УДК 004.49, 004.453, 004.451.35

А. А. Переберина[1,2], *А. В. Костюшко*[1,2]

[1]Московский физико-технический институт (государственный университет)
[2]ООО «Акронис»

Разработка инструментария для динамического анализа вредоносного программного обеспечения

Рассматривается разработка инструментов для глубокого динамического анализа вредоносного программного обеспечения. Наша основная идея — обеспечить полный контроль над исполнением образца программного обеспечения на тестовом сервере. Для этого мы отделяем код приложения от системного кода путём составления карты памяти и контроля над доступом к её участкам. Модуль, осуществляющий глубокий динамический анализ, следит за внутренними событиями исследуемого образца, при этом используя инвазивные методы исследования, такие как перехват вызовов системных функций или патч исполняемого файла. В работе описаны ключевые стадии создания базового прототипа модуля глубокого динамического анализа, а также некоторые технические идеи решения проблем анализа многопоточных приложений, маскировки аналитических инструментов и снижения нагрузки на операционную систему. В будущем авторы собираются применить разрабатываемые инструменты для детектирования вредоносной активности и определения подозрительных паттернов поведения с использованием модели машинного обучения.

Ключевые слова: вредоносное программное обеспечение, динамический анализ вредоносного программного обеспечения.

A. A. Pereberina[1,2], *A. V. Kostyushko*[1,2]

[1]Moscow Institute of Physics and Technology (State University)
[2]Acronis LLC

Approach to dynamic malware analysis based on separation of the system code from the application code

This paper discusses the development of deep dynamic malware analysis tools. The main idea is to provide the total control over malware sample execution by a test server. The proposed approach is to separate the application code from the system code using memory pages access control. The Deep Analysis Module of monitors internal target process events using invasive methods such as the system call hook or executable file patch. This research includes the key creation stages of the prototype with basic functionality and some technical ideas of solving problems such as analysis of multithreaded application, cloacking of analytical tools presence and mitigation of the performance degradation of the operation system. The Deep Analysis Module can be used to detect malware activity and determine suspicious behaviour patterns using the Machine Learning model.

Key words: malware, dynamic malware analysis.

© Перебрина А. А., Костюшко А. В., 2018
© Федеральное государственное автономное образовательное учреждение высшего образования «Московский физико-технический институт (государственный университет)», 2018

1. Введение

Рассматривается разработка инструментов глубокого динамического анализа вредоносного программного обеспечения (ToolChain), которые будут работать на тестовом сервере программно-аппаратного комплекса Sandbox, предназначенного для запуска и анализа вредоносного ПО. Проектирование и структура Sandbox были рассмотрены нами в статье [1]. Разрабатываемые нами инструменты предназначены для анализа вредоносного ПО, ориентированного на операционные системы семейства Windows. Главным образом наша программа исследований сосредоточена на шифровальщиках-вымогателях (crypto-ransomware), однако инструментарий ToolChain может быть применён для различных типов вредоносного ПО.

Комплекс Sandbox предназначался для запуска вредоносного программного обеспечения и сбора данных о его поведении. Собранные данные подавались на вход модели машинного обучения, осуществляющей классификацию программ на потенциально представляющие угрозу (suspicious) и не представляющие угрозы (clean). До этого мы собирали события, доступные для внешнего (неинвазивного) мониторинга (файловая, реестровая активность и т. д.), видимые такими инструментами мониторинга, как Process Monitor [2]. Инструментарий ToolChain предназначен для сбора внутренних событий исследуемого процесса и использует инвазивные методы исследования, такие как перехват вызовов системных функций или патч исполняемого файла. Эти события также могут использоваться для обучения предсказывающей модели, но уже другого рода. Мы расширили своё видение роли машинного обучения, о чём будет упомянуто в данной статье.

С момента появления первых экземпляров вредоносного программного обеспечения увеличилось видовое разнообразие вредоносного ПО, его сложность, количество новых образцов и скорость их появления, а также масштабы угроз. Традиционный антивирус, ориентирующийся на проверку сигнатур по базам данных, которые пополнялись экспертом-аналитиком, не может противостоять вновь появляющимся, неизвестным ранее угрозам. Поэтому появляются инструменты статического анализа исполняемого файла и инструменты динамического анализа вредоносного ПО, исследующие поведение образца во время его исполнения. Усложняются также методы борьбы с антивирусными сканерами со стороны разработчиков вредоносного ПО, что вынуждает аналитические инструменты применять техники маскировки.

Мы провели исследование существующих подходов к динамическому анализу вредоносного программного обеспечения. Традиционные методы используют перехват вызовов функций, анализ параметров. Есть техника W$\oplus$X, которая используется для преодоления упаковщиков, в ней применяется модификация прав доступа к странице. Инструменты, модифицирующие части исследуемого исполняемого файла, могут скрывать свои изменения, объявляя соответствующие страницы не присутствующими и при обращении к ним предъявляя приложению теневые не модифицированные копии. Такие инструменты работают, как правило, на уровне эмулятора или гипервизора. Наше решение опирается на существующие техники и предлагает новый подход.

Основной подход ToolChain — обеспечение полного контроля над исполнением образца вредоносного программного обеспечения на тестовом сервере. Для этого мы отделяем код приложения от системного кода. Это достигается путём составления карты памяти и контроля над доступом к её участкам. Нами был предложен метод обеспечения контроля в случае как однопоточного, так и многопоточного приложения, для чего разработан специальный механизм. Применяются также различные подходы маскировки аналитических инструментов, как встроенные в ToolChain, так и в качестве отдельных компонентов.

На тестовых серверах Sandbox запуск происходит на физическом аппаратном обеспечении, при этом аналитические инструменты работают на уровне пользователя или ядра операционной системы. ToolChain находится на стадии прототипа, разработан для 64-битной Windows архитектуры, содержит компоненты уровня пользователя.

2. Терминология

В данной статье используются следующие понятия:

1) Глубокий динамический анализ (deep dynamic analysis) — динамический анализ (происходящий во время исполнения исследуемого образца), использующий инвазивные методы исследования, такие как перехват, патч и т. д.

2) Импортируемые функции, импорты (imports, import functions) — функции, предоставляемые динамически загружаемыми модулями (например, системными библиотеками), используемые приложением. Таблица IAT (Import Address Table) содержит адреса импортируемых функций, используется для их вызова, заполняется загрузчиком после размещения модуля в адресном пространстве процесса.

3) Системное API (system API) — функции системных библиотек Windows (Windows API), нижележащих системных библиотек (Native API, ntdll.dll), а также системные вызовы (system calls).

4) Перехват, хук (hook) — технология, позволяющая подменить код, исполняющийся при вызове какой-либо функции/модуля/компонента. Хуком мы называем код на C, который реализует изменённую логику. Передача управления на код с изменённой логикой осуществляется благодаря использованию патча. Перехват может осуществляться как и с помощью патча адреса в IAT, так и с помощью патча тела функции или подмены системной DLL на свою реализацию.

5) Патч (patch) (бинарный) — модифицированные бинарные данные, замещающие оригинальные, а также процесс модификации бинарных данных. Здесь целью модификации является передача управления на хук, при этом для вызова изменённой логики может использоваться стаб.

6) Стаб (stub) — код-переходник, написанный на ассемблере, применяется для передачи управления на код хука.

7) Карта памяти — состояние виртуальной памяти процесса при разделении на области памяти одной категории с одинаковыми правами доступа, а также структура, описывающая это состояние.

    - Открытая часть карты памяти — страницы виртуальной памяти процесса, доступ к которым не был искусственно запрещён путём объявления данного набора страниц, не присутствующими в физической памяти.
    - Закрытая часть карты памяти — страницы виртуальной памяти процесса, доступ к которым был искусственно запрещён путём объявления данного набора страниц, не присутствующими в физической памяти.

8) Компьютер конечного пользователя (End Point) — персональный компьютер пользователя продукта с установленным программным обеспечением.

9) Песочница (Sandbox) — программно-аппаратный комплекс для запуска и анализа вредоносного программного обеспечения, в котором обеспечена безопасность и автоматизация. Состоит из управляющих и тестовых серверов.

