Эксплуатация уязвимостей на переполнение буфера в куче

Игорь Черватюк

Александр Трифанов

Андрей Басарыгин

Москва, 2018

Введение

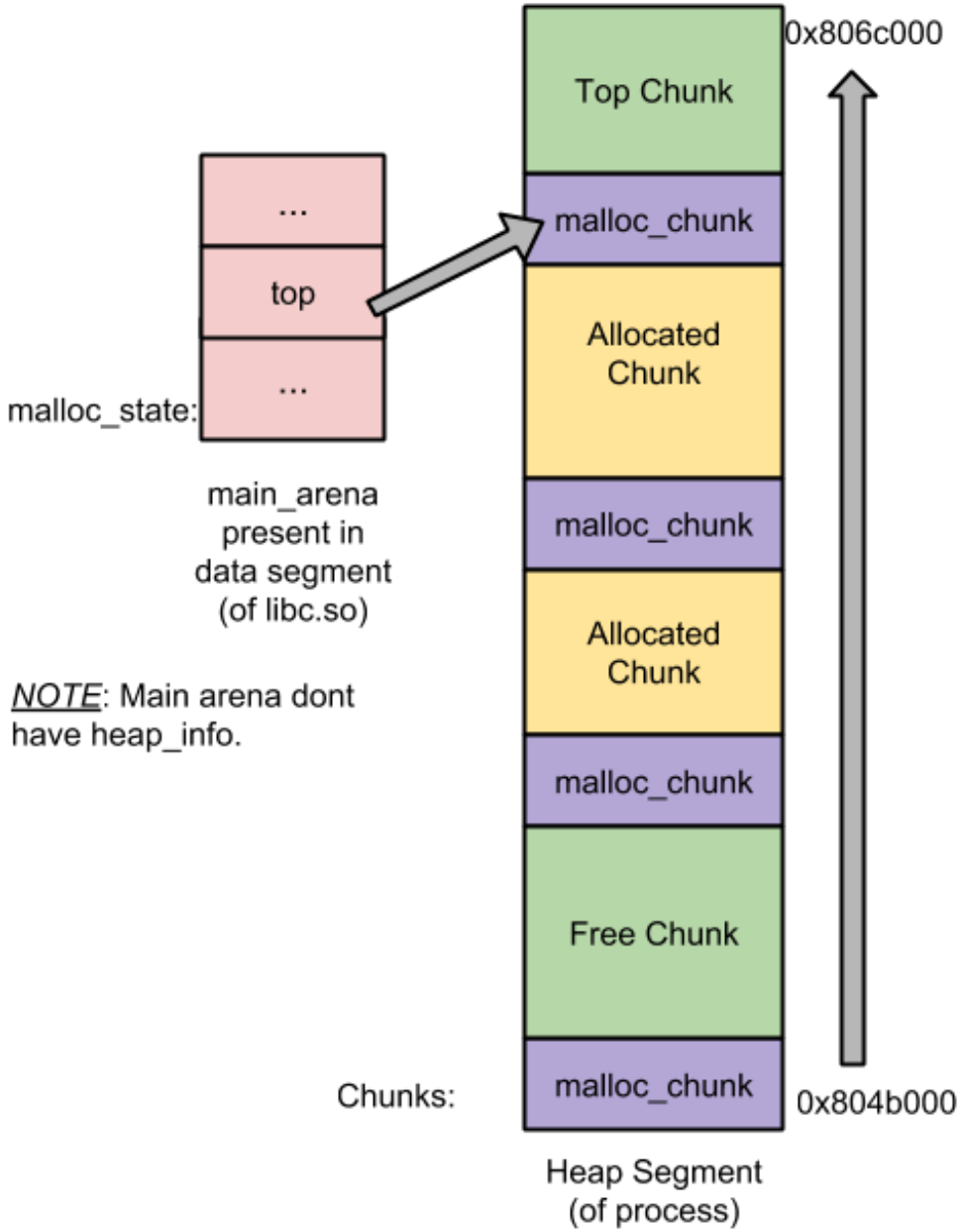
- Существует множество различных реализаций динамического выделения памяти.
  - dlmalloc – General purpose allocator
  - ptmalloc2 – glibc
  - jemalloc – FreeBSD and Firefox
  - tcmalloc – Google
  - libumem – Solaris
- Подход к эксплуатации каждой реализации индивидуален, мы постараемся выделить некоторые общие случаи.
- Размер имеет значение – от размера буфера зависит место в памяти в котором буфер будет размещён.

