Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»
Факультет управления и прикладной математики
Кафедра теоретической и прикладной информатики

Направление подготовки/специальность: 03.04.01 Прикладные математика и физика
Направленность (профиль) подготовки: Математические и информационные технологии
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения
_____/_____
(подпись) *расшифровка подписи*
«____» _____20____г.

ПРОЕКТИРОВАНИЕ ПРОГРАММНО-АППАРАТНОГО КОМПЛЕКСА ДЛЯ АВТОМАТИЧЕСКОГО ЗАПУСКА ВРЕДОНОСНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И РАЗРАБОТКА ИНСТРУМЕНТАРИЯ ДЛЯ ДИНАМИЧЕСКОГО АНАЛИЗА

(магистерская диссертация)

Обучающаяся: Переберина Анастасия Александровна

_____
(подпись обучающейся)

Научный руководитель: Емельянов Павел Владимирович, к. ф.-м. н.

_____
(подпись научного руководителя)

Научный консультант: Костюшко Алексей Владиславович

_____
(подпись научного консультанта)

Москва 2018

Аннотация

Совершенствование технологий проактивной защиты от вредоносного ПО требует автоматизации анализа и улучшения средств мониторинга происходящих в системе событий. В данной работе рассматриваются вопросы запуска и анализа вредоносного ПО. Целью исследования является обеспечение автоматического сбора данных о поведении вредоносного ПО для последующего анализа. Для достижения поставленной цели были сформулированы следующие задачи:

1. Построение программно-аппаратного комплекса для автоматического запуска образцов вредоносного ПО;
2. Интеграция с аналитическими инструментами, осуществляющими сбор данных о поведении запускаемых образцов;
3. Разработка собственных аналитических инструментов для сбора данных о поведении вредоносного ПО;
4. Интеграция с лабораториями, осуществляющими анализ собранных данных, для автоматической поставки необходимой информации.

В результате работы были разработаны и апробированы прототипы автоматизированного программно-аппаратного комплекса для запуска вредоносного ПО (Sandbox) и инструментов для глубокого динамического анализа (ToolChain). Была разработана архитектура системы, схема развёртывания и управления системой. Основное внимание было уделено пропускной способности и безопасности комплекса. Получаемые в Sandbox данные подаются на вход модели машинного обучения, осуществляющей классификацию программ. Sandbox используется также для автоматического тестирования ПО на способность детектировать вредоносное ПО и нейтрализовать его воздействие на ОС. Основная идея ToolChain — обеспечить полный контроль над исполнением образца ПО на тестовом сервере. Предложен метод разделения исполняемого кода на «системный» и «приложения» и последующего контроля с помощью переключения карты памяти. Этот

метод прототипирован для однопоточного приложения и спроектирован для многопоточного. Также мы рассмотрели вопросы производительности и маскировки аналитических инструментов.

Для реализации предложенного метода в многопоточном случае требуется провести дополнительный анализ алгоритма. Предложены сценарии использования разработанных прототипов.

Оглавление

1 Введение 8
1.1 Анализ состояния проблемы 8
1.2 Обоснование выбора темы и актуальность 9
1.3 Исследование. Объект и предмет. Цели и задачи 9
1.4 Методы исследования 10
1.5 Степень разработанности тематики 11
1.6 Теоретическая значимость и новизна работы 11
1.7 Практическая значимость и апробация результатов 12
1.8 Сведения о публикациях 12
1.9 Структура работы 13
2 Вредоносное программное обеспечение и антивирусное ПО 13
2.1 Виды вредоносного ПО 14
2.1.1 Вымогатели-шифровальщики 15
2.2 Антивирусное ПО 19
2.3 Статический и динамический анализ 21
2.3.1 Статический анализ 21
2.3.2 Динамический анализ 22
2.4 Гонка вооружений 22
2.4.1 Самомодифицирующийся код и упаковщики 23
2.4.2 Детектирование аналитических инструментов 24
2.4.3 Логическая бомба 26
2.4.4 Анализ производительности 26
2.5 Аналитические инструменты и гонка вооружений на компьютерах конечных пользователей 27
2.5.1 Детектирование признаков шифрования и упаковки 27
2.5.2 Эвристический анализ 27
2.5.3 Энтропия 29

2.5.4 Детектирование запуска в тестовой среде 29
2.5.4.1 Использование функции *GetTickCount* 30
2.5.4.2 Проверка количества ядер 31
2.5.4.3 Выделение большого объёма памяти 31
2.5.4.4 Проверка мьютексов 31
2.5.5 Детектирование отладчика 32
3 Запуск вредоносного программного обеспечения 33
3.1 Стратегии имплементации 33
3.2 Существующие решения 35
4 Сведения об ОС Windows 37
4.1 PE-формат (Portable Executable) 37
4.1.1 Импорты 37
4.1.2 Экспорты 40
4.2 Инжектирование DLL в адресное пространство процесса 41
4.2.1 Инжекция кода 42
4.2.2 Модификация работы приложения 46
4.3 Механизм ApiSetSchema 46
4.4 Механизм SHIMS 54
4.5 SEH и обработка исключений 58
4.5.1 Обработка исключений 59
4.5.2 Structured Exception Handling 60
4.5.3 Frame-Based Exception Handling 61
4.5.4 VEH 62
5 Создание программно-аппаратного комплекса для запуска вредоносного ПО и разработка инструментария для динамического анализа 64
5.1 Терминология 64

5.2 Проектирование программно-аппаратного комплекса для запуска вредоносного ПО . . . 69
5.2.1 Постановка задачи . . . 69
5.2.2 Требования к автоматизации . . . 70
5.2.3 Требования к интеграции . . . 70
5.2.4 Выбор методов и ключевые архитектурные решения . . . 71
5.2.5 Архитектура Sandbox . . . 73
5.2.5.1 Общий обзор архитектуры . . . 73
5.2.5.2 Сетевая архитектура . . . 74
5.2.5.3 Развёртывание системы на физических серверах . . . 76
5.2.6 Агенты . . . 78
5.2.6.1 Process Monitor . . . 78
5.2.7 Набор образцов вредоносного ПО . . . 79
5.2.8 Проблема окружения . . . 80
5.2.9 Проблема защиты логов . . . 84
5.2.10 Производительность аппаратного обеспечения . . . 85
5.2.11 Машинное обучение . . . 85
5.2.12 Виртуальные машины . . . 86
5.2.12.1 Производительность . . . 87
5.2.13 Linux в качестве тестовой системы . . . 87
5.3 Разработка инструментария для динамического анализа вредоносного программного обеспечения . . . 89
5.3.1 Постановка задачи . . . 89
5.3.2 Выбор методов и ключевые архитектурные решения . . . 90
5.3.3 Глубокий динамический анализ . . . 91
5.3.4 Патч импортов . . . 94
5.3.5 Работа с ApiSet . . . 97
5.3.6 Работа с SHIMS . . . 102

5.3.7 Контроль памяти 102
5.3.7.1 Производительность 105
5.3.8 Маскировка аналитических инструментов 106
5.3.9 Многопоточность 108
5.3.10 Примеры работы 112
5.3.11 Машинное обучение 115
5.4 Сценарии использования инструментов 115
6 Заключение 120
7 Список литературы 125

1 Введение

1.1 Анализ состояния проблемы

Исследование поведения вредоносного ПО продолжает оставаться актуальным вопросом информационной безопасности. С момента появления первых экземпляров увеличилось видовое разнообразие вредоносного программного обеспечения, его сложность, количество новых образцов и скорость их появления, а также масштабы угроз.

Вредоносное ПО эволюционирует и совершенствуется. Появляются новые виды вредоносного ПО — вымогатели-шифровальщики, майнеры криптовалюты. Количество вредоносного ПО в существующих базах данных велико, и каждый день появляются новые угрозы. Так, немецкая лаборатория AV-TEST регистрирует более 250000 новых вредоносных программ в день [1]. Традиционный антивирус, ориентирующийся на проверку сигнатур по базам данных (так называемая реактивная защита), которые пополнялись экспертом-аналитиком, не может противостоять вновь появляющимся, неизвестным ранее угрозам [2]. Поэтому появляются инструменты статического анализа PE-файла и динамического анализа вредоносного ПО, исследующие поведение образца, данных о котором не содержится в базе сигнатур. При этом со стороны вредоносного ПО усложняются методы борьбы с антивирусными сканерами. Вредоносное ПО детектирует присутствие в системе аналитических инструментов, запуск в тестовой среде или виртуальной машине, не проявляя вредоносной активности в случае обнаружения признаков запуска под анализом. Вредоносное ПО применяет логические бомбы для активации заражения. Появляются новые виды уклонения от детектирования средствами киберзащиты — мутирующее ПО, полиморфное ПО. Современные антивирусы используют эвристики и эмуляторы и ведут активный мониторинг того, что происходит в операционной системе (проактивная защита). Также существует так называемый «next-gen» подход, при котором применяется поведенческий анализ вредоносного ПО и машинное обучение [3].

1.2 Обоснование выбора темы и актуальность

Борьба с большим количеством новых угроз требует ответных мер, куда входят:

1. Автоматизация анализа вредоносного ПО;
2. Совершенствование аналитических инструментов;
3. Совершенствование средств маскировки аналитических инструментов;
4. Применение триггеров, предназначенных для срабатывания условий атаки вредоносного ПО;
5. Применение подходов машинного обучения и совершенствование математических методов, позволяющих автоматизированное нахождение подозрительных паттернов.

В данной работе мы сосредоточены на первых двух пунктах. Другие пункты также в некотором виде присутствуют в работе и определяют потенциальный вектор последующего исследования. Развитие вредоносного ПО и необходимость ответа со стороны средств защиты являются обоснованием актуальности темы данной работы.

1.3 Исследование. Объект и предмет. Цели и задачи

Объектом исследования является вредоносное программное обеспечение. В данный момент исследование сосредоточено на вымогателях-шифровальщиках, ориентированных на операционные системы семейства Windows. **Предметом** исследования является поведение вредоносного ПО. В данный момент в центре рассмотрения находятся такие его аспекты, как:

1. Создание условий для автоматического запуска вредоносного ПО, обеспечивающих корректное и контролируемое заражение тестовой системы;

2. Получение информации о низкоуровневом (на уровне вызовов системного API и генерации динамического кода) поведении вредоносного ПО, которую можно использовать для ручного (экспертом-аналитиком) или автоматизированного (моделями машинного обучения) анализа.

Целью исследования является обеспечение автоматического сбора данных о поведении вредоносного ПО для последующего анализа. Для достижения поставленной цели перед нами стояли следующие **задачи**:

1. Построение программно-аппаратного комплекса для автоматического запуска образцов вредоносного программного обеспечения;

2. Интеграция с аналитическими инструментами, осуществляющими сбор данных о поведении запускаемых образцов;

3. Разработка собственных аналитических инструментов для сбора данных о поведении вредоносного ПО;

4. Интеграция с лабораториями, осуществляющими анализ собранных данных, для автоматической поставки необходимой информации.

1.4 Методы исследования

В процессе изучения состояния предметной области мы использовали такие **методы**, как анализ, синтез, сравнение, аналогия и моделирование, как в области запуска и анализа вредоносного ПО, так и в области уклонения от запуска и анализа со стороны вредоносного ПО. В процессе прототипирования нами применялась гибридная модель разработки, сочетающая традиционную каскадную модель (waterfall) и гибкие итеративные методы (agile). Процесс разработки состоял из следующих этапов:

1. Проектирование и дизайн архитектуры комплекса и аналитических инструментов;

2. Прототипирование разработанных архитектурных решений;
3. Экспериментальное тестирование разработанных прототипов;
4. Измерение производительности разработанных прототипов.

По итогам работы был проведён анализ полученных результатов.

1.5 Степень разработанности тематики

Мы провели обзор существующих исследований и подходов к запуску и анализу вредоносного ПО, который мы также приводим в своей работе. Особенно мы отмечаем статью Egele M., Scholte Th., Kirda E., Kruegel C. A survey on automated dynamic malware analysis techniques and tools [2]. В ней подробно освещены подходы к анализу вредоносного программного обеспечения, упомянут как статический, так и динамический анализ, а также подходы к созданию тестовой среды для запуска образцов вредоносного ПО и различные существующие решения. Основное внимание в большинстве статей уделяется программному обеспечению для динамического анализа образцов, а не инфраструктурным решениям и вопросам развёртывания системы. В своей работе мы останавливаемся на этих вопросах более подробно. В области динамического анализа мы опираемся на существующие технологии, уделяя внимание контролю над запуском приложения, мониторингу происходящих событий и способам маскировки аналитических инструментов. Отметим также, что возможности искусственного интеллекта для анализа вредоносного программного обеспечения использовались многими исследователями [4; 5]. Распространённые подходы включают в себя классификацию образцов на вредоносные и не вредоносные, а также кластеризацию вредоносного ПО.

1.6 Теоретическая значимость и новизна работы

В данной работе речь идёт о технологиях проактивной защиты от вредоносного ПО во время исполнения. Сюда входит как запуск в тестовой среде, так и анализ поведения исполняющегося образца. Мы останавливаемся на запуске образцов вредоносного ПО на реальном оборудовании, т.к. вредоносное ПО может

детектировать запуск в виртуальной машине, при этом используем виртуальные машины для увеличения производительности комплекса. Мы также уделяем внимание вопросам безопасности запуска вредоносного ПО в комплексе. В работе приведено детальное описание технических сложностей, преодолённых в процессе реализации прототипа, особенно с учетом специфики программ-вымогателей. В области динамического анализа мы придерживаемся традиционных подходов к мониторингу системных вызовов и предлагаем механизм контроля над приложением, который не встречается в найденных нами источниках, основанный на отделении кода приложения от кода системных библиотек. Также мы предлагаем сценарии совместного использования разработанных инструментов. Вместо разворачивания тестовой среды на машине конечного пользователя мы предлагаем использовать для детального анализа программно-аппаратный комплекс с облачным транспортом. На тестовом сервере осуществляется полный контроль над исполнением образца и мониторинг внутренних событий процесса, которые используются для обучения модели искусственного интеллекта. Модель машинного обучения определяет уровень подозрительности образца.

1.7 Практическая значимость и апробация результатов

Проведённое исследование имеет практическую значимость. В данный момент разработанные прототипы используются для запуска и получения информации о работе вредоносного ПО в компании Acronis. Полученные данные передаются в лабораторию машинного обучения для обучения модели, представляющей собой классификатор программ на подозрительные и не вызывающие подозрения. Также комплекс используется для тестирования разрабатываемых антивирусных компонент на способность детектировать вредоносное ПО и нейтрализовать его воздействие на операционную систему.

1.8 Сведения о публикациях

Данная работа была представлена на научной конференции МФТИ. По результатам работы была написана статья «Проектирование программно-

аппаратного комплекса для запуска вредоносного программного обеспечения» и статья «Разработка инструментария для динамического анализа вредоносного программного обеспечения», которые будут опубликованы в научном журнале «Труды МФТИ». В ходе исследования были предложены механизмы глубокого анализа вредоносного программного обеспечения, имеющие научно-теоретическую значимость, которые были изложены в качестве патентных заявок.

1.9 Структура работы

Основная часть работы представлена в Главах 2-5. В Главах 2 и 3 приведён проведённый нами обзор существующих решений. Глава 2 посвящена вредоносному ПО, антивирусным подходам и так называемой «гонке вооружений». В Главе 3 рассмотрены ключевые особенности тестовой среды для запуска вредоносного ПО. В Главе 4 содержатся сведения об операционной системе Windows, которые мы используем при разработке аналитических инструментов глубокого анализа. Глава 5 посвящена описанию разработанных прототипов, включая архитектурные решения и решённые проблемы, а также предложения по дальнейшему улучшению инструментов. Также в Главе 5 описаны сценарии использования разработанных прототипов.

2 Вредоносное программное обеспечение и антивирусное ПО

В данной главе будет рассказано о существующих типах вредоносного программного обеспечения. Мы остановимся более подробно на вымогателях-шифровальщиках, которые находятся в центре нашего внимания на текущий момент. Далее будет рассказано о технологиях антивирусной защиты и будет произведён обзор существующих аналитических инструментов для борьбы с неизвестными ранее угрозами. Также мы рассмотрим некоторые методы защиты от обнаружения антивирусным сканером со стороны вредоносного ПО и методы их обхода со стороны аналитических инструментов, приводящие к так называемой «гонке вооружений» между экспертами по киберзащите и разработчиками вредоносного ПО.

2.1 Виды вредоносного ПО

Вредоносное программное обеспечение (malware) — программное обеспечение, предназначенное для получения несанкционированного доступа к вычислительным ресурсам ЭВМ или к информации, хранимой на ЭВМ [6]. В статье [2] при определении этого термина подчёркивается осознанность намерения атакующего причинить вред.

В настоящее время вредоносное программное обеспечение чаще всего не принадлежит какому-либо классу в чистом виде, а обладает признаками разных видов вредоносного ПО. Вредоносное ПО в компьютерном сленге часто именуется вирусом, в то время как вирус — лишь узкий класс вредоносного программного обеспечения. Рассмотрим классификацию вредоносного ПО согласно [2].

Червь (Worm). Программа, которая может работать независимо и может распространить свою полностью рабочую версию на другие машины. Характерное поведение червя — самовоспроизведение. Преобладает в сетевых окружениях, таких как Интернет.

Вирус (Virus). Код, который не может исполняться независимо и для запуска нуждается в «хосте» — программе или коде операционной системы, куда он будет внедрён. Вирусы могут распространяться, заражая доступные уязвимые хосты.

Троянский конь (Trojan Horse). Программное обеспечение, которое маскируется под выполняющее некоторую полезную деятельность (например, оно может симулировать плагины для WEB-браузера или компьютерные игры), фоном осуществляя вредоносную активность. При этом после установки вредоносная часть может загружать дополнительное вредоносное ПО, модифицировать системные настройки, заражать другие файлы.

Шпионское программное обеспечение (Spyware). Программное обеспечение, получающее информацию с уязвимого компьютера и передающее её атакующему (например, данные учётных записей, банковские счета, переписки).

Бот (Bot). Часть вредоносного ПО, позволяющая его владельцу (он имену-

ется bot master) удалённо управлять заражённой системой. Набор ботов, которыми коллективно управляет один владелец, называется ботнет (botnet). Боты обычно запрограммированы рассылать спам или осуществлять шпионскую активность.

Руткит (Rootkit). Программное обеспечение, которое скрывает какую-либо информацию от пользователя компьютера (например, своё присутствие в системе). Управление соответствующей информацией позволяет руткиту скрывать процессы, файлы или сетевые соединения в заражённой системе. Существуют также *буткиты* (bootkit), осуществляющие модификацию MBR (Master Boot Record) для управления загрузкой операционной системы.

2.1.1 Вымогатели-шифровальщики

Вредоносное ПО, называемое ransomware (вымогатели) или crypto-ransomware (вымогатели-шифровальщики), появилось не так давно, но уже вызвало несколько мировых эпидемий заражений и продолжает бурно развиваться. Данный тип вредоносного программного обеспечения ориентирован на блокирование работы компьютера пользователя или доступа пользователя к хранимым данным и требование выкупа за восстановление работы ПК или данных пользователя [7]. Некоторые экземпляры ransomware только имитируют блокировку или шифрование, угрожая пользователю и маскируясь под известные вымогатели. Они также могут вывести на экран фальшивое уведомление о том, что система пользователя использовалась для незаконной деятельности, содержит порнографию или пиратский контент, представляясь, например, правоохранительными органами. Этот приём применяется для того, чтобы пользователи не предпринимали попыток обратиться за помощью к специалистам по информационной безопасности.

Некоторые приложения блокируют систему или даже редактируют главную загрузочную запись и/или таблицу разделов, чтобы препятствовать тому, чтобы операционная система загрузилась, пока операторы вредоносного ПО не получат выкуп [7].

Основной необходимый элемент при создании программы-вымогателя — удобная платёжная система, которую трудно проследить [7].

Самыми интересными для нас являются вымогатели-шифровальщики, которые шифруют файлы пользователя, используя устойчивое шифрование, при этом только у автора вредоносного ПО есть необходимый ключ расшифровки. После заражения пользователь видит всплывающее окно или заблокированный экран с сообщением от атакующих, которое может быть представлено на разных языках. В нём говорится о том, что система жертвы подверглась заражению, все файлы были зашифрованы с использованием «последних» криптографических алгоритмов, и расшифровка невозможна без ключа, который есть только у операторов crypto-ransomware. Для получения ключа жертве предлагается заплатить криптовалютой — такой способ оплаты обеспечивает анонимность операторов. Пример такого сообщения приведён на Рисунке 1. Это требование выкупа от вымогателя-шифровальщика, известного как WannaCry, эпидемия заражения которым произошла в мае 2017 года. Различные версии WannaCry требовали от 300 до 600 долларов в биткоин-эквиваленте. Однако выплата выкупа не гарантирует расшифровку данных. Известны случаи, когда операторы вредоносного ПО физически не могли расшифровать данные из-за ошибок, допущенных при разработке программы (например, WannaCry). Иногда экспертам лабораторий, занимающихся кибербезопасностью, удаётся написать декриптор (как, например, в случае шифровальщика TeleCrypt, обнаруженного в 2016 году и использовавшего протокол мессенджера Telegram для связи с управляющими серверами) или найти баги в реализации вредоносного ПО или в системном коде, позволяющие расшифровать файлы (например, WannaCry использовал для уничтожения сгенерированного ключа шифрования функцию Microsoft CryptoAPI *CryptReleaseContext*, которая на самом деле не стирала ключ, и его можно было найти в оперативной памяти).

Первые ransomware стали появляться в 2005-2006 годах. Наиболее известные атаки пришлись на 2017 год: WannaCry (май), Petya (июнь), BadRabbit (октябрь) [8].

Рисунок 1 — Сообщение шифровальщика-вымогателя WannaCry

Источник: Скриншот с заражённой тестовой системы в прототипе Sandbox

Программы, относящиеся к ransomware, технически представляют собой компьютерный вирус или сетевой червь [8]. Заражение обычно происходит посредством массовой рассылки через электронную почту и фишинговые ссылки или, в более сложных случаях, при атаке через уязвимость в сетевой службе. Например, WannaCry эксплуатировал критическую уязвимость в протоколе SMB, которая присутствовала в системах семейства Windows, начиная с Windows XP/Windows Server 2003 и заканчивая Windows 10/Windows Server 2016. Она позволила вымогателю-шифровальщику распространяться самопроизвольно, без каких-либо действий со стороны пользователя (требовался только открытый порт 445). 14 марта 2017 года Microsoft выпустила серию обновлений, призванных нейтрализовать уязвимость во всех поддерживаемых ОС.

Обычно вымогатели-шифровальщики имеют список расширений файлов,

на которые ориентирована атака, он может быть довольно обширным — от текстовых файлов до виртуальных машин. Атакуемые директории могут быть предопределены в коде вредоносного ПО (например, Documents), или же получены запросом к операционной системе, при этом обычно есть список директорий, которые не трогаются шифровальщиком (например, %SystemRoot%).

Для шифрования могут использоваться, например, ключи RSA (2048-битный ключ RSA использовался вымогателем WannaCry). Для генерации ключей вредоносное ПО может использовать криптографическое API, предоставляемое операционной системой, или стороннюю (реже — свою собственную) реализацию. В статье [9] описан следующий протокол с тремя раундами, выполняющийся между атакующим и жертвой [7]:

1. Атакующий генерирует пару ключей для асимметричного шифрования (публичный и приватный) и помещает соответствующий открытый ключ во вредоносное программное обеспечение.

2. При осуществлении атаки вредоносное программное обеспечение генерирует случайный симметричный ключ и с его помощью шифрует данные жертвы. Открытый ключ во вредоносном программном обеспечении используется, чтобы зашифровать симметричный ключ. Это известно как гибридное шифрование: на выходе получается небольшой асимметрично зашифрованный текст, а данные жертвы шифруются симметричным ключом шифрования. Симметричный ключ в памяти должен быть уничтожен, как и оригинальные данные пользователя, чтобы предотвратить восстановление информации. Программа показывает пользователю сообщение, которое включает асимметрично зашифрованный текст с ключом шифрования данных и инструкцию по выплате выкупа. Жертва отправляет асимметрично зашифрованный ключ шифрования и электронные деньги атакующему.

3. Атакующий получает оплату, дешифрует асимметрично зашифрованный ключ своим приватным ключом и отправляет симметричный ключ для рас-

шифровки данных жертве. Атака завершена, теперь жертва может дешифровать зашифрованные данные с необходимым симметричным ключом.

Симметричный ключ сгенерирован случайным образом и не поможет другим жертвам [7]. Приватный ключ знает только атакующий, жертва пересылает атакующему очень маленький шифрованный текст (зашифрованный ключ симметричного шифрования). Для генерации рандомных величин на компьютере жертвы может использоваться текущее время, размер жёсткого диска, размер свободной памяти, номер процесса или NETBIOS-имя компьютера [10].

В настоящий момент вымогатели-шифровальщики представляют собой один из ключевых типов угроз информационной безопасности [11; 12].

2.2 Антивирусное ПО

Антивирусное ПО предназначено для обнаружения вредоносных программ, восстановления заражённых такими программами файлов, предотвращения заражения операционной системы [13].

Традиционные антивирусные решения используют хеши и сигнатуры для идентификации вредоносного ПО. Хеши (MD5, SHA-1, SHA-256) и сигнатуры хранятся в базе данных, которая может быть локальной или расположенной в облаке. Обнаружение, основанное на сигнатурах, называется реактивной технологией защиты, которая занимается поиском в системе уже известного вредоносного ПО [14]. Такой подход может плохо работать из-за большого количества вредоносного ПО, применяющего механизмы обхода антивирусных сканеров. К тому же, такой подход не может защитить систему от неизвестных ранее угроз, информации о которых не содержится в базе данных.

Для защиты от неизвестных угроз используются проактивные технологии защиты, основная цель которых — предотвращение заражения системы [15]. В рамках проактивной защиты используются запуск в тестовой среде и аналитические инструменты, осуществляющие мониторинг происходящих в системе событий. Запуск в тестовой среде необходим для того, чтобы вредоносное ПО не могло

нанести вред системе пользователя. При этом может применяться эмуляция кода, виртуализация окружения, антивирусная песочница (в которой приложение исполняется в условиях ограничения уровня привилегий и имеет ограниченный доступ к операционной системе). Во время запуска в тестовой среде осуществляется анализ поведения приложения. Недостатками такого подхода являются ограничение ресурсов на тестовый запуск и возможность обнаружения тестовой среды со стороны вредоносного ПО. Это ведёт к так называемой «гонке вооружений» и необходимости маскировать присутствие аналитических инструментов и тестовой среды от вредоносного ПО.

При мониторинге происходящих событий традиционные антивирусные решения применяют эвристический подход. Эвристические движки представляют собой экспертные системы, основанные на обработке событий и заложенном механизме принятия решения. Недостатком эвристического анализа является большое количество ложных срабатываний (false positives), т. е. блокирование легитимных программ.

В решениях в области информационной безопасности нового поколения (next-gen) акцент делается на другие методы обнаружения и противодействия, при этом также используются инструменты для устранения последствий начавшихся атак. Рассмотрим методы, которые применяются до исполнения (pre-execution), во время исполнения (on-execution) и после исполнения (post-execution).

До исполнения собирается вся доступная информация об объекте методами статического анализа. Традиционный антивирус применяет здесь проверку сигнатур или эвристики (подозрительная энтропия, странные секции). Next-gen подход может использовать модели машинного обучения для принятия решения. В зависимости от результатов проверок объект может быть признан вредоносным (и его исполнение может быть не допущено) или подозрительным и требующим особенного внимания во время исполнения.

Во время исполнения применяются методы динамического анализа. На этой стадии происходит сбор данных о поведении образца, которые могут обрабаты-

ваться экспертными системами (традиционный антивирус) или методами искусственного интеллекта (next-gen). При высоком уровне угрозы работа процесса приостанавливается. В случае вредоносности образца процесс терминируется.

После исполнения при обнаружении атаки производится устранение последствий заражения и откат изменений.

2.3 Статический и динамический анализ

Инструменты анализа вредоносного программного обеспечения могут быть *статическими*, ориентированными на анализ исходного или объектного кода, структуры PE-файла без запуска соответствующего процесса, и *динамическими*, работающими во время выполнения образца. Представленный ниже обзор аналитических инструментов выполнен на основе материалов статьи [2].

2.3.1 Статический анализ

Инструменты, анализирующие программное обеспечение, не исполняя его, называют статическими. Методы статического анализа могут быть использованы для различных представлений. Это может быть исходный код, если он доступен, что бывает редко. Это может быть бинарное представление, анализ которого осложняют потеря семантической информации из исходного кода (комментарии, названия переменных, функций и структур) и структурные изменения бинарного кода (инлайн-подстановки, локальная и глобальная оптимизация кода). Статические инструменты могут строить граф вызовов, вычислять возможные значения параметров, заниматься исследованием всех возможных ветвей исполнения программы (в то время как во время исполнения не все из них могут реализоваться).

Для противостояния статическому анализу вредоносное программное обеспечение использует методы самомодификации, такие как упаковщики и криптоpы. Также используются значения, которые не могут быть определены статически (например, текущее время).