10) Тестовый сервер — сервер Sandbox, предназначенный для запуска вредоносного программного обеспечения.

11) События типа «A» (Events A) — события операционной системы, видимые средствам внешнего мониторинга (такие как файловые, реестровые операции, создание нитей и процессов). Собираются как на компьютере конечного пользователя, так и на тестовом сервере Sandbox.

12) События типа «B» (Events B) — внутренние события процесса, доступные средствам глубокого анализа. Собираются на тестовом сервере Sandbox. Сбор данных событий может приводить к деградации производительности операционной системы.

3. Обзор существующих решений

Аналитические инструменты, работающие во время выполнения программы, называются инструментами динамического анализа. Мы провели исследование существующих инструментов и техник. Статья [3] подробно освещает подходы к анализу вредоносного программного обеспечения — упомянут и статический анализ, и различные подходы к созданию тестовой среды для запуска образцов, но основная часть посвящена техникам динамического анализа. Рассмотрим основные механизмы динамического анализа.

1) Мониторинг вызываемых функций (Function Call Monitoring).

   Сюда относится перехват вызовов Windows API (системные библиотеки Windows), Windows Native API (системная NT библиотека ntdll.dll) и системных вызовов (syscall). При этом анализ может осуществляться до и/или после перехода к оригинальной вызываемой функции.

2) Анализ параметров функций (Function Parameter Analysis).

   Анализ и возможная модификация передаваемых системной функции аргументов. Здесь также могут быть детектированы операции, осуществляющиеся с теми же объектами, например, файловыми дескрипторами (так называемый объекто-центричный — object centric — анализ).

3) Отслеживание потока информации (Information Flow Tracking).

   Данные помечаются метками (taint-label), эти метки передаются другим данным, распространяясь через зависимости данных (например, при присвоении значений).

4) Отслеживание последовательности машинных инструкций (Instruction Trace).

5) Мониторинг добавления на автозапуск (ASEPs: Autostart extensibility points).

Антивирусное ПО детектирует вредоносное программное обеспечение, в том числе методами динамического анализа. Вредоносное ПО стремится защититься от аналитических средств, не показывая вредоносное поведение, используя различные техники анти-ревёрсинга и анти-отладки. Для маскировки от антивирусов вредоносное ПО применяет средства детектирования запуска в тестовой среде, присутствия в системе аналитических инструментов. Аналитические инструменты, в свою очередь, маскируют факт анализа, что приводит к гонке вооружений между экспертами-аналитиками и хакерами. При этом присутствие самих методов анти-ревёрсинга и анти-дебаггинга может быть задетектировано антивирусами и расценено как подозрительное или вредоносное поведение. Рассмотрим некоторые механизмы, применяемые обеими сторонами. Подробнее об этом можно прочитать в статьях [3] и [4].

1) Самомодифицирующийся код и упаковщики.

   В эту категорию входят такие методы, как обфускация, шифрование, рекурсивная упаковка, полиморфические и метаморфические бинарные файлы. Для борьбы с этими методами аналитические инструменты используют, например, W$\oplus$E технологию. Алгоритм описывается следующим образом:

   - На начальном этапе все страницы памяти анализируемого процесса помечаются исполняемыми и доступными для чтения, но не для записи.

- Когда процесс пишет в память, происходит исключительная ситуация отказа доступа к странице памяти (page fault). Когда система ловит и обрабатывает исключение на запись, она устанавливает защиту страницы в чтение/запись (не исполнение, NX).
- Когда распаковщик завершил работу, он передаёт управление исходному бинарному коду. Это вызывает нарушение прав доступа на исполнение (NX page fault). В этот момент в памяти находится дамп (dump) не модифицированного кода вредоносного ПО.

При этом со стороны аналитических инструментов важно маскировать модификацию прав доступа, т. к. вредоносное программное обеспечение может обнаружить применение этого подхода, запрашивая настройки защиты страницы.

2) Детектирование аналитических инструментов.

Некоторое вредоносное ПО детектирует запуск в эмуляторе или виртуальной машине и прекращает свою работу. Для этого могут быть использованы следующие соображения.

- Анализ аппаратного обеспечения.

  Виртуальные устройства в виртуальных машинах часто могут быть опознаны по характерным особенностям, артефактам гипервизора (например, сетевой адаптер «pcnet32» у гипервизора VMware, виртуальный диск «QEMU HARD-DISK» у гипервизора KVM).
- Анализ среды исполнения.

  Артефакты в среде анализируемого процесса, которые могут выдать запуск под аналитическими инструментами, например, присутствие отладчика внутри процесса.
- Анализ внешних приложений.

  Наличие известных приложений мониторинга в операционной системе — отладчики, средства для мониторинга файловых и реестровых операций (например, Process Monitor).
- Особенности поведения.

  Исполнение привилегированных инструкций отличается на реальном хосте и в виртуальной среде. Вредоносное ПО может заметить задержки в исполнении инструкций.

Для предотвращения детектирования аналитические инструменты используют руткит-подходы. Например, фильтруют результаты вызовов системного API, которые возвращают список процессов в операционной системе или перечисляют загруженные в адресное пространство процесса модули.

Аналитические системы, которые изменяют анализируемый образец во время выполнения в памяти, могут скрывать эти изменения, сохраняя копию не модифицированных страниц и помечая изменённые страницы не присутствующими в физической памяти. Поскольку вредоносное ПО может проверять свою целостность (integrity), например, путём вычисления хеша, обработчик исключения доступа к странице (page fault handler) может быть настроен для предъявления не модифицированной версии страницы при запросе приложения. Таким образом, проверка целостности будет пройдена.

В системах, которые напрямую изменяют записи таблицы страниц (Page Table Entry, PTE) текущего процесса, модификации могут быть замаскированы путём сохранения теневых копий структур таблицы страниц. Кроме того, сами таблицы страниц,

содержащие PTE, могут быть помечены не присутствующими. Если анализируемый процесс обращается к модифицированной PTE, это вызовет page fault, а обработчик вернёт теневую версию этой записи.

Некоторые аналитические методы устанавливают trap флаг в регистре EFLAGS, чтобы провести детальный («мелкомодульный», fine-grained, т. е. на уровне машинной команды) анализ. Вредоносное ПО может обнаружить такие подходы, читая регистр EFLAGS и проверяя определённый бит. В этом случае для маскировки аналитические инструменты могут хранить теневую копию регистра EFLAGS для предъявления вредоносному процессу.

3) Логическая бомба.

   В программе может быть скрыта логическая бомба, которая активируется при наступлении какого-либо события. Сюда относятся временные бомбы, ожидание пользовательских действий или поступления команды от владельца бота. Это затрудняет статический и динамический анализ, т. к. вредоносный код может даже не присутствовать напрямую в исполняемом файле. Например, он может быть зашифрован ключом, который может быть получен только во время исполнения в форме команды от владельца бота.

4) Анализ производительности.

   Наличие аналитических инструментов может привести к деградации производительности операционной системы, которая может быть замечена вредоносным ПО. Методы сокрытия (патч инструкции RDTSC, замедление времени в виртуальной среде) могут нарушить взаимодействие вредоносного ПО с другими компонентами в сети. Также низкая производительность влияет на количество анализируемых образцов в единицу времени (пропускную способность аналитической системы).

4. Глубокий динамический анализ

Инструменты глубокого анализа получают доступ к так называемым внутренним событиям процесса, используя инвазивные технологии, такие как патчи, хуки, инжекцию DLL. Наша основная идея состоит в том, чтобы обеспечить полный контроль над исполнением образца на тестовом сервере. Мы хотим отделить исполнение кода приложения от системного кода, осуществлять мониторинг всех системных вызовов и передачи управления другими способами. Общий взгляд на ToolChain представлен на рис. 1. Аналитические инструменты работают на уровне пользователя или ядра операционной системы. В данной статье будут подробно рассмотрены компоненты режима пользователя. Мы также планируем разработку kernel-компонентов анализатора, но постараемся как можно дольше оставаться в режиме пользователя и сначала покрыть все доступные в нём способы анализа и маскировки.

