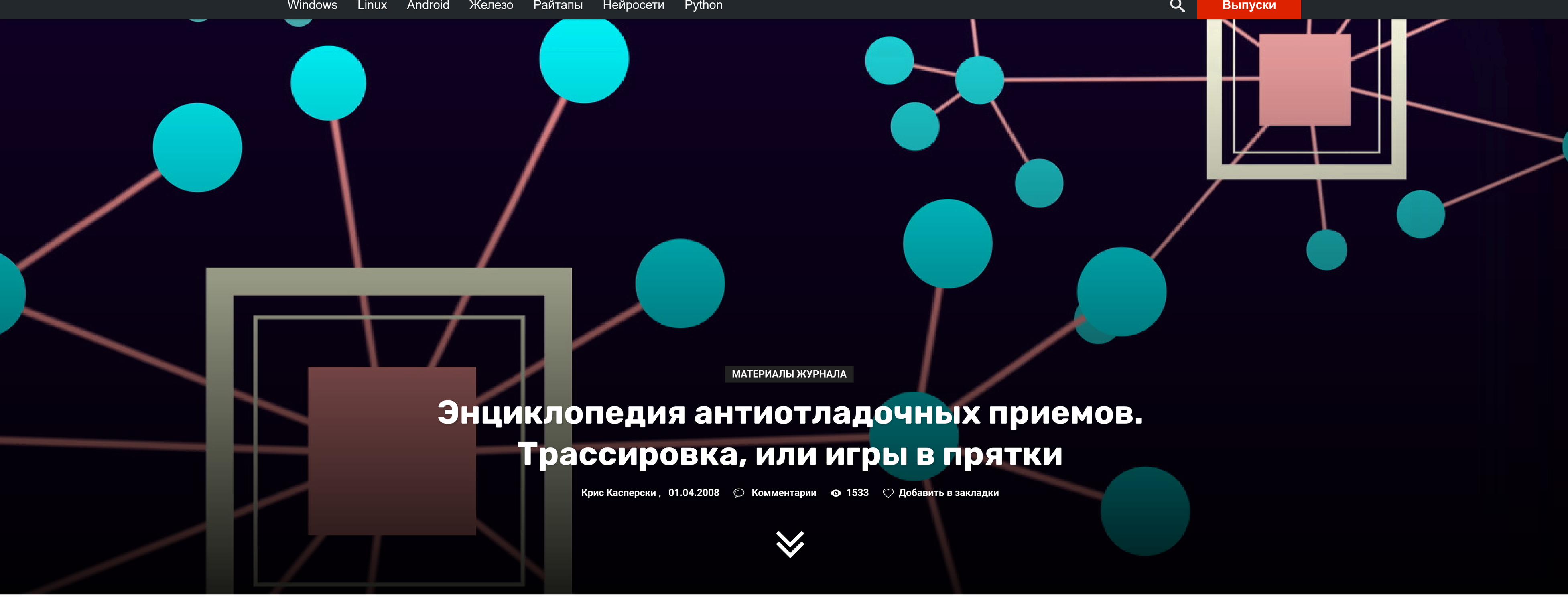
Временная скидка 60% на годовую подписку!



МАТЕРИАЛЫ ЖУРНАЛА

Энциклопедия антиотладочных приемов. Трассировка, или игры в прятки

Крис Касперски , 01.04.2008 Комментарии 1533 Добавить в закладки

Содержание статьи

01. Трассировка в процессорах x86
02. Эксперимент #1 — «чистый» PUSHFD
03. Эксперимент #2 — игры с префиксами
04. Эксперимент #3 — прерывания в маске
05. Антиантиотладка

Обзор антиотладочных приемов мы начнем с базовых понятий, фундаментальным из которых является трассировка (или пошаговое исполнение кода). Сначала мы узнаем, зачем нужна трассировка, как и в каких целях она используется отладчиками, по каким признакам защитный код может определить, что его трассируют, и какие примочки к отладчикам позволят хакеру избежать расправы.

Уже давно никто не трассирует программы от начала и до конца — слишком утомительно и непродуктивно. Однако не стоит полностью списывать трассировку со счетов, она и сейчас живее всех живых!