Динамическое распределение памяти (вся информация только о glibc)

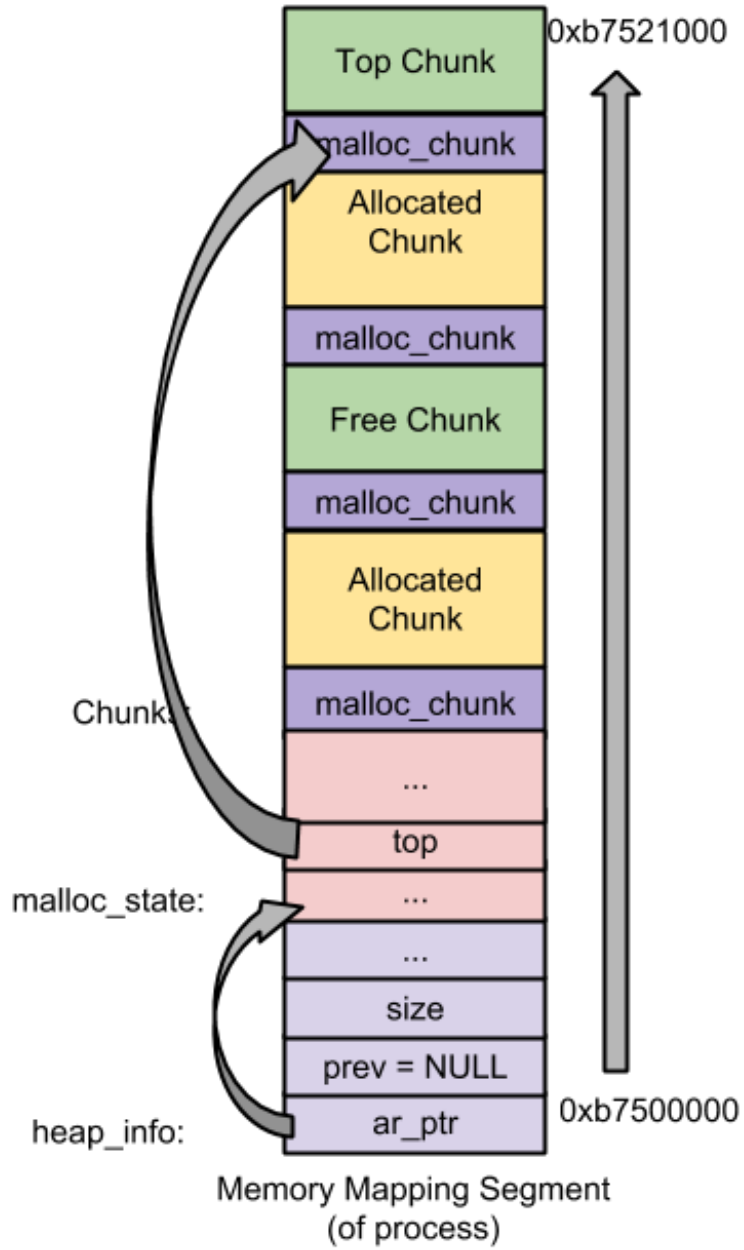
Main Arena		Thread 1 Arena	Thread 2 Arena
Heap	Heap	Heap	Heap
Chunk 1	Chunk 1	Chunk 1	Chunk 1
Chunk 2	Chunk 2	Chunk 2	Chunk 2
Chunk 3	Chunk 3	Chunk 3	Chunk 3
Chunk 4	Chunk 4	Chunk 4	Chunk 4
…	…	…	…
Chunk n	Chunk n	Chunk n	Chunk n

Fast bin	Unsorted bin
Chunk 1	Chunk 10
Chunk 3	Chunk 18
Chunk 8	Chunk 21

Число арен 2*количество ядер для 32 бит, 8*количество ядер для 64 бит.
Арена может содержать несколько куч (обычно минимум 1 куча на поток).
Куча состоит из участков памяти (чанков).
Чанк – «атом» динамической памяти, malloc выделяя память занимает чанк.
Корзины (bins) – хранят указатели на **свободные** чанки.



