Кафедра системного программирования

математико-механического факультета

Санкт-Петербургского государственного университета

Дополнительная защита кода, обфусцированного посредством виртуальной машины

Выпускная квалификационная бакалаврская работа

Григория Ниценко

Допущен к защите.

Зав. кафедрой:

д. ф.-м. н., профессор Терехов А.Н.

Научный руководитель:

ст. пр. Сартасов С.Ю.

Рецензент:

Мордвинов Д.А.

2014

Saint-Petersburg State University

Faculty of Mathematics and Mechanics

Department of Software Engineering

Additional protection of VM-obfuscated code

Graduation project

Grigory Nitsenko

Head of department:

Professor Andrey N. Terekhov

Doctor of Physics and Mathematics

Advisor:

Assistant professor Stanislav U. Sartasov

Reviewer

Dmitry A. Mordvinov

2014

Оглавление

1. Введение

Проблема защиты проприетарного программного обеспечения от обратной разработки всегда была актуальна и повсеместна. Разработчик (или IT-компания) хочет обеспечить защиту своего продукта от копирования или предотвратить кражу интеллектуальной собственности, например, созданных алгоритмов. Тем не менее злоумышленники или недобросовестные программисты также имеют желание обратно разработать успешные чужие программные продукты, например, для их использования в собственных решениях или для поиска уязвимостей в программных системах для их последующего взлома.

Один из известных примеров украденного программного обеспечения является разработка компанией Phoenix Technologies Ltd. собственного BIOS, который был бы совместим с проприетарным BIOS от компании IBM. Исследуя спецификацию и принцип работы BIOS компании IBM, команда разработчиков из Phoenix Technologies Ltd. создала его собственную версию, которую затем компания начала продавать. Этот случай, произошедший в 1980-х годах в США, стал первым прецедентом обратной разработки программного обеспечения, после которого появилась проблема защиты программных продуктов от копирования **Ошибка! Источник ссылки не айден.**.

Конечно, обратная разработка может использоваться и для добросовестных целей. Например, для продолжения работы над программными продуктами, которые уже не поддерживаются и не содержат документации. В таких случаях обратная разработка используется для создания максимально приближённых к изначальным систем. Кроме того, компании, специализирующиеся на защите от вирусных программ, используют обратную разработку для исследования принципов работы

вредоносного программного обеспечения для последующего создания эффективных методов борьбы с ними.

В современном мире методами анализа программного обеспечения для его последующей обратной разработки является динамический и статический анализы.

Статический анализ исследования обычно связан с изучением файлов программного продукта. В качестве исходных данных для статического анализа может быть использован код программы. Также это может быть различная метаинформация или сопровождающая программное обеспечение документация.

Динамический анализ предполагает исследование программы во время её работы. При этом подходе изучаются обращения программы к памяти, потоки данных, которыми обмениваются процессы программы в ходе её работы, и любая другая необходимая информация, используется режим отладки (debugging), трассировка или логирование каких-либо интересных исследователю действий программы.

Популярным методом защиты от статического и динамического анализа является обфускация программ, одним из самых эффективных подходов которой является обфускация с помощью виртуальной машины. В данной работе был предложен способ для защиты программ, обфусцированных с помощью виртуальных машин, от анализа потока системных вызовов.

2. Обфускация и деобфускация

Обфускация, или запутывание кода, это приведение исходного кода программ к виду, трудному для программного анализа и человеческого понимания. Концепция, которой следует подход обфускации, заключается в безопасности через неясность (security through obscurity) [2]**Ошибка! сточник ссылки не найден.**.

Деобфускация – это процесс анализа обфусцированного кода с целью понять принципы его работы, иными словами, обратная разработка обфусцированных программ.

Формально, если дана программа P, состоящая из объектов исходного кода $\{S_1, .., S_k\}$ (классы, методы, переменные, структуры данных, и т.д.), преобразование $T = \{T_1, .., T_n\}$, то программа $P' = T(P)$ будет являться *обфусцированной программой* P, если:

1) P' имеет то же видимое поведение (так называемые динамические свойства), что и P, то есть преобразование T *сохраняет семантику*;
2) *неясность* P' максимизирована, то есть понимание и обратная разработка P' занимает больше времени, чем обратная разработка P, при использовании одних и тех же подходов к обратной разработке;
3) *эффективность* преобразования $T_i(S_i)$ максимизирована, то есть крайне трудно разработать автоматический инструмент для отмены преобразования или использование такого инструмента крайне времязатратно;
4) *схожесть статических свойств* $T_i(S_i)$ и S_i максимизирована;
5) *отличие производительностей* P' и P минимизировано.