Запутанные участки кода, ответственные за проверку серийного номера, ключевого файла или расшифровку программы, довольно часто прогоняются отладчиком в пошаговом режиме. Кроме того, отладчик может «негласно» задействовать трассировку для выполнения некоторых операций. В частности, в OllyDbg установка точки останова на команду и/или диапазон EIP-адресов реализуется как раз через трассировку. Ее же используют многие плагины, например популярный FindString, выполняющий поиск заданной строки в регистрах (трактует их как указатели). Распаковщики упакованных файлов (особенно универсальные) активно используют трассировку для освобождения от упаковщика и восстановления оригинальной точки входа в программу (Original Entry Point, или, сокращенно, OEP).

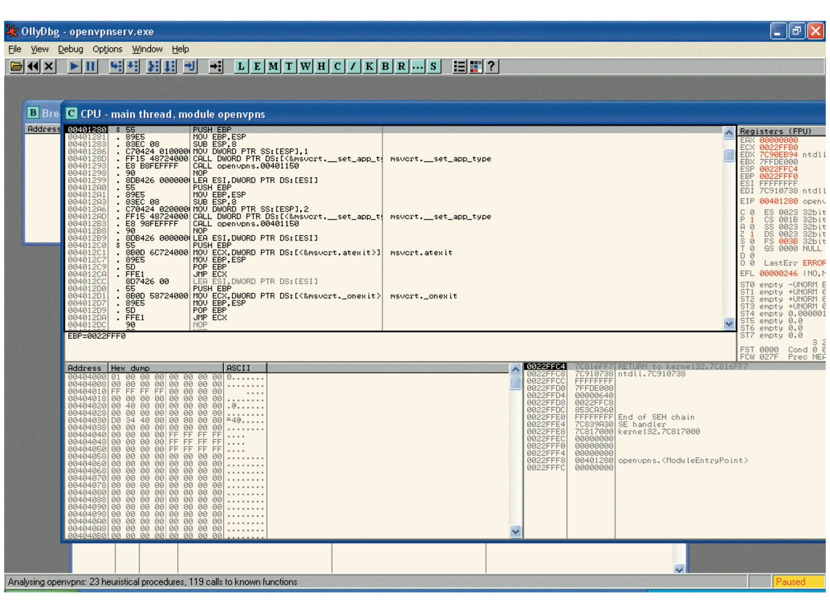
Защита, умело препятствующая трассировке, затрудняет взлом программы, хотя и не делает его невозможным, поскольку на каждый антиотладочный болт с хитрой резьбой уже давно придуман свой антиантиотладочный ключ.

ТРАССИРОВКА В ПРОЦЕССОРАХ X86

Если TF-флаг, хранящийся в регистре EFLSGS (и гнездящийся в восьмом бите, считая от нуля), взведен, то после исполнения каждой команды процессор генерирует прерывание `INT 01h` или `EXCEPTION_SINGLE_STEP` (80000004h) — как его «обозвали» разработчики Windows. Исключение составляют команды, модифицирующие регистр SS (селектор стека) и маскирующие прерывание на выполнение следующей команды. На этот шаг разработчики процессоров пошли потому, что в коде часто встречаются конструкции вида `MOV SS, new_ss/MOV ESP, new_ESP`. Легко сообразить, что, если прерывание произойдет после того, как новый селектор стека уже обозначен, а указатель вершины стека еще не инициализирован, мы получим неопределенное поведение системы, ведущее к краху (а ведь существует команда LSS, одним махом загружающая и SS, и ESP, но она не относится к числу самых популярных).

Простейший способ обнаружения трассировки состоит в чтении регистра флагов (EFLAGS) и проверке состояния бита TF. Если он не равен нулю — нас кто-то злостно трассирует. С прикладного уровня прочитать содержимое регистра флагов можно самыми разными способами: командой PUSHFD, заталкивающей флаги в стек, генерацией исключения (при которой SEH-обработчику передается контекст потока вместе со всеми регистрами, включая регистр флагов); наконец, контекст можно получить API-функцией `GetThreadContext`.

