сишные трюки
(0xB выпуск)

крис касперски ака мыщъх, ака souriz, aka nezumi, aka elraton, no-email

сегодняшний выпуск трюков всецело посвящен хитроумным приемам программирования, затрудняющим дизассемблирование и отладку откомпилированной программы, то есть увеличивающих ее сопротивляемость взлому, причем все это — без всякого ассемблера и других шаманских ритуалов!

сладкая парочка setjump/longjump

Трассировка программы без захода в функции (step-over) — основной способ хакерской навигации, на пути которого разработчики защитных механизмов стремятся расположить всякие подводные рифы и другие неожиданные ловушки типа функций никогда не возвращающих управление в точку возврата, что приводит к потере контроля над отлаживаемой программой и сильно напрягает хакера, заставляя входить в _каждую_ функцию, а так же во все вызываемые ее функции. При большом уровне вложенности взлом растягивается на многие часы, дни, недели, месяцы, годы…

Самый простой (и легальный) способ "обхода" точки возврата основан на использовании стандартных Си-функций setjump/longjump. Первая — запоминает состояние стека функции в переменной типа jmp_buf (включая адрес текущей выполняющейся инструкции), вторая — передает управление по этому адресу вместе с аргументом типа int, что создает безграничный простор для всякого рода "трюкачества", наглядный пример которого приведен ниже:

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf stack_state;

A(){printf("func A()\n");longjmp(stack_state, -1 );}
B(){printf("func B()\n");longjmp(stack_state, -2 );}
C(){printf("func C()\n");longjmp(stack_state, -3 );}

main()
{
        int jmpret;
        // __asm{int 03}; // для отладки

        // запоминаем состояние стека
        jmpret = setjmp(stack_state);

        // выполняем C(), из которой мы возвращаемся в точку jmpret
        if (jmpret==-3) return 0;

        // выполняем C(), из которой мы возвращаемся в точку jmpret
        if (jmpret==-2) C();

        // выполняем B(), из которой мы возвращаемся в точку jmpret
        if (jmpret==-1) B();

        // выполняем A(), из которой мы возвращаемся в точку jmpret
        if (jmpret==00) A();