Main Arena

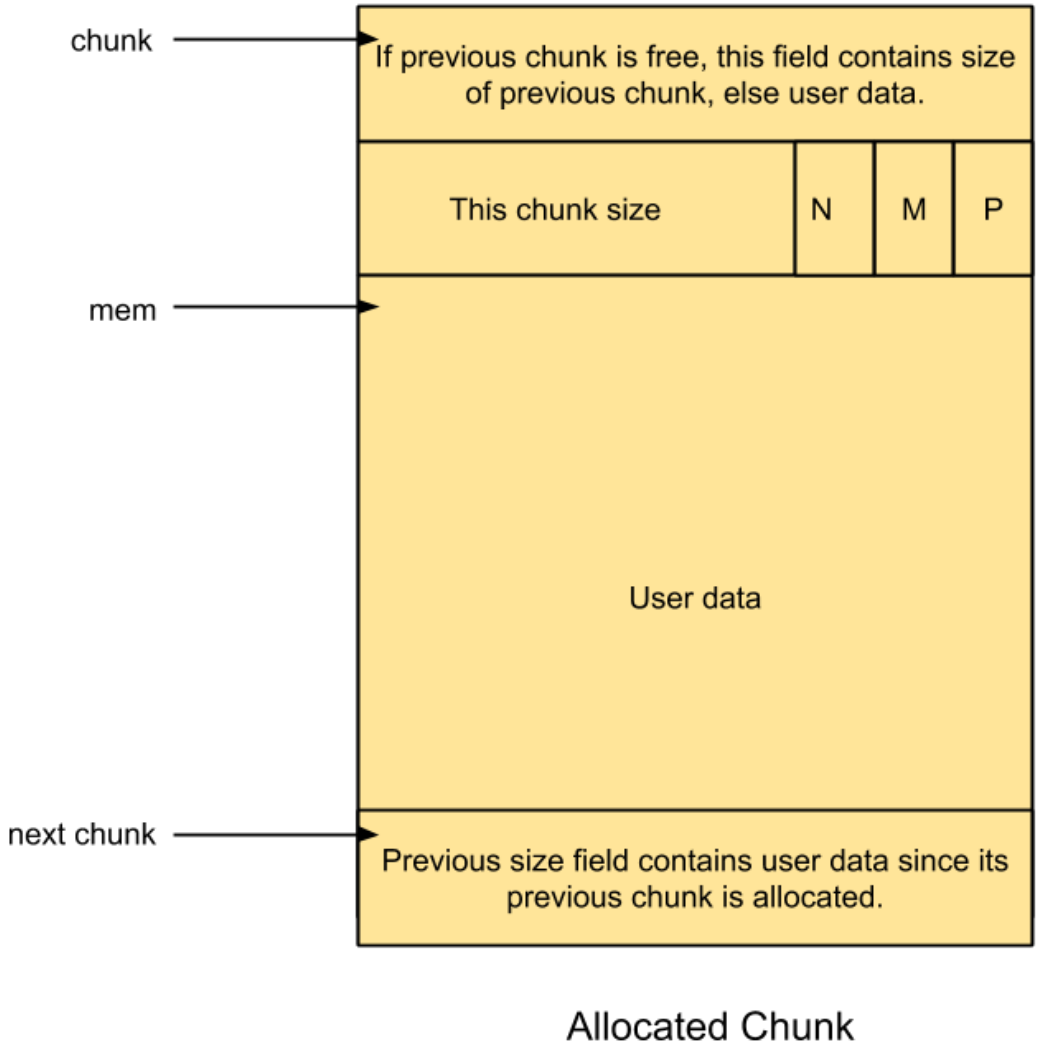


Thread Arena

Типы чанков (chunks)

- Allocated chunk
  - Не содержит указателей на соседние чанки
  - Содержит размер предыдущего чанка, если тот пуст
  - Содержит свой размер
  - Из размеров можно рассчитать указатели на соседние чанки
- Free chunk
  - Содержит указатели на соседние чанки
  - Содержит свой размер
  - Двух соседних свободных чанков не бывает – они объединяются в один
- Top chunk
  - Чанк, которых находится у верхней границы арены. Не входит ни в одну корзину. Используется когда нет свободных чанков. При выделении памяти может разделиться на два чанка (пользовательский и Last Remainder chunk)
- Last Remainder chunk
  - Становится новым Top chunk

