Программные уязвимости – на стыке аппаратуры и программного обеспечения

Лекция 3. Уязвимости переполнения кучи. Dlmalloc, организация heap, эксплуатация free().

Total recall

- Размещение объектов в памяти

```
#iclude <stdio.h>
int main()
{
        int a = 5; /*Размер задаём на этапе
компиляции*/
}
```

- Сегменты .bss, .data
- Динамическая память

Динамическая память

- Системные вызовы
  - brk()
  - mmap()
- Libc (dlmalloc - http://gee.cs.oswego.edu/dl/html/malloc.html)
  - Автор – Doug Lea

```
#include <stdio.h>
int main()
{
        int *a = malloc(sizeof(int));  /*Размер задаём во время выполнения*/
}
```

  - malloc() испольуует brk(), затем разбивает полученный блок памяти на более мелкие сегменты

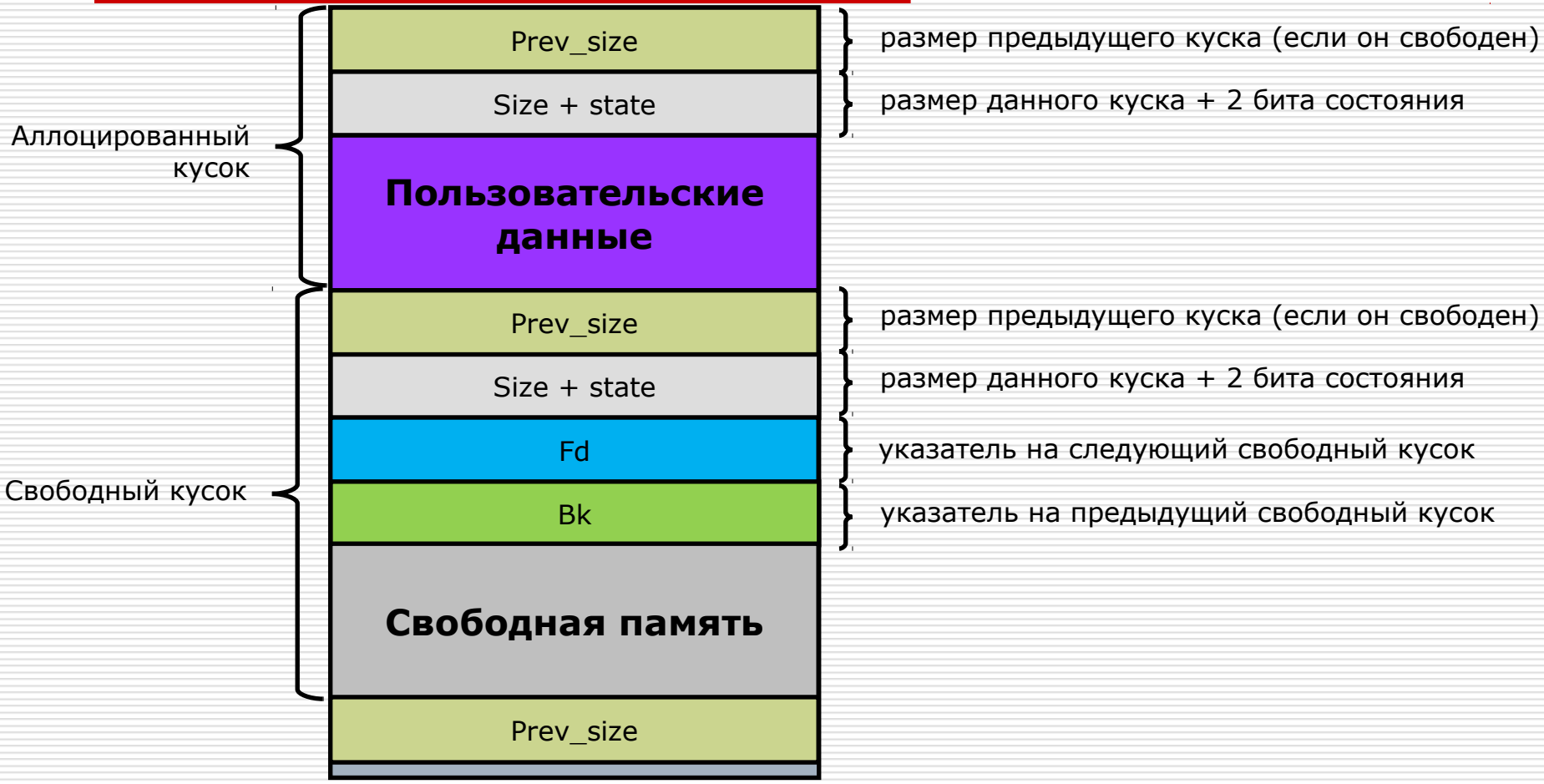
Основные принципы dlmalloc

- **Совместимость, переносимость**
- **Производительность**
  - Аллокатор должен минимизировать накладные расходы, а также фрагментацию памяти.
  - Вызовы malloc(), free() и realloc должны быть настолько быстрыми для среднего случая, насколько это возможно
- **Возможность настройки**
- **Локальность**
  - Аллоцированные куски должны по-возможности располагаться близко к друг другу в виртуальной (и физической) памяти
- **Обнаружение ошибок**
  - Возможность обнаружения ошибок обычно конфликтует с требованиями производительности. Тем не менее, базовая функциональность по обнаружению должна присутствовать.
- **Минимизация аномалий**
  - Аллокатор с настройками по-умолчанию должен быть пригоден для широкого класса приложений – от графических оконных систем до компиляторов и сетевых приложений.

