МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С.П. КОРОЛЕВА»
(Самарский университет)

*Борисов А.Н.*

# ПРОСТЫЕ УЯЗВИМОСТИ. ПЕРЕПОЛНЕНИЕ НА СТЕКЕ. УЯЗВИМОСТИ ФОРМАТНОЙ СТРОКИ

*Методические указания к лабораторной работе 4*

Самара, 2024

# СОДЕРЖАНИЕ

**Цели и задачи лабораторной работы**

Цель лабораторной работы: изучение способов эксплуатации уязвимостей переполнения буфера на стеке и связанных с ними уязвимостей форматной строки.

Задание на лабораторную работу: для заданных вариантом исполняемых файлов добиться вывода на экран надписи “Access granted”.

# 1 Введение

## 1.1 Буферы

Буфер – область памяти, предназначенная для временного размещения данных.

Наиболее часто буферы используются при операциях ввода-вывода. Поскольку данные, поступающие при вводе, могут быть невалидными или напрямую вредоносными, задача программиста – обеспечивать проверку данных.

Наиболее простым случаем является ситуация, при которой размер вводимых данных превышает размер буфера. В языке C++ данная проблема решается автоматически классами стандартной библиотеки. В языке C задача контроля за размером данных требует участия программиста.

## 1.2 Об аргументах программы и переменных среды

Помимо очевидных каналов ввода информации (поток ввода, файлы, сетевые сокеты) существует еще два канала – аргументы программы и переменные среды.

Переменная среды – строка вида “<name>=<value>”, где name – имя переменной среды. Переменные среды могут использоваться для изменения поведения программы, если она ее использует. Например, переменная PATH содержит пути поиска по умолчанию и используется самой ОС, а переменная CUDA_VISIBLE_DEVICES влияет на работу программ, использующих NVIDIA GPU.

Переменные среды задаются до запуска программы. В Windows PowerShell задать значение переменной среды можно командой:

```
$env:<name>=<value>
```

В Linux для оболочки bash и ее наследников используется команда

```
export <name>=<value>
```

Чтение переменных среды в языках C/C++ производится с помощью функции `char* getenv(const char* name)`, которая возвращает указатель на значение переменной среды либо NULL.

Доступ к аргументам программы осуществляется через аргументы `argc` и `argv` функции main, где `argc` – количество аргументов, `argv` – массив указателей на строки аргументов. Данный массив оканчивается указателем NULL. Первым аргументом `argv[0]` по общему соглашению является команда, запустившая процесс (обычно – имя программы).

И аргументы программы, и переменные среды копируются в память процесса, даже если программа их не использует.

В Windows переменные среды и аргументы программы находятся в отдельном сегменте, содержащем Process Environment Block.

В Linux переменные среды и аргументы программы находятся в начале стека. Т.е., читая стек мы рано или поздно дойдем до аргументов и переменных, что может использоваться в том числе при эксплуатации уязвимостей форматной строки (printf+%n).

В Linux аргументы программы могут использоваться для того, чтобы передавать 0, поскольку пустая строка содержит только байт 0. К примеру, в результате передачи следующих аргументов в области аргументов будет сформировано число 0x00000031(пустыми кавычками передаётся пустой аргумент)

```
./foo 1 "" ""
```

### 1.3 Спецификаторы форматной строки

Таблица 1. Спецификаторы printf

| **Спецификатор** | **Тип аргумента** | **Примечание** |
|---|---|---|
| %d | `int` | |
| %u | `unsigned int` | |
| %lld | `long long` | |
| %llu | `unsigned long long` | |
| %x | `unsigned int` | в 16-ричном формате |
| %llx | `unsigned long long` | в 16-ричном формате |
| %p | `void*` | |
| %s | `const char[]` | Вывод завершается при встрече символа с кодом 0 |
| %c | `char` | |
| %f | `double` | |
| %lf | `double` | |
| %n | `int*` | по указателю записывается количество записанных на момент встречи спецификатора символов |