2.3.2 Динамический анализ

Аналитические инструменты, работающие во время выполнения программы, называются инструментами 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).

2.4 Гонка вооружений

Антивирусное ПО детектирует вредоносное программное обеспечение, в том числе методами динамического анализа. Вредоносное ПО стремится защи-

титься от аналитических средств, не показывая вредоносное поведение, применяя различные техники анти-ревёрсинга и анти-отладки. Для маскировки от антивирусов вредоносное ПО применяет различные 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 инструментов анти-ревёрсинга и анти-дебаггинга может быть задетектировано антивирусами и расценено как подозрительное или вредоносное поведение. Рассмотрим некоторые механизмы, применяемые обеими сторонами. Представленный в данном разделе материал подготовлен на основе статей [2] и [16], в которых можно найти подробности по этой теме.

2.4.1 Самомодифицирующийся код и упаковщики

В эту категорию входят такие методы, как обфускация, шифрование, рекурсивная упаковка, полиморфические и метаморфические бинарные файлы.

Обфускация кода связана с перемешиванием исходного текста бинарного файла без порчи функций. Этот метод затрудняет статический анализ и изменяет хеш-сигнатуры бинарного файла. Обфускацию можно реализовать при помощи добавления нескольких строк мусорного кода или изменения последовательности выполнения инструкций программным путем. Данная техника может позволить избежать детектирования антивирусами.

Упаковщик сжимает исходный файл и объединяет сжатые данные с кодом распаковки в один исполняемый файл. Когда сжатый исполняемый файл запускается, вначале программа распаковки воссоздаёт первоначальный код из сжатого файла, после чего происходит выполнение. Когда антивирусы сканируют упакованное вредоносное ПО, им нужно определить алгоритм сжатия и распаковать исполняемый файл. Поскольку упакованные файлы сложнее проанализировать, злоумышленники заинтересованы в упаковщике, который не известен разработчикам аналитического ПО.

Крипторы предназначены для шифрования бинарного файла с целью затруднения анализа или реверс-инжиниринга. У шифровщика есть две части: сборщик и стаб. Сборщик шифрует указанный бинарный файл и помещает внутрь стаб, являющийся самой важной частью криптора. При запуске бинарного файла сначала стаб расшифровывает первоначальную версию в память, а затем запускает расшифрованный файл при помощи метода RunPE (в большинстве случаев).

Для борьбы с упаковщиками и крипторами аналитические инструменты используют, например, W⊕E (Write-XOR-Execute, т. е. запись либо исполнение, но не одновременно) технологию. Алгоритм описывается следующим образом:

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

2. 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).

3. 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.

2.4.2 Детектирование аналитических инструментов

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

1. 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);

2. Анализ среды исполнения;

   Артефакты в среде анализируемого процесса, которые могут выдать запуск под аналитическими инструментами — например, присутствие отладчика внутри процесса.

3. Анализ внешних приложений;

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

4. 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 модифицированному PTE, это вызовет page fault, а обработчик вернёт теневую версию этой записи.

Некоторые аналитические инструменты 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.

2.4.3 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.

2.4.4 Анализ производительности

Наличие аналитических инструментов может привести к деградации производительности операционной системы, которая может быть замечена вредоносным ПО. Методы сокрытия падения производительности (патч инструкции RDTSC, возвращающей значение регистра Time Stamp Counter, счётчика количества тактов работы CPU с последней перезагрузки, или 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).

2.5 Аналитические инструменты и гонка вооружений на компьютерах конечных пользователей

Аналитические инструменты на компьютерах конечных пользователей могут применять различные подходы для детектирования. Наряду с анализом сигнатур и попытками распознать вредоносные паттерны (код криптора, код упаковки, анти-ревёрсинговый и анти-отладочный код), применяются также эвристические и энтропийные подходы. Мы расскажем об антивирусных сканерах и способах их обхода по материалам статьи [16].

2.5.1 Детектирование признаков шифрования и упаковки

На сегодняшний день антивирусные программы, помимо поиска сигнатур и вредоносного воздействия, ищут в исследуемом ПО признаки крипторов и упаковщиков. Детектирование крипторов и упаковщиков является частной задачей и, к тому же, более простой, так как данные инструменты имеют одинаковые подозрительные алгоритмы, наподобие расшифровки PE-файла и запуска в памяти.

2.5.2 Эвристический анализ

Эвристический анализ используется многими антивирусными программами для детектирования ранее неизвестных компьютерных угроз, а также новых вариантов вредоносного ПО, уже существующих в «дикой природе». Данные инструменты анализа формируют экспертную систему, которая определяет степень восприимчивости операционной системы по отношению к конкретной угрозе при помощи различных правил принятия решений или весовых методов.

Эвристические движки основаны на статическом либо динамическом анализе и механизме правил. Детектирование происходит посредством группировки

и оценки угроз отдельных фрагментов кода, согласно предопределённым критериям. Даже при сканировании простейшей программы, которая только печатает «Привет, мир», эвристический движок определяет уровень угрозы и риска. Если оценка превышает пороговый уровень, файл помечается как вредоносный. Эвристические движки — наиболее продвинутая компонента антивирусных продуктов и используют значительное количество правил и критериев. Поскольку антивирусные компании не выпускают документации, описывающей эвристические движки, известны лишь некоторые критерии оценки угроз. Ниже перечислены подозрительные активности, которые могут использоваться в эвристических правилах:

- Присутствие цикла дешифровки;
- Чтение текущего имени компьютера;
- Чтение криптографического машинного GUID;
- Соединение со случайными доменами;
- Чтение даты установки Windows;
- Удаление исполняемых файлов;
- Присутствие потенциального IP-адреса в бинарном файле;
- Модификация настроек прокси-сервера;
- Установка хуков/патчей к запущенному процессу;
- Инжектирование в explorer;
- Инжектирование в удалённый процесс;
- Запрос информации о процессе;
- Установка процесса в режим ошибки для сокрытия сообщения об ошибке;
- Нетипичная энтропия;
- Возможная проверка на присутствие антивирусного движка;
- Присутствие функционала для расширения привилегий;

- Модификация настроек политик приложений;
- Чтение версии BIOS системы/видеокарты;
- Конец PE-заголовка находится внутри нестандартной секции;
- Создание защищённых областей памяти;
- Создание множества процессов;
- Попытка приостановить работу на долгое время;
- Нестандартные секции;
- Чтение Windows Product Id;
- Присутствие функционала для запуска/взаимодействия с драйверами устройств;
- Присутствие функционала для блокировки пользовательского ввода.

2.5.3 Энтропия

Энтропия представляет собой случайную последовательность, собираемую операционной системой или приложением для использования в криптографических или других целях, где требуются случайные данные. Случайная последовательность часто формируется при помощи аппаратной платформы, например, на базе движения мыши или специальных генераторов случайных чисел. Слабая энтропия может оказать негативное влияние на производительность и безопасность системы.

При шифровании PE-файла серьёзно увеличивается энтропия. Если антивирусный сканер детектирует нетипичный уровень энтропии, вполне возможно, PE-файл будет помечен как подозрительный.

2.5.4 Детектирование запуска в тестовой среде

Аналитические инструменты на компьютерах конечных пользователей обычно сохраняют баланс между глубиной анализа и влиянием на производительность операционной системы пользователя. Многие антивирусные решения

разворачивают на машине конечного пользователя тестовую среду для запуска программного обеспечения (антивирусную песочницу). Это может быть эмуляционная или виртуализационная среда с ограниченными ресурсами и временем на сканирование, иначе такой антивирус будет существенным образом влиять на производительность системы пользователя. Этими ограничениями может пользоваться вредоносное программное обеспечение. Продемонстрируем здесь некоторые техники, которые использует вредоносное ПО для детектирования запуска в антивирусной песочнице на машине конечного пользователя [16].

2.5.4.1 Использование функции *GetTickCount*

Этот метод основан на ограничении по времени на сканирование. Вначале разработчики вредоносного ПО использовали функцию *sleep* с целью дождаться окончания сканирования, но сейчас этот трюк практически не работает, поскольку антивирусные продукты пропускают данную функцию. В статье [16] демонстрируется другой подход, который использует API-функцию *GetTickCount*. Данная функция извлекает количество миллисекунд, прошедших с момента запуска системы (до 49.7 дней). В примере на Рисунке 2 программа вызывает данную функцию, затем происходит остановка на 1 секунду. После функции *sleep* осуществляется проверка, был ли участок сна пропущен, на основании сравнения двух значений, полученных при помощи *GetTickCout*.

```
//                                  }
int Tick = GetTickCount();        //|
Sleep(1000);                      //|
int Tac = GetTickCount();         //|
if((Tac - Tick) < 1000){          //|=> Check if the sleep function is skipped...
    BypassAV(argv,NameTrigger);//|
}                                 //|
//                                  }
```

Рисунок 2 — Проверка выполнения функции *sleep*
Источник: [16]

2.5.4.2 Проверка количества ядер

Этот метод основан на ограничении по ресурсам в среде для тестового запуска. Он проверяет количество ядер процессора, присутствующего в системе (см. Рисунок 3). Многие антивирусы не поддерживают мульти-ядерную обработку и не могут зарезервировать более одного ядра для песочницы.

```
SYSTEM_INFO SysGuide;//
GetSystemInfo(&SysGuide);//
int CoreNum = SysGuide.dwNumberOfProcessors;//
if(CoreNum < 2){//
    return false;//
}//
```

Рисунок 3 — Проверка количества ядер
Источник: [16]

2.5.4.3 Выделение большого объёма памяти

Этот метод основан на ограничении по времени на сканирование. Происходит выделение около 100 Мб памяти, данное пространство заполняется пустыми байтами (см. Рисунок 4). Затем память освобождается. Когда память, используемая программой, начинает расти, через некоторое время антивирусные сканеры останавливаются, чтобы не тратить много времени на анализ файла.

```
char * Memdmp = NULL;
Memdmp = (char *)malloc(100000000);
if (Memdmp != NULL) {
    memset(Memdmp, 00, 100000000);
    free(Memdmp);
}
```

Рисунок 4 — Выделение большого объёма памяти
Источник: [16]

2.5.4.4 Проверка мьютексов

Суть техники заключается в проверке присутствия определённого мьютекса в системе (см. Рисунок 5). Если функция *CreateMutex* не вернёт ошибку

ERROR_ALREADY_EXISTS, вредоносное ПО запускается ещё раз, поскольку большинство антивирусных продуктов не позволяют программам, которые подвергаются динамическому анализу, запускать новые процессы или работать с файлами вне песочницы. Если по результатам проверки мьютекса возвращается ошибка ERROR_ALREADY_EXISTS, можно запускать функцию дешифровки.

```
HANDLE AmberMutex = CreateMutex(NULL, TRUE, "FakeMutex");
if(GetLastError() != ERROR_ALREADY_EXISTS){
    WinExec(argv[0],0);
}
```

Рисунок 5 — Проверка мьютекса
Источник: [16]

2.5.5 Детектирование отладчика

Для детектирования отладчика уровня пользователя (user-mode debugger) может быть применена функция системного API Windows *IsDebuggerPresent*. Однако многие экземпляры вредоносного ПО ожидают перехвата или мониторинга вызова этой функции и могут напрямую искать байт BeingDebuged в PEB-блоке (Process Environment Block, структура данных, описывающая процесс в операционной системе Windows). Фрагмент кода на Рисунке 6 извлекает байт BeingDebuged из PEB-блока. Если отладчик присутствует, проверка повторяется заново, и так до тех пор, пока в стеке не будет переполнение. После переполнения сработает исключение, и процесс будет закрыт. Этот путь — самый простой для выхода из программы.

Здесь также может быть применена обфускация кода, чтобы защититься от статического анализа строк в случае готовности анализатора к такой проверке (см. Рисунок 7). Добавление переходов (инструкций JZ) после каждой операции не повлияет на логику работы функции, но добавление мусорных байтов между переходами позволит обойти фильтры статического анализа строк.

```
//BOOL WINAPI IsDebuggerPresent(void);

__asm{
CheckDebugger:
    PUSH EAX                        //Save EAX value to stack
    MOV EAX,DWORD PTR FS:[0x18]     //Get PEB structure address
    MOV EAX,DWORD PTR [EAX+0x30]    //Get BeingDebuged byte
    CMP BYTE PTR [eax+2],0          //Check if BeingDebug byte is set
    JNE CheckDebugger               //If debugger present check again...
    POP EAX                         //Put back the EAX value
}
```

Рисунок 6 — Функция для детектирования отладчика
Источник: [16]

3 Запуск вредоносного программного обеспечения

Запуск образцов вредоносного ПО под контролем аналитических инструментов требует наличия тестовой среды. Важными являются вопросы автоматизации запуска образцов и получения данных, производительности системы (она определяет количество образцов, которое может быть запущено и проанализировано, например, в течение суток), а также безопасности запуска вредоносного ПО (чтобы не допустить неконтролируемого заражения системы).

В данной главе будет рассказано о возможных тестовых средах для анализа вредоносного программного обеспечения.

3.1 Стратегии имплементации

Запуск вредоносного ПО ставит вопрос о выборе тестовой среды. Рассмотрим потенциальные среды для запуска и анализа вредоносного ПО [2]:

1. Анализ в пространстве пользователя (user-space) или в пространстве ядра (kernel-space);
2. Анализ в эмуляторе;
3. Анализ в виртуальной машине.

Внедрение функций анализа в эмулятор или гипервизор виртуальных ма-

```
__asm
{
CheckDebugger:
    PUSH EAX
        MOV EAX, DWORD PTR FS : [0x18]
        __asm
    {
        PUSH EAX
        XOR EAX, EAX
        JZ J
        __asm __emit(0xea)
        J:
        POP EAX
    }
    MOV EAX, DWORD PTR[EAX + 0x30]
        __asm
    {
        PUSH EAX
        XOR EAX, EAX
        JZ J3
        __asm __emit(0xea)
        J3:
        POP EAX
    }
    CMP BYTE PTR[EAX + 2], 0
    JNE CheckDebugger
    POP EAX
}
```

Рисунок 7 — Функция для детектирования отладчика с применением обфускации

Источник: [16]

шин потенциально позволяет скрыть присутствие анализатора даже от вредоносного ПО, которое выполняется в пространстве ядра. Конечно, вредоносная программа, которая выполняется на более высоком уровне привилегий, чем аналитический компонент, всегда может скрыть себя и избежать анализа. Однако внедрение аналитического инструментария в эмулятор или гипервизор приводит к так называемой *семантической яме* (semantic gap): мы имеем доступ к низкоуровневой информации, которую сложно сопоставить с высокоуровневым представлением.

От выбираемой среды анализа зависит также количество времени, которое требуется для отката среды при исследовании очередного образца. Рассмотрим различные способы восстановления тестовой системы [2]:

1. Программное решение;

Образ жёсткого диска, содержащий среду анализа, после каждого запус-

ка восстанавливается из чистой операционной системы, защищённой от несанкционированного доступа, например, Linux Live CD.

2. Снепшоты (snapshot) виртуальных машин;

   Большинство виртуальных машин и эмуляторов предоставляют механизмы для получения моментального снимка состояния (виртуального диска, CPU, содержимого памяти). В этом случае для воссоздания чистого экземпляра среды не требуется перезагрузка.

3. Аппаратная поддержка снепшотов;

   Существуют аппаратные решения, которые перенаправляют все изменения блоковой структуры диска на другой физический носитель или выделенную рабочую зону (зарезервированные блоки). Сброс системы осуществляется путём сбрасывания изменений. Этот подход требует перезагрузки системы между прогонами анализа. Эти решения архитектурно похожи, например, на технологию S.M.A.R.T. [17] (технология самоконтроля, анализа и отчётности), которая позволяет восстанавливать сбойные блоки из резерва, но в данном случае резервные блоки используются для хранения изменений, вместе с картой соответствия оригинальных и изменённых блоков. Сброс или обновление системы осуществляется либо путём обнуления карты соответствия (данные снова читаются из оригинальных блоков), либо путём предварительного копирования обновлённых блоков в оригинальные.

3.2 Существующие решения

В статье [18] подробно описывается решение, использующее виртуальные машины для запуска вредоносного ПО. Cuckoo Sandbox также использует виртуальные машины [19].

Важным является вопрос об обеспечении Интернет-доступа вредоносному ПО. Если отключить программе доступ в Интернет, мы можем не увидеть часть вредоносного поведения. Если программе разрешён полный доступ к сети Ин-

тернет, это может привести к её участию в рассылке спама и другим нежелательным последствиям. В большинстве случаев применяется симуляция Интернета или фильтрация Интернет-доступа [2]. В статье [20] можно увидеть краткое описание инфраструктурного решения от SANS Institute, в котором также используется эмуляция Интернета.

В статье [2] описываются различные подходы к анализу вредоносного программного обеспечения и обозреваются существующие решения. На Рисунке 8 представлен сравнительный анализ рассмотренных решений.

	Anubis (5.1)	Multipath exp. (5.2)	Clustering (6.1)	CWSandbox (5.3)	Learning & Clas. (6.3)	Norman Sandbox (5.4)	Joebox (5.5)	Ether (5.6)	WiLDCat (5.7)	Hookfinder (5.8)	Justin (5.9.1)	Renovo (5.9.2)	PolyUnpack (5.9.3)	OmniUnpack (5.9.4)	Behav. Spyw. (5.10.1)	TQana (5.10.2)	Panorama (5.10.3)	Behav. Clas. (6.2)
Analysis implementation																		
User-mode component	○	○	○	●	●	○	●	○	○	○	●	○	●	●	●	○	○	●
Kernel-mode component	○	○	○	●	●	○	●	○	●	●	●	●	○	●	●	○	●	●
Virtual machine monitor	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
Full system emulation	●	●	●	○	○	○	○	○	○	●	○	●	○	○	○	●	●	○
Full system simulation	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Analysis targets																		
Single process	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●
Spawned processes	●	○	●	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○
All processes on a system	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	●	●	●
Complete operating system	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	○
Analysis support for																		
API calls	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
System calls	●	●	●	●	●	●	●	●	○	●	○	○	○	●	●	●	●	●
Function parameters	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
File system operations	●	●	●	●	●	●	●	○	○	●	○	○	○	●	●	●	●	●
Process/thread creation	●	●	●	●	●	●	●	○	○	●	○	○	○	○	●	●	●	●
Registry operations	●	●	●	●	●	●	●	○	○	●	○	○	○	●	●	●	●	●
COM components	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○
Detecting dyn. generated code	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	○	○
Restoring packed binaries	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	○	○
W ⊕ X page protection	○	○	○	○	○	○	○	○	○	○	●	●	○	●	○	○	○	○
Multiple layers of packers	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
Signature matching after unpacking	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○
Instruction trace	○	●	●	○	○	○	○	●	●	●	○	●	●	○	○	○	●	○
Information flow tracking	○	●	●	○	○	○	○	○	○	●	○	○	○	○	○	●	●	○
Multiple path exploration	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ASEP monitoring	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○
Networking support																		
Simulated network services	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Internet access (filtered)	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●
Performs cloaking																		
Shadow EFLAGS register	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○
Process hiding	○	○	○	●	●	○	●	○	●	○	○	○	○	○	○	○	○	○
Dynamic module hiding	○	○	○	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○
Disguise modified memory pages	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○
Disguise elapsed time	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○
Clustering Support	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●
Simulated User Interaction	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	●	○

Рисунок 8 — Сравнение инструментов анализа вредоносного ПО
Источник: [2]

В качестве основных критериев сравнения выступают стратегии имплементации тестовой среды, анализируемые процессы, техники динамического анализа (большая часть из них была рассмотрена в разделе, посвящённом аналитическим инструментам, подробнее об этом можно прочитать в статье [2]), предоставление доступа в Интернет и методы маскировки аналитических инструментов.

4 Сведения об ОС Windows

Разработанный нами инструментарий для динамического анализа вредоносного ПО предназначен для операционных систем Windows. В данной главе представлены некоторые сведения из области системного программирования в ОС Windows, использованные при создании соответствующего инструментария.

4.1 PE-формат (Portable Executable)

PE-формат — формат исполняемых файлов в операционных системах Windows. Данному формату соответствуют файлы с такими расширениями, как EXE (Executable, исполняемая программа) или DLL (Dynamic-Link Library, динамически загружаемая библиотека). Структура PE-файла состоит из заголовка (PE-header) и секций, в которых содержатся данные (секция *.data*), код (секция *.text*) и другая информация — например, служебная информация для загрузчика программ может находиться в секции *.rdata* (см. Рисунок 9). Подробнее о PE-формате можно узнать из статьи [21].

4.1.1 Импорты

Для нас основной интерес представляет таблица импортируемых функций, или импортов (import table). Приложения используют функции, предоставляемые (экспортируемые) системными библиотеками (например, библиотекой kernel32.dll). В таблице импортов описаны функции, импортируемые приложением из других модулей, таких как системные библиотеки. Виртуальный адрес структуры, описывающей таблицу импортов PE-файла (в случае нормального импорта это структура IMAGE_IMPORT_DESCRIPTOR), можно получить из следу-

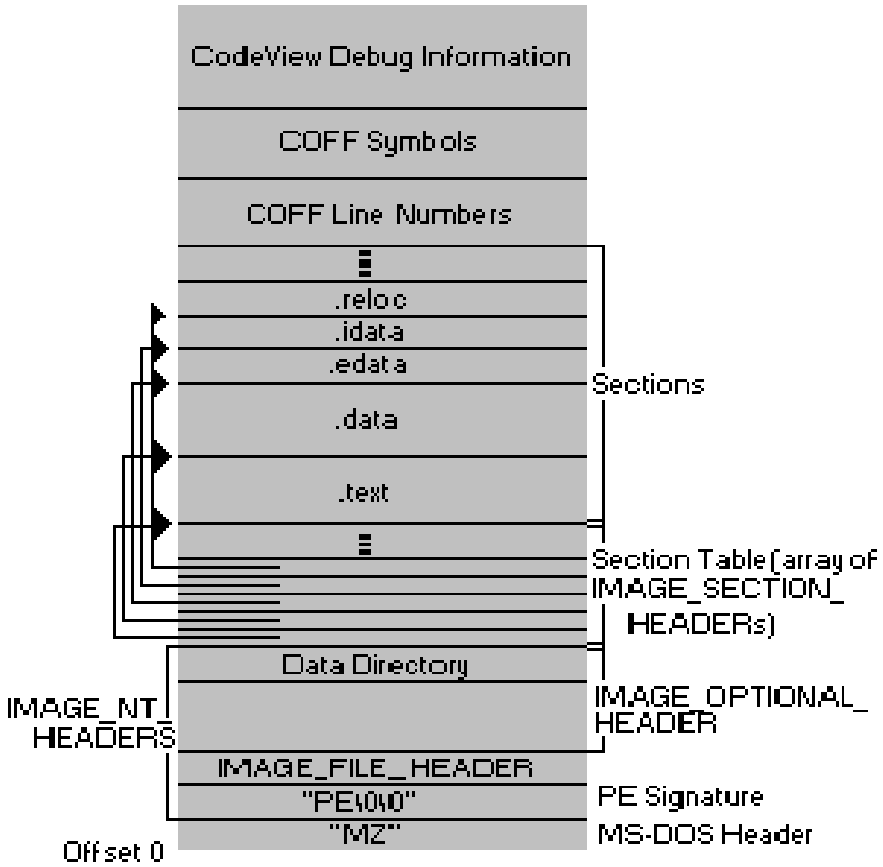


Рисунок 9 — Структура PE-файла
Источник: [21]

ющих структур:

```
typedef struct _IMAGE_NT_HEADERS {
DWORD                    Signature;
IMAGE_FILE_HEADER        FileHeader;
IMAGE_OPTIONAL_HEADER    OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;

typedef struct _IMAGE_OPTIONAL_HEADER {
...
IMAGE_DATA_DIRECTORY  DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;

typedef struct _IMAGE_DATA_DIRECTORY {
DWORD VirtualAddress;
DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

На Рисунке 10 представлен IMPORT_IMAGE_DESCRIPTOR, а также таблица имён функций и таблица адресов функций (Import Address Table, IAT), ко-

торая заполняется системным загрузчиком. Функция может импортироваться по имени или по номеру (ординалу). При загрузке приложения загрузчик PE-файла отображает используемые модули в адресное пространство процесса и заполняет IAT фактическими адресами данных функций в адресном пространстве процесса. Это самый простой и наиболее часто используемый, но самый медленный механизм импорта [22]. По сути это вызов *LoadLibrary* для каждой требуемой библиотеки и затем вызовы *GetProcAddress* для каждой импортируемой функции.

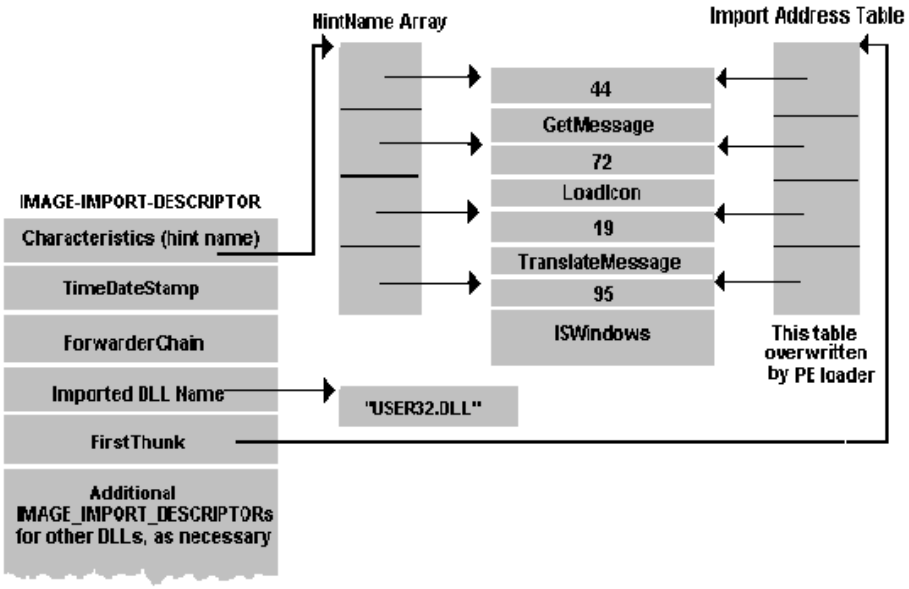


Рисунок 10 — Структуры, описывающие импортируемые функции
Источник: [21]

Существуют 2 других механизма импорта [22; 23]:

1. Отложенные (delay-loaded) импорты;

   Основная идея данного механизма состоит в вызове некоторого обработчика, который должен получить адрес требуемой функции и записать его в таблицу адресов импортируемых функций по мере необходимости, не на этапе загрузки файла, а в процессе его исполнения.

```
typedef struct ImgDelayDescr {
        DWORD grAttrs; // attributes
        LPCSTR szName; // pointer to dll name
        HMODULE* phmod; // address of module handle
```

```
        PimgThunkData pIAT; // address of the IAT
        PCImgThunkData pINT; // address of the INT
        PCImgThunkData pBoundIAT; // address of the optional bound IAT
        PCImgThunkData pUnloadIAT; // address of optional copy of original
            IAT
        DWORD dwTimeStamp; // 0 if not bound,
        // O.W. date/time stamp of DLL bound to Old BIND
} ImgDelayDescr, * PImgDelayDescr;
```

2. Привязанные (binding) импорты.

   В этом случае на адресное пространство исполняемого файла проецируются библиотеки, а в таблице импорта уже заполнены адреса функций. Это быстрый механизм, но любое изменение динамических библиотек повлечет за собой обязательную перекомпиляцию (перелинковку) исполняемого файла, чтобы адреса импортируемых таким образом функций были пересчитаны. Тем не менее, этот механизм используется в Windows для всех стандартных исполняемых файлов (Калькулятор, Блокнот и т. д.), так как стандартные библиотеки Windows меняются крайне редко.