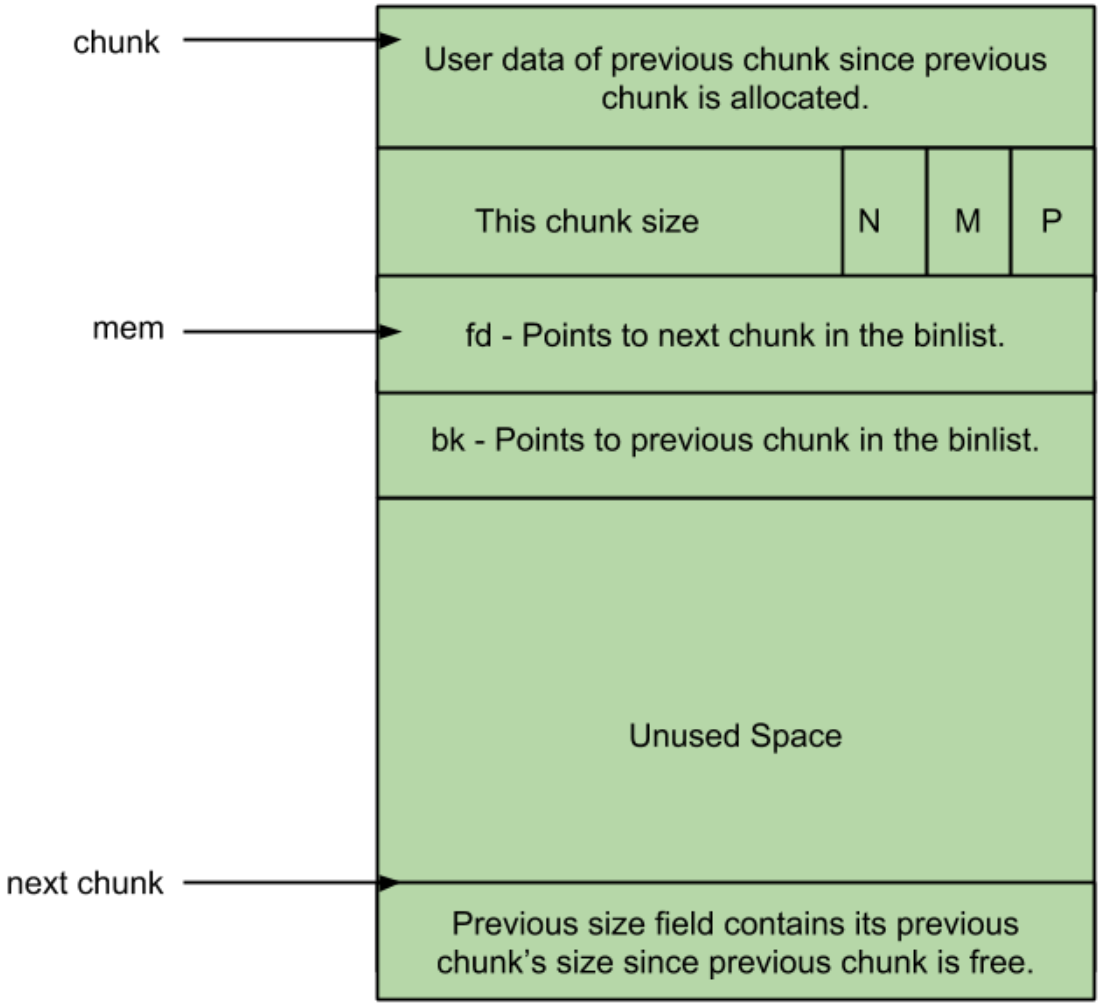
Allocated chunk



Allocated Chunk

- prev_size: Если предыдущий чанк свободен – это поле содержит его размер. Если занят – там находятся данные.
  size: размер Allocated chunk. Последние три бита – флаги:
  - PREV_INUSE (P) – равен 1, если предыдущий чанк занят
  - IS_MMAPPED (M) – равен 1, если к чанку применена функция mmap.
  - NON_MAIN_ARENA (N) – равен 1, если чанк принадлежит к thread arena.
- Указателей как во free chunk здесь нет.
- Запрашиваемый пользователем размер памяти конвертируется для выделения места под служебные данные (и становится кратным 8, чтобы последние 3 бита использовать под флаги).

