УДК: 004.056

Виктор В. Ерохин
Московский государственный институт (университет) международных отношений Министерства иностранных дел Российской Федерации,
пр-т Вернадского, 76, Москва, 119454, Россия
Московский государственный технический университет имени Н.Э. Баумана,
ул. 2-я Бауманская, 5, стр. 1, Москва, 105005, Россия
e-mail: erohinvv@mail.ru, https://orcid.org/0000-0002-8754-0012

ПОИСК ВРЕДОНОСНЫХ СЦЕНАРИЕВ POWERSHELL С ИСПОЛЬЗОВАНИЕМ СИНТАКСИЧЕСКИХ ДЕРЕВЬЕВ

DOI: http://dx.doi.org/10.26583/bit.2023.3.05

Аннотация. Цель статьи: поиск более абстрактного представления функциональности сценария PowerShell с использованием абстрактных синтаксических деревьев, чтобы можно было обнаружить невидимый обфусцированный сценарий PowerShell, если связанный сценарий PowerShell уже является известным вредоносным программным обеспечением. Метод исследования: анализ обфускации сценариев PowerShell осуществляется на трех типах обфускации: токенов, строк и абстрактного синтаксического дерева. Полученный результат; определили, что простые функции на основе AST программной среды PowerShell, такие как количество функций AST и их распределенная глубина, а также параметр косинусного расстояния обфускации сходства AST, рассчитанные на основе типов функций и их расположения в AST, вполне достаточны для атрибуции обфусцированных сценариев PowerShell их исходному сценарию, не подверженному обфускации; описан и реализован метод создания расширенного набора данных обфусцированного PowerShell, включая маркировку исходных файлов; предоставлен обширный анализ набора данных и несколько функций для представления структуры PowerShell.

Ключевые слова: вредоносное программное обеспечение, обфускация, сценарий PowerShell, программирование, исходный код, моделирование, операционная система.
Для цитирования: ЕРОХИН Виктор В. ПОИСК ВРЕДОНОСНЫХ СЦЕНАРИЕВ POWERSHELL С ИСПОЛЬЗОВАНИЕМ СИНТАКСИЧЕСКИХ ДЕРЕВЬЕВ. Безопасность информационных технологий, [S.l.], т. 30, № 3, с. 77–89, 2023. ISSN 2074-7136. URL: https://bit.spels.ru/index.php/bit/article/view/1531. DOI: http://dx.doi.org/10.26583/bit.2023.3.05.

Viktor V. Erokhin
Moscow State Institute of International Relations (University) of the Ministry of Foreign Affairs,
76 Vernadsky Ave., Moscow, 119454, Russia
Bauman Moscow State Technical University
2nd Baumanskaya Str., 5/1, Moscow, 105005, Russia
e-mail: erohinvv@mail.ru, https://orcid.org/0000-0002-8754-0012

Search for malicious powershell scripts using syntax trees

DOI: http://dx.doi.org/10.26583/bit.2023.3.05

Abstract. Purpose of the paper: a search for a rather abstract representation of the PowerShell script functionality using abstract syntax trees such that an invisible obfuscated PowerShell script can be detected provided the associated PowerShell script is already known malware. Research method: PowerShell script obfuscation analysis is performed on three types of obfuscation: token, string, and abstract syntax tree. The obtained result: 1) we have found that simple PowerShell AST-based features, such as the number of AST functions and their distributed depth, as well as the AST similarity obfuscation distance parameter calculated from the types of functions and their location in the AST are sufficient to attribute obfuscated PowerShell scripts to their original script, not subject to obfuscation; 2) a method for creating an extended data set of obfuscated PowerShell is described and implemented including marking source files; 3) an extensive analysis of the data set and several functions are provided to represent the PowerShell structure.

Keywords: malware, obfuscation, PowerShell script, programming, source code, modeling, operating system.
For citation: EROKHIN Viktor V. Finding malicious powershell scripts using syntax trees. IT Security (Russia), [S.l.], v. 30, n. 3, p. 77–89, 2023. ISSN 2074-7136. URL: https://bit.spels.ru/index.php/bit/article/view/1531. DOI: http://dx.doi.org/10.26583/bit.2023.3.05.

Введение

Безопасность и обнаружение вредоносного программного обеспечения (ПО) – серьезная проблема, с которой сталкиваются как научные исследователи, так и пользователи. Злоумышленники создают вредоносное программное обеспечение и скрипты, которые используют уязвимости в операционной системе или других программах для получения доступа к конфиденциальным данным и критическим системным функциям. Они могут украсть данные, потребовать выкуп или вызвать сбой в работе компьютерной системы, что повлияет на удобство использования компьютера. Это особенно важно, когда целевое ПО широко распространено или, когда атака происходит по большой компьютерной сети, что позволяет ей быстро распространяться. Защитники ищут способы обнаружить и остановить эти атаки до того, как они начнутся. Это можно сделать путем статического анализа программы с использованием сигнатур известных атак или с помощью моделей машинного обучения.