```
typedef struct _IMAGE_BOUND_IMPORT_DESCRIPTOR {
        DWORD TimeDateStamp;
        WORD OffsetModuleName;
        WORD NumberOfModuleForwarderRefs;
        // Array of zero or more IMAGE_BOUND_FORWARDER_REF follows
} IMAGE_BOUND_IMPORT_DESCRIPTOR, *PIMAGE_BOUND_IMPORT_DESCRIPTOR;
```

4.1.2 Экспорты

Теперь поговорим об экспортах. Динамически загружаемые библиотеки экспортируют некоторые свои функции для других модулей. При этом они могут также импортировать функции из других библиотек и даже экспортировать импортируемые из других библиотек функции (в этом случае происходит перенаправление импорта). Структура, описывающая экспорт (IMAGE_EXPORT_DIRECTORY), также может быть получена из структур, описывающих заголовок PE-файла. Данная структура представлена на Рисунке 11.

Для экспортируемых функций существует таблица адресов, таблица имён и таблица ординалов.

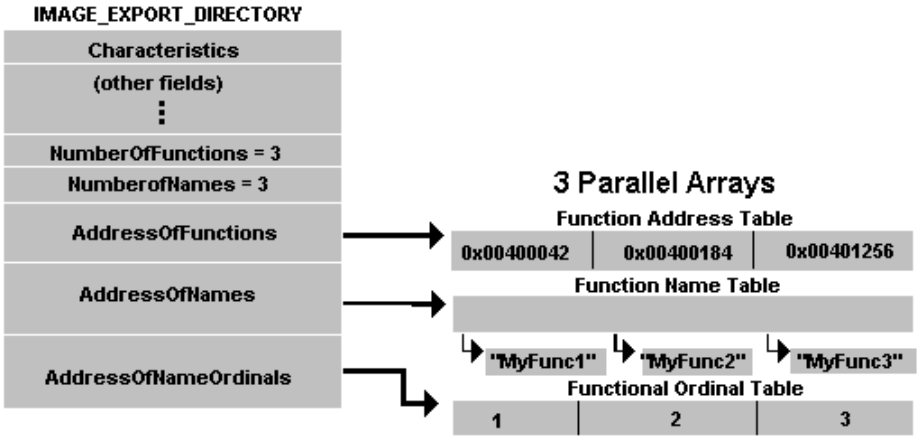


Рисунок 11 — Структуры, описывающие экспортируемые функции
Источник: [21]

4.2 Инжектирование DLL в адресное пространство процесса

Под инжектированием (инжекцией) кода понимается внедрение чужеродного кода в процесс, исполняющийся в операционной системе, или в бинарный файл на диске. Подробнее о том, какие технологии применяет вредоносное ПО для осуществления инжектирования, можно прочитать здесь [24]. Мы остановимся на таком подходе, как классическое инжектирование DLL.

Инжектирование DLL (DLL Injection) — инвазивная технология, позволяющая внедрить чужеродный код в приложение посредством загрузки в его адресное пространство динамически загружаемой библиотеки (DLL). Она основана на том, что в адресное пространство процесса обычно уже загружена системная библиотека kernel32.dll, которая экспортирует функцию *LoadLibrary*:

```
HMODULE WINAPI LoadLibrary(
_In_ LPCTSTR lpFileName
);
```

Данная функция загружает в адресное пространство вызвавшего её процесса модуль, имя которого передаётся ей в качестве параметра — это может быть динамически загружаемая библиотека (.dll) или исполняемый файл (.exe).

4.2.1 Инжекция кода

Инжекция кода в другой процесс может быть осуществлена непосредственно при помощи стандартного набора Windows API, позволяющего:

1. Выделить (allocate) память в целевом процессе с правом на исполнение:

```
LPVOID WINAPI VirtualAllocEx(
_In_      HANDLE hProcess,
_In_opt_  LPVOID lpAddress,
_In_      SIZE_T dwSize,
_In_      DWORD  flAllocationType,
_In_      DWORD  flProtect
);
```

   Для этого достаточно иметь дескриптор (HANDLE) процесса. Подробнее о дескрипторах процессов можно прочитать здесь [25]. Можно запустить целевой процесс самостоятельно при помощи функции *CreateProcess* (в этом случае целевой процесс становится процессом-потомком исходного) или получить дескриптор на уже исполняющийся в операционной системе процесс с помощью функции *OpenProcess* (в этом случае для работы с процессом может потребоваться повышение привилегий).

   За права доступа к выделенной памяти отвечает параметр flProtect, определяющий защиту региона страниц памяти, которые будут выделены по запросу. Например, защита памяти PAGE_EXECUTE_READWRITE позволяет чтение/запись и исполнение кода в выделенном регионе.

2. Записать код в память целевого процесса:

```
BOOL WINAPI WriteProcessMemory(
_In_  HANDLE  hProcess,
_In_  LPVOID  lpBaseAddress,
_In_  LPCVOID lpBuffer,
_In_  SIZE_T  nSize,
_Out_ SIZE_T  *lpNumberOfBytesWritten
);
```

С помощью этой функции можно, например, записать необходимый код в выделенный с помощью *VirtualAllocEx* регион памяти.

3. Исполнить код в целевом процессе:

```
HANDLE WINAPI CreateRemoteThread(
_In_   HANDLE                  hProcess,
_In_   LPSECURITY_ATTRIBUTES   lpThreadAttributes,
_In_   SIZE_T                  dwStackSize,
_In_   LPTHREAD_START_ROUTINE  lpStartAddress,
_In_   LPVOID                  lpParameter,
_In_   DWORD                   dwCreationFlags,
_Out_  LPDWORD                 lpThreadId
);
```

Для этого в целевом процессе создаётся нить, начинающая исполнять код функции, указатель на которую передаётся в переменной lpStartAddress. Переменная имеет тип LPTHREAD_START_ROUTINE:

```
typedef DWORD (__stdcall *LPTHREAD_START_ROUTINE) (
[in] LPVOID lpThreadParameter
);
```

Наиболее удобный подход состоит в том, что необходимый функциональный код, который требуется внедрить (написанный, например, на C), помещают в отдельный модуль (.dll) и инжектируют в целевой процесс небольшой код (обычно написанный на ассемблере), который только загружает .dll в адресное пространство целевого процесса. Такой код называют шеллкод (shellcode). Простейший пример 32-битного шеллкода представлен на Рисунке 12. Он вызывает функцию *LoadLibrary*, передавая ей на вход имя модуля для загрузки (injector.dll). Этот код представляет собой шаблон (стаб), который затем будет переведён в машинный код. При записи кода инструкций в целевой процесс в шаблоне нужно будет заменить сигнатуру 0xDEADFACE на адрес функции *LoadLibrary* в адресном пространстве целевого процесса.

Сигнатура функции *LoadLibrary* совпадает с LPTHREAD_START_ROUTINE,

```
__asm {
        pushad
        call start
    start :
        pop ebx
        sub ebx, start
        mov eax, 0xDEADFACE ; LoadLibrary
        lea edx, [data + ebx] ; 'injector.dll'
        push edx
        call eax
        popad
        ret
    data : 'injector.dll'
}
```

сигнатура

имя инжектируемой .dll

Рисунок 12 — Простейший шеллкод

поэтому можно, не используя shellcode, создать в целевом процессе нить, которая загрузит DLL. Для этого достаточно записать в целевой процесс имя модуля, который требуется загрузить, и передать указатель на него в качестве параметра стартовой процедуры (lpParameter), а в качестве адреса стартовой процедуры (lpStartAddress) указать адрес *LoadLibrary*. Однако это верно только в простейшем случае — подход с shellcode более предпочтителен, потому что, используя более сложный шеллкод, можно сделать важную дополнительную работу (например, найти адрес *LoadLibrary* в адресном пространстве целевого процесса, проверить успешность загрузки инжектируемого модуля и т. д.).

Есть также различные трудности, с которыми предстоит столкнуться при инжектировании DLL:

1. Необходимо найти адрес *LoadLibrary* в памяти;

   Для этого можно найти kernel32.dll в адресном пространстве целевого процесса и вычислить адрес *LoadLibrary* относительно базового адреса моду-

ля (это смещение можно найти, например, воспользовавшись kernel32.dll в собственном процессе). Для нахождения адреса kernel32.dll можно использовать PEB, в котором содержится указатель на структуру PEB_LDR_DATA (поле Ldr), содержащую информацию о загруженных модулях процесса. Ldr в PEB заполняется во время инициализации режима пользователя и равен NULL при создании процесса в приостановленном состоянии.

2. Необходимо не сломать стек для правильной его размотки операционной системой;

   При использовании шеллкода на x64 необходимо зарегистрировать информацию для размотки стека — структуру UNWIND_INFO.

Также есть тонкости работы с приложениями, которые применяют различные техники для защиты от инжектирования.

После этого важно понять, что инжектированная DLL успешно загрузилась. Здесь также полезно использование шеллкода. По возвращаемому значению нити не всегда можно понять, что загрузка произошла не успешно (возвращаемый указатель на модуль будет приведён к типу DWORD):

```
BOOL WINAPI GetExitCodeThread(
_In_ HANDLE hThread,
_Out_ LPDWORD lpExitCode
);
```

А если в процессе загрузки что-то пошло не так, код возврата нити (NULL, приведённый к DWORD) не может прояснить ситуацию — необходим результат вызова функции *GetLastError*.

Чтобы понять, что injector.dll удалось осуществить необходимую работу, или для другого взаимодействия с родительским процессом можно использовать какой-либо сигнал или событие. Простейшим сигналом может быть запись контрольного значения в память, выделенную вместе с шеллкодом. Инжектированная DLL также может отобразить всплывающее окно для взаимодействия с пользователем или взвести событие для общения с родительским процессом.

После этого загруженный код может осуществлять произвольную активность, не связанную с тем, что делает основное приложение. В отдельных случаях может стоять задача интеграции с приложением в целях изменить его поведение.

4.2.2 Модификация работы приложения

Для внесения изменений в работу приложения можно непосредственно модифицировать его код, воспользовавшись *WriteProcessMemory*. Однако это узкоспецифичная задача, поэтому чаще всего патч тел функций приложения не применяется. Вместо этого в адресное пространство процесса внедряется модуль с необходимым функционалом. После этого важной задачей остается перехватить управление потоком исполнения (control flow), т. е. встроиться в работу целевого процесса и передать управление инжектированному коду. Для этого можно пропатчить таблицу импортов приложения, подменив адреса импортируемых функций. На Рисунке 13 представлен пример модификации работы DirectX в целевом процессе. Изменённый функционал содержится в модуле injector.dll. Когда целевой процесс обращается к библиотеке d3d11.dll, происходит вызов с использованием адреса из IAT. Данный адрес был заменён, поэтому управление получит инжектированный модуль. Он производит необходимую активность — это может быть простой мониторинг с передачей управления оригинальному модулю, заглушка без осуществления предполагаемой работы или полноценная собственная реализация необходимого функционала.

4.3 Механизм ApiSetSchema

ApiSetSchema — это механизм перенаправления импортов из виртуальных DLL в логические (хостовые). Механизм впервые появился в Windows 6.x [26]. Функции теперь экспортируются новыми виртуальными модулями, в то время как реальная имплементация расположена в других местах. Разрешение этого механизма импортов происходит при загрузке модуля, поэтому инструменты статического анализа PE-файла (например, Dependency Walker [27]) могут иметь проблемы при работе с этой реорганизацией системных DLL.

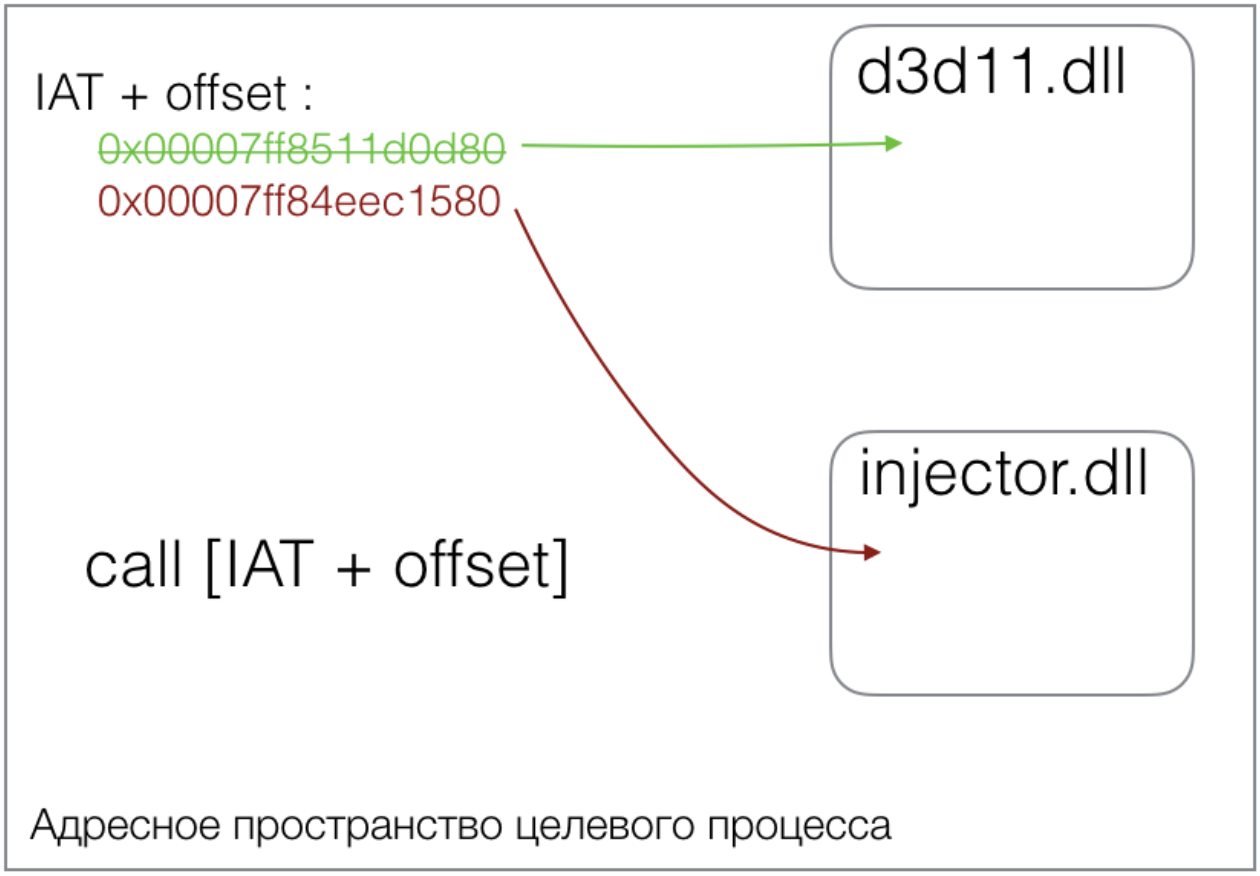


Рисунок 13 — Модификация работы DirectX с помощью патча IAT

Компоненты Windows были реорганизованы с введением MinWin (Minimal Windows Kernel — Минимальное ядро Windows). В процессе реорганизации в системном API были выделены слои, таким образом, чтобы API нижнего уровня никогда не вызывали более высокоуровневое API. Вследствие этого каждый уровень API может развиваться независимо.

Концепция MinWin была введена в Windows 7. Принцип её реализации основан на разделении ядра на слои, где базовый слой (KernelBase.dll и т. д.) суммарно занимает 30 Мб, а остальная функциональность остаётся в старых библиотеках (к примеру, kernel32.dll и т. д.) [28]. Для работы приложений и драйверов устройств это изменение незаметно благодаря тщательно продуманной схеме разделения функциональности.

Теперь системные DLL перенаправляют вызовы функций в MinWin DLL [26], например:

- Kernel32.dll → KernelBase.dll
- Advapi32.dll → KernelBase.dll

Проблема заключается в том, что после такой реорганизации базовые DLL содержат несколько логических наборов API. Например, KernelBase теперь будет содержать различные логические наборы, такие как обработка реестровых операций, обработка операций с процессами и потоками, работа с памятью и т. д. Чтобы уклониться от возникающих трудностей, Microsoft отделила архитектуру API от имплементации API с помощью механизма виртуальных DLL. Отображение между виртуальными DLL и DLL с имплементацией осуществляется с помощью одной DLL, которая называется ApiSetSchema.dll. Наименование ApiSetSchema используется для обозначения этого конкретного механизма, изображённого на Рисунке 14.

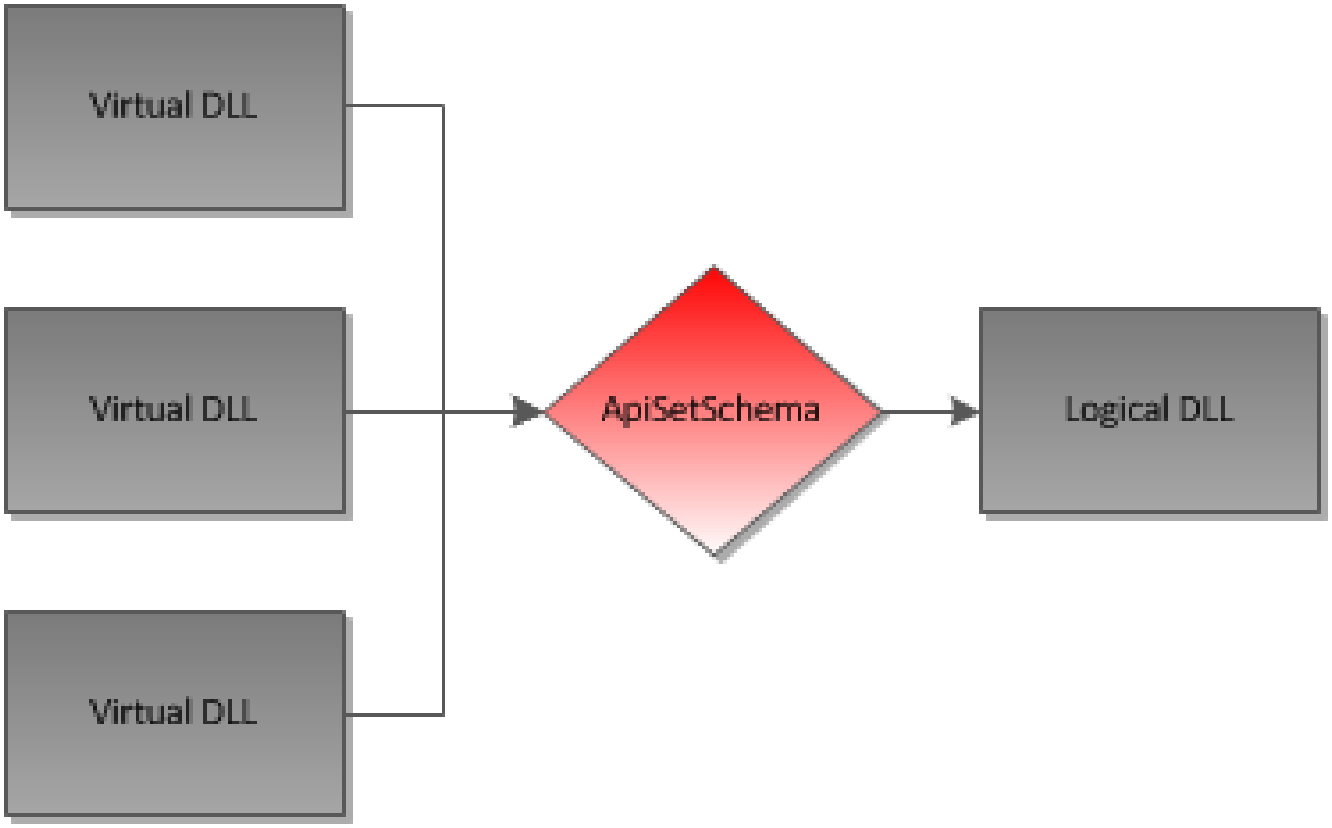


Рисунок 14 — Механизм ApiSetSchema
Источник: [26]

Таким образом, весь механизм ApiSetSchema основан на одном файле, ApiSetSchema.dll, расположенном в директории \%SystemRoot%\System32. Этот

модуль грузится на раннем этапе загрузки операционной системы процессом winload.exe одновременно с такими важными образами, как ядро Windows и слой абстракции аппаратного обеспечения (Hardware Abstraction Layer, HAL).

ApiSet DLL — это «виртуальные библиотеки», фактически реализующие только контракт по предоставлению функций Windows API, которые будут разрешены загрузчиком NT [29]. Согласно презентации А. Ионеску «Esoteric Hooks» (которую также можно найти по ссылке [29]), официальная цель данной реорганизации состоит в том, чтобы отделить функции API (выделение памяти в куче, манипуляция со строками и т. д.) от имплементации подсистем (работа с процессами, реестром и т. д.).

API Set — это определённое имя для набора Win32 API. Соглашение о присвоении определённого имени набору API — то, что происходит при использовании имени DLL [30]. Цель API Set — обеспечить архитектурное разделение между именем набора API и ассоциированной с ним хостовой DLL для улучшения переносимости приложений. Таким образом, имя API Set — это лишь уникальная символьная строка, а не имя DLL. Данная строка используется для отложенной загрузки модуля.

API Set полагается на поддержку операционной системы в реализации загрузчика библиотек, чтобы эффективно внедрить перенаправление имён в процесс привязки библиотеки [30]. В зависимости от различных входных данных, включая имя API Set и контекст привязки импорта, загрузчик в момент исполнения осуществляет перенаправление ссылки на целевой хостовой бинарный файл, в котором расположена соответствующая реализация набора API. Разделение реализации и интерфейса, предоставляемого API Set, дает множество инженерных преимуществ и может потенциально уменьшить количество DLL, загруженных в процесс.

Виртуальные DLL начинаются с префикса api- (api-ms-win-) и в Windows 10 расположены в \%SystemRoot%\System32\downlevel. В них представлено, например, API для взаимодействия с COM-компонентами (api-ms-win-core-com-l1-

1-0.dll), разрешающееся в combase.dll. Возможен также префикс ext- [29].

Отображение между виртуальной api-dll и хостовой DLL имплементировано в особой секции модуля ApiSetSchema.dll, которая называется .apiset (см. Рисунок 15) [29]. Эта секция представлена для каждого процесса в его PEB (см. Рисунок 16). Возможно, это реализовано с применением Copy On Write механизма.

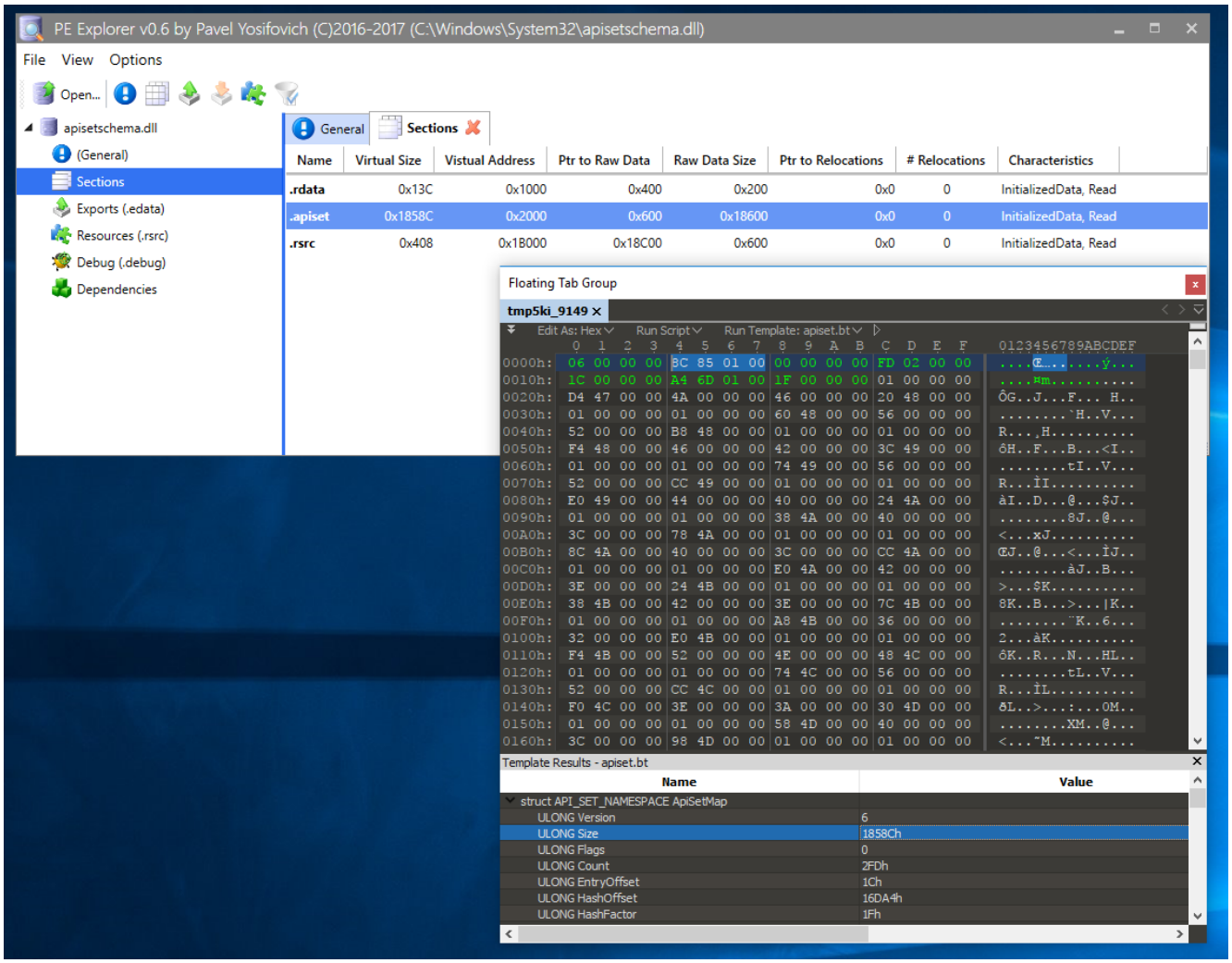


Рисунок 15 — Секция .apiset модуля ApiSetSchema.dll
Источник: [29]

Поле PEB.ApiSetMap указывает на структуру API_SET_NAMESPACE [29]:

```
typedef struct {
        ULONG Version;      // v2 on Windows 7, v4 on Windows 8.1 and v6
            on Windows 10
        ULONG Size;         // apiset map size (usually the .apiset section
            virtual size)
        ULONG Flags;        // according to Geoff Chappell, tells if the
            map is sealed or not.
```

```
Pid 17288 - WinDbg:10.0.16232.1000 AMD64
File  Edit  View  Debug  Window  Help
Command
0:015> dt nt!_PEB 000000f558212000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x4 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 IsPackagedProcess : 0y0
   +0x003 IsAppContainer   : 0y0
   +0x003 IsProtectedProcessLight : 0y0
   +0x003 IsLongPathAwareProcess : 0y0
   +0x004 Padding0         : [4]  ""
   +0x008 Mutant           : 0xffffffff`ffffffff Void
   +0x010 ImageBaseAddress : 0x00007ff7`e9af0000 Void
   +0x018 Ldr              : 0x00007fff`4bc0f3a0 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x000002d6`b0ea28c0 _RTL_USER_PROCESS_PARAMETERS
   +0x028 SubSystemData    : (null)
   +0x030 ProcessHeap      : 0x000002d6`b0ea0000 Void
   +0x038 FastPebLock      : 0x00007fff`4bc0eec0 _RTL_CRITICAL_SECTION
   +0x040 AtlThunkSListPtr : (null)
   +0x048 IFEOKey          : (null)
   +0x050 CrossProcessFlags : 0
   +0x050 ProcessInJob     : 0y0
   +0x050 ProcessInitializing : 0y0
   +0x050 ProcessUsingVEH  : 0y0
   +0x050 ProcessUsingVCH  : 0y0
   +0x050 ProcessUsingFTH  : 0y0
   +0x050 ProcessPreviouslyThrottled : 0y0
   +0x050 ProcessCurrentlyThrottled : 0y0
   +0x050 ReservedBits0    : 0y000000000000000000000000 (0)
   +0x054 Padding1         : [4]  ""
   +0x058 KernelCallbackTable : 0x00007fff`4b2921e0 Void
   +0x058 UserSharedInfoPtr : 0x00007fff`4b2921e0 Void
   +0x060 SystemReserved   : 0
   +0x064 AtlThunkSListPtr32 : 0
   +0x068 ApiSetMap        : 0x000002d6`aed90000 Void
   +0x070 TlsExpansionCounter : 0
   +0x074 Padding2         : [4]  ""
   +0x078 TlsBitmap        : 0x00007fff`4bc0f320 Void
   +0x080 TlsBitmapBits    : [2] 0x1fffff
   +0x088 ReadOnlySharedMemoryBase : 0x00007ff7`e8e40000 Void
   +0x090 SharedData       : (null)
   +0x098 ReadOnlyStaticServerData : 0x00007ff7`e8e40730  -> (null)
   +0x0a0 AnsiCodePageData : 0x00007ff7`e8f40000 Void
   +0x0a8 OemCodePageData  : 0x00007ff7`e8f50228 Void
   +0x0b0 UnicodeCaseTableData : 0x00007ff7`e8f60650 Void
0:015>
Ln 0, Col 0  Sys 0:<Local>  Proc 000:4388  Thrd 015:426c  ASM  OVR  CAPS  NUM
```

Рисунок 16 — ApiSetMap в PEB процесса
Источник: [29]

```
        ULONG Count;          // hash table entry count
        ULONG EntryOffset;    // Offset to the api set entries values
        ULONG HashOffset;     // Offset to the api set entries hash indexes
        ULONG HashFactor;     // multiplier to use when computing hash
} API_SET_NAMESPACE;
```

Данная структура представляет собой хеш-таблицу, состоящую из следующих структур:

```
// Hash table index (just an optimization "trick")
typedef struct {
        ULONG Hash;
        ULONG Index;
} API_SET_HASH_ENTRY;
```

```
// Hash table value
typedef struct {
        ULONG Flags;          // sealed flag in bit 0
        ULONG NameOffset;     // Offset to the ApiSet library name PWCHAR (e
            .g. "api-ms-win-core-job-l2-1-1")
        ULONG NameLength;     // Ignored
        ULONG HashedLength;   // Apiset library name length
        ULONG ValueOffset;    // Offset the list of hosts library implement
            the apiset contract (points to API_SET_VALUE_ENTRY array)
        ULONG ValueCount;     // Number of hosts libraries
} API_SET_NAMESPACE_ENTRY;

// Host Library entry
typedef struct {
        ULONG Flags;          // sealed flag in bit 0
        ULONG NameOffset;     // Offset to the ApiSet library name PWCHAR (e
            .g. "api-ms-win-core-job-l2-1-1")
        ULONG NameLength;     // Apiset library name length
        ULONG ValueOffset;    // Offset to the Host library name PWCHAR (e.g
            . "ucrtbase.dll")
        ULONG ValueLength;    // Host library name length
} API_SET_VALUE_ENTRY;
```

Может показаться, что отображение виртуальной DLL в хостовую однозначно, однако это не совсем так, и внутри одной виртуальной DLL могут быть функции, разрешающиеся в различные хостовые DLL. Отображение имплементировано в 3 функциях системной библиотеки ntdll.dll, которые не экспортируются:

- *ApiSetResolveToHost* используется для высокоуровневого разрешения (получает на вход имя в формате UNICODE_STRING и возвращает заполненную структуру UNICODE_STRING);
- *ApiSetpSearchForApiSet* имплементирует поиск по хеш-таблице ApiSetMap;
- *ApiSetpSearchForApiSetHost* определяет DLL в случае, когда виртуальная DLL имеет несколько хостовых DLL (это не распространённый случай).

ApiSetpSearchForApiSet — традиционная функция поиска по хеш-таблице. Поскольку могут происходить коллизии, есть дополнительная проверка имени библиотеки apiset, возвращаемого хеш-таблицей.