Free Chunk



Free Chunk

- prev_size: Свободные чанки не могут быть соседями (исключение – быстрые чанки), поэтому предыдущий чанк занят и хранит данные пользователя.
  size: размер текущего чанка
  fd: Forward pointer – указатель на следующий чанк в той же корзине
  bk: Backward pointer – указатель на предыдущий чанк в той же корзине

Protostar heap0

```
1  struct data {
2    char name[64];
3  };
4  struct fp {
5    int (*fp)();
6  };
7  void winner() { printf("level passed\n"); }
8  void nowinner() { printf("level has not been passed\n"); }
9  int main(int argc, char **argv) {
10   struct data *d;
11   struct fp *f;
12   d = malloc(sizeof(struct data));
13   f = malloc(sizeof(struct fp));
14   f->fp = nowinner;
15   printf("data is at %p, fp is at %p\n", d, f);
16   strcpy(d->name, argv[1]);
17   f->fp();
18 }
```

- Две структуры данных
- В одной – есть указатель на функцию
- Если перезаписать этот указатель – можно выполнить произвольный код
- Почти ничем не отличается от эксплуатации буфера в стеке

Demo time

Типы корзин (Bin types)

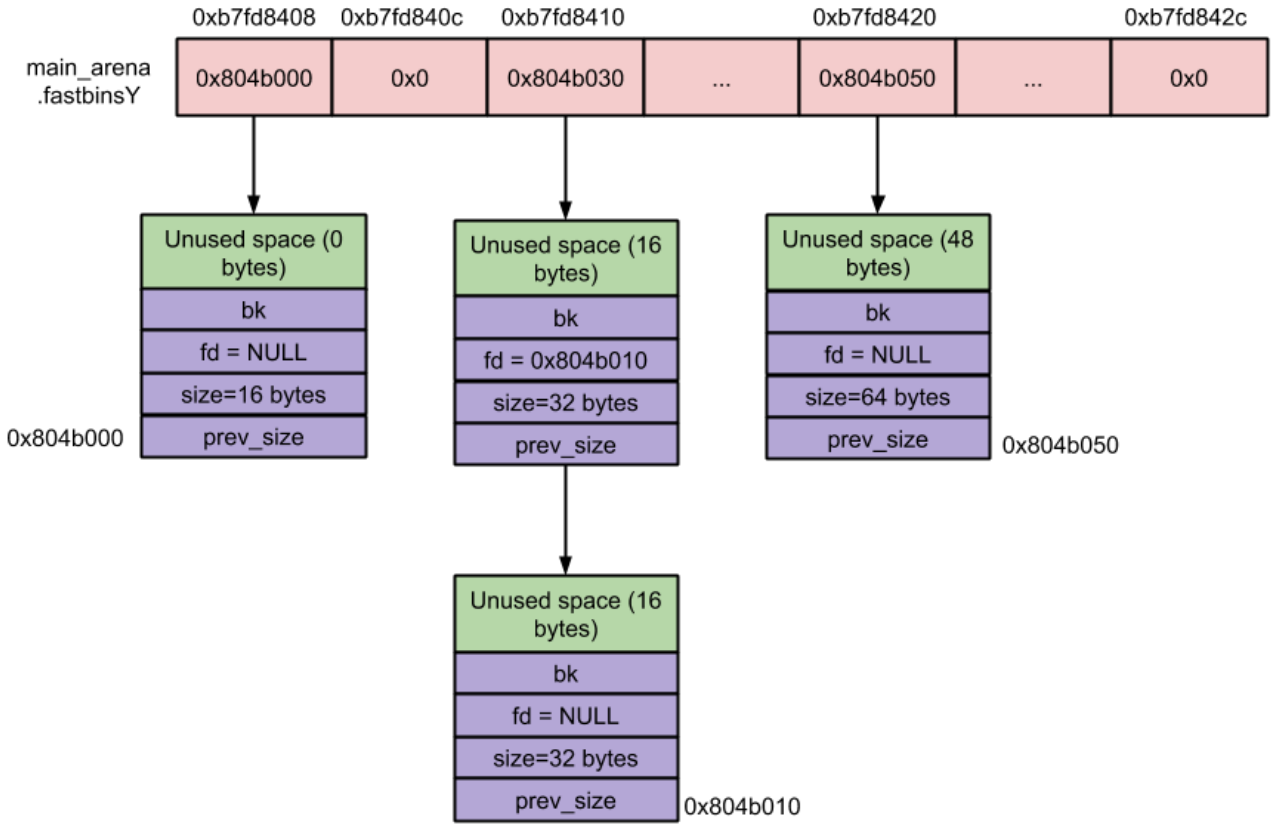
- Fast bin (быстрая корзина)
- Unsorted bin (несортированная корзина)
- Small bin (малая корзина)
- Large bin (большая корзина)