2.1. Основные подходы к обфускации

В данном разделе будут описаны основные методы обфускации, которые применяются разработчиками программного обеспечения.

Один из методов – это объединение структурных частей исходного кода программ. Исходный код программ, в зависимости от используемых для его создания технологий, может быть разделён структурно. Так, например, для кода, написанного на языке C#, такими структурными компонентами будут классы, сборки, пространства имён. После обфускации несколько классов могут быть объединены в один, а исходный код, изначально содержащий несколько программных имён, будет находиться в единственном пространстве имён.

Другой метод – это переименовывание компонент исходного кода. Различные компоненты исходного кода могут быть переименованы в не несущие никакого смысла названия.

Кроме того, обфускация может происходить на уровне порядка выполнения исходного кода, добавления не несущего никакой семантики кода, шифрования строк и с помощью многих других подходов.

Однако ни одна из перечисленных методик не защищает от динамического анализа, хотя все они достаточно эффективны для противодействия статическому анализу.

2.2. Обфускация с помощью виртуальных машин

Эффективный метод обфускации программ, который затрудняет как статический, так и динамический анализ, является обфускация с помощью виртуальных машин (англ. virtualization-obfuscated software) [2]**Ошибка! Источник ссылки не найден.**.

При данном подходе программа P, которую необходимо обфусцировать, преобразуется в байт-код для некоторой виртуальной машины EM, которая имеет интерфейс для её исполнения L **Ошибка! Источник ссылки не айден.**. Таким образом, обфусцированная программа P_L исполняется специально разработанной виртуальной машиной EM (рис. 1).

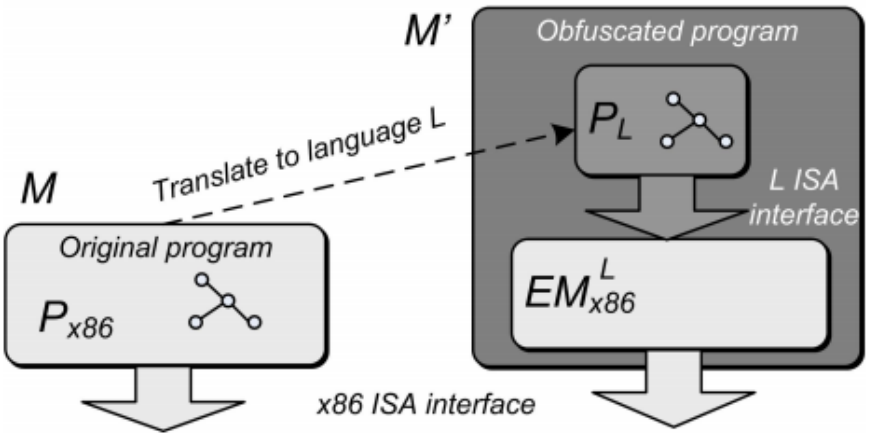

Рис. 1. Архитектура visualization-obfuscated software (рис. взят из [3]).

Также при данном подходе можно реализовать многослойную эмуляцию, в результате которой одна созданная виртуальная машина исполняется на другой. Количество слоёв эмуляции не ограничено, и ограничивается только пятым требованием к обфускации (см. раздел 2), а именно сохранением производительности.

2.3. Применение анализа потока системных вызовов при деобфускации

Подход к обфускации посредством виртуальных машин является крайне эффективным с точки зрения защиты от деобфускации, но, тем не менее, имеет серьёзную уязвимость. В данном разделе описаны преимущества данного подхода и главный недостаток, а также предложен подход, который позволяет использовать этот недостаток и максимально эффективно деобфусцировать программы, обфусцированные данным методом.

При применении статического анализа следует обратно разрабатывать виртуальную машину, а не исследовать обфусцированный код программы, который она исполняет, так как, даже имея доступ к этому коду, его крайне трудно исследовать, по причине того, что он может оказаться семантически бессмысленным. Если же понять, как виртуальная машина исполняет

обфусцированный код, то расшифровать его уже возможно. Но статический анализ также может быть неэффективным, так как при трансляции кода программы обфускатором в байт-код виртуальной машины может использоваться фактор случайности и, таким образом, инструкции обфусцированного кода разных обфусцированных программ будут исполняться иначе [4].