Таблица 2. Спецификаторы scanf

| **Спецификатор** | **Тип аргумента** | **Примечание** |
|---|---|---|
| %d | `int*` | |
| %u | `unsigned int*` | |
| %lld | `long long*` | |
| %llu | `unsigned long long*` | |
| %x | `unsigned int*` | в 16-ричном формате |
| %llx | `unsigned long long*` | в 16-ричном формате |
| %d | `int*` | формат числа выводится автоматически |
| %p | `void**` | |
| %s | `char*` | В буфере должно быть достаточно места для приема строки |
| %c | `char*` | 1 символ |
| %f | `float*` | |
| %lf | `double*` | |
| %n | `int*` | по указателю записывается количество считанных на момент встречи спецификатора символов |

Спецификаторы, помимо типа значения, могут содержать дополнительные модификаторы. Например, добавление числа сразу после % задает ширину вывода, что позволяет выводить больше символов, чем содержится в значении и обычно используется для выравнивания. Например, `printf("%10d", 1)` выведет «         1» (отметьте дополнительные пробелы), а `printf("%010d", 1)` выведет «0000000001».

Кроме того, что в Unix-подобных ОС спецификаторы printf() могут переупорядочиваться с помощью специального модификатора. Например, `printf("%s %s", "hello", "world")`выведет **«hello world»**, в то время как `printf("%2$s %1$s", "hello", "world")` выведет «**world hello»**. Использование данного модификатора может упростить решение задач на Unix-подобных системах, поскольку фактически позволяет осуществлять почти произвольное чтение с помощью printf().

## 2 Переполнение буфера на стеке

### 2.1 Общая схема

Стек вызовов – область памяти, в которой находятся данные, необходимые для выполнения функций. К таким данным относятся локальные переменные, адреса возврата и сохраненные значения регистров.

Среди локальных переменных встречаются массивы, обычно играющие роль буфера, в который записываются вводимые данные. Буфер на стеке обычно имеет фиксированный размер (но не всегда, см. alloca). При этом буфер не может быть большим, т.к. максимальный размер всего стека ограничен (обычно 1-2 МБ).

Из-за того, что размер буфера фиксирован, программист обязан проверять размер вводимых данных, дабы избежать выхода данных за пределы буфера. Однако стандартные функции scanf, gets и strcpy по умолчанию не проверяют длину данных, что автоматически приводит к опасности переполнения буфера.

Рассмотрим ситуацию на рисунке снизу. Для пользовательского ввода используется буфер S размером 16 символов. Если пользователь введет больше, то будут затронуты сначала локальные переменные, затем сохраненное значение RBP, затем адрес возврата, а затем – кадр вызывающей функции. На рисунке показан результат ввода строки «Lorem ipsum dolor sit amet, consectetur adipiscing»

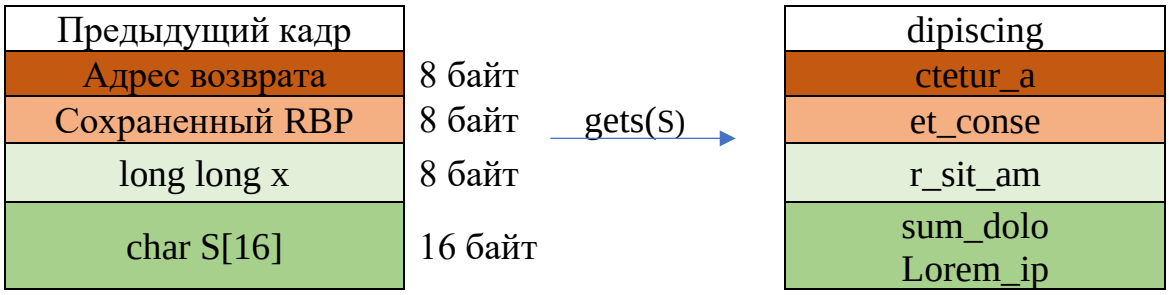


Рисунок 1 – Результат переполнения буфера на стеке

В большинстве случаев подобное поведение не является желательным, однако возможность менять значения переменных и/или адрес возврата, изменяя тем самым ход выполнения программы, могут быть полезны для достижения определенных целей.

