УДК 681.3.066

Анализ проблем верификации драйверов Windows

© Н.Г. Ершов, Н.Ю. Рязанова

МГТУ им. Н.Э. Баумана, Москва, 105005, Россия

Статья посвящена вопросам, связанным с безопасностью работы ОС Windows. Показано, что драйверы режима ядра, в которых используются средства взаимоисключения, могут быть потенциально опасными для системы и являться причиной краха системы. Анализируются результаты работы включенных в систему средств верификации Driver Verifier. Для проведения анализа разработаны драйверы режима ядра, содержащие разные типы угроз, и выполнена их верификация. Определены возможности и сформулированы практические рекомендации по использованию системных средств верификации с целью обнаружения и исключения из их кода потенциально опасных участков.

***Ключевые слова*:** *верификация, крах системы, драйвер, спин-блокировка, тупик.*

Операционные системы (ОС) семейства Windows, в отличие от систем Linux, являются «закрытыми». Это значит, что изменить код ядра системы нельзя. Однако часто возникает потребность адаптировать систему к выполнению конкретных задач. Изменить функциональность операционной системы Microsoft Windows можно, только написав соответствующий драйвер режима ядра. Некорректно написанный драйвер может разрушить систему, что приведет к потере хранимой в системе информации. Драйвер, содержащий потенциально опасный код, может быть успешно откомпилирован, отлажен и установлен в системе. При этом такой код может быть не выявлен ни на одном из этапов разработки драйвера. Опасность заключается в том, что при возникновении определенных ситуаций в системе такой драйвер приведет к краху системы.

При написании драйвера могут потребоваться создание в нем дополнительного потока, реализация монопольного доступа к внутренним для драйвера структурам или к глобальным структурам данных ядра системы средствами взаимоисключения, предоставляемыми ОС. Взаимоисключение в ОС Windows реализуется с помощью специальных объектов ядра, отличающихся друг от друга особенностями реализации. Некорректно выполненное в драйвере взаимоисключение, особенно в ситуациях многократного обращения к разделяемым ресурсам, может привести к взаимоблокировкам потоков ядра и краху системы. Контроль над объектами взаимоисключения очень сложен и ресурсоемок, поэтому в системе по умолчанию не осуществляется.

При отладке драйверов необходимо их тщательно тестировать и верифицировать. В состав операционных систем Microsoft Windows XP и созданных позднее для выявления проблемных драйверов включен Диспетчер проверки драйверов Microsoft Driver Verifier.exe [1]. Утилита Driver Verifier проверяет драйверы режима ядра и графические драйверы для выявления недопустимых вызовов функций или действий, способных аварийно завершить работу Windows. Однако драйверы, в которых могут возникнуть взаимоблокировки, очень сложно выявить. Замечено, что на практике взаимоблокировки возникают не мгновенно, и система реагирует на них не сразу. Поэтому после возникновения ошибки блокировки функция драйвера может еще какое-то время выполнять критические действия. В результате система диагностики укажет на последние действия функций установленного в системе драйвера, причем это может быть даже не драйвер, работа которого анализируется. В результате участки кода, в которых происходит взаимоблокировка, определены не будут. Кроме того, при крахе системы в принципе на диск должны записываться результаты ошибочных действий, однако содержание и информативность этих сообщений можно оценить только после краха системы.

Команда **!verifier** отладчика ядра и средство Verifier.exe показывают текущие параметры проверки драйверов и статистику в режиме реального времени. Наиболее распространенные системные ошибки, возникающие при проверке драйверов:

- IRQL_NOT_LESS_OR_EQUAL 0xA
- PAGE_FAULT_IN_NONPAGED_AREA 0x50
- PAGE_FAULT_IN_NONPAGED_AREA 0x50
- ATTEMPTED_WRITE_TO_READONLY_MEMORY 0xBE
- SPECIAL_POOL_DETECTED_MEMORY_CORRUPTION 0xC1
- DRIVER_VERIFIER_DETECTED_VIOLATION 0xC4
- DRIVER_CAUGHT_MODIFYING_FREED_POOL 0xC6
- TIMER_OR_DPC_INVALID 0xC7
- DRIVER_VERIFIER_IOMANAGER_VIOLATION 0xC9

В материалах фирмы Microsoft даются рекомендации по верификации драйверов режима ядра. Однако конкретные ситуации и примеры их отладки не приводятся. Анализ различных ситуаций, способных привести к краху системы, показывает, что не всегда можно

по информации, генерируемой штатными средствами верификации, получить достаточное количество данных, а иногда и просто выявить опасные участки кода.