Переполнение стека vs переполнение кучи

- Стек
  - Сохраняет и восстанавливает EIP в явном виде
  - Изменяя EIP, мы напрямую управляем потоком исполнения программы
- Heap
  - Предназначен только для хранения данных
  - Как же это эксплуатировать?

Dlmalloc – основы работы с динамической памятью в libc

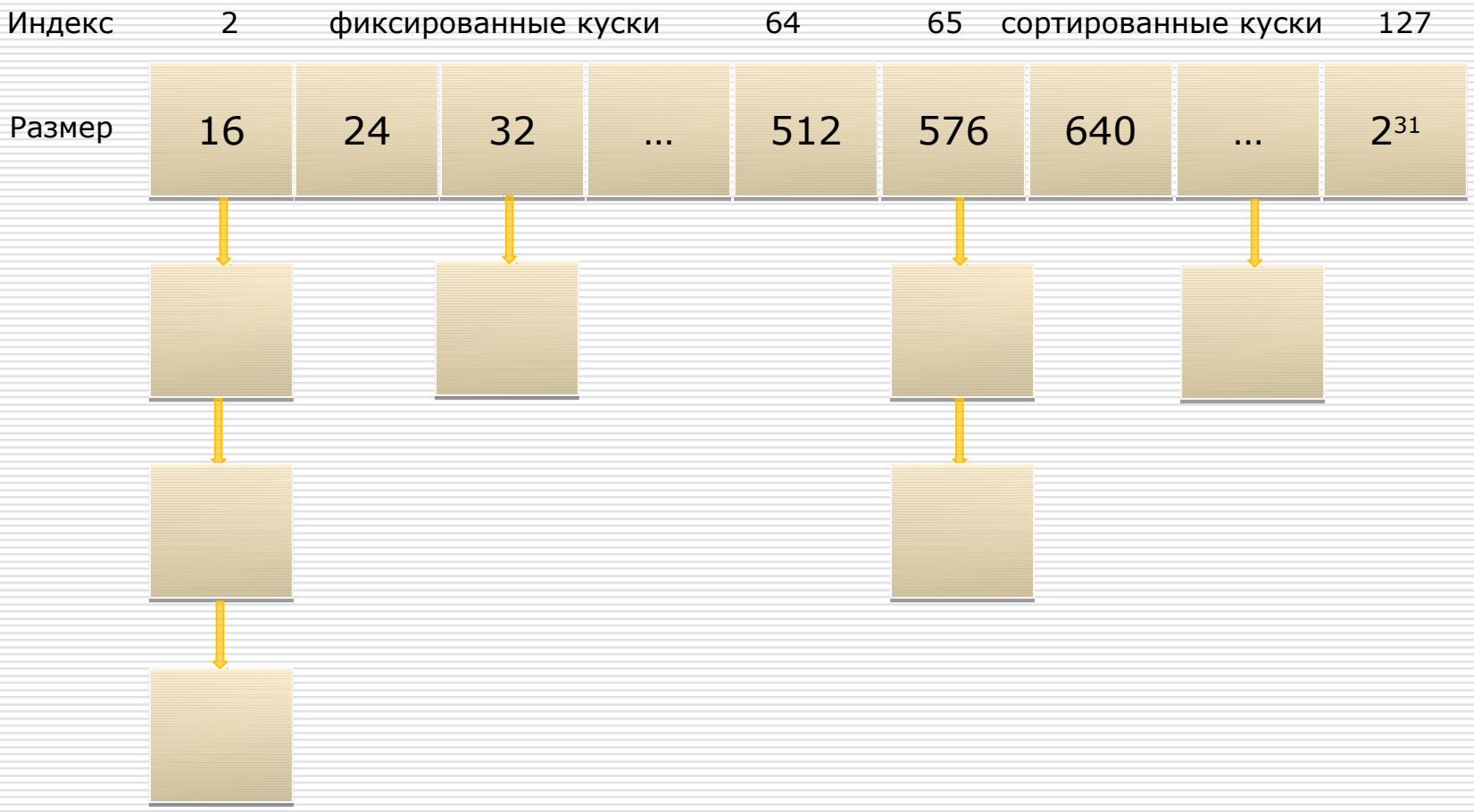


Правила работы с кусками памяти

- Есть два ключевых правила работы с кусками памяти:
  - Никакие два свободных куска не могут являться соседями
  - На границе аллоцированной dlmalloc и свободной системной памяти находится специальный “wilderness” кусок

Размещение кусков в памяти

Организация кусков в «корзины»



Макросы для работы с кусками - Unlink()

Чтобы удалить свободный кусок из соответствующей корзины:

```
#define unlink( P, BK, FD ) {            \
    BK = P->bk;                          \
    FD = P->fd;                          \
    FD->bk = BK;                         \
    BK->fd = FD;                         \
}
```

Макросы для работы с кусками - FrontLink()

Чтобы добавить освободившийся кусок в корзину:

```
#define frontlink( A, P, S, IDX, BK, FD ) {                  \
    if ( S < MAX_SMALLBIN_SIZE ) {                           \
        IDX = smallbin_index( S );                           \
        mark_binblock( A, IDX );                             \
        BK = bin_at( A, IDX );                               \
        FD = BK->fd;                                         \
        P->bk = BK;                                          \
        P->fd = FD;                                          \
        FD->bk = BK->fd = P;                                 \
    } else {                                                 \
        IDX = bin_index( S );                                \
        BK = bin_at( A, IDX );                               \
        FD = BK->fd;                                         \
        if ( FD == BK ) {                                    \
            mark_binblock(A, IDX);                           \
        } else {                                             \
            while ( FD != BK && S < chunksize(FD) ) {        \
                FD = FD->fd;                                 \
            }                                                \
            BK = FD->bk;                                     \
        }                                                    \
        P->bk = BK;                                          \
        P->fd = FD;                                          \
        FD->bk = BK->fd = P;                                 \
    }                                                        \
}
```

