

чались либо слишком часто, либо слишком редко в подаваемой последовательности бинарных данных. На данный момент ведется дальнейшая оптимизация алгоритма работы с шаблонами, с целью их оптимизации. Под оптимизацией будет пониматься выделение минимального количества шаблонов из множества бинарных векторов заданной мощности, которые бы однозначно характеризовали данный образ.

Список литературы:

1. Круглов В.В., Борисов В.В. Искусственные нейронные сети. Теория и практика. – 1-е. – М.: Горячая линия – Телеком, 2001. – 20 с.
2. Хайкин С. Нейронные сети: полный курс = Neural Networks: A Comprehensive Foundation. – 2-е. – М.: «Вильямс», 2006. – 958 с.

ВИДЫ ПРОГРАММНЫХ УЯЗВИМОСТЕЙ

© Гончаров М.О.*

Московский государственный университет леса, г. Мытищи

В данной статье рассмотрены основные виды программных уязвимостей. В статье также приведены рекомендации, следуя которым, можно избежать возникновения уязвимых конструкций в программах. Все примеры приведены на языке программирования Си. Тестирование примеров производилось в среде Cygwin.

Переполнение буфера

Проблема переполнения буфера давно известна программистам, использующим низкоуровневые языки. Проблема заключается в записи введенных пользователем данных за границей выделенной памяти. Последствия переполнения буфера варьируются от обычного программного сбоя вплоть до полного захвата системы злоумышленником (такое возможно в случае, если уязвимая программа выполняется с повышенными привилегиями).

Рассмотрим фрагмент кода, содержащий уязвимость переполнения буфера:

```
void segfault(char* input) {  
    char buf[4];  
    strcpy(buf, input);  
    printf("%s\n", buf);  
}
```

Единственным аргументом функции `segfault` является строка с пользовательским вводом `input`. Эта строка копируется в переменную `buf`, под ко-

* Магистрант кафедры Вычислительной техники.

торию выделено 4 байта памяти, однако, никакой проверки на несоответствие длины пользовательского ввода объему выделенной памяти не производится. Далее, строка выводится на экран. Посмотрим, что из этого выйдет:

```
$ ./a.out segfault_testing
Segmentation fault (core dumped)
```

Чтобы устранить потенциальную уязвимость, необходимо внести в функцию `segfault` следующие изменения:

```
#define BUF_MEM 4
...
void segfault(char* input) {
    char buf[BUF_MEM];
    if (strlen(input) < BUF_MEM) error(OUT_OF_MEM);
    strcpy(buf, input);
    printf("%s\n", buf);
}
```

В функцию добавлен условный оператор, проверяющий длину пользовательского ввода, и, если длина введенной строки больше возможной длины, установленной программистом, вызывается функция `error` с кодом ошибки `OUT_OF_MEM`, которая в свою очередь обрабатывает данную исключительную ситуацию.

Уязвимость форматной строки

Уязвимость форматной строки даёт атакующему возможность записи по произвольному адресу в памяти, что тождественно полному контролю системы, при условии, что уязвимая программа выполняется с повышенными привилегиями. Рассмотрим следующий фрагмент кода:

```
int main(int argc, char* argv[]) {
    char string[1024];
    int value = 5;
    int *pv;

    pv = &value;

    if (argc > 1) {
        strcpy(string, argv[1]);
        printf(string);
    }

    printf("\nvalue at [%p] is %d\n", pv, value);
    return 0;
}
```

Погрешность состоит в отказе от использования спецификаторов формата, при выводе пользовательских данных функцией `printf`. Переменная

value и указатель на нее приведены здесь, чтобы показать каким образом злоумышленник может воспользоваться уязвимостью форматной строки. Итак, протестируем приложение:

```
$ ./a.out test
test
value at [0x22a888] is 5

$ ./a.out %x
61276bb0
value at [0x22a888] is 5

$ ./a.out $(python3 -c 'print("%08x|" * 7)')
612753c8|0022a894|00000003|0022ab16|0022a88c|0022a888
|00000005|
value at [0x22a888] is 5

$ ./a.out $(python3 -c 'print("%08x|" * 5)')%n
612753c8|0022a894|00000003|0022ab16|0022a88c|
value at [0x22a888] is 45
```

Если пользователь при вводе данных задает спецификатор формата, например, `%x`, функция `printf` воспринимает это как потребность вывести данные в шестнадцатеричной форме, а, так как, функции явно не задана переменная для вывода, она достаёт ее из стека (это продемонстрировано при втором запуске программы).