В качестве первого примера работы штатных средств верификации ОС Windows рассмотрим спин-блокировки, которые являются одним из основных средств взаимоисключения в ядре.

Спин-блокировки используются в режиме активного ожидания. Это означает, что поток, ожидающий освобождения спин-блокировки, постоянно проверяет, не освободилась ли она, и тратит на эту проверку процессорное время. Спин-блокировки используются в тех кодах ядра, которые не могут быть заблокированы, например, в обработчиках прерываний. Спин-блокировки в ядре реализуются с помощью двух функций: KeAsquireSpinLock(), обеспечивающей захват блокировки, и KeReleaseSpinLock(), освобождающей блокировку. Спин-блокировки никак не контролируются операционной системной.

Для реализации взаимоисключения с помощью спин-блокировок характерно два вида ошибок:

- спин-блокировкам в ядре назначен уровень IRQL DPC/dispatch, который блокирует все действия, так как на этом уровне заблокированы действия по диспетчеризации потоков, поэтому код, установивший блокировку, может привести к краху системы, если попытается заставить планировщик выполнить операцию диспетчеризации или вызовет ошибку страницы [2];
- несоблюдение иерархии захвата спин-блокировок, из-за чего происходит повторный ошибочный захват, в результате которого поток войдет в цикл активного ожидания процессом, что приводит к зависанию системы и неизбежному краху.

Для анализа способов диагностики приведенных ошибок был разработан простейший драйвер режима ядра.

Проведенный анализ показал, что ошибки, как правило, надо искать не по всему коду драйвера. Они обычно концентрируются в некоторых специфических процедурах, таких как NTSTATUS DeviceControlRoutine (IN PDEVICE_OBJECT fdo, IN PIRP Irp), которая распознает IRP (I/O Request Packet — пакеты запросов ввода-вывода).

В функции на рис. 1 в неперемещаемой памяти создается объект типа KSPIN_LOCK с именем Qlock.

Функция сохраняет предыдущий уровень приоритета, определяя его до вызова блокировки и перехода процесса на уровень DPC/dispatch. Чтобы организовать ошибку чтения страницы, создается указатель на страницу в области памяти. Специальная переменная data создается с квалификатором volatile, который указывает компилятору

```
typedef struct _DEVICE_EXTENSION
        {
                KSPIN_LOCK QLock;
        }DEVICE_EXTENSION, *PDEVICE_EXTENSION;
VOID  SpinLock (VOID) // Спин-блокировка
{
  KIRQL  prevIrql;          //предыдущий IRQL
  PCHAR  memoryPtr; //указатель на область памяти ( страницу в буферном пуле )
  int    i = 0;
  volatile int data;
        PDEVICE_EXTENSION pdx = ExAllocatedPool(NonPagedPool,
sizeof(_DEVICE_EXTENSION));   //объявление объекта блокировки и выделение памяти под него
в неперемещаемой области.
        KeInitializeSpinLock(&pdx->Qlock);
        /*Распределение размера. Размер "должен быть" меньше, чем размер страницы  минус
заголовок. Для верификации информация передается из особого пула*/
        #define ALLOCATION_SIZE  2048
  //    Захват, а затем освобождение страницы памяти
  //    ExAllocatePool - выделяет память нужного объема и возвращает указатель не нее.
        //          ExFreePool - освобождает память по указателем
  memoryPtr = ExAllocatePool( PagedPool, ALLOCATION_SIZE );
  ExFreePool( memoryPtr );
  //    Пройтись по выделенной памяти IRQL и продолжать дальше
  //    читать данные за размер страницы при высоком IRQL.
  //    KeAcquireSpinLock - повышает уровень приоритета до DPC\dispatch
        KeAcquireSpinLock(&pdx->Qlock, &prevIrql);
  while( 1 ) {
    data = *((PULONG) (memoryPtr+i));
    i += 4096;
  }
        KeReleaseSpinLock(&pdx->Qlock, prevIrql);
}
```

Рис. 1. Функция, моделирующая выход за пределы страницы

собирать код без оптимизации. Объект блокировки инициализируется, затем выделяется страница в памяти и тут же освобождается. Далее процесс входит в режим блокировки и выполняет цикл чтения, обращаясь за пределы области памяти выделенной страницы. Возникает страничное исключение, и защитный механизм операционной системы организует ее крах.

