



ЗАО «ПОЗИТИВ ТЕХНОЛОДЖИЗ»
107061 / МОСКВА / ПРЕОБРАЖЕНСКАЯ ПЛОЩАДЬ / Д.8
ТЕЛ.: +7 (495) 744 01 44 / ФАКС: +7 (495) 744 01 87 / PT@PTSECURITY.RU
WWW.PTSECURITY.RU / WWW.MAXPATROL.RU / WWW.SECURITYLAB.RU

Как обезвредить Windows 8.1 Kernel Patch Protection (PatchGuard)

Артём Шишкин, Марк Ермолов

Периодически, как правило во вторую среду месяца, можно услышать истории о том, что Windows после очередного обновления перестает загружаться, показывая синий экран смерти. В большинстве случаев причиной такой ситуации оказывается либо руткит, либо специфичное системное ПО, фривольно обращающееся со внутренними структурами ОС. Винят, конечно, все равно обновление, ведь «до него все работало». С таким отношением не удивительно, что «Майкрософт» не поощряет использование всего, что не документировано. В какой-то момент, а именно с релизом Windows Server 2003, MS заняла более активную позицию в вопросе борьбы с чудо-поделками сторонних разработчиков. Тогда появился механизм защиты целостности ядра — kernel patch protection, более известный как PatchGuard.

С самого начала он не позиционировался как механизм защиты от руткитов, поскольку руткиты работают в ядре с теми же привилегиями, а следовательно, PatchGuard может быть обезврежен. Это скорее фильтр, отсекающий ленивых разработчиков руткитов.

Что охраняет PatchGuard

Самым популярным местом модифицирования ядра была таблица системных вызовов. При помощи модификации указателей на функции системных вызовов можно было легко их перехватывать, фильтровать, логировать и т. п. Причем этот патч был популярен как для руткитов, так и для антивирусного ПО. Другие интересные для патча объекты — таблицы дескрипторов (GDT, IDT). Путем модифицирования глобальной таблицы дескрипторов можно было изменять атрибуты сегментов, создавая бекдоры для кода, а через таблицу дескрипторов прерываний можно было перехватывать... прерывания! Продвинутые же парни

«сплайсили» непосредственно функции ядра.

Соответственно, первая версия PatchGuard защищала:

- таблицы системных вызовов (SST),
- глобальную таблицу дескрипторов (GDT),
- таблицу дескрипторов прерываний (IDT),
- образ ядра,
- ядерные стеки.

С развитием NT перерабатывалось множество компонентов ядра, в том числе и PatchGuard. На текущий момент уже сложно перечислить все, что защищается с его помощью:

- множество системных образов, не только образ ядра (nt, hal, WerLiveKernelApi, tm, clfs, pshed, kdcom, bootvid, ci, msrpc, ndis, ntfs, tcpip, fltmgr),
- критически важные структуры данных ядра (например, список процессов),
- набор MSR (например, model specific регистр IA32_LSTAR),
- KdpStub — процедура отладчика, получающая управление после исключений.

Как охраняет PatchGuard

Стоит отметить, что PatchGuard активно использует новую реализацию обработки исключений, введенной в x64-версиях Windows. Используется он как для обфускации самого PatchGuard, так и для проверки целостности защищаемых образов.

В предыдущих версиях Windows обработчик исключений использовал структуры данных прямо на стеке, что даже позволяло обходить stack cookies при эксплуатации уязвимостей. Основное изменение заключается в хранении специальной таблицы внутри исполняемого образа с записями для каждой отдельной его функции.

```
typedef struct _IMAGE_RUNTIME_FUNCTION_ENTRY {  
    uint32_t BeginAddress;           // Начало функции  
    uint32_t EndAddress;           // Конец функции  
    union {  
        uint32_t UnwindInfoAddress; // Указатель на данные, используемые д  
    }  
    uint32_t UnwindData;           // обработки исключений
```

```
};  
} _IMAGE_RUNTIME_FUNCTION_ENTRY, *_PIMAGE_RUNTIME_FUNCTION_ENTRY;
```

За счет того, что адрес начала и конца любой функции можно получить прямо в рантайме, задача подсчета контрольной суммы отдельно взятой функции становится тривиальной. Для сравнения — в x86-версиях контроль целостности образов невозможен из-за того, что непонятно, как определить границы отдельной функции, а образ целиком (или даже отдельные его секции) накрывать контрольной суммой нельзя, поскольку в том же ядре присутствуют функции, которые патчатся самим ядром на лету.

