Министерство науки и высшего образования РФ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Новгородский государственный университет имени Ярослава Мудрого»

Кафедра информационных технологий и систем

**Переполнение буфера**

Лабораторная работа по дисциплине:

«Специальные методы информационной безопасности»

Выполнил

Студент группы №1094

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.А.Менделеев

\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2021

Проверил

Преподаватель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_П.М.Довгалюк

\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_2021

**Великий Новгород**

**2021**

**Оглавление**

[**1.** **Постановка задачи** 3](#_Toc86700202)

[**2.** **Общие сведения для решения задач** 3](#_Toc86700203)

[**3.** **Программа auth\_overflow** 3](#_Toc86700204)

[**4.** **Программа auth\_overlow2** 5](#_Toc86700205)

## **Постановка задачи**

С помощью эксплуатации уязвимости переполнения буфера добиться того, чтобы обе программы выдавал сообщение «**Access granted**».

В отчёте описать каким образом формировались входные данные и что для этого потребовалось выяснить.

## **Общие сведения для решения задач**

Когда программа начинает свою работу, для её выполнения выделяется место в памяти под две структуры: стек и куча.

В куче хранится динамически распределяемая память приложения, выделенная с помощью оператора **new** или функции **malloc()**.

Стек хранит в себе информацию о порядке вызовов функций и их локальных переменных и параметров. Стек растёт в сторону уменьшения адресов. Каждый элемент стека занимает одно машинное слово в зависимости от режима работы процессора. Для работы со стеком существует два регистра: **rbp** и **rsp**. Первый регистр предназначен для обращения к локальным переменным и параметрам вызванной функции, второй регистр указывает на вершину стека, куда добавляются данные программы.

Перед вызовом какой – либо функции в стеке выделяется память под адрес возврата, который необходим для возврата на инструкцию, которая следует после вызванной функции, и старый адрес регистра **rbp**. После того, как необходимые данные были сохранены формируется стековый кадр – это область памяти, предназначенная для хранения локальных переменных вызванной функции и её параметров. Выделение памяти под них происходит в прологе функции. Пример пролога функции:

*Листинг 1. Пролог функции:*

push %rbp

mov %rsp, %rbp

sub $0x10, %rsp

……

В листинге 1 выделение памяти под локальные переменные функции происходит путём смещения адреса, хранящегося в регистре **rsp**, на 16 байт (0x10 в шестнадцатиричной системе счисления).

## **Программа auth\_overflow**

Работа данной программы устроена следующим образом. В начале программы у пользователя через терминал запрашивается пароль для выдачи прав пользователю. Если пароль совпал с **brillig** или **outgrade**, то будут выданы права и пользователю в консоли отобразится надпись «**Access Granted**». В противном случае программа выведет «**Accesss Denied**». Проверка пароля происходит при помощи статического символьного массива **password\_buffer**, размер которого равен 16 байтам. В него при помощи функции **strcpy()** копируется введённый пароль, переданный в функцию через параметр password.

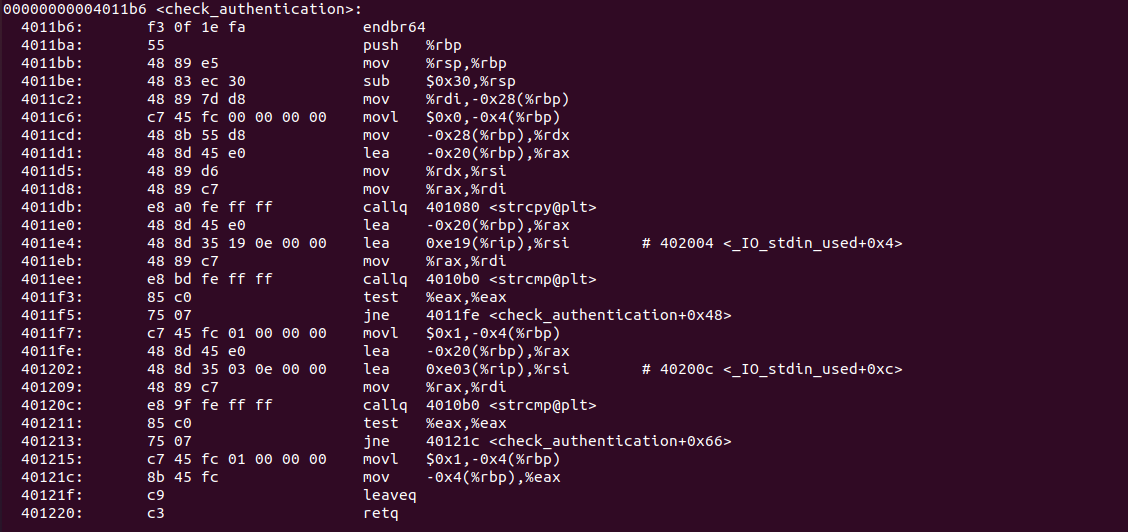
Для обнаружения уязвимости в программе дизассемблируем исполняемый файл программы при помощи команды **objdump -d auth\_overflow** в терминале Ubuntu. После выполнения данной команды получим следующий ассемблерный код функции **check\_authentication()**.