- Обычно корзина это LIFO структура: последним пришел – первым ушёл.

- Структуры, хранящие корзины:
- **fastbinsY**: Массив, хранящий быстрые корзины. **bins**: Массив хранит несортированные, малые и большие корзины. Всего 126 корзин:
- Bin 1 – Unsorted bin
- Bin 2 - Bin 63 – Small bin
- Bin 64 - Bin 126 – Large bin

Fast Bin

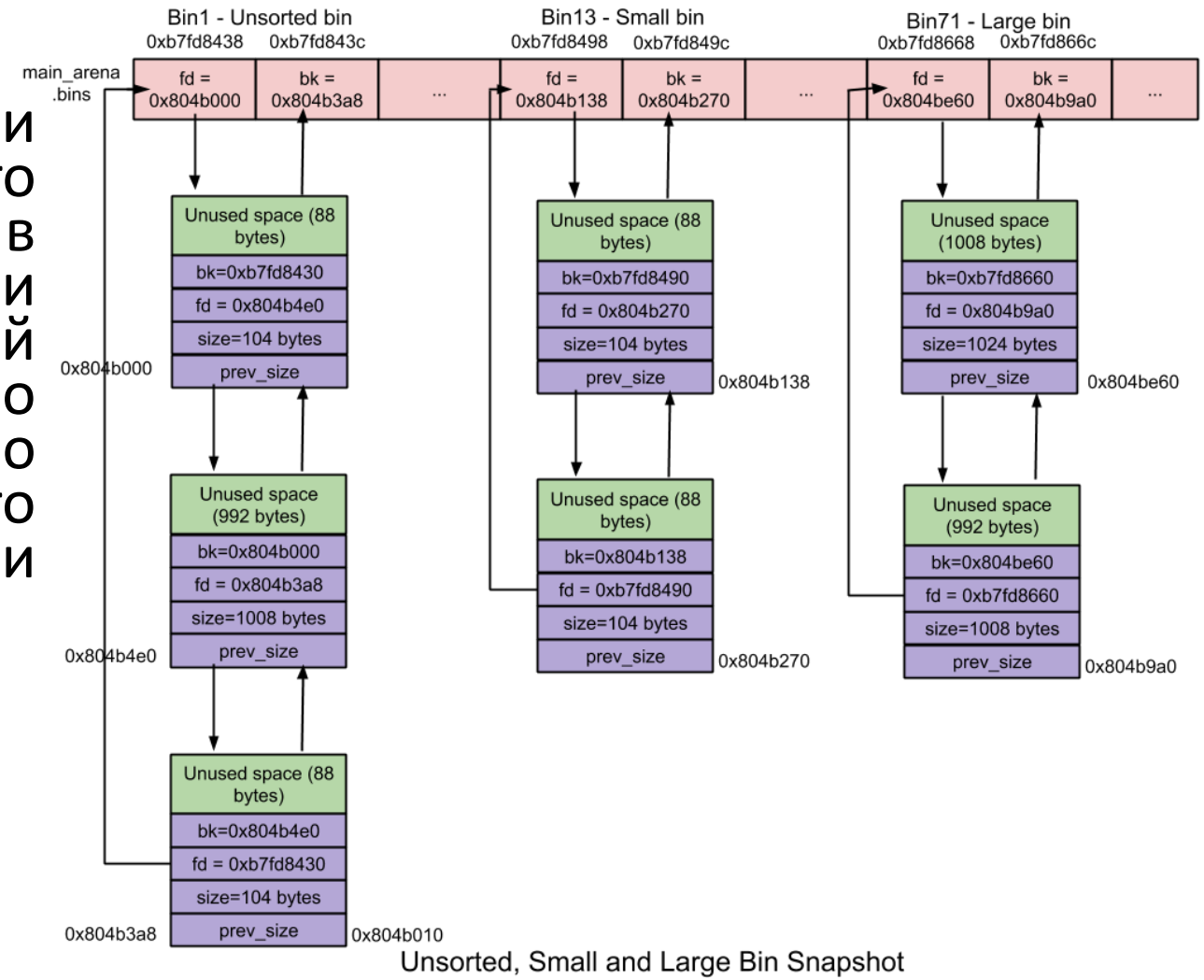
- Размер чанка от 16 до 80 байт. Исходя из названия – они быстро выполняют операции выделения и освобождения памяти.
- Число корзин – 10
  - Каждая корзина односвязный список свободных чанков. LIFO структура.
- Размер чанка отличается на 8 байт
  - Первая корзина хранит чанки по 16 байт, вторая по 24 и т.д.
  - В пределах одной корзины – чанки одного размера
- Соседние свободные чанки не объединяются в один.
- malloc(fast chunk) –
  - Проверка размера чанка (выполняются ли ограничения для fast chunk)
  - Память выделяется в первом чанке из списка fast bin
  - Чанк удаляется из списка fast bin
- free(fast chunk) –
  - Рассчитывается позиция освобождаемого чанка
  - Он добавляется в начало fast bin