PowerShell стал популярным языком сценариев благодаря своей гибкости и доступу к службам операционной системы, таким как файловая система и разделы реестра [1], поставляется с предустановленной Windows и в последние годы, благодаря своей кроссплатформенной версии, приобрел популярность в других операционных системах. Мощность PowerShell и широкое использование делают его отличным инструментом и мишенью для вредоносных атак. Злоумышленники также могут скрыть свой программный код в сценариях PowerShell, чтобы его труднее было обнаружить. Злоумышленники могут запускать свое вредоносное ПО в PowerShell без установки его на компьютер и не оставлять следов своей деятельности [2]. Программные команды преобразуются в кодировку, не удобочитаемую человеком, например, в двоичный код. Это предотвращает обнаружение известных вредоносных сценариев при визуальном осмотре программного кода. Традиционные антивирусные программные средства вычисляют известную сигнатуру вредоносного ПО, которая используется для обнаружения этой атаки. На основе статического анализа исходного файла формируется сигнатура, которая достаточно часто не может быть использована для обнаружения похожих скриптов с той же функцией. Злоумышленник может использовать обфускацию, чтобы взломать подпись. Обфускация – это манипулирование сценарием или фрагментом кода, которое изменяет сигнатуру кода без изменения его функции. Это означает, что антивирусные программные средства должны не только идентифицировать существующую атаку, а затем создать сигнатуру для ее обнаружения, но и делать это для всех слегка модифицированных или запутанных атак. Для обхода традиционного обнаружения вредоносных программ в PowerShell в основном используются бесплатные инструменты запутывания, которые очень легко и эффективно изменяют сценарии PowerShell [3, 4].

Тот факт, что обфускация распространена в атаках вредоносных программ, может указывать на то, что простое обнаружение обфускации является жизнеспособным методом обнаружения вредоносных программ. Однако PowerShell можно запутать как по не злонамеренным, так и по злонамеренным причинам. Компания или программист могут защитить интеллектуальную собственность, скрыв исходную программу, чтобы другой программист не смог ее легко понять и скопировать. В этом случае, антивирусное программное средство обнаружит обфускацию и отклонит как безопасные, так и полезные

скрипты. При этом цель запутывания вредоносного PowerShell состоит в том, чтобы создать достаточную разницу между запутанным файлом и оригиналом. В этом случае запутанный файл не может быть обнаружен как вредоносный. Часто к скрипту злоумышленники применяют различные виды обфускации, чтобы увеличить это расстояние. Обфусцированный скрипт, нарушающий обнаружение сигнатур вредоносных программ, называется состязательным примером злоумышленника, который находит слепую зону в модели обнаружения вредоносных программ Powershell и использует его для обхода детектора. Для антивирусных программных средств можно найти такую слепую зону. Для этого необходимо определить максимальное расстояние, на котором вредоносный PowerShell может избежать обнаружения [5, 6]. Если слепая зона найдена, необходимо дополнить вредоносный набор данных антивирусных программных средств автоматически сгенерированными враждебными примерами. Например, это позволит детектировать вредоносный сценарий PowerShell и предотвратить его запуск.

1. Постановка задачи исследования

Для решения проблемы обнаружения вредоносных программ PowerShell в основном используют методы машинного обучения, преимущество использования которых заключается в том, что с их помощью можно научить распознавать структуру и функции программы. Это означает, что нам не нужно вручную подписывать новые вредоносные программы или даже заранее определять новые атаки для их обнаружения. Современные научные исследования машинного обучения для обнаружения вредоносных PowerShell включают методы NLP (Natural Language Processing) и сверточных нейронных сетей, а также структурный анализ программ на основе абстрактного синтаксического дерева (Abstract Syntax Tree, AST) [7–9]. В данной статье приводятся результаты исследований метода AST и определение расстояний обфускации PowerShell.

В [3] показано, что для точной классификации семейств вредоносных программ можно использовать простые функции PowerShell AST. AST – это представления функции PowerShell, которые описывают структуру программы (PowerShell SDK 6.2.0 или 7.0.0 для Windows PowerShell 5.1.0.0):

1. ScriptBlockAst.
  1.1. NamedBlockAst.
    1.1.1. AssignmentStatementAst.
      1.1.1.1. VariableExpressionAst.
      1.1.1.2. CommandExpressionAst.
        1.1.1.2.1. StringConstantExpressionAst.
        1.1.1.2.2. ArrayLiteralAst.
        1.1.1.2.3. ConstantExpressionAst.
      1.1.1.3. PipelineAst.
        1.1.1.3.1. CommandAst.
      1.1.1.4. ConvertExpressionAst.
        1.1.1.4.1. TypeConstraintAst.
        1.1.1.4.2. VariableExpressionAst.
    1.1.2. PipelineAst.
      1.1.2.1. CommandExpressionAst.
        1.1.2.1.1. TypeExpressionAst.
    1.1.3. IfStatementAst.
      1.1.3.1. StatementBlockAst.
      1.1.3.2. PipelineAst.

1.1.3.2.1. CommandExpressionAst.
1.1.4. ForStatementAst.
1.1.4.1. AssignmentStatementAst.
1.1.4.2. PipelineAst.
1.1.4.2.1. CommandExpressionAst.
1.1.4.3. StatementBlockAst.
1.1.4.3.1. PipelineAst.

Узлы AST представляют собой функциональные блоки в программе, а ветки AST представляют поток управления программой (операторы if и циклы for). Включение дополнительной информации о структуре AST в качестве функций может раскрыть больше функциональных возможностей сценария PowerShell. Кроме того, функции могут быть применены и к другим задачам, таким как связывание обфускаций с исходными сценариями.

Основная цель данного исследования – это поиск представлений функции PowerShell, независимой от различных методов запутывания и манипуляций с кодом.