При загрузке ОС PatchGuard создает от 1 до 4 контекстов — структур данных, в которых хранятся копии используемых им функций, контрольные суммы защищаемых структур и ключи шифрования самого контекста. Эти контексты хранятся в неподкачиваемом пуле в зашифрованном виде. О проверке контекстов поговорим чуть позже.

Инициализируются контексты PatchGuard в фазе 1 загрузки ОС. Функция, непосредственно создающая контекст, не имеет публичного символа (будем называть ее `KiInitializePatchGuardContext`), но найти ее можно внутри функции `KiFilterFiberContext`. Мы нашли два места, в котором возможно создание контекста PatchGuard:

```
... -(call)-> Phase1InitializationDiscard -(call)-> KeInitAmd64SpecificState -(exception)-> KiFilterFiberContext
```

```
... -(call)-> Phase1InitializationDiscard -(call)-> sub_14071815C -(call)-> ExpLicenseWatchInitWorker -(call)-> KiFilterFiberContext
```

Первый вариант всегда создает хотя бы один контекст, в то время как второй только в 4% случаев. Первый вариант примечателен и тем, что вызывает функцию `KiFilterFiberContext` неявно, а именно через «вброс» исключения.

```
__int64 KeInitAmd64SpecificState()  
{  
    signed int v0; // edx@2  
    __int64 result; // rax@2  
  
    // В безопасном режиме PatchGuard не работает  
    if ( !InitSafeBootMode )  
    {
```

```

    v0 = __ROR4__(KdPitchDebugger | KdDebuggerNotPresent, 1);
    // При отсутствии отладчика деление вызовет исключение (переполнение при
    делении на -1),
    // обработчиком которого как раз будет KiFilterFiberContext
    result = (v0 / ((KdPitchDebugger | KdDebuggerNotPresent) != 0 ? -1 : 17));
}
return result;
}

```

Функция sub_14071815C очевидно не имеет публичного символа, поскольку связана с проверкой лицензии ОС.

```

VOID ExpLicenseWatchInitWorker()
{
    PVOID KiFilterParam;
    NTSTATUS (*KiFilterFiberContext)(PVOID pFilterparam);
    BOOLEAN ForgetAboutPG;

    // KiServiceTablesLocked == KiFilterParam
    KiFilterParam = KiInitialPcr.Prcb.HalReserved[1];
    KiInitialPcr.Prcb.HalReserved[1] = NULL;

    KiFilterFiberContext = KiInitialPcr.Prcb.HalReserved[0];
    KiInitialPcr.Prcb.HalReserved[0] = NULL;

    ForgetAboutPG = (InitSafeBootMode != 0) |
(KUSER_SHARED_DATA.KdDebuggerEnabled >> 1);

    // 96% случаев
    if (__rdtsc() % 100 > 3)
        ForgetAboutPG |= 1;

    if (!ForgetAboutPG && KiFilterFiberContext(KiFilterParam) != 1)
        KeBugCheckEx(SYSTEM_LICENSE_VIOLATION, 0x42424242, 0xC000026A, 0, 0);
}

```

Ниже приведен псевдокод функции KiFilterFiberContext, выбирающей способ проверки конкретного контекста и вызывающей функцию создания самого контекста.

```

BOOLEAN KiFilterFiberContext(PVOID pKiFilterParam)
{
    BOOLEAN Result = TRUE;
}

