Статья ROPInjector: Using Return-Oriented Programming for Polymorphism and Antivirus Evasion

ROPInjector: Using Return-Oriented Programming for Polymorphism and Antivirus Evasion

Аннотация

Один из недостатков существующих методов **полиморфизма** заключается в требовании наличия доступного для записи сегмента кода, либо помеченного как таковой в заголовке соответствующего раздела PE, либо путем изменения разрешений во время выполнения. Оба подхода идентифицируются антивирусами как зловредные, так как они редко встречаются в "доброкачественных" PE (только если они не упакованы). В данной работе мы предлагаем использовать **возвратно-ориентированное программирование** (**ROP**) в качестве нового способа достижения полиморфизма и обхода антивирусов. Для этого мы сумели разработать **ROPInjector**, который, получив любой ну упакованный 32-битный PE, преобразует полученный шеллкод в его "ROP-эквивалент" и пропатчивает его в PE файл (т.е заражает). После тестирования различных комбинаций результаты показывают, что **ROPInjector** может почти полностью обойти все антивирусы, используемые в онлайн-сервисе VirusTotal. Основным результатом данного исследования являются разработанные алгоритмы для:
- Анализа и манипулирования кодом на ассемблере x86
- Автоматического объединения гаджетов в цепочки (**ROP-chains**) с помощью **ROPInjector** для формирования безопасного и функционального
ROP-кода, эквивалентного заданному шеллкоду.

1. Вступление

Возвратно-ориентированное программирование (ROP) привлекло повышенное внимание в конце 2000-х годов как продвинутый метод повреждения стека, который может обой механизмы **предотвращения выполнения данных** (Data Execution Prevention). **ROP** - это переоткрытие кода потока, в котором программы обычно состоят из цепочки адресов в стеке, указывающих на фрагменты кода в атакуемом исполняемом файле (или его загруженных библиотеках), каждый из которых заканчивается командой возврата (`ret` and etc.). Эти "заимствованные" фрагменты кода называются гаджетами (**ROP-gadget**), а их "возврат" является вызовом следующего гаджета в цепи. Для большего понимания и сравнения с обычным кодом: **ROP-гаджеты** это те же самые инструкции, а **ESP** - счетчик.

Полиморфизм - это техника для обхода антивирусов, при которой код изменяетс себя (мутирует) при каждом новом запуске, но семантика (функция) кода при этом не изменяется. Благодаря этой технике антивирусы не могут создать и поставить сигнатуру для обнаружения шеллкода. Однако, существует недостаток, он заключается в требовании доступного для записи сегмента кода, либо помеченного как таковой в заголовке соответствующего раздела в PE, либо в возможности изменения разрешений во время выполнения. Оба подхода определяются антивирусами как зловредные, поскольку они редко встречаются в обычных PE, если только они не упакованы.

В этой работе мы утверждаем, что **ROP** является достаточно сильной альтернативой полиморфизму, которая устраняет необходимость в записываемом участке кода. Если более конкретно, то наиболее важным преимуществом в использовании **ROP** для обхода антивирусов является то, что такой заимствованный код (который из гаджетов) всегда является "доброкачественным" и проверяется на ложные срабатывания. Очевидно, что цепочка адресов возврата (return address) должна быть каким-то образом "встроена" в стек. Этот процесс включает в себя либо пуша адресов возврата в стек, либо простое копирование всей цепи из другой области памяти (возможно, из .data секции) и коррекцию указателя стека. Мы утверждаем, что:

- Код, необходимый для подобных операций является весьма распространенным и сойдет за "доброкачественный".
- Код может быть каким-либо образом рандомизирован и/или закодирован многими тривиальными и не только способами.
- Код в значительной степени зависит от PE, подверженного атаке и его image base, поскольку в наихудшем исходе он представляет из себя
кучу операций `push <VAi>`

Это происходит из-за того, что адреса гаджетов меняются в каждом PE и image base, следовательно, будет и изменяться размер и база инструкций для построения цепочки, даже если они "порождаются" из одного и того же исходного шеллкода. Учитывая эти особенности, **ROP** обеспечивает полиморфизм, не требуя наличия записываемого участка кода в памяти. Кодирование или декодирование может быть применено к цепи гаджетов в памяти (в стеке, не в кодовой секции). Также различные гаджеты могут быть случайно выбраны для одной и той же операции. Таким образом изменяется объем шеллкода.