Для реализации этой цели необходимо знать, как выглядят сценарии PowerShell с запутыванием и без него. Сценарии можно исследовать визуально и с помощью статической проверки, которая исследует структуру скрипта, а также его символы, а значит позволяет многое узнать о функции скрипта. В этом случае визуально читается и сравнивается PowerShell в наборе данных с обфускациями, что позволит получить количественные показатели сходства и различия между AST-представлениями исходного набора данных и с обфускациями.

В следующих разделах рассмотрим: наиболее используемые инструменты для обфускации PowerShell и методы обнаружения вредоносного PowerShell; набор данных и методы, которые будут использованы для изучения влияния различных методов обфускации на сценарии PowerShell; наиболее полезные функции, полученные в результате проведенного исследования, и их способность моделировать функциональность сценариев PowerShell; точность разработанных моделей и задачи дальнейших исследований, необходимые для их улучшения.

2. Инструменты обфускации PowerShell

Обфускацию PowerShell можно разделить на несколько категорий: обфускация на токенах, на основе AST, на основе кодирования, сжатия и средств запуска, а также обфускация базовой станции и конкатенации строк (CONCATENATE STRING). Для исследований используются наиболее распространенные инструменты обфускации: Invoke-Obfuscation, Invoke-CradleCrafter и ISEsteroids.

Обфускация на основе токенов или символов изменяет символы в сценарии и часто применяется только к одному типу токенов за раз, например, к именам переменных или команд:

1. Фрагмент кода оригинального сценария PowerShell:

```
Funtion Get_Resource
{
   [CmdletBinding()]
   [OutputType([System.Collection.Hashtable])]
   param
   (
      [parameter(Mandatory = $true)]
      [System.String]
      $Source,
```

```
    [parameter(Mandatory = $true)]
    [System.String]
    $Destination,
    [System.String]
    $Files,
```

2. Фрагмент кода обфусцированного сценария PowerShell на основе токенов:

```
Funtion get_REsourCe
{
  [CmdletBinding()]
  [OutputType([System.Collection.Hashtable])]
  param
  (
    [parameter(ManDatory = ${tru'E})]
    [System.String]
    ${So'Urce},
    [parameter(MandatOrY = ${tr'Ue})]
    [System.String]
    ${DeStin'Ation},
    [System.String]
    ${FI'Le'S},
```

3. Обфускация на основе AST манипулирует структурой сценария, перестраивая заменяемые элементы, такие как назначения переменных или параллельные вызовы методов. Фрагмент кода обфусцированного сценария PowerShell на основе AST:

```
Funtion Get_Resource
{
  [CmdletBinding()]
  [OutputType([System.Collection.Hashtable])]
  param
  (
    [Bool]
    $MultiThreaded=$False,
    [System.String]
    $ExcludeFiles,
    [parameter(Mandatory = $true)]
    [System.String]
    $Source,
    [System.UInt32]
    $Wait,
    [System.String]
    $Files,
```

Сценарии также можно запутать, например, закодировав их как шестнадцатеричные или двоичные строки, или сжать в одну командную строку. Наконец, методы запуска (run) и подставки изменяют способ выполнения команды. Обфускация средства запуска создает отдельную команду или исполняемый файл для запуска из другого инструмента, например, Python 3.x, или собственной командной строки. Обфускация источника создает новую команду, которая загружает исходную команду из другого источника. Оба типа обфускации позволяют вызывать и выполнять команду без ведома пользователя, не оставляя следов на локальном компьютере.

Обфускация конкатенации строк воспроизводится при применении точки между каждой строкой, которая указывает на языке PHP соединить эти строки символов вместе и запустить их как единую функцию – например, 'cr'.'ea'.'te'.'_f'.'un '.'c'.'ti'.'o'.'n'; станет create_function.

Invoke-Obfuscation [10] – инструмент обфускации с открытым исходным кодом, включающий следующие категории обфускации:

- токенная обфускация, которая использует манипуляции со строками, такие как случайный регистр, конкатенация, изменение порядка с помощью методов форматирования и добавление делений в строку для следующих типов токенов: command, member, argument, string, variable, comment, type, whitespace;
- AST обфускация, которая переупорядочивает следующие типы узлов AST: CommandAst, ParamBlockAst, NamedAttributeArgumentAst, ScriptBlockAst, BinaryExpressionAst, AttributeAst, HashtableAst, TypeConstraintAst, TypeExpressionAst, AssignmentStatementAst;
- кодировочная обфускация, которая кодирует всю команду как: ASCII, secure string (AES), hex, octal, binary, bxor, whitespace, special characters;
- обфускация сжатия, которая преобразовывает команду в однострочную команду и сжимает её до base64;
- обфускация запуска, которая конвертирует команду в исполняемый файл следующих типов: CMD, PS, RUNDLL, WMIC, VAR+, CLIP+, STDIN+, VAR++, CLIP++, STDIN++, MSHTA++, RUNDLL++.

Invoke-CradleCrafter [2] – это расширенная версия Invoke-Obfuscation, которая применяется для обфускации сценариев PowerShell в виде удаленно загружаемых исполняемых файлов. Сценарии загружаются в оперативную память или на носитель (HDD, SSD и т.д.) компьютера на нескольких языках и в нескольких формах:

- обфускация памяти, которая загружается в память как PS, CERTUTIL, .NET, в форматах: STRING (команда в виде одной строки), WEBREQUEST (читаемый поток, байтовый массив или структурированные данные), COMOBJECTS (объекты, которые загружаются при взаимодействии с программой Windows), CSHARP (встроенный объект, скомпилированный на языке C#)
- обфускация носителя информации, которая загружается на носитель через локальную программу SYSTEM, CERTUTIL, BITS, BITSADMIN.