Malloc() – основные шаги

Шаг 1: проверяем сначала корзины с фиксированными кусками, доступен ли кусок подходящего размера. Если да, отделяем его от корзины с помощью Unlink():

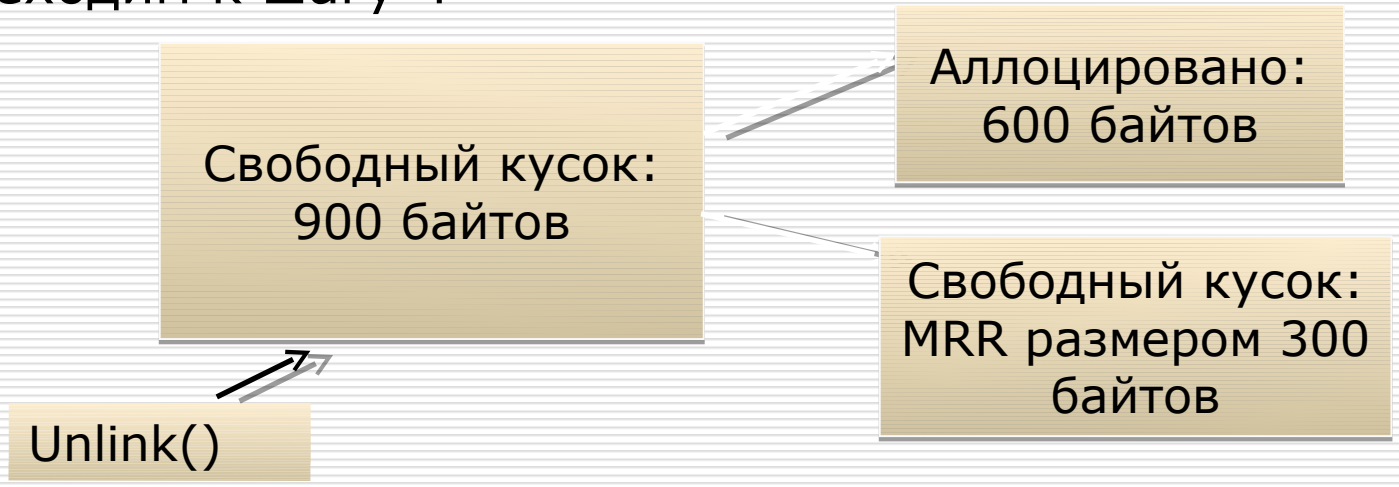
Malloc() – основные шаги

Шаг 2: используем MRR (“most recently remaindered”) кусок, если он достаточно большой. Если позволяет размер, кусок делится на два. Первый аллоцируем, второй становится новым MRR. Если нет, кладём его в корзину и переходим на шаг 3.

Malloc() – основные шаги

Шаг 3: ищем в следующих корзинах в порядке возрастания размера. Если найден достаточно большой кусок, удаляем его из корзины с помощью Unlink() и делим на два куска. Первый аллоцируем, второй становится MRR.

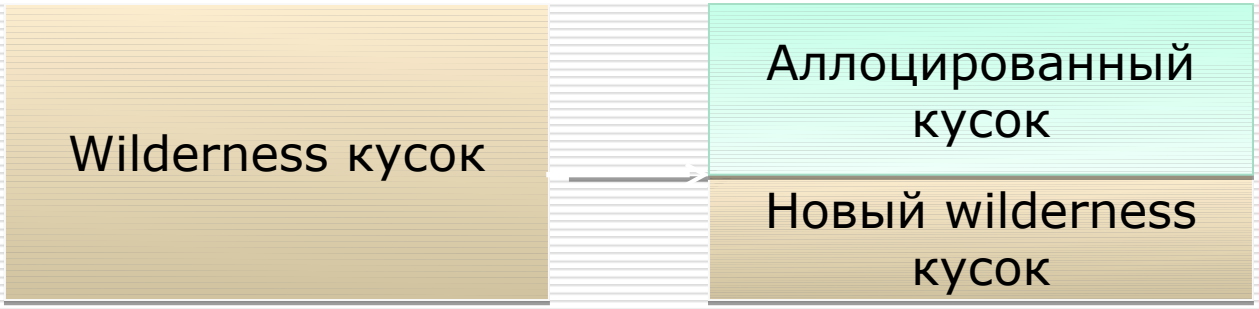
Если нигде в корзинах нет достаточно большого куска, переходим к шагу 4



Malloc() – основные шаги

Шаг 4: проверяем wilderness-кусок, если он достаточно большой, делим его на два. Аллоцируем младшую часть, старшая становится новым wilderness-куском.

Если этот кусок недостаточно велик, переходим к шагу 5



Malloc() – основные шаги

Шаг 5: Просим помощи у операционной системы. Используем sbrk() и mmap() для расширения доступной памяти.