Системная функция KeBugCheckEx() в момент краха возвращает ошибку DRIVER_IRQL_NOT_LESS_OR_EQUAL, однако не определяет локализацию ошибки. Анализируя каждую строку стека

STACK_TEXT (рис. 2), можно определить, что вызван драйвер Primer+0x557, затем прошла верификация захвата спин-блокировки, после этого система диагностировала исключение при повышенном уровне приоритета и драйвер nt+0xcb270 вызвал крах системы. Очевидно, что в данном случае определить причину краха легко, так как функция, вызывающая крах, известна.

```
ADDITIONAL_DEBUG_TEXT:
You can run '.symfix; .reload' to try to fix the symbol path and load symbols.
MODULE_NAME: Primer
FAULTING_MODULE: 81e1a000 nt
DEBUG_FLR_IMAGE_TIMESTAMP:  516e9505
DV_VIOLATED_CONDITION:  ExAllocatePoolWithTagPriority can be called at DISPATCH_LEVEL only
if a NonPagedXXX is specified for PoolType, not PagedPool.
DEFAULT_BUCKET_ID:  WIN8_DRIVER_FAULT
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be wrong.
af80faec 85426301 000000c4 00020004 8542a93a nt+0xcb270
af80fb08 85422def 8542a93a 00020004 00000000 VerifierExt+0x7301
af80fb20 822b208e 00000081 00000800 70617257 VerifierExt+0x3def
af80fb40 aa9fe557 00000001 00000800 f82d6388 nt+0x49808e
af80fbc8 822b0f9b 8ac8e468 b3670f68 00000100 Primer+0x557
af80fbe8 81fbb3eb 8205b331 b642da38 b3670f68 nt+0x496f9b
SYMBOL_STACK_INDEX:  4
IMAGE_NAME:  Primer.sys
...
```

Рис. 2. Аварийный дамп, полученный при крахе системы

Для исключения подобных ситуаций необходимо выявить драйвер, выполняющий ошибочное чтение, инициировав специальный режим Driver Verifier, при котором драйверы используют специальный пул во всех случаях, когда требуется дополнительное выделение памяти. Существует способ определить конкретный участок кода, в котором произошел сбой из-за несоответствия приоритетов IRQL. При постановке драйвера на верификацию диспетчер Driver Verifier самостоятельно организует крах системы. Функция KeBugCheckEx() вернет ошибку DRIVER_VERIFIER_DETECTED_VIOLATION и поместит в дамп максимум информации об ошибке (рис. 3).

Место локализации ошибки указывается с точностью до строки кода, однако не следует забывать причину возникновения краха системы — невозможность генерации исключений на высоком уровне приоритетов IRQL.

```
FAULTING_SOURCE_CODE:
    377:
    378:                    KeAcquireSpinLock(&Crashing, &prevIrql);
    379:                    while( 1 ) {
    380:
  > 381:                            data = *((PULONG) (memoryPtr+i));
    382:                            i += 4096;
    383:                    }
    384:                    KeReleaseSpinLock(&Crashing, prevIrql);
    385:
    386:            }
  SYMBOL_STACK_INDEX: 2
  SYMBOL_NAME: Primer!DeviceControlRoutine+145
 FOLLOWUP_NAME: MachineOwner
  IMAGE_NAME: Primer.sys
  BUCKET_ID: WRONG_SYMBOLS
Followup: MachineOwner
```

Рис. 3. Дамп, формируемый Driver Verifier

Для выявления ошибок такого рода рекомендуется, помимо использования Driver Verifier, анализировать также стек ядра. Информация, содержащаяся в нем, демонстрирует реакцию системы на события в коде, т. е. раскрывает механизм верификации и выявляет этапы краха системы.