```
PAPI_SET_NAMESPACE_ENTRY
__fastcall ApiSetpSearchForApiSet(
_In_ PAPI_SET_NAMESPACE ApiNamespace,
_In_ PWCHAR ApiNameToResolve,
_In_ uint16_t ApiNameToResolveSize
);
```

Поскольку возвращаемая запись может ссылаться на несколько хостовых библиотек, есть дополнительная функция, чтобы вернуть точное значение хостовой DLL на основании переменной ParentName:

```
PAPI_SET_VALUE_ENTRY
__stdcall ApiSetpSearchForApiSetHost(
_In_ PAPI_SET_NAMESPACE_ENTRY Entry,
_In_ PWCHAR *ParentName,
_In_ SHORT ParentNameLen,
_In_ PAPI_SET_NAMESPACE ApiNamespace
);
```

Если ParentName равно NULL, возвращается значение по умолчанию. *ApiSetResolveToHost* — это обертка для предыдущих двух функций, скрывающая детали имплементации хеш-таблицы. При этом также

1. Происходит проверка, что имя библиотеки имеет префикс api- или ext-;
2. Имя библиотеки усекается перед тем, как будет передано в *ApiSetpSearchForApiSet*: не используется расширение .dll и часть после последнего дефиса. Например, api-ms-win-core-apiquery-l1-1-0.dll будет усечено до api-ms-win-core-apiquery-l1-1.

Итак, механизм ApiSetSchema работает следующим образом [26]:

1. Windows использует механизм для перенаправления виртуальных DLL в логические DLL с имплементацией API;

2. Во время старта системы Winload.exe загружает файл ApiSetSchema.dll;

3. Во время первой фазы инициализации ядра секция данного модуля .apiset отображается в системную память ядра: создаётся секция (объект режима ядра), которая проецируется в память (map view);

4. Когда процесс запускается, отображение секции создаётся в адресном пространстве процесса и доступно через поле _PEB.ApisetMap: таким образом, содержимое секции .apiset из ApiSetSchema.dll доступно в пространстве пользователя.

4.4 Механизм SHIMS

Shim — это небольшая библиотека, которая прозрачно вмешивается в вызовы API и изменяет передаваемые аргументы, обрабатывает запрос самостоятельно или перенаправляет его в другое место [31]. Может быть использована для поддержки старого API в новом окружении или нового API в старом окружении. Также может быть использована для запуска приложений на программных платформах, отличных от той, для которой они были разработаны.

Shims («заплатки») для более старых API обычно появляются, когда поведение API меняется, вызывая проблемы совместимости для более старых приложений, полагающихся на старую функциональность. В таких случаях старый API всё ещё может поддерживаться тонким слоем обратной совместимости поверх нового кода. Shims для более новых API обычно реализуют новый API для старой среды, используя только средства этой среды.

Microsoft Windows Application Compatibility ToolKit (ACT) использует данный механизм, чтобы обеспечить обратную совместимость библиотек. Shims симулирует поведение более старых версий Windows для устаревших (legacy) приложений, полагающихся на некорректную или запрещённую функциональность, или исправляют путь, которым плохо написанные приложения вызывают не изменившийся API (например, чтобы исправить least-priveledged user account (LUA) баги).

Данная подсистема реализована в виде набора динамически загружаемых библиотек Application Compatibility Infrastructure (библиотека функций shimeng.dll содержит сам механизм Shim Engine, а apphelp.dll — Application Compatibility Interface, Application Compatibility Client Library) [32]. Их работа обслуживается средствами загрузчика ntdll.dll с использованием ряда ключей реестра и системной базы данных.

Смысл наложения «системных заплаток» состоит в том, что в реестр вносятся записи, соответствующие запускаемому приложению и служащие индикатором для загрузчика, чтобы при запуске этой программы производилось подключение механизма Shim Engine, выбор «системных заплаток» из базы данных sysmain.sdb и их применение согласно указанному методу. Фактически «shims» выполняют роль перехватчиков вызовов системных функций. Нужный эффект достигается благодаря замещению элементов в таблице импорта Import Address Table (IAT) исполняемого файла, что позволяет при обработке исполняемого файла Windows PE (Portable Executable) перехватывать вызов системных функций с использованием таблиц импорта и экспорта функций.

Системная база данных с информацией обо всех системных и дополнительных (для конкретных приложений) заплатках для применяемой версии ОС располагается в каталоге %SystemRoot%\AppPath. Здесь же находятся ключевые загружаемые dll-библиотеки с новым кодом функций, отличным от стандартного. Системные «заплатки» находятся в файлах acgenral.dll и aclayers.dll, дополнительные — в acspecfic.dll. Существуют и другие группы «системных заплаток», которые используются, когда проблемы совместимости невозможно решить простой заменой вызовов системных функций.

Необходимость применения «системной заплатки» определяется при создании каждого нового процесса. Это осуществляется в подсистеме Win32 (kernel32.dll), где функция создания новых процессов *CreateProcess* получает через внутреннюю системную функцию *Kernel32!CreateProcessInternalW* дополнительный код, обеспечивающий совместимость обрабатываемого приложения. Это

просто управляющий код, который помогает позвать Windows API, обладающий привилегиями пользовательского кода.

Как многие специфические для Win32 особенности, Shim Engine инициализируется родительским процессом в kernel32.dll, а не процедурами загрузчика PE-файла (PE Loader/Startup) внутри системной NT библиотеки (ntdll.dll) [33]. Функция *CreateProcessInternalW* вызывает *BasepCheckBadApp* (которая является частью экспортируемого API). Эта процедура должна проверить, нуждается ли приложение в заплатках. Вначале осуществляется проверка, включён ли Shims Engine. После этого в процедуре *BaseCheckAppcompatCache* происходит поиск в кэше обратной совместимости приложений (Application Compatibility Shim Cache).

Кэш имплементирован двумя способами, в зависимости от операционной системы. До Windows 2003 kernel32 поддерживала разделяемую секцию (shared section), которую можно было использовать для кэширования информации, при этом для доступа к кэшу были реализованы блокировки. После Windows 5.2 появилось новое Native API, *NtApphelpCacheControl*. В обоих случаях, если в кэше информации не нашлось, происходит длинная процедура поиска (lookup). В её реализации архитектурные различия сильнее. В Windows XP SP0 используется CSRSS и вызов *BaseSrvCheckApplicationCompatibility* в basesrv.dll. В SP2 импортируется модуль apphelp.dll и функция *ApphelpCheckExe* вызывается напрямую. В Windows Server 2003 происходит подключение к LPC порту и посылается LPC-сообщение. Наконец, в Windows Vista вернулись к методу SP2, обновив API до *ApphelpCheckExeEx*.

По результатам проверок поле pShimData в PEB процесса (см. Рисунок 17) заполняется Apphelp-информацией, если это действительно «плохое» приложение (в том смысле, что оно нуждается в заплатках).

Когда происходит вызов *LdrpInitializeProcess* (часть загрузчика PE-файла в ntdll.dll), проверяется, что указатель pShimData не равен NULL. В этом случае указатель сохраняется и очищается, и *LdrpLoadShimeEngine* загружает Shim

```
+0x128 ImageSubsystem   : Uint4B
+0x12c ImageSubsystemMajorVersion : Uint4B
+0x130 ImageSubsystemMinorVersion : Uint4B
+0x134 Padding4         : [4] UChar
+0x138 ActiveProcessAffinityMask : Uint8B
+0x140 GdiHandleBuffer  : [60] Uint4B
+0x230 PostProcessInitRoutine : Ptr64     void
+0x238 TlsExpansionBitmap : Ptr64 Void
+0x240 TlsExpansionBitmapBits : [32] Uint4B
+0x2c0 SessionId        : Uint4B
+0x2c4 Padding5         : [4] UChar
+0x2c8 AppCompatFlags   : _ULARGE_INTEGER
+0x2d0 AppCompatFlagsUser : _ULARGE_INTEGER
+0x2d8 pShimData        : Ptr64 Void
+0x2e0 AppCompatInfo    : Ptr64 Void
+0x2e8 CSDVersion       : _UNICODE_STRING
+0x2f8 ActivationContextData : Ptr64 _ACTIVATION_CONTEXT_DATA
+0x300 ProcessAssemblyStorageMap : Ptr64 _ASSEMBLY_STORAGE_MAP
+0x308 SystemDefaultActivationContextData : Ptr64 _ACTIVATION_CONTEXT_DATA
+0x310 SystemAssemblyStorageMap : Ptr64 _ASSEMBLY_STORAGE_MAP
+0x318 MinimumStackCommit : Uint8B
2: kd>
```

Рисунок 17 — Shims Data в PEB процесса

Engine DLL. Затем в функции *LdrpGetShimEngineInterface* устанавливается большое количество обратных вызовов, которые в основном состоят из пре- и пост-инициализации и нотификаций о загрузке и выгрузке DLL.

Позже во время инициализации, если Shime Engine был предварительно загружен, вызывается пре-инициализационный хук, и старый указатель pShimData передаётся Shime Engine, чтобы она могла начать инициализацию (функция *SE_InstallBeforeInit* в shieng.dll). Большая часть работы выполняется *SeiGetShimData* (эта функция распаковывает информацию из переданного ей указателя pShimData, а также проверяет, отключены ли заплатки для ntsd и windbg) и *SeiInit* (эта функция обрабатывает все уровни совместимости, заплатки, флаги и, наконец, патчит определённые в Apphelp записи исполняемого файла).

Заплатки обычно состоят из внутренних флагов, которые хранятся в shimeng.dll или PEB (AppCompatFlags, AppCompatFlagsUser, см. Рисунок 17). Также может применяться патч IAT процесса, в ходе которого импорты приложения перенаправляются в один из модулей Ac***.dll, которые содержат альтернативную имплементацию API. Они содержат 2 основных экспорта (см. Рисунок 18): *GetHookApis* и *NotifyShims*, которые позволяют Shims Engine узнать, какие функции должны быть перехвачены, и отправлять уведомления во время событий загрузчика. Механизм Shim Engine является интеллектуальным и также перехватывает функцию *GetProcAddress*, чтобы быть уверенным, что будет использоваться

правильная версия API.

Name	Address	Ordinal
GetHookAPIs	0000000180001430	1
NotifyShims	0000000180001210	2
DllEntryPoint	0000000180004430	[main entry]

Рисунок 18 — Эскпорты AcLayers.dll (%SystemRoot%\SysWOW64) в дизассемблере Ida PRO

4.5 SEH и обработка исключений

Обработка исключительных ситуаций (exceptions) в ОС Windows доступна в режиме пользователя с помощью механизма SEH (и его расширения VEH).

Исключение — событие, которое происходит во время выполнения программы и требует выполнения кода вне нормального потока исполнения [34]. Есть два вида исключений: аппаратные исключения (hardware exceptions) и программные исключения (software exception). Аппаратные исключения инициируются CPU. Они могут произойти в результате выполнения определённых инструкции, таких как деление на нуль или попытка получить доступ к недопустимому адресу памяти. Программные исключения инициируются явно приложениями или операционной системой. Например, система может обнаружить, когда значение параметра является недопустимым.

CPU и операционная система обеспечивают передачу информации о произошедших исключительных ситуациях пользовательским обработчикам или отладчику режима пользователя. Когда происходит исключительная ситуация, в Intel-архитектуре управление передаётся в обработчик режима ядра, который определён в IDT (interrupt description table) [35]. Затем управление передаётся в функцию ntdll.dll *KiUserExceptionDispatcher*, которая вызывает *RtlDispatchException*. Статья [36] описывает цепочку обработки исключения, куда входит первое уведомление отладчика, поиск обработчика VEH/SEH и размотка стека, второе уведомление отладчика. Если процесс не находится под отладкой или обработки исключения не происходит, система использует обработчик

по умолчанию, который зависит от типа исключения. Для большинства исключительных ситуаций происходит вызов функции *ExitProcess*.

4.5.1 Обработка исключений

Исключения могут инициироваться аппаратными средствами или программным обеспечением и могут произойти как в коде режима ядра (kernel-mode), т. е. коде привилегированного режима, так и в коде режима пользователя (user-mode), т. е. непривилегированном коде [37].

Выполнение определенных последовательностей инструкции может привести к исключениям, которые инициируются аппаратными средствами. Например, нарушение прав доступа (access violation) генерируется аппаратными средствами, когда процесс пытается читать или писать, обращаясь по виртуальному адресу, к которому у него нет необходимого доступа.

События, которые требуют обработки исключений, могут также иметь место во время выполнения программной функции (например, при использовании недопустимого значения параметра). Когда это происходит, поток может явно инициировать исключение, вызвав функцию *RaiseException*. Эта функция позволяет вызывающему потоку указать информацию, которая описывает произошедшую исключительную ситуацию.

Исключение может быть допускающим продолжение исполнения (continuable) или не допускающим дальнейшее исполнение (noncontinuable). Исключение второго типа возникает, когда исполнение не может быть продолжено аппаратно или продолжение не имеет никакого смысла. Оно не приводит к принудительному завершению процесса, так что приложение может поймать (catch) исключительную ситуацию и продолжить работу. Однако такие исключения обычно возникают в результате повреждения стека или другой серьёзной проблемы, делая восстановление работоспособности довольно сложной задачей.

4.5.2 Structured Exception Handling

Структурированная обработка исключений (Structured Exception Handling, SEH) — механизм для обработки обоих типов исключений (аппаратных и программных) [34]. Этот код обработает аппаратные и программные исключения одинаковым образом. Структурированная обработка исключений обеспечивает единый механизм для обработки исключений привилегированного и непривилегированного режима [37]. Структурированная обработка исключений позволяет иметь полный контроль над обработкой исключений, предоставляет поддержку для отладчиков и применима на всех языках программирования и машинах [34]. Векторная обработка исключений (Vectored Exception Handling, VEH) — расширение структурированной обработки исключений.

Система также поддерживает обработку завершения (termination handling), которая позволяет гарантировать, что каждый раз, когда защищённый (guarded) блок кода выполняется, выполняется также указанный блок завершения. Код завершения выполнится независимо от того, как поток управления выходит из защищённого кода. Например, обработчик завершения может гарантировать, что освобождение ресурсов произойдет, даже если исключение или некоторая другая ошибка происходит во время выполнения защищённого блока кода.

Структурированная обработка исключений и механизмы обработки завершения — неотъемлемые части системы, которые позволяют ей быть устойчивой к отклонениям от нормальной работы [38]. Их рекомендуется использовать для создания устойчивых и надёжных приложений.

Структурированная обработка исключений доступна, прежде всего, через поддержку компилятора. Например, Microsoft C/C++ Optimizing Compiler поддерживает ключевое слово **__try**, которое идентифицирует защищённый блок кода, ключевое слово **__except**, которое идентифицирует обработчик исключений, и ключевое слово **__finally**, которое идентифицирует обработчик завершения.

4.5.3 Frame-Based Exception Handling

Фреймовый (frame-based) обработчик исключений предполагает, что исключение может произойти в определённом блоке кода [39]. Он состоит из следующих элементов:

1. Защищённый (guarded) код;
2. Фильтрующее выражение (filter expression);
3. Блок обработчика исключений.

Фреймовые обработчики исключений имеют специфичный для языка синтаксис. Например, в Microsoft C/C++ Optimizing Compiler используются ключевые слова **__try** и **__except**.

Защищённый код представляет собой последовательность из одного или нескольких операторов. Это может быть как блок кода, так и ряд вложенных блоков, или же код процедуры или функции полностью. В Microsoft C/C++ Optimizing Compiler защищённый код расположен в фигурных скобках ({}) после ключевого слова **__try**.

Фильтрующее выражение фреймового обработчика проверяется системой, когда в защищённом коде происходит исключение. После этого происходит одно из следующих действий:

- Система прекращает поиск обработчика исключений, восстанавливает состояние машины и продолжает выполнение потока с того места, в котором произошло исключение;
- Система продолжает поиск обработчика исключений;
- Управление передаётся обработчику исключений, и выполнение потока продолжается последовательно в стековом фрейме, в котором был найден

обработчик исключений. Если обработчик исключений не найден в стековом фрейме, в котором произошло исключение, система осуществляет размотку стека (stack unwind), оставляя текущий стековый фрейм и последующие стековые фреймы, пока не найдёт обработчик исключений. Перед выполнением кода обработчика исключений, выполняются обработчики завершения всех участков защищённого кода, которые завершились в результате передачи управления обработчику исключений.

4.5.4 VEH

Векторная обработка исключений (Vectored Exception Handling, VEH) — расширение механизма SEH [40]. Приложение может зарегистрировать функцию, чтобы следить за всеми происходящими исключительными ситуациями и обрабатывать их. Векторные обработчики не связаны со стековым фреймом, поэтому обработчик будет вызван независимо от места возникновения исключительной ситуации. Векторные обработчики исключений вызываются в том порядке, в котором они были зарегистрированы. Вызов векторных обработчиков исключений происходит после того, как получает первое уведомление присоединённый отладчик (first chance notification), но перед тем, как система начнёт разматывать стек.

Чтобы добавить векторный обработчик исключений (Vectored Exception Handler), используется функция *AddVectoredExceptionHandler*:

```
PVOID WINAPI AddVectoredExceptionHandler(
_In_ ULONG                        FirstHandler,
_In_ PVECTORED_EXCEPTION_HANDLER  VectoredHandler
);
```

Первый её параметр отвечает за порядок, в котором будут вызываться зарегистрированные обработчики исключений. Второй параметр является указателем на обработчик исключений, который необходимо позвать:

```
LONG CALLBACK VectoredHandler(
_In_ PEXCEPTION_POINTERS ExceptionInfo
);
```

Обработчик исключений получает доступ к информации о произошедшем

исключении посредством структуры EXCEPTION_POINTERS:

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT          ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Параметр ExceptionRecord содержит машинно-независимое описание исключительной ситуации:

```
typedef struct _EXCEPTION_RECORD {
DWORD                 ExceptionCode;
DWORD                 ExceptionFlags;
struct _EXCEPTION_RECORD  *ExceptionRecord;
PVOID                 ExceptionAddress;
DWORD                 NumberParameters;
ULONG_PTR             ExceptionInformation[
    EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

Адрес, по которому произошла исключительная ситуация, содержится в поле ExceptionAddress. Информацию о том, какого рода исключение произошло, можно найти в поле ExceptionCode. Здесь содержится код, сгенерированный аппаратным исключением или переданный программно функцией *RaiseException*. Например, он может принимать значения EXCEPTION_ACCESS_VIOLATION, EXCEPTION_STACK_OVERFLOW и т. д. В массиве ExceptionInformation содержится информация, специфичная для различного рода исключений.

Обработчик исключений может проверить причину возникновения исключительной ситуации и обработать исключение. После обработки исключения обработчик возвращает EXCEPTION_CONTINUE_EXECUTION. В этом случае управление передаётся в место возникновения исключения (происходит retry). Обработчик также может вернуть EXCEPTION_CONTINUE_SEARCH — в этом случае система передаст управление следующему по очереди зарегистрированному векторному обработчику исключений или, в случае отсутствия таковых, перейдёт к следующему этапу цепочки обработки исключения.

Чтобы удалить зарегистрированный обработчик исключений, используется

функция *RemoveVectoredExceptionHandler*.

Также существует векторный обработчик продолжения исполнения (Vectored Continue Handler). Он вызывается после того, как обработчик исключения вернул значение EXCEPTION_CONTINUE_EXECUTION.

Чтобы зарегистрировать векторный обработчик продолжения исполнения, используют функцию *AddVectoredContinueHandler*. Удалить зарегистрированный обработчик можно с помощью функции *RemoveVectoredContinueHandler*.

5 Создание программно-аппаратного комплекса для запуска вредоносного ПО и разработка инструментария для динамического анализа

В данном разделе содержится описание проведённого исследования и представлены результаты работы. Основная часть работы состояла в разработке прототипа комплекса для запуска вредоносного программного обеспечения (Sandbox) и инструментов динамического анализа (ToolChain). Также в данном разделе описываются наиболее интересные проблемы, с которыми мы столкнулись в процессе прототипирования, и реализованные нами решения. Ниже представлена ключевая терминология, которая используется при описании исследования. Первая часть раздела посвящена разработке прототипа Sandbox, вторая — разработке прототипа ToolChain.

5.1 Терминология

При описании работы использовалась следующая терминология. Некоторые из данных терминов уже встречались в предыдущих разделах, мы приводим их повторно для полноты и независимости описания.

1. Разработка программно-аппаратного комплекса для запуска вредоносного программного обеспечения (Sandbox):

    1.1. **Тестовый сервер** — сервер, предназначенный для запуска вредоносного программного обеспечения.

    1.2. **Управляющий сервер** — сервер, предназначенный для управления

Sandbox, обеспечивающий цикл развёртывания и запуск различных сценариев на тестовых серверах.

1.3. **Тестовая операционная система** — операционная система, работающая на тестовых серверах и предназначенная для запуска вредоносного ПО.

1.4. **Управляющая операционная система** — операционная система, работающая на тестовых серверах и предназначенная для осуществления действий, необходимых для поддержки развёртывания Sandbox (установка тестовой ОС, создание резервной копии тестовой ОС, восстановление тестовой ОС из резервной копии).

1.5. **Агент** — программное обеспечение, установленное на тестовой операционной системе, осуществляющее анализ состояния системы и создающее логи для последующего анализа.

1.6. **Логи** — выходные данные, поставляемые агентом.

1.7. **Набор образцов вредоносного ПО** — список экземпляров вредоносного ПО, которые будут автоматически по очереди (после отката тестовой среды) запущены на тестовом сервере в рамках одного задания.

1.8. **Лаборатория машинного обучения** — команда исследователей, занимающаяся разработкой и обучением модели, осуществляющей классификацию программ на потенциально представляющие угрозу (suspicious) и не представляющие угрозы (clean).

2. Разработка инструментария динамического анализа вредоносного программного обеспечения (ToolChain):

2.1. **Глубокий динамический анализ** (deep dynamic analysis) — динамический анализ, использующий инвазивные методы исследования, такие как перехват, патч и т. д.

2.2. **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.

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

2.4. **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 с помощью патча адреса в IAT, так с помощью патча тела функции или подмены системной DLL на свою реализацию.

2.5. **Патч** (patch) — 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.

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

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

2.7.1. **Открытая часть карты памяти** — страницы виртуальной памяти процесса, доступ к которым не был искусственно запрещён путём объявления данного набора страниц не присутствующи-

ми в физической памяти.

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

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

2.9. **Традиционный антивирусный подход** — проверка хешей и сигнатур по базам данных сигнатур, которые пополняются экспертом-аналитиком. Не может противостоять неизвестным ранее угрозам.

2.10. **Next-gen подход** — подход к анализу вредоносного ПО, при котором применяется поведенческий анализ вредоносного ПО и машинное обучение.

2.11. **Статический анализ** — анализ исходного или объектного кода ПО, структуры исполняемого файла без запуска соответствующего процесса.

2.12. **Динамический анализ** — анализ ПО, происходящий во время его исполнения.

2.13. **Ловушки** (Honey Pots) — механизм, предназначенный для срабатывания векторов атаки вредоносного ПО, основанный на имитации интересующих вредоносное ПО объектов — пользовательских файлов, которые можно зашифровать, пользовательской информации, которую можно использовать — биткоин-кошельки, данные кредитных карт и т. д.

2.14. **События типа «А»** (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.

2.15. **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.

2.16. **Модуль маскировки аналитических инструментов** (Cloaking Module) — компонент, предназначенный для сокрытия от исследуемого образца факта запуска в тестовой среде, в которой присутствуют аналитические инструменты.

2.17. **Модуль контроля времени** (Time Machine Module) — компонент, управляющий течением времени для исследуемого образца, предназначенный для ускорения времени (например, для срабатывания временной бомбы) или замедления времени (например, чтобы скрыть от исследуемого образца падение производительности из-за анализа).

2.18. **Модуль контроля памяти** (Memory Control Module) — компонент, осуществляющий управление картой памяти и доступом к различным её областям, предназначенный для обеспечения полного контроля над исполнением приложения и сокрытия от исследуемого образца присутствия аналитических инструментов путём сокрытия модификаций страниц памяти.

2.19. **Модуль машинного обучения** (Machine Learning Module) — компонент, предназначенный для до-обучения модели машинного обучения на основе данных, получаемых на тестовом сервере Sandbox. Обученная модель работает на компьютере конечного пользователя и предсказывает вредоносность исполняющихся процессов на основе найденных корреляций между событиями типа «A» и «B».

5.2 Проектирование программно-аппаратного комплекса для запуска вредоносного ПО

5.2.1 Постановка задачи

Мы ставим перед собой задачу проектирования программно-аппаратного комплекса (Sandbox) для запуска вредоносного программного обеспечения в целях сбора данных о поведении вредоносной программы (таком как операции по работе с файлами, системным реестром, стек вызовов) для последующего анализа. Настоящее исследование сосредоточено на шифровальщиках-вымогателях, ориентированных на операционные системы Windows.

Sandbox должен удовлетворять следующим требованиям:

- (Автоматизация) Запуск образцов и сбор данных об их поведении должны быть автоматизированы;
- (Корректность запуска) Образцы должны запускаться корректно и не отказываться от заражения тестовой системы;
- (Безопасность) Управляющие серверы не должны подвергнуться воздействию вредоносного ПО;
- (Изоляция) Не должно быть интерференции запусков различных образцов, т. е. перед запуском следующего образца должен происходить откат тестовой среды, а тестовые серверы не должны заражать друг друга по сети;
- (Интеграция) Необходимо осуществить интеграцию с аналитическими инструментами для сбора данных и лабораториями для дальнейшего анализа собранных данных.

Прототип Sandbox призван продемонстрировать начальную итерацию реализации комплекса и принципиальную возможность удовлетворения проектного решения данным требованиям.

5.2.2 Требования к автоматизации

В автоматизированный сценарий на тестовом сервере должны входить цикл запуска образца вредоносного ПО и восстановления системы, прогон набора образцов под контролем агента, генерация отчёта. Должна быть возможность удалённого управления через интерфейс расписания запуска автоматизированных задач, где также может осуществляться настройка задач и сценариев: выбор набора образцов, агентов, пула тестовых машин и т. д.

Тестовые серверы должны находиться под контролем управляющих серверов, т. е. должна быть обеспечена возможность удалённого управления тестовым сервером, в том числе принудительная перезагрузка тестового сервера, гарантированное восстановление тестовой системы, надёжность канала коммуникации. Запускающееся вредоносное ПО не должно повлиять на контроль над тестовым сервером и на дальнейший цикл автоматизированной работы, даже если уничтожает тестовую операционную систему, повреждает MBR и т. д.

Для физических тестовых серверов откат тестовой среды должен происходить из чистой операционной системы, на которую не способно оказать воздействие запускающееся вредоносное ПО. Для виртуальных машин откат тестовой среды происходит посредством восстановления из снепшота, которое должно поддерживаться выбираемым гипервизором.

5.2.3 Требования к интеграции

Мы также ставим перед собой задачи интеграции с аналитическими инструментами, осуществляющими сбор данных, и интеграции с лабораториями для передачи собранных данных. Для решения этих задач необходимо обеспечить взаимодействие перед прогоном вредоносного ПО, во время прогона вредоносного ПО и после прогона вредоносного ПО:

1. Перед прогоном вредоносного ПО. Сюда входит скачивание/установка агента (в том числе получение последней версии агента через внутренние коммуникации), получение задания по запуску сценария от лаборатории

(через интерфейс расписания или очередь, например, с применением базы данных).

