controlled region, which is a fake allocated chunk.
  - This could potentially lead to future allocations returning the maliciously fake chunk.

Example: Unsafe Unlink Attack on Heap

One of the most famous heap exploitation techniques is the *unsafe unlink attack* that abuses the unlink mechanism of double-linked lists in heap allocators.

Unsafe Unlink Attack

The following code snippet is from glibc version 2.3.3.

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  FD->bk = BK;
  BK->fd = FD;
}
```

By modifying a forward point (P$\rightarrow$fd) into a desired encoded location and a backward point (P$\rightarrow$bk) into a desired value, attackers can achieve **arbitrary writes** (P$\rightarrow$fd$\rightarrow$bk $=$ P$\rightarrow$bk).

Example: Unsafe Unlink Attack on Heap

P in the aforementioned code is an allocated chunk, and the definition in glibc version 2.3.3 is as follows:

```
struct malloc_chunk {

  INTERNAL_SIZE_T      mchunk_prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      mchunk_size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

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

Example: Unsafe Unlink Attack on Heap

The double-linked list is shown in Figure. 6. The chunk in the middle is P and the forward chunk is pointed by P$\rightarrow$fd and the backward chunk is pointed by P$\rightarrow$bk.

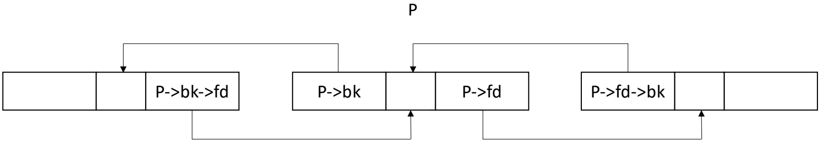


Figure: 6: A double-linked list with three chunks

Example: Unsafe Unlink Attack on Heap

In order to unlink the chunk P, P→fd→bk needs to point to P→bk and P→bk→fd needs to point to P→fd, as shown in Figure. 7.

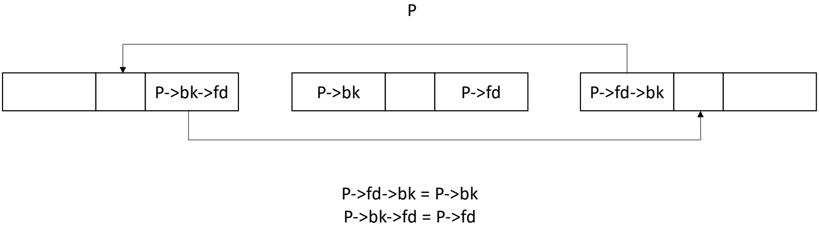


Figure: 7: Unlink chunk P

Example: Unsafe Unlink Attack on Heap

As we can see, during the unlink process, P→fd→bk is overwritten with the value in P→bk. If an attacker can modify the value of P→fd and P→bk in chunk P, and then the attacker can achieve arbitrary write primitive, as shown in Figure. 8.

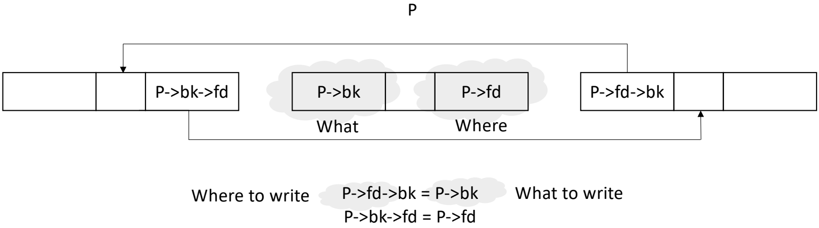


Figure: 8: Unsafe unlink of chunk P

Mitigation of Unsafe Unlink Attack

The following code introduced in glibc version 2.3.4 adds checks about the pointers.

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

Mitigation of Unsafe Unlink Attack

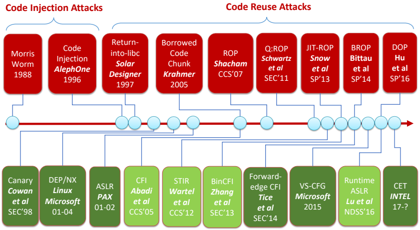
glibc version 2.26 further adds a check on the previous chunk size, as shown in the following.

```
/* Take a chunk off a bin list */
#define unlink(AV, P, BK, FD) {                                              \
    if (__builtin_expect (chunksize(P) != prev_size (next_chunk(P)), 0))      \
      malloc_printerr (check_action, "corrupted size vs. prev_size", P, AV);  \
    FD = P->fd;                                                               \
    BK = P->bk;                                                               \
    if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                     \
      malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
    else {                                                                    \
        FD->bk = BK;                                                          \
        BK->fd = FD;                                                          \
        if (!in_smallbin_range (chunksize_nomask (P))                         \
            && __builtin_expect (P->fd_nextsize != NULL, 0)) {                \
            if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)        \
                || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
              malloc_printerr (check_action,                                  \
                               "corrupted double-linked list (not small)",    \
```

