сишные трюки
(13h выпуск)

крис касперски ака мыщъх, a.k.a. souriz, a.k.a. nezumi, no-email

в прошлом выпуске мы говорили о том как упросить отладку. сегодня же займемся обратной задачей, сосредоточив свое внимание на анти-отладочных приемах, препятствующих взлому программ. сразу же предупредим — все они системно-зависимы и выходят за рамки классического Си, а потому: применять их или не применять — супер неоднозначный вопрос, но если вы все же их применяете, то эта статья научит вас применять их _правильно_

трюк 1: самомодифицирующийся код

Упрощенная стратегия создания самомодифицирующегося кода выглядит приблизительно так: 1) получаем указатель на "подопытную" функцию; 2) вычисляем размер функции, вычитая из указателя на следующую функцию указатель на "подопытную", надеясь, что компилятор расположит функции в памяти в порядке их объявления; 3) выделяем блок памяти в стеке или куче; 4) копируем туда подопытную функцию; 5) издеваемся над ней как заблагорассудится, например, расшифровываем на лету.

Так или примерно так поступают тысячи программистов, удивляющихся почему программа падает при изменении ключей компиляции. А она и должна падать. Формально, язык Си позволяет получать указатель на функцию, но оставляет компилятору большую свободу и во многих случаях вместо указателя на функцию мы получаем указатель на переходник к ней вида JMP [MEM].

Существует только один _надежный_ способ достоверного определить адрес и размер функции — спросить об этом у нее самой! Идея заключается в следующем: при передаче "магических" аргументов, функция либо возвращает адрес своего начала/конца и тут же завершается, либо самостоятельно копирует себя в обозначенную локацию.

Для нейтрализации возможных побочных эффектов следует использовать квалификатор naked, поддерживаемый MS VC, при котором компилятор не вставляет в функцию никакой "отсебятины" и даже пролог, эпилог и инструкцию возврата мы должны точить самостоятельно. Простейший вариант реализации приведен ниже. Обработка аргументов для упрощения не показана. Функция foo просто возвращает адрес своего начала. Адрес конца определяется аналогичным образом.

```
__declspec(naked) foo(int x)
{
        __asm{
                call xxx                ; заталкиваем в стек адрес xxx
        xxx:
                pop eax                 ; адрес xxx -> EAX
                sub eax, 5              ; отнимаем длину CALL
                retn                    ; возвращаем адрес функции в регистре EAX
}
}
```

Листинг 1 достоверное определение адреса начала функции

Для обхода аппаратного DEP (по умолчанию задействованного только для системных приложений) необходимо выделить блок памяти функцией VirtualAlloc с флагами PAGE_EXECUTE_READWRITE или же изменить атрибуты уже выделенного блока вызовом VirtualProtect.

трюк 2: функции которые не возвращают управление

Ничто не аноит хакеров так, как функции, не возвращающие управления. При трассировке типа Step Over (т.е. без захода в функции), отладчик как бы "проваливается" внутрь очередного CALL'a, теряя управление над отлаживаемой программой. И чем чаще это происходит, тем больше матерится хакер. Конечно, пошаговая трассировка (Step Into) отрабатывает нормально, но это такой геморрой — от взлома не остановит, но по крайней мере доставит психологическое удовлетворение, что мы нагадили хакеру. Мелочь, а приятно!

Кажется, что для поставленной задачи идеальным образом подходит пресловутый оператор goto, но он действует только в пределах одной функции, а за ее пределы вылетать обламывается. Это связано с тем, что каждая функция имеет свой стековый фрейм, свои аргументы, etc и потому попытка перехода в середину "чуждой" функции в общем случае ведет к краху программы, даже если использовать различные ассемблерные извращения.

Структурные исключения позволяют передавать управление за пределы функции, однако, о них знают практически все хакеры и устроить им ловушку не получится. Только время зря потратим. К тому же, дальнобойность структурных исключений так же ограничена телом одной функции, а для ловли исключений за ее пределами приходится прибегать к ассемблерным вставкам и ручной установке фильтра посредством модификации указателя, хранящегося в ячейке FS:[0], что во-первых, не переносимо, а, во-вторых, на FS:[0] легко установить точку останова на доступ к данным и тогда отладчик будет отлавливать все исключения и утраты управления над отлаживаемой программой не произойдет.

ИМХО, самое лучшее, что можно сделать — подменить адрес возврата из функции. Для этого даже необязательно прибегать к ассемблеру и можно получить практически полностью переносимый вариант, что очень даже хорошо!