Основываясь на вышеизложенных наблюдениях и доводах, в данной работе мы представляем вам **ROPInjector**, инструмент, который учитывая любой фрагмент шеллкода (далее также может упоминаться как исходный (шелл)код) и любой неупакованный исполняемый файл, будет преобразовывать шеллкод в его ROP-эквивалент и патчить его (т.е заражать) PE-файл. **ROPInjector** (написан на C + WinAPI) заражает PE и работает под архитектуру x86. Так как антивирусы достаточно часто обнаруживают малейшие отклонения от типичного расположения секций файла и их характеристик (как пример, вторая исполняемая секция с правами **RWX**), помимо преобразования кода в неопределяемый и неповторяющийся вид, разработанный инструмент решает несколько дополнительных вопросов для достижения скрытности, таких как позиционирование шеллкода в исполняемом файле-носителе и способ передачи управления шеллкоду. Более того, было произведено несколько тестов для оценки эффективности предложенного инструмента путем внедрения шеллкодов в известные программы, такие как Adobe Acrobat Reader, Firefox, Java и т.д. Результаты показывают, что предложенная нами техника в сочетании с простыми методами антипрофилирования может сделать обнаружение антивирусными программами неосуществимым.

2. Связанные работы и PoC

Насколько мы знаем, это первая работа, в которой PE-файлы заражаются с помощью ROP-закодированной полезной нагрузкой (**ROP-encoded payload**). Тем не менее, в этом разделе мы рассмотрим еще два связанных инструмента, имеющих одинаковые цели с **ROPInjector**: заражение PE-файлов обычным или зашифрованным шеллкодом с целью обхода антивирусов.

Первый, **Shellter**, сфокусирован на сохранении ориганльной структуры PE-файла, избегая внедрение шеллкода в заранее определенные места или изменения характеристик существующих разделов. Он реализует это путем перезаписи существующего кода, для которого будет передан контроль во время выполнения программы. **Shellter** также способен на повторное использование импортов оригинального PE для изменения разрешений на запись секции, содержащей шеллкод таким образом, чтобы можно было использовать шифрованный и самомодифицирующийся код. Он также способен внедрять "мусорный код" перед шеллкодом, который будет задерживать выполнение в качестве средства защиты от эмуляции. **Shellter** прокачен с точки зрения динамического выбора местоположения патча в шеллкоде (в отличие от расширения `.text` секции). Однако, не смотря на то, что в нем используется методика патчей, которая вносит изменчивость (в каком месте файла будет внедряться шеллкод), он основывается на традиционных методах **полиморфизма**, которые по-прежнему зависят от генерации сигнатур и обнаружения разрешений на запись или модификации секции .text в памяти. Более того, предложенный нами подход также вносит изменчивость, благодаря преобразованию в **ROP** (который зависит от PE-файла).

PEInject - все же это больше метод, нежели полнофункциональный инструмент. Он заключается во внедрении шеллкода в первое достаточно большое незаполненное пространство, найденое в секции `.text` и не кодирует и не модифицирует полезную нагрузку каким-либо образом, а также не ожидает других самомодифицирующихся или зашифрованных данных (или шеллкодов). Управление передается внедренному шеллкоду путем модификации адреса `EP` `NT_HEADER` PE-файла.

"Коэффиценты" скрытия обоих методов сравниваются с нашим подходом в разделе №4.

3. ROPInjector

3.1 Патчинг PE и передача управления шеллкоду

Для начала, патчинг PE-файла и передача управления шеллкоду должны быть выполнены наименее заметным способом. Вторая исполняемая секция будет слишком явной, т.к в подавляющем большинстве исполняемых файлов существует только одна исполняемая секция. Следующим, наиболее простым способом для реализации будет внедрение шеллкода в промежуток с 0xCC, обычно оставляемую компоновщиком между сегментами кода (обычно OBJ-файлами) в `.text` секции PE. Однако, не всегда размер таких промежутков будет достаточным. Важно заметить, что **ROPInjector** использует это пространство в наших целях, как мы заметим и проанализируем ниже.