### 2.2 Перезапись переменных и регистров

Перезапись значения переменной становится возможной, если переменные на стеке находятся над буфером (переменная *x* на рисунке 1).

Стоит отметить, что в настоящее время компиляторы стараются удерживать локальные переменные в регистрах, и используют стек только, если все регистры уже использованы, если переменная находится в изменяемом регистре перед вызовом функции, или если ниже по коду берется адрес локальной переменной (например, для передачи по ссылке) – т.к. у регистра нет адреса, компилятору приходится «материализовывать» переменную на стеке.

Перезаписать значение регистра можно только косвенно. Если компилятор сохранил значение регистра на стеке в прологе, то он восстановит его в эпилоге. Перезапись сохраненного значения на стеке не поможет изменить поведение вызывающей функции, т.к. в текущей функции данное значение не используется. На рисунке 1 в момент выхода из функции будет изменено значение регистра RBP.

### 2.3 Перезапись адреса возврата

Адрес возврата указывает точку, в которую будет передано управление по завершении функции.

Каждая инструкция кода имеет адрес. Перезаписывая адрес возврата, мы можем перейти к данной инструкции, минуя все остальные. К примеру, можно обойти проверку некоторого значения. На рисунке 2 показан результат перезаписи адреса возврата в функции vulnerable.

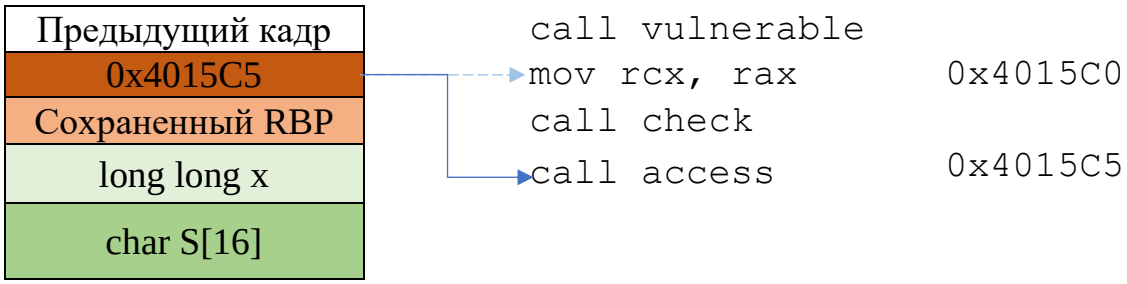


Рисунок 2 – Изменение адреса возврата

Стоит отметить, что итоговый адрес становится известен только после начала выполнения программы. Если программа всегда загружается по одному и тому же адресу, то целевой адрес достаточно узнать один раз. При использовании отладчика задача расчета целевого адреса вовсе отсутствует, т.к. адрес любой переменной и любой точки кода можно просто посмотреть, однако при осуществлении реальных атак возможности подключить отладчик к процессу нет.

Стоит также отметить, что данные в пределах одного сегмента имеют то же самое расположение друг относительно друга. Например, пусть 2

инструкции I1 и I2 расположены в исполняемом файле на расстоянии N байт. После загрузки программы смещение не изменится, поэтому если после загрузки программы инструкция I1 будет размещена по адресу A, то инструкция I2 будет размещена по адресу A+N.

### 2.4 Выполнение произвольного кода

Опкоды инструкций являются обычными двоичными числами. Как следствие, введенная строка может быть вполне проинтерпретирована, как некоторый набор инструкций, если передать на нее управление.

К примеру, последовательность инструкций

```
xor rcx,rcx
mov eax, 0xAABBCCDD
```

после компиляции будет иметь вид

```
0x31 0xC9 0xB8 0xDD 0xCC 0xBB 0xAA
```

Что может быть представлено в виде строки `1É¸ÝÌ»ª`

Данная атака потребует перезаписи адреса возврата, чтобы передать управление на записанный в буфер код.

