

# Основы построения защищенных компьютерных сетей

## Лекция 9 Переполнения буфера

Семён Новосёлов

2024



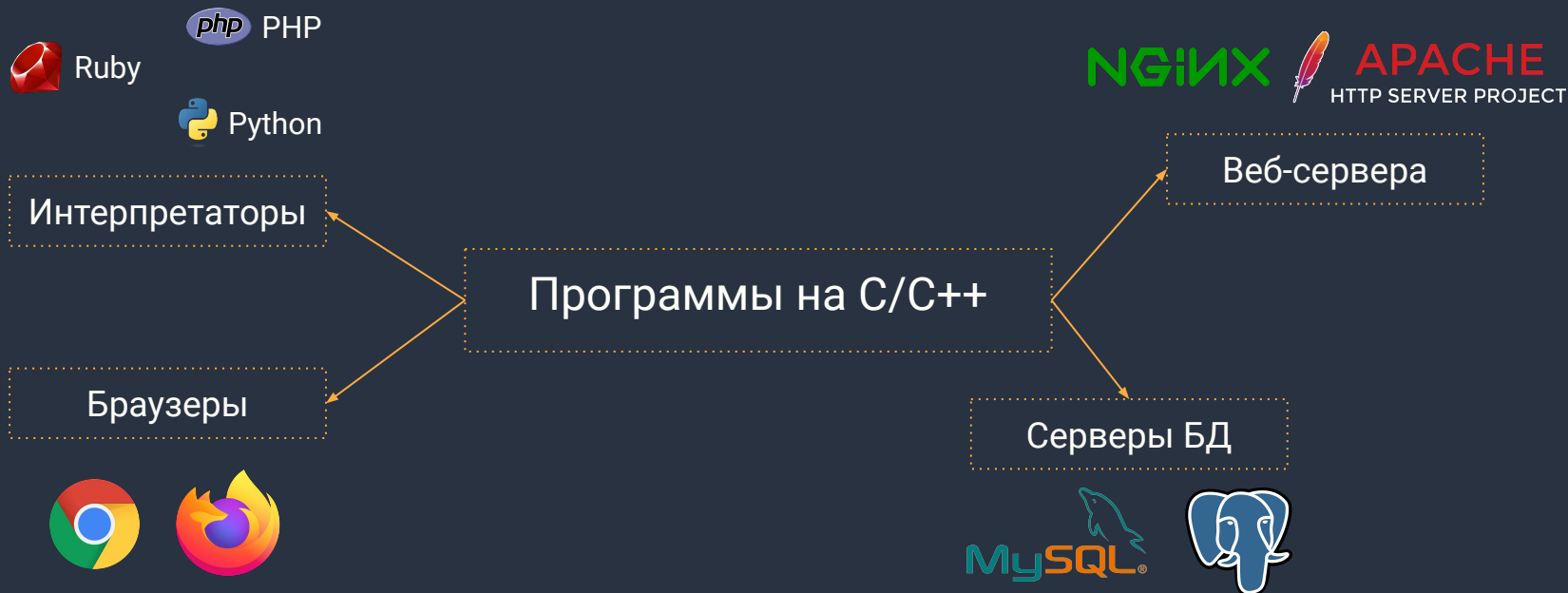
**БФУ**  
ИМЕНИ И. КАНТА

# Переполнения буфера

- **Переполнение стека**
- Переполнение кучи



# В каких программах может встречаться?



**В общем случае:** любые программы на языках с прямым управлением памятью

# Переполнение буфера. Вводный пример

В примере производится запись за пределы буфера.

В отсутствии защиты, получается бесконечный цикл. **Почему?**

```
int main(){
    char buf[100];
    int i;
    for(i=0; i<=108; i++){
        buf[i] = 'A';
    }
    return 0;
}
```

# Переполнение буфера. Вводный пример

В примере производится запись за пределы буфера.

```
int main(){
    char buf[100];
    int i;
    for(i=0; i<=108; i++){
        buf[i] = 'A';
    }
    return 0;
}
```