По указанным выше причинам мы решили внедрять шеллкод в существующий раздел `.text` исполняемого файла и соответствующим образом исправить все заголовки разделов и релокаций. Чтобы передать управление шеллкоду мы заменим команды, на которые указывает `NT_HEADER.AddressOfEntryPoint` , на переход к шеллкоду, который добавляет эти замененные командды, после чего мы вернемся обратно к исходному потоку выполнения. Непосредственно указание адреса `EP` на шеллкод в этом случае игнорируется, т.к многие эвристические движки антивирусов озадачит тот факт, что адрес `EP` будет указывать на конец секции `.text` . Альтернатива передаче управления шеллкоду является хук любых вызовов `ExitProcess` , `exit` или другим подобным функциям. Эта техника, как будет показано далее, обходит поведенческое профилирование у антивирусов, использующих эмуляцию или sandbox. Это может быть связано с тем, что антивирусы либо эмулируют небольшую часть `EP` исполняемого файла из-за ограничений по времени сканирования, либо из-за отсутствия (универсальных) методов для запуска корректного выхода (многие программы не обрабатывают сигналы SIGINT и SIGTERM).

3.2 Анализ машинного кода архитектуры x86

Анализ машинного кода в структурах данных, которые можно легко обрабатывать, имеют ключевое значение для выполнение любого вида патчей, модификаций, пересборки и любого преобразования в **ROP**. Два наиболее важных фрагмента требуемой информации (англ. "Two are the most important pieces of information required"):
1) Источник и "пункт назначения" всех относительных ссылок (например, относительный переход и его "цель").
2) Какие регистры задеиствованы (чтение/запись) во время выполнения каждой команды, а также какие регистры свободны для изменения.
Первое необходимо для введения или удаления команд из сегмента кода без нарушения его работы. Второе будет полезным для более подходящей выборки гаджетов, либо путем выполнения пермутаций, либо путем использования гаджетов, содержащие лишние, но безопасные команды (в данном случае небезопасными являются команды ветвления, привилегированные команды или команды режима косвенной адресации, так как они могут вызывать ошибки, такие как нарушение прав доступа).

3.3 MOD/REG/RM и развертывание SIB

Команды, использующие режим косвенной адресации **MOD/REG/RM** со смещением или схему адресации **Scaled Index Byte (SIB)** в шеллкоде будут обрабатываться особым образом. К примеру, схема адресации **SIB** в шеллкоде обрабатываются не совсем привычным образом перед преобразованием в **ROP**. Такие команды нежелательны по следующим причинам:
1) Они довольно длинные (в лучшем, но не очень веротяном случае отводится 3 байта: 1 для опкода, 1 для **MOD/REG/RM** и 1 для **SIB**)
2) Они часто работают со многими регистрами общего назначения одновременно, тем самым резервируя их. А как уже упоминалось ранее, чем больше свободных регистров, тем лучше.
3) Их соответствующие гаджеты (если их получиться найти или внедрить) скорее всего не смогут повторно использоваться из-за использования констант смещения и индекса (например: `mov edx, [esi*2 + 16]`)
Для того, чтоб успешно избегать подобные ситуации, мы сводим такие команды к их арифметическим эквивалентам один за другим. Этот процесс называется разверткой и он выполняется для шеллкода перед любым преобразованием в **ROP**. Например, `mov eax, [ebx+ecx*2]` может быть заменено на:

Code:

```
[1'] mov eax, ecx
[2'] sal eax, 1
[3'] add eax, ebx
[4'] mov eax, [eax]
```

Если регистр `eax` занят для выполнения арифметических операций, то любой другой свободный регистр общего назначения подойдет для подобной операции.

Примечательно то, как развертывание разблокирует доступ к регистрам от одной атомарной команды ко многим. Например, в последнем примере `ecx` освобождается по адресу [1'], а `ebx` - по адресу [3']. Если бы, например, `eax` был освобожден на предыдущих 10 инструкциях, то инструкции [1'] - [3'] могли бы быть перемещены на 10 за ними, что приведет к появлению дополнительного регистра (т.е `ebx` и `ecx` , но не `eax` , который будет занят) в этом предшествующем сегменте кода.

3.4 Поиск гаджетов

Гаджеты в исполняемых секциях PE-файла должны заканчиваться одной из команд: `ret` , `retn` , `pop regX` , `jmp regX` . Исключением будет последний вариант, когда рассматриваемый гаджет должен быть сначал сопряжен с гаджетом-загрузчиком, который загружает требуемый адрес возврата в `regX` . Процесс начинается с поиска всех окончаний гаджетов и их временного помещения в "список" для хранения. Для каждого из этих окончаний дизассемблируется (разбирается) n байт предшествующего машинного кода для каджого n до максимальной глубины N (обычно около 20 байт). Если такое дизассемблирование выравнивается с окончанием (не может быть гарантировано, т.к команды x86 имеют переменную длину), значит гаджет найден. Гаджеты, содержащие любые привилегированные (к примеру: `sysenter` , `int` , `iret`) ветвления или инструкции, модифицирующие esp - отфильтровываются.