Пользовательские данные в третьем вызове аналогичны строке

```
%08x|%08x|%08x|%08x|%08x|%08x|%08x|,
```

что предполагает вывод семи значений из стека. Здесь видно, что шестое и седьмое значения соответствуют указателю `pv` и переменной `value`. Таким образом, в следующем вызове программы, используя также спецификатор формата `%n` (выводит количество символов, выведенных до настоящего момента), мы записываем по адресу `0x0022a888` (адрес переменной `value`) значение 45. Регулируя длину входных данных, фактически, по любому адресу можно записать любое возможное значение, что, как ранее было замечено, тождественно захвату системы.

Чтобы исправить данную погрешность, достаточно использовать спецификаторы формата при вызове функции `printf`:

```
printf("%s\n", string);
```

Следует также обращать внимание на другие функции форматного вывода.

Целочисленные переполнения

Целочисленные и арифметические переполнения характерны для языков программирования, в которых не реализованы целочисленные типы

переменного размера. Наибольшему риску подвержены такие языки как С и С++. Последствия целочисленного переполнения варьируются от обычной логической ошибки до повышения уровня привилегий атакующего.

Рассмотрим следующий фрагмент кода:

```
const unsigned int PASS_VALUE = 0xDeadBeef;
int main() {
    short value;
    printf("Enter your value: ");
    scanf("%x", &value);
    printf("Your value is: 0x%08x\n", value);
    if (value < PASS_VALUE)
        printf("|Access denied|\n");
    else printf ("|Access granted|\n");
}
```

Пользователю предлагается ввести определённое значение. Если это значение меньше некоторого фиксированного значения, выводится сообщение «В доступе отказано», иначе – «Доступ разрешен». Проблема состоит в том, что при сравнении пользовательских данных с фиксированным значением производится преобразование типа `short` в тип `unsigned int`, что может привести к неприятным последствиям, например:

```
$ ./a.out
Enter your value: 0xff
Your value is: 0x000000ff
|Access denied|

$ ./a.out
Enter your value: 0xffff
Your value is: 0xffffffff
|Access granted|
```

Оба значения `0xff` и `0xffff` (отрицательное число) меньше фиксированного `0xDeadBeef`, однако, из-за преобразования типа значение `0xffff` преобразуется в `0xffffffff`, которое больше фиксированного.

Чтобы избежать подобного рода уязвимостей в коде, следует обращать внимание на любые преобразования чисел в числа с большей разрядностью, на преобразования беззнаковых чисел в числа со знаками (и наоборот), результат арифметических операций, а также на понижающие преобразования.

Возможность внедрения команд

Разновидность этой проблемы возникает в передаче пользователем аргументов, доступных, как предполагалось, только в режиме командного интерпретатора. Рассмотрим следующую программу:

```
int main() {
    char exes[1024];
    char input[512];
    printf("Enter the file name: ");
    scanf("%s", &input);
    sprintf(exes, "cat %s", input);
    system(exes);
    return 0;
}
```

Программа, запрашивает у пользователя имя файла и выводит содержимое файла на экран. Здесь `exes` – строка, которая передается на выполнение командному интерпретатору. Посмотрим, что из этого выйдет:

```
$ ./a.out
Enter the file name: text
There is some text here.

$ ./a.out
Enter the file name: ;rm /etc/resolv.conf
```

В первом случае выводится содержимое файла `text`. Во втором злоумышленник использует специальный символ ‘;’, который воспринимается командным интерпретатором как потребность ввода следующей команды, и, таким образом, получает контроль над системой.

Вообще, не рекомендуется использовать обращения к командному интерпретатору в программе, но, если по каким-либо причинам существует такая потребность, следует тщательно проверять пользовательские данные и не допускать возможности запуска приложения с повышенными привилегиями.

Список литературы:

1. Эриксон Дж. Хакинг: искусство эксплойта: пер. с англ. – 2-е изд. – СПб.: Символ-Плюс, 2010 – 512 с.: ил.
2. Ховард М. 24 смертных греха компьютерной безопасности. Библиотека программиста. – СПб.: Питер, 2010 – 400 с.: ил.

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ ДЛЯ УЧЕТА И АНАЛИЗА ТРАВМАТИЗМА НА ПРОИЗВОДСТВЕ

© Студенникова Н.С.*

Орловский государственный аграрный университет, г. Орел

В статье обоснована необходимость использования информационных технологий для учета и анализа травматизма на производстве, а также рассмотрены существующие информационные системы.

* Старший научный сотрудник Всероссийского научно-исследовательского института социального развития села.