ISESteroids применяет обфускацию символов, чисел или двоичных кодов к параметрам, переменным, функциям и строкам. Он удаляет комментарии и пустые строки и применяет обфускацию уникального идентификатора к идентификатору сценария PowerShell. Программа ISESteroids включает в себя графический инструмент для создания обфускации, что затрудняет автоматический запуск отдельных команд [11].

3. Методы деобфускации PowerShell

Для деобфускации применяются методы глубокого обучения искусственных нейронных сетей и программных средств PowerDrive и PowerDecode.

1. Методы глубокого обучения.

Hendler D. и др. [4] представили метод обнаружения вредоносного PowerShell с использованием компьютерного зрения и методов нейролингвистического программирования. Методы компьютерного зрения кодируют первые 1024 символа команды Powershell в виде матрицы. В этой матрице каждая строка является одним горячим вектором с нулевыми элементами, за исключением кода символа в этом индексе, и применяют к матрицам сверточных нейронных сетей. Методы нейролингвистического

программирования кодируют команду в вектор длиной 1024, который содержит код для каждого из первых 1024 символов, и передают этот вектор в рекуррентную нейронную сеть. Эти методы сосредоточены на символьном содержании сценария, которым можно легко манипулировать с помощью обфускации.

В [3, 4] описывается программная платформа FireEye, построенная на основе нейролингвистического программирования и предназначенная для интерпретации сценария PowerShell с помощью групповых команд. FireEye первоначально декодирует и токенизирует сценарий, а затем привязывает токены к их семантической основе. Этап декодирования позволяет FireEye обрабатывать удаленную загрузку и исполняемое вредоносное ПО, созданное методами: обфускацией базовой станции и обфускацией запуска. Затем декодированные токены используются для создания вектора признаков для сценария. Далее вектор признаков классифицируется с помощью контролируемого алгоритма, такого как Kernel SVM (Support Vector Machine).

Rusak G. и др. [3] показали, что семейства вредоносных PowerShell можно точно классифицировать с помощью анализа AST по количеству узлов и глубине. Они предложили методику для изучения векторизованных представлений AST для более надежной классификации, используя меру расстояния для поддеревьев AST с применением подобия между различными типами узлов. Это также позволяет найти взаимосвязь между обфускациями сценариев Powershell с их исходным родительским сценарием.

Mou LL. и др. [7–9] рассматривают критерий кодирования, основанный на представлениях AST программ для использования в глубоком обучении. Критерий применяется для создания векторного представления программы, которое можно использовать в алгоритме глубокого обучения. Поскольку конечное число типов узлов позволяет использовать машинное обучение, то критерий использует детализацию узлов AST. Один нейронный слой кодирует каждый узел AST, используя представления его дочерних элементов. Это позволяет изучить векторное представление программы по ее структуре.

2. Деобфускация с использованием программных средств PowerDrive и PowerDecode.

В [12, 13] представлены исследования по применению деобфускации к PowerShell перед анализом обнаружения вредоносных программ с использованием программного средства PowerDrive. PowerDrive реализует процесс, который требует перед выполнением послойной деобфускации проанализировать сценарий Powershell на наличие уровней обфускации. Перед деобфускацией сценарий Powershell очищается от синтаксических ошибок или команд отладки. Деобфускация в PowerDrive использует регулярное выражение для поиска и удаления распространенных шаблонов обфускации, например, конкатенации при обфускации строк. Это подобно запуску закодированного сценария Powershell с использованием Invoke-Expression и перехват декодированного текста сценария Powershell во время его выполнения. После деобфускации части сценария запускается анализируемый сценарий для того, чтобы убедиться, что не было допущено ошибок при деобфускации, способных повредить выполнению сценария Powershell. Это не всегда рациональный подход при работе с вредоносными программами, т.к. запуск деобфусцированного сценария Powershell может заразить систему вирусом. Однако, запустив сценарий в песочнице виртуальной машины, можно решить эту проблему. Это требует дополнительных затрат времени и программного обеспечения на настройку виртуальной машины и частично сводит на нет цель обнаружения вредоносных программ, т.к. сценарии всё же должны быть запущены.

В исследованиях Университета Кальяри [13, 14] для противодействия вредоносным сценариям PowerShell были разработаны эффективные методы, позволяющие

деобфусцировать эти сценарии при использовании программного средства PowerDecode [15], который способен получить оригинал кода сценария PowerShell из обфусцированного кода.