В отсутствии защиты, получается бесконечный цикл. **Почему?**

**Ответ:** бесконечно перезаписывается переменная `i`, лежащая в памяти после буфера

# Переполнение стека. Пример

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

void hello(char* str){
    char buf[128];
    strcpy(buf, str);
    printf("Hello, %s\n", buf);
}

int main(int argc, char* argv[]){
    if(argc < 2){
        printf("Enter argument!\n");
        return 1;
    }
    hello(argv[1]);
    return 0;
}
```

Код hello.c

- программа принимает аргумент извне и записывает в массив **buf**
- нет проверки выхода за границы массива
- локальные переменные хранятся в стеке
- перезаписывается память в стеке за массивом

# Чем опасно?

В памяти хранятся адреса, по которым переходит программа в процессе работы.

## Примеры:

- адреса возврата при вызове функций
- указатель на функцию-callback

Перезапись позволяет **выполнить код по любому адресу**.

# Как выполняются программы?

Упрощенно:

1. ОС загружает код из исполняемого файла в память
2. выделяет память под стек и другие данные
3. передает управление на точку входа (Entry Point)



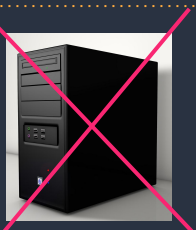
# Распределение памяти для hello.c

Start Address	End Address	Permissions	Name
0x0000561d049f5000	0x0000561d049f8000	r-x	/media/sf_vbox-shared/bof/hello
0x0000561d049f8000	0x0000561d049fa000	rwX	/media/sf_vbox-shared/bof/hello
0x00007f771e575000	0x00007f771e5a1000	r-x	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f771e5a2000	0x00007f771e5a4000	rwX	/usr/lib/x86_64-linux-gnu/ld-2.31.so
0x00007f771e5a4000	0x00007f771e5a5000	rwX	
0x00007fffed5ad000	0x00007fffed5ce000	rwX	[stack]
