Статья Introducing MIDNIGHTTRAIN - A Covert Stage-3 Persistence Framework weaponizing UEFI variables

Приблизительное время чтения: 16 минут.

Introducing MIDNIGHTTRAIN - A Covert Stage-3 Persistence Framework weaponizing UEFI variables

TL;DR
(Too long; Didn't read)

MIDNIGHTTRAIN предназначен для ред-тимеров, исследователей и т.д. Инструмент является опен-сурсным и бесплатным.

Доступен он здесь.

Предупреждение: Этот фреймворк был создан как "уик-энд" проект и получил ограниченное тестирование, так что баги обеспечены. Я не профессиональный программист, следовательно будьте готовы к прочтению говно-кода.
Тем не менее, я готов исправлять свои оплошности в свободное время. Если вы найдете какие-либо ошибки/баги или более того - возможность что-то улучшить, не стесняйтесь мне написать!

Вступление

Одно из моих любимых увлечений - это чтение отчетов по **APT** и поиск интересных **TTP** (Tactics, Techniques and Procedures), и в последующем их воссоздание на моей виртуальной машине.

Прошлая пятница не была чем-то отличительным, кроме того, что я рылся в *Vault7*, в ветке документов *EDB*. Мое внимание привлек документ, описывающий **NVRAM Variables** (Non-Volatile Random Access Memory Variables), что это такое, где оно есть, в общем их теорию. В прочем, это и вызвало мой интерес и я начал копать глубже.

Оказывается, что эти переменные не только могут записываться и считываться в юзер-моде (ring3), но и прекрасно подходят для сокрытия конфигурационных данных, ключей шифрования, украденных данных и др., что я узнал после просмотра доклада на одной из DEFCON конференции, благодаря *Topher Timzen* и *Michael Leibowitz*.

В докладе использовался C#, но докладчики предлагают посетителям (и не только) придумать свой собственный метод использования этой техники.

Это заставило меня задуматься над различными способами применения этой техники. Тут я внезапно вспомнил, как *ESET* "замалчивал" отчет о *DePriMon* - загрузчике, предположительно принадлежащий (какая ирония!) ЦРУ. Который был зарегистрирован как обычный **Print Monitor** для достижения устойчивости на хосте (отсюда и название - **De**fault **Pri**nt **Mon**itor).

Именно это можно считать рождением фреймворка **MIDNIGHTTRAIN**. Последующие 2 дня я провел за написанием кода, рефакторингом и написанием этой статьи.

NVRAM Variables, Print Monitors, Execution Guardrails вместе с DPAPI, Thread Hijacking и прочее

Для "непросветленных" читателей: не пугайтесь этих "умных словечек", я могу гарантировать, что тут нечего бояться, я так же постараюсь объяснить каждый из индивидуальных компонентов.

Но для начала давайте пройдемся по основным понятиям.

Для опытных читателей: можете спокойно пропустить эту часть статьи и перейти сразу к архитектуре фреймворка.

NVRAM Variables

Я не собираюсь докучать тебя обилием теоретической части, это не моя цель. Просто знай, что все современные машины с **UEFI** используют эти переменные для хранения важных "boot-time" и "vendor-specific" данных в *флэш-памяти*. Так же необходимо сказать, что данные, содержащиеся в таких переменных переживут полную переустановку ОС, и как говорит CIA: "невидимы для форензического образа жесткого диска" (англ: "are invisible to a forensic image of the hard drive").

Достаточно скрытное место для наших маленьких секретов, не так ли?

Что ещё? Как бы ни было легко записывать данные в "firmware variables" из юзер-мода (ring3), то вайтхетам невероятно трудно достать и "перечислять" эти же данные из одного и того же режима.

Но как так, спросишь ты? Скоро увидишь, нетерпеливый друг.

Для нас, редхетов, Microsoft предоставляет полностью задокументированный доступ к **API**, к "волшебной" стране "firmware variables", используя:

`SetFirmwareEnvironmentVariableA()` - для создания и присвоения значения переменной **NVRAM**:

C:

```
BOOL SetFirmwareEnvironmentVariableA(
  LPCSTR lpName,
  LPCSTR lpGuid,
  PVOID  pValue,
  DWORD  nSize
);
```

`GetFirmwareEnvironmentVariableA()` - для получения значения переменной **NVRAM**:

C:

```
DWORD GetFirmwareEnvironmentVariableA(
  LPCSTR lpName,
  LPCSTR lpGuid,
  PVOID  pBuffer,
  DWORD  nSize
);
```

Если тебе интересно, что такое **GUID** (Globally Unique Identifier) вместе с именем переменной, то это просто способ определить конкретную переменную, о которой будет идти речь. Поэтому каждая переменная имеет свои уникальные **GUID** и имя.