4. Набор данных для эксперимента

Используем корпус PowerShell от Palo Alto Networks [16], размером 3,76 ГБ с 412075 сценариями PowerShell с неизвестным риском и набором из 4079 известных вредоносных сценариев PowerShell. Этот корпус расширяем посредством обфускации сценариев с помощью инструментов, указанных в п. 1. Автоматизированный процесс обфускации сценариев следующий: сначала используем один тип обфускации, а затем последовательно проводим обфускацию выходных сценариев. Это имитирует методы, используемые злоумышленниками для обхода вредоносных детекторов PowerShell, и позволяет проанализировать все возможные комбинации типов обфускации. Предполагаем, что эффект обфускации одинаков для всех файлов с различным размером. Дополненные данные хранятся удаленно, т.к. с каждой итерацией применяемой обфускации количество обфускаций растет экспоненциально. Применяем каждую обфускацию только один раз, а затем снова обфусцируем каждый из этих файлов, за исключением обфускаций запуска, кодирования и сжатия. Они исключены, т.к. должны быть деобфусцированы перед извлечением AST. В эксперименте используется 33 типа обфускации, каждый из которых применяется один раз к исходным файлам и еще раз к каждому файлу, подверженному обфускацией. Всего обфусцируем десять исходных файлов из корпуса PowerShell от Palo Alto Networks [16] и создаем 1122 обфускации, 33 однотипных обфускации и 1089 цепочек обфускаций для каждого из десяти исходных файлов. Общий набор данных составляет 11220 файлов, а значит использование десяти исходных файлов из корпуса PowerShell от Palo Alto Networks [16] вполне достаточно для анализа рассматриваемых методов обфускации.

Применяемые 33 метода обфускации представлены в табл. 1.

5. Методика проведения эксперимента

Автоматическую обфускацию реализуем в Python 3.8, используя Invoke-Obfuscation.

Первоначально выбираются десять случайных файлов из корпуса PowerShell от Palo Alto Networks [16]. Далее используется каждая обфускация при активации процесса Python для запуска сценария PowerShell, который перебирает все доступные команды запутывания в Invoke-Obfuscation. Далее производим запись обфусцированной команды в новый файл, имя которого состоит из исходного имени файла и метода применяемой обфускации. Это позволяет легко сравнивать одни и те же методы обфускации файлов, а также для одного и того же файла разные

Анализ обфускации сценариев PowerShell осуществляется на трех типах обфускации: токенов, строк и абстрактного синтаксического дерева. Все категории включают в себя 33 метода обфускации (см. п. 3), которые относятся к конкретному замаскированному элементу в сценарии PowerShell. Обфускации запуска, кодирования и сжатия исключены, т.к. они преобразуют команды в формат, который необходимо декодировать в исходный сценарий, прежде чем можно будет извлечь его AST.

К каждому исходному файлу применяем последовательно 33 метода обфускации. Получаем файлы с несколькими типами запутывания. Это позволяет в нашем исследовании понять: уменьшается ли влияние обфускации, увеличивается или остается постоянным обфускация с каждой итерацией обфускации 33-мя методами.методы обфускации.

Таблица. Методы обфускации, их типы и экспериментальные расстояния AST

Метод обфускации	Тип обфусцирования	Косинусное расстояние между распределением типов функций AST (п. 6. Эксперимент)
CONCATENATE	Токенная обфускация для типа STRING	AST ≈ 1,0
REORDER	Токенная обфускация для типа STRING	AST ≈ 1,0
TICK	Токенная обфускация для типа COMMAND	AST ≈ 1,0
CONCATENATE	Токенная обфускация для типа COMMAND	AST ≈ 0,9
REORDER	Токенная обфускация для типа COMMAND	AST ≈ 0,85
RANDOM CASE	Токенная обфускация для типа ARGUMENT	AST ≈ 1,0
TICK	Токенная обфускация для типа ARGUMENT	AST ≈ 1,0
CONCATENATE	токенная обфускация для типа ARGUMENT	AST ≈ 0,32
REORDER	Токенная обфускация для типа ARGUMENT	AST ≈ 0,318
RANDOM CASE	Токенная обфускация для типа MEMBER	AST ≈ 1,0
TICK	Токенная обфускация для типа MEMBER	AST ≈1,0
CONCATENATE	Токенная обфускация для типа MEMBER	AST ≈1,0
REORDER	Токенная обфускация для типа MEMBER	AST ≈1,0
RANDOM CASE	Токенная обфускация для типа VARIABLE	AST ≈ 1,0
CONCATENATE	Токенная обфускация для типа TYPE	AST ≈ 1,0
REORDER	Токенная обфускация для типа TYPE	AST ≈ 1,0
REMOVE	Токенная обфускация для типа COMMENT	AST ≈ 1,0
RANDOM	Токенная обфускация для типа WHITESPACE	AST ≈ 1,0
ALL	Токенная обфускация для всех представленных типов	AST ≈ 0,43
REORDER	Обфускация AST для функции NamedAttributeArgumentAst	AST ≈ 1,0
REORDER	Обфускация AST для функции ParamBlockAst	AST ≈ 1,0
REORDER	Обфускация AST для функции ScriptBlockAst	AST ≈ 1,0
REORDER	Обфускация AST для функции AttributeAst	AST ≈ 1,0
REORDER	Обфускация AST для функции BinaryExpressionAst	AST ≈ 1,0
REORDER	Обфускация AST для функции HashtableAst	AST ≈ 1,0
REORDER	Обфускация AST для функции CommandAst	AST ≈ 1,0
REORDER	Обфускация AST для функции AssignmentStatementAst	AST ≈ 1,0
REORDER	Обфускация AST для функции TypeExpressionAst	AST ≈ 1,0
REORDER	обфускация AST для функции TypeConstraintAst	AST ≈ 1,0
REORDER	Обфускация AST для всех представленных функций	AST ≈ 1,0
CONCATENATE STRING	Обфускация строк посредством их конкатенации	AST ≈ 0,35
REORDER STRING	Обфускация команды для PowerShell, используя свойства строки REORDER, т.е. изменения их порядка расположения	AST ≈ 0,59
REVERSE STRING	Обфускация команды для PowerShell, используя свойства строки REVERSE, т.е. изменение строки команды через её реверс	AST ≈ 0,58

