Тема: Виды программных уязвимостей (buffer overflow, format string, heap overflow).

:Intro:

Наличие программных уязвимостей на сегодняшний момент самая актуальная тема компьютерной безопасности, т.к. именно ошибки в ПО приводят к таким нежелательным последствиям, как проникновение в систему хакеров, компьютерных вирусов/червей и т.п. В данном эссе будет затронута некоторая часть данных уязвимостей.

:Buffer Overflow:

Данная уязвимость заключается в том, что происходит затирание данных в стеке.
Для этого нужно знать как устроен стек.

Допустим имеется программа:

```
Int Main(int argc, char **argv) {
        Int a;
        Char *p;
        Char abc[4*32];

        Strcpy(abc,argv[1]); // уязвимое место программы (копирование без контроля размера
                             // копируемых данных в массив с ограниченным размером)
        return 0;
}
```

Все временные данные программы хранятся в стеке, причем сохраняются данные при вызове функции main в следующем порядке:

```
|abc     | // 4*32 байта char abc[4*32]
|p       | // 4 байта char*p
|a       | // 4 байта int a
|tmpdata | // временные данные, используемые компилятором в своих целях (могут иметь длину 0-32 байта)
|ebp     | // 4 байта extended base pointer (базовый указатель при работе со стеком по косвенному адресу)
|eip     | // 4 байта extended instruction pointer (адрес возврата)
|argc    | // 4 байта int argc
|argv    | // 4 байта char **argv
```

Как видно, при копировании в abc данных, превышающих объем abc (>32*4) , можно затереть p, a и все что ниже..

Опасность данного вида уязвимости состоит в том, что при завершении функции происходит возврат на тот адрес, откуда эта функция была вызвана, чтобы продолжить исполнение программы далее… И, очевидно, что перезаписав eip на свой адрес мы произведем прыжок именно на этот адрес при выходе из функции. Задача состоит в том, чтобы указать правильный адрес, по которому будет располагаться shellcode (он же asmcode/bytecode… в общем набор машинных инструкций), который в свою очередь будет делать то, что мы захотим.

В данной ситуации атакующий может сохранить свой шеллкод в переменных abc+p+a+tmpdata и перезаписать eip на адрес в стеке, где начинается abc. Атакующий так же может использовать переменные окружения для хранения своего шеллкода, что гораздо проще.

Вычислить адрес в стеке можно разными методами: 1) посмотреть в дебаггере 2) вычислить самому, зная с какого адреса растет стек. В RedHat Linux’е он начинается с 0xc0000000 (т.е. с адреса 0xbfffffff будут заноситься данные), в FreeBsd – 0xbfc00000 и т.п.

:Format String:

Данная уязвимость связана с принципом работы функций семейства *printf*.
Рассмотрим данный уязвимый пример:

```
Int main(int argc, char **argv) {
        Printf(argv[1]); // уязвимое место
        Return 0;
}
```

при таком использовании функции printf в неявном виде (т.е. не указав как именно распечатать данные) printf сам произведет анализ данных и подберет для них подходящий спецификатор.
Но ничего не мешает подать функции строку “%x %x”. В данном случае printf возьмет 2 4х-байтных аргумента из стека и напечатает их, но т.к. функции printf не были переданы эти самые аргументы, то возьмутся 2 чужих аргумента из стека.

Риск данной уязвимости состоит в том, что атакующий может воспользоваться спецификатором %n , чтобы произвести запись по данному аргументу из стека кол-во байт, которые стоят перед ним. В результате чего, можно перезаписать любой адрес на своё значение (обычно перезаписываются адреса из GOT (Global Offset Table) таблицы каких-либо функций…в последствие чего при вызове данной функции, адрес которой был перезаписан, произойдет прыжок по перезаписанному адресу и выполнение тамошних инструкций).

:Heap Overflow:

Данный вид уязвимости очень схож с Buffer Overflow. Единственное различие состоит в том, что данные перезаписываются не в стеке, а в куче (Heap). Данный метод не очень прост в реализации, т.к. для него необходимо правильно сконструировать свою фальшивую кучу, которая сможет активизировать уязвимость при определенных условиях.

Куча в Linux начинается с адреса 0x080xxxxx.

Сама куча выглядит так:

```
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
```

Изначально (при стандартных условиях) 2 последних указателя не используются и данные начинают писаться с fd. Но создав ситуацию, когда вызывается функция, которая использует эти 2 указателя… можно добиться желаемого результата. Например, этой функцией может быть unlink, которая вызывается в том случае, если PREV_INUSE бит был активирован для prev_size’a.