Сегодня мы будем говорить лишь о первом способе — команде PUSHFD. При кажущейся незамысловатости она скрывает целый пласт хитростей, известных далеко не всякому хакеру.



ЭКСПЕРИМЕНТ #1 — «ЧИСТЫЙ» PUSHFD

Напишем несложную программку, заталкивающую в стек регистр флагов через PUSHFD и тут же выталкивающую ее обратно в EAX для тестирования значения бита TF.

Простейшая программа TF-0x0-simple.c для обнаружения трассировки через PUSHFD

```
char yes[]="debugger is detected :-)";
char noo[]="debugger is not detected";

nezumi()
{
    char *p=noo;              // Презумпция невиновности is on ;-)
    __asm
    {
        ; int 03              ; Для отладки
        pushfd                ; сохраняем флаги в стеке, включая и TF
        pop eax               ; Выталкиваем сохраненные флаги в eax
        and eax, 100h             ; Проверяем состояние TF-бита
        jz not_under_dbg          ; Если TF не взведен, нас не трассируют…
        mov [p], offset yes       ; …ну или мы не смогли это обнаружить ;)
    not_under_dbg:
    }

    MessageBox(0, p, p, MB_OK);
}
```

Откомпилируем ее следующим образом.

```
cl.exe /c /Ox  /Os /G6 TF-0x0-simple.c
link.exe TF-0x0-simple.obj /ENTRY:nezumi /MERGE:.rdata=.text
      /ALIGN:16 /DRIVER /FIXED /SUBSYSTEM:CONSOLE KERNEL32.LIB USER32.lib
```

Все это шаманство потребовалось:

- чтобы убить стартовый код и начать программу с интересующей нас функции `nezumi()`;
- чтобы сократить размер программы, равный в данном случае 768 байтам.

Не обращая внимания на ругательство линкера warning LNK4078: multiple ".text" sections found with different attributes (40000040), запустим программу. Убедимся, что она честно говорит: debugger is not detected. А теперь загрузим ее в MS VC dbg и будем трассировать (клавиша F11), пока не достигнем первого call'а (им будет MessageBox). Ага, debugger is detected! Цель достигнута!

Теперь испытаем cdb.exe из набора Debugging Tools. Поскольку он органически не умеет стопиться на OEP, раскомментируем `int 03` и перекомпилируем программу: загрузим ее в отладчик, указав имя файла в командной строке. Первый раз отладчик всплывает в ntdll!DbgBreakPoint по `int 03h`. Это всплытие нам совершенно не интересно, так что пишем g для продолжения выполнения программы и попадаем на «наш» собственный `int 03h`, стоящий в начале `nezumi()`.

Последовательно отдавая команду t, трассируем функцию до достижения call'а, а потом говорим g. Отладчик не обнаружен! Как так? А очень просто — CDB отслеживает команду PUSHFD и эмулирует ее выполнение, «вычищая» TF-бит из стека. Аналогичным образом себя ведут Soft-Ice, Syser, OllyDbg и многие другие «правильные» отладчики. А вот IDA и GDB «честно» показывают TF-бит, как он есть, чем и обнаруживают свое присутствие.

ЭКСПЕРИМЕНТ #2 — ИГРЫ С ПРЕФИКСАМИ

В лексиконе x86, помимо самостоятельных команд, есть так называемые префиксы (prefix). Например, префикс повторения (REPE/PEPNE), префикс перекрытия сегмента (CS:, DS:, SS:, ES:, FS:, GS:), префикс изменения разрядности (с опкодом 66h). Префиксы работают только со своим набором команд; в частности, префикс повторения применяется лишь совместно со строковыми инструкциями (MOVSD, LODSD, STOSD). На остальные команды он никак не воздействует (разве что увеличивает время их декодирования), а потому PUSHFD и REPE:PUSHFD — синонимы.

Умный отладчик должен учитывать, что перед командой PUSHFD может стоять один или несколько «мусорных» префиксов, и автоматически отбрасывать их. Но это в теории. Добавим `REPE` перед `PUSHFD` в нашу программу и перекомпилируем ее, переименовав в TF-0x1-prefix.c.