Динамический же анализ ещё менее эффективен, так как мы не сможем применить такие средства как отладку, трассировку, исследование потоков данных, так как обфусцированная программа взаимодействует только с виртуальной машиной.

Одним из возможных и эффективных подходов для исследования программ, обфусцированных таким способом, является анализ взаимодействия виртуальной машины с операционной системой. Данный метод предлагает исследовать, какие исполняемые инструкции обфусцированного кода взаимодействуют с операционной системой, а также идентифицировать инструкции, которые непосредственно на это влияют. Полученное множество инструкций есть приближение к коду оригинальной программы [3].

При данном подходе мы получаем доступ к тем частям программы, которые исполнялись в процессе её работы. Таким образом, посредством применения метода исполнения программой различных функций можно восстановить оригинальную программу полностью.

Авторы статьи [3] предлагают использовать такие инструменты для отслеживания взаимодействия виртуальной машины с системой как qemu, OllyDbg, Ether и другие, позволяющие проводить трассировку выполненных системных инструкций, после чего вычленять из трассы релевантные, относящиеся к исполняемой обфусцированной программе, системные вызовы.

3. Постановка задачи

Целью данной работы было создание подхода обфускации, противодействующего обозначенному в предыдущем разделе подходу деобфускации из статьи [3], связанному с исследованием потока системных вызовов. Операционные системы, для которых должен быть разработан инструмент, являются системы Windows NT. Технология системных вызов, исследованная в рамках работы, относится к процессорам Intel x86. Для достижения цели работы были сформулированы следующие задачи.

1. Исследование подходов к обфукации и деобфускации кода, в частности обфускации посредством виртуальных машин.
2. Исследование технологии системных вызовов – механизма взаимодействия процессов с ядрами операционных систем Windows NT, а также возможности логирования потока системных вызовов, которые выполняет операционная система при исполнении процесса.
3. Разработка подхода, препятствующего обратной разработке программ, обфусцированных с помощью виртуальных машин, посредством анализа потока системных вызовов, предложенного в [3].
4. Реализовать программную компоненту (операционные системы Windows NT, процессоры Intel x86), реализующую созданный алгоритм.

4. Системные вызовы в системах Windows NT

В данном разделе описано, что такое системные вызовы, и как данный механизм реализован в операционных системах Windows NT.

4.1. Обзор технологии системных вызовов

В операционных системам системный вызов – это способ обращения процессов к ядру операционной системы для запроса на выполнение некоторых операций. Обычно операционная система предоставляет библиотеку (или API), которая является посредником при взаимодействии между процессами и ядром. В системах Windows NT такое API является частью Native API, некоторые функции которого реализованы в системной библиотеке $ntdll.dll$. Функции данной библиотеки вызываются посредством Windows API – набора интерфейсов программирования приложений, которые реализованы в библиотеках $kernerl32.dll, gdi32.dll, user32.dll$ и др. Особенностью Native API, в отличии от Windows API, является его отсутствие документации к ней [5].

Так как нас в первую очередь интересует возможность логирования потока системных вызовов и создания собственных системных вызов, то Native API следует более подробно изучить.

4.2. Обзор взаимодействия процессов с Native API

Взаимодействия процессов с Native API осуществляется посредством библиотеки $ntdll.dll$. Основная часть данной библиотеки – это заглушки, которые позволяют переключать поток управления в привилегированный режим. Это осуществляется посредством вызова программного исключения.

Каждая «заглушка» внутри $ntdll.dll$ имеет следующую структуру (приложение 1). Первая инструкция загружает в регистр процессора *eax* индекс функции Native API. Каждая функция в Native API имеет уникальный номер, который генерируется автоматически. Вторая инструкция загружает в регистр *edx* указатели на параметры вызываемой функции. Третья инструкция

вызывает прерывание, которое переводит поток управления в режим ядра. Последняя инструкция заглушки возвращает управление в режим пользователя.

Важно отметить, что, начиная с Windows 2000, в операционных системах Windows NT вместо третьей инструкции используется команда процессора SYSENTER.