```

```

    DWORD64 dwDpcIdx1 = __rdtsc() % 13; // Выбор DPC, в которой будет
осуществляться проверка
    DWORD64 dwRand2 = __rdtsc() % 10; // 50 на 50, что создастся второй
контекст
    DWORD64 dwMethod1 = __rdtsc() % 6; // Выбор метода запуска проверки

    AntiDebug();

    Result = KiInitializePatchGuardContext(dwDpcIdx, dwMethod1, (dwRand2 < 6) + 1,
pKiFilterParam, TRUE);

    if (dwRand2 < 6)
    {
        DWORD64 dwDpcIdx2 = __rdtsc() % 13;
        DWORD64 dwMethod2 = __rdtsc() % 6;

        do
        {
            dwMethod2 = __rdtsc() % 6;
        }
        while ((dwMethod1 != 0) && (dwMethod1 == dwMethod2));

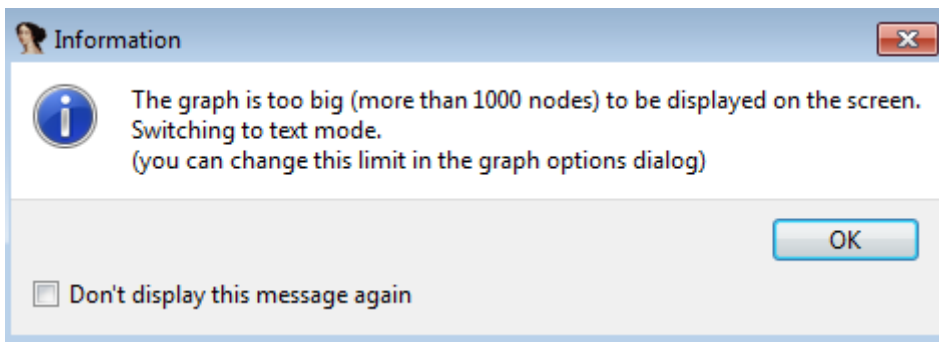
        Result = KiInitializePatchGuardContext(dwDpcIdx2, dwMethod2, 2,
pKiFilterParam, FALSE);
    }

    AntiDebug();

    return Result;
}

```

Функция, создающая контекст PatchGuard, обфусцирована настолько, что автоматические средства с ней не справляются, а исследователям неинтересно заниматься обратной разработкой. В статике это полная каша, более 10 000 строк декомпилированного «в лоб» кода (сама декомпиляция в IDA Pro занимает около 40 минут).



Все говорит об обширном использовании макросов:

- даже простейшая операция, такая как взятие случайного числа, занимает более 50 строк ассемблерного кода;
- все циклы развернуты;
- вставлено много «мертвого» кода;
- используется косвенное обращение к переменным и внешним функциям.

Динамика тоже довольно непроста. Вот пара примеров.

```
cli
xor eax, eax
cmp byte ptr cs:KdDebuggerNotPresent, al
jnz short loc_140F3CFBD
jmp short loc_140F3CFBB
sti
```

Что делает этот антиотладочный трюк? При подключенном отладчике входит в бесконечный непрерываемый цикл.

```
cli
sidt fword ptr [rbp+320h]
lidt fword ptr [rbp+228h]
mov dr7, r13
lidt fword ptr [rbp+320h]
sti
```

Что делает второй антиотладочный трюк? Загружает временную невалидную таблицу дескрипторов прерываний. Если мы следили за доступом к отладочным регистрам, произойдет отладочное исключение, которое при данных условиях приведет к tripple fault с последующей перезагрузкой.

Рассмотрим параметры функции KiInitializePatchGuardContext.

1. Индекс DPC функции, которая будет вызвана для проверки контекста и может быть одной из следующих:

- KiTimerDispatch
- KiDpcDispatch
- ExpTimerDpcRoutine
- IopTimerDispatch
- IopIrpStackProfilerTimer
- PopThermalZoneDpc
- CmpEnableLazyFlushDpcRoutine
- CmpLazyFlushDpcRoutine
- KiBalanceSetManagerDeferredRoutine
- ExpTimeRefreshDpcRoutine
- ExpTimeZoneDpcRoutine
- ExpCenturyDpcRoutine

2. Метод планирования проверки:

1. KeSetCoalescableTimer

Создается объект таймера, который запустит проверку через 2м:05с ± 5 с.