        // эта функция никогда не получает управление
        printf("good bye, world!\n");
}
```

Листинг 1 обход точки возврата через longjmp

Откомпилируем программу и убедимся, что она последовательно вызывает функции A(), B(), C(), после чего раскомментируем строку _asm{int 03}, откомпилируем еще раз и запустим полученный exe-файл под отладчиком OllyDbg (или любым другим), нажав <F9> (run) для достижения строки int 03h (см. рис. 1). Нанимаем трассировать программу по <F8> (step over) и… не доходя до строки "printf("good bye, world!\n");" и не успев выполнить функции A(),

отладчик неожиданно теряет контроль за подопытной программой, и она вырвавшись из-под трассировки, благополучно завершается по return 0, что OllyDbg и констатирует. Сказанное относится не только к OllyDbg, но так же к Soft-Ice и всем остальным отладчикам.

К сожалению, в дизассемблере типа IDA Pro ловушка становится слишком очевидной и установив точку останова на функцию setjmp, хакер без труда сможет отладить защищенную программу, разобрав защитный механизм на составные части выкинув из него все лишние детали.

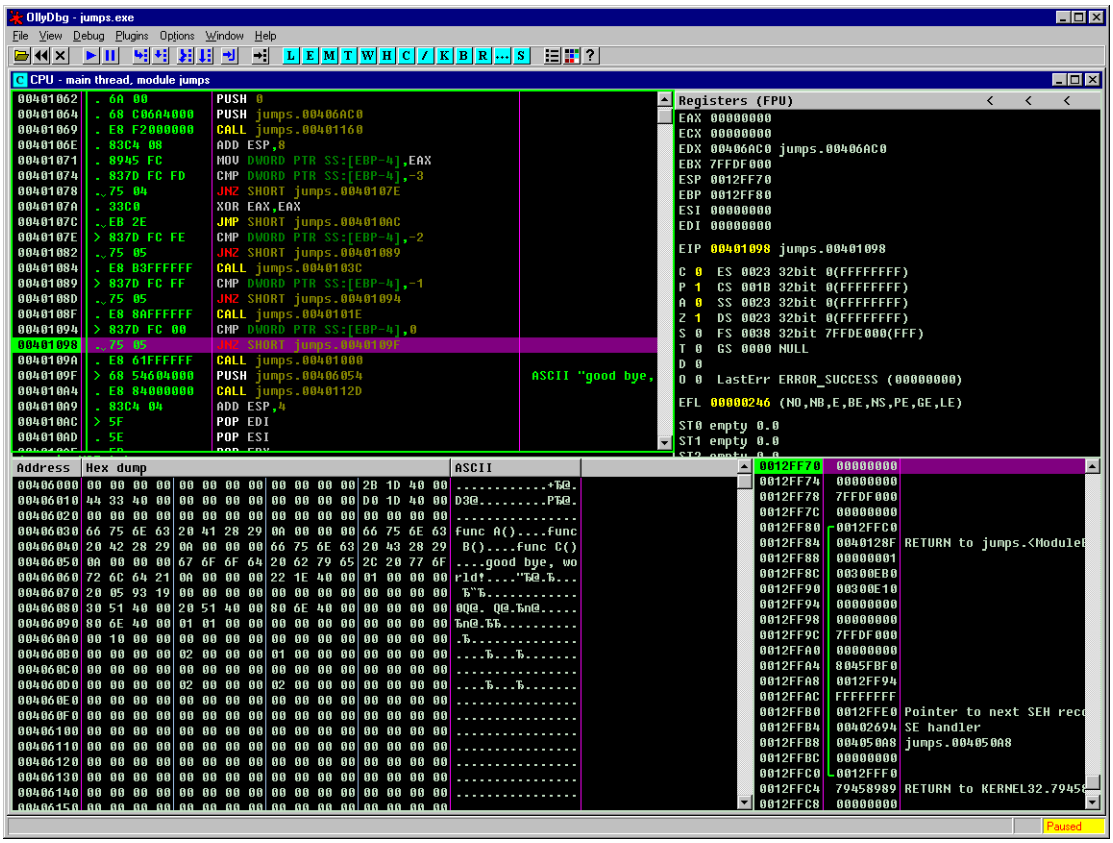

Рисунок 1 при входе в следующую функцию, отладчик безвозвратно теряет контроль над отлаживаемой программой

подмена адреса возврата

При step-over трассировке отладчики устанавливают программную (реже аппаратную) точку возврата за концом команды CALL func_A, куда func_A возвращает управление посредством оператора return, стягивающего со стека адрес возврата, положенный туда процессором перед вызовом func_A. Таким образом, чтобы вырваться из-под трассировки нам достаточно заменить подлинный адрес возврата на адрес какой-нибудь другой функции (назовем ее функцией func_B), куда и будет передано управления.

Проблема в том, что положение адреса возврата в стеке нельзя узнать штатными средствами языка Си, а если задействовать нелегальные средства, то это уже будет не трюк, а хак, то есть грязный прием программирования, работающих не на всех платформах и зависящий от компилятора. Тем не менее, кое-какие зацепки у нас есть. Мы знаем, что стек растет снизу вверх (т.е. из области старших адресов в область младших), так же мы знаем, что по Си-соглашению аргументы функции заносятся в стек справа налево, после чего туда заносится адрес возврата и между последним аргументом и адресом возвратом компилятор не имеет права класть ничего другого, в противном случае, функция просто не сможет найти свои аргументы, а поскольку программист имеет право использовать функции, откомпилированные разными компиляторами, то компилятору ничего не остается, кроме как следовать соглашениям.

Таким образом, нам надо просто получить адрес самого левого аргумента (что можно сделать оператором &) и, преобразовав его к указателю на машинное слово (на x86 составляющее 32 бита и совпадающее по размеру с указателем на int), уменьшить его на

единицу и… записать по данному адресу указатель на функцию func_B, куда и будет передано управление по завершению func_A. При этом случает помнить, что при выходе из функции func_B управление будет передано… обратно на саму функцию func_B! Почему? Да потому, что она вызвана "нечестным" способом и в стек не занесен адрес возврата. Тем не менее, func_B может спокойно вызывать остальные функции "честным" путем, ничем не рискуя.

Законченный пример приведен ниже:

```
// функция B(), которой функция A()
// скрытно передает управление
B(){printf("func B();\n");exit(0);}


// явно объявляем функцию как _cdecl,
// чтобы оптимизатор "случайно" не реализовал ее как fastcall
_cdecl A(int x)
{
        //  подмена адреса возврата
        *((((int*)&x)-1))=x;

        printf("func A();\n");
}

main()
{
        // __asm{int 03}; // для отладки

        // функция A() подменяет свой адрес возврата на B()
        A((int) B);