При создании каждой обфускации извлекается её AST. В AST записывается последовательность функций (узлов, классов) управления программой: назначение переменной или вызов функции. PowerShell располагает ограниченным количеством типов

функций AST, что делает анализ на основе AST более удобным. Для извлечения AST используется скрипт Python, аналогично источника [3] с определенными совершенствованиями. Этот скрипт Python анализирует сценарий PowerShell и записывает в текстовый файл все типы функций вместе с их родителем и строками, которые она охватывает.

Анализ обфускации программного кода файлов проводим визуально и с использованием рекуррентной искусственной нейронной сети (РНС). Однако на данный момент исследования ни одна архитектура РНС с её различными гиперпараметрами не преодолела точность распознавания обфускации более 72,3%. И анализ экспериментальных данных проводится только визуально. Визуальный метод анализа хотя и не производителен, но очень точен при участии в анализе квалифицированных экспертов. Ожидается, что можно легко увидеть изменения для обфускации строк, т.к. этот тип обфускации меняет читабельность кода. Время анализа набора данных из 11220 файлов составило при использовании РНС – 12 минут, при применении визуального метода – 452 часа. Для анализа обфускации AST визуально проверяются сгенерированные файлы AST относительно расположения или изменения типа функции AST. Предполагается, что длина файла AST сильно не меняется, т.к. обфускации только изменяют строки или изменяют порядок функций AST. Визуальный метод наблюдения даст информацию для последующего расчёта метрики параметра расстояния обфускации.

На основе обфусцированных сценариев PowerShell вычисляем параметр расстояния обфускации с использованием библиотеки SciPy (scipy.spatial.distance.cosine) для Python. Применяя методы статического анализа, вычисляем косинусное расстояние между распределением символов в исходном и запутанном файлах [17, 18]. Затем рассчитывается косинусное расстояние между распределением типов функций AST в обоих файлах.

После того, как проведена обфускация сценариев PowerShell, осуществляется их тестирование, чтобы проверить их исходную функциональность. Для обеспечения безопасности компьютерной системы сценарии запускаются в виртуальной машине VirtualBox, на которой установлена операционная система Windows 10. При этом компьютер физически отключается от локальной и глобальной сети и ограничения на объём потребляемой оперативной памяти не используются.

6. Эксперимент

Эксперимент проводился с целью получения количественных параметров влияния обфускации на сценарий PowerShell. С этой целью сравниваются сценарии PowerShell, подверженные обфускациям, с их исходными сценариями, используя несколько мер расстояния, и отдельно сравниваем необфусцированные сценарии друг с другом, используя те же меры.

При эксперименте получены метрики косинусного расстояния для распределения AST сценариев PowerShell для методов обфускации, представленные в третьем столбце табл. 1, дополнительно к которым рассмотрены:

- кодировочная обфускация ASCII, расстояние обфускации сходства AST ≈ 0,42.
- кодировочная обфускация HEX, расстояние обфускации сходства AST ≈ 0,41.
- кодировочная обфускация OCTAL, расстояние обфускации сходства AST ≈ 0,48.
- кодировочная обфускация BINARY, расстояние обфускации сходства AST ≈ 0,23.
- кодировочная обфускация SECURE STRING (AES), расстояние обфускации сходства AST ≈ 0,77.
- кодировочная обфускация BXOR, расстояние обфускации сходства AST ≈ 0,18.

– кодировочная обфускация SPECIAL CHARACTERS, расстояние обфускации сходства AST ≈ 0,11.

– кодировочная обфускация WHITESPACE, расстояние обфускации сходства AST ≈ 0,64.

– обфускация сжатия, расстояние обфускации сходства AST ≈ 0,637.

Анализ результатов показал, что косинусное расстояние, основанное на количестве функций AST, относительно стабильно для всех методов обфускации. Однако на метрику косинусного расстояния обфускации, основанную на AST, в большей степени влияет обфускация на основе символов и строк. Эти случаи имеют место для усеченного сценария PowerShell или обфускации, которая изменила функцию кода. Кроме того, кодировочная обфускация и обфускация сжатия имеют низкое сходство. Это определяется тем, что эти обфускации делают восстановление AST невозможным без предварительной деобфускации сценария PowerShell.

Также проведены эксперименты по исследованию разницы в распределении расстояний между файлами, которые были обфусцированы один раз, и файлами, которые обфусцировались дважды. Это позволило определить, как множественные обфускации влияют на показатели расстояния. Во всех случаях значения косинусных расстояний AST немного выше при двойном запутывании, чем при одиночном запутывании.

Заключение

Проведенные эксперименты позволяют визуально идентифицировать различные типы обфускации и определить метрики расстояния обфускации как на основе символов и строк в файле, так и на основе структуры кода сценария PowerShell. Метрики обфускации сходства AST позволят различать файлы, подверженные обфускации и без обфускации. Порядок распределения функций AST важен для создания полезного представления сценария PowerShell. Обнаружено, что характеристики всего дерева функций AST, такие как количество функций и глубина, сильно влияют на точность распознавания обфускации сценария PowerShell.