Для реализации в рамках разработки инструментов глубокого динамического анализа, осуществляющих мониторинг внутренних событий приложения (событий типа «B», в то время как внешние события мы называем событиями типа «A»), запущенного на тестовом сервере Sandbox, нами были выбраны следующие техники:

1) Управление доступом к страницам памяти процесса с контролем передачи управления из кода приложения в системный код и обратно, а также контролем всех динамически сгенерированных областей кода.

2) Мониторинг вызовов функций с логированием вызовов системного API (Windows API, Native API, системные вызовы), при котором происходит сбор событий типа «B».

3) Перехват отдельных функций (специальные хуки) с анализом параметров, пре- и/или пост-анализом, производящийся как для цели мониторинга, так и для цели контроля доступа к коду в памяти и сокрытия присутствия аналитических инструментов.

Рис. 1. Инструменты глубокого динамического анализа

Мониторинг вызываемого системного API осуществляется при помощи перехвата функций системного API на основе техники инжектирования в адресное пространство приложения динамически загружаемой библиотеки (DLL) [5]. Инжектируемая DLL реализует следующий функционал:

1) Осуществляет парсинг таблицы импортов PE-файла [6] и составляет карту импортов приложения.

2) Осуществляет перехват системных функций. Для этого применяются следующие подходы:

   - Патч таблицы импортов приложения: адреса всех импортируемых функций заменяются на адреса автоматически генерируемых стабов, которые передают управление функции, представляющей собой универсальный логер, осуществляющей логирование вызываемых системных функций и передачу им управления.
   - Специальные виды перехвата путём модификации кода системных библиотек с анализом параметров, пре- и/или пост-анализом, возможным изменением функциональности.

3) Осуществляет составление карты памяти приложения, куда входят секции кода приложения и системных DLL.

4) Выключает системные библиотеки, объявляя страницы с их кодом не присутствующими (PAGE_NO_ACCESS).

5) Регистрирует (используя Vectored Exception Handling или VEH механизм [7]) обработчик исключения доступа к странице (page fault) EXCEPTION_ACCESS_VIOLATION, который вновь объявляет не присутствующие страницы присутствующими и выключает другую часть страниц кода, при этом

логируя переключение из системного кода в код приложения и обратно. Обработчик исключения EXCEPTION_ACCESS_VIOLATION применяет различные политики в зависимости от того, что послужило причиной возникновения исключительной ситуации. Если это была попытка доступа на исполнение, то применяется механизм переключения карты памяти (Access Switch). Если это была попытка доступа на запись, решение принимается на основании дизассемблирования записываемых инструкций (задетектирована попытка патча). Если это была попытка доступа на чтение, для разрешения однократного чтения применяются следующие механизмы:

- Обработчик EXCEPTION_ACCESS_VIOLATION возобновляет доступ к целевой области и устанавливает флаг trap в регистр EFLAGS [8]. Регистрируется обработчик исключения EXCEPTION_SINGLE_STEP, где доступ к целевой области снова запрещается.
- Обработчик EXCEPTION_ACCESS_VIOLATION возобновляет доступ к целевой области и заменяет следующую за вызвавшей исключительную ситуацию инструкцию на INT 3. Регистрируется обработчик исключения EXCEPTION_BREAKPOINT, где доступ к целевой области снова запрещается.

Мы применяем VEH-механизм вместо, например, реализации отладчика, поскольку детектирование отладчика является распространённым приёмом, применяемым вредоносным ПО [3, 4]. Вредоносное ПО также может использовать различные методы, препятствующие отладке, например, «отлаживать само себя» с помощью Debug API.

4.1. Патч импортов

Мы парсили таблицу импортов PE-файла и составляли карты импортов. Для прототипа мы работаем только с нормальными импортами, отложенные (delay-loaded) и привязанные (binding) импорты мы пока не рассматриваем. В дальнейшем планируется включить их в рассмотрение. После обработки импортов составлялись отдельные карты с информацией о функциях, обращения к которым мы собирались перехватывать.

Прежде всего, важно было определить, что импорт является функцией, и не патчить адреса импортируемых переменных. Для этого мы проверяли наличие PRUNTIME_FUNCTION [9] в секции .pdata [10] для раскручивания стека вызовов (stack unwinding) при обработке исключений, т. к. разработка велась для 64-битной платформы. Такой подход не работал для перенаправленных (redirected) импортов (импортируемых системными библиотеками, в свою очередь, из других, нижележащих, системных библиотек) и с листовыми (leaf) функциями, которые не используют стек (например, такая функция может возвращать значение из регистра системного пользования). Листовую функцию можно идентифицировать по секции, которой принадлежит импорт (например, .text или .data).

Мы столкнулись с перенаправлениями 2-х типов:

1) Перенаправление вида dll_name.function_name, которое позволяет проследить, какая функция будет вызвана на самом деле. Мы парсили и хранили эту информацию.

2) Перенаправление с использованием прыжка (jmp) на IAT, который можно детектировать по опкоду 0x48 0xff 0x25. В таком случае нужно было вычислить адрес, по которому осуществляется прыжок (относительный), и проверить по таблице импортов данной DLL, какая функция будет вызвана на самом деле. Эта информация также сохранялась.

Также мы столкнулись с виртуальными DLL, реализованными механизмом ApiSetSchema, который необходимо рассматривать отдельно. Подробнее о нём можно

прочитать в [11, 12, 13]. Для работы с ApiSet мы составляли отдельные карты, в которых описывалось отображение виртуальных DLL в логические. В будущем также планируется специальная обработка механизма SHIMS [14, 15], обеспечивающего обратную совместимость для приложений, использующих устаревший системный API.

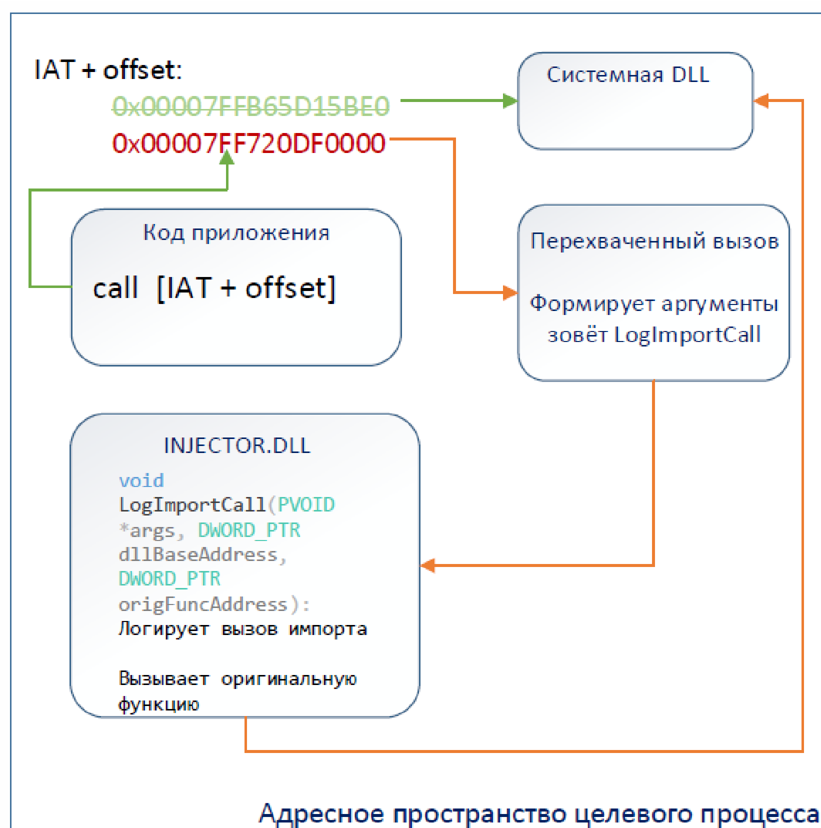


Рис. 2. Перехват функции через патч IAT

Техники бинарного патча широко известны, поэтому мы не будем вдаваться в подробности. Перехват системных функций может осуществляться как с помощью механизма патча IAT (Import Address Table) (см. рис. 2), так и с помощью патча тела импортируемой функции (см. рис. 3) или подмены системной DLL на свою реализацию. На рис. 2 патч модифицирует адрес в IAT, при этом происходит передача управления на стаб, формирующий аргументы для универсального логера импортов. Универсальный логер представляет собой хук импорта, после записи в лог передающий управление оригинальному коду. Помимо универсального перехвата для целей мониторинга, могут применяться специальные хуки для дополнительного обеспечения контроля. Рисунок 3 демонстрирует пример специального хука, осуществляющегося путём патча кода функции в системной библиотеке. Здесь изображён хук Native API, это низкоуровневое API системных вызовов к ядру операционной системы семейства Windows NT, реализованное в ntdll.dll, которое используют различные системные библиотеки. Управление будет передано в наш хук независимо от того, из какого модуля будет вызвана данная функция. После анализа и/или модификации параметров в данном примере осуществляется оригинальный системный вызов.