2. Во время прогона вредоносного ПО. Сюда входит SDK для создания агента как оболочки вокруг аналитического инструмента, в том числе для обеспечения возможности внедрения в сценарий необходимого взаимодействия с агентом (например, отправки различных команд).

3. После прогона вредоносного ПО. Сюда входит формирование отчёта и артефактов как оболочки вокруг логов (для чтения человеком и для дальнейшего анализа), а также возможность автоматизированной передачи логов через внутренние коммуникации в аналитическую лабораторию.

Также возможны другие виды интеграции, такие как совместное использование служб с аналитическими лабораториями (например, разделяемое хранилище образцов вредоносного ПО).

5.2.4 Выбор методов и ключевые архитектурные решения

Нами был произведён обзор существующих решений по автоматическому анализу вредоносного программного обеспечения. Общие подходы, предложенные нами, такие как использование Linux в качестве управляющей операционной системы при исследовании вредоносного ПО, ориентированного на Windows-системы, и цикл исследования образца вредоносного ПО с откатом тестовой среды, характерны для большинства работ, посвящённых аналогам Sandbox. Основное внимание в большинстве статей уделяется программному обеспечению для динамического анализа образцов, а не инфраструктурным решениям и вопросам развёртывания системы. Однако проектирование архитектуры закладывает фундамент для выбора аналитических инструментов.

Мы остановились на запуске вредоносного ПО на реальном аппаратном обеспечении. Вредоносное ПО может детектировать виртуальное окружение и прекращать свою работу, маскируя вредоносную активность в ситуациях потен-

циального исследования аналитическими инструментами. Предложенная нами архитектура оставляет возможность запускать вредоносное ПО, не чувствительное к виртуальному окружению, в виртуальных машинах на части тестовых серверов (для увеличения производительности и пропускной способности системы). При этом аналитические инструменты на уровне эмулятора или монитора виртуальных машин, призванные скрыть присутствие анализирующего ПО даже в случае наличия у вредоносного программного обеспечения компонент режима ядра (kernel-mode), нами применяться не будут. Мы ограничимся средствами анализа, исполняющимися на том же уровне привилегий, что и анализируемая программа, т. к. количество вредоносного ПО, использующего драйверные компоненты, довольно мало (благодаря механизму защиты Windows-драйверов электронной цифровой подписью), даже без учёта нашей нацеленности на анализ вымогателей-шифровальщиков.

В существующих решениях Интернет-соединение эмулируется или фильтруется, чтобы не допустить рассылку спам-сообщений или участие в DDOS-атаке. Мы же, в силу специфики работы программ-вымогателей, решили предоставить тестовым серверам полный доступ в Интернет. В дальнейшем, при расширении фокуса до других типов вредоносного программного обеспечения, будет применяться фильтрация Интернет-доступа.

В ходе работы была разработана архитектура системы, схема развёртывания (deployment) и управления системой. Основное внимание было уделено безопасности и производительности Sandbox, а также её интеграции с системами автоматизации. На данном этапе работы был запущен прототип Sandbox, состоящий из семи тестовых серверов и управляющего сервера, расположенных в ЦОД. Был подготовлен и запущен в Sandbox тестовый набор образцов вредоносного ПО, состоящий из шифровальщиков-вымогателей.

Тестовый сервер Sandbox имитирует пользовательский ПК. Вредоносное ПО запускается как на реальном аппаратном обеспечении, так и в виртуальных машинах. На части тестовых серверов Sandbox был установлен гипервизор для

увеличения пропускной способности системы (благодаря скорости отката среды при восстановлении виртуальной машины из снепшота). В качестве тестовых операционных систем используются Windows и Linux.

На текущий момент существуют автоматические сценарии работы для агентов 2-х типов: (1) осуществляющих сбор данных о происходящем в операционной системе, (2) обеспечивающих антивирусную защиту. В первом сценарии нас интересуют такие события, как файловая, реестровая активность, стек вызовов. Нами были собраны и переданы для анализа данные, полученные с помощью утилиты Proccess Monitor. Получаемые в Sandbox данные проходят предобработку и фильтрацию, а затем подаются на вход модели машинного обучения. Во втором сценарии Sandbox используется для автоматического тестирования ПО на способность детектировать вредоносное программное обеспечение и нейтрализовать его воздействие на операционную систему. Доступно управление запуском задач через графический WEB-интерфейс, а также графические отчёты о запусках.

Также Sandbox используется для сбора данных, полученных с помощью инструментов динамического анализа вредоносного ПО ToolChain, разработка которых в данный момент находится в состоянии прототипа.

5.2.5 Архитектура Sandbox

5.2.5.1 Общий обзор архитектуры

Аппаратное обеспечение Sandbox состоит из управляющего сервера, тестовых серверов, сетевого коммутатора и другого сетевого оборудования, обеспечивающего 1 GigE.

На управляющем сервере расположены 3 виртуальных сервера, представленные в Таблице 1: PXE-сервер (используется для загрузки тестовых серверов), MAN-сервер (используется для запуска управляющих скриптов на тестовом сервере), Jenkins-сервер (используется для запуска заданий из графического WEB-интерфейса).

Тестовые серверы, предназначенные для запуска вредоносного программ-

Таблица 1 — Управляющие серверы Sandbox

Сервер	Программное обеспечение	Задачи
PXE-сервер	PXE server NFS server DHCP server	Сетевая загрузка тестовых серверов Присвоение IP-адресов
MAN-сервер	SSH server Python environment Jenkins slave agents	Управление циклом работы тестовых серверов (включение и выключение питания через IP-KVM, запуск скриптов на управляющей и тестовой операционных системах)
Jenkins-сервер	Jenkins HTTP server SSH server	Запуск управляющих скриптов на MAN-сервере с помощью Jenkins WEB-интерфейса

ного обеспечения, сконфигурированы для сетевой загрузки через PXE-сервер. В качестве управляющей операционной системы используется Linux. Для запуска агентов и экземпляров шифровальщиков-вымогателей на начальном этапе использовались операционные системы семейства Windows. Затем в качестве тестовых были добавлены Linux-системы.

5.2.5.2 Сетевая архитектура

Одним из ключевых вопросов запуска вредоносного программного обеспечения на тестовых серверах была безопасность управляющих серверов и компьютеров корпоративной сети, с которых осуществляется удалённое управление Sandbox. Чтобы не допустить заражения через каналы коммуникации, мы должны были решить следующие задачи:

1. Задача защиты управляющих серверов от вредоносной активности на тестовых серверах;

2. Задача защиты тестовых серверов от вредоносной активности на других тестовых серверах.

Для решения поставленных задач была предложена следующая архитектура (см. Рисунок 19). Внутренняя сетевая конфигурация Sandbox состоит из двух логических локальных компьютерные сетей (VLAN) — доверенная сеть (GoodNet) и не доверенная сеть (DarkNet). Доверенная сеть предназначена для взаимодействия управляющих виртуальных серверов. Тестовые серверы получают IP-адреса из не доверенного диапазона от DHCP-сервера. Все запросы проходят через межсетевой экран (firewall) на управляющем сервере. Тестовые серверы имеют доступ в Интернет, а запросы к управляющим серверам блокируются. Также на этом уровне происходит изоляция тестовых серверов друг от друга.

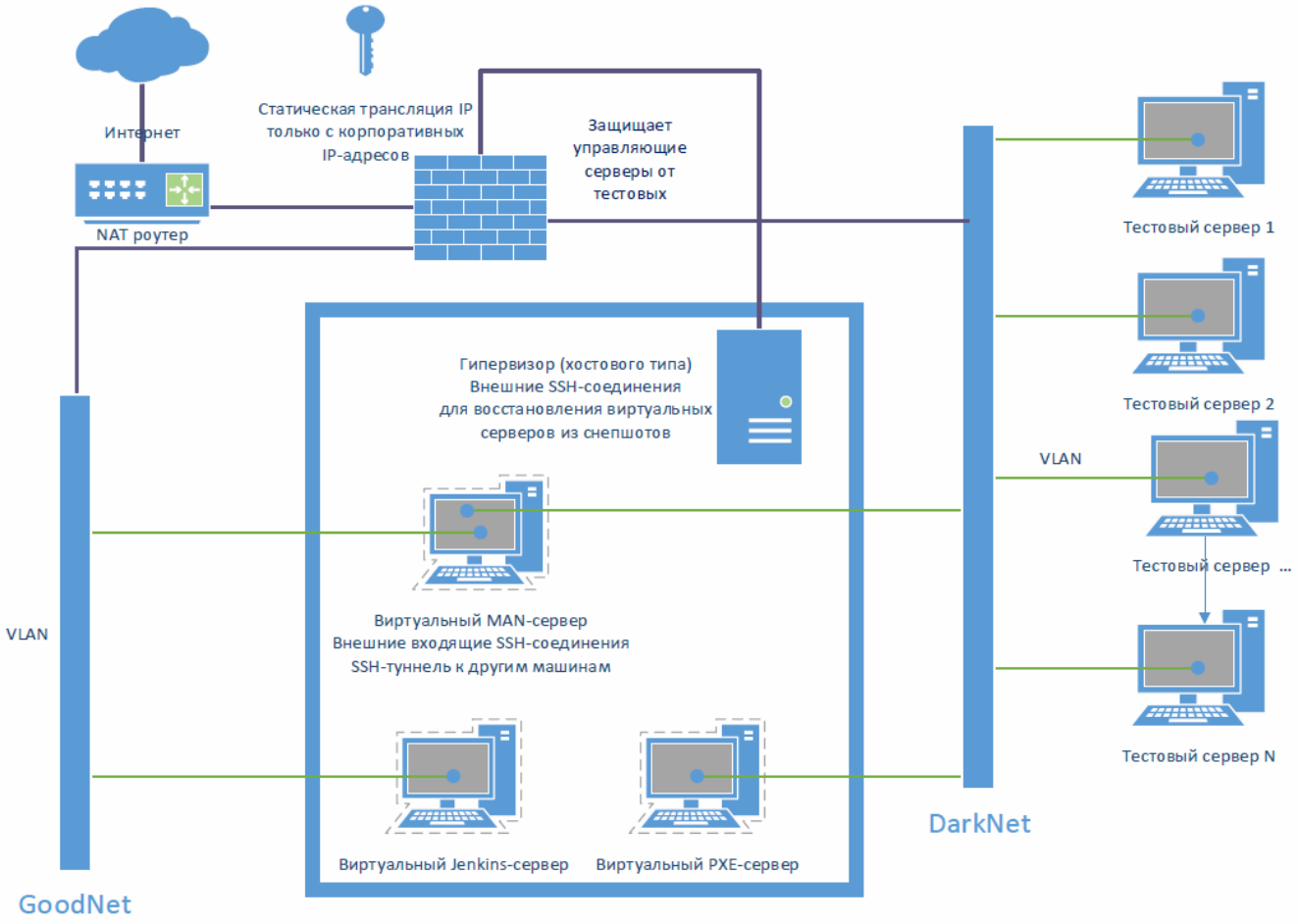


Рисунок 19 — Сетевая конфигурация Sandbox

Внешний доступ к управляющему серверу разрешён только с IP-адресов корпоративной сети. Возможен также доступ по RDP к тестовым серверам, происходящий через SSH-туннель на MAN-сервере. RDP является сложным протоколом, работа с которым представляет потенциальную опасность — например,

при подключении пользователь может случайно предоставить целевой системе доступ к локальным дискам машины, с которой производится подключение, тем самым подвергнув их риску шифрования. Поэтому такой способ подключения рассматривается нами как небезопасный и для его осуществления используются специальные машины во внешней изолированной подсети.

5.2.5.3 Развёртывание системы на физических серверах

Ещё одной важной задачей, стоявшей перед нами, была задача автоматизации цикла работы Sandbox и сбора данных. Цикл развёртывания Sandbox начинается с загрузки по сети (с использованием PXE-сервера) образа Linux в RAM (см. Рисунок 20). В качестве тестовых операционных систем используются образы Windows 7x64, Windows 8.1x64, Windows 10x64. Образы были специальным образом подготовлены для тихой (unattended) установки и содержат предустановленный SSH-сервер. Резервные копии (бэкапы) тестовых систем размещаются на отдельном диске или разделе, файловая система на котором не распознаётся тестовой операционной системой. Тем не менее, перед запуском процесса восстановления происходит проверка целостности бэкапа, и в случае повреждения он загружается с MAN-сервера.

Цикл развёртывания и тестирования на реальном аппаратном обеспечении происходит следующим образом:

1. Одноразовая подготовка образа тестовой ОС с предустановленным программным обеспечением (таким, как SSH-сервер) и его загрузка на MAN-сервер;

2. Одноразовая установка тестовой ОС и создание её бэкапа на отдельном диске либо разделе;

3. Для каждого экземпляра вредоносного ПО из тестового набора:

    - перезагрузка тестового сервера с использованием IP-KVM;

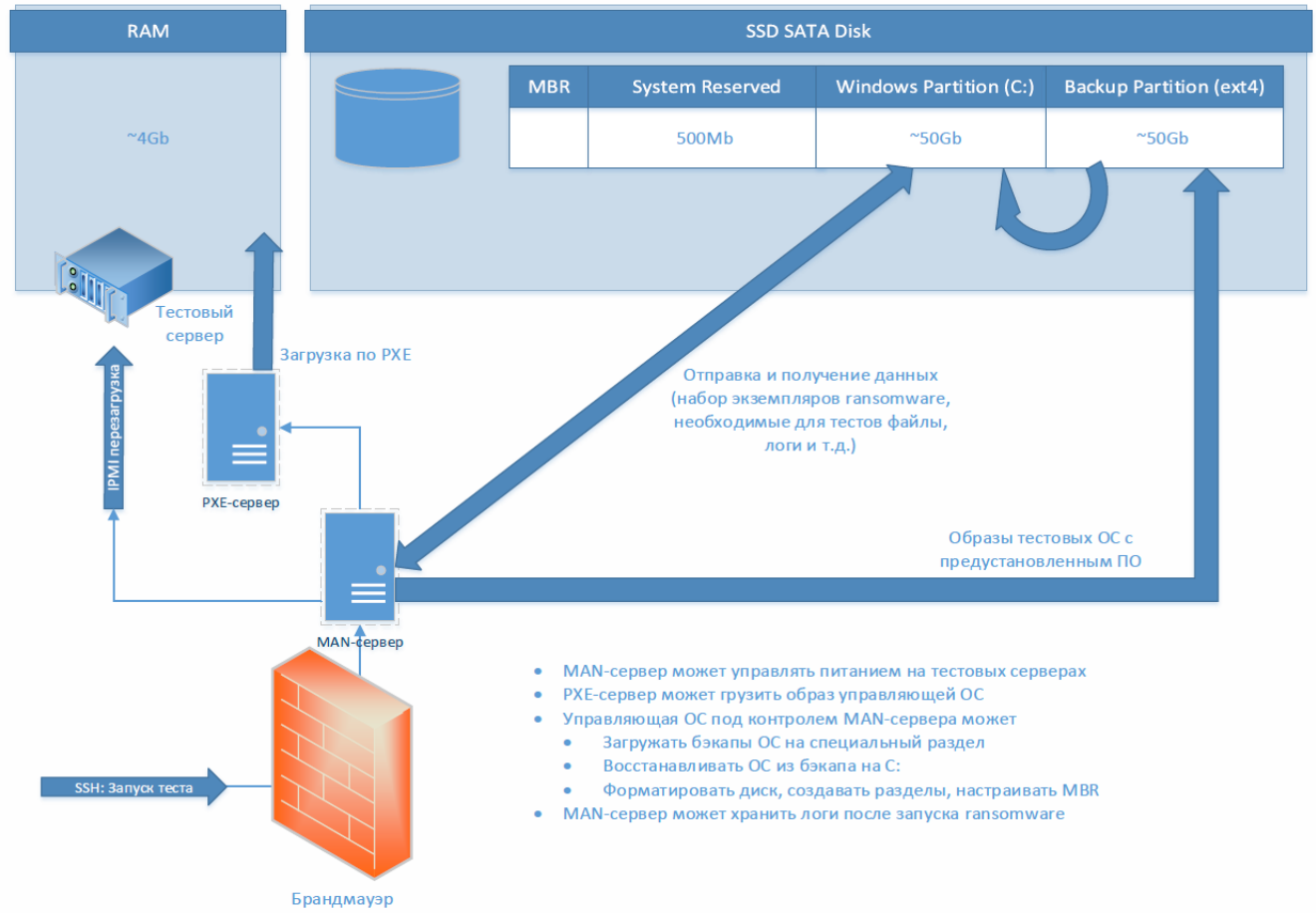


Рисунок 20 — Развёртывание тестовой системы на реальных тестовых серверах Sandbox

- загрузка управляющей операционной системы;
- восстановление тестовой операционной системы из бэкапа;
- перезагрузка тестового сервера с использованием IP-KVM;
- загрузка тестовой операционной системы;
- установка агента;
- запуск вредоносного ПО, ожидание сигнала от агента либо тайм-аута;
- сбор логов и другой выходной информации.

Перезагрузка и восстановление тестовой системы из бэкапа на каждой итерации цикла тестирования гарантирует нам «чистоту» системы после запуска вредоносного программного обеспечения. Таким образом происходит откат тестовой среды к начальному состоянию. Переключение между загрузкой тестовой опера-

ционной системы с локального диска (localboot) и загрузкой управляющей операционной системы (netboot) обеспечивается конфигурацией PXE-сервера.

Каждый этап развёртывания Sandbox может быть запущен с помощью соответствующей задачи в WEB-интерфейсе Jenkins. После завершения задачи фреймворком Allure генерируется WEB-отчёт, а логи и другая выходная информация попадает в специальное хранилище (Artifactory).

5.2.6 Агенты

В данный момент в Sandbox запускаются различные агенты. Агенты делятся на 2 типа:

1. Занимающиеся только мониторингом и сбором доступной им информации о том, что происходит в системе;
2. Антивирусного типа, занимающиеся также защитой системы и ориентированные на детектирование вредоносного программного обеспечения.

5.2.6.1 Process Monitor

В качестве примера мы здесь рассмотрим агент Process Monitor. Данная утилита, разработанная Sysinternals, осуществляет мониторинг файловой системы, системного реестра, сетевой активности и процессов. Эта информация записывается в лог в формате .pml. На Рисунке 21 можно видеть образец лога, полученный с системы, заражённой шифровальщиком-вымогателем WannaCry (SHA-256 хеш исследуемого образца b9318a66fa7f50f2f3ecaca02a96268ad2c63db7554ea3acbde43bf517328d06). Приведённый фрагмент лога содержит файловую активность порождённого WannaCry процесса tasksche.exe, в котором происходит шифрование файлов. При этом на выходе получается большое количество данных (размер лога за 5 минут работы может достигать 1 Gb). Полученные логи проходят несколько фильтраций и переводятся в формат .xml, удобный для синтаксического анализа и обработки данных, и в сжатом виде передаются на управляющий сервер.

Process Monitor - C:\Users\anastasia.pereberina\Downloads\procmon-log-b9318a66fa7f50f2f3ecaca02a96268ad2c63db7554ea3acbde43bf517328d06\procmon-log-filtered.pml

File Edit Event Filter Tools Options Help

Tim...	Process Name	PID	Operation	Path	Result	Detail
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures\USER-40.PNG.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Delete, Disposition:
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures\user-48.png.WNCRY	REPARSE	Desired Access: Read Attributes, Disposition: Open, (
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures\USER-48.PNG.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Disposition: Open, (
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures	REPARSE	Desired Access: Write Data/Add File, Synchronize, D
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures	SUCCESS	Desired Access: Write Data/Add File, Synchronize, D
23:58:...	tasksche.exe	3068	CloseFile	C:\ProgramData\Microsoft\User Account Pictures	SUCCESS	
23:58:...	tasksche.exe	3068	CloseFile	C:\ProgramData\Microsoft\User Account Pictures\user-48.png.WNCRY	SUCCESS	
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures\user-48.png.WNCRY	REPARSE	Desired Access: Write Attributes, Synchronize, Dispos
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\Microsoft\User Account Pictures\user-48.png.WNCRY	SUCCESS	Desired Access: Write Attributes, Synchronize, Dispos
23:58:...	tasksche.exe	3068	SetBasicInformationFile	C:\ProgramData\Microsoft\User Account Pictures\user-48.png.WNCRY	SUCCESS	CreationTime: 01.01.1601 3:00:00, LastAccessTime: 0
23:58:...	tasksche.exe	3068	CloseFile	C:\ProgramData\Microsoft\User Account Pictures\user-48.png.WNCRY	SUCCESS	
23:58:...	tasksche.exe	3068	CreateFile	C:\hiberfil.sys.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Disposition: Open, (
23:58:...	tasksche.exe	3068	CreateFile	C:\hiberfil.sys	SHARING VIO...	Desired Access: Generic Read, Disposition: Open, O
23:58:...	tasksche.exe	3068	CreateFile	C:\hiberfil.sys.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Delete, Disposition:
23:58:...	tasksche.exe	3068	CreateFile	C:\pagefile.sys.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Disposition: Open, (
23:58:...	tasksche.exe	3068	CreateFile	C:\pagefile.sys.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Delete, Disposition:
23:58:...	tasksche.exe	3068	CreateFile	C:\swapfile.sys.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Disposition: Open, (
23:58:...	tasksche.exe	3068	CreateFile	C:\swapfile.sys	SHARING VIO...	Desired Access: Generic Read, Disposition: Open, O
23:58:...	tasksche.exe	3068	CreateFile	C:\swapfile.sys.WNCRY	NAME NOT F...	Desired Access: Read Attributes, Delete, Disposition:
23:58:...	tasksche.exe	3068	CreateFile	C:\Windows\Temp\0.WNCRYT	NAME NOT F...	Desired Access: Read Attributes, Delete, Disposition:
23:58:...	tasksche.exe	3068	CreateFile	C:\Users	SUCCESS	Desired Access: Read Data/List Directory, Synchroni
23:58:...	tasksche.exe	3068	QueryDirectory	C:\Users*	SUCCESS	Filter: *, 1: .
23:58:...	tasksche.exe	3068	QueryDirectory	C:\Users	SUCCESS	0: ., 1: acrtest, 2: Administrator, 3: All Users, 4: cyg_ser
23:58:...	tasksche.exe	3068	CreateFile	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	Desired Access: Generic Read, Disposition: Open, O
23:58:...	tasksche.exe	3068	QueryAttributeTagFile	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	Attributes: A, ReparseTag: 0x0
23:58:...	tasksche.exe	3068	QueryStandardInformation...	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	AllocationSize: 245 760, EndOfFile: 245 760, NumberO
23:58:...	tasksche.exe	3068	QueryBasicInformationFile	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	CreationTime: 20.11.2017 23:58:27, LastAccessTime: 2
23:58:...	tasksche.exe	3068	QueryStreamInformationFile	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	0: ::$DATA
23:58:...	tasksche.exe	3068	QueryBasicInformationFile	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	CreationTime: 20.11.2017 23:58:27, LastAccessTime: 2
23:58:...	tasksche.exe	3068	QueryEaInformationFile	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	EaSize: 0
23:58:...	tasksche.exe	3068	CreateFile	C:\Users\acrtest\Desktop\@WanaDecryptor@.exe	SUCCESS	Desired Access: Generic Read/Write, Delete, Write D
23:58:...	tasksche.exe	3068	QueryAttributeInformationV...	C:\Users\acrtest\Desktop\@WanaDecryptor@.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitiv
23:58:...	tasksche.exe	3068	QueryBasicInformationFile	C:\Users\acrtest\Desktop\@WanaDecryptor@.exe	SUCCESS	CreationTime: 20.11.2017 23:58:42, LastAccessTime: 2
23:58:...	tasksche.exe	3068	QueryAttributeInformationV...	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	SUCCESS	FileSystemAttributes: Case Preserved, Case Sensitiv
23:58:	tasksche.exe	3068	QueryRemoteProtocolInfo	C:\ProgramData\fnfxqagjga445\@WanaDecryptor@.exe	INVALID PARA	

Showing 61 992 of 1 073 950 events (5.%) Backed by C:\Users\anastasia.pereberina\Downloads\procmon-log-b9318a66fa7f50f2f3ecaca02a96268ad2c63db7554ea3acbde43bf517328d06\procmon-log-filtered.pml

Рисунок 21 — Логи Process Monitor с системы, заражённой вредоносным ПО WannaCry

Источник: скриншот лога, полученного на тестовом сервере

5.2.7 Набор образцов вредоносного ПО

На начальном этапе в Sandbox запускалось программное обеспечение, симулирующее работу программы-шифровальщика, выполняющее исключающее ИЛИ (XOR) с некоторым заданным ключом для файлов в указанной директории.

На следующем этапе работы для запуска в Sandbox был подготовлен набор образцов шифровальщиков-вымогателей, для формирования которого была использована база вредоносных программ службы VirusTotal [41] и данные компании F-Secure о шифровальщиках [42], появившихся в 2010 — 2016 годах (на Рисунке 22 представлен фрагмент с 2013 года, полную версию можно найти в источнике). Также были рассмотрены некоторые образцы, появившиеся в 2017 году (такие как WannaCry [43], ExPetr [44] и др.). Тестовый набор составил 317 образцов. Данный набор был запущен с агентом 2-го типа, при этом происходило тройное тестирование, включающее в себя:

1. Тестирование агента 2-го типа на способность детектировать активность

шифровальщика и восстановить зашифрованные файлы. По результатам было выявлено несколько техник, позволяющих скрыть вредоносную активность от тестируемого агента.

2. Верификацию тестового набора. По результатам тестирования было выявлено, что около трети найденных нами образцов не являются действующими шифровальщиками (например, серверы, к которым они обращаются, более не доступны).

3. Отладку процесса запуска вредоносного программного обеспечения. По результатам тестирования были выявлены проблемы окружения и защиты логов, о которых подробнее рассказывается в следующих разделах.

После этого образцы, прошедшие верификацию, были запущены с агентом 1-го типа для сбора данных об их вредоносной активности.

В дальнейшем планируется создание новых наборов crypto-ransomware и пополнение их новыми образцами.

В рамках интеграции с лабораторией машинного обучения мы получаем набор хешей для запуска по запросу тестового сценария из предоставленной базы данных. Образцы для запуска скачиваются с помощью VirusTotal или из кэширующего хранилища.