Для анализа второй из перечисленных проблем — несоблюдения иерархии захвата блокировок — написана функция (рис. 4), моделирующая данную ситуацию в простейшем виде. В функции инициализируется два объекта спин-блокировок. Сначала захватываются оба объекта, затем один из них освобождается, а другой повторно захватывается. Тем самым нарушается иерархия блокировок (спин-блокировки в Windows не рекурсивны). При попытке повторного захвата спин-блокировки операционная система зависает и организует крах.

Детальный анализ данной проблемы очень важен, так как попадание системы в тупик приводит к крайне разрушительным последствиям. С целью выяснения реакции защитных механизмов операционной системы анализ проводился в двух режимах: с выключенным и с включенным режимом контроля взаимоблокировок (deadlock detection) в Driver Verifier.

```
#if DBG
  DbgPrint("-Primer- System Crash On SpinLock  2   ------ ");
#endif
  KIRQL  prevIrql;          //предыдущий IRQL
// Инициализация двух объектов спин-блокировок
  KeInitializeSpinLock(&Spin1);
  KeInitializeSpinLock(&Spin2);
  //  KeAcquireSpinLock - повышает уровень приоритета до DPC\dispatch
  KeAcquireSpinLock(&Spin1, &prevIrql);
  KeAcquireSpinLock(&Spin2, &prevIrql);
//Захват Spin1 и Spin2, а затем освобождение блокировки Spin1 и повторный захват Spin2
  KeReleaseSpinLock(&Spin1, prevIrql);
  KeAcquireSpinLock(&Spin2, &prevIrql);
```

Рис. 4. Моделирование повторного захвата спин-блокировки

В первом случае при отключенном режиме Driver Verifier система самостоятельно вызвала крах, но определить источник ошибки не смогла. Анализ аварийного дампа также не позволил определить драйвер, виновный в крахе системы, а в уцелевшем стеке ядра экспериментальный драйвер вообще не фигурировал. Фрагмент дампа памяти с пятью последними элементами стека показан на рис. 5. В дампе указано, что ошибку вызвал процесс ntoskrnl.exe (исполнительная система и ядро для однопроцессорных систем).

В случае вызова функции SpinLock() анализировать крах системы очень сложно, потому что он происходит в момент обращения к поврежденным данным, а не входа в блокировку. Так как система некоторое время продолжает работать после ошибки чтения страницы, анализ аварийного дампа может указать на другой процесс, функции которого исполнялись непосредственно в момент краха. В данном случае последними были вызваны драйвер hal и процесс ntoskrnl.exe.

Во втором случае режим анализа взаимоблокировок в Driver Verifier был включен. После генерации краха системы, инициированного программой Driver Verifier, удалось установить драйвер, явившийся причиной краха. Однако локализация ошибки не выполнена. При анализе (рис. 6) легко определить, что ошибка организована двумя файлами — halmacpi.dll и Primer.sys.

Вызов экспериментального драйвера в стеке находится очень «глубоко», но поскольку после произошедшей ошибки вызывались только системные драйвера типа hal и nt, вероятно, виновник краха — экспериментальный драйвер.

```
9b0b19bc 8236213e 00000133 00000001 00000780 nt+0xcb270
9b0b1a20 822667ec 00049235 00049236 82410102 nt+0x15713e
9b0b1a8c 827b8ffe 9b0b1aa4 822a72a1 ffffffff nt+0x5b7ec
9b0b1aac 827ca60d 00000002 000000d1 9b0b1b50 hal+0x1cffe
9b0b1abc 822a72a1 badb0d00 00000000 004e6580 hal+0x2e60d
STACK_COMMAND:  kb
FOLLOWUP_IP:
nt+cb270
822d6270 55              push    ebp
SYMBOL_STACK_INDEX:  0
SYMBOL_NAME:  nt+cb270
FOLLOWUP_NAME:  MachineOwner
IMAGE_NAME:  ntoskrnl.exe
BUCKET_ID:  WRONG_SYMBOLS
Followup: MachineOwner
```

