**Лекция 9**

**Пример мапинга динамической библиотеки в память в рантайме**

Пусть функция динамической библиотеки выглядит так:

double myfunc(double x) {  
 return x \* x;  
}

Превратим ее в бинарный код: *g++ -c func.c*

Будем передавать ее в этот код:

#include <sys/mman.h>  
#include <stdlib.h>  
#include <sys/stat.h>  
#include <fcntl.h>  
#include <unistd.h>  
#include <stdio.h>  
  
int main(int argc, char\* argv[]) {  
 const char\* file\_name = argv[1];  
 double argument = strtod(argv[2], NULL);  
  
 int fd = open(file\_name, O\_RDONLY);  
 struct stat st = {};  
 fstat(fd, &st);  
  
 void\* addr = mmap(NULL, st.st\_size,  
 PROT\_READ | PROT\_EXEC, MAP\_PRIVATE, fd, 0);  
 double (\*func)(double) = (double (\*)(double))((char\*)addr + 0x40);  
  
 close(fd);  
 double result = func(argument);  
 printf("func(%f) = %f\n", argument, result);  
 munmap(addr, st.st\_size);  
}

*./a.out func.o 5* — на выходе получим *func(5.000000) = 25.000000*.

Таким образом загрузили бинарный код в память и исполнили его.

Конец heap, то есть число, на котором он заканчивается, называется **program break**. Можно попросить ОС отодвинуть эту верхнюю границу разрешенных адресов кучи, то есть изменить положение program break. Это можно сделать с помощью сискола **sbrk**:

*void\* sbrk(intptr\_t increment)*. При успешном выполнении вернет предыдущую границу program break.

**brk** — сискол, который возвращает эту доступную верхнюю границу

*int brk(void\* addr)*

**Рассмотрим изменение адресов вектора при его реаллокации:**

#include <iostream>  
#include <vector>  
  
int main() {  
 std::vector<char> v;  
 for (int i = 0; i < 1000000; ++i) {  
 if ((i & i - 1) == 0) {  
 std::cout << i << ' ' << (void\*) &v[0] << std::endl;  
 }  
 v.push\_back(i);  
 }  
}

Получим такую картину:

0 0

1 0x61aa755f62c0

2 0x61aa755f62e0

4 0x61aa755f62c0

8 0x61aa755f62e0

16 0x61aa755f62c0

32 0x61aa755f6300

64 0x61aa755f6330

128 0x61aa755f6380

256 0x61aa755f6410

512 0x61aa755f6520

1024 0x61aa755f6730

2048 0x61aa755f6b40

4096 0x61aa755f7350

8192 0x61aa755f8360

16384 0x61aa755fa370

32768 0x61aa755fe380

65536 0x61aa75606390

131072 0x774fc003d010

262144 0x774fbfcd8010

524288 0x774fbfc57010

То есть пока размер вектора маленький, при реаллокации просто отодвигается верхняя граница кучи, но когда размер вектора становится достаточно большим, размещение вектора происходит в другом месте с помощью mmap. Это можно пронаблюдать и с помощью strace: сначала будет вызываться brk, затем mmap.

Если же вместо vector будет list, то можно увидеть, что каждый раз будет вызываться только brk и будет отодвигаться граница кучи.

**setrlimit** — сискол, который может увеличить стек

*int setrlimit(int resource, const struct rlimit\* rlim)*

**Пример с увеличением размера стека:**

#include <iostream>  
#include <sys/resource.h>  
  
int main() {  
 const rlim\_t kStackSize = 210 \* 1024 \* 1024; // min stack size = 16 MB  
 struct rlimit rl;  
 int result;  
  
 result = getrlimit(RLIMIT\_STACK, &rl);  
 if (result != 0) {  
 std::cerr << "Failed to get current stack size\n";  
 abort();  
 }  
  
 std::cerr << rl.rlim\_cur << '\n';  
  
 if (rl.rlim\_cur < kStackSize) {  
 rl.rlim\_cur = kStackSize;  
 result = setrlimit(RLIMIT\_STACK, &rl);  
 if (result != 0) {  
 std::cerr << "Failed to get bigger stack size\n";  
 abort();  
 }  
 std::cerr << "Stack size is successfully set to " << kStackSize << '\n';  
 }  
}

Вывод:

8388608

Stack size is successfully set to 220200960

**Работа malloc**

Главная задача malloc — минимизировать количество вызовов sbrk и mmap.