Данное исследование может быть использовано при разработке эффективных детекторов вредоносных программ на основе расширенных наборов данных известных вредоносных сценариев PowerShell.

СПИСОК ЛИТЕРАТУРЫ:

1. JuanPablo Jofre et al. 2022. PowerShell Scripting. Microsoft Docs. URL: https://docs.microsoft.com/ru-ru/powershell/scripting/overview?view=powershell-7.2 (дата обращения: 20.06.2023).
2. Bohannon D. 2018. Invoke-CradleCrafter v1.1. URL: https://github.com/danielbohannon/Invoke-CradleCrafter (дата обращения: 20.06.2023).
3. Rusak G., Al-Dujaili A., O'Reilly UM. 2018. AST-Based Deep Learning for Detecting Malicious PowerShell. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Association for Computing Machinery, New York, NY, USA, p. 2276–2278. DOI: http://dx.doi.org/10.1145/3243734.3278496.
4. Hendler D., Kels S., Rubin A. 2018. Detecting Malicious PowerShell Commands using Deep Neural Networks. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18). Association for Computing Machinery, New York, NY, USA, p. 187–197. DOI: http://dx.doi.org/10.1145/3196494.3196511.
5. Al-Dujaili A., Huang A., Hemberg E., O'reilly UM. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA. 2018, p. 76–82. DOI: http://dx.doi.org/10.1109/SPW.2018.00020.
6. O'Reilly UM., Toutou J., Pertierra, M. et al. Adversarial genetic programming for cyber security: a rising application domain where GP matters. Genet Program Evolvable Mach 21, p. 219–250 (2020). DOI: http://dx.doi.org/10.1007/s10710-020-09389-y.

7. Peng H., Mou L., Li G., Liu Y., Zhang L., Jin Z. Building Program Vector Representations for Deep Learning. In: Zhang, S., Wirsing, M., Zhang, Z. (eds) Knowledge Science, Engineering and Management. KSEM 2015. Lecture Notes in Computer Science(), vol. 9403. Springer, Cham. DOI: http://dx.doi.org/10.1007/978-3-319-25159-2_49.
8. Mou LL., Li G., Zhang L., Wang T., Jin Z. Convolutional neural networks over tree structures for programming language processing. Thirtieth AAAI conference on artificial intelligence. 2016, p. 1287–1293. DOI: http://dx.doi.org/10.13140/RG.2.1.2912.2966.
9. Sun ZY., Zhu QH., Xiong YF., Sun YC., Mou LL., Zhang L. TreeGen: A tree-based transformer architecture for code generation. Thirty-fourth AAAI conference on artificial intelligence, the thirty-second innovative applications of artificial intelligence conference and the tenth AAAI symposium on educational advances in artificial intelligence. 2020, v. 34, p. 8984–8991. DOI: http://dx.doi.org/10.48550/arXiv.1911.09983.
10. Bohannon D. 2018. Invoke-Obfuscation v1.8. URL: https://github.com/danielbohannon/Invoke-Obfuscation (дата обращения: 20.06.2023).
11. Weltner T. 2016. PowerShell Obfuscator. URL: http://www.powertheshell.com/powershell-obfuscator/ (дата обращения: 20.06.2023).
12. Liu C., Xia B., Yu M., Liu YZ. PSDEM: A feasible de-obfuscation method for malicious PowerShell detection. IEEE symposium on computers and communications (ISCC). 2018, p. 830–836. DOI: http://dx.doi.org/10.1109/ISCC.2018.8538691.
13. Ugarte D., Maiorca D., Cara F., Giacinto G. PowerDrive: Accurate De-obfuscation and Analysis of PowerShell Malware. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2019. Lecture Notes in Computer Science(), vol 11543. Springer, Cham. DOI: http://dx.doi.org/10.1007/978-3-030-22038-9_12.
14. Malandrone, G.M., Virdis, G., Giacinto, G., Maiorca, D. PowerDecode: a PowerShell script decoder dedicated to malware analysis. In 5th Italian Conference on CyberSecurity (ITASEC), 2021. URL: https://www.semanticscholar.org/paper/PowerDecode%3A-A-PowerShell-Script-Decoder-Dedicated-Malandrone-Virdis/0f0eaa095288ef07df278a525da1dbb039604bb1 (дата обращения: 20.06.2023).
15. PowerDecode. URL: https://github.com/Malandrone/PowerDecode (дата обращения: 20.06.2023).
16. PaloAlto. Networks. PowerShell Corpus. Fileset, 2018. URL: https://www.paloaltonetworks.com (дата обращения: 20.06.2023).
17. Weber R., Schek H.J., Blott S. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. Proceedings of the 24th VLDB Conference, New York. 1998, p. 194–205. URL: https://www.semanticscholar.org/paper/A-Quantitative-Analysis-and-Performance-Study-for-Weber-Schek/63eaeb0c48175065ffd096aad10aed712c6d7bbb (дата обращения: 20.06.2023).
18. Mares I. (2006). A QUANTITATIVE ANALYSIS. In Taxation, Wage Bargaining, and Unemployment (Cambridge Studies in Comparative Politics, p. 61-82). Cambridge: Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511625688.003.