3.5 Преобразование гаджетов в Intermediate Representation (IR)

Гаджеты, найденные в вышеупомянутом процессе сначал будут анализироваться в виде инструкция-за-инструкцией (instruction-by-instruction) чтобы определить доступ к регистру. Поскольку гаджеты могут содержать "безопасные", но избыточные команды, их доступ к регистру должен проверяться на возможность модификации данного регистра (например, `mov ecx, eax; pop ecx; ret;` гаджет не может быть использован для перемещения `eax` в `ecx`), а также занятых регистров исходной инструкции, которая должна быть закодирована.

После этого они преобразуются в **IR**, состоящее из типо операции и 3-х операндов с различными значениями (в зависимости от типа). Если гаджет с несколькими командами содержит более одной представляемой команды, рассматриваться будет только первая. Однако, последующие будут также рассматриваться в других гаджетах с тем же окончанием из-за обратного процесса поиска гаджетов, который был описан в предыдущем абзаце. Примечателен тот факт, что при синтаксическом анализе подобного уровня, **IR** автоматически выполняет взаимно-однозначные перестановки. Это связано с тем, что как гаджеты, так и команды классифицируются в один из этих типов, на основе которого позже выполняется кодирование, а не инструкции как таковые. **IR** также полезен для выборки функции кодировщика, сопровождающего каждый гаджет. Такие кодировщики отвечают нам на вопрос: *"А может ли назначенный им гаджет кодировать данную инструкцию"*, а также за кодирование ее в список стековых операций, если это необходимо.

3.6 Внедрение гаджетов

Для того, чтобы улучшить преобразование исходного шеллкода, нам необходимо внедрять новые гаджеты, поскольку не всегда все нужные гаджеты находятся в PE-файле. Во-первых, для внедрения будут использоваться пространства `0xCC`, если они заполнены, то секция `.text` расширяется. Внедрение выполняется наиболее незаметным способом, чтобы избежать срабатывания антивирусов. Если стандартный эпилог (`mov esp, ebp; pop ebp; ret`) был найден непосредственно перед пространством `0xCC`, то гаджет внедряется между предыдущим кодом и эпилогом. На рисунке 1 показан пример такого внедрения гаджета `mov ecx, eax`.

```
mov esp, ebp                    jmp epilogue; normal flow avoiding gadget
pop ebp                         mov ecx, eax; the injected gadget
ret(n)                          jmp return  ; gadget flow avoiding std. epilogue
CCCCCCCCCCCCCCCCC       epilogue:
                                mov esp, ebp
                                pop ebp
                        return:
                                ret(n)
                                CCCCCCCC
```

Figure 1: Injection of gadget (*right*) in 0xCC cave preceded by standard epilogue (*left*)

Если эпилог не найден на "границе" с пространством `0xCC`, то вводится псевдофункция со стандартным прологом и эпилогом, чтоб избежать эвристического срабатывания. Эта функция имеет следующий вид:

```
        push ebp
        mov ebp, esp
        <gadget code>
        jmp return
        mov esp, ebp
        pop ebp
return:
        ret
```

Figure 2: Pseudo-function ending used during gadget injection

Последующие внедрения гаджетов будут повторно использовать этот псевдоэпилог, точно также, как было описано выше, тем самым делая его похожим на настоящую функцию.

3.7 Перестановка исходного кода

Предопределенные взаимно-однозначные (т.е одна инструкция к одному гаджету) перестановки достигаются с помощью **IR** функций и функции кодировщика. Кодировщики также выполняют основные алгебраические перестановки на основе таких функций, как сложения, вычитания, умножения и деления. Например, если кодируемая инструкция имеет тип `ADD_IMM ( add reg, imm )`, кодировщик станет проверять все, что похоже на `add reg, x` (где `x` является целым делителем `imm, imm/x` раз). Сложение и вычитание с константами также будут переставлены местами, если знаки констант поменялись местами. *M - N* перестановок быстро масштабируются в экспоненциально возрастающее пространство и выходят за рамки данной работы.