2. Prcb.AcpiReserved

DPC сработает при определенном событии ACPI, например при переходе в состояние низкого энергопотребления. Сработает не раньше чем через 2м:05с ± 5 с.

3. Prcb.HalReserved

DPC сработает при тике таймера HAL. Не раньше чем через 2м:05с ± 5 с.

4. PsCreateSystemThread

Создается отдельный системный поток, спящий 2м:05с ± 5 с. После этого вызывается проверка контекста.

5. KeInsertQueueApc

Создается regular kernel APC, срабатывающая сразу, но ждущая 2м:05с ± 5 с внутри work item.

6. KiBalanceSetManagerPeriodicDpc

DPC сработает по таймеру менеджера балансировки, не раньше чем через 2м:05с ± 5 с.

3. Назначение параметра до конца не ясно, известно лишь, что он влияет на количество проверок в контексте.

4. Параметр, специфичный для выбранного метода планирования.

5. Параметр, сообщающий о необходимости пересчета контрольных сумм для контекста.

DPC, которые вызывают проверку через исключение внутри себя, «смотрят» — является ли параметр DeferredContext указателем на неканоническую память. Если указатель в порядке, DPC выполняет свою законную работу. Иначе DPC вызывает цепочку рекурсивных функций, приводящих в конечном итоге к исключению (из-за разыменовывания неканонического адреса) и исполнению его обработчика.

Последовательности вызовов рекурсивных функций в зависимости от DPC-функции

ExpTimerDpcRoutine -> KiCustomAccessRoutine0 -> KiCustomRecurseRoutine0...
KiCustomRecurseRoutineN
IopTimerDispatch -> KiCustomAccessRoutine1 -> KiCustomRecurseRoutine1...
KiCustomRecurseRoutineN
IopIrpStackProfilerTimer -> KiCustomAccessRoutine2 -> KiCustomRecurseRoutine2...
KiCustomRecurseRoutineN
PopThermalZoneDpc -> KiCustomAccessRoutine3 -> KiCustomRecurseRoutine3...
KiCustomRecurseRoutineN
CmpEnableLazyFlushDpcRoutine -> KiCustomAccessRoutine4 ->
KiCustomRecurseRoutine4... KiCustomRecurseRoutineN
CmpLazyFlushDpcRoutine -> KiCustomAccessRoutine5 -> KiCustomRecurseRoutine5...
KiCustomRecurseRoutineN
KiBalanceSetManagerDeferredRoutine -> KiCustomAccessRoutine6 ->
KiCustomRecurseRoutine6... KiCustomRecurseRoutineN
ExpTimeRefreshDpcRoutine -> KiCustomAccessRoutine7 -> KiCustomRecurseRoutine7...
KiCustomRecurseRoutineN
ExpTimeZoneDpcRoutine -> KiCustomAccessRoutine8 -> KiCustomRecurseRoutine8...
KiCustomRecurseRoutineN
ExpCenturyDpcRoutine -> KiCustomAccessRoutine9 -> KiCustomRecurseRoutine9...
KiCustomRecurseRoutineN