Теперь стало понятно, почему почти невозможно достать и "перечислить" данные из юзер-мода. Перечисление требует точного **GUID** и имени переменной. Более того, чтоб проверить существование такой переменной вам опять же понадобится **GUID** и имя.

Окей, это очень хорошо чтоб быть правдой, но должны же быть оговорки, не так ли? Могу ли я вызвать **API**, будучи юзером?

Вопрос хороший, но ответ - нет.

Использование этих **API** функций требует от вас прав "локального администратора", чтоб были определенные привелигии, которые доступны и включены в токене: `SeSystemEnvironmentPrivilege/SE_SYSTEM_ENVIRONMENT_NAME` . Это значит, что наш фреймворк не сможет установится без этих прав.

Вторая оговорка - это размер этих переменных. Сколько данных может содержаться в **NVRAM** переменных и сколько можно их создать?

Опытным путем было выявлено, что можно создать 50 таких переменных, каждая из которых может содержать в себе по 1000 символов. Превышение этих чисел выкинет ошибку 1470.

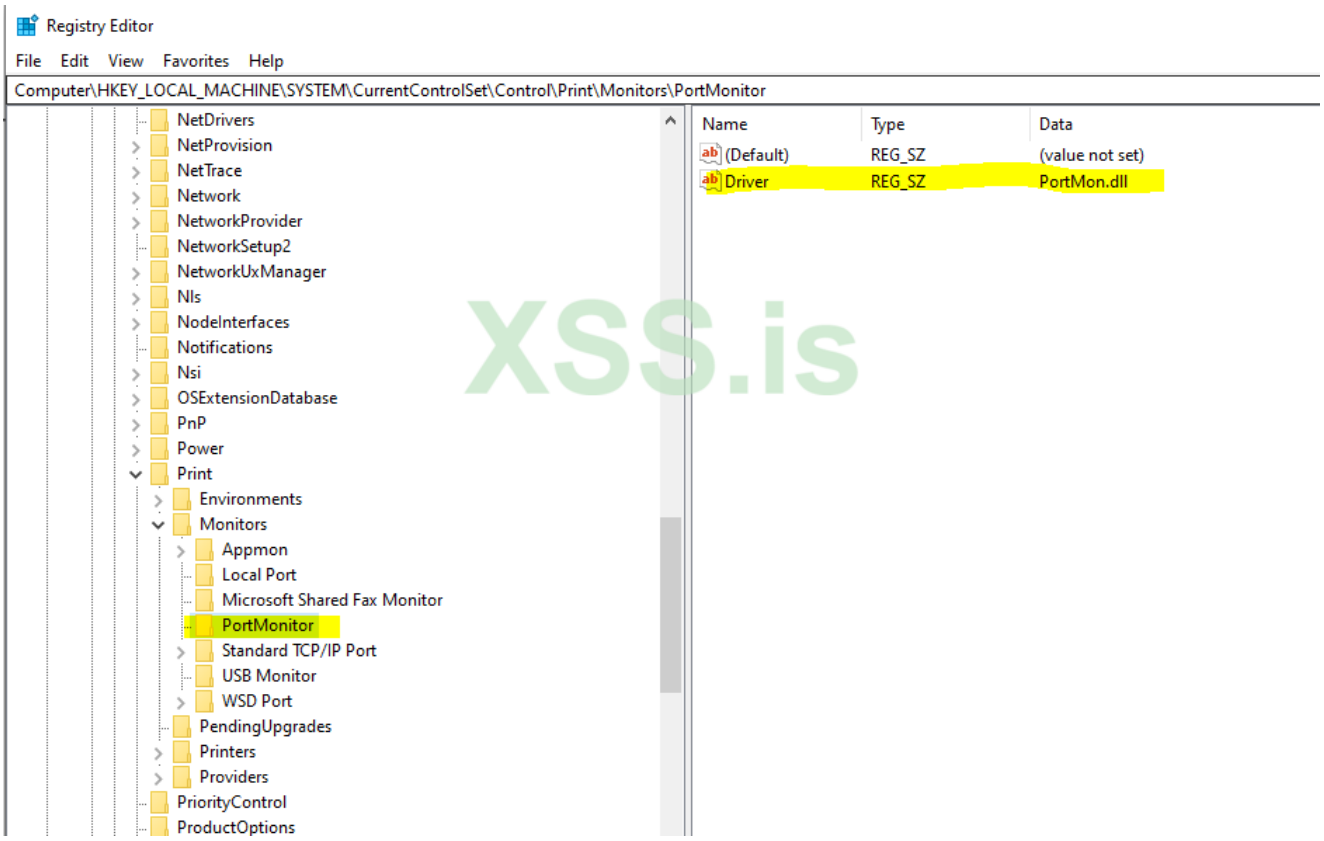
И еще, хороший момент, чтоб сказать, что такие переменные можно вытащить из кернел-мода (ring0), используя такие фреймворки, как *CHIPSEC* или физический доступ к машине с **UEFI**.

