Атаки переполнения integer

Выполнил студент 4-го курса
Московского Физико-Технического Института
Чухланцев А.А.

Введение

В этом эссе я опишу два типа ошибок, которые могут привести к инжектированию зловредного кода. Оба типа ошибок возникают вследствие того, что программе передаются переменные с совершенно неожиданными значениями. Типы, ошибок которые я здесь опишу, сами по себе не дают возможности инжектировать код, но они могут неплохо работать в сочетании с другими типами ошибок – ошибки переполнения буфера. Что же такое integer и почему в нем возможны переполнения? Integer это целочисленный тип данных, как правило его длина равна разрядности системы на которой используется этот тип(на 32 разрядных системах – 32 бита, на 64 разрядных -64 бита). Integer, как и другие типы данных, это просто ячейки памяти. Рассмотрим 32 разрядный integer. Максимальное значение, которое можно записать в этот integer равно 0xffffffff. Сразу возникает вопрос, а что же будет, если к этому числу прибавить 1? В результате прибавления к 0xffffffff единицы получиться число 0x100000000, которое явно не влезает в тип integer, поэтому, когда процессор выполнит сложение, он запишет 0 в содержимое переменной, а разряд, который не уместился в integer, он перенесет в CarryFlag(один из флагов в регистре flags). Отсюда следует, что число 0xffffffff ведет себя как -1. Можно считать отрицательными числа с установленным первым наиболее значащим битом, так и делается в современной компьютерной технике.

Переполнение длины

Таким образом, ошибки переполнения integer результат попытки сохранить значение в переменной, которая слишком мала, чтобы в ней могло находиться это значение. Самый простой пример этого, может быть продемонстрирован, просто-напросто присвоением содержимого большой переменной маленькой. Example2_5_2:

```
#include <stdio.h>

int main(void){
        int l;
        short s;
        char c;

        l = 0xdeadbeef;
        s = l;
        c = l;

        printf("l = 0x%x (%d bits)\n", l, sizeof(l) * 8);
        printf("s = 0x%x (%d bits)\n", s, sizeof(s) * 8);
        printf("c = 0x%x (%d bits)\n", c, sizeof(c) * 8);

        return 0;
}
```

Результатом работы данной программы будет что-то похожее на это:

```
l = 0xdeadbeef (32 bits)
s = 0xffffbeef (16 bits)
c = 0xffffffef (8 bits)
```

Так как каждое присвоение является причиной выхода за пределы границ новой переменной, значение обрезается, так чтобы оно могло влезть в переменную, которой оно было присвоено.

Давайте рассмотрим простой пример:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]){
        unsigned short s;
        int i;
        char buf[80];

        if(argc < 3){
                return -1;
        }

        i = atoi(argv[1]);
        s = i;

        if(s >= 80){            /* [w1] */
                printf("Oh no you don't!\n");
                return -1;
        }

        printf("s = %d\n", s);

        memcpy(buf, argv[2], i);
        buf[i] = '\0';
        printf("%s\n", buf);

        return 0;
}
```

Хотя именно такую конструкцию вы вряд ли встретите в реальном коде, но подобные этой конструкции участки кода могут встретиться. Итак, посмотрим как наша программа будет реагировать на разные переданные данные:

```
nova:signed {100} ./width1 5 hello
s = 5
hello
nova:signed {101} ./width1 80 hello
Oh no you don't!
nova:signed {102} ./width1 65536 hello
s = 0
Segmentation fault (core dumped)
```

Аргумент, характеризующий длину строки, берется из командной строки как integer i. Когда значение переводиться в short integer s, оно обрезается, если значение слишком велико, для того чтобы влезть в s. Поэтому возможно пройти проверку границы массива на строчке [w1]. После обнаружения подобной ошибки могут быть использованы стандартные техники переполнения стека.

Арифметическое переполнение

Арифметическое переполнение возникает, тогда когда с данной переменной производиться операции сложения, умножения или вычитания, причем результатом этих операций может стать значение, которое не умещается в переменную результата. Рассмотрим пример арифметического переполнения:

```
#include <stdio.h>

int main(void){
        int l, x;

        l = 0x40000000;

        printf("l = %d (0x%x)\n", l, l);

        x = l + 0xc0000000;
        printf("l + 0xc0000000 = %d (0x%x)\n", x, x);

        x = l * 0x4;
        printf("l * 0x4 = %d (0x%x)\n", x, x);

        x = l - 0xffffffff;
        printf("l - 0xffffffff = %d (0x%x)\n", x, x);

        return 0;
}
```

Вывод программы:

