Основы построения защищенных компьютерных сетей

Лекция 9
Переполнения буфера

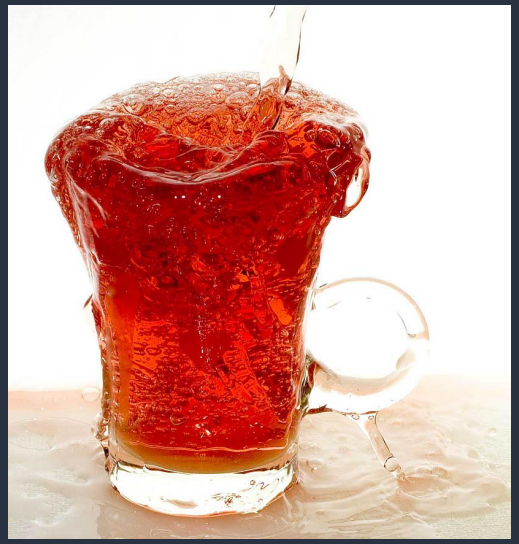
Семён Новосёлов

2024

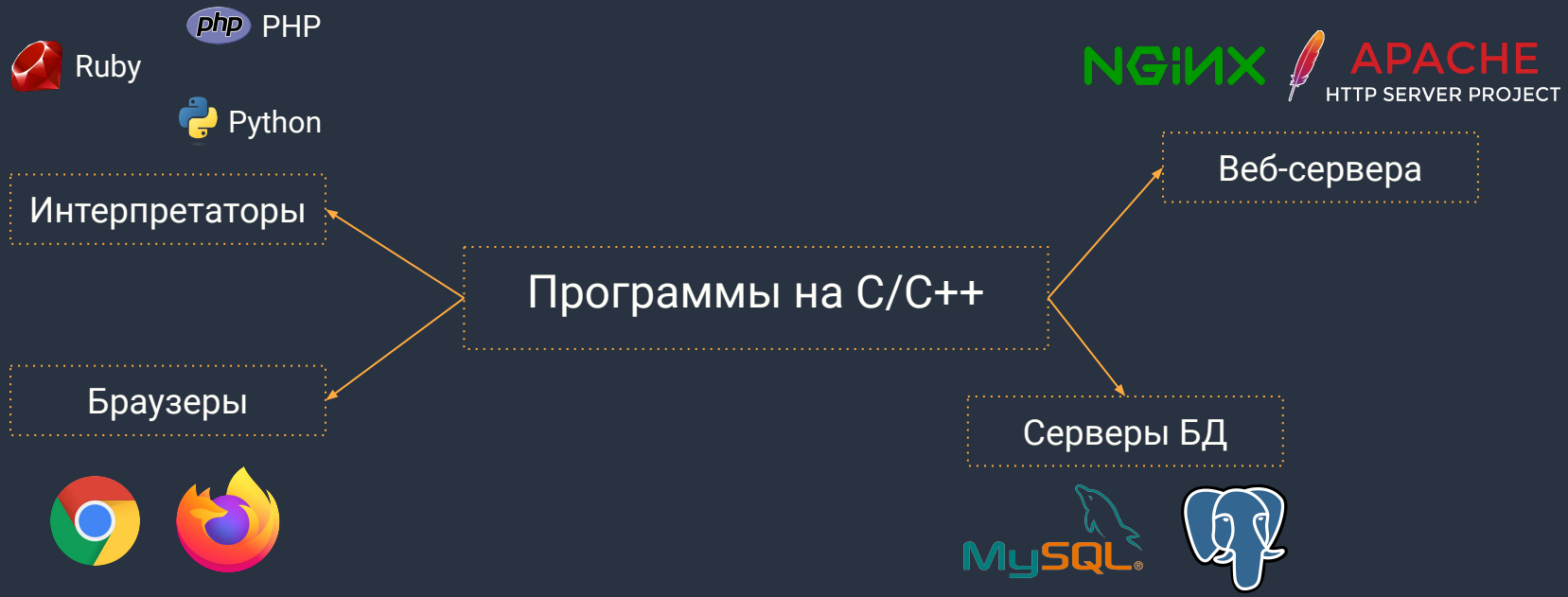
БФУ
ИМЕНИ И. КАНТА

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.

```
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?**

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

```c
#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:

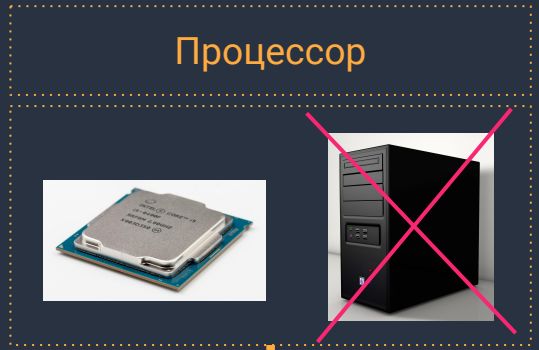
1. ОС загружает код из исполняемого файла в память
2. выделяет память под стек и другие данные
3. передает управление на точку входа (Entry Point)

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]
0xffffffffff600000	0xffffffffff601000	--x	[vsyscall]

Edb: Memory Regions (Linux, x64)

Машинный код



Двоичный код программы: последовательность кодов команд (опкодов) процессора

Ассемблер: текстовая запись для удобства

Система команд зависит от архитектуры:

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

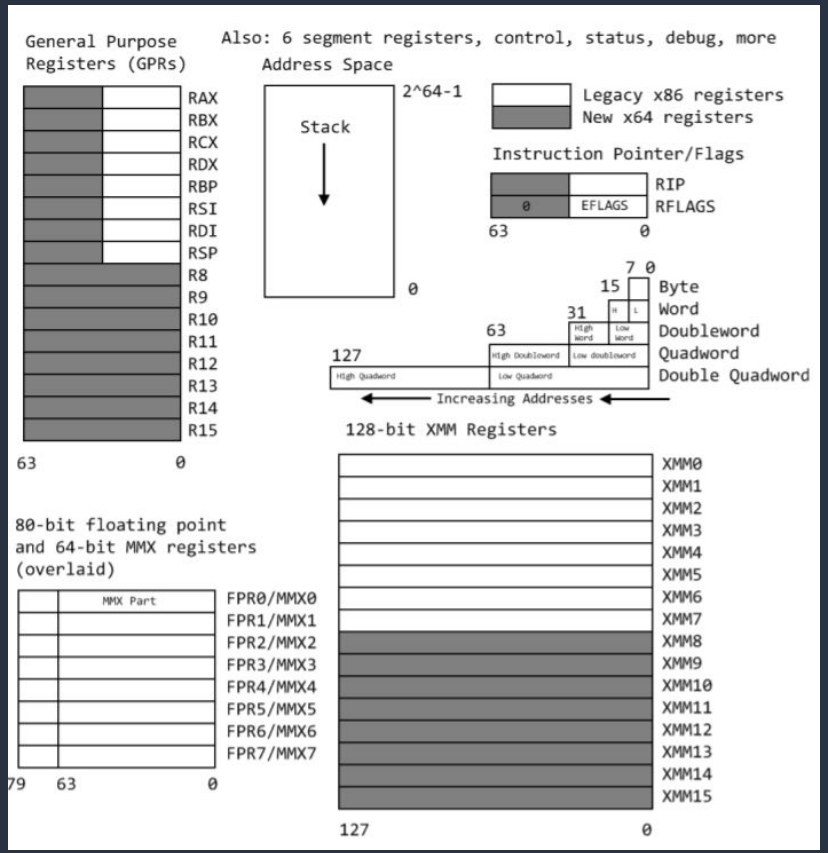
Пример

```
000055d6:ac1551d1  f3              db 0xf3
000055d6:ac1551d2  0f              db 0x0f
000055d6:ac1551d3  1e              db 0x1e
000055d6:ac1551d4  fa              cli
000055d6:ac1551d5  55              push rbp
000055d6:ac1551d6  48 89 e5        mov rbp, rsp
000055d6:ac1551d9  48 83 ec 10     sub rsp, 0x10
000055d6:ac1551dd  89 7d fc        mov [rbp-4], edi
