Ведение в специальность

Основы анализа вредоносного ПО

Веричев Александр

Статический и динамический анализ

- Статический анализ исполняемых файлов
- Динамический анализ исполняемых файлов
- Принципы отладки исполняемых файлов
- Методы противодействия статическому и динамическому анализу

Статический анализ файлов

- **Статическим анализом** называется исследование подозрительного файла без его открытия/исполнения
- Методы статического анализа применяются для получения полезной информации и определения направления дальнейшего анализа подозрительного файла
- Основными методы статического анализа:
  - определение типа файла
  - хеширование файла и сканирование антивирусным ПО
  - поиск строк и констант в файле
  - поиск подозрительных функций в таблицах импорта и экспорта
  - распознавание техник обфускации/упаковки исполняемого файла
  - сравнение с образцами и классификация вредоносного ПО

Определение типа файла

- Первый шаг анализа подозрительного файла – определение типа файла
- Тип файла можно определить по его **расширению** (например, исполняемые файлы ОС Windows могут иметь расширения *(.exe*, *.dll*, *.sys*, *.drv*, *.com* и др.)
- Однако подобный подход, очевидно, не является надёжным
- Более надёжным методом, который использует сама ОС при загрузке файла, является анализ т.н. *сигнатуры файла*
- **Сигнатурой файла** называют уникальную последовательность байт, записанную в заголовке (в начале) файла
- Анализ сигнатуры файла можно проводить как вручную, так и с применением специальных инструментов

Определение типа файла

Ручной анализ

- Для проведения сигнатурного анализа вручную требуется обыкновенный хекс-редактор и список сигнатур:
  https://en.wikipedia.org/wiki/List_of_file_signatures

Автоматический анализ

- В *nix системах определить тип файла можно с помощью утилиты **file**
- На ОС Windows для автоматического анализа можно применять утилиту *TrID*
  https://mark0.net/soft-trid-e.html
- Также в ОС Windows может быть полезной утилита *CFF Explorer*
  https://ntcore.com/?page_id=388

Хеширование файлов и сканирование антивирусами

- Как и расширение файла, его название не может быть использовано для детектирования вредоносного ПО (*почти* никто не запустит файл *virus-trojan-ransomware.exe*, присланный по почте)
- Вместе с тем, если сам исполняемый файл не подвергался изменениям, любые детерминированные преобразования данных (такие как **криптографические хэш функции**), будут принимать одинаковые значения, независимо от названия файла
- Значения хэш-функций используются в качестве опознавателя (ID) вредоносного файла и используются антивирусами для автоматического сканирования файлов защищаемой системы
- Большое количество антивирусов доступно *VirusTotal:* www.virustotal.com
- Значения хэш-функций также публикуются на сайтах вроде *VirusTotal*
- В *nix системах предустановлены утилиты подсчёта хэш-сумм (*md5sum*, *sha1sum* и др.)
- Для вычисления хэш-суммы конкретного файла на ОС Windows можно воспользоваться утилитой *HashMyFiles*…
- … или же просто посчитать на питоне:

```
python -c “import hashlib; print(hashlib.md5(open('file.exe', 'rb').read()).hexdigest())”
python -c “import hashlib; print(hashlib.sha1(open('file.exe', 'rb').read()).hexdigest())”
python -c “import hashlib; print(hashlib.sha256(open('file.exe', 'rb').read()).hexdigest())”
```

Поиск строк и констант

- Строки (последовательности *ASCII* и *Unicode-printable* символов) необходимы для функционирования любого ПО, в том числе и вредоносного
- Извлекаемые из подозрительного исполняемого файла строки могут содержать:
  - названия файлов и пути в файловой системе
  - ключи реестра
  - доменные имена, IP-адреса и URL
  - название импортируемых модулей и функций, информацию о типах данных (*RTTI*)
  - строковые представления команд интерпретатора (в том числе форматные строки)
  - пугающе-информационные сообщения жертве
  - и др.
- Для поиска похожих на строки последовательностей символов в *nix применяется утилита **strings** (портирована на ОС Windows Русиновичем)
- Схожий алгоритм реализован в IDA, результаты представляются в *String View* (*View - Open Subviews – Strings* или *<shift>+F12*)
- *pestudio (*winitor.com*)* также имеет функцию поиска строк