Основные группы функциональности для перехвата

Перехват группы API, реализующей какую-либо функциональность, представляет собой отдельный модуль — например, модуль контроля памяти или модуль контроля течения времени для исследуемого образца. Ниже перечислены основные API, реализующие груп-

пы функциональности, которые необходимо перехватить для того, чтобы предложенный нами метод контроля над исполнением приложения работал.

1) Для целей контроля виртуального адресного пространства, в том числе для исполняемых страниц памяти:

   - Низкоуровневое API — перехват *NtQueryVirtualMemory* для сокрытия областей инжектированного кода, перехват *NtAllocateVirtualMemory* и *NtProtectVirtualMemory* с установкой PAGE_EXECUTE для отслеживания динамически добавляемых областей кода приложения и т. д.
   - Высокоуровневое API — перехват *CreateHeap*, *malloc*, *VirtualProtect* и т. д.

2) Для целей контроля переключений доступных регионов памяти — перехват функций VEH (*AddVectoredExceptionHandler*, *AddVectoredContinueHandler*), SEH (*RtlAddFunctionTable*, *RtlInstallFunctionTableCallback*, *SetUnhandledExceptionFilter*, *RtlCaptureContext*), Debug API (*DebugActiveProcess*, *SetThreadContext*, *Wow64SetThreadContext* и др.).

3) Для работы с многопоточными приложениями и приложениями, порождающими другие процессы — перехват функций *NtSuspend/ResumeProcess/Thread*, *NtQuery/SetInformationProcess/Thread*, *NtCreateThread/Process* и т. д.

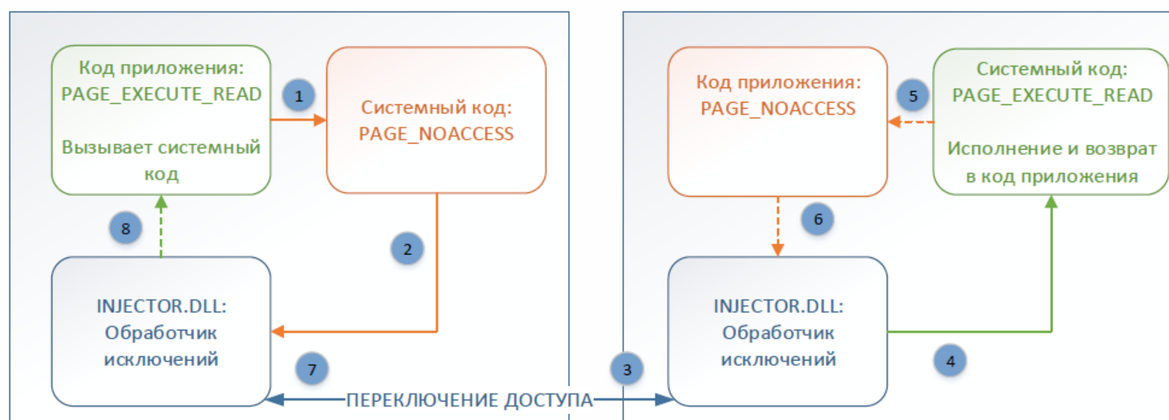
Перечисленные группы API составляют необходимую «холостую» нагрузку метода. Также есть группы API, контроль которых представляет собой полезную нагрузку. Например, перехват *NtWriteProcessMemory* для детектирования попыток инжектирования в другие процессы со стороны приложения, перехват CryptoAPI для детектирования шифрования и т. д.

Рис. 3. Перехват функции через патч тела

4.2. Контроль памяти

Контроль над работой приложения осуществляется следующим образом. Память делится на 2 части — код приложения и код системных библиотек. Часть карты памяти «закрывается» путём объявления этих страниц кода не присутствующими. В один момент «открыта» только одна часть карты памяти. Когда работает код приложения, системные библиотеки не доступны. При передаче доступа в системный код произойдёт переключение карты памяти: при обращении к таким страницам произойдёт исключительная ситуация типа page fault, после чего вызовется зарегистрированный нами обработчик исключений; обработчик исключений зафиксирует в лог запрос доступа на исполнение и переключит карту памяти, т. е. сделает доступной другую часть карты, при этом закроет первую. Симметричный механизм позволит нам узнать о передаче контроля исполнения обратно.

В этой схеме была трудность с доступом на чтение, а не для передачи управления. Чтобы разрешить чтение из закрытой части карты памяти, мы устанавливаем TF (trap flag) в EFLAGS и разрешаем доступ ровно на одну инструкцию, применяя тот же механизм обработки исключений. В случае доступа на запись процесс терминируется, т. к. задетектирована попытка патча кода и для дальнейшего анализа необходим дизассемблер. Механизм переключения карты памяти при исполнении представлен на рис. 4.



Комментарий к рисунку: Когда работает код приложения, код всех системных библиотек объявлен не присутствующим в памяти. Когда происходит передача управления коду системной библиотеки (1), возникает исключительная ситуация (EXCEPTION_ACCESS_VIOLATION) и вызывается зарегистрированный при инжектировании обработчик исключений (2). Обработчик исключений инспектирует причину возникновения исключительной ситуации (в рассматриваемом случае DEP_VIOLATION) и производит переключение карты памяти (3). Теперь страницы кода приложения объявлены не присутствующими, а системный код доступен. Управление передаётся системному коду (4), который может свободно вызывать другой системный код. При передаче управления назад в код приложения (5) возникает исключительная ситуация (6) по причине доступа на исполнение, что приводит к обратному переключению карты памяти (7) и возврату в код приложения (8). Независимо от этого обрабатываются случаи доступа к закрытой части карты на чтение и запись.

Рис. 4. Переключение карты памяти в однопоточном случае

Данный подход позволит не потерять контроль в случаях, если код вредоносного ПО попытается скопировать часть кода из системной библиотеки и передать управление в середину системной функции. Будут также задетектированы случаи, когда код приложения получает управление посредством обратного вызова из системной функции (например, *EnumFonts* или *CreateThread*). Некоторая проблема возникает в том случае, когда вредоносное ПО напрямую вызывает syscall/sysenter. Есть различные способы её решения.

Детектировать такое поведение можно путём сравнения счётчиков вызова syscall: всех системных вызовов (статистика ядра ОС) и системных вызовов из ntdll.dll (требует полной пересборки ntdll.dll с добавлением подсчёта статистики вызовов). Если для исследуемого образца было задетектировано расхождение этих величин, то он использует syscall. Чтобы образец мог это делать, ему нужно знать номера API, т. е. соответствие между номером syscall и функцией Native API, разное в каждой версии библиотеки. Образец для этого может сканировать ntdll.dll, что также можно задетектировать. Наконец, можно искать инструкции syscall/sysenter с помощью дизассемблера.

При работе с переключением карты памяти мы столкнулись с рядом трудностей:

1) Многопоточность.

   Многопоточные приложения делят адресное пространство, при этом может быть ситуация, когда часть потоков исполняет код из одной части карты, в то время как другая часть — из другой. Работа нескольких потоков может привести к неопределённому состоянию карты памяти и логу, на который не следовало бы полагаться при анализе. Мы решили остановиться на однопоточном прототипе, для многопоточных приложений нами был разработан особый механизм запуска приложения под нашим контролем, который будет описан далее.

2) Использование системных DLL.