000055d6:ac1551e0  48 89 75 f0     mov [rbp-0x10], rsi
000055d6:ac1551e4  83 7d fc 01     cmp dword [rbp-4], 1
000055d6:ac1551e8  7f 13           jg 0x55d6ac1551fc
000055d6:ac1551ea  48 8d 3d 1f 0e 00 00   lea rdi, [rel 0x55d6ac15600f]   ASCII "Enter argument!"
000055d6:ac1551f1  e8 8b fe ff ff  call 0x55d6ac155080
000055d6:ac1551f6  b8 01 00 00 00  mov eax, 1
000055d6:ac1551fb  eb 18           jmp 0x55d6ac155214
000055d6:ac1551fd  48 8b 45 f0     mov rax, [rbp-0x10]
000055d6:ac155201  48 83 c0 08     add rax, 8
000055d6:ac155205  48 8b 00        mov rax, [rax]
000055d6:ac155208  48 89 c7        mov rdi, rax
000055d6:ac15520b  e8 7a ff ff ff  call hello!hello
000055d6:ac155210  b8 00 00 00 00  mov eax, 0
000055d6:ac155215  c9              leave
000055d6:ac155216  c3              ret
```

```c
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?

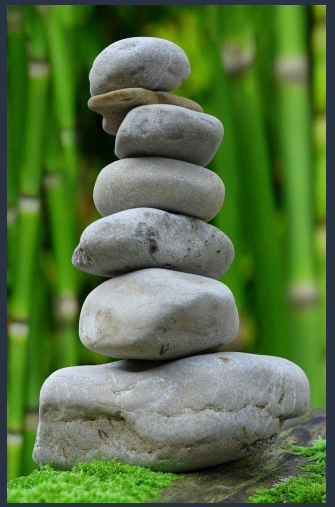
Стек – зарезервированная область памяти для работы с динамическими данными.

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