Поиск строк и констант

- Как следствие существования различных утилит автоматического поиска и извлечения строк из исполняемых файлов, строки в вредоносном ПО часто «прячут»: подвергают некоторым преобразованиям
- Для обфускации строк в исполняемых файлах применяют самые разные средства:
  - шифрование простейшими шифрами Цезаря/ROT13
  - XOR с небольшой (короткой – несколько байт) константой
  - применение табличных подстановок (наподобие шифра подстановки)
  - разделение строки на две части, поэлементный XOR которых даёт исходную строку
  - шифрование алгоритмом RC4
  - и многие другие
- С некоторыми обфускациями строк может справится утилита *FLOSS*, FireEye Labs Obfuscated String Solver: https://github.com/fireeye/flare-floss
- Применяемый алгоритм основан на анализе графа потока управления и описан в документации утилиты на *github*
- Применение утилиты тривиально: *floss “/path/to/malware/binary”*

Анализ используемых функций

- Зачастую вредоносному ПО требуется выполнять вполне конкретные действия, требующие использования библиотечных функций и *WinAPI*: создание/изменение файлов, работа с реестром, работа с сетью, загрузка динамически подключаемых библиотек и др.
- Эти функции реализованы в сторонних библиотеках и подключаются динамически – в момент загрузки
- Некоторые функции весьма специфичны и крайне редко напрямую используются обычными приложениями:
  - LoadLibraryA и GetProcAddress
  - GetCurrentProcess, GetProcessHeap
  - и LdrGetProcAddress, LdrLoadDll
  - OpenFile, ReadFile, WriteFile
  - RegisterClassEx, RegisterHotKey
  - LowLevelKeyboardProc
  - и др.
- Для анализа таблицы импорта можно использовать *IDA*, *pestudio*, *Dependency Walker* и др.

Сравнение и классификация вредоносного ПО

- Хеширование подозрительного файла с целью поиска в базе известных образцов – полезный метод первичного анализа исполняемого файла, однако данный подход может детектировать только **точные копии** вредоносного файла
- Даже при наличии **малейших отличий** (*достаточно одного байта*), **хэш-суммы** идентичных по функционалу образцов зачастую **значительно отличаются** друг от друга – хеммингово расстояние максимально
- Это следствие т.н. *лавинного эффекта* некоторых криптографических примитивов
- Детектирование *похожих* образцов требует применения других методов:
  - нечёткое хеширование (*fuzzy hashing*)
  - хеширование имён импортируемых функций (т.н. Import Hash, *imphash*)
  - хеширование секций исполняемого файла по отдельности (*section hashing*)
  - применение классификаторов и других методов *Машинного Обучения*

Сравнение образцов: *fuzzy hashing*

- *ssdeep*
  - разработана в 2006 году Джесси Корнблумом (Jesse Kornblum) на основе утилиты поиска изменений в файлах *spamsum*
  - основана на разбиении файлов на блоки и применении скользящего хэша (md5 или sha-1)
  - URL: https://ssdeep-project.github.io/ssdeep/index.html
  - примеры использования:

вычисление хэш-сумм	ssdeep config.h INSTALL m4/libtool.m4
сравнение файлов	ssdeep -m config.h INSTALL m4/libtool.m4

- *sdhash*
  - утилита sdhash (Similarity Digest Hash) разработана Василем Русевом в 2010 году
  - основана на вычислении нормированной энтропийной меры схожести значений двух хеш-сумм (файл разбивается на блоки по 64 байта, блоки хешируются алгоритмом sha-1 и хеш-суммы подаются на вход фильтра Блума)
  - URL: http://roussev.net/sdhash/sdhash.html
  - примеры использования:

вычисление хэш-сумм	sdhash -o hashes.sdbf config.h INSTALL m4/libtool.m4
сравнение файлов	sdhash -c config.h INSTALL m4/libtool.m4

Сравнение образцов: *fuzzy hashing*

- *mvHash*
  - Утилита *mvHash* (Majority Vote Hash) разработана в 2006 году Джесси Корнблумом (Jesse Kornblum) на основе утилиты поиска изменений в файлах *spamsum*
  - основана на применении фильтра Блума и мажоритарном голосовании, результат которого кодируется длинами серий (*RLE*) и преобразуется в хеш
  - URL: ?