Такие отладчики, как CDB, Soft-Ice и Syser, автоматически отбрасывают префиксы, препятствуя их обнаружению. MS VC, IDA и GDB как обнаруживались, так и обнаруживаются, а OllyDbg (даже в новой версии со всеми плагинами) палится даже на банальном REPE, не говоря уже о сочетании нескольких префиксов!

ЭКСПЕРИМЕНТ #3 — ПРЕРЫВАНИЯ В МАСКЕ

Немного видоизменим нашу тестовую программу, добавив перед инструкцией PUSHFD пару команд `MOV AX,SS/MOV SS,AX`. И хотя реальной модификации регистра SS при этом не происходит, процессор все равно маскирует трассировочное прерывание на время команды, следующей на `MOV SS,AX`, которой и является `PUSHFD`.

Ловля TF-бита через маскирование трассировочного прерывания

```
nezumi()
{
    char *p=noo;                  // Презумпция невиновности is on ;-)
    __asm
    {
        int 03                ; Для отладки
        mov ax,ss             ; маскируем трассировочное прерывание...
        mov ss,ax             ; ...на время выполнения команды PUSHFD
        pushfd                ; Сохраняем флаги в стеке, включая и TF
        pop eax               ; Выталкиваем сохраненные флаги в eax
        and eax,100h              ; Проверяем состояние TF-бита
        jz not_under_dbg          ; Если TF не взведен, нас не трассируют
        mov [p],offset yes
    not_under_dbg:
    }
    MessageBox(0, p, p, MB_OK);
}
```

Откомпилируем и посмотрим, как справятся отладчики. Вот мы доходим до `MOV SS,AX`, нажимаем F7 (Step into) и перескакиваем через PUSHFD, позволяя ей сохранить в стеке истинное состояние TF-бита, что немедленно приводит к обнаружению отладчика.

И MS VC, и CDB, и Soft-Ice, и OllyDbg, и IDA, и GDB — все ловятся на этот крючок. Syser (вплоть до версии 1.95.1900.0894) тоже ловился, пока я не отписал его разработчикам и они не пофиксили этот баг. В результате Syser стал единственным (на сегодняшний день) отладчиком, распознающим инструкции, модифицирующие SS. Если за ними следует PUSHFD, включается специальный «эмулятор», который подсовывает программе сброшенный TF-бит.

	MS VC	CDB	SOFT-ICE	SoftICE +IceExt	Syser	OllyDbg	OllyDbg +Phanom	IDA	CDB
PUSHFD	+	-	-	-	-	-	-	+	+
XX: PUSHFD	+	-	-	-	-	+	+	+	+
MOV	+	+	+	+	-	+	+	+	+

Сводная таблица с результатами экспериментов (+ — обнаруживается защитой, - — не обнаруживается). Как видно, отладчик Syser лидирует!

АНТИАНТИОТЛАДКА

Пользователям Syser'a хорошо! Им вообще ни о чем заботиться не нужно! А что делать приверженцам остальных отладчиков? При «ручной» трассировке программы, обнаружив PUSHFD, достаточно прекратить трассировку и, установив точку останова за ее концом, сказать отладчику Run или Go. Тогда фрагмент кода прогонят без трассировки, что (естественно) не позволит обнаружить трассировку, поскольку ее нет.

При автоматизированных прогонах в OllyDbg можно поставить точки останова на все команды, модифицирующие SS. Тем самым заставляя его всплывать и передавая бразды правления в наши лапы (чтобы разрулить ситуацию по описанной методике). Проблема в том, что таких команд очень много. Это не только `MOV SS`, 16-bit Reg/Mem и `POP SS`, но еще и `MOV, SS/POP SS` плюс различные префиксы. В частности, MOV SS, EAX выполняется точно так же, как и `MOV SS,AX`, но имеет другой опкод. Необходимо это учитывать при составлении списка команд, на которые мы брякаемся.

Знаешь ли ты?