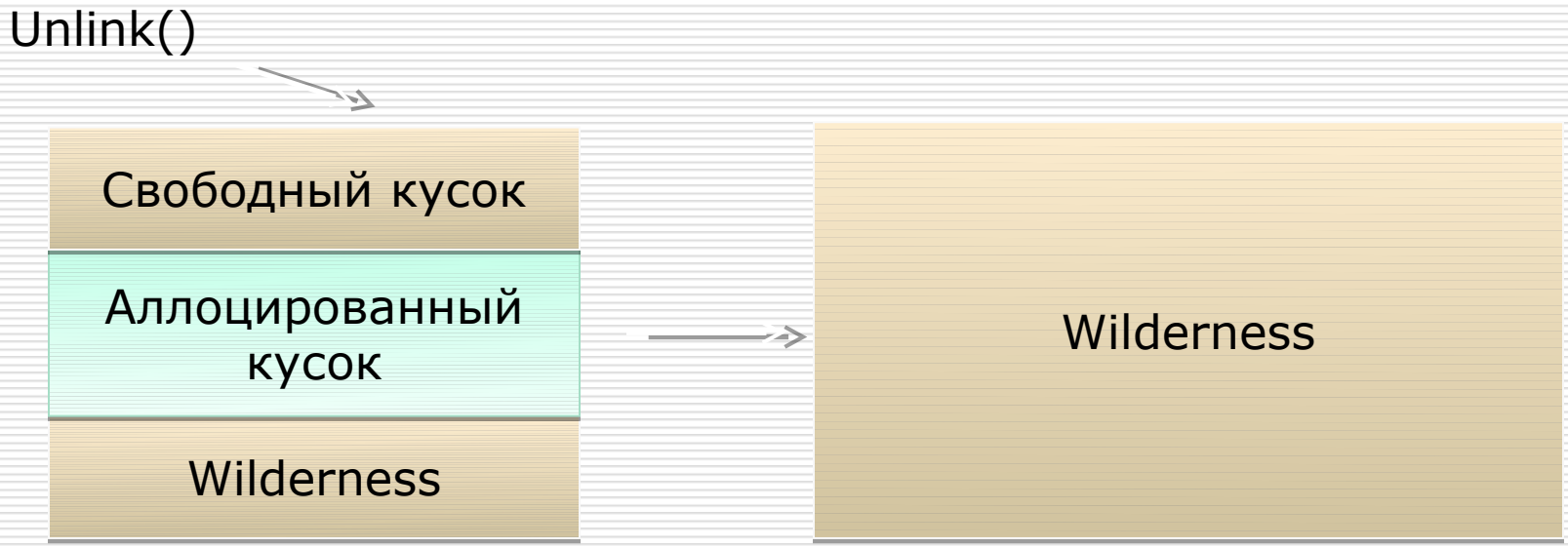
Если и это не удаётся, сдаёмся.

Free() – работает обратно malloc()

Освобождает аллоцированный кусок, добавляя его в корзину и объединяя с соседними свободными кусками в корзине.

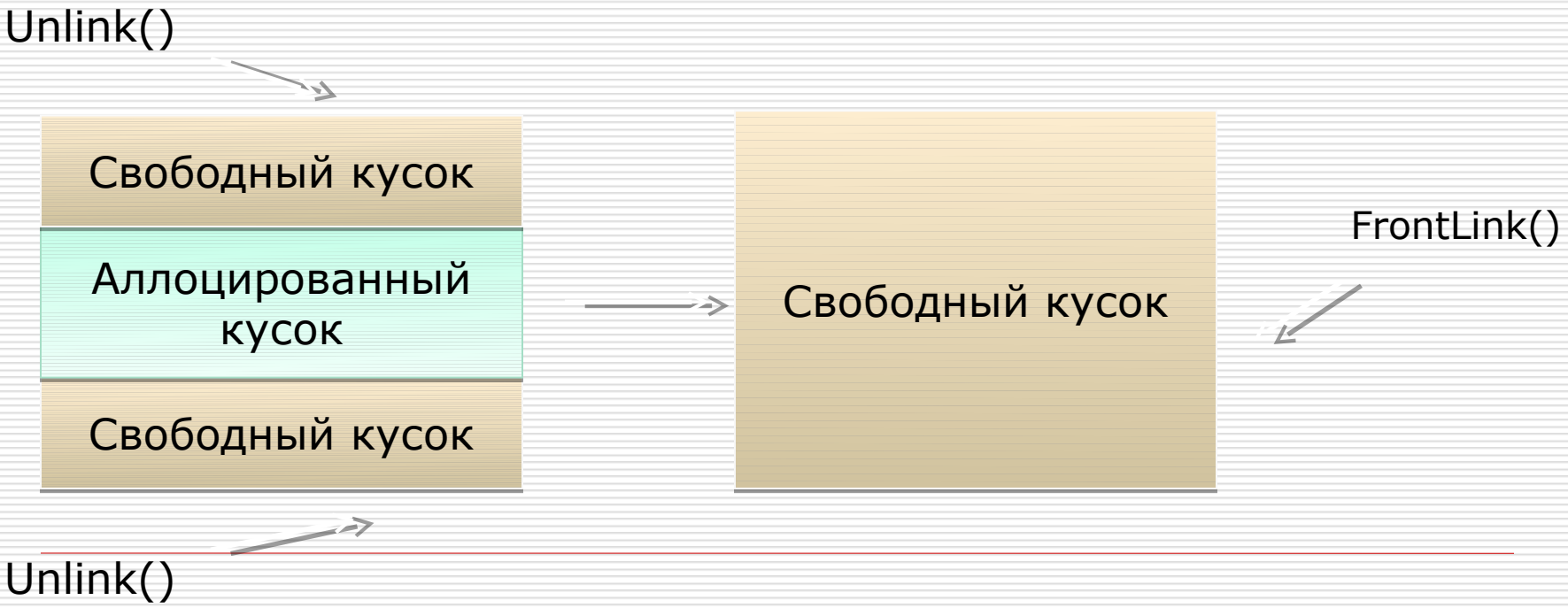
Free() – шаг 1

Если кусок граничит с wilderness куском, сливаем их в больший wilderness кусок. Кроме того, сливаем с предыдущим свободным куском, если он есть.