Fast Bin Snapshot

Unsorted Bin

- При освобождении малых и больших чанков, вместо возвращения их в соответствующие коризны они попадают в unsorted bin. Такой подход позволяет повторно использовать недавно освобождённые чанки. Это немного ускоряет выделение и освобождение памяти.
- Количество корзин – 1
  - Двусвязный список чанков
- Ограничений по размеру чанков нет



Unsorted, Small and Large Bin Snapshot

Small Bin

- Размер чанка меньше 512 байт.
- Количество корзин – 62
  - Двусвязный список свободных чанков.
  - FIFO структура.
- Размер чанков кратен 8 байтам
  - 16 байт, 24 байта и т.д.
  - Chunks inside a small bin are of same sizes and hence it doesnt need to be sorted.
- Используется слияние соседних свободных чанков. Замедляет освобождение памяти, устраняет сегментацию.

- malloc(small chunk) –
  - Изначально подходящий чанк ищется в Unsorted bin
  - При первой попытке выделения памяти в small bin она инициализируется
  - Далее возвращается последний чанк из small bin
- free(small chunk) –
  - При освобождении, проверяется свободны ли соседние чанки, если да – происходит их слияние
  - Получившийся свободный чанк помещается в Unsorted bin

Large Bin

- Чанки больше 512 байт.
- Количество корзин – 63
  - Двусвязный список, чанки добавляются и удаляются из произвольных мест списка
  - 32 корзины содержат чанки с размером кратным 64 байтам.
  - 16 корзин содержат чанки с размером кратным 512
  - 8 корзин содержат чанки с размером кратным 4096
  - 4 корзины содержат чанки с размером кратным 32768
  - 2 корзин содержат чанки с размером кратным 262144
  - 1 корзина содержит чанк **оставшегося** размера
  - Могут хранить чанки разных размеров в пределах одной корзины. В списке чанки сортируются по размеру (по убыванию).
- Используется слияние соседних свободных чанков.

- malloc(large chunk) –
  - Изначально подходящий чанк ищется в Unsorted bin
  - При первой попытке выделения памяти в large bin она инициализируется
  - Далее возвращается последний чанк из large bin достаточного размера, если размер больше – чанк делится на две части, неиспользуемый чанк помещается в unsorted bin
  - Если подходящих чанков нет, производится сканирование памяти
  - Если подхощий чанк не найден на предыдущем этапе – используется top chunk
- free(large chunk) –
  - При освобождении, проверяется свободны ли соседние чанки, если да – происходит их слияние
  - Получившийся свободный чанк помещается в Unsorted bin