- *mrsh v2*
  - утилита mrsh (Multi-Resolution Similarity Hashing) разработана Василем Русевом и коллегами в 2007 году
  - в 2013 году Фрэнк Брайтингер (Frank Breitinger) реализовал обновлённую версию
  - алгоритм похож на *ssdeep* и основан на вычислении скользящего хеша (применяется полиномиальный хеш djb2) и применении фильтра Блума
  - URL: https://github.com/juxtin/mrsh-v2
  - примеры использования:

    вычисление хэш-сумм mrsh-v2 -p config.h INSTALL m4/libtool.m4

    сравнение фалйов mrsh-v2 -g config.h INSTALL m4/libtool.m4

Сравнение образцов: *section hashing* и *imphash*

- Метод *imphash*, хеширование **таблицы импорта**, основывается на предположении о том, что исполняемые файлы, компилируемые на основе одних исходников одинаковым способом, будут иметь одинаковый imphash
- Совпадение *imphash* двух файлов, разумеется, не означает, что их функционал одинаков, но даёт повод продолжить анализ другими средствами
- В каком-то смысле развитием предыдущего метода является **раздельное хеширование секций** исполняемого файла – кода, данных, импорта/экспорта и др.
- Метод *Section hashing* обладает следующими преимуществами:
  - хеш-суммы секций вычисляются независимо друг от друга, поэтому изменения в секции данных не приводят к изменениям хеш-сумм других секций
  - совпадение хеш-сумм нескольких (хотя бы одной) секций более вероятно, чем совпадение хеш-сумм всего файла
  - совпадение хеш-сумм сразу нескольких секций увеличивает вероятность того, что сравниваемые образцы действительно похожи
- Оба метода хеширования реализованы в *pestudio*

YARA

- *YARA* – инструмент идентификации и классификации и вредоносного ПО
- *YARA* позволяет создавать описание (сигнатуры) вредоносных исполняемых файлов на основе строковых и бинарных шаблонов
- Помимо фиксированных строк файлы-описания *YARA* предоставляют возможность использования развитых механизмом задания условных выражений и даже циклов
- Пример описания файла на языке *YARA* приведён ниже:

```
rule OpinionSpyA
{
    meta:
        description = "OSX.OpinionSpy"
        xprotect_rule = true
    strings:
     $a = {504B010214000A0000000800547D8B3B9B0231BC [4] 502D0700250000000000 [12]
636F6D2F697A666F7267652F697A7061636B2F70616E656C732F706F696E7374616C6C6572}
    condition:
          $a
}
```

- Описания на языке *YARA* известных вредоносных образцов позволяют реализовывать полу-автоматическое сканирование и классификацию исследуемого образца

Литература и полезные ресурсы

1. VirusTotal
   https://www.virustotal.com/gui/home
2. TrID - File Identifier
   https://mark0.net/soft-trid-e.html
3. Windows Sysinternals
   https://docs.microsoft.com/en-us/sysinternals/
4. PEStudio - Malware Initial Assessment
   https://www.winitor.com/
5. PEiD - Packers, Cryptors and Compilers Identifier
   https://www.aldeid.com/wiki/PEiD
6. FireEye Labs Obfuscated String Solver
   https://github.com/fireeye/flare-floss
7. YARA – Malware Classification Tool
   https://virustotal.github.io/yara/
8. Michael Sikorski, Andrew Honig. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. - No Starch Press, 2012 г. - 802 с.
9. Monnappa K.A. Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware. - Packt Publishin, 2018 г. - 512 с.