1. О трассировке ветвлений

Процессоры Pentium умеют трассировать ветвления (условные/безусловные переходы и вызовы функций). Для этого нужно взять MSR-регистр `MSR_DEBUGCTLA` и взвести в нем бит BTF (single-step on branches). Тогда при взведенном TF-бите в регистре флагов EFLAGS трассировочное прерывание будет генерироваться не после каждой машинной команды, а лишь на инструкциях ветвления. Это очень полезно для разбивки программы на функциональные блоки (например, можно написать real-time-трассер, сравнивающий прогоны ветвлений программы до и после истечения испытательного срока, что позволит легко найти тот «заветный» jxx, который нужно захачить).

С другой стороны, если защита взведет BTF-бит, то все известные мне отладчики не смогут нормально работать, поскольку не проверяют его состояния при трассировке. Запись MSR-регистров выполняется привилегированной командой WRMSR, и при попытке ее исполнения на прикладном уровне процессор генерирует исключение. Однако писать собственный драйвер для игр с BTF-битом совершенно не обязательно. Можно воспользоваться недокументированной native API функцией `NtSystemDebugControl()`, экспортируемой из NTDLL.DLL, пример вызова которой можно найти на сайте openrce.org. Для этого необходимо обладать правами администратора.

Замечу, что в последних пакетах обновления для Server 2003 и XP возможности этой функции были существенно урезаны. По-видимому, политика урезания продолжится, так что когда-нибудь без драйвера будет не обойтись.

2. Что случилось с точками останова?

Маскирование прерываний после команд, модифицирующих содержимое регистра SS, распространяется также и на отладочные прерывания. В частности, генерируемые аппаратными точками останова по исполнению, которые установлены на команду, следующую за инструкцией и модифицирующую регистр SS. Они, согласно документации от Intel и AMD, не срабатывают, и отладчик их мирно пропускает. Это не баг в отладчике — это особенность x86-процессоров.

Программные точки останова (представляющие собой опкод CCh) и аппаратные точки останова на чтение/запись данных продолжают работать, как ни в чем не бывало.

3. Как еще можно маскировать прерывания?

Существует два основных способа анализа программ без исходных текстов: статический (дизассемблирование) и динамический (отладка). Дизассемблирование очень плохо справляется с самомодифицирующимся и самогенерируемым кодом. Действительно, защита может затолкать в стек кучу непонятных «циферок», перемешав их самым причудливым образом, и передать туда управление. А что у нас там? Дизассемблер молчит как партизан, хоть пытай его, хоть не пытай! Такой код обычно смотрят под отладчиком.

Представим себе код, который расположен в стеке и помещает поверх себя несколько машинных команд. Первой из них идет команда модификации регистра SS, затирающая предыдущее содержимое, на которое указывает регистр EIP. Благодаря маскированию прерываний она «проскакивает» следующую команду, которая, в свою очередь, может затирать предыдущую. Как следствие — все отладчики, за исключением Syser'a, отобразят лишь часть команд, а остальные команды будут затерты прежде, чем отладчик получит управление.

Один из примеров реализации трюка приведен в программе TF-0x3-crackme.c, которую и предлагаю тебе взломать (благо исходные тексты снабжены подробными комментариями, так что задача будет по зубам даже новичкам).

Оцени статью:

Крис Касперски
Известный российский хакер. Легенда][, ex-редактор ВЗЛОМа. Также известен под псевдонимами мыщъх, nezumi (яп. 鼠, мышь), n2k, elraton, souriz, tikus, muss, farah, jardon, KPNC.

Теги: assembler Взлом Выбор редактора Отладка Подборка Статьи Трассировка

Подпишись на нас в Telegram!
Только важные новости и лучшие статьи
Подписаться

Подписаться
Вы вошли как spektr_07 | Выйти
Поделитесь своим мнением
0 КОММЕНТАРИЕВ

Вопросы по материалам и подписке: support@glc.ru
Отдел рекламы и спецпроектов: yakovleva.a@glc.ru
Контент 18+
Сайт защищен Qrator — самой заботливой защитой от DDoS в мире
Подписка для физлиц
Подписка для юрлиц
Реклама на «Хакере»
Контакты