5.2.8 Проблема окружения

Образы Windows, предназначенные для установки на тестовых серверах, содержат SSH-сервер, который используется для подключения к тестовым серверам и запуска на них управляющих скриптов, агентов и вредоносного программного обеспечения. Однако такой способ запуска программ отличается от стандартного запуска пользователем ПК и приводил к проблеме окружения. Под окружением здесь понимается совокупность следующих факторов: переменные окружения (environment variables), сессия пользователя (terminal session), профиль пользователя (user profile), конфигурация в системном реестре (HKEY_CURRENT_USER),

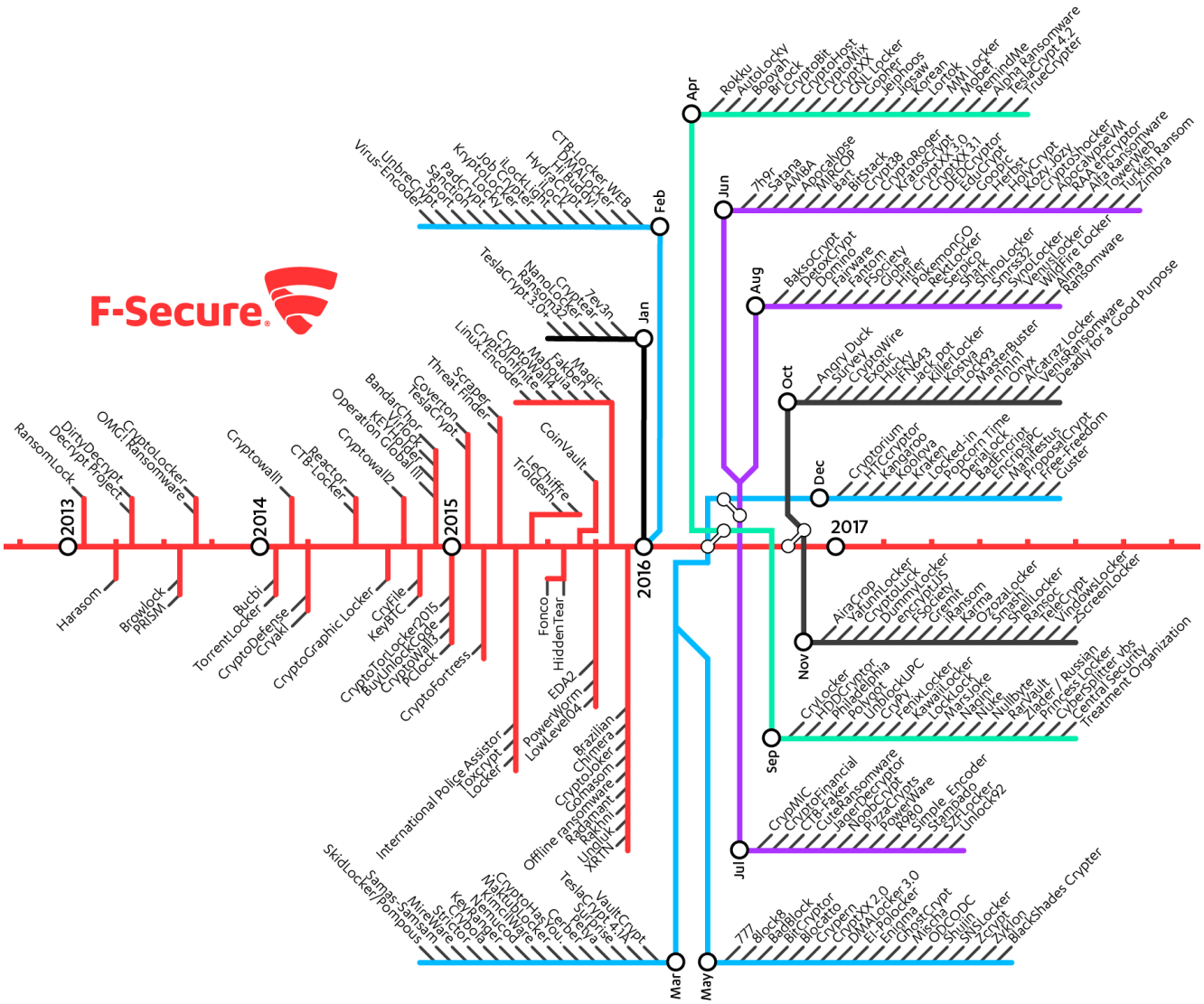
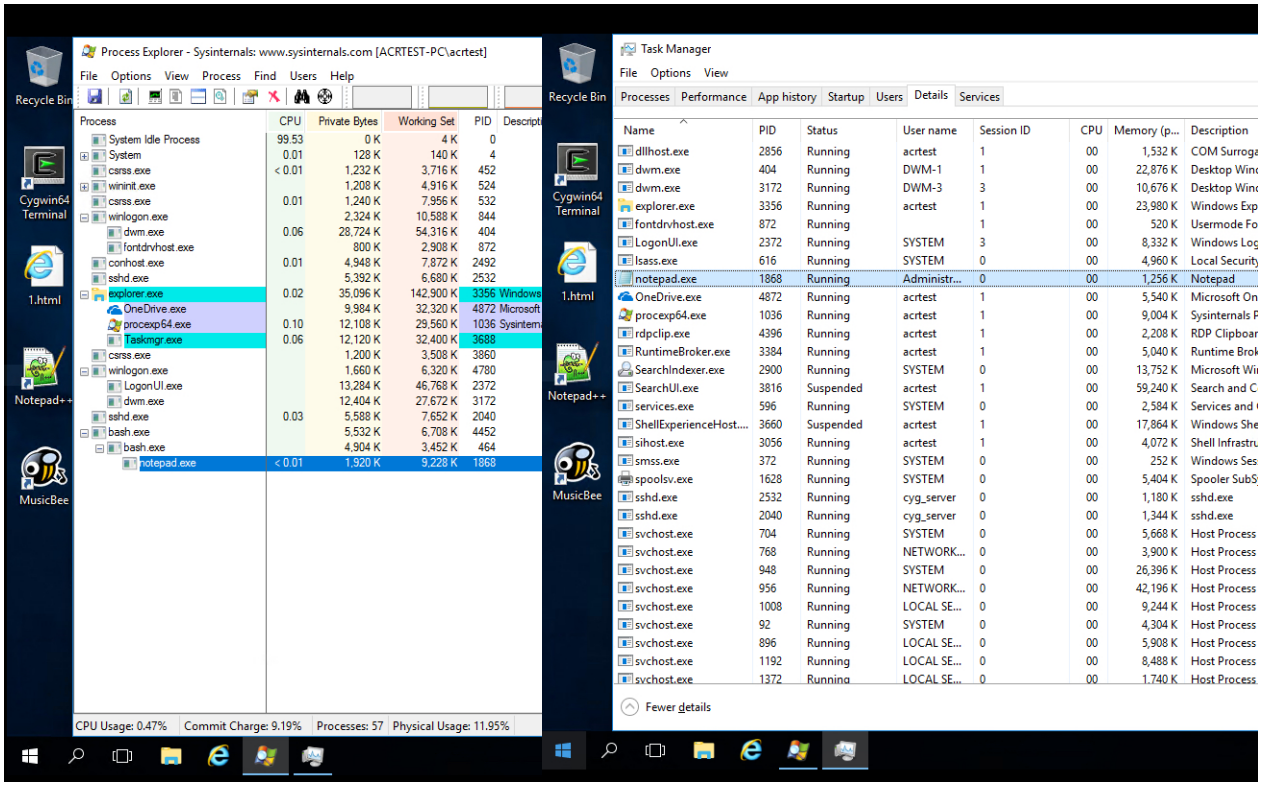


Рисунок 22 — Шифровальщики-вымогатели, появившиеся в 2013-2016 годах
Источник: отчёт F-Secure [42]

идентификатор пользовательского сеанса (logon session, LUID), идентификатор безопасности (SID). Некоторые экземпляры crypto-ransomware не могли осуществить необходимую им активность в системе в нестандартном окружении либо отказывались запускаться, поэтому потребовалось вспомогательное программное обеспечение для запуска шифровальщиков-вымогателей (Environment Helper).

Проиллюстрируем проблему окружения на примере запуска стандартной программы Блокнот (notepad.exe). На Рисунке 23 представлены снимки экрана (скриншоты) при подключении по RDP к тестовому серверу, на котором по SSH был запущен notepad.exe. На данной группе скриншотов видно, что идентификатор сессии (Session Id), в которой запущен notepad.exe, равен 0 — это неин-

терактивная сессия, в которой нет графической оболочки пользователя (Windows shell), поэтому графическое окно не отображается. Также, при помощи ещё одной утилиты SysInternals — Process Explorer, можно видеть, что родительский процесс (parent process) для notepad.exe — bash.exe (консоль SSH-сервера), а не explorer.exe, как у Task Manager и Process Explorer, запущенных при подключении по RDP.

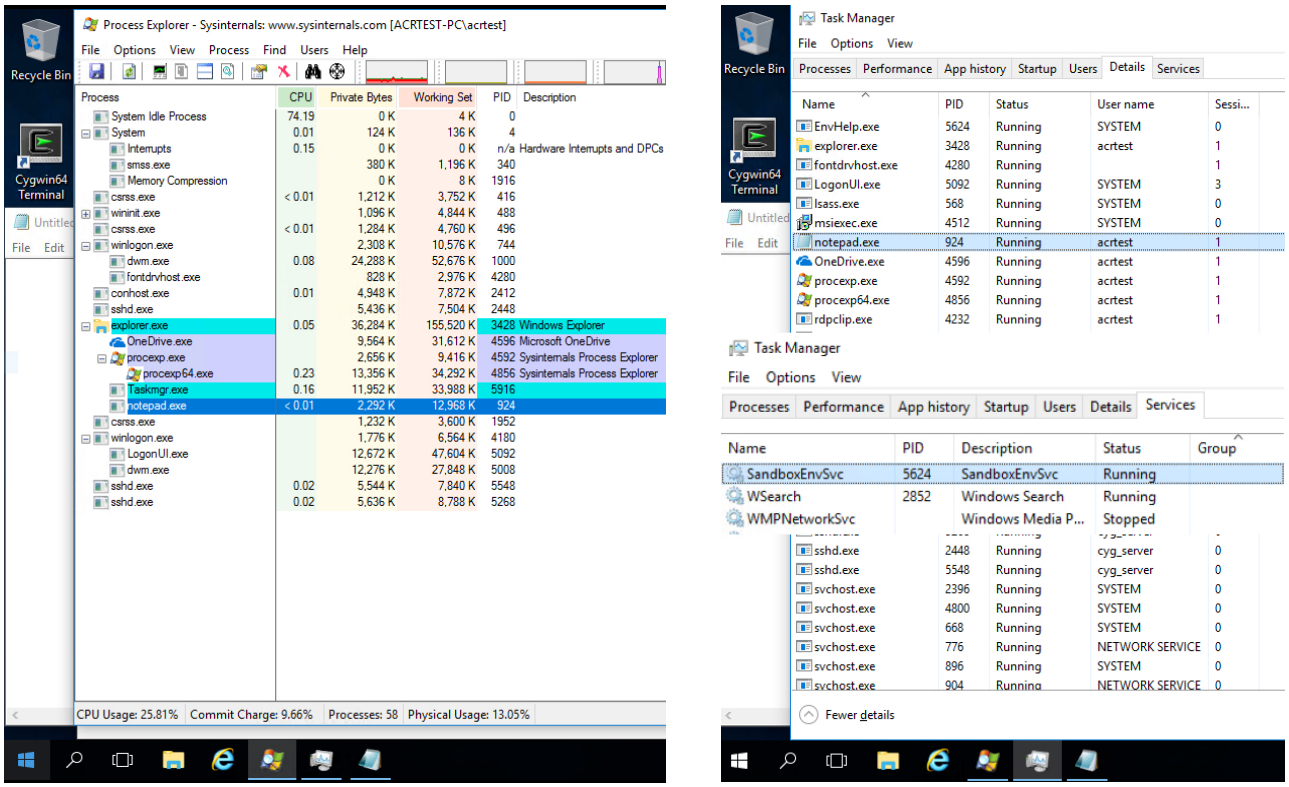


а) Process Explorer б) Task Manager

Рисунок 23 — Запуск notepad.exe на тестовом сервере через SSH
Источник: скриншот с тестового сервера

Нашей задачей было запустить вредоносное программное обеспечение таким образом, чтобы эмулировать стандартный сценарий запуска пользователем интерактивной сессии. На второй группе скриншотов (см. Рисунок 24) notepad.exe запущен с корректно настроенным окружением, которое обеспечивается вспомогательным ПО. В данном случае, идентификатор сессии notepad.exe равен 1, а его родительский процесс — explorer.exe. Такое окружение не отличается от стандартного и не должно помешать корректной работе crypto-ransomware.

Архитектурно Environment Helper состоит из сервиса, который запускает



а) Process Explorer б) Task Manager

Рисунок 24 — Корректный запуск notepad.exe на тестовом сервере с помощью Environment Helper

Источник: скриншот с тестового сервера

программу, имя которой сообщается ему клиентом, в нужной сессии (получив необходимые привилегии, которыми не обладает SSH-сервер). Сервис в Windows не может осуществить интерактивный вход в систему. (До Windows 8 существовал обходной путь через использование GINA [45], однако, начиная с Windows 8, где больше нет данного компонента, эта задача становится недокументированной и сложной.) Поэтому в нашем решении сервис открывает уже существующую сессию пользователя, а система настроена так, что происходит автоматический интерактивный вход созданного нами пользователя, при котором загружаются стандартные параметры окружения. При необходимости, в дальнейшем пользователь получит MS аккаунт, электронную почту и так далее. Клиент передаёт серверу запрос на запуск вспомогательной утилиты, которая использует COM-интерфейс для взаимодействия с графической оболочкой, чтобы запустить шифровальщик-вымогатель. При этом происходит эмуляция интерактивного запуска ransomware

самим пользователем.

5.2.9 Проблема защиты логов

Другой важной проблемой стала проблема хранения данных в системе, в которой запущено вредоносное ПО, шифрующее данные (см. Рисунок 25). Опасности подвергались как выходная информация, которую мы хотим получить с системы (логи), так и необходимые для теста программы и утилиты. Чтобы защитить необходимые для работы агентов файлы, мы написали драйвер мини-фильтр файловой системы [46], который контролирует доступ к рабочим директориям. Для этого он обрабатывает события открытия файловых дескрипторов, регистрируя *Preoperation* и *Postoperation Callback Routines*. Доступ разрешён Sandbox программам и некоторому белому списку доверенных процессов (см. Рисунок 26).

Name	Date modified	Type	Size
@Please_Read_Me@.txt	12/2/2017 9:07 PM	Text Document	1 KB
@WanaDecryptor@.exe	5/12/2017 3:22 AM	Application	240 KB
config.pmc	11/27/2017 10:30 PM	PMC File	3 KB
driver.log	12/2/2017 9:09 PM	Text Document	35 KB
honeypot.pmc	12/2/2017 9:09 PM	PMC File	4 KB
image.pmc	12/2/2017 9:09 PM	PMC File	4 KB
procmon-log.pml.WNCRY	12/2/2017 9:09 PM	WNCRY File	387,184 KB
procmon-log-honeypot.xml	12/2/2017 9:09 PM	XML Document	3,086 KB
procmon-log-honeypot.zip.WNCRY	12/2/2017 9:09 PM	WNCRY File	211 KB
procmon-log-image.xml	12/2/2017 9:09 PM	XML Document	3,080 KB
procmon-log-image.zip.WNCRY	12/2/2017 9:09 PM	WNCRY File	211 KB
SandboxEnvSvc.log	12/2/2017 9:09 PM	Text Document	1 KB

Рисунок 25 — Зашифрованные логи утилиты Process Monitor
Источник: скриншот с тестового сервера

```
[SelfDefence] Protected folder (\Device\HarddiskVolume2\SandboxProgram\wship6.dll) access by \Device\HarddiskVolume2\SandboxProgram\Procmon64.exe: Success (Sandbox process)
[SelfDefence] Protected folder (\Device\HarddiskVolume2\SandboxData\) access by \Device\HarddiskVolume2\ProgramData\fnfxqagjga445\tasksche.exe: Access denied
[SelfDefence] Protected folder(\Device\HarddiskVolume2\SandboxProgram\) access by \Device\HarddiskVolume2\ProgramData\fnfxqagjga445\tasksche.exe: Access denied
[SelfDefence] Protected folder (\Device\HarddiskVolume2\SandboxData\procmon-log-1.pml) access by \Device\HarddiskVolume2\SandboxProgram\Procmon64.exe: Success
```

Рисунок 26 — Драйвер SelfDefence защищает директории Sandbox от WannaCry

5.2.10 Производительность аппаратного обеспечения

В данный момент Sandbox состоит из 7 тестовых серверов и 1 управляющего сервера, расположенных в ЦОД. Возможности текущей версии прототипа Sandbox без учёта использования виртуальных машин описаны в Таблице 2.

Таблица 2 — Производительность аппаратного обеспечения Sandbox

Тип агента	Развёртывание системы	Время работы	Образцов за 24 часа
1	10 минут	2-6 минут	630 — 840
2		1-12 минут	458 — 916

5.2.11 Машинное обучение

Процесс машинного обучения не является фокусом данной работы, поэтому подробно описываться не будет. Получение входных данных является важным этапом машинного обучения: необходимо не только собрать большой объём данных, но и должным образом подготовить их — произвести предварительную обработку.

Происходит обучение модели Random Forest, которая представляет собой классификатор программ на подозрительные (suspicious) и не вызывающие подозрения (clean). Концептуально, роль Sandbox в этом процессе заключается в поставке сырых данных, возможно, с переходом на следующий этап, где происходит их предобработка (см. Рисунок 27).

Сейчас для обучения модели используются отфильтрованные логи Procmon. При этом в Sandbox собираются данные только о вредоносных образцах, данные о поведении не вредоносного ПО поступают из других источников. В рамках подготовки данных мы проводим фильтрацию событий по классам и процессам. Процесс подготовки данных происходит итеративно: получая обратную связь о работе обученной модели, мы уменьшаем объём поставляемых данных, оставляя только существенную для классификации информацию.

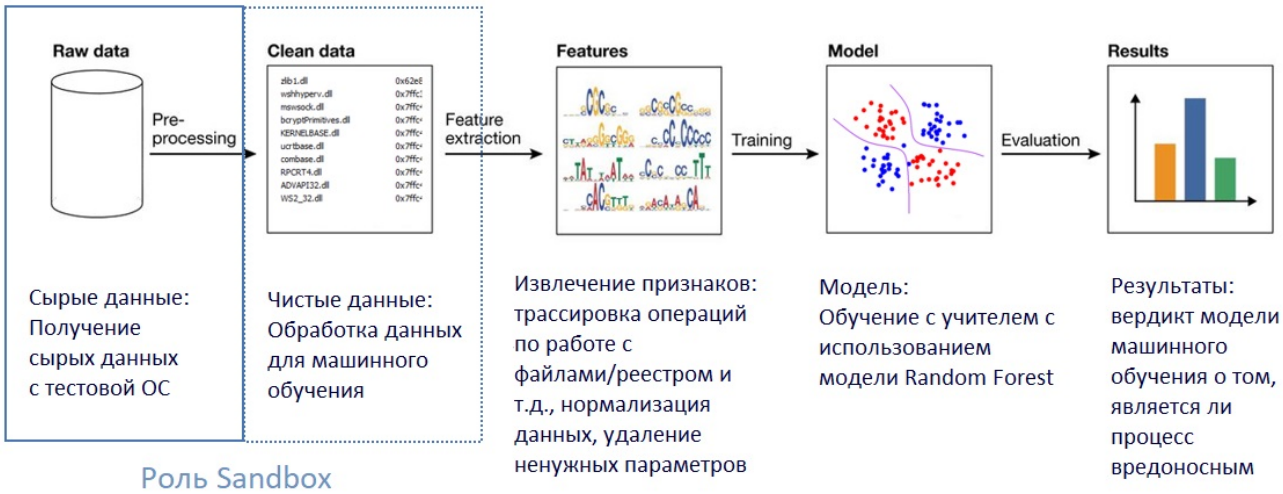


Рисунок 27 — Машинное обучение и Sandbox
Источник: лаборатория машинного обучения

5.2.12 Виртуальные машины

После запуска первой версии прототипа Sandbox полностью на реальном аппаратном обеспечении, мы осуществили «VM split», в рамках которого на части тестовых серверов Sandbox были размещены виртуальные машины под гипервизором Virtuozzo 6. Мы провели исследование различных гипервизорных решений, при этом также рассматривались VMware vSphere 6.7 и Virtuozzo 7. Мы остановились на решении от компании Virtuozzo, гипервизор Virtuozzo 7 не подошёл для текущего комплекса потому, что CPU на тестовых серверах не поддерживает необходимую опцию unrestricted guest. Тестовый сервер выдерживает 3 виртуальных машины конфигурации 4 CPU cores, 4Gb RAM, 64Gb disk. Виртуальные машины доступны для управления с помощью PXE- и MAN-сервера и могут участвовать в цикле развёртывания и тестирования, описанном ранее и предназначенном для физических тестовых серверов. При этом добавляется дополнительный цикл работы, в котором происходит управление виртуальными машинами через гипервизор на хосте:

1. Одноразовая установка тестовой ОС с предустановленным программным обеспечением и создание её снепшота;

2. Для каждого экземпляра вредоносного ПО из тестового набора:

- восстановление виртуальной машины из снепшота;
- установка агента;
- запуск вредоносного ПО, ожидание сигнала от агента либо тайм-аута;
- сбор логов и другой выходной информации.

5.2.12.1 Производительность

При использовании виртуальных машин время на откат тестовой среды уменьшается с 10 минут до приблизительно 1.5 секунд. На 3 тестовых серверах был размещен гипервизор. Производительность текущей версии прототипа Sandbox с учётом добавления виртуальных машин представлена в Таблице 3.

Таблица 3 — Производительность Sandbox

Тип тестового сервера	Тип агента	Развёртывание системы	Время работы	Образцов за 24 часа
Физические серверы	1	10 минут	2-6 минут	360 — 480
	2		1-12 минут	261 — 523
Виртуальные машины	1	1.5 секунды	2-6 минут	2160 — 6480
	2		1-12 минут	1080 — 12960
Итого	1			2520 — 6960
	2			341 — 13483

Таким образом, добавление виртуальных машин существенным образом (до 15 раз) увеличило пропускную способность комплекса.

5.2.13 Linux в качестве тестовой системы

После запуска первой версии прототипа Sandbox для вредоносного ПО, ориентированного на Windows-системы, мы рассмотрели также Linux в качестве тестовой операционной системы. В данный момент Linux представлен образом CentOS7x64. Для него предоставлены как физические серверы, так и виртуальные машины.

Т. к. ранее для хранения бэкапов мы использовали диск, форматированный в файловой системе ext4, нужно было улучшить систему хранения. Этот вари-

ант был хорош, когда тестовые системы были представлены только семейством ОС Windows, которые не могли распознать диск с бэкапами. Вначале мы думали об использовании малораспространённой файловой системы, но в Linux встроена поддержка большого количества файловых систем. В перспективе мы также планируем добавить Mac OS для тестирования, поэтому HFS/APFS также не подходили.

В настоящий момент для защиты бэкапов мы шифруем диск с бэкапами как блочное устройство, используя утилиту cryptsetup. Это не защитит данные от шифрования блочного устройства, но большинство шифровальщиков для Linux достаточно примитивны и не обращаются к девайсам напрямую. В нашем случае, чтобы даже примонтировать устройство, нужно знать мастер-ключ.

В данный момент для тестовых систем Linux доступен только агент 2-го типа, занимающийся защитой системы и ориентированный на детектирование вредоносного программного обеспечения, который находится в стадии разработки. Тестовые серверы применяются для тестирования данного программного обеспечения.

Набор вымогателей-шифровальщиков для Linux-систем, собранный нами для запуска в Sandbox, невелик и состоит из 7 экземпляров (таких как Linux.Encoder, Erebus, KillDisk). Часть этих программ плохо написана и критически ломает систему, при этом лишая себя возможности попросить выкуп у пользователя.

В перспективе, при необходимости, нами будут проделаны шаги, аналогичные потребовавшимся для запуска ransomware на Windows-системах — защита логов или настройка окружения, близкого к пользовательскому. Сейчас запуск из командной строки и запуск через SSH представляется нам довольно естественным для пользователей Linux-систем и не создающим препятствий для корректного старта вредоносного программного обеспечения.

5.3 Разработка инструментария для динамического анализа вредоносного программного обеспечения

5.3.1 Постановка задачи

Мы ставим перед собой задачу разработки инструментов для глубокого динамического анализа вредоносного программного обеспечения (ToolChain), которые будут работать на тестовом сервере программно-аппаратного комплекса Sandbox. Таким образом, ToolChain представляет собой наш собственный Sandbox-агент 1-го типа. Разрабатываемые нами инструменты предназначены для анализа вредоносного ПО, ориентированного на операционные системы семейства Windows. Главным образом наше исследование сосредоточено на шифровальщиках-вымогателях (crypto-ransomware), однако инструментарий ToolChain может быть применён для различных типов вредоносного ПО.

До этого мы собирали события, доступные для внешнего (неинвазивного) мониторинга, видимые такими инструментами, как, например, Process Monitor. Модуль, осуществляющий глубокий динамический анализ, следит за внутренними событиями исследуемого образца, при этом используя инвазивные методы исследования, такие как перехват, патч. Целью анализа является детектирование вредоносной активности, определение подозрительных паттернов поведения, а также предоставление данных для модуля машинного обучения, который сможет предсказывать наличие внутренних событий по данным мониторинга внешней активности программы.

ToolChain должен удовлетворять следующим требованиям:

- (Контроль) Должен быть обеспечен полный контроль над запуском приложения;
- (Мониторинг) Должен осуществляться мониторинг низкоуровневых событий процесса (вызовов системного API);
- (Корректность запуска) Вредоносное ПО должно работать корректно под

контролем аналитических инструментов;

- (Маскировка) Вредоносное ПО не должно обнаружить присутствие аналитических инструментов.

Прототип ToolChain призван продемонстрировать начальную итерацию реализации инструментов и принципиальную возможность удовлетворения проектного решения данным требованиям.

5.3.2 Выбор методов и ключевые архитектурные решения

Мы провели исследование существующих подходов к динамическому анализу вредоносного программного обеспечения. Подробнее об этом можно прочитать в соответствующем разделе. В Таблице 4 представлены базовые техники, выбранные нами для реализации в рамках ToolChain:

Таблица 4 — Основные техники динамического анализа

Analysis support for	**ToolChain**
API calls	Да
System calls	Планируется
Function parameters	Да/Планируется
File system operation	Process Monitor
Process/thread creation	Process Monitor
Registry operations	Process Monitor
COM components	Нет
Detecting dyn. generated code	Планируется
Restoring packed binaries	Планируется
W⊕X page protection	Да, модифицированный
Multiple layers of packers	Планируется

Часть этих техник включает мониторинг внешних событий, не требующих инвазивных методов анализа (файловые, реестровые операции, создание процессов). Для сбора таких событий мы можем продолжить применять сторонние утилиты, такие как Process Monitor. Больший интерес представляют внутренние события, требующие глубокого динамического анализа.

Традиционные подходы в области динамического анализа используют перехват вызовов функций, анализ параметров. Есть техника W⊕X, которая использу-

ется 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 был разработан специальный механизм. Будут применяться также различные подходы маскировки аналитических инструментов, как встроенные в ToolChain, так и в качестве отдельных компонентов.

Разрабатываемые нами аналитические инструменты будут работать на уровне пользователя или ядра операционной системы. ToolChain находится на стадии прототипа, разработан для x64 Windows архитектуры, в данный момент содержит компоненты уровня пользователя.

В разделе 5.3.3 описаны архитектурные решения проектирования ToolChain, в последующих разделах раскрываются детали реализации. Мы также останавливаемся на проблемах, с которыми мы столкнулись в процессе прототипирования, и предлагаем различные способы их решения.

5.3.3 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 Рисунке 28. Аналитические инструменты работают на уровне пользователя или ядра операционной системы. В данной работе 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.

Мониторинг вызываемого системного API будет осуществляться при помощи перехвата функций системного API на основе техники инжектирования в адресное пространство приложения динамически загружаемой библиотеки. Инжектируемая DLL реализует следующий функционал:

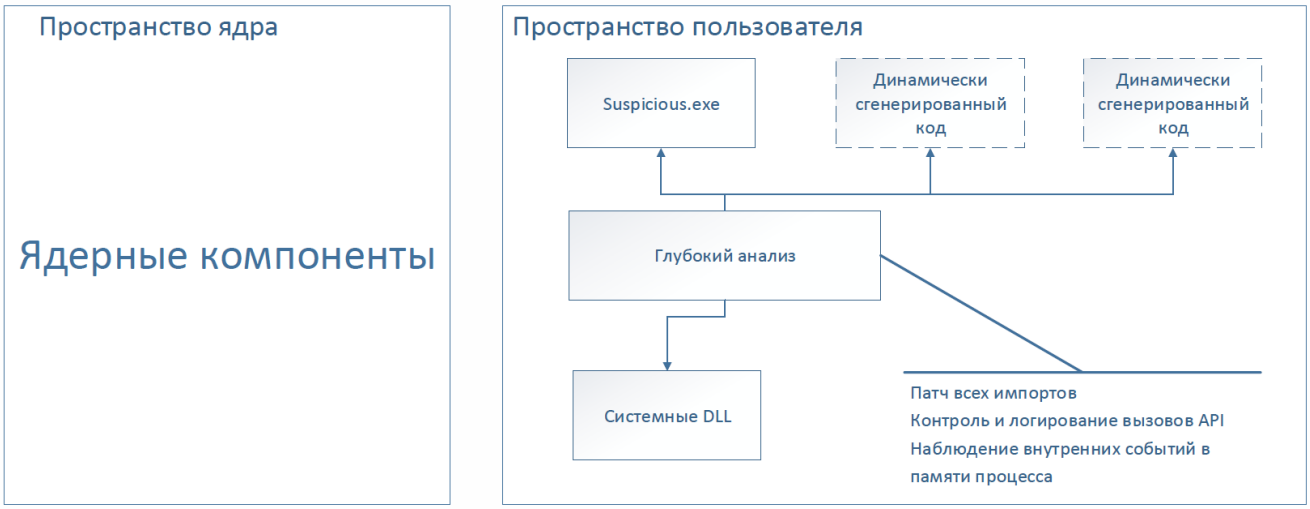


Рисунок 28 — Инструменты глубокого динамического анализа

1. Осуществляет парсинг таблицы импортов PE-файла и составляет карту импортов приложения;

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

   (a) 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;

   (b) 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 VEH-механизм) 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:

(a) Обработчик EXCEPTION_ACCESS_VIOLATION возобновляет доступ к целевой области и устанавливает флаг trap в регистр EFLAGS [47]. Регистрируется обработчик исключения EXCEPTION_SINGLE_STEP, где доступ к целевой области снова запрещается.

(b) 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.

5.3.4 Патч импортов

Мы парсили таблицу импортов PE-файла и составляли карты импортов, используя STL контейнер map. Для прототипа мы работаем только с нормальными импортами, delay-loaded и 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 [48] в секции .pdata [49] для размотки стека, т.к. разработка велась для x64. Такой подход не работал для перенаправленных (redirected) импортов (импортируемых системными библиотеками, в свою очередь, из других, нижележащих, системных библиотек) и с «простыми» функциями (которые, например, осуществляют взятие указателя из регистра системного пользования). «Простую» 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. Мы рассказывали о нём в соответствующем разделе. Для работы с ApiSet мы составляли отдельные карты, в которых описывалось отображение виртуальных DLL в логические. В будущем также планируется специальная обработка механизма SHIMS, обеспечивающего обратную совместимость для приложений, использующих устаревший системный API.