Обработчик вызываемого заглушкой исключения в Native API является KiSystemService. Он определяет, является ли корректным указанный номер функции, и, если является, то передаёт управление ядру операционной системы. Ядро вызывает необходимую функцию посредством поиска её номера в специальной системной таблице System Service Table.

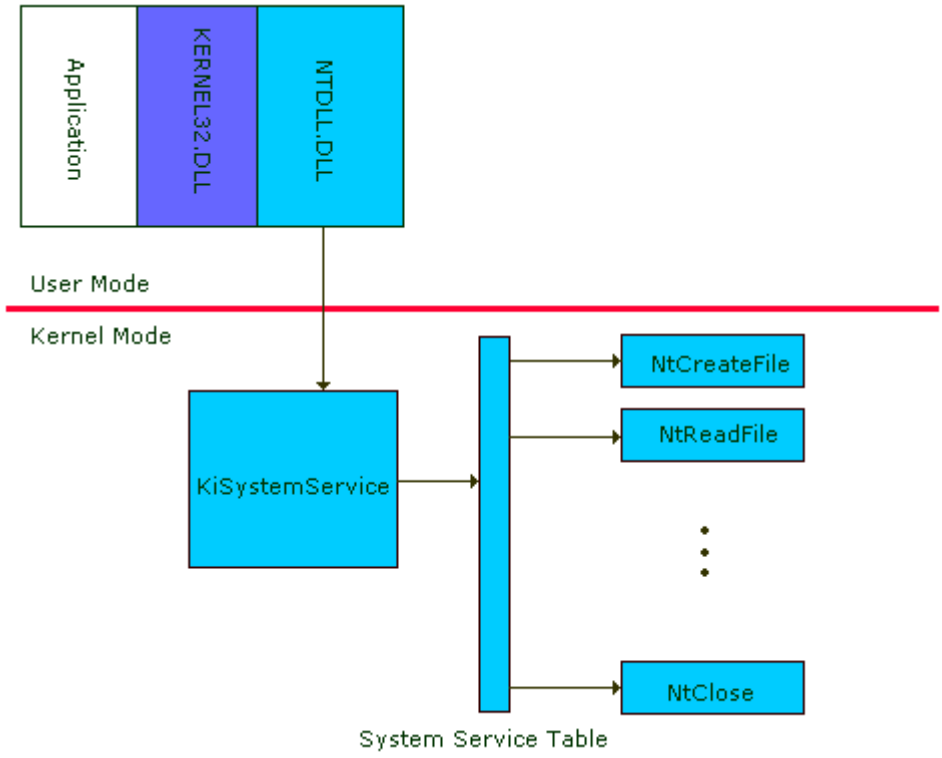

Рис. 2. Архитектура технологии системных вызовов систем Windows NT (рис. взят из [5]).

4.3. Логирование вызываемых процессом системных вызовов

Очевидным является то, что для отслеживания вызываемых процессом системных вызовов достаточно отслеживать исключения, вызываемые в результате обращения к библиотеке $ntdll.dll$. Данный подход был предложен в [6].

5. Разработка подхода для защиты кода, обфусцированного посредством виртуальных машин

В данном разделе описан подход, предложенный для противодействия обратной разработке программ, обфусцированных с помощью виртуальных машин, посредством анализа потока системных вызов, а также описан созданный для такого противодействия драйвер для операционных систем Windows NT и процессоров Intel x86.

5.1. Описание подхода

Так как предложенный в [3] метод деобфускации программ, обфусцированных посредством виртуальной машины, предполагает анализ потока системных вызов, то для противодействия такому подходу была предложена генерация системных вызовов, которая должна выполняться при каждом обращении виртуальной машины, исполняющей инструкции обфусцированной программы, к операционной системе.

Алгоритм, предложенный в статье [3], предлагает идентифицировать множество релевантных системных вызовов, по которому впоследствии можно восстановить оригинальный код обфусцированной программы, посредством анализа, основанного на значениях (*value-based approach*). Данный анализ формирует релевантное множество посредством выбора из потока системных вызов тех, что взаимодействуют с множеством аргументов системных вызовов или непосредственно влияют на это. Возьмём в качестве примера следующий набор инструкций из потока системных вызовов (приложение 2). В результате value-based анализа в множество релевантных инструкции попадёт только инструкция I_3, имеющая аргументом *print*, при этом будут проигнорированы инструкции I_1, I_2, использующиеся для вычисления адресов.