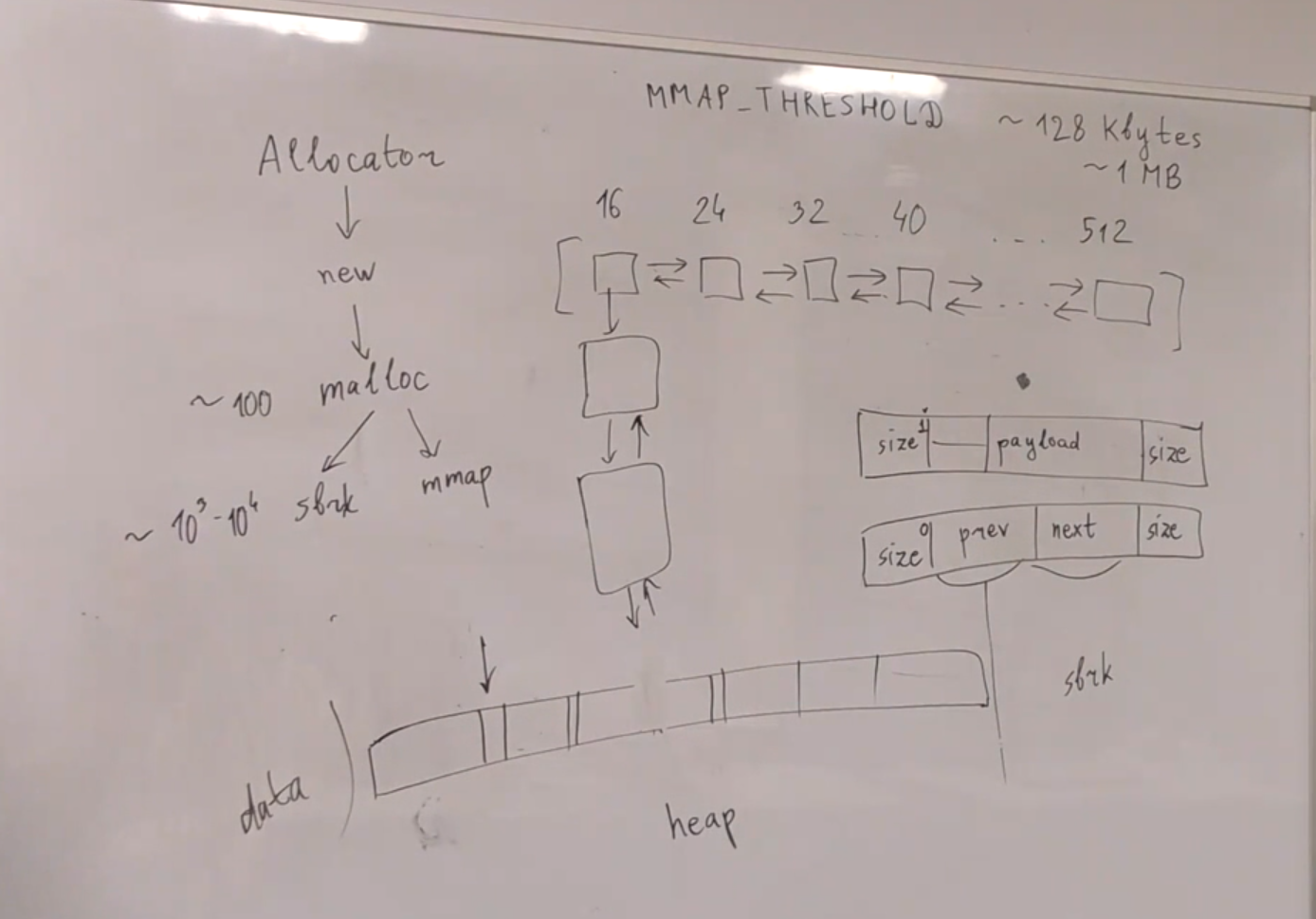
У маллока есть параметр **M\_MMAP\_THRESHOLD** — число в байтах, верхний порог, после которого он начнет вызывать mmap. По умолчанию равен 128 kB. Когда зовем mmap, в первых 16 байтах записывается размер выделенной памяти. Затем, когда нужно освободить память, если этот размер достаточно большой, вызываем munmap, иначе происходят хитрые вещи.

