SysSoft Security

Introduction to Heap Overflow Vulnerability

Sanjay Rawat

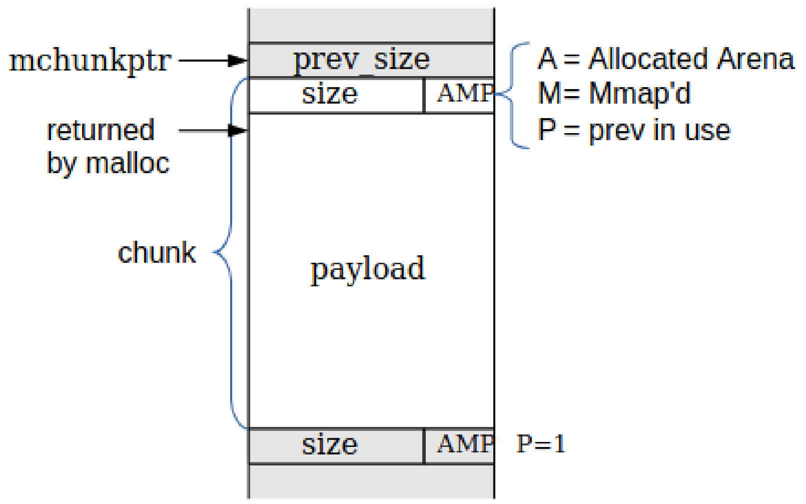
Heap

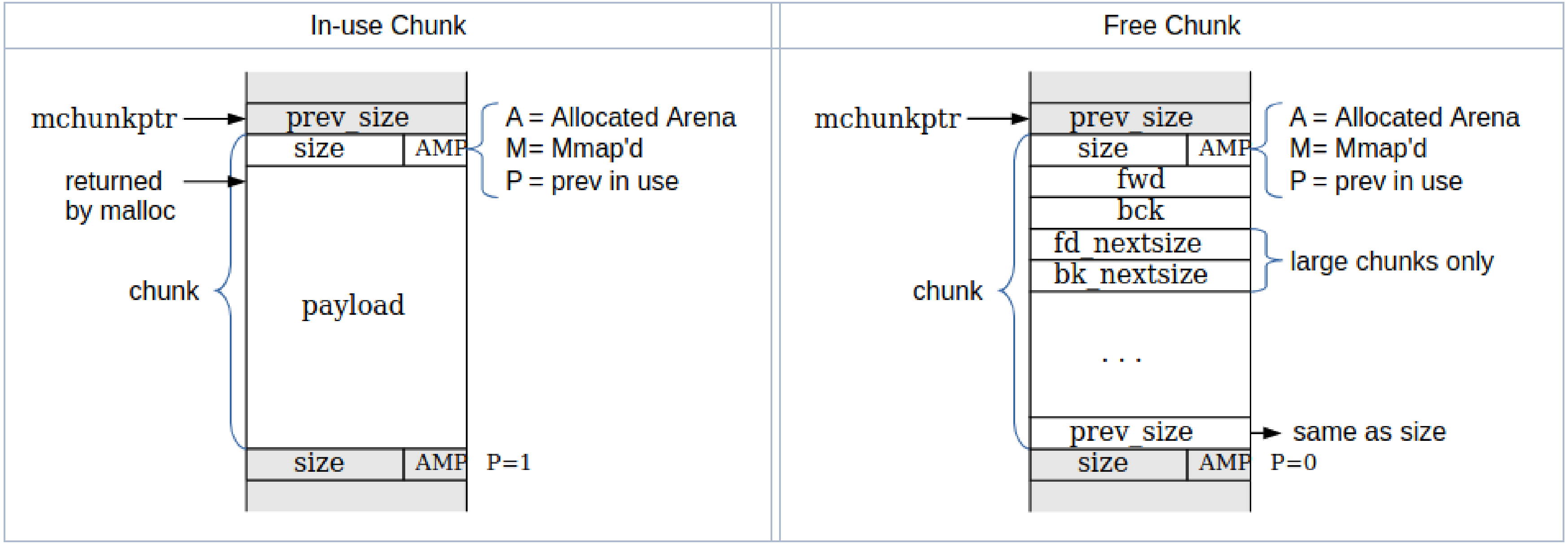
- Dynamic memory allocation
- Linux: Malloc(), alloc(), realloc(), free()
- Windows: HeapAlloc(), HeapFree(), …
- Heap consists of smaller chunks of free memory.
- These chinks are organized in bins, based on size.
- Modern malloc implementation is evolved a lot since early days of libc 2.x.
- In older versions, each of these chunks is represented by doubly linked-list type of structures containing a header and data section (Doug Lea’s malloc).