3.8 Цепочка гаджетов

Цепочка адресов возврата может быть построена либо в рантайме, либо во время компиляции и созранена в инициализированной секции `.data` файла (чтобы затем скопировать ее в рантайме в стек). Наиболее опасным будет первый вариант (в рантайме). Мы выбрали его для оценки "коэффицента" скрытности (также выбранного в качестве варианта реализации). Во время этого процесса, помимо помещения VA в стек, компилятор **ROP** должен учитывать помещение констант, поправки на модификации указателя стека в гаджете (например избыточные `pop` , `retn`) и гаджеты с гаджетами-загрузчиками. Для этого определены следующие типы операций со стеком:
Code:

```
PUSH_VA ; push a (loader) gadget VA onto the stack
PUSH_IMM ; push an immediate constant onto the stack
ADVANCE ; advance (subtract from) the stack pointer a number of bytes
CHAIN ; pseudo operation denoting a placeholder for the next gadget’s VA
```

Результатом процесса кодирования данной инструкции данным гаджетом является череда операций со стеком для вызова гаджета. Список таких операций для всех вызово гаджета описывают инструкции ассемблера, которые при выполнении построят цепь в стеке. В другом случае, такие операции могут быть использованы для создания необходимого **stack frame** во время компиляции, сохранения его как инициализированных данных и копирования из секции `.data` во время выполнения. Есть также возможность кодировать и декодировать **stack frame**. К примеру, когда выполняется несколько вызовов одного и того же гаджета (например, как при использовании `inc eax` для получения `add eax, X`), компилятор обернет вызов циклом условного перехода с использованием свободного регистра.

Однако, не все типы команд могут быть так легко закодированы в **ROP**. В данной работе мы не рассматриваем кодирование ветвлений (переходов, вызовов, циклов, прерываний), привелигированных команд и `pop` команды. А следовательно, **возвратно-ориентированный код** должен в конце концов вернуться обратно в исходный шеллкод. Это достигается путем обертывания команд построения цепи следующим образом:

Code:

```
[1] call build_chain
[2] jmp past_the_chain
build_chain:
[3] push <VA of gadget N>
[4] ....
[5] push <VA of gadget 1>
[6] ret
past_the_chain:
[7] <other instructions / chains>
```

Таким образом, последний гаджет (N) вернется к инструкции [2], проскочив мимо построения цепочки и продолжит нормальный поток выполнения.

4. Эксперименты и результаты

Для того, чтоб "оценить" **ROPInjector**, мы использовали онлайн сервис для сканирования - VirusTotal, который на момент написания статьи включает в себя 57 антивирусов. Для PE-носителей (т.е зараженных) мы выбрали 9 популярных 32-битных исполняемых файлов различного размера, большинство из которых содержат сертификаты (см. таблицу 1):

Исполняемый файл	Размер (KB)	Версия	SHA256
AcroRd32.exe	1489	Version 10.1.12 of Adobe Acrobat Reader X	a03297789b5a784af3765c523b33b9d54578e38a178ca67103b5e0e74f905331
Acrobat.exe	321	Version 10.0.0.396 of Adobe Acrobat X Pro	281529dbd6c45cc1706d5cd66456b5c983aa5e6e3dc64723779d9b2bd48b769d
cmd.exe	296	Version 6.1.7601.17514 Windows Command Processor	17f746d82695fa9b35493b41859d39d786d32b23a9d2e00f4011dec7a02402ae
Rainmeter.exe	39	Version 2.4.0.1678 of Rainmeter	00c8f2b58ffb318cf1031f58f4fe86a73bcb9716c7072012114bd42f157dd071
firefox.exe	331	Version 35.0.0.5486 of Mozilla Firefox	11740f07a822637874da4eb4eafa309d145a1ca729779e30cb3d1e592c5484df
java.exe	172	Version 7.0.710.14 of Oracle Java	06889c037faab8379aaafb2bf9e77807e3d432da435cdab1244bff36c5c562d5
wmplayer.exe	163	Version 12.0.9600.17415 of Microsoft Windows Media Player	c7adbfeeb7993928cb542751625dac6b10e96b18fcdcd836a72cad62ae797250
nam.exe	1829	Version 1.0a11a of “The Network Animator"	5d329bb39ba744cdba5e1afe107551c18ba0acd46cb6764391024a73aa2d583f
notepad++.exe	2348	Version 6.6.9.0 of the GNU text editor for Windows	a11077cb6c209c67eb2d507d650fbee0925f3cbe860c70e0cd779b73f5af4b80

Что касается исходного шеллкода, то мы выбрали две наиболее популярные нагрузки из Metasploit:
- *Reverse TCP Shell*
- *Reverse TCP Meterpreter*

Для каждого PE и каждого шеллкода мы выполнили 4 "сценария" патча, как указано в таблице 2, в результате чего получили в общей сложности 72 образца.