Таким образом, мы можем предотвратить качественное формирование множества релевантных системных вызовов, если добавим в поток системных

вызов свои, не влияющие никаким образом на исполнение обфусцированной программы, но расширяющие получающееся в результате применения анализа релевантное множество «шумом». В результате деобфускация будет невозможно или крайне затруднена, так как начальные данные будут не точны.

Согласно [3], формирование множества релевантных системных вызовов происходит следующим образом:

1) используя инструменты для трассировки, указанные в разделе 2.3, получаем информацию об аргументах системных вызовов;

2) формируется множество S, состоящее из адресов, содержащих аргументы системных вызовов;

3) для каждой инструкции I в потоке системных вызовов, если I использует адрес $l \in S$, то I является релевантной, а l удаляется из S;

4) множество адресов, использующихся I, добавляется в S.

Формирование множества заканчивается, когда $S = \emptyset$.

Таким образом, добавляя в поток системных вызовов «шум», мы не позволяем алгоритму корректно сформировать множество S, что доказывает эффективность описанного в работе подхода для противодействию данному анализу.

5.2. Разработка драйвера для создания системных вызовов

Так как для исполнения системных вызовов в системах Windows NT используется обработчик KiSystemService, достаточно заменить его на собственную программную компоненту, сохраняющую функциональность KiSystemService и расширяющую её, позволяя создавать «шум» из системных вызовов. Предполагается заменить код KiSystemService на собственный, который при каждом обращении виртуальной машины к Native API создавал несколько дополнительных системных вызовов. Так как замена

KiSystemService предполагает размещение своего кода в ядре ОС, то для этого необходимо написать драйвер для систем Windows NT, который реализовывал бы данную возможность [6].

5.2.1. Обзор функциональности

Разработанная программная компонента должна соответствовать следующим требованиям.

1. Быть драйвером для систем Windows NT и процессоров Intel x86.
2. Компонента должна реализовывать функциональность системного обработчика KiSystemService.
3. Компонента должна при каждом обращении к ней создавать системный вызов помимо того, который должен быть вызван для обработки прерывания SYSENTER.

5.2.2. Используемые технологии

Для разработки драйвера использовался набор средств разработки Windows Driver Kit 8.1. В качестве среды разработки использовалась Microsoft Visual Studio 12. Разработка велась на языке Visual C++, а также были использованы вставки на ассемблере.

5.2.2. Особенности реализации

Каждый драйвер для систем Windows NT имеет точку входа – DriverEntry. После запуска драйвера происходит подмена системного обработчика KiSystemSevice на собственный, что реализовано в функции HookSysenter, которая вызывается внутри DriverEntry (приложение 3). Функция HookSysenter загружает в регистр процессора указатель на функцию NewKiFastCallEntry (приложение 4), которая является новой реализацией обработчика прерывания SYSENTER. NewKiFastCallEntry имеет такую же функциональность, что и KiSystemService, единственное отличие состоит в том, что при её исполнении вызывается функция SystemCall,

которая позволяет создать свой собственный системный вызов (приложение 5).

На данный момент созданная драйвер позволяет исполнять системный вызов ZwCreateFile, который создаёт файл. Впоследствии он удаляется, поэтому такие действия никак не влияют на исполнение программы и опасаться некорректного результата работы нет оснований. Исполнение данных системных вызов происходит внутри функции SystemCall (приложение 5). Первым шагом происходит возврат полномочий обработчику KiSystemService, что необходимо для корректного вызова ZwCreateFile. После этого в регистр *eax* загружается указатель на данный системный вызов (индекс в таблице System Service Table). Далее в регистр *edx* подается указатель на адрес файла и вызывается прерывание с помощью инструкции int 0x2E. Системный обработчик KiSystemService вызывает ZwCreateFile, и управление возвращается пользователю посредством исполнения инструкции ret 0x2C. После этого обработчик вновь подменяется с помощью функции UnhookSystenter (приложение 6). Данная реализации впоследствии позволит создавать не только системный вызов ZwCreateFile, но и любой другой посредством просто подмены индекса необходимой функции в регистре *eax*.

7. Заключение

В результате проделанной работы были достигнуты следующие результаты.