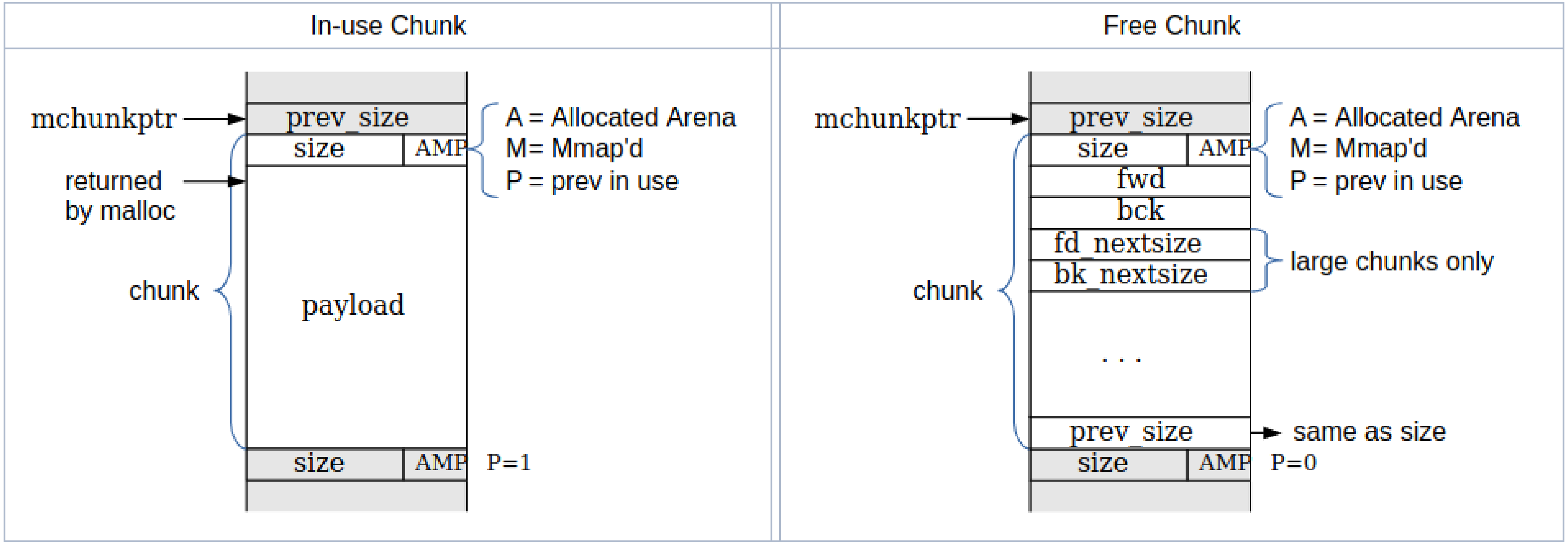
Chunk header

- When a chunk is in use: only `size` field is available
- When the chunk is free'd, the memory is re-purposed for pointers within linked lists, such that suitable chunks can quickly be found and re-used when needed.
- Also, the last word in a free'd chunk contains a copy of the chunk size
- Since all chunks are multiples of 8 bytes, the 3 LSBs of the chunk size can be used for flags. `A (0x04), M (0x02), P (0x01)`-Previous chunk is in use.

In-use Chunk


mchunkptr
prev_size
size
AMP
A = Allocated Arena
M= Mmap'd
P = prev in use
returned
by malloc
chunk
payload
size
AMP
P=1


In-use Chunk
Free Chunk
mchunkptr
prev_size
size
AMP
A = Allocated Arena
M= Mmap'd
P = prev in use
returned
by malloc
chunk
payload
P=1
mchunkptr
prev_size
size
AMP
fwd
bck
fd_nextsize
bk_nextsize
large chunks only
A = Allocated Arena
M= Mmap'd
P = prev in use
chunk
. . .
prev_size
same as size
size
AMP
P=0



Chunks are adjacent in memory → can be overflowed!