Статический и динамический анализ

- Статический анализ исполняемых файлов
- Динамический анализ исполняемых файлов
- Принципы отладки исполняемых файлов
- Методы противодействия статическому и динамическому анализу

Динамический анализ

- Динамическим анализом называют любой вид анализа исполняемого файла и окружения, проводимый *во время и после* запуска файла
- Основные методы динамического анализа исполняемого файла:
  1. Трассировка: отслеживание используемых функций (собственных и библиотечных), построение графа потока управления (*Control Flow Graph, CFG*)
  2. Отслеживание файловых операций
  3. Отслеживание попыток доступа к ресурсам ОС: сокеты, мьюткексы/семафоры и прочие объекты ядра, реестр и иные конфигурационные файлы
  4. Отслеживание и анализ сетевой активности: определение используемых сетевых протоколов, устанавливаемых соединений и проч.

Динамический анализ

1. Динамический анализ – следующий шаг в процессе исследования подозрительного образца
2. К динамическому анализу прибегают в ситуациях, когда статический анализ исчерпал себя, и дальнейшее исследование исполняемого файла требует его исполнения
3. Так как подозрительный образец может оказаться вредоносным ПО, запуск исполняемого файла на рабочей машине может привести к значительным разрушительным (возможно, необратимым) последствиям
4. В связи с этим, для запуска подозрительных образцов используют:
    1. специализированные песочницы (Cuckoo Sandbox, *GFI Sandbox*, *Norman Sandbox*, *Falcon Sandbox* и др.)
    2. изолированные ПК/рабочие станции
    3. виртуальные машины

Динамический анализ в VM

Динамический анализ исполняемых файлов в виртуальной машине позволяет выполнять ряд крайне полезных операций:

1. запускать образец в виртуальной среде
2. строить виртуальную сеть для образца и изолировать его от сети Интернет
3. использовать снимки VM (*snapshots*) для изучения разностного образа жёсткого диска
4. записывать производимые действия и проигрывать из заново (*record and replay*)

Динамический анализ: трассировка *cfg* и *fs*

Трассировка исполняемого файла позволяет получать информацию об используемых функциях, файловых операциях, работе с реестром и т.п.

Linux

1. функции: strace, ltrace, valgrind’s callgrind

   strace /bin/ls

   ltrace -S /bin/ls

   valgrind --tool=callgrind --time-stamp=yes --dump-instr=yes --callgrind-out-file=curl.tr.%p curl -v https://yandex.ru/

2. файловые операции: lsof

   lsof -u user

   lsof -c kcachegrind

Windows

1. отслеживание операций с файловой системой, реестром: *Process Monitor*
2. отслеживание операций запущенных процессов: *Process Explorer*
3. diff состояний реестра Windows: *Regshot*

Динамический анализ: контроль сетевой активности

Трассировка исполняемого файла позволяет получать информацию об используемых функциях, файловых операциях, работе с реестром и т.п.

Linux

1. сбор сетевого трафика: *tcpdump*, *tshark*

   ```
   tcpdump -i any -w traffic.pcap
   tcpdump -i enp0s3 -w traffic.pcap port 80
   valgrind --tool=callgrind --time-stamp=yes --dump-instr=yes --callgrind-out-file=curl.tr.%p curl -v https://yandex.ru/
   ```

2. имитация сетевых служб: *iNetSim*

   ```
   sudo inetsim
   ```

3. консольный клиент/сервер (*«швейцарский ножик»*): *netcat*

   ```
   nc 192.168.56.3 8888
   nc -l 8888
   ```

Windows

1. сбор сетевого трафика: *Wireshark*
2. консольный клиент: *netcat*
3. имитация службы *DNS*: *ApateDNS*, *fakenet-ng*