   Мы не могли использовать системные DLL в коде обработчика исключений (чтобы не допустить бесконечной рекурсии) и универсального логера импортов (чтобы сделать injector.dll невидимой при переключениях, иначе полученный лог был бы «изменён под влиянием проводимых измерений»). На стадии разработки Proof Of Concept мы могли выключать все системные библиотеки, кроме kernel32.dll, kernelbase.dll и ntdll.dll. Для того чтобы отказаться от вызовов функций kernel32.dll и kernelbase.dll, мы перешли на Native API, вызывая нижележащие функции ntdll.dll напрямую, а также реализовали минимальный C RunTime. Чтобы отказаться от ntdll.dll, мы стали вызывать системные вызовы напрямую через sysenter. Однако мы не смогли полностью отказаться от ntdll.dll, так как в ней реализована обработка исключений. Это требует изменения в ядре ОС и появления kernel-компонента анализатора. На данном этапе, когда разработка ведётся в режиме пользователя, предлагается отключить все страницы ntdll.dll, не связанные с обработкой исключений.

4.3. Производительность

Генерация page fault'ов может привести к деградации производительности операционной системы. Чтобы снизить это влияние, можно напрямую передавать управление «свитчеру» карты памяти из перехваченных вызовов системного API. Также для улучшения производительности и маскировки аналитических инструментов могут применяться TLB cache split [16] или ключи защиты для страниц пользователя от Intel (PKRU регистр) [17]. Эти подходы требуют добавления ядерных компонентов анализатора.

Важными являются вопросы ускорения процедуры переключения карты памяти. Сейчас мы используем *ZwProtectVirtualMemory*, изменяя доступ постранично для страниц 4Kb или 2Mb. Вместо постраничного выключения памяти можно воспользоваться аппаратной поддержкой от Intel. В параграфе 5.13.2 руководства для разработчиков от Intel [18] говорится о том, что инструкции не могут исполняться процессором, если бит XD (Execution Disable) стоит хотя бы на одном уровне структур описания страниц, в которые входит данная страница. Таким образом, XD бит в элементах таблицы Page Directory Pointer (PDP) позволит отключать гигабайтные регионы, а используя XD бит в таблице страниц 4 уровня (Page Map Level 4, PML4), можно отключить регион памяти в 512Gb. Этот подход требует поддержания структуры памяти, при которой код приложения и код системных библиотек расположены в разных регионах 4 уровня (сюда входит контроль

ZwMapViewOfSection и т. д.). При применении такого механизма нежелателен конфликт с кодом операционной системы. Поскольку нет API для установки XD бита в структурах страниц высоких уровней, необходимо оперировать с XD битом напрямую, при этом вмешиваясь в работу Менеджера Памяти (Memory Manager). Таким образом, чтобы не было конфликтов, нужно, чтобы Memory Manager Windows не трогал XD бит. В этом смысле предпочтительнее использовать 4 уровень (PML4), так как, насколько нам известно, в данный момент Memory Manager не отгружает на диск страницы 4 уровня. Этот вопрос требует дополнительного исследования.

Альтернативно можно использовать более сложную технику, основанную на аппаратной поддержке виртуализации. В данном подходе мы используем для анализируемого процесса гостевое страничное преобразование. При этом гостевая операционная система имеет доступ к аппаратным ресурсам, гостевые и реальные виртуальные адреса совпадают, но бит X в Extended Page Table (EPT) позволит запретить чтение в кэш инструкций (fetch). Эта техника также может помочь бороться с деградацией производительности в многопоточном случае, о котором пойдёт речь далее.

В перспективе также можно будет воспользоваться новой технологией CET [19], которая, к сожалению, на текущий момент не представлена ни в одном процессоре. Данная технология предназначена для борьбы с уязвимостями к ROP (Return Oriented Programming), в неё входит концепция Shadow Stack, а также Indirect Branch Tracking. Последний механизм может быть крайне полезен в случае, когда мы разнесли код приложений и системных библиотек в адресном пространстве более чем на 4Gb, обеспечив тем самым переходы между ними indirect. В этом случае детектирование переходов возможно без манипуляции с битами доступа к страницам. Данный подход может не работать с ПО, осведомлённым об этом механизме и использующим способы обхода Indirect Branch Tracking.

4.4. Маскировка

Чтобы сделать инжектируемую DLL невидимой при переключениях карты памяти, необходимо отказаться от использования системных библиотек и C RunTime библиотеки (вызывающей системные DLL). При этом необходимо сделать невидимой только ту часть инжектируемой DLL, которая работает во время работы приложения. Инжекция осуществляется классическим способом с использованием *CreateRemoteThread* и *LoadLibrary*. Адрес kernel32.dll получается из указателя Ldr в PEB процесса (Process Environment Block), который заполняется во время инициализации режима пользователя и равен NULL при создании процесса в приостановленном состоянии. Чтобы PEB.Ldr был проинициализирован, необходимо начать исполнение, поэтому при инжектировании DLL мы патчим точку входа приложения (Entry Point) на бесконечный цикл (jmp $-2) и запускаем процесс. Инициализационная часть DLL исполняется в то время, как точка входа приложения пропатчена, а контроль над памятью ещё не осуществляется, и поэтому может свободно использовать системные библиотеки. Независимой от системных библиотек должна быть только та часть инжектируемой DLL, которая исполняется во время работы приложения, когда контроль передан в его оригинальную точку входа после завершения инициализационной части (при этом точка входа пропатчена обратно к исходному коду).

Таким образом, можно разделить инжектируемую DLL на две части, где вторая часть полностью независима от системных библиотек и вызывает системные вызовы ОС напрямую (например, через инструкцию sysenter). При этом инициализационная часть копирует необходимую в момент исполнения под анализом часть кода в адресное пространство исследуемого процесса, после чего инжектированная DLL выгружается. С помощью директивы #pragma alloc для run-time части инжектируемой DLL можно создать отдельные секции кода и данных (.text2 и .data2). При таком подходе вредоносное ПО не обнаружит в списке загруженных модулей инжектированной DLL. Альтернативно, для сокрытия инжектированного модуля может применяться перехват системного API, возвращающего список загруженных DLL, но этот подход сложнее и не работает в случае, если процесс сам достаёт

список модулей, используя PEB. Также для маскировки аналитических инструментов в системе работают 2 дополнительных модуля — модуль контроля времени (Time Machine Module) и модуль контроля памяти (Memory Control Module).

В перспективе планируется от модификации IAT перейти к патчу тела функций. При этом необходимо скрывать сделанные изменения таким образом, чтобы исполнялся модифицированный код, а при чтении нельзя было увидеть модификацию (возвращались бы оригинальные инструкции). Это важно, например, для проверок целостности. Маскировать изменения также можно несколькими способами. Можно использовать TLB cache split механизм, при котором модифицированные данные хранятся только в кэше инструкций, а при чтении возвращается оригинальный код системной библиотеки. С помощью расширенного механизма защиты страниц Intel (PKRU) можно применять уровень доступа, при котором будет разрешён fetch инструкций, а при чтении возникнет page fault. В свою очередь, через виртуализацию можно будет скрыть артефакты в CPUID, указывающие на то, что данный механизм включён. Альтернативно можно применять дизассемблер и препятствовать исполнению непривилегированных инструкций RDPKRU/WRPKRU.

4.5. Многопоточность

Наша концепция предполагает, что в один момент исполняется либо код приложения, либо системный код. Ситуация, в которой исполняется код обоих классов, недопустима. При многопоточном приложении у нас есть два пула потоков — исполняющие системный код и исполняющие код приложения. В каждый момент разрешено исполнение либо одного пула, либо другого. При этом нити из не разрешённого пула должны быть в остановленном состоянии либо в состоянии ожидания. После этого возникает вопрос: какой механизм используется для разделения классов потоков? Здесь могут применяться различные подходы. Разделить классы можно либо посредством адресного пространства (это требует дуплицирования VAD и введения концепции нитей-близнецов), либо по квантам времени (это требует реализации планировщика задач режима пользователя внутри процесса). Опишем оба этих подхода более подробно.