```
nova:signed {55} ./ex4
l = 1073741824 (0x40000000)
l + 0xc0000000 = 0 (0x0)
l * 0x4 = 0 (0x0)
l - 0xffffffff = 1073741825 (0x40000001)
```

Из этого примера видно, что сложение, вычитание и умножение могут вызывать переполнения integer. В случае умножения и вычитания результат слишком велик для того, чтобы влезть в integer, поэтому он обрезается до размеров integer. С вычитанием все по-другому, оно вызывает переполнение скорее нижней границы integer, чем

переполнение верхней, как в сложении. В данном примере делается попытка сохранить в integer значение меньше чем минимальное значение, которое может содержаться в integer, поэтому происходит переворот. Таким образом, мы можем заставить вычитание складывать, сложение вычитать и умножение делить.
Часто подобные ошибки переполнения integer могут возникнуть при подсчете размера буфера. Рассмотрим пример:

```
---------------------------------------------------------------------------
   int myfunction(int *array, int len){
      int *myarray, i;

      myarray = malloc(len * sizeof(int));    /* [1] */
      if(myarray == NULL){
         return -1;
      }

      for(i = 0; i < len; i++){              /* [2] */
         myarray[i] = array[i];
      }

      return myarray;
   }
---------------------------------------------------------------------------
```

Этот с виду безобидный участок кода, является хорошим примером часто встречающейся ошибки. Если мы зададим len достаточно большое значение, такое что len*4 вызовет переполнение integer, то мы можем выделить памяти под массив в строчке [1] намного меньше, чем перезапишем в строчке [2]. Отсюда мы получаем типичную ошибку переполнения буфера.
Следующий пример:

```
-----------------------------------------------------------------------------
   int catvars(char *buf1, char *buf2, unsigned int len1,
            unsigned int len2){
      char mybuf[256];

      if((len1 + len2) > 256){    /* [3] */
         return -1;
      }

      memcpy(mybuf, buf1, len1);      /* [4] */
      memcpy(mybuf + len1, buf2, len2);

      do_some_stuff(mybuf);

      return 0;
   }
-----------------------------------------------------------------------------
```

В этом примере проверка в строке [3] может быть пройдена используя подходящие значения для len1 и len2, которые вызовут переполнение в сложении чисел и перевернут результирующее значение в маленькое число. К примеру, можно взять следующие значения:

```
   len1 = 0x104
   len2 = 0xfffffffc
```

если одно значение прибавить к другому, то результат будет равен 0x100(десятичное 256). Такое значение суммы позволит пройти проверку в строке [3], потом memcpy скопирует данные далеко за пределы буфера.

Ошибки, связанные со знаковостью и без знаковостью

Ошибки знаковости возникают, когда без знаковая переменная интерпретируется как знаковая, или когда знаковая переменная интерпретируется как без знаковая.
Знаковые ошибки могут принимать множество обликов, но есть некоторые характерные вещи, которые стоит искать в коде:

- знаковые переменные, использующиеся в сравнениях
- использование знаковых переменных в арифметике
- сравнение без знаковых переменных со знаковыми

Вот классический пример знаковой ошибки. Example2_5_4:

```
---------------------------------------------------------------------------
   int copy_something(char *buf, int len){
      char kbuf[800];

      if(len > sizeof(kbuf)){        /* [1] */
         return -1;
      }

      return memcpy(kbuf, buf, len);  /* [2] */
   }
---------------------------------------------------------------------------
```

Проблема в этом коде заключается в том, что на входе memcpy переменная len должна быть без знаковой, но проверка границ массива осуществляется до memcpy и она выполняется используя знаковое целое. Подавая на вход данной функции отрицательное значение len можно пройти проверку границ в строке [1], и потом в строке [2] вызовется memcpy, переменная len будет интерпретирована как без знаковая, в результате чего память будет перезаписана далеко за пределами буфера.
Другая проблема возникает, когда вдобавок к ошибке знаковости в программе выполняются арифметические операции. Рассмотрим следующий пример:

```
---------------------------------------------------------------------------
   int table[800];

   int insert_in_table(int val, int pos){
      if(pos > sizeof(table) / sizeof(int)){
         return -1;
      }

      table[pos] = val;

      return 0;
   }
---------------------------------------------------------------------------
```

Поскольку строчка

```
table[pos]=val;
```

эквивалентна строчке

```
*(table+(pos*sizeof(int)))=val;
```

Мы можем заметить, что код здесь не ожидает отрицательного значения pos. Ошибка сделанная в этой программе на много опаснее той, которая была сделана в предыдущей, поскольку в предыдущей программе нам приходилось подавать на вход отрицательное значение, которое трактовалось как очень большое положительное функцией memcpy, таким образом перезаписывался очень большой участок памяти, что могло вызвать ошибку Segmentation fault. В этом же примере мы можем подобрать такое значение pos, чтобы пройти проверку границы и перезаписать адрес возврата из функции.
Рассмотрим еще один пример:

```
---------------------------------------------------------------------------
   int get_two_vars(int sock, char *out, int len){
      char buf1[512], buf2[512];
      unsigned int size1, size2;
      int size;

      if(recv(sock, buf1, sizeof(buf1), 0) < 0){
         return -1;
      }
      if(recv(sock, buf2, sizeof(buf2), 0) < 0){
         return -1;
      }