Рисунок 1. Ассемблерный код функции check\_authentication().

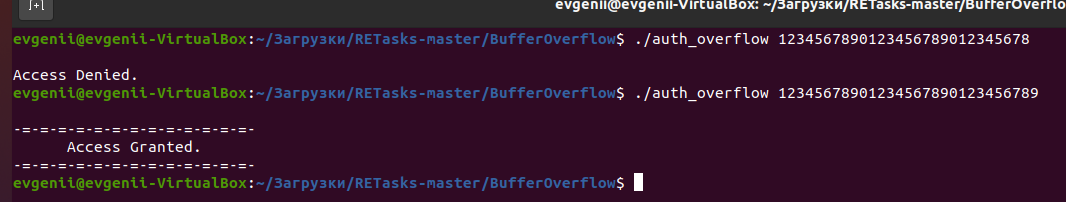
Исходя из рис. 1, можно сказать, что на стеке выделяется память в 48 байт. Переменная auth\_flag, отвечающая за выдачу прав пользователю, находится по адресу **-0х4(%rbp)**. Об этом свидетельствует строка кода **movl $0x0, -0х4(%rbp)**, которая сохраняет ноль, которым была инициализирована переменная **auth\_flag** в начале функции, по указанному адресу, а также размер области памяти от начала стекового кадра до адреса **-0х4(%rbp)** равно четырём байтам. Область память, выделенная под массив **password\_buffer**, находится по адресу **-0x20(%rbp)**. Это можно узнать исходя из строки кода **lea -0x20(%rbp), %rax**. Таким образом, расстояние, на котором находятся переменная **auth\_flag** и массив **password\_buffer** равно 28 байтам. Учитывая ещё то, что начало массива находится выше переменной **auth\_flag**, мы можем переписать значение переменной, если в массив будет введено больше 28 байтов. Таким образом, переменная **auth\_flag** будет перезаписана иможет принять десятичный код символа 29 байта массива отличного от нуля, в результате чего даже если введённый пароль пользователя не будет совпадать с теми, что уже прописаны в программе, права будут ему всё равно выданы. Пример работы программы в таком случае выглядит следующим образом:

Рисунок 2. Результат работы программы auth\_overflow при переполнении буфера.

При вводе строки размером в 28 символов программа выдаёт «Access Denied», но уже при вводе строки более чем 29 символов последний байт строки сохраняется в участок памяти, где хранится значение переменной **auth\_flag**, которому в десятичной системе счисления соответствует число 39. Таким образом, результатом работы функции **check\_autentication()** является число 39, в результате чего выдаются права пользователю.

По этой причине данная программа имеет уязвимость.

## **Программа auth\_overlow2**

Данная программа выполняет те же действия, что и программа **auth\_overflow** с той разницей, что на этот раз считывание в массив **password\_buffer** происходит с помощью функции **scanf**. Если пароль, хранящийся в массиве, совпадает с теми, что прописаны в программе, то функция **check\_authentication()** при помощи ключевого слова **return** возвращает единицу, в результате чего вызывается функция **granted()** и пользователю выдаются права, в противном случае ноль.

Для анализа уязвимости дизассемблируем программу при помощи команды **objdump -d auth\_overflow2** в терминале Ubuntu.