Для работы с многопоточными приложениями нами был разработан механизм, при котором приложение разделяется на два процесса-близнеца. В одном из них выключен и никогда не включается весь системный код, в другом выключен и никогда не включается код приложения. Таким образом, вместо переключения прав доступа к страницам памяти мы переключаем исполнение нити-близнеца в соседнем процессе, сохраняя состояние и передавая необходимые аргументы. То есть, когда нить в одном процессе вызывает системный код, мы будем передавать управление нити-близнецу в соседнем процессе (ожидающей или приостановленной), а исполнение текущей нити приостанавливать (уходить в ожидание). Данный процесс изображён на рис. 5. В приведённом примере треды 2…N исходного процесса исполняют код приложения, в это время их треды-близнецы 2…N приостановлены либо ожидают в обработчике исключений. Пусть тред 1 вызывает системный код. При этом происходит исключительная ситуация EXCEPTION_ACCESS_VIOLATION и вызывается обработчик исключений (1). Здесь происходит проверка причины вызова и обработка передачи доступа на исполнение: сохранение контекста и его передача в процесс-близнец напрямую либо через процесс-посредник (2), где происходит восстановление необходимого контекста и управление передаётся вызываемому системному коду (3). Тред-близнец 1 исполняет системный код. Его возврат в код приложения сгенерирует исключение доступа на исполнение (4), где произойдёт сохранение контекста и его передача исходному процессу (5). Обработчик исключений восстановит необходимое состояние, и тред 1 продолжит исполнение кода приложения (6). Независимо от этого происходит обработка исключений доступа к закрытой части карты на чтение и запись.

Создание процесса-близнеца является задачей, сложной с точки зрения технической реализации. Кратко опишем ключевые идеи реализации. Для создания близнецов будет применяться дуплицирование PEB (Process Environment Block), дуплицирование TEB (Thread

Environment Block), дуплицирование системных дескрипторов (handles), а также дуплицирование других системных структур. Дуплицирование будет происходить как в начальный момент при инициализации, так и в процессе исполнения. Также будет применяться дуплицирование виртуального адресного пространства (VAD), как в начальный момент, так и в процессе работы с картой памяти. Может потребоваться работа с такими структурами, как SHARED_USER_DATA, LDR_LIST, TLS. Для предотвращения системных вызовов напрямую через sysenter может применяться технология PICO/Minimal process [20, 21], имплементированная для поддержания Linux-процессов на Windows. Мы дуплицируем ключевые структуры таким образом, чтобы процесс, в котором исполняется код приложения, не обнаружил артефакты, связанные с разделением процессов. Список специальных хуков при необходимости будет расширен, чтобы блокировать попытки увидеть нить в процедуре switch. Маскировка switch требует также защиты стека.

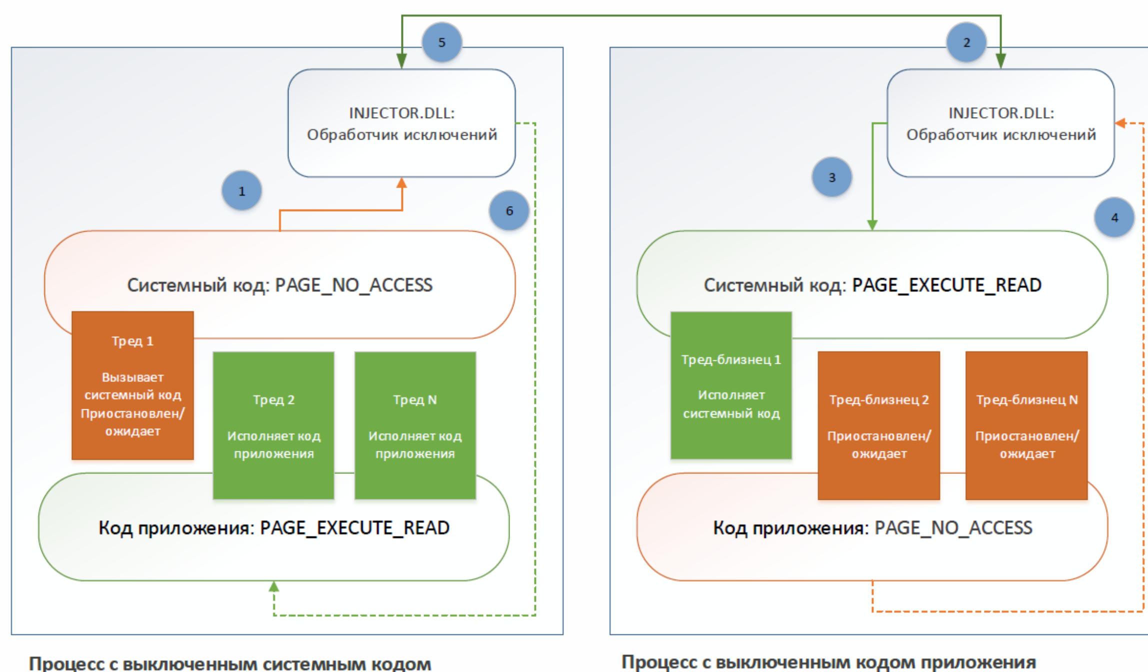


Рис. 5. Переключение карты памяти в многопоточном случае

Таким образом, синхронизация адресных пространств представляет собой сложную задачу, однако позволяет использовать простой механизм переключения. Для снижения потери производительности от сброса кэша страничных преобразований (учитывая, что они в процессах-близнецах идентичны) можно воспользоваться механизмами последних процессоров Intel (VCID или PCID), т. е. использовать для процессов-близнецов одинаковые ID контекста. Эта возможность требует исследования, т. к. при этом необходимо сбросить «плохие» страницы (с различием в бите XD).

Альтернативный подход, при котором не требуется дуплицирование VAD, требует создания планировщика (scheduler) внутри процесса. Это может быть планировщик пользовательского режима, который учитывает существование двух классов кода процесса, и в каждый момент выполняются только те нити, которые исполняют код из открытой части карты, при этом планировщик следит за тем, чтобы оба пула получали свой квант. Этот подход не требует решения сложной задачи дуплицирования адресного пространства, но при этом ставит математическую задачу разработки алгоритма переключения карты памяти и пулов потоков. Когда мы получаем исключение page fault, на основе учёта времени исполнения мы должны переместить поток в другой пул или блокировать исполнение потока (suspend/wait). Также мы принимаем решение о том, следует ли переключить исполняющийся пул потоков. В процессе переключения мы останавливаем треды в испол-

няющемся пуле и возобновляем исполнение тредов из другого пула. В этом случае нужно также перехватить API, который позволит нитям обнаружить аномальное состояние заблокированного потока из другого пула, что также представляется нам сложной задачей.

При выборе механизма для контроля многопоточных приложений нужно ориентироваться на достижение согласованной работы с ядром ОС Windows (в отличие от Linux-систем, где можно просто модифицировать код ядра). С этой точки зрения подход с разработкой планировщика кажется нам более перспективным.