Перехват системных функций может осуществляться как с помощью механизма патча IAT (Import Address Table) (см. Рисунок 29), так и с помощью патча тела импортируемой функции (см. Рисунок 30) или подмены системной DLL на

свою реализацию. На Рисунке 29 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. Например, перехват *NtQueryMemory* для сокрытия областей инжектированного кода, перехват *NtAllocateVirtualMemory* и *NtVirtualProtect* с установкой PAGE_EXECUTE для отслеживания динамически добавляемых областей кода приложения, перехват *NtWriteProcessMemory* для детектирования попыток инжектирования в другие процессы со стороны приложения, перехват системного API создания процессов и т. д. Рисунок 30 демонстри-

Рисунок 29 — Перехват импортов через IAT

рует пример специального хука, осуществляющегося путём патча кода функции в системной библиотеке. Здесь изображен хук Native API, это достаточно низко лежащее API, и кто бы ни позвал данную функцию, управление будет передано в наш хук. После анализа и/или модификации параметров в данном примере осуществляется оригинальный системный вызов.

Рисунок 30 — Перехват импортов через патч тела функции

5.3.5 Работа с ApiSet

При составлении карты импортов мы столкнулись с механизмом ApiSetSchema от Microsoft. Мы провели обзор существующих источников и описали этот механизм в соответствующем разделе. Впервые мы столкнулись с этим механизмом при исследовании импорта тестовой программы *VirtualQuery*:

```
SIZE_T WINAPI VirtualQuery(
```

```
_In_opt_ LPCVOID                    lpAddress ,
_Out_   PMEMORY_BASIC_INFORMATION lpBuffer ,
_In_     SIZE_T                     dwLength
);
```

Это функция для работы с виртуальной памятью процесса, импортируемая из kernel32.dll, которая, в свою очередь, импортирует *VirtualQuery* из api-ms-win-core-memory-l1-1-0.dll. Так следует из Рисунка 31, на котором представлены импорты kernel32.dll в дизассемблере IDA Pro. Открыв с помощью дизассемблера api-ms-win-core-memory-l1-1-0.dll, которая содержится в \%SystemRoot%\System32\downlevel, мы увидим, что экспортируемая api-ms-win-core-memory-l1-1-0.dll функция *VirtualQuery* импортируется из kernel32.dll (см. Рисунок 32). На самом деле это виртуальная DLL, и этот импорт резолвится загрузчиком в процессе исполнения с помощью ApiSetSchema. Вот как будет выглядеть это перенаправление в процессе исполнения:

```
kernel32.dll: VirtualQuery (api-ms-win-core-memory-l1-1-0.dll: VirtualQuery ->
    kernelbase.dll: VirtualQuery) [import redirect with JMP]
```

Вот примеры отображений, которые нам удалось обнаружить:

- api-ms-win-core-com-l1-1-0.dll -> combase.dll
- api-ms-win-core-synch-l1-2-0.dll -> kernelbase.dll
- api-ms-win-core-rtlsupport-l1-1-0.dll -> ntdll.dll
- api-ms-win-core-errorhandling-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-processthreads-l1-1-0.dll -> kernel32.dll, kernelbase.dll
- api-ms-win-core-profile-l1-1-0.dll -> ntdll.dll
- api-ms-win-core-sysinfo-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-libraryloader-l1-2-0.dll -> kernelbase.dll
- api-ms-win-core-winrt-string-l1-1-0.dll -> combase.dll
- api-ms-win-core-synch-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-winrt-error-l1-1-0.dll -> combase.dll

```
.idata:0000000180074FD0 ; Imports from api-ms-win-core-memory-l1-1-0.dll
.idata:0000000180074FD0 ;
.idata:0000000180074FD0                 extrn UnmapViewOfFile_0:qword
.idata:0000000180074FD0                                         ; CODE XREF: sub_180001540+659↑p
.idata:0000000180074FD0                                         ; sub_180024120+52↑p
.idata:0000000180074FD0                                         ; DATA XREF: ...
.idata:0000000180074FD8                 extrn CreateFileMappingW_0:qword
.idata:0000000180074FD8                                         ; CODE XREF: sub_180001540+145↑p
.idata:0000000180074FD8                                         ; sub_180018600+60↑p
.idata:0000000180074FD8                                         ; DATA XREF: ...
.idata:0000000180074FE0                 extrn FlushViewOfFile_0:qword
.idata:0000000180074FE0                                         ; DATA XREF: FlushViewOfFile↑r
.idata:0000000180074FE8                 extrn MapViewOfFile_0:qword
.idata:0000000180074FE8                                         ; CODE XREF: sub_180001540+17E↑p
.idata:0000000180074FE8                                         ; sub_180018600+83↑p
.idata:0000000180074FE8                                         ; DATA XREF: ...
.idata:0000000180074FF0                 extrn MapViewOfFileEx_0:qword
.idata:0000000180074FF0                                         ; DATA XREF: MapViewOfFileEx↑r
.idata:0000000180074FF8                 extrn OpenFileMappingW_0:qword
.idata:0000000180074FF8                                         ; CODE XREF: OpenFileMappingA+39↑p
.idata:0000000180074FF8                                         ; DATA XREF: OpenFileMappingW↑r ...
.idata:0000000180075000                 extrn ReadProcessMemory_0:qword
.idata:0000000180075000                                         ; CODE XREF: sub_180001C24+5E↑p
.idata:0000000180075000                                         ; BaseReadAppCompatDataForProcessWorker+14C↑p ...
.idata:0000000180075008                 extrn VirtualAlloc_0:qword
.idata:0000000180075008                                         ; CODE XREF: sub_18000C5D0+5D↑p
.idata:0000000180075008                                         ; WerpGetDebugger+72↑p ...
.idata:0000000180075010                 extrn VirtualAllocEx_0:qword
.idata:0000000180075010                                         ; CODE XREF: WerpLaunchAeDebug+1EC↑p
.idata:0000000180075010                                         ; DATA XREF: VirtualAllocEx↑r ...
.idata:0000000180075018                 extrn VirtualFree_0:qword
.idata:0000000180075018                                         ; CODE XREF: sub_1800243F1+28↑p
.idata:0000000180075018                                         ; sub_18000C5D0+1F3F9↑p ...
.idata:0000000180075020                 extrn VirtualFreeEx_0:qword
.idata:0000000180075020                                         ; CODE XREF: WerpLaunchAeDebug+761↑p
.idata:0000000180075020                                         ; DATA XREF: VirtualFreeEx↑r ...
.idata:0000000180075028                 extrn WriteProcessMemory_0:qword
.idata:0000000180075028                                         ; CODE XREF: WerpLaunchAeDebug+31B↑p
.idata:0000000180075028                                         ; DATA XREF: WriteProcessMemory↑r ...
.idata:0000000180075030                 extrn VirtualProtect_0:qword
.idata:0000000180075030                                         ; DATA XREF: VirtualProtect↑r
.idata:0000000180075038                 extrn VirtualQueryEx_0:qword
.idata:0000000180075038                                         ; DATA XREF: VirtualQueryEx↑r
.idata:0000000180075040                 extrn VirtualQuery_0:qword
.idata:0000000180075040                                         ; CODE XREF: sub_18000A868+6D↑p
.idata:0000000180075040                                         ; DATA XREF: sub_18000A868+6D↑r ...
.idata:0000000180075048                 extrn VirtualProtectEx_0:qword
```

Рисунок 31 — Импорты kernel32.dll в дизассемблере IDA Pro

- api-ms-win-core-string-l1-l1-0.dll -> kernelbase.dll
- api-ms-win-core-winrt-l1-1-0.dll -> combase.dll
- api-ms-win-core-memory-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-datetime-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-debug-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-heap-l1-1-0.dll -> kernelbase.dll, ntdll.dll
- api-ms-win-core-heap-l2-1-0.dll -> kernelbase.dll
- api-ms-win-core-localization-l1-2-0.dll -> kernelbase.dll
- api-ms-win-core-processenvironment-l1-1-0.dll
- api-ms-win-core-registry-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-shlwapi-legacy-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-string-l2-1-0.dll -> kernelbase.dll

```
.rdata:00000001800012C7                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:00000001800012DC aVirtualfreeex  db 'VirtualFreeEx',0    ; DATA XREF: .rdata:off_1800010C8↑o
.rdata:00000001800012EA ; Exported entry  11. VirtualFreeEx
.rdata:00000001800012EA                 public VirtualFreeEx
.rdata:00000001800012EA ; BOOL __stdcall VirtualFreeEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD dwFreeType)
.rdata:00000001800012EA VirtualFreeEx   db 'kernel32.VirtualFreeEx',0
.rdata:00000001800012EA                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:0000000180001301 aVirtualprotect db 'VirtualProtect',0   ; DATA XREF: .rdata:off_1800010C8↑o
.rdata:0000000180001310 ; Exported entry  12. VirtualProtect
.rdata:0000000180001310                 public VirtualProtect
.rdata:0000000180001310 ; BOOL __stdcall VirtualProtect(LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PDWORD lpflOldProtect
.rdata:0000000180001310 VirtualProtect  db 'kernel32.VirtualProtect',0
.rdata:0000000180001310                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:0000000180001328 aVirtualprote_0 db 'VirtualProtectEx',0 ; DATA XREF: .rdata:off_1800010C8↑o
.rdata:0000000180001339 ; Exported entry  13. VirtualProtectEx
.rdata:0000000180001339                 public VirtualProtectEx
.rdata:0000000180001339 ; BOOL __stdcall VirtualProtectEx(HANDLE hProcess, LPVOID lpAddress, SIZE_T dwSize, DWORD flNewProtect, PD
.rdata:0000000180001339 VirtualProtectEx db 'kernel32.VirtualProtectEx',0
.rdata:0000000180001339                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:0000000180001353 aVirtualquery   db 'VirtualQuery',0     ; DATA XREF: .rdata:off_1800010C8↑o
.rdata:0000000180001360 ; Exported entry  14. VirtualQuery
.rdata:0000000180001360                 public VirtualQuery
.rdata:0000000180001360 ; SIZE_T __stdcall VirtualQuery(LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer, SIZE_T dwLength)
.rdata:0000000180001360 VirtualQuery    db 'kernel32.VirtualQuery',0
.rdata:0000000180001360                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:0000000180001376 aVirtualqueryex db 'VirtualQueryEx',0   ; DATA XREF: .rdata:off_1800010C8↑o
.rdata:0000000180001385 ; Exported entry  15. VirtualQueryEx
.rdata:0000000180001385                 public VirtualQueryEx
.rdata:0000000180001385 ; SIZE_T __stdcall VirtualQueryEx(HANDLE hProcess, LPCVOID lpAddress, PMEMORY_BASIC_INFORMATION lpBuffer,
.rdata:0000000180001385 VirtualQueryEx  db 'kernel32.VirtualQueryEx',0
.rdata:0000000180001385                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:000000018000139D aWriteprocessme db 'WriteProcessMemory',0
.rdata:000000018000139D                                         ; DATA XREF: .rdata:off_1800010C8↑o
.rdata:00000001800013B0 ; Exported entry  16. WriteProcessMemory
.rdata:00000001800013B0                 public WriteProcessMemory
.rdata:00000001800013B0 ; BOOL __stdcall WriteProcessMemory(HANDLE hProcess, LPVOID lpBaseAddress, LPCVOID lpBuffer, SIZE_T nSize,
.rdata:00000001800013B0 WriteProcessMemory db 'kernel32.WriteProcessMemory',0
.rdata:00000001800013B0                                         ; DATA XREF: .rdata:off_1800010880↑o
.rdata:00000001800013CC ; Debug information (IMAGE_DEBUG_TYPE_CODEVIEW)
.rdata:00000001800013CC asc_1800013CC   db 'RSDS'               ; DATA XREF: .rdata:0000000180001014↑o
.rdata:00000001800013CC                                         ; CV signature
.rdata:00000001800013D0                 dd 4BDC1A7Eh            ; Data1 ; GUID
.rdata:00000001800013D0                 dw 0D3AFh               ; Data2
.rdata:00000001800013D0                 dw 53C0h                ; Data3
.rdata:00000001800013D0                 db 0D1h, 0ADh, 2Dh, 0Ch, 4Ah, 0ADh, 0Bh, 30h; Data4
.rdata:00000001800013E0                 dd 1                    ; Age
```

Рисунок 32 — Экспорты api-ms-win-core-memory-l1-1-0.dll в дизассемблере IDA Pro

- api-ms-win-core-security-base-l1-1-0.dll -> kernelbase.dll
- api-ms-win-core-storage-exports-external-l1-1-0.dll -> windows.storage.dll

Если функции *GetModuleHandle*:

```
HMODULE WINAPI GetModuleHandle(
_In_opt_ LPCTSTR lpModuleName
);
```

в качестве аргумента передать имя виртуальной DLL, то она возвращает базовый адрес хостовой DLL (см. Рисунок 33). Некоторые виртуальные DLL могут иметь несколько логических DLL (см. Рисунок 34).

Работать с механизмом ApiSetSchema можно следующим образом:

1. Перед загрузкой PE. Учитывая имплементацию ApiSet, вычислять результирующие адреса импортов самостоятельно (т. е. реимплементировать недокументированные функции по работе с соответствующими структурами данных);

2. Во время загрузки PE. Использовать недокументированные низкоуровневые

функции по работе с хеш-таблицей ApiSetMap, которые зовёт загрузчик;

3. После загрузки PE. Использовать адреса, которые записал загрузчик, и определять перенаправление постфактум.

Мы выбрали последний подход. Мы составляем карты отображений виртуальных DLL в логические после загрузки для тех импортов, которые мы встречаем в процессе работы. Определить, что перед нами виртуальная DLL, можно по несоответствию между именем модуля и именем, полученным из таблицы экспортов по соответствующему базовому адресу, который вернула функция *GetModuleHandle*. Поскольку виртуальная DLL может иметь несколько логических, отображение хранится для каждой импортируемой функции. По адресу функции можно определить, в каком модуле находится функция на самом деле. Настоящее имя функции можно получить, используя API для отладки.

```
char *GetRealModuleName(char *moduleName)
{
  PVOID imageBase = GetModuleHandle(moduleName);   moduleName  0x000002a6200f7bd0 "api-ms-win-core-processthreads-l1-1-0.dll"

  PIMAGE_NT_HEADERS pImageNtHeaders = (PIMAGE_NT   imageBase  kernel32.dll!0x00007fff22930000 (load symbols for additional information)
  PIMAGE_EXPORT_DIRECTORY pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)RVA_TO_ADDR(imageBase, pImageNtHeaders->Optional

  return (char *)RVA_TO_ADDR(imageBase, pExportDirectory->Name);
}
```

Рисунок 33 — Вызов функции *GetModuleHandle* для виртуальной DLL

```
g_apisetMap | { size=0x000000000000001b }
[0]  ("api-ms-win-core-com-l1-1-0.dll", { size=0x0000000000000009 })
[1]  ("api-ms-win-core-datetime-l1-1-0.dll", { size=0x0000000000000002 })
[2]  ("api-ms-win-core-debug-l1-1-0.dll", { size=0x0000000000000003 })
[3]  ("api-ms-win-core-errorhandling-l1-1-0.dll", { size=0x0000000000000005 })
[4]  ("api-ms-win-core-heap-l1-1-0.dll", { size=0x0000000000000003 })
first   "api-ms-win-core-heap-l1-1-0.dll"
second    { size=0x0000000000000003 }
[0]   ("GetProcessHeap", {virtualDllName=0x00000211d56af260 "api-ms-win-core-heap-l1-1-0.dll" hostDllName=0x00000211d54bcef0 "kernelbase.dll" ...})
[1]   ("HeapFree", {virtualDllName=0x00000211d56af260 "api-ms-win-core-heap-l1-1-0.dll" hostDllName=0x00000211d54bd210 "ntdll.dll" ...})
[2]   ("HeapSetInformation", {virtualDllName=0x00000211d56af260 "api-ms-win-core-heap-l1-1-0.dll" hostDllName=0x00000211d54be110 "kernelbase.dll" ...})
[Raw View]  0x00000211d91152d8 {...}
[12] ("api-ms-win-core-profile-l1-1-0.dll", { size=0x0000000000000001 })
[13] ("api-ms-win-core-registry-l1-1-0.dll", { size=0x0000000000000004 })
[14] ("api-ms-win-core-rtlsupport-l1-1-0.dll", { size=0x0000000000000003 })
```

Рисунок 34 — Отображение виртуальной DLL в разные логические

5.3.6 Работа с SHIMS

В будущем планируется поддержать также механизм SHIMS, о котором мы писали в соответствующем разделе. Мы столкнулись с механизмом заплаток при обработке импортов (см. Рисунок 35). С помощью функции *GetProcAddress* мы хотим получить адрес функции *RtlFreeHeap* в библиотеке ntdll.dll. Однако функция *GetProcAddress* пропатчена загрузчиком в рамках механизма SHIMS и возвращает адрес функции из AcLayers.dll. Таким образом, функция *RtlFreeHeap* была перехвачена, и вместо неё будет вызываться функция *APIHook_RtlFreeHeap*.

```
198   HMODULE hNTDLL = GetModuleHandle("ntdll.dll");      hNTDLL | ntdll.dll!0x00007fff06e90000 {unused=0x00905a4d }
199   PVOID pRtlFreeHeap = GetProcAddress(hNTDLL, "RtlFreeHeap");
200
201   if (!GetSectionAddress("ntdll.dll", (DWORD_PTR)pRtlFreeHeap, &g_sectionInfo))      g_sectionInfo | {name=0x0000000000000000 <NULL> base=0x000000
202   {
203     printf("Can't get section info");
204     return -1;
205   }
206
207   if (!SetSectionNotAccessible(piProcInfo, &g_sectionInfo))
208   {
209     printf("Can't set section unaccessible");
210     return -1;
100 %
Autos
Name                                  Value
&g_sectionInfo                        0x00007ff62387c458 {DMA_POC.exe!_SECTION_INFO g_sectionInfo} {name=0x0000000000000000 <N
GetProcAddress                        0x00007ff62385c8d8 {DMA_POC.exe!GetProcAddress}
hNTDLL                                ntdll.dll!0x00007fff06e90000 {unused=0x00905a4d }
pRtlFreeHeap                          0x00007ffeda7c4120 {AcLayers.dll!NS_FaultTolerantHeap::APIHook_RtlFreeHeap(void)}
```

Рисунок 35 — Перехват *RtlFreeHeap* механизмом SHIMS

5.3.7 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 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 памяти при исполнении представлен на Рисунке 36.

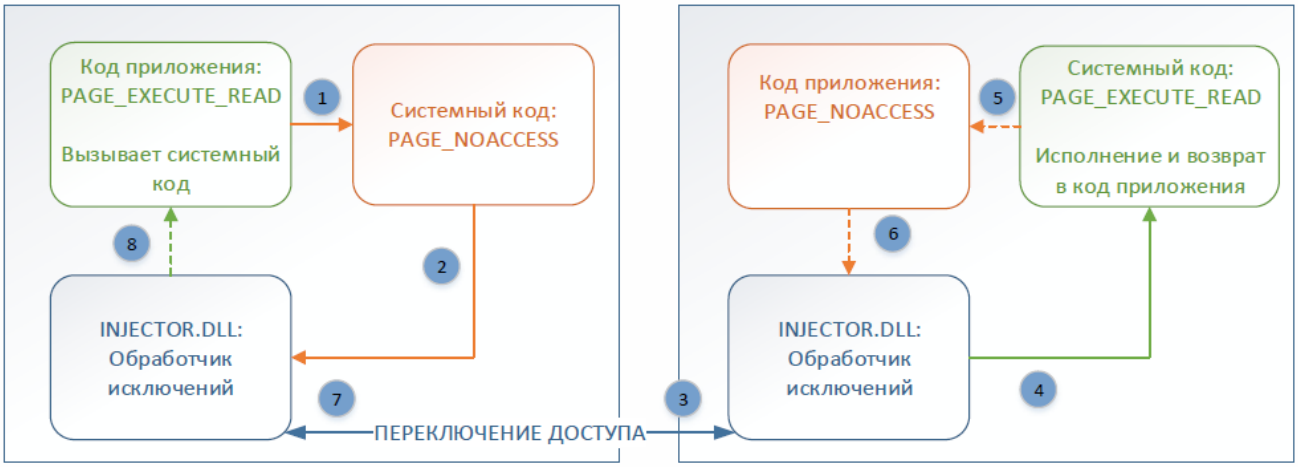


Рисунок 36 — Переключение карты памяти в однопоточном случае
Когда работает код приложения, код всех системных библиотек объявлен не присутствующим в памяти. Когда происходит передача управления коду системной библиотеки (1), возникает исключительная ситуация (EXCEPTION_ACCESS_VIOLATION), и вызывается 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.

Данный подход позволит не потерять контроль в случаях, если код вредоносного ПО попытается скопировать часть кода из системной библиотеки и пе-

редать управление в середину системной функции. Будут также задетектированы случаи, когда приложение получает control flow посредством обратного вызова из системной функции (например, *EnumFonts*).

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 была 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. Однако мы не могли отказаться от ntdll.dll, в которой 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.

5.3.7.1 Производительность

Генерация page fault'ов может привести к деградации производительности 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 может применяться TLB cache split [50] или ключи защиты для страниц пользователя от Intel (PKRU регистр) [51]. 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 мануала Intel [52] 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 в данный момент Memory Manager не отгружает на диск страницы 4 уровня.

Альтернативно, можно использовать более сложную технику, основанную

на аппаратной поддержке виртуализации. В данном подходе мы используем для анализируемого процесса гостевое страничное преобразование. При этом гость 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 CETP [53], 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.

5.3.8 Маскировка аналитических инструментов

Чтобы сделать инжектируемую DLL невидимой при переключениях карты памяти, необходимо отказаться от использования системных библиотек и C RunTime библиотеки (вызывающей системные DLL). Мы уже отказались от использования kernel32.dll, kernelbase.dll и перешли на Native API. Таким образом, в настоящий момент инжектированный модуль использует только ntdll.dll. В дальнейшем предлагается перейти на системные вызовы, но полностью избавиться от ntdll.dll можно будет только с введением ядерных компонент, т. к. в ntdll.dll реализован механизм обработки исключений. Мы отказались от C RunTime и вместо этого используем свою минимально необходимую реализацию, куда входит работа с памятью (*malloc*, *free*). Ассоциативный массив из STL контейнера мы заменили своей реализацией с использованием хеш-таблицы и хеш-функции

MurmurHash3 [54].

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, 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 достает список модулей, используя PEB. Также для маскировки аналитических инструментов в системе будут работать 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 разрешен 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.

5.3.9 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 2 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 про-

цесса). Опишем оба этих подхода более подробно.

Для работы с многопоточными приложениями нами был разработан механизм, при котором приложение разделяется на 2 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 изображен на Рисунке 37. 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 инспектирование 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 тяжелейшей технической задачей. Кратко опишем ключевые идеи реализации. Для создания близнецов будет применяться дуплицирование PEB (Process Environment Block), дуплицирование TEB (Thread Environment Block), дуплицирование системных дескрипторов (handles),

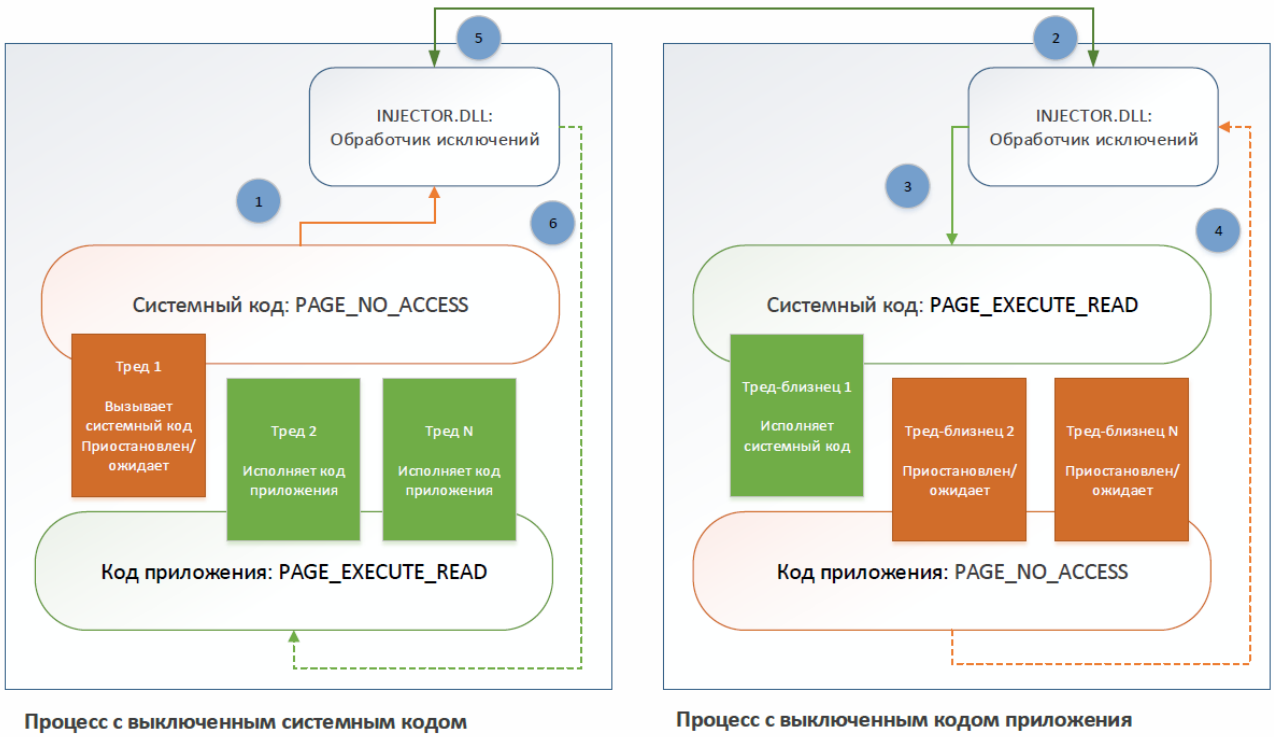


Рисунок 37 — Переключение карты памяти в многопоточном случае при использовании механизма дуплицирования VAD

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 [55; 56], 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.

Таким образом, синхронизация адресных пространств представляет собой сложную задачу, однако позволяет использовать простой механизм переключе-

ния. 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 исполняющемся 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 операционной системы (в отличие от Linux-систем, где можно просто модифицировать код ядра). С этой точки зрения подход с разработкой планировщика кажется нам более перспективным.

5.3.10 Примеры работы

В данный момент нами был разработан прототип ToolChain для анализа однопоточных приложений, который был протестирован на Windows 10x64. Механизм патча и мониторинга импортов не зависит от поточности, хоть и требует некоторой синхронизации лога. Мы демонстрируем его работу на notepad.exe. На Рисунке 38 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.

```
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
```

Рисунок 38 — Патч импортов notepad.exe

На Рисунке 39 представлен фрагмент лога, полученного в процессе работы notepad.exe.

```
[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]
```

Рисунок 39 — Лог системных вызовов 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]
```

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 Рисунке 40. 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 здесь [57]. 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
```

Рисунок 40 — Переключение карты памяти

5.3.11 Машинное обучение

В отличие от внешних событий, таких как файловые, реестровые операции, создание процессов, мониторинг которых легко осуществить на машине конечного пользователя, 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. Более подробно об этом рассказано в следующем разделе.

5.4 Сценарии использования инструментов

Разработанный комплекс для запуска вредоносного ПО и инструментарий динамического анализа могут быть частью более широкой системы аналитических инструментов. Самый простой сценарий использования разработанных инструментов представлен на Рисунке 41. Здесь образцы для анализа управляющими средствами Sandbox скачиваются из доступных баз данных в Интернете — например, VirusTotal. Они запускаются на тестовом сервере Sandbox, где собирается информация о поведении образца. Собранная информация обрабатывается экспертом-аналитиком. На основе этих данных он может предлагать улучшения инструментов защиты конечного пользования: обновление базы данных хешей и сигнатур, улучшение эвристик и предсказывающих моделей, утилиты для очист-

ки системы (например, декрипторы).