Для облегчения попадания в начало кода используется последовательность инструкций `NOP` (код 0x90). В начале кода вставляется длинная последовательность таких инструкций, и при попадании на любую из них исполнение постепенно доходит до требуемого кода (т.е. не требуется точного попадания на начало кода).

### 2.5 Методы защиты и их отключение

#### 2.5.1 Использование защищенных функций

Наиболее надежным и простым методом защиты является использование функций, поддерживающих указание максимального размера буфера (таблица 3).

Таблица 3. Уязвимые функции и их безопасные аналоги.

| Уязвимая функция | Безопасная функция |
|---|---|
| `gets(buffer)` | `fgets(buffer, bufsize, stdin)` |
| `scanf("%s", buffer)` | `scanf_s("%s", buffer, bufsize)` |
| `strcpy(buffer, from)` | `strncpy(buffer, from, bufsize)` |
| `strcat(buffer, from)` | `strncat(buffer, from, bufsize)` |

#### 2.5.2 Канарейки

Канарейкой называется специальное значение, которое кладется на стек между буфером и адресом возврата (см. рисунок 3). Данное значение

генерируется заново при каждом запуске программы. Копия данного значения сохраняется в другом сегменте.

Значение канарейки на стеке и ее копии сравнивается в эпилоге функции. При переполнении буфера значение канарейки будет изменено, и проверка провалится. При неудачной проверке процесс завершает работу.

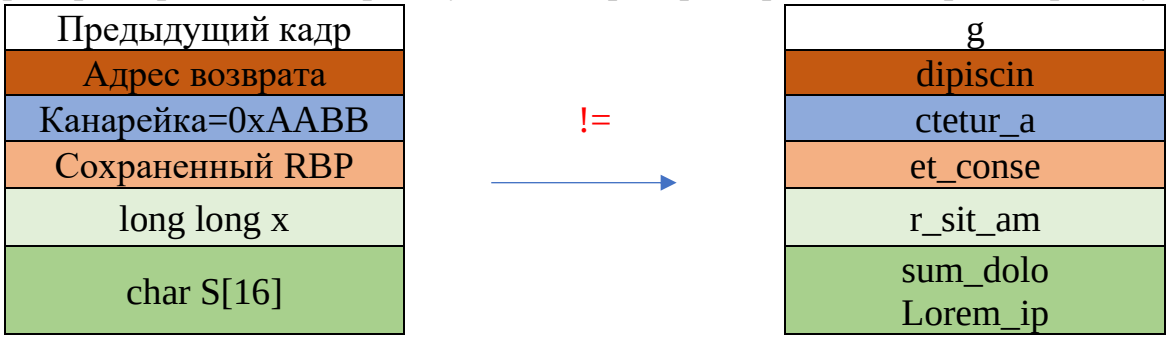


Рисунок 3 – Канарейка

Канарейки — это мера, реализуемая компилятором. В gcc/g++ за включение/отключение канареек отвечает флаг -f[no-]stack-protector (https://godbolt.org/z/Kxsc4fsE6). В MSVC для той же цели служат флаги /GS[-]

### 2.5.3 Запрет исполнения кода в сегментах данных

Изначально вся память была доступна для чтения, записи и исполнения. Со временем каждому сегменту памяти были назначены собственные разрешения: на чтение, на запись и на выполнение. К примеру, сегмент .text имеет разрешения на чтение и на выполнение, сегмент .rodata – только на чтение, а сегменты .stack/.data/.bss имеют разрешения на чтение/запись, но не на выполнение.

Если у сегмента нет разрешения на выполнение, то при попытке передать управление на адрес внутри этого сегмента (инструкциями jmp/call/ret) возникнет аппаратное исключение и процесс завершится. Подобная защита делает невозможным прямую запись кода в стек с последующим выполнением.