4.6. Примеры работы

В данный момент нами был разработан прототип ToolChain для анализа однопоточных приложений, который был протестирован на 64-битной Windows 10. Механизм патча и мониторинга импортов не зависит от поточности, хоть и требует некоторой синхронизации лога. Мы демонстрируем его работу на notepad.exe. На рис. 6 демонстрируется хук IAT, при котором указатели на функции системных библиотек заменяются на указатели на автоматически генерируемые стабы, ведущие к универсальному логеру. Также здесь показано, что ряд импортов не были перехвачены, т. к. являются переменными. На рис. 7 представлен фрагмент лога, полученного в процессе работы notepad.exe.

```
Get functions to patch
_acmdln (00007FFE6AAE35A0) will not be hooked: from section .data
_fmode (00007FFE6AAE366C) will not be hooked: from section .data
_commode (00007FFE6AAE46B8) will not be hooked: from section .data
Patch functions
OpenProcessToken (00007FFE6B2A6220) -> 00007FF696880000
GetTokenInformation (00007FFE6B2A5E10) -> 00007FF696870000
DuplicateEncryptionInfoFile (00007FFE6B2BA6D0) -> 00007FF696860000
RegSetValueExW (00007FFE6B2A64A0) -> 00007FF696850000
RegQueryValueExW (00007FFE6B2A5D10) -> 00007FF696840000
RegCreateKeyW (00007FFE6B2A6630) -> 00007FF696830000
RegCloseKey (00007FFE6B2A5FD0) -> 00007FF696820000
RegOpenKeyExW (00007FFE6B2A5BE0) -> 00007FF696810000
EventSetInformation (00007FFE6CF6FBE0) -> 00007FF696800000
EventRegister (00007FFE6CF6FE60) -> 00007FF6967F0000
EventUnregister (00007FFE6CF63C10) -> 00007FF6967E0000
EventWriteTransfer (00007FFE6CF62320) -> 00007FF6967D0000
IsTextUnicode (00007FFE6B2A5D20) -> 00007FF6967C0000
DecryptFileW (00007FFE6B2BA650) -> 00007FF6967B0000
CreateStatusWindowW (00007FFE5A26C650) -> 00007FF6967A0000
ChooseFontW (00007FFE6B9F4190) -> 00007FF696790000
GetFileTitleW (00007FFE6B9A1260) -> 00007FF696780000
FindTextW (00007FFE6B9F3180) -> 00007FF696770000
PageSetupDlgW (00007FFE6B9F8E90) -> 00007FF696760000
GetSaveFileNameW (00007FFE6B9EE0B0) -> 00007FF696750000
GetOpenFileNameW (00007FFE6B9EDFC0) -> 00007FF696740000
CommDlgExtendedError (00007FFE6B9E9800) -> 00007FF696730000
```

Рис. 6. Патч импортов notepad.exe

Продемонстрируем, как будут описаны различные типы импортов при их вызове:

1) Нормальный импорт:

```
[Thread 2e34] user32.dll: SendMessageW [import normal]
```

2) Перенаправление импорта:

```
[Thread 2e34] advapi32.dll:
 EventRegister (ntdll.dll: EtwEventRegister) [import redirect normal]
[Thread 2e34] advapi32.dll:
 EventSetInformation (ntdll.dll: EtwEventSetInformation) [import redirect normal]
```

3) Перенаправление импорта с прыжком:

```
[Thread 2e34] kernel32.dll:
 LocalAlloc (kernelbase.dll: LocalAlloc) [import redirect with JMP]
```

```
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: GetMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: TranslateAcceleratorW [import normal]
[Thread 2e34] user32.dll: TranslateMessage [import normal]
[Thread 2e34] user32.dll: DispatchMessageW [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: GetMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: TranslateAcceleratorW [import normal]
[Thread 2e34] user32.dll: TranslateMessage [import normal]
[Thread 2e34] user32.dll: DispatchMessageW [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: SetThreadDpiAwarenessContext [import normal]
[Thread 2e34] user32.dll: GetWindowPlacement [import normal]
[Thread 2e34] user32.dll: SetThreadDpiAwarenessContext [import normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: DestroyWindow [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DestroyWindow [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] user32.dll: SendMessageW [import normal]
[Thread 2e34] user32.dll: PostQuitMessage [import normal]
[Thread 2e34] user32.dll: DefWindowProcW (NTDLL.dll: NtdllDefWindowProc_W) [import redirect normal]
[Thread 2e34] gdi32.dll: DeleteObject [import normal]
[Thread 2e34] user32.dll: GetMessageW [import normal]
[Thread 2e34] kernel32.dll: GetFileAttributesExW [import normal]
[Thread 2e34] user32.dll: UnhookWinEvent [import normal]
[Thread 2e34] api-ms-win-core-winrt-string-l1-1-0.dll: WindowsCreateStringReference -> combase.dll: WindowsCreateStringReference [import apiset]
[Thread 2e34] api-ms-win-core-winrt-l1-1-0.dll: RoGetActivationFactory -> combase.dll: RoGetActivationFactory [import apiset]
[Thread 2e34] msvcrt.dll: free [import normal]
[Thread 2e34] api-ms-win-core-com-l1-1-0.dll: CoUninitialize -> combase.dll: CoUninitialize [import apiset]
[Thread 2e34] advapi32.dll: EventUnregister (ntdll.dll: EtwEventUnregister) [import redirect normal]
[Thread 2e34] msvcrt.dll: exit [import normal]
```

Рис. 7. Лог системных вызовов notepad.exe

Все эти типы могут быть осложнены использованием механизма ApiSetSchema. Покажем работу с ApiSet:

```
[Thread 2e34] kernel32.dll:
 GetStartupInfoW (api-ms-win-core-processthreads-l1-1-0.dll:
GetStartupInfoW -> kernelbase.dll: GetStartupInfoW) [import redirect with JMP]
[Thread 2e34] kernel32.dll:
GetModuleHandleW (api-ms-win-core-libraryloader-l1-2-0.dll:
GetModuleHandleW -> kernelbase.dll: GetModuleHandleW) [import redirect with JMP]
```

```
[Thread 2e34] api-ms-win-core-com-l1-1-0.dll: CoCreateGuid -> combase.dll:
 CoCreateGuid [import apiset]
[Thread 2e34] kernel32.dll:
 HeapSetInformation (api-ms-win-core-heap-l1-1-0.dll:
 HeapSetInformation -> kernelbase.dll:
HeapSetInformation) [import redirect with JMP]
[Thread 2e34] api-ms-win-core-com-l1-1-0.dll: CoInitializeEx -> combase.dll:
CoInitializeEx [import apiset]
[Thread 2e34] kernel32.dll: HeapFree (api-ms-win-core-heap-l1-1-0.dll:
 HeapFree -> ntdll.dll: RtlFreeHeap) [import redirect with JMP]
```

Для тестирования механизма переключения карты памяти нами было написано однопоточное приложение, которое получает адрес *LoadLibrary* из kernel32.dll и читает его код. Фрагмент полученного лога с вызовами системных функций и переключениями карты памяти продемонстрирован на рис. 8. Здесь, помимо переключения между системным кодом и кодом приложения, также тестируется механизм разрешения однократного доступа на чтение. В Windows 10 используется механизм TppWorkerThread, благодаря которому даже однопоточное приложение в процессе создания получит от операционной системы несколько дополнительных нитей. Подробнее о том, как его отключить, можно прочитать в [22]. Мы воспользовались специальным хуком *NtCreateUserThread*.

```
Access to TestApplication.EXE!0x00007FF604691011 reason REASON_EXECUTE
kernel32.dll: GetProcAddress [import normal]
Access to kernel32.dll!GetProcAddress (0x00007FFAF1D9BFB0) reason REASON_EXECUTE
Access to TestApplication.EXE!0x00007FF60469103F reason REASON_EXECUTE
Access to kernel32.dll!LoadLibraryA (0x00007FFAF1DA13F0) reason REASON_READ
Set single step (00007FF604691070)
Single step on 00007FF604691074
```

Рис. 8. Переключение карты памяти

4.7. Машинное обучение

В отличие от внешних событий, таких как файловые, реестровые операции, создание процессов, мониторинг которых легко осуществить на машине конечного пользователя (в статье [1] для мониторинга соответствующих событий использовалась утилита Process Monitor), исследование глубокими аналитическими инструментами может привести к существенной деградации производительности операционной системы. Поэтому для анализатора этого уровня наша идея состоит в том, чтобы по результатам запуска образца в Sandbox формировать отчёт, в котором представлены внутренние события (полная выборка), внешние события (которые, на самом деле, отражают внутренние события, а потому являются частью выборки) и конечный вердикт (является ли образец вредоносным). На этих данных обучается классифицирующая модель (например, нейронная сеть). Обученная модель получает на вход только данные внешнего мониторинга (на машине конечного пользователя). По этим данным модель пытается восстановить полную выборку событий (на основе найденных корреляций между внутренними и внешними событиями) и предсказать конечный вердикт (обнаружить подозрительные паттерны).

Разработка и обучение модели представляет собой самостоятельное исследование, не освещающееся в данной статье.