Стандартная схема эксплуатации

Нельзя просто так взять и нарисовать стандартную схему эксплуатации уязвимостей в куче

Protostar heap1

```
1  struct internet {
2    int priority;
3    char *name;   };
4  void winner() {
5    printf("and we have a winner @ %d\n", time(NULL));}
6  int main(int argc, char **argv) {
7    struct internet *i1, *i2, *i3;
8    i1 = malloc(sizeof(struct internet));
9    i1->priority = 1;
10   i1->name = malloc(8);
11   i2 = malloc(sizeof(struct internet));
12   i2->priority = 2;
13   i2->name = malloc(8);
14   strcpy(i1->name, argv[1]);
15   strcpy(i2->name, argv[2]);
16   printf("and that's a wrap folks!\n");
17 }
```

В куче образуется 4 объекта:

- internet1
- name1
- internet2
- name2

Первым вызовом **strcpy** можно переписать указатель на name2 в структуре internet2

Тогда второй вызов strcpy запишет данные по нужному нам адресу.

Demo time

Protostar heap2

```
1  struct auth {
2    char name[32];
3    int auth; };
4  struct auth *auth;
5  char *service;
6  int main(int argc, char **argv) {
7    char line[128];
8    while(1) {
9        printf("[ auth = %p, service = %p ]\n", auth, service);
10       if(fgets(line, sizeof(line), stdin) == NULL) break;
11       if(strncmp(line, "auth ", 5) == 0) {
12           auth = malloc(sizeof(auth));
13           memset(auth, 0, sizeof(auth));
14           if(strlen(line + 5) < 31)
15               strcpy(auth->name, line + 5);
16       }
17       if(strncmp(line, "reset", 5) == 0)
18           free(auth);
19       if(strncmp(line, "service", 6) == 0)
20           service = strdup(line + 7);
21       if(strncmp(line, "login", 5) == 0) {
22           if(auth->auth) {
23               printf("you have logged in already!\n");
24           } else {
25               printf("please enter your password\n");
26           }}}}
```

- Что задумал автор (use-after-free):
  - Создаём объект auth (36 байт)
  - Освобождаем его
  - Создаем объект в куче функцией **strdub** (будет указывать туда же, куда и auth)
  - Переписывает значение переменной int auth
  - Profit

Demo time

Protostar heap2

```
1  struct auth {
2    char name[32];
3    int auth; };
4  struct auth *auth;
5  char *service;
6  int main(int argc, char **argv) {
7    char line[128];
8    while(1) {
9        printf("[ auth = %p, service = %p ]\n", auth, service);
10       if(fgets(line, sizeof(line), stdin) == NULL) break;
11       if(strncmp(line, "auth ", 5) == 0) {
12           auth = malloc(sizeof(auth));
13           memset(auth, 0, sizeof(auth));
14           if(strlen(line + 5) < 31)
15               strcpy(auth->name, line + 5);
16       }
17       if(strncmp(line, "reset", 5) == 0)
18           free(auth);
19       if(strncmp(line, "service", 6) == 0)
20           service = strdup(line + 7);
21       if(strncmp(line, "login", 5) == 0) {
22           if(auth->auth) {
23               printf("you have logged in already!\n");
24           } else {
25               printf("please enter your password\n");
26           }}}}
```

- Компилятор решил так:
  - Создаём объект auth (4 байта)
  - Создаем объект в куче функцией **strdub** (будет указывать на середину auth)
  - Переписываем значение переменной int auth
  - Profit

Demo time

Protostar heap3

```
1  void winner() {
2    printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
3  }
4  int main(int argc, char **argv) {
5    char *a, *b, *c;
6    a = malloc(32);
7    b = malloc(32);
8    c = malloc(32);
9    strcpy(a, argv[1]);
10   strcpy(b, argv[2]);
11   strcpy(c, argv[3]);
12   free(c);
13   free(b);
14   free(a);
15   printf("dynamite failed?\n");
16 }
```

Полезные ссылки

- https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/
- https://github.com/shellphish/how2heap
- https://sensepost.com/blog/2017/painless-intro-to-the-linux-userland-heap/
- https://thesprawl.org/research/exploit-exercises-protostar-heap/

Вопросы?

Игорь Черватюк

Александр Трифанов

Андрей Басарыгин

Москва, 2018