В упрощенном виде реализация выглядит так:

```
// функция вызываемая из foo
// путем подмены адреса возврата
__cdecl bar(int a, int b)
{
        // печатаем аргументы
        printf("%x, %x ***\n",a,b);

        // сейчас на вершине стека расположен
        // указатель на функцию exit, которой
        // и будет передано управление при
        // выходе из функции bar
}

__cdecl foo(int a, int b)
{
        // подменяем адрес возврата в main
        // на адрес функции bar, которой
        // и будет передано управление
        *(  (int*)&a - 1 ) = (int*)bar;
}

main()
{
        // вызываем foo
        foo((int)exit, 2);

        // сюда мы уже не вернемся
}
```

Листинг 2 подмена адреса возврата из функции без ассемблерных извращений

Идея основана на том, что в Си-соглашении аргументы функции заносятся в стек справа налево, то есть в момент вызова функции на вершине стека оказывается крайний левый аргумент, поверх которого забрасывается адрес возврата. Получив указатель на крайний левый аргумент (что можно сделать легальными средствами) и уменьшив его на величину машинного слова, мы локализуем положение адреса возврата, которое можно беспрепятственно модифицировать по своему усмотрению, в частности, заменить его адресом другой функции (в данном случае это функция bar).

Конструкция "((int*)(&a) - 1)" определяет местоположение адреса возврата, закладываясь на тот факт, что на 32-разрядных платформах указатель на функцию имеет размер равный 32 битам и потому может адресоваться как int*. Это единственный системно-зависимый участок и чтобы избываться от зависимости, необходимо вместо int* использовать указатели на функцию, однако, они имеют чуть более сложный синтаксис, загромождающий пример лишними круглыми скобками, что отнюдь не способствует его пониманию.

При запуске программы на экран вывалится "2, 12ffc0 ***" Как легко видеть, аргументы функции bar оказались сдвинутыми на одну позицию. Почему это произошло? А потому, что передача управления на bar осуществляется командой RET, которая работает как JMP, то есть совершает прыжок на bar, "забыв" положить в стек адрес возврата, роль которого приходится играть аргументу "a" функции foo, соответствующего аргументу "b" функции bar.

Следовательно, чтобы сохранить все аргументы, необходимо добавить к функции foo один фиктивный аргумент, расположенный слева.

трюк 3: необрабатываемые исключения

Про возможность передачи управления посредством структурных исключений мы уже говорили. Да и не только мы говорили. Об этом все говорят. Толку-то от этих исключений… Это даже не антиотладочный прием, а так… Однако, в win32 API есть одна довольно любопытная функция SetUnhandledExceptionFilter, устанавливающая фильтр для необрабатываемых исключений, получающий управление только в том случае, если программа находится не под отладкой. Причем, это не баг, а документированная фича. Если отладчик установлен, то все необрабатываемые исключения будет ловить он. До выполнения фильтра дело просто не дойдет!

Рассмотрим простой пример:

```
int a = 0;                                                              // делитель

// фильтр необработанных исключений
// (выполняется _только_ когда программа не под отладкой)
LONG WINAPI foo(struct _EXCEPTION_POINTERS *ExceptionInfo)
{
        // отмечаем свое присутствие на экране
        printf("***\n");

        // увеличиваем делитель
        a++;

        // пытаемся разделить еще раз
        return -1;
}

main()
{
        int *p=0; int b=0;

        // устанавливаем фильтр необработанных исключений
        SetUnhandledExceptionFilter(foo);

        // совершаем недопустимую операцию
        b = b /a;

        // отмечаем свое присутствие на экране
        printf("here!\n");
}
```

Листинг 3 программа, защищенная фильтром необработанных исключений

При нормальном выполнении он выбрасывает исключение, возникающее при делении на ноль, подхватываемое функцией-фильтром foo, которая выводит на экран "***", увеличивает делитель на единицу, повторяя операцию деления еще раз, в результате чего все работает нормально и ни одной мыши при выполнении программы не страдает.

А теперь запустим программу под OllyDebugger'ом или другим прикладным отладчиком. И что же?! Отладчик, спотыкнувшись об исключение, застывает, как кролик перед питоном. Пытаемая передать исключение в программу (в OllyDebugger'е это Shift-F7/F8/F9). Обычно это помогает, но только не сейчас! Отладчик зацикливается на исключении, отлавливая его вновь и вновь, а все потому что функция-фильтр, увеличивающая делитель, _не_ получает управления и увеличивать его становится некому.

Soft-Ice (будучи запущенным) так же всплывает ругаясь на исключение (даже если программа и не находится под отладкой), но по выходу из него, все продолжает работать без проблем, а потому, против Soft-Ice этот прием реально никак не действует, хотя если исключения будут сыпаться как из рога изобилия, то хакеру придется конкретно попотеть!