REFERENCES:

[1] JuanPablo Jofre et al. 2022. PowerShell Scripting. Microsoft Docs. URL: https://docs.microsoft.com/ru-ru/powershell/scripting/overview?view=powershell-7.2 (accessed: 20.06.2023).
[2] Bohannon D. 2018. Invoke-CradleCrafter v1.1. URL: https://github.com/danielbohannon/Invoke-CradleCrafter (accessed: 20.06.2023).
[3] Rusak G., Al-Dujaili A., O'Reilly UM. 2018. AST-Based Deep Learning for Detecting Malicious PowerShell. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). Association for Computing Machinery, New York, NY, USA, p. 2276–2278. DOI: http://dx.doi.org/10.1145/3243734.3278496.
[4] Hendler D., Kels S., Rubin A. 2018. Detecting Malicious PowerShell Commands using Deep Neural Networks. In Proceedings of the 2018 on Asia Conference on Computer and Communications Security (ASIACCS '18). Association for Computing Machinery, New York, NY, USA, p. 187–197. DOI: http://dx.doi.org/10.1145/3196494.3196511.
[5] Al-Dujaili A., Huang A., Hemberg E., O'reilly UM. Adversarial Deep Learning for Robust Detection of Binary Encoded Malware. IEEE Security and Privacy Workshops (SPW), San Francisco, CA, USA. 2018, p. 76–82. DOI: http://dx.doi.org/10.1109/SPW.2018.00020.
[6] O'Reilly UM., Toutou J., Pertierra, M. et al. Adversarial genetic programming for cyber security: a rising application domain where GP matters. Genet Program Evolvable Mach 21, p. 219–250 (2020). DOI: http://dx.doi.org/10.1007/s10710-020-09389-y.
[7] Peng H., Mou L., Li G., Liu Y., Zhang L., Jin Z. Building Program Vector Representations for Deep Learning. In: Zhang, S., Wirsing, M., Zhang, Z. (eds) Knowledge Science, Engineering and Management. KSEM 2015.

Lecture Notes in Computer Science(), vol. 9403. Springer, Cham. DOI: http://dx.doi.org/10.1007/978-3-319-25159-2_49.

[8] Mou LL., Li G., Zhang L., Wang T., Jin Z. Convolutional neural networks over tree structures for programming language processing. Thirtieth AAAI conference on artificial intelligence. 2016, p. 1287–1293. DOI: http://dx.doi.org/10.13140/RG.2.1.2912.2966.

[9] Sun ZY., Zhu QH., Xiong YF., Sun YC., Mou LL., Zhang L. TreeGen: A tree-based transformer architecture for code generation. Thirty-fourth AAAI conference on artificial intelligence, the thirty-second innovative applications of artificial intelligence conference and the tenth AAAI symposium on educational advances in artificial intelligence. 2020, v. 34, p. 8984–8991. DOI: http://dx.doi.org/10.48550/arXiv.1911.09983.

[10] Bohannon D. 2018. Invoke-Obfuscation v1.8. URL: https://github.com/danielbohannon/Invoke-Obfuscation (accessed: 20.06.2023).

[11] Weltner T. 2016. PowerShell Obfuscator. URL: http://www.powertheshell.com/powershell-obfuscator/ (accessed: 20.06.2023).

[12] Liu C., Xia B., Yu M., Liu YZ. PSDEM: A feasible de-obfuscation method for malicious PowerShell detection. IEEE symposium on computers and communications (ISCC). 2018, p. 830–836. DOI: http://dx.doi.org/10.1109/ISCC.2018.8538691.

[13] Ugarte D., Maiorca D., Cara F., Giacinto G. PowerDrive: Accurate De-obfuscation and Analysis of PowerShell Malware. In: Perdisci, R., Maurice, C., Giacinto, G., Almgren, M. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2019. Lecture Notes in Computer Science(), vol 11543. Springer, Cham. DOI: http://dx.doi.org/10.1007/978-3-030-22038-9_12.

[14] Malandrone, G.M., Virdis, G., Giacinto, G., Maiorca, D. PowerDecode: a PowerShell script decoder dedicated to malware analysis. In 5th Italian Conference on CyberSecurity (ITASEC), 2021. URL: https://www.semanticscholar.org/paper/PowerDecode%3A-A-PowerShell-Script-Decoder-Dedicated-Malandrone-Virdis/0f0eaa095288ef07df278a525da1dbb039604bb1 (accessed: 20.06.2023).

[15] PowerDecode. URL: https://github.com/Malandrone/PowerDecode (accessed: 20.06.2023).

[16] PaloAlto. Networks. PowerShell Corpus. Fileset, 2018. URL: https://www.paloaltonetworks.com (accessed: 20.06.2023).

[17] Weber R., Schek H.J., Blott S. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. Proceedings of the 24th VLDB Conference, New York. 1998, p. 194–205. URL: https://www.semanticscholar.org/paper/A-Quantitative-Analysis-and-Performance-Study-for-Weber-Schek/63eaeb0c48175065ffd096aad10aed712c6d7bbb (accessed: 20.06.2023).

[18] Mares I. (2006). A QUANTITATIVE ANALYSIS. In Taxation, Wage Bargaining, and Unemployment (Cambridge Studies in Comparative Politics, p. 61-82). Cambridge: Cambridge University Press. DOI: http://dx.doi.org/10.1017/CBO9780511625688.003.

Поступила в редакцию – 26 июня 2023 г. Окончательный вариант –22 Августа 2023 г.
Received – June 26, 2023. The final version – August 22, 2023.