1. Исследованы подходы к обфускации и деобфускации и изучен метод обфускации с помощью виртуальных машин.
2. Исследована технология системных вызовов в операционных системах Windows NT для процессоров Intel x86, а также найдена возможность логирования потока системных вызовов, которые выполняет операционная система при исполнении процесса.
3. Разработан подход, препятствующий обратной разработке программ, обфусцированных с помощью виртуальных машин, посредством анализа потока системных вызовов.
4. Реализована компонента, позволяющая создавать системные вызовы для генерации «шума», затрудняющего анализ потока системных вызовов (операционные системы Windows 2000 и выше, процессоры Intel x86).

В перспективе необходимо разработать алгоритм, который определял бы достаточное количество «шума» для эффективной обфускации, а также улучшить разработанный драйвер таким образом, чтобы генерируемый «шум» из системных вызовов был более разнообразный. Также необходимо изучить влияние обфускации на скорость исполнения обфусцированных программ и всецело протестировать функциональность созданного драйвера.

Данная работа ценна с исследовательской точки зрений, так как затрагивает малоизученную область анализа потока системных вызов и открывает широкие перспективы для дальнейшего совершенствования предложенного в работе подхода.

8. Список литературы

[1] Moy, R. (2000). A Case Against Software Patents. Santa Clara High Technology Law Journal, pp. 72-73.

[2] Christian Collberg, C. T. (2000). Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection. University of Arizona Computer Science Technical Report, pp. 5.

[3] Kevin Coogan, G. L. (2011). Deobfuscation of virtualization-obfuscated software: a semantics-based approach. Proceedings of the 18th ACM conference on Computer and communications security, pp. 1.

[4] Rofl, R. (2009). Unpacking virtualization obfuscators. 3rd USENIX Workshop on Offensive Technologies, pp. 1-2.

[5] Russinovich, M. (23 November 2004 г.). Inside the Native API. Получено из NetCode: http://www.netcode.cz/img/83/nativeapi.html

[6] Одеров Роман, Т. Е. (2011). Способы размещения своего кода в ядре ОС Microsoft Windows Server 2008. Материалы межвузовской научно-практической конференции "Актуальные проблемы организации и технологии защиты информации". Санкт-Петербург.

Приложение 1. Фрагмент шаблонного кода для функций Native API в библиотеке ntdll.dll

```
mov eax, [индекс функции в Native API]
lea edx, [указатель на параметры вызываемой функции]
int 0x2E
ret 0x2C
```

Приложение 2. Последовательность инструкции из потока системных вызовов

```
/* I1 */ mov eax, [ecx+edx]
/* I2 */ push eax
/* I3 */ call print
```

Приложение 3. Фрагмент кода созданного драйвера: точка входа DriverEntry

```
NTSTATUS DriverEntry(DRIVER_OBJECT *driverObject, UNICODE_STRING
*registryPath)
{
  driverObject->DriverUnload = DriverUnload;
  HookSysenter();
  return STATUS_SUCCESS;
}
```

Приложение 4. Фрагмент кода созданного драйвера: функция HookSysenter и функция NewKiFastCallEntry

```
static void HookSysenter()
{
  __asm
  {
    mov ecx, IA32_SYSENTER_EIP
    rdmsr
    mov oldKiFastCallEntry, eax
    mov eax, NewKiFastCallEntry
    xor edx, edx
    wrmsr
  }
}
static void  __declspec(naked) NewKiFastCallEntry()
{
  __asm
  {
    pushad
    pushfd
    mov ecx, 0x23
    push 0x30
    pop fs
    mov ds, cx
    mov es, cx
    call SystemCall
    popfd
    popad
```

```
    jmp [oldKiFastCallEntry]
  }
}
```

Приложение 5. Фрагмент кода созданного драйвера: функция SystemCall

```
static void __stdcall SystemCall()
{
  UnhookSysenter();

  __asm
  {
    mox eax, 0x52
    lea edx, [esp+4]
    int 0x2E
    ret 0x2C
  }

  HookSysenter();
}
```

Приложение 6. Фрагмент кода созданного драйвера: функция UnhookSysenter

```
static void UnhookSysenter()
{
  __asm
  {
    mov ecx, IA32_SYSENTER_EIP
    mov eax, oldKiFastCallEntry
    xor edx, edx
    wrmsr
  }
}
```