Литература и полезные ресурсы

1. Windows Sysinternals
   https://docs.microsoft.com/en-us/sysinternals/
2. Dependency Walker – Инструмент анализа зависимостей исполняемых файлов
   https://www.dependencywalker.com/
3. Regshot
   https://sourceforge.net/projects/regshot/
4. Wireshark
   https://www.wireshark.org/
5. INetSim - Software Suite for Simulating Common Internet Services
   https://www.inetsim.org/
6. FakeNet-NG - Network Analysis Tool
   https://github.com/fireeye/flare-fakenet-ng
7. Michael Sikorski, Andrew Honig. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. - No Starch Press, 2012 г. - 802 с.
8. Monnappa K.A. Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware. - Packt Publishin, 2018 г. - 512 с.

Статический и динамический анализ

- Статический анализ исполняемых файлов
- Динамический анализ исполняемых файлов
- Принципы отладки исполняемых файлов
- Методы противодействия статическому и динамическому анализу

Принципы отладки исполняемых файлов

- Принципы «классической» отладки:
  1. *source-level* vs *assembly-level*
     отладка может осуществляться на уровне исходного кода – по строкам (т.е. блоками инструкций), либо же на уровне инструкций машинного кода
  2. *user-mode* vs *kernel-mode*
     механизмы отладки позволяют работать исследовать как обычные процессы (располагающиеся в т.н. *User space*), так и драйверы и компоненты ядра (располагающиеся, соответственно, в *Kernel space*)
  3. *исключения*
     процесс отладки основан на порождении специальных исключительных ситуаций и обработке их в отладчике: при возникновении исключения в отлаживаемом процессе (например, срабатывание программной точки останова) исполнение приостанавливается и управление передаётся отладчику, который может обработать исключение, и/или «вернуть» его отлаживаемому процессу

     «штатные исключения» называются *first-chance exceptions*, а фатальные для отлаживаемого процесса (приводящие при обычном запуске процесса к аварийному завершению) – *second chance*; отладчик перехватывает, обрабатывает и при необходимости пересылает процессу оба типа исключений
  4. *точки останова*
     существуют два типа точек останова:
     - программные: реализуются с помощью инструкции *int 3* (оп. код *“\xcc”* или *“\xcd\x03”*), выполнение которой порождает специальное исключение, называемое *software interrupt*
     - аппаратные: реализуются на уровне процессора и позволяют остановить исполнение процесса при доступе на чтение, запись или исполнение *конкретного адреса* виртуального адресного пространства
  5. *пошаговая трассировка*
     в процессах архитектуры *x86-64* имеется возможность установить специальный флаг *Trap flag* в *1,* в результате чего *software interrupt* будет порождаться после исполнения каждой инструкции

- Другим подходом к отладке исполняемых файлов является динамическая бинарная инструментализация: вместо установки программных точек останова путём перезаписи первого байта инструкции байтом “*0xcc*” в код отлаживаемого приложения добавляются т.н. *трамплины*, переводящие управление на специальные обработчики (называемые *инструменты*), которые реализуют логику отладки исследуемого процесса

Статический и динамический анализ

- Статический анализ исполняемых файлов
- Динамический анализ исполняемых файлов
- Принципы отладки исполняемых файлов
- Методы противодействия статическому и динамическому анализу

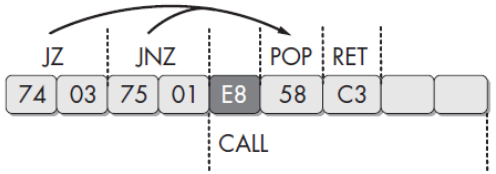
Противодействие статическому анализу

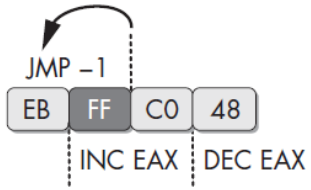
В качестве мер противодействия различным формам статического анализа применяются многочисленные техники защиты, в совокупности называемых обфускация:

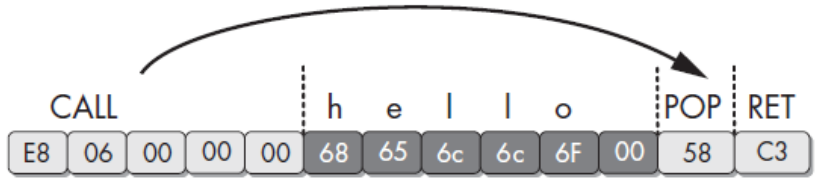
- значительное изменение исходного кода исполняемого файла без изменения его функционала (или почти без)
- шифрование используемых строк и прочих ресурсов исполняемого файла
- анти-дизассемблирование: модификация скомпилированного кода с целью противодействия дизассемблированию
- модификации графа потока управления: «висячие» указатели, косвенные переходы, применение SEH/обработчиков сигналов и др.
- комбинации приведённых выше методов, в т.ч. т.н. упаковка исполняемого файла

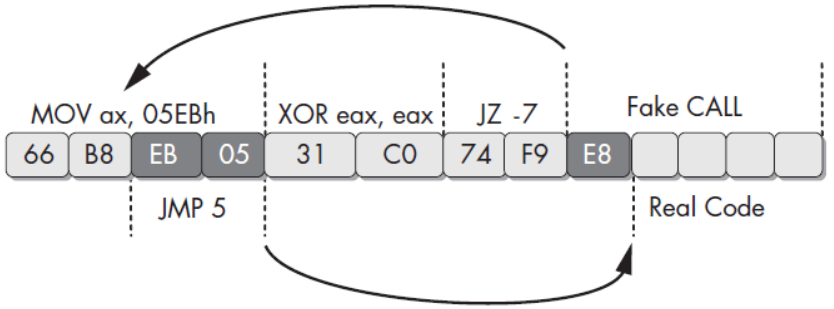
Техники анти-дизассемблирования

- Наивный подход *linear sweep* заключается в последовательном *линейном* дизассемблировании последовательности байт: следующая инструкция ищется сразу же за обнаруженной текущей
- Метод *Flow-oriented* основывается на анализе адресов перехода инструкций условного и безусловного перехода и маркировании соответствующих адресов в качестве начала потенциальной инструкции; такой подход позволяет игнорировать участки адресного пространства, до которых не доходит управление
- Тем не менее даже *flow-oriented* алгоритм, будучи статической по типу методикой анализа, не способен точно восстанавливать
- Перечисленные недостатки алгоритмов дизассемблирования часто используют для защиты исполняемого файла от анализа
- Основные «гаджеты» анти-дизассемблирования
  - «упаковка» строк в сегменте кода
  - расщепление инструкции безусловного перехода jmp
  - условный переход по константному условию
  - использование некоторых байт кода нескольких инструкциях









*иллюстрации из Michael Sikorski, Andrew Honig. Practical Malware Analysis

Противодействие динамическому анализу

- Методы противодействия отладке основываются на анализе механизма отладки ПО и в основном реализуют один из следующих подходов:
  - противодействие механизмам отладки исполняемых файлов
  - поиск признаков (артефактов) производимой отладки в системе и окружении исследуемого процесса

- Многочисленные методы противодействия отладке ПО можно подразделить на следующие группы:
  - *универсальные* – не зависящие от ОС или аппаратной платформы
  - *аппаратно-ориентированные* – техники, основывающиеся на специфике конкретной аппаратной платформы
  - *ОС-ориентированные* – техники, специфичные для конкретной операционной системы (основанные на эксплуатации внутренних механизмов)