Проверка контекста состоит из двух этапов: сперва проверка структуры самого контекста, которая происходит на DPC-уровне, затем планируется work item, осуществляющий проверку защищаемых структур в системном потоке. Если проверка была удачной, старый контекст удаляется и вместо него создается новый, который будет запущен через случайный интервал времени. Если проверка не удалась, PatchGuard зачищает все свои следы, в том числе зануляя стек, и демонстрирует синий экран с кодом ошибки 0x109: CRITICAL_STRUCTURE_CORRUPTION.

```

rmp = 0xFFFFFFFF00000000
esp = 0xFFFFD0006D5F0638
rip = 0xFFFFE00103C7C001
PF ZF IOPL = 0

Disasm
FFFFE00103C7BFFA: 00 00          add byte ptr [rax], al
FFFFE00103C7BFFC: 00 00          add byte ptr [rax], al
FFFFE00103C7BFFE: 00 40 A9      add byte ptr [rax-0x57], al
FFFFE00103C7C001: 2E 48 31 11   xor qword ptr [rcx], rdx
FFFFE00103C7C005: A0 BF 9F ED 7D 49 E4 .. mov al, byte ptr [0xa021e4497ded9fbf]
FFFFE00103C7C00E: BF 9F FD 7D 49   mov edi, 0x497dfd9f
FFFFE00103C7C013: E4 11         in al, 0x11
FFFFE00103C7C015: A0 BF 9F CD 7D 49 E4 .. mov al, byte ptr [0xa001e4497dcd9fbf]
FFFFE00103C7C01E: BF 9F DD 7D 49   mov edi, 0x497ddd9f
FFFFE00103C7C023: E4 71         in al, 0x71
FFFFE00103C7C025: A0 BF 9F AD 7D 49 E4 .. mov al, byte ptr [0xa061e4497dad9fbf]
FFFFE00103C7C02E: BF 9F BD 7D 49   mov edi, 0x497dbd9f
FFFFE00103C7C033: E4 51         in al, 0x51
FFFFE00103C7C035: A0 BF 9F 8D 7D 49 E4 .. mov al, byte ptr [0xa041e4497d8d9fbf]
FFFFE00103C7C03E: BF 9F 9D 7D 49   mov edi, 0x497d9d9f
FFFFE00103C7C043: 24 B1         and al, 0xb1
FFFFE00103C7C045: E8 8E CE AD 04   call 0xffffe00108758ed8
FFFFE00103C7C04A: E9 3D 31 E8 8E   jmp 0xffffe00092aff18c
FFFFE00103C7C04F: 86 D4         xchg ah, dl
FFFFE00103C7C051: A4           movsb byte ptr [rdi], byte ptr [rsi]
FFFFE00103C7C052: E8 B5 31 E8 C6   call 0xffffe000caaff20c
FFFFE00103C7C057: FF 74 AD 78     push qword ptr [rbp+rbp*4+0x78]
FFFFE00103C7C05B: B5 31         mov ch, 0x31
FFFFE00103C7C05D: A0 BF 5F 45 35 78 B5 .. mov al, byte ptr [0xd979b57835455fbf]
FFFFE00103C7C066: 1F           ???
FFFFE00103C7C067: 66 E5 35      in ax, 0x35
FFFFE00103C7C06A: 78 FD         js 0xffffe00103c7c069
FFFFE00103C7C06C: 00 79 3E      add byte ptr [rcx+0x3e], bh
FFFFE00103C7C06F: CE           ???
FFFFE00103C7C070: E5 35      in eax, 0x35

```

Саморасшифровывающийся контекст

Как победить

Существует несколько подходов к обезвреживанию PatchGuard:

- Такой патч образа ядра, чтобы PatchGuard вообще не инициализировался.
- Патч процедур проверки контекста.
- Хук KeBugCheck с восстановлением состояния системы.
- Отмена запланированных проверок.

Нам понравился последний способ, поскольку он является самым «чистым»: ничего не нужно хукать и пачтить, необходимо просто заменить значение некоторых переменных.

1. KeSetCoalescableTimer

Необходимо просканировать все таймеры, DPC для которых будет содержать DeferredContext с неканоническим адресом, и увеличить интервал ожидания для найденных до бесконечности.

2. `Prcb.AcpReserved`
Просто занулить данное поле.
3. `Prcb.HalReserved`
Просто занулить данное поле.
4. `PsCreateSystemThread`
Просканировать спящие потоки и раскрутить их стек. Если он упирается в функцию из структуры `KiServiceTablesLocked`, это наш клиент. Выставляем время сна, равное бесконечности.
5. `KeInsertQueueApc`
Просканировать все рабочие потоки с раскруткой стека. Если в стеке встречаются функции не из кодовой секции ядра, причем раскручивающиеся с использованием данных для функций `FsRtlMdlReadCompleteDevEx` и `FsRtlUninitializeSmallMcb`, это точно рабочий поток `PatchGuard`. Обезвреживаем так же, как в предыдущем варианте.
6. `KiBalanceSetManagerPeriodicDpc`
Восстановить «законную» процедуру — `KiBalanceSetManagerDeferredRoutine`.