      /* packet begins with length information */
      memcpy(&size1, buf1, sizeof(int));
      memcpy(&size2, buf2, sizeof(int));

      size = size1 + size2;       /* [1] */

      if(size > len){             /* [2] */
         return -1;
      }

      memcpy(out, buf1, size1);
      memcpy(out + size1, buf2, size2);

      return size;
   }
-----------------------------------------------------------------------------------------
```

Этот пример показывает ошибку, которая временами встречается в сетевых демонах, особенно когда информация о длине передается как часть пакета. Сложение в строке [1] используется, для того чтобы в строке [2] сделать проверку границы. Эту проверку можно обойти, если задать значения size1 и size2 такими, которые вызовут переполнение в переменной size. Вот пример таких значений:

```
size1 = 0x7fffffff
size2 = 0x7fffffff
size1+size2 = 0xfffffffe(-2)
```

Когда на вход подаются такие значения, проверка в [2] проходиться успешно, и память перезаписывается далеко за пределами буфера out.

Пример уязвимого кода из реальной жизни

Данный пример взят из [3]. Данная ошибка переполнения буфера была найдена в исходном тексте библиотеки отвечающей за открытие BMP фалов в Internet Explorer. Предоставляю вашему внимаю уязвимую часть кода:

```
---------------------------------------------------------------------------
   // Before we read the bits, seek to the correct location in the file
   while (_bmfh.bfOffBits > (unsigned)cbRead)  //1
   {
      BYTE abDummy[1024];
      int cbSkip;

      cbSkip = _bmfh.bfOffBits - cbRead;      //2

      if (cbSkip > 1024)                      //3
            cbSkip = 1024;
```

```
    if (!Read(abDummy, cbSkip))            //4
        goto Cleanup;

    cbRead += cbSkip;
}
```

Данный код используется для того, чтобы пропустить заголовок и перейти непосредственно к считыванию изображения. Стоит заметить, что если мы запишем в переменную bfOffBits, которая находиться в заголовке BMP-файла, какое-либо отрицательное значение(т.е. bfOffBits > 0x80000000+cbRead), то после присваивания в строке 2 имеем cbSkip < 0(т.е. cbSkip >0x80000000), а поскольку переменная cbSkip – signed integer, ее значение трактуется как отрицательное число. Далее идет проверка в строке 3, но она с успехом проходится, поскольку cbSkip < 0 < 1024 . Происходит вызов Read(abDummy, cbSkip). Осталось узнать только одно : как трактует cbSkip функция Read, как знаковое целое или как без знаковое? Для этого возьмем BMP–файл и по смещению 0xA запишем в него какое-либо значение, превосходящее 0x80000000, а затем попробуем открыть его в IE. Результат зависит от размера BMP-файла. Исследования показывают, что самые интересные вещи начинаются, когда файл имеет размер около 2500 байт. В этом случае IE завершает работу с ошибкой Access Violation, отсюда возникает предположение, что данная ошибка вызвана перезаписью адреса возврата. После просмотра содержимого регистров (eip и ebp) оказывается, что произошла типичная ошибка переполнения буфера. После дальнейших исследований оказывается, что функция Read считывает из файла только до тех пор пока не закончиться файл, а не число байт которое ей было реально передано. Это означает, что при вызове Read(abDummy, cbSkip) файл будет просто прочитан до конца, а потом, поскольку возникнет ошибка чтения из файла, функция Read вернет управление вызывающей функции и будет выполнена команда goto Cleanup. Определим, какое конкретно место в стеке надо перезаписывать (ведь из всех 2400 байт BMP-файла только 4 будут использоваться как адрес возврата). Сделать это можно просто заполнив файл разными значениями, и отследив, какое именно из них попадет в eip. Оказывается, это значение находится по смещению 0x8B6 от начала файла. Итак, мы можем передать управление по любому адресу в пространстве IE.
Если воспользоваться программой StackTrace то можно заметить, что esp после выполнения команды ret будет указывать на байт в буфере по адресу 0x8BE. Значит, именно начиная с этого адреса надо разместить в bmp файле наш зловредный код, поскольку тогда ему будет легко передать управление, конечно, возникает вопрос как это сделать? А сделать это очень просто: надо найти в коде какой-нибудь из системных библиотек используемых IE(KERNEL32.DLL и USER32.DLL) последовательность 0xFFE4 (jmp esp) или 0xFFD4 (call esp) и передать туда управление. И еще одно обычно эти системные библиотеки грузятся по фиксированному адресу, но этот адрес может меняться от сервиспака к сервиспаку. Таким образом, даже при открытии BMP файла в ваш компьютер может проникнуть опасный вирус!!!

Литература:

[1] “Basic integer overflows” by blexim, Phrack Journal Volume 0x0b, Issue 0x3c
[2] “Blended attack exploits, vulnerabilities and buffer-overflow techniques in computer viruses” Eric Chien and Péter Ször
[3] “Использование уязвимости в MSHTML.DLL для выполнения произвольного кода” http://bugtraq.ru/library/security/mshtml.html, автор - ch00k.