Название сценария	Описание
Original	Исполняемый файл, не был пропатчен
ROP-Exit	Это исполняемый файл, созданный ROPInjector. Исполняемый файл пропатчен с развернутым шеллкодом, преобразованным в ROP и точкой входа перед выходом исходной программы (хук ExitProcess или exit)
Exit	В этом сценарии исполняемый файл патчится с нетронутым шеллкодом и точкой входа до выхода исходной программы (хук ExitProcess или выход)
Shellcode	Исполняемый файл пропатчен с нетронутым шеллкодом и точкой входа перед исходной программой

На рисунках 3 и 4 показаны "коэффиценты" скрытности (1 - количество детектов/количество антивирусов) **ROPInjector** для каждого из 4 сценариев для *reverse shell* и *reverse meterpreter payloads* соответственно. Мы также можем заметить, что исполняемые файлы **ROPInjector** (сценарии "**ROP-Exit**") достигают самого высокого "коэффицента" скрытности. В частности, в более чем половине случаев тестирования **ROPInjector** приводит к 100% сокрытию от антивирусовв, в то время как в некоторых PE файлах (например, `java.exe`), **ROPInjector** имеет "коэффициент" сокрытия более 98,5% как для обратного шелла, так и для *meterpreter shell-кодов*. Это означает, что в среднем **ROPInjector** достигает сокрытия от антивирусов, равного 99,31%, как показано на рисунке 5 (сценарий "**ROP-Exit**").

Figure 3: Evasion ratio of ROPInjector for the reverse shell payload

Figure 4: Evasion ratio for the reverse meterpreter payload

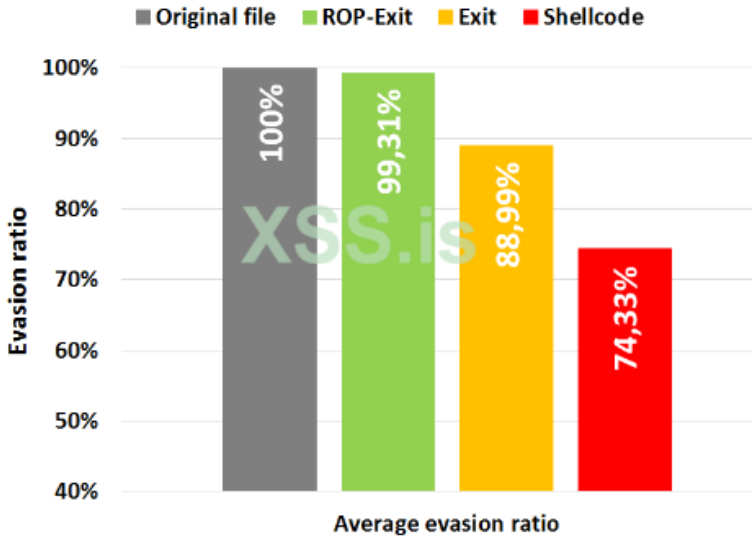

Figure 5: Average evasion ratio per combination of methods

Из этих результатов мы сможем сделать вывод, что уклонение почти в равной степени зависит как от обфускации/трансформации кода, так и от EP (следовательно от поведения). Это может быть объяснено тем фактом, что некоторые антивирусы смогли обнаружить **ROPInjector**, несмотря на отсутствие сигнатуры благодаря полиморфизму **ROP**. Кажется, что поведенческий анализ не менее важен, чем статические сигнатуры для некоторых антивирусов (т.е которые смогли обнаружить ROPInjector) и он в основном выполняется при входе в исполняемые файлы.