0x00007fffed5da000	0x00007fffed5dd000	r--	[vvar]
0x00007fffed5dd000	0x00007fffed5de000	r-x	[vdso]
0xffffffffffff600000	0xffffffffffff601000	--x	[vsyscall]

Edb: Memory Regions (Linux, x64)

# Машинный код

Процессор



**Двоичный код программы:** последовательность кодов команд (опкодов) процессора

**Ассемблер:** текстовая запись для удобства

48 31 c0 48 31 ed 50 48 bd 63 2f 73 68

Память



Система команд зависит от архитектуры:

- **Intel x64/x32:** Intel® 64 and IA-32 Architectures Software Developer's Manual
- **Arm:** Arm® A64 Instruction Set Architecture

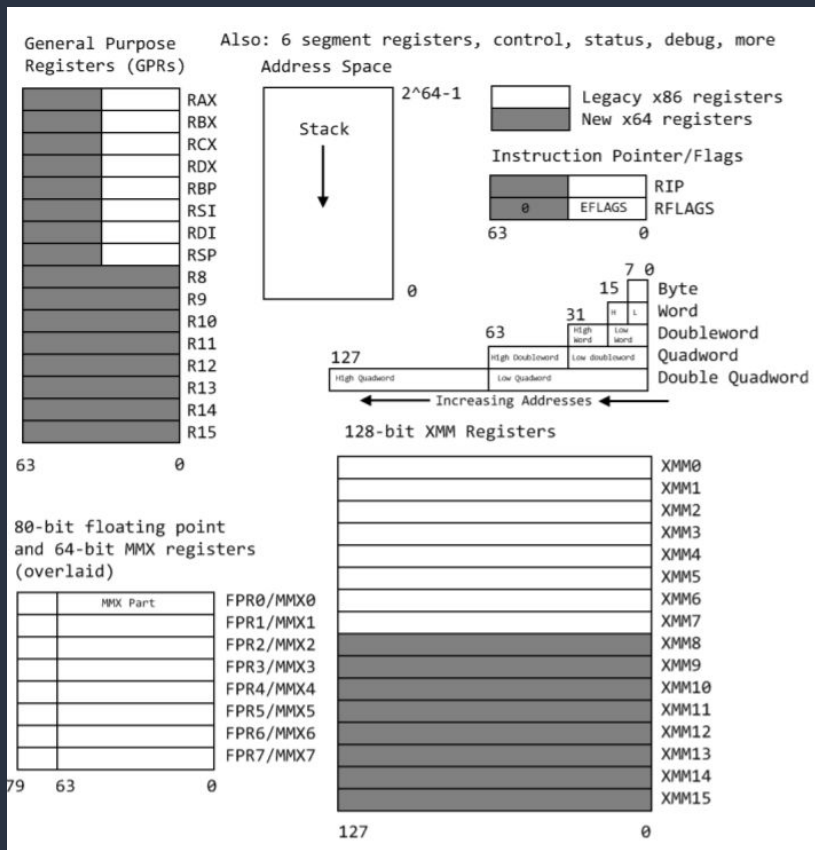
# Пример

000055d6:ac1551d3	f3	db 0xf3	
000055d6:ac1551d4	0f	db 0x0f	
000055d6:ac1551d5	1e	db 0x1e	
000055d6:ac1551d6	fa	cli	
000055d6:ac1551d7	55	push rbp	
000055d6:ac1551d8	48 89 e5	mov rbp, rsp	
000055d6:ac1551d9	48 83 ec 10	sub rsp, 0x10	
000055d6:ac1551da	89 7d fc	mov [rbp-4], edi	
000055d6:ac1551db	48 89 75 f0	mov [rbp-0x10], rsi	
000055d6:ac1551dc	83 7d fc 01	cmp dword [rbp-4], 1	
000055d6:ac1551dd	7f 13	jg 0x55d6ac1551fc	
000055d6:ac1551de	48 8d 3d 1f 0e 00 00	lea rdi, [rel 0x55d6ac15600f]	ASCII "Enter argument!"
000055d6:ac1551df	e8 8b fe ff ff	call 0x55d6ac155080	
000055d6:ac1551e0	b8 01 00 00 00	mov eax, 1	
000055d6:ac1551e1	eb 18	jmp 0x55d6ac155214	
000055d6:ac1551e2	48 8b 45 f0	mov rax, [rbp-0x10]	
000055d6:ac155200	48 83 c0 08	add rax, 8	
000055d6:ac155201	48 8b 00	mov rax, [rax]	
000055d6:ac155202	48 89 c7	mov rdi, rax	
000055d6:ac155203	e8 7a ff ff ff	call hello!hello	
000055d6:ac155204	b8 00 00 00 00	mov eax, 0	
000055d6:ac155210	c9	leave	
000055d6:ac155211	c3	ret	