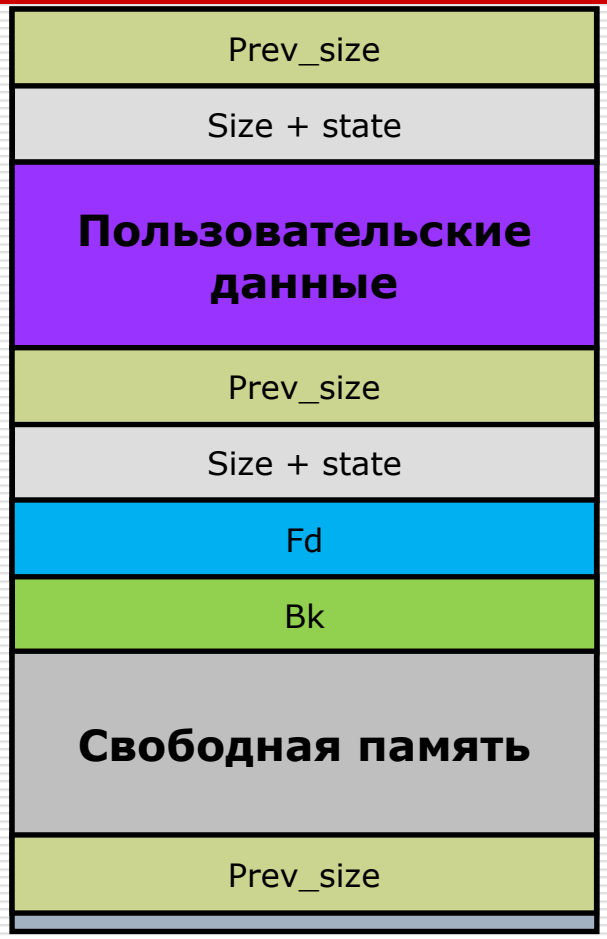


Free() – шаг 2

Сливаем свободный кусок с его свободными соседями.
Удаляем каждый кусок с помощью unlink(), потом сливаем.
В конце используем frontlink() чтобы разместить новый большой кусок.



Вспомним структуру кусков



Поле prev_size: это первое 4-байтовое поле в куске.
Если предыдущий кусок свободен, в этом поле хранится размер этого свободного куска.

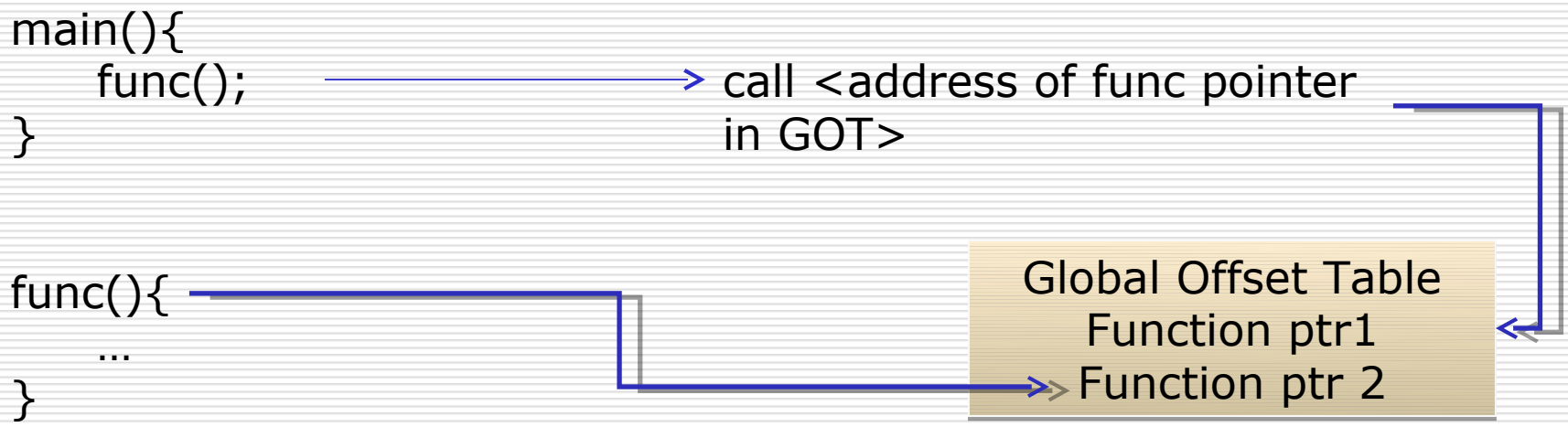
Если предыдущий кусок аллоцирован, то это поле перезаписывается пользовательскими данными предыдущего куска (и это учитывается в malloc())

В аллоцированном куске не хранится информация о том, аллоцирован он или нет. Но в поле размера следующего за ним куска есть два флага, один из которых - PREV_INUSE, который говорит, свободен предыдущий кусок или аллоцирован.

Вызов функций - GOT

Тело программы не содержит прямые адреса функций.