Кроме того, на рисунке 6 приведено сравнение с **Shellter v2.2** и **PEInject**. **Shellter** использовался с опциями по умолчанию (т.е с полиморфным мусорным кодом). Мы можем заметить, что исполняемые файлы, сгенерированные **ROPInjector** (т.е "**ROP-Exit**") имеют самый высокий "коэффицент" уклонения во всех проведенных экспериментах, по сравнению с **Shellter** и **PEinject**. Обратите внимание, что даже простой сценарий "**Exit**" достиг в некоторых исполняемых файлах лучших результатов по сравнению с **Shellter**. Наконец, **PEInject** имеет наихудший "коэффициент" скрытности.

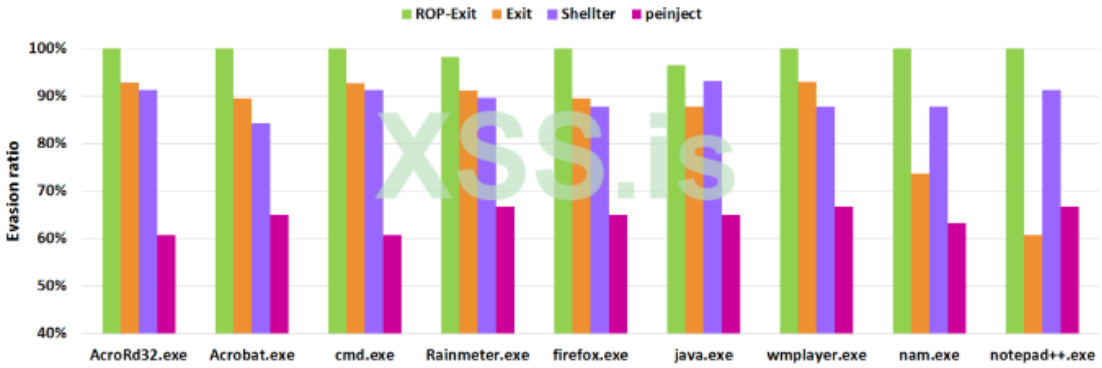

Figure 6: Comparison of evasion ratio between "ROP-Exit", "Exit" scenarios with Shellter and PEinject

Также стоит отметить, что помимо VirusTotal, мы также протестировали эффективность **ROPInjector** против специального инструмента под названием "**Experimental Windows .text section Patch Detector**" от *NCCGroup*. Он сравнивает исполняемые разделы памяти с теми, что находятся на диске, для того, чтоб обнаружить модификации/патчи. Как и ожидалось, ни один исполняемый файл не был обнаружен как исправленный, так как **ROPInjector** не изменяет секцию `.text` в памяти (и не требует этого).

5. Заключение

Большинство антивирусов полагаются на *строковые сигнатуры* и *"мягкие" поведенческие профилирующие механизмы обнаружения* (*mild behavioral profiling detection mechanisms*). Путем кодирования шеллкода в его **ROP** эквивалент, и даже путем выполнения элементарных мутаций (развертывания), первые могут быть байпассныты в подавляющем большинстве случаев. *Поведенческого профилирования* также можно избежать, перехватывая нормальный поток выполнения в тех точках, где антивирусы либо не могут ничего эмулировать, либо не могут просто получить достаточно "доказательств", чтобы классифицировать поведение как угрожающее. В данной работе мы представили в качестве средств перехват общих вызово "процесса" выхода, приводящий во многих случаях к стопроцентному скорытию, а вдругих - более чем 98%.

Все вытекающие представленных методов можно подкорректировать, если подойти к ним индивидуально. Например, сигнатуры могут быть поставлены (или созданы) для инструкций построения **ROP** (хотя ожидается, что они вряд ли будут достаточно эффективными), а поведенческий анализ может быть также выполнен с точки зрения жизненного цикла процесса. Однако, поскольку незначительные вариации и рандомизация могут снова обезвредить сканеры, более надежная контрмера не кажется простой в разработке, практичной в реализации или реалистичной в принципе. Возможно, наиболее перспективным является строгое "соединение" ОС хоста с доверенными сертификатами (или контрольными суммами) ПО и политикой "недоверия всем по умолчанию", очень похоже на то, что было начато компанией Microsoft еще в 2001 году, но не получило развития.

==========
Оригинал: https://www.blackhat.com/docs/us-15...For-Polymorphism-And-Antivirus-Evasion-wp.pdf
Авторы: Giorgos Poulios, Christoforos Ntantogian, Christos Xenakis
Репозиторий: https://github.com/gpoulios/ROPInjector

Автор перевода: atavism
Memphis 12.5.1