```
int main(int argc, char* argv[]){
    if(argc < 2){
        printf("Enter argument!\n");
        return 1;
    }
    hello(argv[1]);
    return 0;
}
```

машинный код функции `main`

# Архитектура IA-64



**Регистры:** встроенная память в процессоре

- процессор сначала загружает данные в регистры, а затем обрабатывает
- используются для передачи параметров в функции

**RIP:** указатель на текущую выполняемую инструкцию в программе

**RSP:** указатель на вершину стека

# Что такое стек?

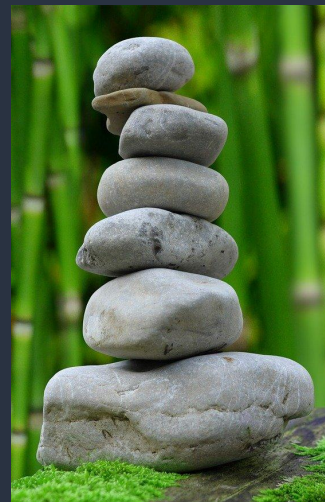
**Стек** – зарезервированная область памяти для работы с динамическими данными.

**Принцип LIFO:** последним пришёл – первым ушёл.

Растет в сторону меньших адресов.

Основные действия со стеком:

- **push:** помещение элемента в стек
- **pop:** извлечение элемента из стека
- при выполнении команд процессор увеличивает/уменьшает регистр **RSP**



# Инструкции call и ret

**call**: вызов функции

- помещает в стек по адресу в регистре **rsp** (то есть в вершину стека), текущее значение регистра **rip** (счетчика команд) — **адрес возврата**
- переходит по адресу, указанному в качестве операнда

**ret**: возврат из функции

- производит обратные действия к **call**
- извлекает из стека значение по адресу в регистре **rsp** и переходит по нему

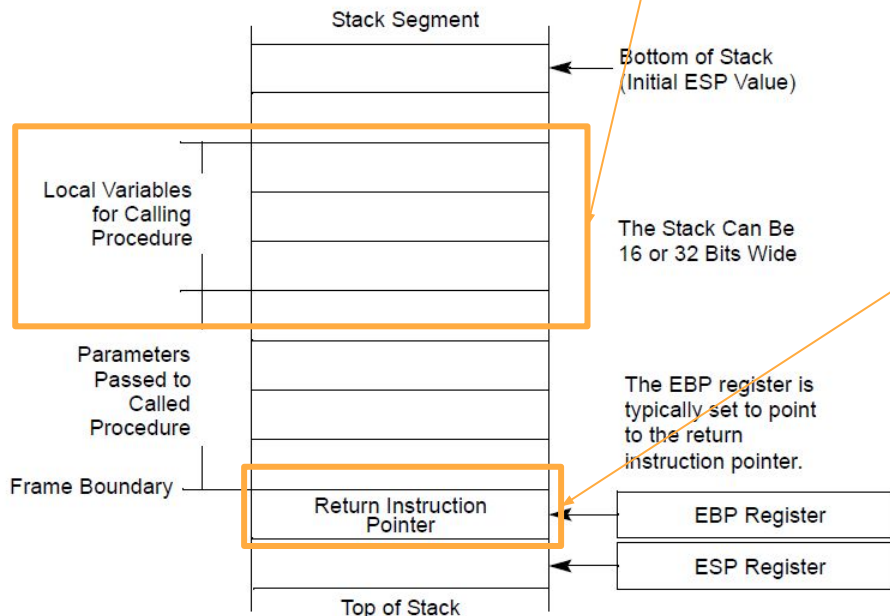
# Структура стека

здесь хранится буфер buf из hello() в hello.c

помещается в стек при вызове hello() с помощью call

указатель на main(), перезаписывается внешними данными при переполнении буфера buf в hello.c

Можно подставить свой адрес и при вызове **ret** программа перейдет по нему



↓ Pushes Move the Top Of Stack to Lower Addresses

↑ Pops Move the Top Of Stack to Higher Addresses

# Эксплуатация

Предполагаем в начале, что нет защитных механизмов.

1. записать рабочий машинный код (шелкод) в буфер
  - база: <https://www.exploit-db.com/shellcodes>
  - Metasploit: модули `payloads` для генерации
2. перезаписать адрес возврата, указав на адрес начала кода в буфере
3. адреса можно узнать с помощью отладчика



# Инструменты отладки



**GDB:** консольный отладчик, анализ core dump

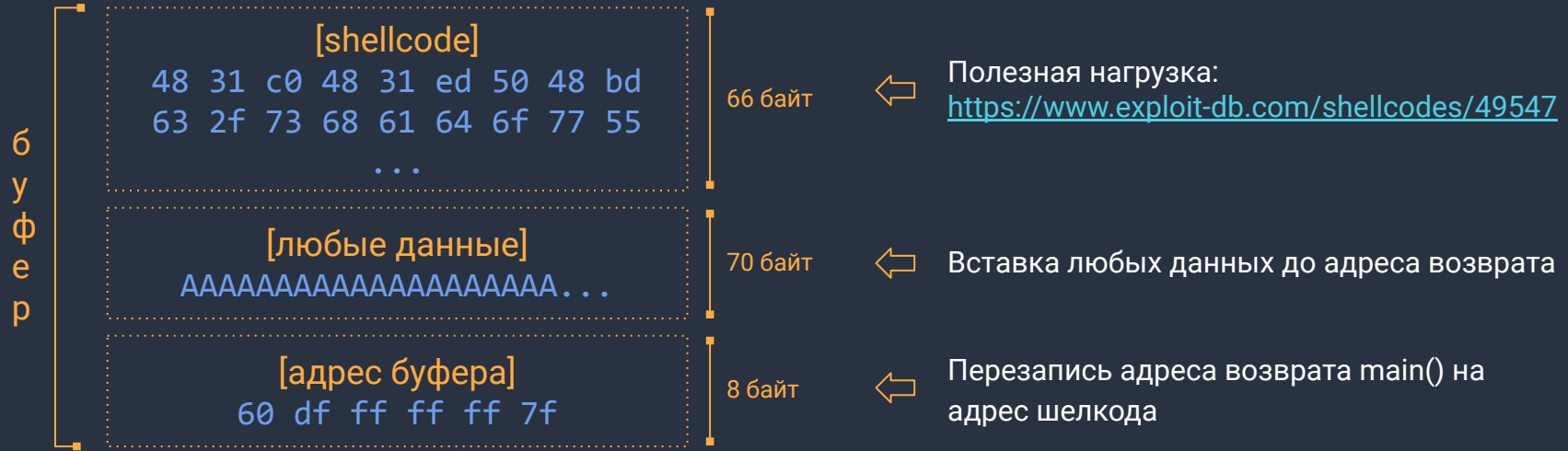
```
edb - /home/eteran/edb-debugger/build/hello [3313]
File View Debug Plugins Options Help
5662:e546 83 cc 8c      sub esp, 0xc
5662:e549 8d 83 28 e6 ff ff  push eax
5662:e54f 59             pop eax
5662:e550 8b 3b fe ff ff    mov ebx, [ebp-4]