Вместо этого существует специальная таблица **Global Offset Table**, которая хранит указатели на функции. Вызов функции сначала обратится к этой таблице, получит указатель на функцию оттуда, после чего перейдёт по указателю



Переполнение кучи - Unlink

Эксплуатация unlink():

Вспомним, как выглядит unlink()

```
(1) #define unlink( P, BK, FD ) {            \
(2)     BK = P->bk;                          \
(3)     FD = P->fd;                          \
(4)     FD->bk = BK;                         \
(5)     BK->fd = FD;                         \
(6)}
```

Как видно, с помощью строки 4 мы можем изменить значение произвольного слова в памяти.

Идея эксплуатации

Мы можем изменить поле в таблице GOT и положить туда указатель на внедрённый нами код. Соответственно, когда будет вызвана эта функция, то будет выполнен внедрённый нами код.

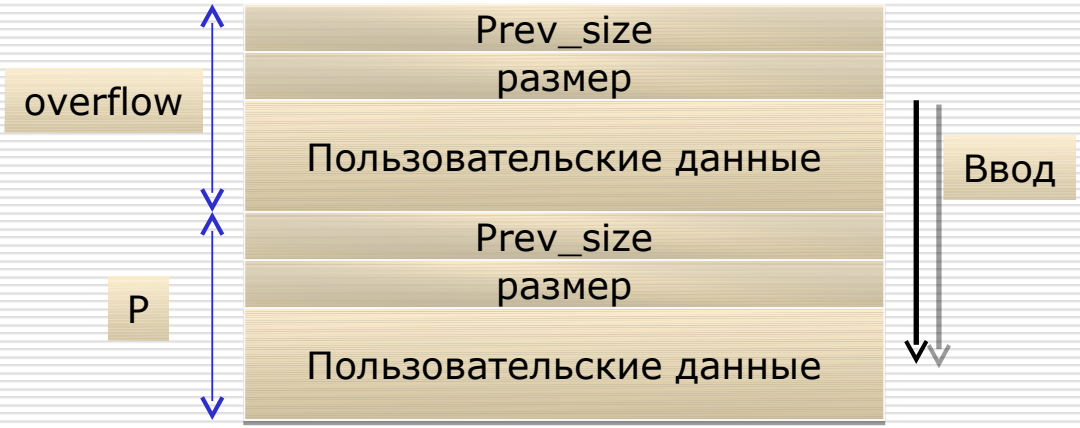
Сделаем так, чтобы FD указывал на GOT, а BK на наш шеллкод.

Пример

```
(1) #include <stdio.h>
(2)
(3) int main(void)
(4) {
(5)    char* overflow = malloc(768);
(6)    char* p = malloc(64);
(7)    scanf("%s",overflow);
(8)    free(overflow);
(9)    free(p);
(10)  return 0;
(11)}
```

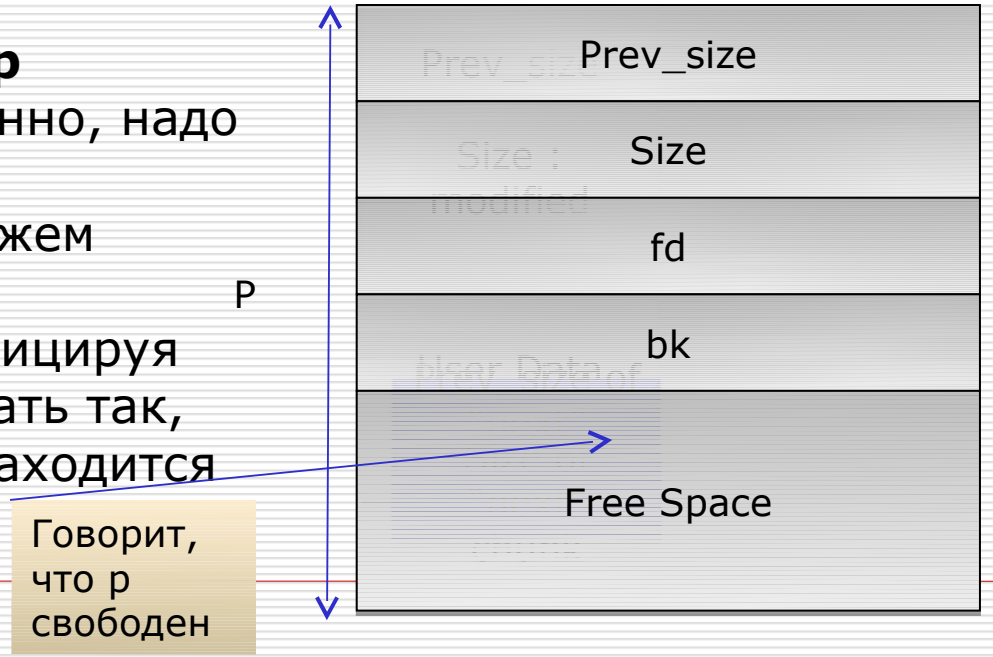
Когда вызывается malloc(768), кусок wilderness разбивается на два. P также будет выделен из куска wilderness.

```
(1) #include <stdio.h>
(2)
(3) int main(void)
(4) {
(5)     char* overflow = malloc(768);
(6)     char* p = malloc(64);
(7)     scanf("%s",overflow);
(8)     free(overflow);
(9)     free(p);
(10)    return 0;
(11)}
```