Рис. 5. Фрагмент дампа памяти с последними элементами стека

```
**********************************************************************
Bugcheck
    Analysis                *
**********************************************************************
BugCheck 133, {1, 780, 0, 0}
*** WARNING: Unable to verify timestamp for halmacpi.dll
*** ERROR: Module load completed but symbols could not be loaded for halmacpi.dll
*** WARNING: Unable to verify timestamp for Primer.sys
*** ERROR: Module load completed but symbols could not be loaded for Primer.sys
***** Kernel symbols are WRONG. Please fix symbols to do analysis.
**********************************************************************
acf889d8 815c713e 00000133 00000001 00000780 nt+0xcb270
acf88a3c 814cb50a 00004f4e 00004f4f 00000002 nt+0x15713e
acf88a64 81437fd4 8150c2a1 ffffffff acf88b24 nt+0x5b50a
acf88a80 8144960d acf88a02 000000d1 acf88b24 hal+0x1cfd4
acf88a90 8150c2a1 badb0d00 00000000 0094dc40 hal+0x2e60d
```

Рис. 6. Аварийный дамп при включенном режиме анализа блокировок

В результате анализа можно сделать вывод, что в случае подобной ошибки рекомендуется провести анализ стека обязательно при включенном режиме Driver Verifier, иначе драйвер в стеке вообще может не фигурировать.

Как отмечалось выше, использование спин-блокировок повышает уровень приоритета до DPC/Dispatch и блокирует любые действия в системе. В тех случаях, когда выполнение кода ядра может быть переведено в состояние ожидания, могут использоваться другие средства синхронизации, такие как быстрые мьютексы [2].

Однако и эти средства не всегда защищены от ситуаций, приводящих к взаимоблокировкам (например, когда приоритет процесса при захвате мьютекса повышается до уровня APC_LEVEL (Asynchronous Procedure Call)). При этом станет невозможным создавать синхронные IPR, так как функция должна вызываться на пользовательском уровне привилегий — PASSIVE_LEVEL, который имеет приоритет ниже, чем APC_LEVEL. Ожидание синхронного IRP приведет к взаимной блокировке по причине несоответствия уровня IRQL. Ошибка, вызванная несоответствием IRQL, рассмотрена выше на примере спин-блокировок. Реакция системы в этом случае будет аналогичной. Кроме того, быстрые мьютексы не рекурсивны, и повторный их захват является серьезной проблемой.

Для анализа ситуации была написана функция, вызывающая повторный захват быстрого мьютекса по аналогии со спин-блокировками (рис. 7). В функции создается два быстрых мьютекса. Каждый из них

```
//      DeadLock
//      Тупик в системе будет организован при попытке захвата быстрого мьютекса
//      не смотря на то, что процесс уже является его владельцем.
FAST_MUTEX    Fmutex1;
FAST_MUTEX    Fmutex2;
// Блокировка на быстрых мьютексах
VOID DeadLock(VOID) {
  ExInitializeFastMutex( &Fmutex1 ); //инициализация быстрого мьютекса позволяет
  передать контроль над тупиками программисту
  ExInitializeFastMutex( &Fmutex2 );
  ExAcquireFastMutex( &Fmutex1 );
  ExAcquireFastMutex( &Fmutex2 );
  ExReleaseFastMutex( &Fmutex1 );
  ExAcquireFastMutex( &Fmutex2 );
}
```

Рис. 7. Функция с двумя быстрыми мьютексами

не может быть захвачен рекурсивно, а поток, захвативший быстрый мьютекс, не может получать некоторые APC, что ограничивает возможность обмена сообщениями через синхронные IRP. ExAcquire Fast Mutex Unsafe ставит поток в состояние ожидания на мьютексе, если указанный быстрый мьютекс не может быть получен немедленно. В противном случае вызывающему потоку дается право собственности на мьютекс и эксклюзивный защищенный доступ к ресурсу. Далее происходит освобождение мьютекса 1 и повторный ошибочный захват мьютекса 2. Возникает тупик.

Были проведены эксперименты с включенным режимом отслеживания взаимоблокировок Driver Verifier и без него. По умолчанию (при отключенном режиме выявления взаимоблокировок) в операционной системе нет штатных средств обработки такого вида тупиков, поэтому при вызове функции Deadlock() система зависает, не организуя при этом крах. Единственным способом ликвидации такого тупика является перезагрузка. Естественно, никакого аварийного дампа при этом не создается, и определить причину зависания невозможно.