Mitigation of Unsafe Unlink Attack

```
                                P, AV);                                   \
            if (FD->fd_nextsize == NULL) {                                \
                if (P->fd_nextsize == P)                                  \
                  FD->fd_nextsize = FD->bk_nextsize = FD;                 \
                else {                                                    \
                    FD->fd_nextsize = P->fd_nextsize;                     \
                    FD->bk_nextsize = P->bk_nextsize;                     \
                    P->fd_nextsize->bk_nextsize = FD;                     \
                    P->bk_nextsize->fd_nextsize = FD;                     \
                  }                                                       \
              } else {                                                    \
                P->fd_nextsize->bk_nextsize = P->bk_nextsize;             \
                P->bk_nextsize->fd_nextsize = P->fd_nextsize;             \
              }                                                           \
          }                                                               \
      }                                                                   \
}
```

Thank You



[1]

Q&A

schen@auburn.edu
schuan.github.io

[1] Instructor appreciates the help from Prof. Zhiqiang Lin.

I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1111–1128. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/yun

M. Miller, "A snapshot of vulnerability root cause trends for Microsoft Code Execution (RCE) CVEs, 2006 through 2017," https://twitter.com/epakskape/status/984481101937651713, (Accessed on 03/27/2021).

Anonymous, "Once upon a free()..." http://phrack.org/issues/57/9.html, (Accessed on 03/27/2021).

jp, "Advanced Doug lea's malloc exploits," http://phrack.org/issues/61/6.html, (Accessed on 03/27/2021).

- P. Phantasmagoria, "Exploiting the wilderness," https://seclists.org/vuln-dev/2004/Feb/25, (Accessed on 03/27/2021).
- g463, "The use of set_head to defeat the wilderness," http://phrack.org/issues/64/9.html, (Accessed on 03/27/2021).
- J. N. Ferguson, "Understanding the heap by breaking it," *black Hat USA*, pp. 1–39, 2007.
- huku, "Yet another free() exploitation technique," http://phrack.org/issues/66/6.html, (Accessed on 03/27/2021).
- blackngel, "Malloc des-maleficarum," http://phrack.org/issues/66/10.html, (Accessed on 03/27/2021).

——, "The house of lore: Reloaded," http://phrack.org/issues/67/8.html, (Accessed on 03/27/2021).

C. Evans and T. Ormandy, "The poisoned NUL byte, 2014 edition," https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html, (Accessed on 03/27/2021).

F. Goichon, "Glibc adventures: the forgotten chunks," https://www.contextis.com/en/resources/white-papers/glibc-adventures-the-forgotten-chunks, (Accessed on 03/27/2021).

I. Kurucsai, "ptmalloc fanzine," http://tukan.farm/2016/07/26/ptmalloc-fanzine/, (Accessed on 03/27/2021).

T. Xie, Y. Zhang, J. Li, H. Liu, and D. Gu, "New exploit methods against ptmalloc of glibc," in *2016 IEEE*

Trustcom/BigDataSE/ISPA. Los Alamitos, CA, USA: IEEE Computer Society, aug 2016, pp. 646–653. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TrustCom.2016.0121

st4g3r, "House of einherjar - yet another heap exploitation technique on GLIBC," https://github.com/st4g3r/House-of-Einherjar-CB2016, (Accessed on 03/27/2021).

I. Yun, D. Kapil, and T. Kim, "Automatic techniques to systematically discover new heap exploitation primitives," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1111–1128.

E. D. Berger and B. G. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA:

Association for Computing Machinery, 2006, p. 158–168. [Online]. Available: https://doi.org/10.1145/1133981.1134000

G. Novark and E. D. Berger, “Dieharder: Securing the heap,” ser. CCS ’10. New York, NY, USA: Association for Computing Machinery, 2010, p. 573–584. [Online]. Available: https://doi.org/10.1145/1866307.1866371

S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, “Freeguard: A faster secure heap allocator,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2389–2403. [Online]. Available: https://doi.org/10.1145/3133956.3133957

S. Silvestro, H. Liu, T. Liu, Z. Lin, and T. Liu, “Guarder: A tunable secure allocator,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 117–133. [Online]. Available:

https://www.usenix.org/conference/usenixsecurity18/presentation/silvestro

sploitfun, "Syscalls used by malloc," https://sploitfun.wordpress.com/2015/02/11/syscalls-used-by-malloc/, (Accessed on 03/14/2021).

——, "Understanding glibc malloc," https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/, (Accessed on 03/14/2021).

M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 99–116. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/eckert