Отключить данную меру защиты в Linux можно на уровне компилятора, который в ходе сборки программы расставляет разрешения для сегментов в таблице сегментов исполняемого файла. GCC/G++ требуют флага -z execstack для включения исполняемого стека.

В Windows 10 данная мера (называемая DEP) может быть отключена только на уровне системы и только для 32-битных приложений. Для 64-битных приложений, а также в Windows 7/8/8.1, данная мера может быть отключена только глобально для всей системы и всех приложений разом,

что делать в основной системе СТРОГО ЗАПРЕЩЕНО (но можно внутри виртуальной машины).

### 2.5.4 Address Space Layout Randomization

Address Space Layout Randomization (рандомизация адресного пространства) – наиболее сильная мера защиты из представленных. Данная мера защиты загружает исполняемый файл и библиотеки и размещает стек и кучу по случайным адресам.

Поскольку адреса варьируются от запуска к запуску, атакующий не может заранее узнать адрес, по которому следует передавать управление. Как следствие, он не может выполнить произвольный код (если только в программе нет второй уязвимости, по которой может «утечь» нужный адрес).

В Linux ASLR реализован полностью – адреса являются случайными при каждом запуске программы.

В Windows ASLR реализован упрощенно – адреса варьируются от перезагрузки к перезагрузке системы (т.е. до следующей перезагрузки адреса библиотек будут те же между разными запусками программы). Кроме того, расположение стека не рандомизируется, если исполняемый файл не был собран с поддержкой ASLR (см. ниже).

Библиотеки, как правило, всегда размещаются по случайным адресам. Исполняемые файлы могут размещаться, если при их сборке были указаны соответствующие флаги: /DYNAMICBASE для MSVC, -fpie для gcc/g++/clang.

Отключать ASLR для выполнения лабораторных в Windows нет смысла, т.к. исполняемые файлы собраны без включения ASLR. В Linux отключить ASLR на общесистемном уровне можно командой

```
echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

## 3 Уязвимости форматной строки

### 3.1 Общая схема

Форматная строка – один из аргументов функций prinf/sprint/fprintf, задающий трансформацию последующих аргументов функции в строку.

Уязвимость форматной строки возникает, если в качестве форматной строки передается пользовательский ввод, который может иметь произвольные форматные спецификаторы.

Код ниже содержит уязвимость форматной строки, но не уязвимость переполнения буфера.