Противодействие анализу: *x86-64 и универсальные*

- Среди возможно бесчисленного количества универсальных методов выделяются следующие:
  1. *Замер времени*
     периодические измерения времени исполнения конкретных участков кода позволяют сделать вывод о том, что процесс запущен в отладке (в случае нереалистично больших временных затрат)
  2. *Поиск отладчиков в дереве процессов*
     отладчики (и *runtime* фреймворков *DBI*) в свою очередь являются исполняемыми файлами, также запускаемыми на целевой ОС, следовательно, сканирование названий запущенных процессов позволяет обнаружить запущенный отладчик и принять соответствующие меры (например, завершить процесс)
  3. *Хэширование машинного кода функций*
     изменение штатного хода исполнения некоторой скомпилированной функции требует внедрения стороннего кода (перезапись первого байта инструкции байтом *0xCC* или установка т.н. *трамплина*), что приводит к изменению значений различных хэш-функций и контрольных сумм; следовательно, периодическое сканирование кода функций и подсчёт контрольных сумм позволяет обнаруживать управляющие воздействия отладчиков и предпринимать необходимые меры (например, аварийно завершать процесс)
  4. *Искусственное порождение исключений*
     подход основан на эксплуатации механизма отладки исполняемых файлов с помощью перехвата исключений: искусственно порождаемые исследуемым исполняемым файлом исключения перехватываются отладчиком, и если эти исключения не пересылаются отлаживаемому процессу (или, например, пересылаются слишком медленно), появляются основания сделать вывод об отладке и предпринимать необходимые меры
  5. *Комбинации указанных выше методов*
- Специфичные для *x86-64:*
  1. *Поиск байт 0xcc или 0xcd,0x03 в сегменте кода* – эти последовательности используются для установки точек останова
  2. *Установка флага отладки Trap flag* – установление флага Trap приводит к порождению исключений после исполнения каждой инструкции отлаживаемого процесса, что мешает работе отладчика

Противодействие анализу: *ОС Windows*

Специфичные для ОС Windows методы в основном основаны на анализе артефактов, появляющихся в окружении процесса во время отладки

- Вызов специальных функции WinAPI:
  1. *IsDebuggerPresent*
     функция возвращает значение поля *IsDebugged* структуры *PEB*, которое устанавливается в *1* для отлаживаемого процесса
  2. *CheckRemoteDebuggerPresent*
     функция выполняет действия, аналогичные *IsDebuggerPresent*
  3. *OutputDebugString*

     функция позволяет послать строку отладчику и тем самым зафиксировать его присутствие
- Анализ полей структуры *_PEB* (*Process Environment Block*) и прочих структур метаданных запущенного процесса:
  1. *NTGlobalFlag*
     в случае отладки ряд битов данного поля структуры *PEB* устанавливаются в *1*
  2. *NtQueryInformationProcess*

     вызов данной нативной функции с конкретными смещениями в качестве аргументов позволяет установить факт отладки заданного процесса
- Скрытный запуск механизмов анти-отладки: *TlsCallback*

  описанные механизмы противодействия отладке должны быть запущены по возможности наименее заметным способом (не в функции *main*, разумеется)

  механизм *TlsCallback* позволяет выполнять код до начала исполнения кода приложения (до вызова функции *main*)

Противодействие противодействию методам анализа

Методы анти-реверса и анти-отладки разнообразны и многочисленны, однако способы борьбы с данными способами противодействия исследованию исполняемых файлов сводятся к двум подходам:

1. маскирование артефактов отладки:
   - «угон» специализированных функций с помощью *dll hijacking* – замена на собственные функции, возвращающие нужные значения
   - игнорирование искусственно порождаемых исключений
   - изменение названий исполняемого файла и процесса отладчика
2. локализация места запуска механизма защиты и *патчинг* исполняемого файла: замена соответствующей части кода последовательностью инструкций *nop*

Литература и полезные ресурсы

1. Anti-Debugging Protection Techniques with Examples https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software
2. Park, Juhyun, et al. "Automatic Detection and Bypassing of Anti-Debugging Techniques for Microsoft Windows Environments." Advances in Electrical and Computer Engineering 19.2 (2019): 23-29.
3. Peter Ferrie. The "Ultimate" Anti-Debugging Reference. - 2012 г. - 145 с.
4. Michael Sikorski, Andrew Honig. Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software. - No Starch Press, 2012 г. - 802 с.
5. Chris Eagle. The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler. 2nd Edition. - No Starch Press, 2011 г. - 672 с.