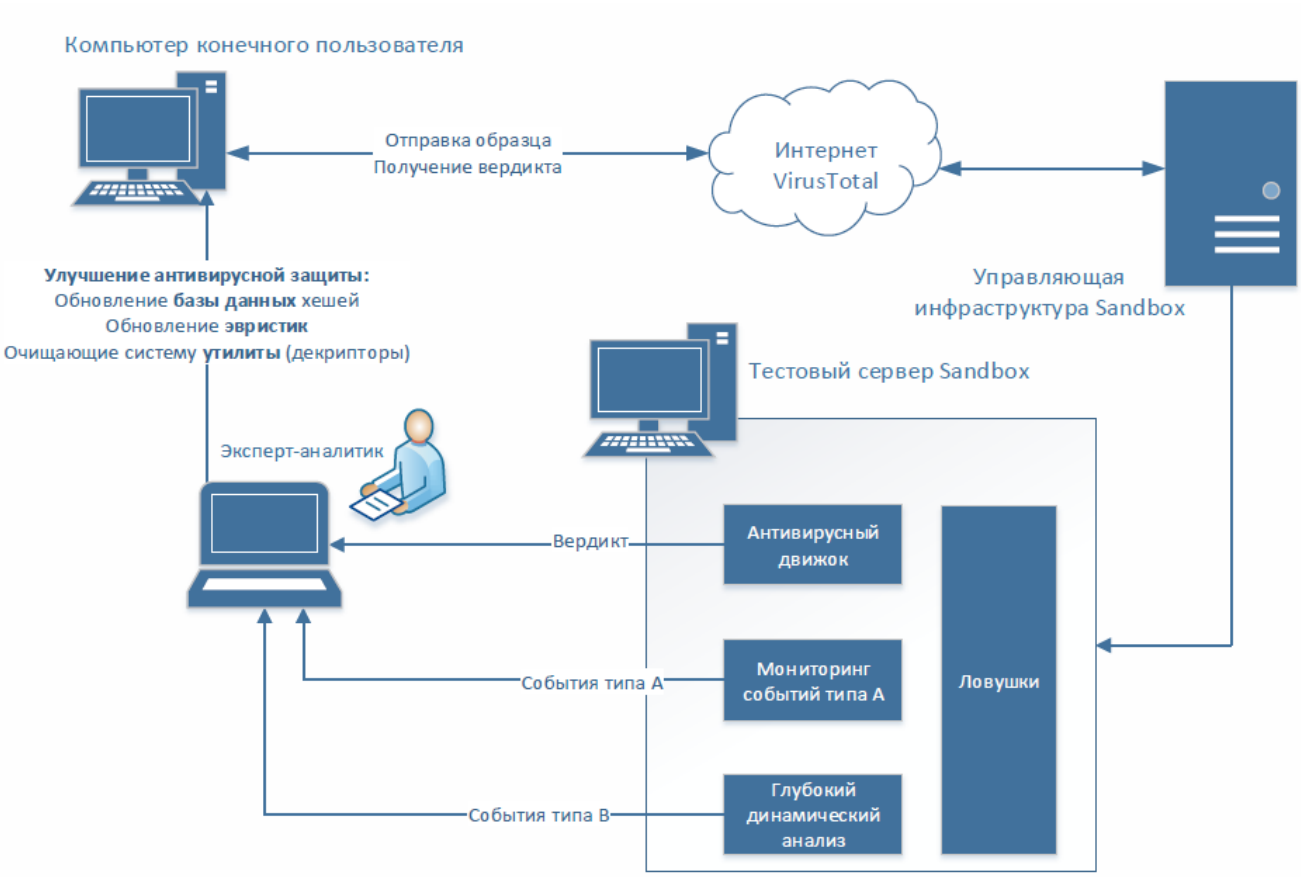


Рисунок 41 — Простой сценарий использования инструментов

Мы подробно опишем более сложный сценарий, в рамках которого похожий цикл работы аналитических инструментов автоматизирован с применением облачных технологий и машинного обучения.

Тестовый запуск программного обеспечения (т. е. контролируемый, под наблюдением аналитических инструментов) позволяет точнее определить вредоносность образца, но требует развёртывания песочницы на машине конечного пользователя (виртуальной или эмулируемой среды для запуска) и сопряжён с рисками заражения операционной системы, потери производительности системы, недостаточной точности вердикта вследствие столкновения с защитой вредоносного ПО от запусков в тестовой среде.

Традиционные методы всегда искали баланс между полнотой анализа и деградацией производительности системы конечного пользователя. При этом не происходило разделения анализа на «легкий» и «тяжёлый» и переноса «тяжёлого»

анализа с компьютеров конечных пользователей на отдельный комплекс тестовых серверов. «Тяжёлый» анализ производился на компьютере конечного пользователя с применением технологий виртуализации или эмуляции, либо производился аналитиком-экспертом на лабораторном оборудовании. Наше решение позволяет не развёртывать песочницу с аналитическими инструментами на машине конечного пользователя, а воспользоваться преимуществом облачных технологий.

Наша идея состоит в том, что аналитический инструментарий разделен на 2 части: аналитические инструменты, работающие на компьютерах конечных пользователей, и аналитические инструменты глубокого анализа (ToolChain). Таким образом, среда для тестового запуска программного обеспечения разворачивается не на пользовательском компьютере, а на специальных тестовых серверах. Коммуникации между пользовательским и тестовым компонентами производятся через облачный сервис. Необходимость детального исследования определяется модулем машинного обучения. Целью является повышение точности анализа и снижение нагрузки на ресурсы пользовательских машин. Решение о передаче неизвестного образца для дополнительного исследования может приниматься пользователем или аналитиком (в зависимости от бизнес-модели). Метаданные — хеши образцов и данные профилирования событий A — передаются в любом случае, в том числе для детектирования эпидемий.

На машине конечного пользователя аналитическими инструментами обрабатываются события типа A (внешние события), сбор которых не приводит к деградации производительности операционной системы, и работает модель машинного обучения, которая по собранным событиям типа A предсказывает потенциальное наличие событий типа B (внутренние события, прямой сбор которых оказывает значительное воздействие на производительность системы) и вероятность того, что исследуемое ПО является вредоносным.

В облачной инфраструктуре развернут механизм передачи подозрительного ПО на программно-аппаратный комплекс Sandbox, где на тестовых серверах полученные образцы запускаются под контролем аналитических инструментов

глубокого анализа.

Таким образом, мы позволяем вредоносному ПО заразить тестовые машины вместо компьютера конечного пользователя. При этом тестовые серверы:

1. Максимально имитируют пользовательский PC;
2. Содержат большое количество ПО устаревших версий с уязвимостями для повышения вероятности заражения;
3. Используют механизм Ловушек (Honey Pots) для повышения вероятности срабатывания векторов атаки вредоносного ПО;
4. Способны ускорять течение календарного времени для преодоления задержек активации вредоносного ПО или временных бомб;
5. Способны скрыть наличие аналитических инструментов от вредоносного ПО;
6. Содержат аналитические инструменты глубокого анализа, которые осуществляют полный контроль над работой исследуемого приложения (в том числе с деградацией производительности), собирая события B.

Модуль машинного обучения постоянно до-обучается по результатам запуска подозрительного ПО на тестовых серверах (путём нахождения корреляций между событиями типа A и событиями типа B), при этом модель на машинах конечных пользователей обновляется в режиме Continuous Integration. На основе событий A обученная модель детектирует подозрительные процессы и временно блокирует их исполнение на машине конечного пользователя (до получения вердикта из облака).

На высоком уровне схема, к которой мы стремимся, выглядит следующим образом (см. Рисунок 42). Встроенные в продукт инструменты на компьютерах конечных пользователей собирают и обрабатывают события типа A. На основе

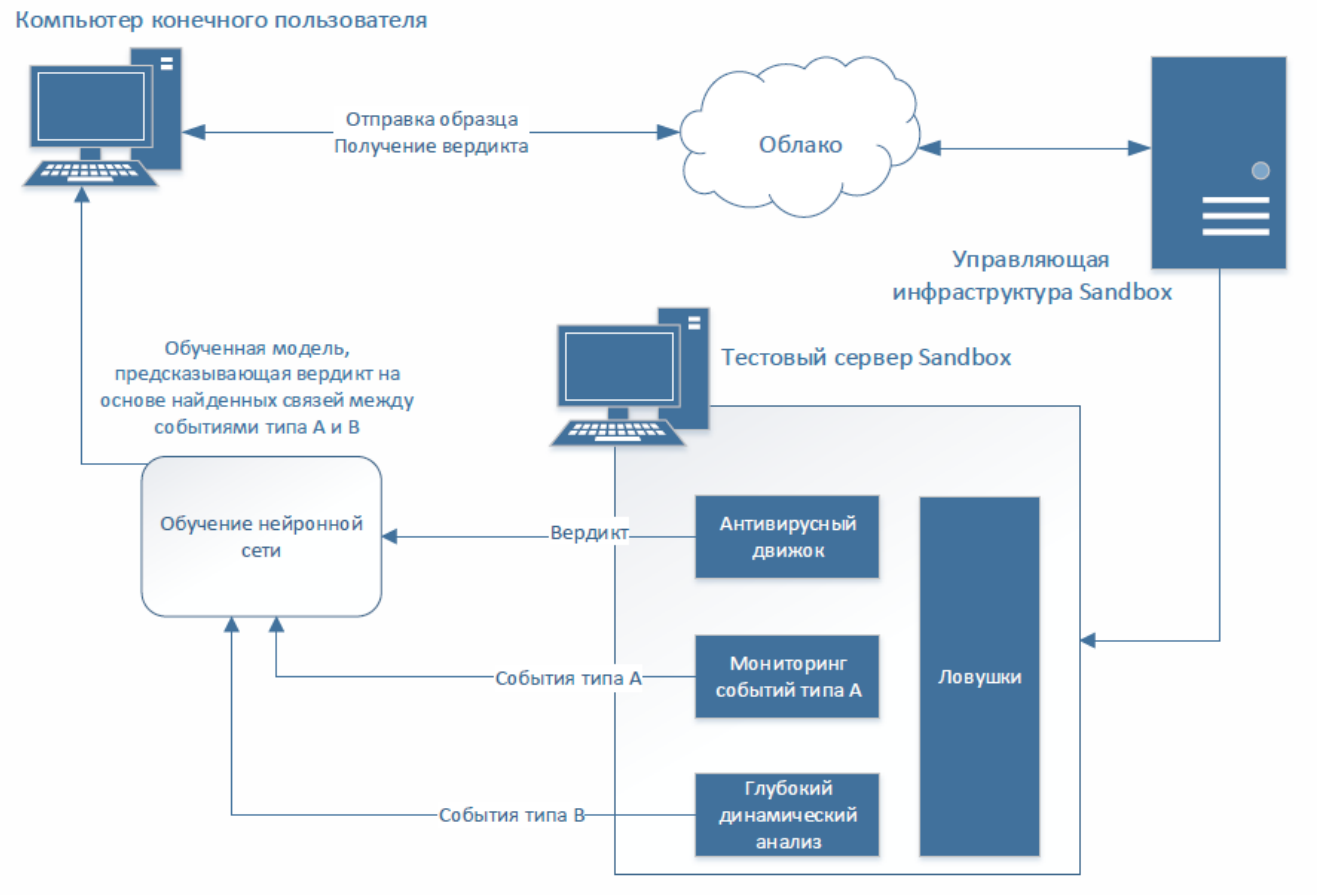


Рисунок 42 — Автоматизированный сценарий использования инструментов

этих событий анализатор этого уровня (основанный, например, на нейронной сети) делает заключение о том, что процесс является подозрительным, останавливает его и предлагает пользователю отправить его в Sandbox для детального анализа, где образец будет запущен на тестовом сервере, и заражение будет происходить под контролем аналитических инструментов глубокого анализа. Если пользователь подтверждает своё согласие, образец загружается в облако, откуда управляющими средствами Sandbox отправляется на один из тестовых серверов. Инструментарий тестового сервера Sandbox, помимо средств мониторинга внешних событий типа А, может включать в себя антивирусный движок, ловушки (Honey Pots) и инструменты глубокого динамического анализа. Здесь формируется отчёт о зарегистрированных событиях типа А и В и вердикт о том, является ли образец вредоносным. Вердикт отправляется пользователю, а собранные данные (события А, В и конечный вердикт) используются для обучения модели машинного обучения (например, нейронной сети), позволяющей по известным результатам типа А

предсказывать вердикт на основе найденных корреляций между событиями типа A и B. Обучение модели происходит в отдельной подсистеме, после чего обновлённая модель поставляется на компьютеры конечных пользователей. Обученная модель работает на компьютерах конечных пользователей для детектирования подозрительного программного обеспечения.

6 Заключение

В данной работе рассматривается обеспечение автоматического сбора данных о поведении вредоносного ПО для последующего анализа. Мы провели обзор существующих решений по запуску и анализу вредоносного ПО и остановились на запуске вредоносного ПО на реальном аппаратном обеспечении (с применением виртуальных машин для увеличения производительности) под контролем аналитических инструментов, работающих на уровне пользователя или ядра операционной системы. По результатам работы были созданы соответствующие прототипы, которые в данный момент проходят апробацию.

В работе описаны сетевая архитектура и архитектура развёртывания программного-аппаратного комплекса Sandbox, включающая решения по обеспечению безопасности и автоматизации запуска вредоносного программного обеспечения. По результатам исследования был спроектирован и запущен прототип Sandbox, состоящий из 7 тестовых серверов и 1 управляющего сервера, расположенных в ЦОД. Исследование было сфокусировано на шифровальщиках-вымогателях (crypto-ransomware), ориентированных на операционные системы семейства Windows. Для тестирования работы системы был подготовлен набор ransomware, состоящий из 317 образцов. Затем мы расшили фокус до Linux-систем. В настоящее время набор шифровальщиков, ориентированных на заражение Linux, не очень большой, нам удалось собрать 7 образцов.

Для увеличения пропускной способности Sandbox на части тестовых серверов был установлен гипервизор Virtuozzo 6. Тестовые сценарии можно запускать как на виртуальных машинах, так и на реальном аппаратном обеспечении,

поскольку вредоносное ПО может отказываться работать в виртуальной среде.

На текущий момент существуют автоматические сценарии работы для агентов 2-х типов: (1) осуществляющих сбор данных о происходящем в операционной системе, (2) обеспечивающих антивирусную защиту.

Таким образом, нами была выполнена задача построения программно-аппаратного комплекса для автоматического запуска образцов вредоносного программного обеспечения (1). Прототип Sandbox удовлетворяет сформулированным требованиям автоматизации, безопасности, изоляции и корректности запуска.

Мы произвели интеграцию с лабораторией машинного обучения, в рамках которой из базы данных мы получаем хеши образцов для запуска, необходимые файлы скачиваются из базы данных VirusTotal или из кэширующего хранилища, вредоносное ПО запускается на тестовом сервере Sandbox под наблюдением агента Process Monitor. На тестовом сервере настроено окружение, эмулирующее пользовательский ПК, а также работают инструменты защиты аналитических программ и логов. Собранные логи фильтруются, переводятся в формат .xml и передаются в лабораторию машинного обучения. Полученные логи используются для обучения модели Random Forest, осуществляющей классификацию программ на подозрительные и не вызывающие подозрения.

Sandbox используется для автоматического тестирования программного обеспечения на способность детектировать вредоносное ПО. Тестирование производится как для аналитических инструментов для Windows-систем, так и для находящихся в разработке инструментов, обеспечивающих защиту Linux-систем. Мы произвели интеграцию с данными агентами, в рамках которой из хранилища сборок мы получаем последнюю версию анти-ransomware инструментов, устанавливаем их на тестовую систему и запускаем в системе образцы вредоносного ПО. Взаимодействуя с агентами с помощью REST API, мы отправляем контрольные команды и проверяем, что заражение системы было предотвращено.

По результатам проделанной работы мы выполнили задачи интеграции с

аналитическими инструментами, осуществляющими сбор данных о поведении запускаемых образцов (2), и интеграции с лабораториями, осуществляющими анализ собранных данных (4). В соответствии с поставленными требованиями, интеграция проведена на стадиях перед, во время и после запуска вредоносного ПО.

Следующим важным этапом развития Sandbox стала разработка собственных инструментов динамического анализа вредоносного программного обеспечения — ToolChain. В данной работе 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.

Разработанные инструменты представляют собой Sandbox-агент 1-го типа, а запуск образца вредоносного ПО с агентом ToolChain является одним из автоматизированных сценариев на тестовом сервере Sandbox. Следовательно, выполнена задача разработки собственных аналитических инструментов для сбора данных о поведении вредоносного ПО (3). Текущий прототип соответствует предъявленным требованиям и демонстрирует возможность мониторинга и хука про-

извольного системного API и контроля над передачей управления.

Таким образом, мы достигли поставленной цели и обеспечили автоматический сбор данных о поведении вредоносного ПО для последующего анализа. В дальнейшем планируется сосредоточиться на улучшении инструментов динамического анализа. Было проведено исследование механизмов детектирования аналитических инструментов и уклонения вредоносного ПО от анализа. 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).

Мы предложили сценарий для совместного использования программно-аппаратного комплекса для запуска вредоносного ПО и инструментов глубокого динамического анализа в рамках более широкой системы аналитических инструментов. Сценарий использования может быть как простой, в котором Sandbox и ToolChain выступают лишь в качестве платформы для автоматической поставки логов, которые затем анализируются экспертом по информационной безопасности для определения алгоритма работы вредоносного ПО, написания утилит для очистки системы, добавления хешей в базу данных, написания эвристик и другой аналитической деятельности (работа с известными образцами, размещёнными в Интернете). Так и сложный, в рамках которого применяется передача образцов в Sandbox напрямую с компьютеров конечных пользователей, где работает до-обучающаяся на собираемых данных модель машинного обучения, предсказывающая вредоносность образца. В основе её работы — предполагаемые нами корреляции между внутренними событиями, доступными глубокому анализатору ToolChain, и внешними событиями, видимыми на компьютерах конечных пользователей.

Разработанные прототипы проходят апробацию в компании Acronis. Комплекс Sandbox был спроектирован, имплементирован, при этом в процессе имплементации были преодолены различные барьеры, которые подробно описаны в работе, а также внедрён в процессы исследования и тестирования. Создание комплекса сопровождалось детальной документацией с точки зрения архитектурных решений и практического использования. Инструменты ToolChain были прототипированы для анализа однопоточных приложений и спроектированы для многопоточного случая. Мы также произвели интеграцию для автоматической поставки собираемых в Sandbox данных в лабораторию машинного обучения. Таким образом, продемонстрирован теоретический и практический пример средств проактивной защиты от вредоносного программного обеспечения.

7 Список литературы

1. Malware // AV-TEST. Дата обновления: 08.05.2018. URL: https://www.av-test.org/en/statistics/malware/ (дата обращения: 15.05.2018).

2. Egele M., Scholte Th., Kirda E., Kruegel C. A survey on automated dynamic malware analysis techniques and tools // ACM Computing Surveys, 2012. Vol. 44, No. 2, Article 6. URL: https://www.cs.ucsb.edu/%7Echris/research/doc/acmsurvey12_dynamic.pdf (дата обращения: 15.05.2018).

3. Antivirus software // Wikipedia. Дата обновления: 05.05.2018. URL: https://en.wikipedia.org/wiki/Antivirus_software (дата обращения: 15.05.2018).

4. Ucci D., Aniello L., Baldoni R. Survey on the Usage of Machine Learning Techniques for Malware Analysis // ACM Transactions on the Web, 2017. Vol. 1, No. 1, Article 1. URL: https://arxiv.org/pdf/1710.08189.pdf (дата обращения: 15.05.2018).

5. Pfeffera A., Ruttenberga B., Kellogga L., Howarda M., Calla C., O'Connora A., Takataa G., Reillya S., Pattena T., Taylora J., Halla R., Lakhotiab A., Milesb C., Scofieldc D., Frankc J. Artificial Intelligence Based Malware Analysis // arxiv.org. 2017. Дата обновления: 27.04.2017. URL: https://arxiv.org/pdf/1704.08716.pdf (дата обращения: 15.05.2018).

6. Вредоносная программа // Википедия. Дата обновления: 17.04.2018. URL: https://ru.wikipedia.org/wiki/Вредоносная_программа (дата обращения: 15.05.2018).

7. Ransomware // Wikipedia. Дата обновления: 26.04.2018. URL: https://en.wikipedia.org/wiki/Ransomware (дата обращения: 15.05.2018).

8. Вирус-вымогатель // Википедия. Дата обновления: 18.04.2018. URL: https://ru.wikipedia.org/wiki/Вирус-вымогатель (дата обращения: 15.05.2018).

9. Young A., Yung M. Cryptovirology: extortion-based security threats and countermeasures // IEEE Symposium on Security and Privacy. 1996. pp. 129–140. doi:10.1109/SECPRI.1996.502676.

10. Генератор псевдослучайных чисел // Википедия. Дата обновления: 21.01.2018. URL: https://ru.wikipedia.org/wiki/Генератор_псевдослучайных_чисел (дата обращения: 15.05.2018).

11. 2017 state of malware report // Malwarebytes LABS. 2017. URL: https://www.malwarebytes.com/pdf/white-papers/stateofmalware.pdf (дата обращения: 15.05.2018).

12. Crowe J. Must-Know Ransomware Statistics 2017 // blog.barkly.com. 2017. https://blog.barkly.com/ransomware-statistics-2017 (дата обращения: 15.05.2018).

13. Антивирусная программа // Википедия. Дата обновления: 06.05.2018. URL: https://ru.wikipedia.org/wiki/Антивирусная_программа (дата обращения: 15.05.2018).

14. Обнаружение, основанное на сигнатурах // Википедия. Дата обновления: 10.04.2017. URL: https://ru.wikipedia.org/wiki/Обнаружение,_основанное_на_сигнатурах(дата обращения: 15.05.2018).

15. Проактивная защита // Википедия. Дата обновления: 21.03.2018. URL: https://ru.wikipedia.org/wiki/Проактивная_защита (дата обращения: 15.05.2018).

16. Balci E. Искусство антидетекта. Часть 1 — Введение в техники детектирования // securitylab.ru. 2017. Дата обновления: 27.03.2017. URL: https://www.securitylab.ru/analytics/485677.php (дата обращения: 15.05.2018).

17. S.M.A.R.T. // Википедия. Дата обновления: 16.05.2018. URL: https://ru.wikipedia.org/wiki/S.M.A.R.T (дата обращения: 29.05.2018).

18. Wojner Ch. Mass Malware Analysis: A Do-It-Yourself Kit // CERT.at. 2009. URL: http://cert.at/static/downloads/papers/cert.at-mass_malware_analysis_1.0.pdf (дата обращения: 15.05.2018).

19. What is Cuckoo? // Cuckoo Sandbox. URL: https://cuckoo.sh/docs/introduction/what.html (дата обращения: 15.05.2018).

20. Building an Automated Behavioral Malware Analysis Environment using Open Source Software // SANS Institute. 2009. URL: https://www.sans.org/reading-room/whitepapers/tools/buildingautomated-behavioral-malware-analysis-environment-open-source-software-33129 (дата обращения: 15.05.2018).

21. 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 (дата обращения: 15.05.2018).

22. PE-формат. Часть 3 — Импорт // kaimi.io. 2011. Дата обновления: 04.09.2011. URL: https://kaimi.io/2011/09/pe-format-import/ (дата обращения: 15.05.2018).

23. Касперски К. Путь воина — внедрение в PE/COFF-файлы // Системный администратор. №6 (19). 2004. URL: http://samag.ru/archive/article/297 (дата обращения: 15.05.2018).

24. Hosseini A. Ten Process Injection Techniques: A Technical Survey Of Common And Trending Process Injection Techniques // www.endgame.com/blog/technical-blog. 2017. Дата обновления: 18.06.2017. URL: https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process (дата обращения: 15.05.2018).

25. Process Handles and Identifiers // Microsoft. URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms684868(v=vs.85).aspx (дата обращения: 15.05.2018).

26. Renaud S. Runtime DLL name resolution: ApiSetSchema - Part I // blog.quarkslab.com. 2012. Дата обновления: 06.04.2012. URL: https://blog.quarkslab.com/runtime-dll-name-resolution-apisetschema-part-i.html (дата обращения: 15.05.2018).

27. Shilon O. On API-MS-WIN-XXXXX.DLL, and Other Dependency Walker Glitches // ofekshilon.com. 2016. Дата обновления: 05.2016. URL: https://ofekshilon.com/2016/03/27/on-api-ms-win-xxxxx-dll-and-other-dependency-walker-glitches/ (дата обращения: 15.05.2018).

28. MinWin // Wikipedia. Дата обновления: 09.04.2015. URL: https://ru.wikipedia.org/wiki/MinWin (дата обращения: 15.05.2018).

29. Api set resolution // lucasg.github.io. Дата обновления: 15.10.2017. URL: https://lucasg.github.io/2017/10/15/Api-set-resolution/ (дата обращения: 15.05.2018).

30. Windows API Sets // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/hh802935(v=vs.85).aspx?ranMID=24542&ranEAID=TnL5HPStwNw&ranSiteID=TnL5HPStwNw-sIxDxkaFicyshruRQoe53Q&tduid=(a589f20c7e87bca2f26a3a8c19b469ef)

(256380)(2459594)(TnL5HPStwNw-sIxDxkaFicyshruRQoe53Q)() (дата обращения: 15.05.2018).

31. Shim (computing) // Wikipedia. Дата обновления: 10.05.2018. URL: https://en.wikipedia.org/wiki/Shim_(computing) (дата обращения: 15.05.2018).

32. Батырь А. Windows 7: технология «системных заплаток» (Shims) // pcmag.com. URL: http://ru.pcmag.com/iznutri/4123/help/windows-7-bezopasnost-i-sovmestimost?p=5 (дата обращения: 15.05.2018).

33. Ionescu A. Secrets of the Application Compatilibity Database (SDB) – Part 3 // alex-ionescu.com. Дата обновления: 26.03.2007. URL: http://www.alex-ionescu.com/?p=41 (дата обращения: 15.05.2018).

34. Structured Exception Handling // Microsoft. URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms680657(v=vs.85).aspx (дата обращения: 15.05.2018).

35. Pietrek M. A Crash Course on the Depths of Win32™ Structured Exception Handling // Microsoft Systems Journal. 1997. URL: http://bytepointer.com/resources/pietrek_crash_course_depths_of_win32_seh.htm (дата обращения: 15.05.2018).

36. Exception Dispatching // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms679327(v=vs.85).aspx (дата обращения: 15.05.2018).

37. Exception Handling // Microsoft. URL: https://msdn.microsoft.com/ru-ru/library/windows/desktop/ms679329(v=vs.85).aspx (дата обращения: 15.05.2018).

38. About Structured Exception Handling // Microsoft. URL: https: //msdn.microsoft.com/ru-ru/library/windows/desktop/ms679270(v=vs.85).aspx (дата обращения: 15.05.2018).

39. Frame-based Exception Handling // Microsoft. URL: https://msdn.microsoft. com/en-us/library/windows/desktop/ms679353(v=vs.85).aspx (дата обращения: 15.05.2018).

40. Vectored Exception Handling // Microsoft. URL: https://msdn.microsoft. com/en-us/library/windows/desktop/ms681420(v=vs.85).aspx (дата обращения: 15.05.2018).

41. VirusTotal. URL: https://www.virustotal.com/ (дата обращения: 15.05.2018).

42. 2017 state of cyber report // F-Secure. 2017. P. 41. URL: https://www.f-secure.com/documents/996508/1030743/cyber-security-report-2017 (дата обращения: 15.05.2018).

43. WannaCry ransomware used in widespread attacks all over the world // Kaspersky LAB. 2017. Дата обновления: 12.05.2017. URL: https://securelist.com/wannacry-ransomware-used-in-widespread-attacks-all-over-theworld/78351/ (дата обращения: 15.05.2018).

44. New Petya/NotPetya/ExPetr ransomware outbreak // Kaspersky LAB. 2017. Дата обновления: 28.06.2017. URL: https://www.kaspersky.com/blog/new-ransomware-epidemics/17314/ (дата обращения: 15.05.2018).

45. Winlogon and GINA // Microsoft. URL: https://msdn.microsoft.com/ ruru/library/windows/desktop/aa380543(v=vs.85).aspx (дата обращения: 15.05.2018).

46. Filter Manager Concepts // Microsoft. 2017. Дата обновления: 20.04.2017. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/ifs/filter-manager-concepts (дата обращения: 15.05.2018).

47. 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. Vol. 1 3-15. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf (дата обращения: 15.05.2018).

48. RtlAddFunctionTable function // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680588(v=vs.85).aspx (дата обращения: 15.05.2018).

49. PE Format: The .pdata Section // Microsoft. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx#the_.pdata_section (дата обращения: 15.05.2018).

50. TLB Desynchronization (Split TLB) // Uninformed. 2007. URL: http://uninformed.org/index.cgi?v=6&a=1&p=21 (дата обращения: 15.05.2018).

51. Intel Software Developer's Manual. Vol. 3A 4-31.

52. Intel Software Developer's Manual. Vol. 3A 5-30.

53. Control-flow Enforcement Technology Preview // Intel. 2017. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf (дата обращения: 15.05.2018).

54. MurmurHash3 // github.com/aappleby. Дата обновления: 01.08.2016. URL: https://github.com/aappleby/smhasher/wiki/MurmurHash3 (дата обращения: 15.05.2018).

55. Ionescu A. The Linux kernel hidden inside Windows 10 // BlackHat. 2016. URL: https://github.com/ionescu007/lxss/blob/master/The%20Linux%20kernel%20hidden%20inside%20windows%2010.pdf (дата обращения: 15.05.2018).

56. Hammons J. Pico Process Overview // blogs.msdn.microsoft.com. 2016. Дата обновления: 23.05.2016. URL: https://blogs.msdn.microsoft.com/wsl/2016/05/23/pico-process-overview/ (дата обращения: 15.05.2018).

57. Tang J. How Windows 10 Implements Parallel Loading // threatmatrix.cylance.com. 2017. Дата обновления: 03.10.2017. URL: https://threatmatrix.cylance.com/en_us/home/windows-10-parallel-loading-breakdown.html (дата обращения: 15.05.2018).