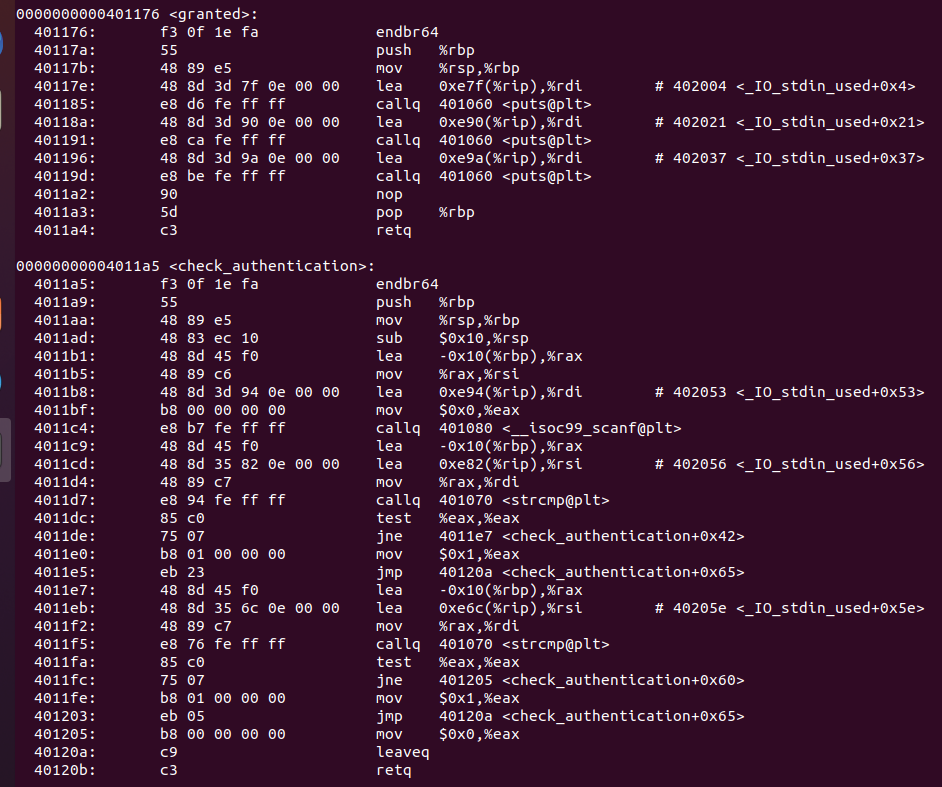


Рисунок 3. Участок ассемблерного кода программы auth\_overflow2.

Исходя из пролога функции **check\_authentication()**, для локальных переменных функции выделяется 16 байт под массив **password\_buffer**. Об этом свидетельствует строчка кода **sub $0x10, %rsp**. Уязвимость данной программы заключается в том, что пользователь может перезаписать элемент стека, который хранится в себе адрес возврата на инструкцию после вызова функции **check\_authentication()**. Таким образом, он может вписать адрес функции **granted()** в памяти для того, чтобы перенаправить исполнение кода для получения определённых прав, диктуемые программой.

Для того, чтобы определить количество байт, которое необходимо перезаписать, чтобы добраться до участка памяти, где хранится адрес возврата, необходимо определить размер сегмента данных в стеке, в котором хранится старый адрес регистра **%rbp**, иными словами необходимо узнать размер машинного слова. В моём случае используется 64 – разрядный процессор, поэтому для него размер машинного слова будет равно 8 байтам.

Исходя из описанного выше, выходит, что 16 байт необходимо для записи символов в массив **password\_buffer**, 8 байт необходимо, чтобы затереть старый адрес регистра **%rbp**, что в сумме даёт 24 байта. Начиная с 25 байта необходимо вписать адрес функции **granted()**, который равен 0x0000000000401176.

Пример работы программы в таком случае выглядит следующим образом:

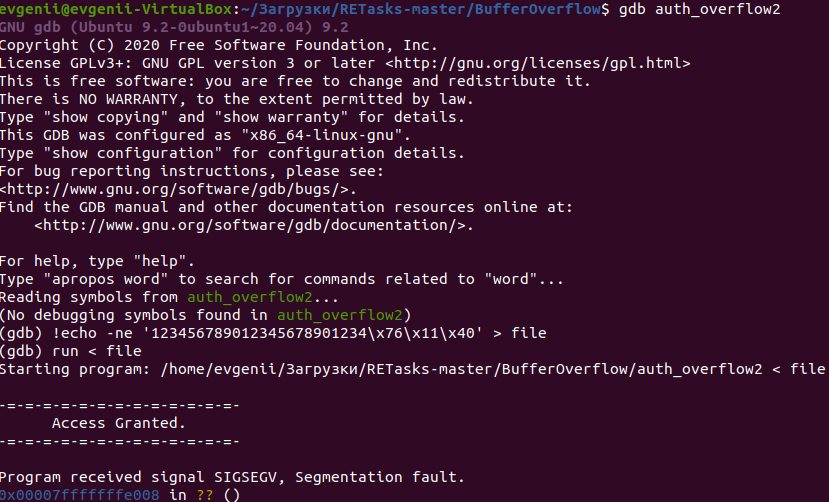


Рисунок 4. Работа программы auth\_overflow2 при переполнении буфера.

Команда **!echo -ne ‘123456789012345678901234\x76\x11\x40’ > file** использовалась для того, чтобы в массив **password\_buffer** правильно сохранялся адрес функции **granted()**. При вводе строки размером более 24 символа, программа становится уязвимее, так как начиная с 25 символа пользователь может перезаписать адрес возврата, в результате чего может перенаправить нить исполнения кода. Из рис. 4 видно, что при таком условии можно вписать адрес функции **granted()**, которая выдаёт права.

По этой причине данная программа имеет уязвимость.