Эти действия необходимо успеть совершить за 2 минуты по описанным выше причинам. Результат — проверка контекста никогда не будет запущена, а также не будет запланирована новая. `PatchGuard` не будет работать.

Windows 10

При осмотре `KiFilterFiberContext` из Windows 10 Technical Preview мы заметили небольшое изменение. Все старые методы планирования остались прежними. Однако появился новый, который пока что безусловно возвращает `STATUS_HV_FEATURE_UNAVAILABLE`. Немного покопавшись, мы обнаружили функцию `KiSwInterruptDispatch`, внутри которой явно идет расшифровка и вызов проверки контекста. Очевидно, что будет добавлена возможность осуществлять проверку контекстов по запросу гипервизора Hyper-V. От гипервизора при определенных условиях будет приходить синтетическое прерывание, обработчик которого будет проверять целостность ядра.

История продолжается

В статье мы старались не указывать имена конкретных функций, поскольку имена функций, используемых для расшифровки и проверки контекстов, намеренно изменены разработчиками `PatchGuard` и меняются в разных версиях ОС.

Вот пример несоответствия названия функции тому, чем она действительно

занимается. Это та самая функция, копия которой используется для саморасшифровки контекста.

```
      CmpAppendDllSection proc near
                                db      2Eh
00000000 2E 48 31 11      xor     [rcx], rdx
00000004 48 31 51 08      xor     [rcx+8], rdx
00000008 48 31 51 10      xor     [rcx+10h], rdx
0000000c 48 31 51 18      xor     [rcx+18h], rdx
00000010 48 31 51 20      xor     [rcx+20h], rdx
00000014 48 31 51 28      xor     [rcx+28h], rdx
00000018 48 31 51 30      xor     [rcx+30h], rdx
0000001c 48 31 51 38      xor     [rcx+38h], rdx
00000020 48 31 51 40      xor     [rcx+40h], rdx
00000024 48 31 51 48      xor     [rcx+48h], rdx
00000028 48 31 51 50      xor     [rcx+50h], rdx
0000002c 48 31 51 58      xor     [rcx+58h], rdx
00000030 48 31 51 60      xor     [rcx+60h], rdx
00000034 48 31 51 68      xor     [rcx+68h], rdx
00000038 48 31 51 70      xor     [rcx+70h], rdx
0000003c 48 31 51 78      xor     [rcx+78h], rdx
```

Одно хорошо: все эти функции находятся рядом, так что начать можно с функции KiFilterFiberContext. Очевидно, они все лежат в одном файле исходного кода. Однако проверка целостности ядра не ограничивается одним PatchGuard. В различные части ядра вставлены макросы, осуществляющие проверку тех или иных структур. Каждое такое место приходится искать вручную. Пример:

```
... --> Phase1InitializationDiscard --> CcInitializeCacheManager --> CcInitializeBcbP
rofiler
```

С вероятностью 50% данная функция осуществляет подсчет контрольной суммы для произвольной функции ядра и планирует ее проверку каждые 2 минуты в DPC с функцией CcBcbProfiler.

О компании Positive Technologies

Positive Technologies — лидер европейского рынка систем анализа защищенности и соответствия стандартам. Деятельность компании лицензирована ФСТЭК и ФСБ, продукция сертифицирована ФСТЭК, «Газпромом» и Минобороны РФ. Более 1000 организаций в 30 странах мира используют решения Positive Technologies для оценки уровня безопасности своих сетей и приложений, выполнения требований регуляторов и блокирования атак в режиме реального времени. Благодаря многолетним исследованиям специалисты Positive Technologies заслужили репутацию экспертов международного уровня по вопросам защиты SCADA- и ERP-систем, крупнейших банков и телекомов. Согласно исследованиям IDC, в 2013 году компания заняла третье место на российском рынке ПО для безопасности, а также стала лидером по темпам роста на

международном рынке систем управления уязвимостями. Подробнее о компании — на сайте ptsecurity.ru.