```
#include <stdio.h>
int main(){
    char buffer[128];
    fgets(buffer, 128, stdin);
    printf(buffer);
}
```

### 3.2 Чтение стека

Наиболее простым результатом эксплуатации уязвимости форматной строки является возможность чтения стека и некоторых регистров.

Если в качестве ввода передать “%i%i%i%i”, то программа выведет несколько целых чисел вместо введенной строки. Значение этих чисел зависит от используемого соглашения о вызовах. Для cdecl данные значения будут прочитаны напрямую со стека. Для Microsoft x64 первые 3 значения будут прочитаны из RDX, R8, R9, четвертое значение и далее – со стека. Для System V первые 5 целочисленных значений будут взяты из регистров, далее – со стека.

В ходе чтения, на экран сначала будет выведено содержимое текущего кадра стека, затем содержимое предыдущих кадров стека. На Linux далее будут выведены аргументы программы и переменные среды.

### 3.3 Запись с помощью printf

Наиболее интересной возможностью, предоставляемой уязвимостью форматной строки, является возможность записи.

Запись происходит при указании спецификатора %n, который осуществляет запись количества выведенных на экран символов по соответствующему указателю.

Если интересующий адрес известен, можно заранее записать данный адрес на стек, дойти до него с помощью %i и других спецификаторов, а затем записать по этому адресу некоторое ненулевое значение с помощью %n.

### 3.4 Меры защиты

Единственной мерой защиты от уязвимости форматной строки является использование корректной функции `puts`, которая просто выводит строку без форматирования.

## 4 Задание на лабораторную работу

### 4.1 Общие замечания

В данной лабораторной работе предполагается использование специализированного ПО для анализа кода исполняемых файлов. Итоговый выбор ПО и метода анализа исполняемого файла не ограничивается: анализировать файл можно как угодно и чем угодно, главное – чтобы поставленная задача была решена.

### 4.2 О вводе специальных символов

Последовательно байтов, содержащая 0x00, не всегда может быть передана программе через стандартный ввод. Однако возможно передать данную последовательность через аргументы программы или при перенаправлении потока ввода на файл.

Байты из диапазона 0x00-0x1F в кодировке ASCII являются специальными символами. Ввести их в консоли трудно, поэтому можно использовать другой подход – перенаправление ввода.

Создать текстовый файл с заданным контентом можно легко создать с помощью продвинутых текстовых редакторов (Notepad++) или с помощью Python-скрипта. *Рекомендуется использовать Python2, т.к. в Python3 стандартной кодировкой строк является UTF-8.*

Передать содержимое файла на вход программы (перенаправить поток ввода программы) можно следующими командами.
Windows(Powershell): `Get-Content {файл} -Raw | ./{программа}`
Windows(cmd): `{программа} < {файл}`
Linux: `cat {файл}|./{программа}` или `./{программа} <{файл}`

Также можно передать напрямую вывод скрипта (или любой другой программы) командой
Linux и Windows(Powershell): `python {скрипт} | ./{программа}`
Windows (cmd): `python {скрипт} | {программа}`

Стоит отметить, что при использовании Powershell могут наблюдаться проблемы, которых не было при использовании cmd (обычной командной строки).

### 4.3 IDA и ввод-вывод (Linux)

IDA напрямую не позволяет вводить данные и видеть вывод, поэтому ввод/вывод производится путем перенаправления ввода/вывода в файлы. Перенаправление производится в настройках: вкладка Debugger/Process Options, поле Parameters (рисунок 4). Синтаксис перенаправления: < {файл ввода} > {файл вывода}

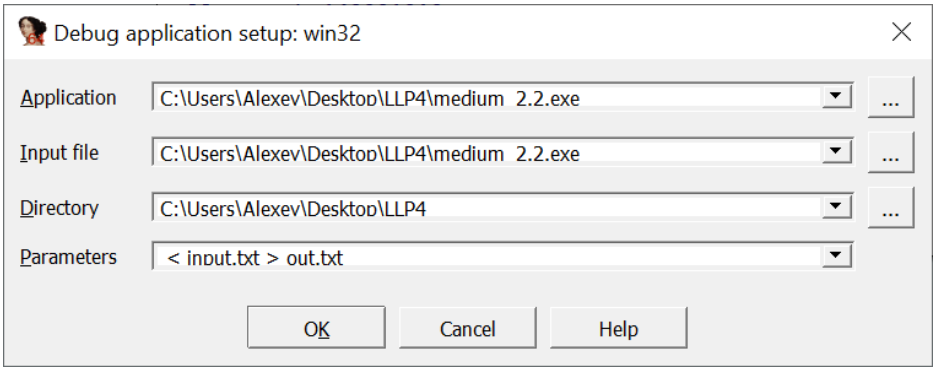


Рисунок 4 – Перенаправление ввода в IDA

### 4.4 Задание на лабораторную работу

При запуске заданного исполняемого файла добиться вывода на экран последовательности символов «Access granted». Программа при этом *может* завершаться с ошибкой, главное добиться вывода фразы.

Комплект исполняемых файлов предоставляется преподавателем. Файлы для разных уровней сложности находятся в разных подпапках с именами уровней.

Номер варианта совпадает с именем файла.

*Выполнение л/р на среднем уровне требует первоначального выполнения л/р на легком уровне.*

*Выполнение л/р на сложном уровне требует первоначального выполнения л/р на среднем и легком уровнях.*

***Редактировать исполняемые файлы и изменять состояние запущенной программы с помощью отладчика запрещается.***