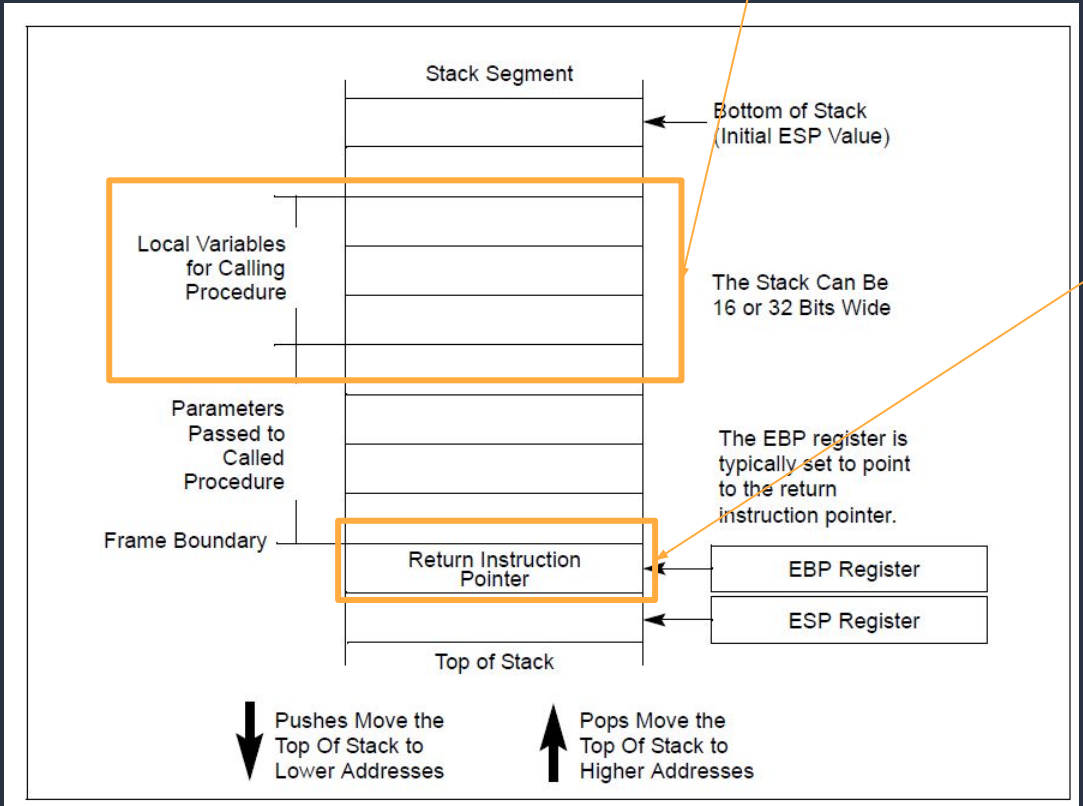
- помещает в стек по адресу в регистре rsp (то есть в вершину стека), текущее значение регистра rip (счетчика команд) — адрес возврата
- переходит по адресу, указанному в качестве операнда

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

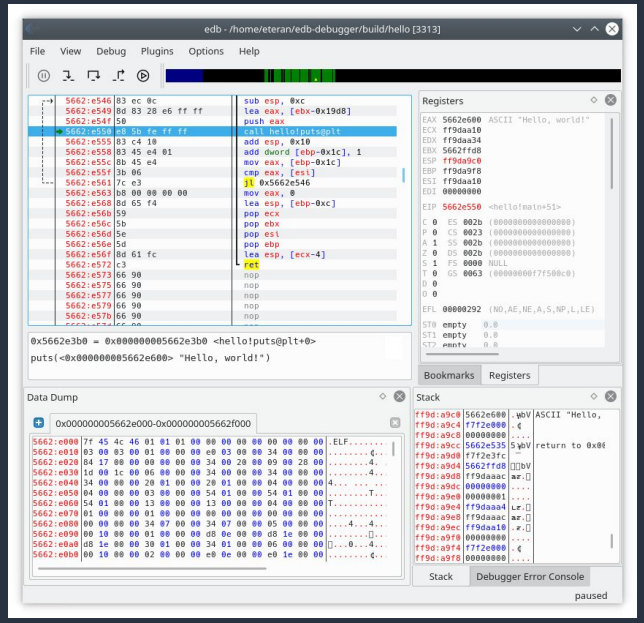
Можно подставить свой адрес и при вызове **ret** программа перейдет по нему

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

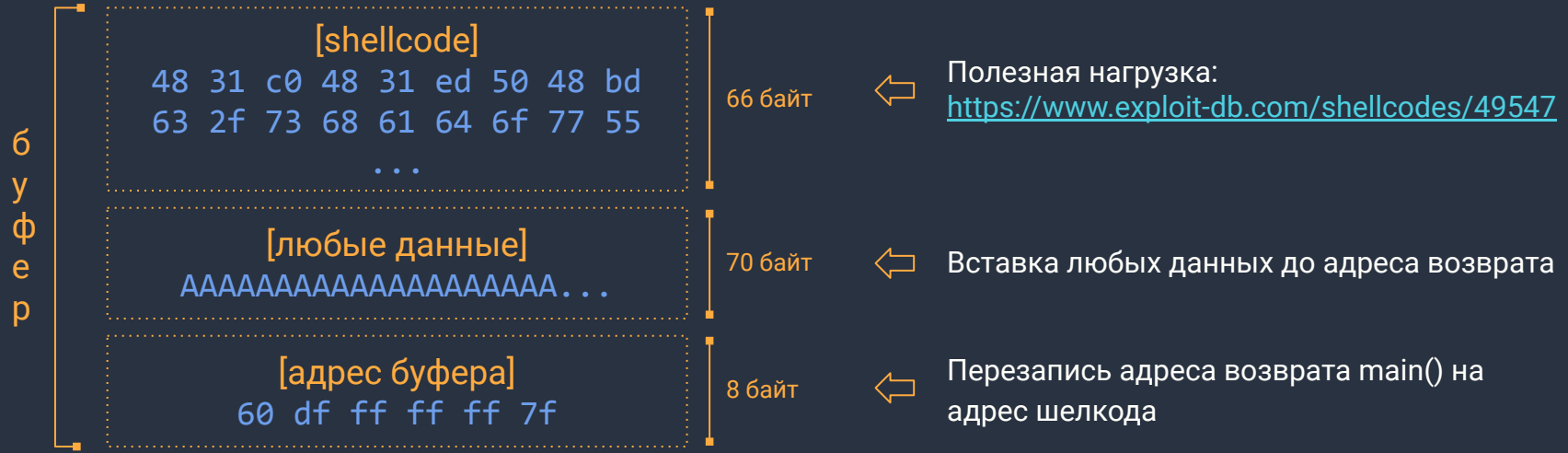


GDB: консольный отладчик, анализ coredump

```
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 0x7fffffffe850:
 rip = 0x40060d in main (test.c:8); saved rip 0x7ffff7a77ead
 source language c.
 Arglist at 0x7fffffffe840, args:
 Locals at 0x7fffffffe840, Previous frame's sp is 0x7fffffffe850
 Saved registers:
  rbp at 0x7fffffffe840, rip at 0x7fffffffe848
(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 “теневой 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