Simple Heap Overflow

```
//heap0-proto.c

struct data {
  char name[64];
};
struct fp {
  int (*fp)();
};
void winner()
{
  printf("level passed\n");
}
void nowinner()
{
  printf("level has not been passed\n");
}
int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;
  //printf("data is at %p, fp is at %p\n",
d, f);
  strcpy(d->name, argv[1]);

  f->fp();

}
```

** Example from Protostar heap bugs

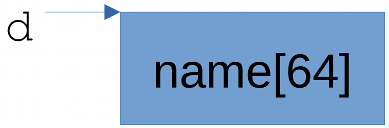
Simple Heap Overflow

```
//heap0-proto.c

struct data {
  char name[64];
};
struct fp {
  int (*fp)();
};
void winner()
{
  printf("level passed\n");
}
void nowinner()
{
  printf("level has not been passed\n");
}
int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;
  //printf("data is at %p, fp is at %p\n",
d, f);
  strcpy(d->name, argv[1]);

  f->fp();

}
```



** Example from Protostar heap bugs

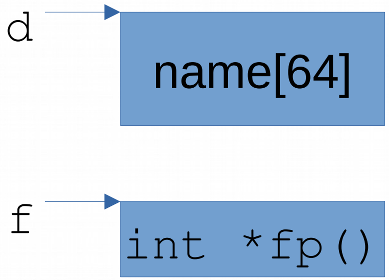
Simple Heap Overflow

```
//heap0-proto.c

struct data {
  char name[64];
};
struct fp {
  int (*fp)();
};
void winner()
{
  printf("level passed\n");
}
void nowinner()
{
  printf("level has not been passed\n");
}
int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;
  //printf("data is at %p, fp is at %p\n",
d, f);
  strcpy(d->name, argv[1]);

  f->fp();

}
```



** Example from Protostar heap bugs

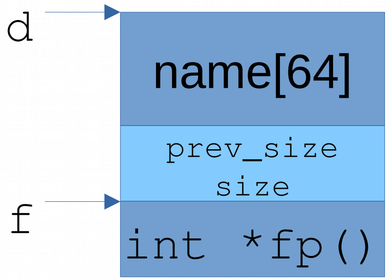
Simple Heap Overflow

```c
//heap0-proto.c

struct data {
  char name[64];
};
struct fp {
  int (*fp)();
};
void winner()
{
  printf("level passed\n");
}
void nowinner()
{
  printf("level has not been passed\n");
}
int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;
  //printf("data is at %p, fp is at %p\n",
d, f);
  strcpy(d->name, argv[1]);

  f->fp();

}
```



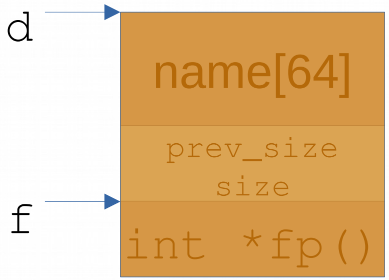
** Example from Protostar heap bugs

Simple Heap Overflow

```c
//heap0-proto.c

struct data {
  char name[64];
};
struct fp {
  int (*fp)();
};
void winner()
{
  printf("level passed\n");
}
void nowinner()
{
  printf("level has not been passed\n");
}
int main(int argc, char **argv)
{
  struct data *d;
  struct fp *f;
  d = malloc(sizeof(struct data));
  f = malloc(sizeof(struct fp));
  f->fp = nowinner;
  //printf("data is at %p, fp is at %p\n",
d, f);
  strcpy(d->name, argv[1]);

  f->fp();
```