Мы хотим уметь поддерживать чанки разного размера. Чанки размеров 16, 24, 32, … , 512. И у нас есть массив из связных списков, в которых лежат свободные чанки. Когда нас просят выделить очередной кусок, смотрим, какого он размера, округляем до ближайшего большего числа, кратного 8, и берем чанк из соответствующего списка соответствующей ячейки массива, помечаем его занятым, вынимаем его из списка.

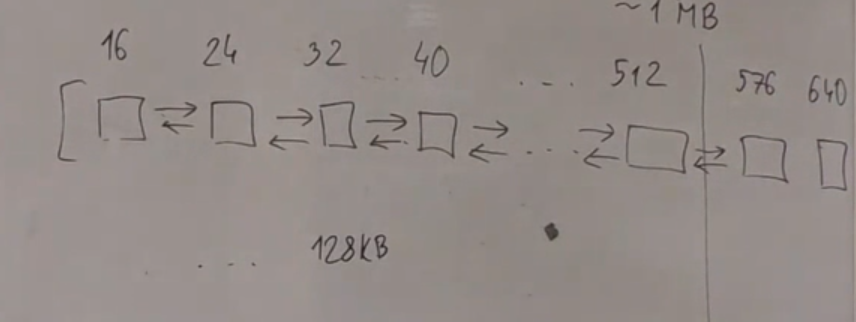
Изначально ситуация выглядит так: мы позвали sbrk на какое-то большое число, таким образом у нас есть в распоряжении много памяти в пределах от data до program break, откуда будем откусывать чанки. Когда попросили n байт, берем best fit, такого размера откусываем память от кучи и формируем чанк, и отодвигаем границу начала кучи. Когда эта граница зайдет слишком далеко, вызовем sbrk еще раз.

Как формируется чанк? В начале и в конце чанка пишется его размер, а также то, свободен ли он. Для того, чтобы хранить информацию о том, свободен кусок или занят, дополнительная память не нужна — так как размеры всех кусочков кратны 8, то 3 последних бита размера свободны, и в последнем бите можно хранить эту информацию. За счет этого, когда освобождается какой-то чанк, и предыдущий чанк тоже оказывается свободен, то мы можем их склеить, и размер получившегося будет суммой размеров тех кусочков.

Если кусочек свободен, то между размерами в нем хранятся не пользовательские данные, а указатели на предыдущий свободный чанк в списке и на следующий. Это нужно, чтобы когда мы занимаем этот кусочек, мы могли легко достать его из списка за О(1). Именно поэтому кусочек должен быть не меньше 16 байт: по 8 байт занимают указатели.



Если при слиянии кусочков получили кусочек размера больше, чем 512, то дальше размеры кусочков в списке увеличиваются экспоненциально: 576, 640 и так, пока не дорастут до размера 128 килобайт. После границы в 512, размеры чанков уже не строго фиксированы: они больше, чем 512 или предыдущего размера, но могут быть меньше следующего размера.



Если запрос больше 128 килобайт, зовем mmap.

Примечание: free умеет проверять много разных условий и сам выводит сообщения об ошибке в случае их нарушения.

https://azeria-labs.com/heap-exploitation-part-2-glibc-heap-free-bins/

<https://stackoverflow.com/questions/855763/is-malloc-thread-safe>