        // эта функция никогда не получает управление
        printf("good bye, world!\n");
}
```

Листинг 2 демонстрация вызова функции с подменой адреса возврата

Компилируем программу, убеждаемся что она работает, затем раскомментируем строку _asm{int 03}, перекомпилируем обратно и запускаем под отладчиком Microsoft Visual Studio Debugger (или любым другим). Нажимаем <F5> (run) и затем несколько раз <F10> (step over). Отладчик входит в функцию A(), но обратно уже не возвращается, поскольку отлаживаемая программа вырывается из лап трассировщика!

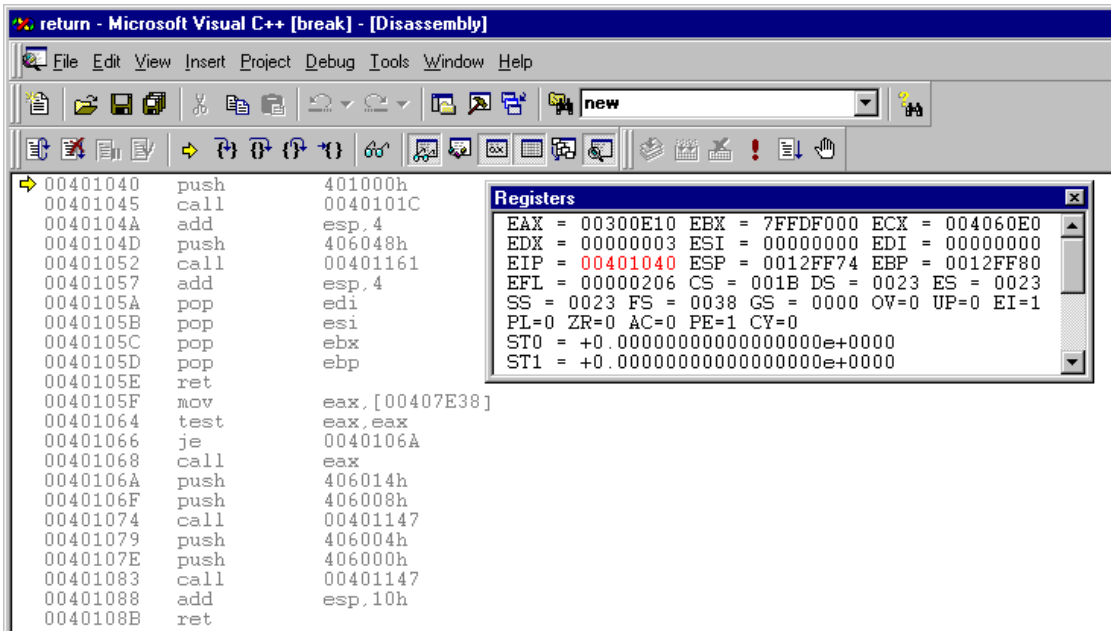

Рисунок 2 подмена адреса возврата вырывает отлаживаемую программу из лап трассировщика

маскировка указателей

Описанный выше прием хорошо успешно борется с отладчиками, но бессилен перед дизассемблерами, поскольку при первом же взгляде на вызов функции func_A, становится заметно (см. листинг 3), что ей в качестве аргумента передается адрес функции func_B. "Это же явно неспроста" — бормочет хакер себе под нос, устанавливая точку останова на func_B, о которую спотыкается защитный механизм в бессильной попытке освободится от гнета отладчика.

```
.text:0040103F        int     3       ; Trap to Debugger
.text:00401040        push    offset func_B
.text:00401045        call    func_A
.text:0040104A        add     esp, 4
.text:0040104D        push    offset aGoodByeWorld ; "good bye, world!\n"
.text:00401052        call    _printf
```

Листинг 3 так выглядит откомпилированный листинг 2 в дизассемблере

Проблема решается легкой ретушью защитного механизма. Достаточно слегка зашифровать указатель на func_B, чтобы он не так бросался в глаза и… хакер ни за что не догадается где зарыта собака, пока не проанализирует весь код целиком, а анализ всего кода программы — дело непростое и отнимающее уйму времени.

Самое просто, что только можно сделать — перед передачей указателя на func_B наложить на него "магическое слово" операцией XOR, а перед подменой адреса возврата наложить XOR еще раз, получая исходный указатель:

```
#define MAGIC_WORLD 0x666999
…
        *((((int*)&x)-1))=x ^ MAGIC_WORLD;
…
        A(((int) B) ^ MAGIC_WORLD);
```

Листинг 4 доработанный вариант листинга 2, маскирующий указатель на func_B

Компилируем программу (не забыв задействовать оптимизацию, чтобы компилятор зашифровал указатель еще на стадии компиляции; в Microsoft Visual C++ это достигается путем указания ключа /Ox, в других компиляторах это может быть ключ -O2 или что-то другое, описанное в справочном руководстве).

```
text:00401040 _main   proc near
text:00401040         push    267999h
text:00401045         call    sub_401020
text:0040104A         push    offset aGoodByeWorld ; "good bye, world!\n"
text:0040104F         call    _printf
text:00401054         add     esp, 8
text:00401057         retn
text:00401057 _main   endp
```

Листинг 5 доработанный листинг 2 в дизассемблере

Теперь указатель на функцию func_B превратился в безликую константу 267999h, в которой даже самые проницательные хакеры навряд ли смогут распознать указатель! Кстати говоря, описанный трюк полезен не только в контексте подмены адреса возврата, но применим ко всем видам указателям — как на функции, так и на данные, в том числе и текстовые строки, перекрестные ссылки к которым автоматически генерируются IDA Pro и другими дизассемблерами, а по перекрестным ссылкам найти код, выводящий сообщение о неверном ключе регистрации или истечении демонстрационного строка использования — минутное дело! Если, конечно, указатели не будут зашифрованы магическим словом!