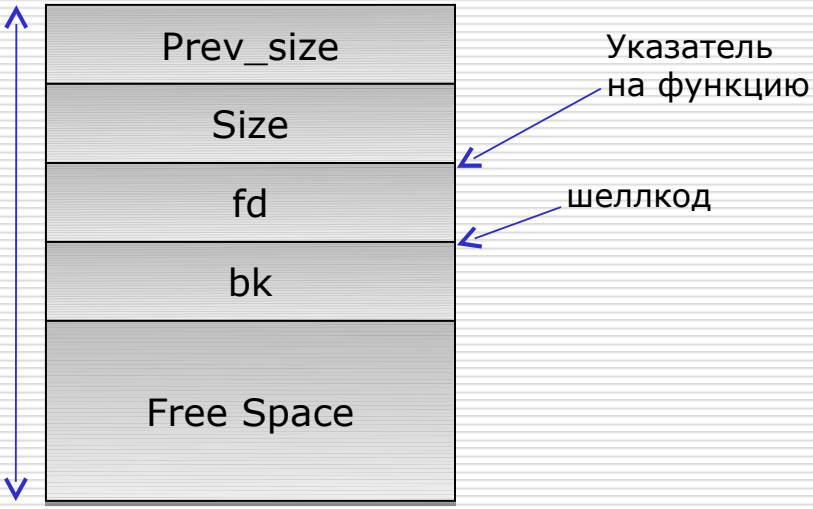
```
(1) #include <stdio.h>
(2)
(3) int main(void)
(4) {
(5)     char* overflow = malloc(768);
(6)     char* p = malloc(64);
(7)     scanf("%s",overflow);
(8)     free(overflow);
(9)     free(p);
(10)    return 0;
(11)}
```

Теперь нам надо сделать **p** “свободным”. Соответственно, надо модифицировать размер следующего куска. Мы можем изменить расположение следующего куска, модифицируя размер **p**. Мы можем сделать так, что «следующий» кусок находится внутри **p**.



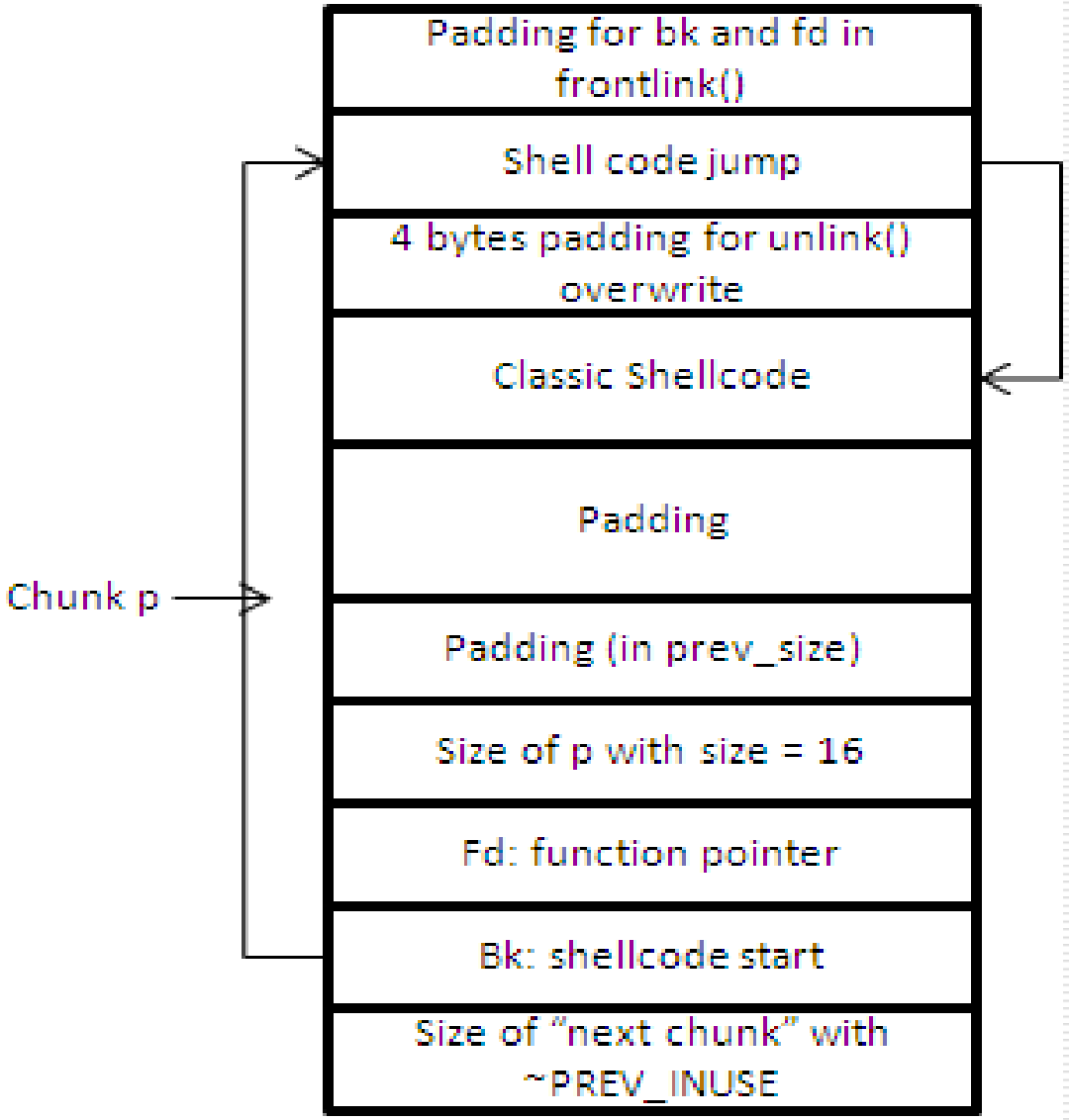
```
(1) #include <stdio.h>
(2)
(3) int main(void)
(4) {
(5)    char* overflow = malloc(768);
(6)    char* p = malloc(64);
(7)    scanf("%s",overflow);
(8)    free(overflow);
(9)    free(p);
(10)   return 0;
(11)}
```

Когда будет вызван free(overflow), т.к. p «свободен», процедура попробует слить его с overflow. Будет вызван unlink для p. Соответственно в сам overflow нужно поместить нужные значения fd и bk.