5662:e555 83 c4 10         add esp, 0x10
5662:e558 83 45 e4 01 01  add dword [ebp-0xc], 1
5662:e55c 8b 45 e4         mov eax, [ebp-0xc]
5662:e55f 3b 06           cmp eax, [esi]
5662:e561 7c c3           jle 0x5662e54e
5662:e563 8b 00 00 00 00   mov eax, 0
5662:e568 8d 65 f4       lea esp, [ebp-0xc]
5662:e56b 59             pop ecx
5662:e56c 5b             pop ebx
5662:e56d 5e             pop esi
5662:e56e 5d             pop ebp
5662:e56f 8d 61 fc       lea esp, [ecx-4]
5662:e572 c3             jee
5662:e573 66 90         nop
5662:e575 66 90         nop
5662:e577 66 90         nop
5662:e579 66 90         nop
5662:e57b 66 90         nop
0x5662e57d <-->
0x5662e3b0 = 0x000000005662e3b0 <-hello!puts@plt+0>
puts(<0x000000005662e600> "Hello, world!")
Data Dump
0x000000005662e000-0x000000005662f000
5662:e000 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00  ELF.....
5662:e010 03 00 01 00 00 00 00 00 00 00 00 00 00 00 00  4.....
5662:e020 04 17 00 00 00 00 00 00 00 00 00 00 28 00  4.....
5662:e030 1d 00 1c 00 00 00 00 00 34 00 00 00 34 00  00.....
5662:e040 24 00 00 00 28 01 00 00 28 01 00 00 04 00  00.....
5662:e050 04 00 00 03 00 00 00 54 01 00 00 54 01 00  00.....
5662:e060 51 01 00 00 13 00 00 00 13 00 00 00 04 00  00.....
5662:e070 01 00 00 01 00 00 00 00 00 00 00 00 00 00  00.....
5662:e080 00 00 00 34 07 00 00 34 07 00 00 05 00 00  4.....
5662:e090 10 10 00 01 00 00 00 00 00 00 00 00 00 00  00.....
5662:e0a0 05 1c 00 00 38 01 00 00 34 01 00 00 0c 00  00.....
5662:e0b0 00 10 00 00 02 00 00 00 e8 0c 00 00 e1 00  00.....
Registers
EAX 5662e600 ASCII "Hello, world!"
ECX ff9daa10
EDX ff9daa34
EBX 5662f10b
ESP ff9da9c0
EBP ff9da978
ESI ff9daa10
EDI 00000000
EIP 5662e550 <-hello!main+51>
C 0 ES 002b (0000000000000000)
0 CS 0023 (0000000000000000)
A 1 SS 002b (0000000000000000)
Z 0 DS 002b (0000000000000000)
I 1 FI 0000 NULL
0 0 CS 0063 (000000007f5000c0)
0 0
EFL 00000292 (NO,OE,NE,A,S,NP,L,IE)
ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
Bookmarks Registers
Stack
ff9da9c0 5662e600 vbv ASCII "Hello,
ff9da9c4 f7720000 c
ff9da9c8 00000000
ff9da9cc 5662e535 5vbv Return to 0x0
ff9da9d0 f772e37c
ff9da9d4 5662f10b TDV
ff9da9d8 ff9daaac ar.
ff9da9dc 00000000
ff9da9e0 00000001
ff9da9e4 ff9daaac ar.
ff9da9e8 ff9daaac ar.
ff9da9ec ff9daa10 r.
ff9da9f0 00000000
ff9da9f4 f7720000 c
ff9da9f8 00000000
Stack Debugger Error Console
paused
```