5. Заключение

В статье [1] мы рассматривали вопросы создания тестового окружения для запуска вредоносного ПО. Следующим важным этапом стала разработка собственных инструментов динамического анализа. В данной статье рассматривается разработка аналитических инструментов для глубокого динамического анализа вредоносного ПО, ориентированного на

операционные системы Windows. Основная идея — обеспечить полный контроль над исполнением образца, логировать все системные вызовы и передачу управления в системный код или код приложения нестандартными способами (динамически сгенерированный код, механизм обратных вызовов, передача управления в середину функции и т. д.). Для этого применяется перехват и мониторинг системных вызовов и контроль исполнения кода с помощью переключений карты памяти. Был разработан прототип для анализа однопоточных 64-битных приложений, а также предложены механизмы контроля над исполнением в случае многопоточного приложения. Первый из них связан с тяжёлой задачей создания процесса-дупликата (дуплицирования VAD и поддержания нитей-близнецов), однако обеспечивает простую процедуру переключения доступа. Второй требует решения математических задач, связанных с разработкой планировщика для двух типов нитей. В дальнейшем планируется реализовать прототип для анализа многопоточных приложений. Разрабатываются подходы по реализации маскировки наличия аналитических инструментов, а также уменьшения влияния аналитических инструментов на производительность системы. Для этого планируется разработка ядерных компонент и применение предоставляемого разработчиками процессоров Intel функционала. Часть механизмов маскировки будет встроена в ToolChain, часть планируется в качестве отдельных модулей (таких как управление течением времени в исследуемом процессе).

Прототипы будут протестированы на наборе вымогателей-шифровальщиков на программно-аппаратном комплексе Sandbox. По итогам этих запусков мы получим статистику о внутренних событиях вредоносного ПО. В совокупности с внешними событиями от средств мониторинга неинвазивного характера собранные данные будут подаваться на вход модели машинного обучения, осуществляющей классификацию программного обеспечения на подозрительное (suspicious) и доверенное (clean).

Работа выполнена при поддержке компании ООО «Акронис».

Литература

1. *Переберина А.А., Костюшко А.В.* Проектирование программно-аппаратного комплекса для запуска вредоносного программного обеспечения // Труды МФТИ. 2018. Т. 10, № 2. С. 114–130.
2. Process Monitor. URL: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon.
3. *Egele M., Scholte Th., Kirda E., Kruegel C.* A survey on automated dynamic malware-analysis techniques and tools // ACM Computing Surveys. 2012. V. 44, N 2, Article 6. URL: https://www.cs.ucsb.edu/%7Echris/research/doc/acmsurvey12_dynamic.pdf.
4. *Balci E.* Искусство антидетекта. Часть 1 — Введение в техники детектирования // securitylab.ru, 2017. URL: https://www.securitylab.ru/analytics/485677.php, 2017.
5. *Hosseini A.* Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques // www.endgame.com/blog/technical-blog, 2017. URL: https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process.
6. *Pietrek M.* Peering Inside the PE: A Tour of the Win32 Portable Executable File Format // Microsoft Systems Journal. 1994. URL: https://msdn.microsoft.com/en-us/library/ms809762.aspx.
7. Vectored Exception Handling // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681420(v=vs.85).aspx.
8. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4 // Intel. 2018. V. 1 3–15. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

9. RtlAddFunctionTable function // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680588(v=vs.85).aspx.
10. PE Format // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx#the_.pdata_section.
11. *Shilon O.* On API-MS-WIN-XXXXX.DLL, and Other Dependency Walker Glitches // ofekshilon.com, 2016. URL: https://ofekshilon.com/2016/03/27/on-api-ms-win-xxxxx-dll-and-other-dependency-walker-glitches.
12. *Renaud S.* Runtime DLL name resolution: ApiSetSchema Part I // blog.quarkslab.com, 2012. URL: https://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-i.html.
13. Api set resolution // lucasg.github.io, 2017. URL: https://lucasg.github.io/2017/10/15/Api-set-resolution.
14. *Батырь А.* Windows 7: технология «системных заплаток» (Shims) // ru.pcmag.com. URL: http://ru.pcmag.com/iznutri/4123/help/windows-7-bezopasnost-i-sovmestimost?p=5.
15. *Ionescu A.* Secrets of the Application Compatilibity Database (SDB) Part 3 // alex-ionescu.com. 2007. URL: http://www.alex-ionescu.com/?p=41.
16. TLB Desynchronization (Split TLB) // Uninformed. 2019. URL: http://uninformed.org/index.cgi?v=6&a=1&p=21.
17. Intel Software Developer's Manual. V. 3A. 4–31.
18. Intel Software Developer's Manual. Vol. 3A. 5–30.
19. Control-flow Enforcement Technology Preview // Intel. 2017. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.
20. *Ionescu A.* The Linux kernel hidden inside Windows 10// BlackHat, 2016. URL: https://github.com/ionescu007/lxss/blob/master/The Linux kernel hidden inside windows 10.pdf.
21. *Hammons J.* Pico Process Overview // Microsoft, 2016. URL: https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/.
22. *Tang J.* How Windows 10 Implements Parallel Loading // threatmatrix.cylance.com, 2017. URL: https://threatmatrix.cylance.com/en_us/home/windows-10-parallel-loading-breakdown.html.

References

1. *Pereberina A.A., Kostyushko A.V.* Hardware and software system design for malware execution (the Sandbox). Proceedings of MIPT. 2018. V. 10, N 2. P. 114–130. (in Russian).
2. Process Monitor. URL: https://docs.microsoft.com/en-us/sysinternals/downloads/procmon.
3. *Egele M., Scholte Th., Kirda E., Kruegel C.* A survey on automated dynamic malware-analysis techniques and tools. ACM Computing Surveys. 2012. V. 44, N 2, Article 6. URL: https://www.cs.ucsb.edu/%7Echris/research/doc/acmsurvey12_dynamic.pdf.
4. *Balci E.* The art of antidetect. Part 1 — Introduction to detection techniques. securitylab.ru. 2017. URL: https://www.securitylab.ru/analytics/485677.php, 2017. (in Russian).
5. *Hosseini A.* Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques. www.endgame.com/blog/technical-blog. 2017. URL: https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process.

6. *Pietrek M.* Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. Microsoft Systems Journal. 1994. URL: https://msdn.microsoft.com/en-us/library/ms809762.aspx.
7. Vectored Exception Handling. Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681420(v=vs.85).aspx.
8. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. Intel. 2018. V. 1. 3–15. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.
9. RtlAddFunctionTable function. Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680588(v=vs.85).aspx.
10. PE Format. Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx#the_.pdata_section.
11. *Shilon O.* On API-MS-WIN-XXXXX.DLL, and Other Dependency Walker Glitches. ofekshilon.com, 2016. URL: https://ofekshilon.com/2016/03/27/on-api-ms-win-xxxxx-dll-and-other-dependency-walker-glitches.
12. *Renaud S.* Runtime DLL name resolution: ApiSetSchema Part I. blog.quarkslab.com, 2012. URL: https://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-i.html.
13. Api set resolution. lucasg.github.io, 2017. URL: https://lucasg.github.io/2017/10/15/Api-set-resolution.
14. *Batyr A.* Windows 7: Shims technology. ru.pcmag.com. URL: http://ru.pcmag.com/iznutri/4123/help/windows-7-bezopasnost-i-sovmestimost?p=5. (in Russian).
15. *Ionescu A.* Secrets of the Application Compatilibity Database (SDB) Part 3. alex-ionescu.com, 2007. URL: http://www.alex-ionescu.com/?p=41.
16. TLB Desynchronization (Split TLB). Uninformed, 2019. URL: http://uninformed.org/index.cgi?v=6&a=1&p=21.
17. Intel Software Developer's Manual. V. 3A 4–31.
18. Intel Software Developer's Manual. V. 3A 5–30.
19. Control-flow Enforcement Technology Preview. Intel, 2017. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.
20. *Ionescu A.* The Linux kernel hidden inside Windows 10. BlackHat, 2016. URL: https://github.com/ionescu007/lxss/blob/master/The Linux kernel hidden inside windows 10.pdf.
21. *Hammons J.* Pico Process Overview. Microsoft, 2016. URL: https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/.
22. *Tang J.* How Windows 10 Implements Parallel Loading. threatmatrix.cylance.com, 2017. URL: https://threatmatrix.cylance.com/en_us/home/windows-10-parallel-loading-breakdown.html.

Поступила в редакцию 20.06.2018