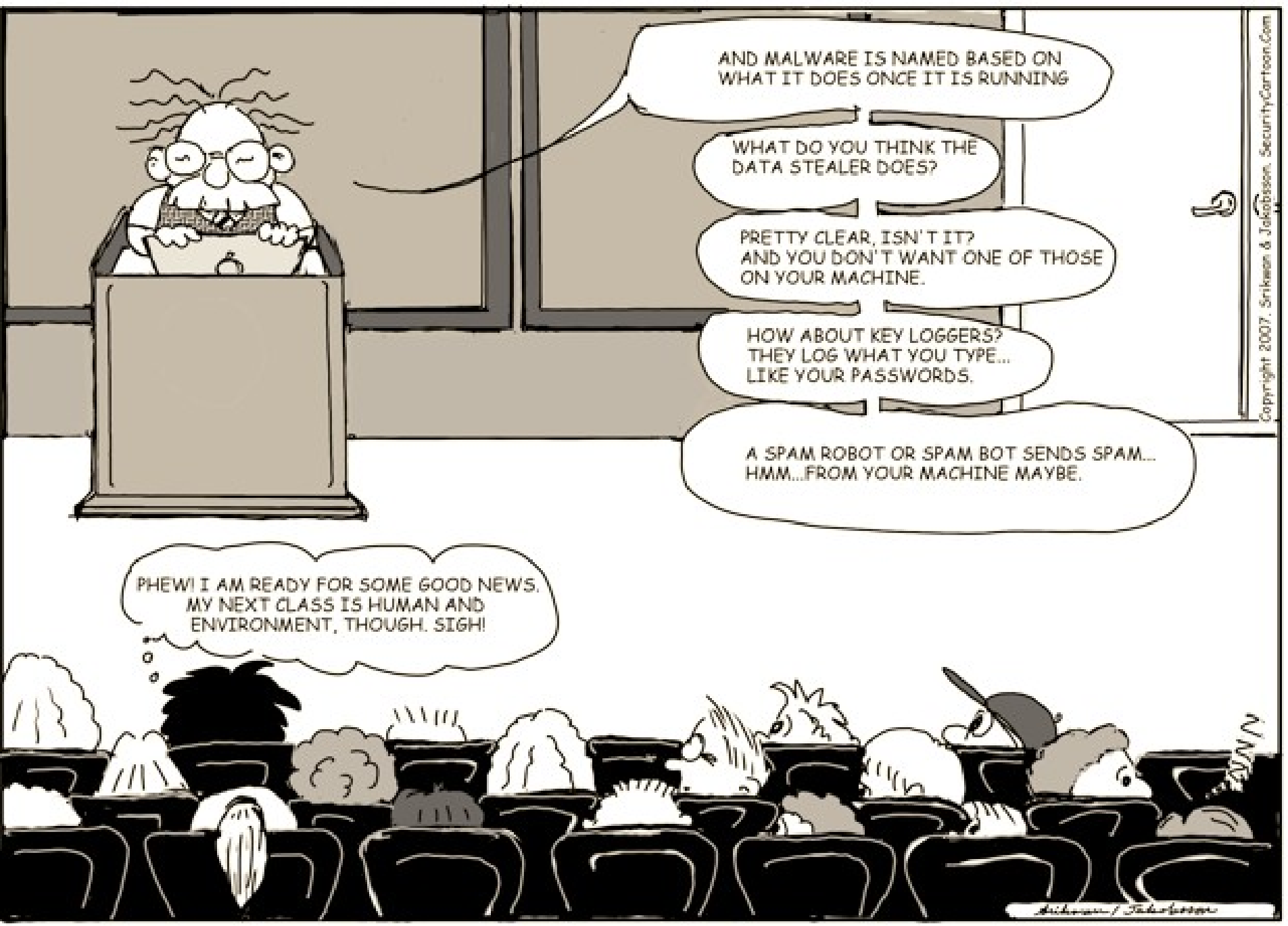
Пример

Данные для переполнения overflow в результате будут выглядеть так: (подменять можно вызов free(), вызов exit())



Дальнейшее чтение

1. Эксплуатация FrontLink(), malloc() – Shellcoder’s Handbook, chapter 5, pp. 89-107.
2. Vudo malloc tricks. Kaempf, Michel "MaXX". 2007, Phrack 57, p. 0x08. http://www.phrack.com/issues.html?issue=57&id=8&mode=txt
3. Heap Oveflow Tutorial. http://www.phiral.net/blackhatbloc/phrack/heaptut.txt


AND MALWARE IS NAMED BASED ON WHAT IT DOES ONCE IT IS RUNNING
WHAT DO YOU THINK THE DATA STEALER DOES?
PRETTY CLEAR, ISN'T IT? AND YOU DON'T WANT ONE OF THOSE ON YOUR MACHINE.
HOW ABOUT KEY LOGGERS? THEY LOG WHAT YOU TYPE... LIKE YOUR PASSWORDS.
A SPAM ROBOT OR SPAM BOT SENDS SPAM... HMM...FROM YOUR MACHINE MAYBE.
PHEW! I AM READY FOR SOME GOOD NEWS. MY NEXT CLASS IS HUMAN AND ENVIRONMENT, THOUGH. SIGH!
Copyright 2007. Srikwan & Jakobsson. SecurityCartoon.com