```
Breakpoint 1, main () at test.c:8
8      int a = 42;
(gdb) bt
#0 main () at test.c:8
(gdb) info frame
Stack level 0, frame at 0x7fffffff850:
rip = 0x40060d in main (test.c:8); saved rip 0x7ffff7a7ead
source language c.
Arglist at 0x7fffffff840, args:
Locals at 0x7fffffff840, Previous frame's sp is 0x7fffffff850
Saved registers:
rbp at 0x7fffffff840, rip at 0x7fffffff848
(gdb) l
3      # include "../headers/liblist_single.h"
4
5      int main (void)
6      {
7          void* ce;
8          int a = 42;
9          int b = 1337;

```

**EDB:** графический отладчик удобно использовать для пошаговой отладки

# Пример. Вывод /etc/passwd через hello.c



Вход программы (./hello "[buf]"):

`[buf] = [shellcode][A x (buf_size - shellcode_size - 8)][buff_addr]`

`buf_size` подбирается, чтобы перезаписывать адрес возврата на main()

# Как узнать адрес буфера?

Адрес плавает в зависимости от среды выполнения (переменных окружения и т.п.)

При первоначальной отладке:

- посмотреть в отладчике:  
передать на вход тестовую строку (`AAAAAA...`) и найти её адрес в памяти

При запуске без отладчика:

- посмотреть в `coredump`
- оставить тот же самый адрес, добавив NOP-дорожку

# NOP-дорожка

- для снижения точности указания адреса можно поставить в начало шелкода команды **NOP** с кодом `\x90`.
- при переходе на такую команду процессор просто переходит к следующей
- пример:  
`[NOP x 100][shellcode][...]`  
можно сделать ошибку в адресе на **100**

## Важно:

- стек постоянно меняется при выполнении
- шелкод не должен перезаписываться

# Защита от переполнения буфера

**Переполнение буфера** — архитектурная проблема, связанная с хранением кода и данных в одном адресном пространстве.

Невозможно эффективно предотвратить, но можно ограничить последствия, сделав атаку нерабочей:

- DEP/NX-бит
- ASLR
- Stack-Canary

# Защита: Запрет выполнения стека

- Делает стек не выполняемым
- На аппаратном уровне реализована с помощью атрибута страницы памяти (**NX/XD – бита**)
- На программном уровне поддерживается начиная с ядра Linux версии 2.3.23 и Windows XP SP1 (**DEP**)

# Атака: Возврат в библиотеку (Return-to-libc attack)

Так как стек стал неисполняемым, то передавать управление на него смысла нет.

Всё ещё можно:

- передать управление на любой исполняемый участок кода программы
- вызвать одну из функций любой подключенной библиотеки

**Проблема:** при этом необходимо поместить аргументы этой функции в буфер или в регистры.

# Атака: ROP (Return oriented programming)

Для атаки используются части кода, которые заканчиваются инструкцией `ret` (“гаджеты”).

- Формируется цепочка из гаджетов и данных для них, которая выполняет нужный код.

**Пример:** нужно записать в регистр `eax` определённое значение.

- Находим в машинном коде: `pop eax; ret`
- Адрес возврата перезаписываем на адрес `pop eax; ret`
- В начало буфера:  
нужное значение + адрес следующего гаджета



# Защита: ASLR (Address space layout randomization)

- Расположение важных структур (стека, кучи и библиотек) по случайным адресам.
- Так как адреса становятся случайными, то нельзя построить цепочку гаджетов или выбрать адрес для возврата в библиотеку.

# ASLR. Обход

- В некоторых системах технология реализована не полностью.
- Перебор адресов.
- Использование уязвимостей, позволяющих читать содержимое памяти.
- В многопоточных приложениях (сервера) адреса не меняются при создании/удалении потоков

# Защита: Stack Canary

Вставляет определенное значение после буфера и проверяет его перед выходом из функции.

Есть несколько видов защиты:

- Terminator canaries
- Random canaries
- Random XOR canaries

gcc использует комбинированный подход:

- 0x00 + 3 случайных байта

# Обход Stack canary

- В некоторых случаях возможен перебор: многопоточные сервера и приложения
- Использование уязвимостей, позволяющих читать память в стеке

# Защита: Control-Flow Enforcement Technology (CET)

- работает на уровне процессора
- вводит дополнительный “теневого стек”
- при вызове **call** в него сохраняется состояние вызывающей процедуры
- при вызове **ret** производится проверка, защищающая от перезаписи адреса возврата

Процессоры: Intel Tiger Lake

# Литература и ссылки

- Д. Фостер, В. Лю. Разработка средств безопасности и эксплойтов. 2011
- База шелкодов:  
<https://www.exploit-db.com/shellcodes>