При включенном режиме выявления взаимоблокировок Driver Verifier определяет проблемный драйвер (рис. 8). Простейшая форма взаимоблокировки, представленная в функции Deadlock(), была в этом режиме выявлена.

```
ADDITIONAL_DEBUG_TEXT:
You can run '.symfix; .reload' to try to fix the symbol path and load symbols.
MODULE_NAME: Primer
FAULTING_MODULE: 81465000 nt
DEBUG_FLR_IMAGE_TIMESTAMP:  516e9505
DV_VIOLATED_CONDITION:  ExAcquireFastMutex should only be called at IRQL <= APC_LEVEL.
SYMBOL_STACK_INDEX:  4
SYMBOL_NAME:  Primer+5b2
```

Рис. 8. Аварийный дамп при блокировке на быстрых мьютексах

Локализацию ошибки в драйвере установить невозможно, однако сама операционная система сформировала правило вызова системных функций (рис. 9).

```
2: kd> !ruleinfo 0x20005

The IrqlExApcLte1 rule specifies that the driver calls ExAcquireFastMutex
and ExTryToAcquireFastMutex only at IRQL <= APC_LEVEL.
```

Рис. 9. Правило вызова системных функций

Рекомендуется подробно изучить дамп памяти. По сформулированным в нем правилам можно установить локализацию ошибки.

Поиск ошибки выполняется в тех местах кода, где вызывается проблемная системная функция.

Driver Verifier организует крах системы с возвратом стандартного кода ошибки: DRIVER_VERIFIER_DETECTED_VIOLATION. При анализе дампа можно определить место возникновения ошибки с точностью до системной функции, что, несомненно, проще, чем проверять весь код драйвера в случае возникновения аналогичной ситуации при спин-блокировках.

Можно сделать вывод, что взаимные блокировки являются потенциальной проблемой любой операционной системы и особенно опасны при работе драйверов режима ядра. В любом случае, взаимоблокировка — это аварийная ситуация в системе, и для предотвращения значительных потерь данных операционной системе важно завершить свою работу корректно и предоставить максимум информации о причинах аварии.

Таким образом, по результатам проведенных экспериментов можно сформулировать правила написания драйверов с использованием средств взаимоисключения:

- отладку драйвера следует проводить при включенном режиме отслеживания взаимоблокировок Driver Verifier;
- для определения ошибочного кода необходимо анализировать дополнительную информацию о крахе, такую как: правила — !ruleinfo, данные детального анализа — !analyze -v, стек потоков — !tbread;
- в случае если невозможно выявить причину при анализе дополнительной информации, нужно исследовать стек ядра, анализируя последовательность вызовов.

Тестирование драйверов на предмет взаимоблокировок усложняется тем, что этот процесс непериодический и сложно моделируется в рабочих драйверах. В частности, ошибки сложно выявить из-за недетерминированности поведения таких процессов. Хорошее проектирование кода драйвера и детальное тестирование позволят минимизировать риск возникновения ошибок.

ЛИТЕРАТУРА

[1] Использование средства проверки драйверов для выявления проблем с драйверами Windows. URL: http://support.microsoft.com/kb/244617

[2] Руссинович М., Соломон Д. *Внутреннее устройство Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер класс*. Москва, «Русская редакция»; Санкт-Петербург, Питер, 2005, 990 с.

[3] Oney W. Programming the Microsoft Windows Driver Model. Microsoft.- Redmond: Microsoft Press, 1999, 342 p.

Статья поступила в редакцию 10.06.2013

Ссылку на эту статью просим оформлять следующим образом:

Ершов Н.Г., Рязанова Н.Ю. Анализ проблем верификации драйверов Windows. *Инженерный журнал: наука и инновации*, 2013, вып. 6. URL: http://engjournal.ru/catalog/it/hidden/770.html

Ершов Никита Георгиевич родился в 1991 г. Студент кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана. e-mail: yershov.n@mail.ru

Рязанова Наталья Юрьевна родилась в 1951 г. Канд. техн. наук, доцент кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н.Э. Баумана. Автор 36 печатных работ. Область научных интересов: разработка системного программного обеспечения, алгоритмы машинной графики. e-mail: ryaz_nu@mail.ru