** Example from Protostar heap bugs

Heap structure (older version/32-bit)

- ```
typedef struct __HeapHdr__{
    struct __HeapHdr__ *next; //4/8 bytes pointer to next available chunk
    struct __HeapHdr__ *prev; //4/8 bytes pointer to previous chunk
    unsigned int prev_size; //4 bytes to state size of the chunk
    unsigned int size/used; //4 bytes to indicates if the chunks is free (0) or in use (1)
// Data portion starts here
} HeapHdr_t;
```
```

Operations

- Melloc() assigns memory (**used** field is set).
- Free() releases the memory.
- Free() uses unlink() internally.
- free() merges adjacent free chunks of blocks.

Free() action (unlink)

- checks if the adjacent chunk is free (`if hdr-> next->used==0`).
- if so, add the sizes of both the chunks (`hdr-> size += hdr->next->size`).
- points its *next* pointer to the *next* pointer of the adjacent chunks (`hdr->next = hdr-> next->next`).
- Points the prev pointer of the next chunk of the adjacent chunk to the current chunk (`hdr-> next->next->prev=hdr->next->prev`).

Heap overflow

- Conditions:
  - The size is calculated wrongly (for example, using integer overflow) and therefore a smaller block is allocated than the required one;
  - When a call to free() is made, the block next to the block being deleted, must be unused (i.e. used bit should not be set).
  - there must be at least two calls to allocate memory on heap (heap grows towards higher memory!).

mechanism

	1-4 bytes	5-8 bytes	9-12 bytes	13-16 bytes
memory addr	next	prev	size	used
m_1	m_2	....	....	01...
m_2	m_3 (A)	m_1 (B)	....	....
m_3	m_4 (C)	m_2 (D)	....	E
m_4	F	G	....	....

- Where: m1 is 1st chunk of wrong size 0; assuming m2 and m3 can be overflowed because of wrong size.
- Recall:
  - `hdr-> next->next->prev=hdr->next-> prev`

- LHS: `hdr-> next->next->prev`
- `=> m1-> next -> next -> prev`
- `=>m2 -> next -> prev`
- `=> C+4`

- `RHS: hdr->next-> prev`

- `=> m1->next-> prev`
- `=> B`
- `we control C and D!! So, what can we do with it??`

	1-4 bytes	5-8 bytes	9-12 bytes	13-16 bytes
memory addr	next	prev	size	used
m_1	m_2	....	....	01...
m_2	m_3 (A)	m_1 (B)	....	....
m_3	m_4 (C)	m_2 (D)	....	E
m_4	F	G	....	....

- LHS: hdr-> next->next->prev
- => m1-> next -> next -> prev
- =>m2 -> next -> prev
- => C+4 where
- RHS: hdr->next-> prev
- => m1->next-> prev
- => B
- we control C and D!! So, what can we do with it??

	1-4 bytes	5-8 bytes	9-12 bytes	13-16 bytes
memory addr	next	prev	size	used
m_1	m_2	....	....	01...
m_2	m_3 (A)	m_1 (B)	....	....
m_3	m_4 (C)	m_2 (D)	....	E
m_4	F	G	....	....

- LHS: `hdr-> next->next->prev`
- `=> m1-> next -> next -> prev`
- `=>m2 -> next -> prev`
- `=> C+4` where
- `RHS: hdr->next-> prev`
- `=> m1->next-> prev`
- `=> B` what
- we control C and D!! So, what can we do with it??

	1-4 bytes	5-8 bytes	9-12 bytes	13-16 bytes
memory addr	next	prev	size	used
m_1	m_2	....	....	01...
m_2	m_3 (A)	m_1 (B)	....	....
m_3	m_4 (C)	m_2 (D)	....	E
m_4	F	G	....	....

The Present Implementation

- From malloc.c
  - New implementation: ptmalloc2 by Wolfram Gloger.
  - Sizes of free chunks are stored both in the front of each chunk and at the end.
  - The size fields also hold bits representing whether chunks are free or in use.

Features!

- The main properties of the algorithms are:
  - For large (>= 512 bytes) requests, it is a pure best-fit allocator, with ties normally decided via FIFO (i.e. least recently used).
  - For small (<= 64 bytes by default) requests, it is a caching allocator, that maintains pools of quickly recycled chunks.
  - In between, and for combinations of large and small requests, it does the best it can trying to meet both goals at once.
  - For very large requests (>= 128KB by default), it relies on system memory mapping facilities, if supported.

Allocated Header

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if allocated            | |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                       |M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk                                     |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Free chunks

- Stored in doubly-linked list

```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk                            |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
    `head:' |             Size of chunk, in bytes                         |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                           |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Observation

- With this new implementation, we can still do something !!!.
- Lets see an example… (`heap-metaof.c`)