Port Monitors

Еще раз, я не собираюсь докучать бесконечной теорией. Просто важно знать несколько вещей. **Port Monitors** это юзер-мод DLL которые, согласно **MSDN**: "**Port Monitors** отвечают за обеспечение коммуникации между юзер-мод спулером и кернел-мод драйверами, имеющими аппаратный доступ к I/O" (англ: "Port Monitors responsible for providing a communications path between the user-mode print spooler and the kernel-mode port drivers that access I/O port hardware").

Эти DLL'ки подгружаются процессом **spoolsv.exe** (Print Spooler Service) на этапе запуска, в первую очередь нам необходимо использовать один из двух методов:

1. Полный путь для записи DLL должен выглядить вот так:
`HKLM\SYSTEM\CurrentControlSet\Control\Print\Monitors`



Этот метод потребует от нас ручной записи в регистр или при помощи **WinAPI**, это позволит нам загружать произвольные DLL'ки.

2. Второй метод более интересный, но имеет пару ограничений:

- DLL должна находится в `system32`
- Произвольная DLL не может быть загружена этой техникой (ну, может, но без "персистентности"(англ: "persistence")), DLL должна быть записана специальным методом; должна экспортироваться функция с именем `InitializePrintMonitor2`, которая вызывается сразу же после загрузки DLL.

И наконец, наш **Port Monitor** может быть зарегистрирован таким образом:

`AddMonitor()` - установка локального **port monitor**:

C:

```
BOOL AddMonitor(
  _In_ LPTSTR pName,
  _In_ DWORD  Level,
  _In_ LPBYTE pMonitors
);
```

Под капотом эта функция добавляет такую же запись реестра и загружает DLL внутри `spoolsv.exe`, но без какого-то прямого вмешательства. Читатели должны взять на заметку, что если DLL создана не в точности по шаблону спецификации, как на **MSDN**, то при загрузке DLL во время текущей сессии нужные значения реестра не будут изменены и DLL не загрузится после перезагрузки устройства, это противоречит назначению самой функции.

И деинсталяция **Port Monitor**:

`DeleteMonitor()` - удаление локального **Port Monitor**.

C:

```
BOOL DeleteMonitor(
  _In_ LPTSTR pName,
  _In_ LPTSTR pEnvironment,
  _In_ LPTSTR pMonitorName
);
```

Для фреймворка был выбран второй метод.

Техника Execution Guardrails вместе с DPAPI

Ладно, я люблю использовать эту технику в моем коде по двум причинам:

1. Чтобы избежать случайное нарушение правило "ведения боя". Это гарантирует, что наш зверек не будет выполнен на любом непреднамеренном хосте.
2. Помешать блу-тимерам зареверсить нашего зверька, помешать анализу на автоматизированных малварь-песочницах.

Чтож, а что такое **DPAPI**?

DPAPI (Data Protection API) это просто набор функций, предоставленный Microsoft и предназначенный для обеспечения конфиденциальности, а так же целостности локально хранящихся учетных данных. Например: пароли браузера, WiFi PSK и т.д.

Все это достигается двумя функциями:

1. `CryptProtectData()` - MSDN:

C:

```
DPAPI_IMP BOOL CryptProtectData(
  DATA_BLOB                 *pDataIn,
  LPCWSTR                   szDataDescr,
  DATA_BLOB                 *pOptionalEntropy,
  PVOID                     pvReserved,
  CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
  DWORD                     dwFlags,
  DATA_BLOB                 *pDataOut
);
```

2. `CryptUnprotectData()` - MSDN:

C:

```
DPAPI_IMP BOOL CryptUnprotectData(
  DATA_BLOB                 *pDataIn,
  LPWSTR                    *ppszDataDescr,
  DATA_BLOB                 *pOptionalEntropy,
  PVOID                     pvReserved,
  CRYPTPROTECT_PROMPTSTRUCT *pPromptStruct,
  DWORD                     dwFlags,
  DATA_BLOB                 *pDataOut
);
```

Эти функции конечно же просты в использовании, но они дают еще одну выгоду.

Если мы можем зашифровать блок данных вместе с **DPAPI** на целевом хосте, то расшифровать это уже нигде не будет представляться возможным, кроме как на самом целевом хосте. Это значит, что, к примеру, **payload**, зашифрованный непосредственно на целевом хосте, должен расшифровываться и выполняться на этом же хосте.

Я вдохновился такой идеей от зверька под именем *InvisiMole*, тех. анализ любезно предоставлен *ESET*.

Вы можете более детально почитать о **DPAPI**здесь.

Thread Hijacking

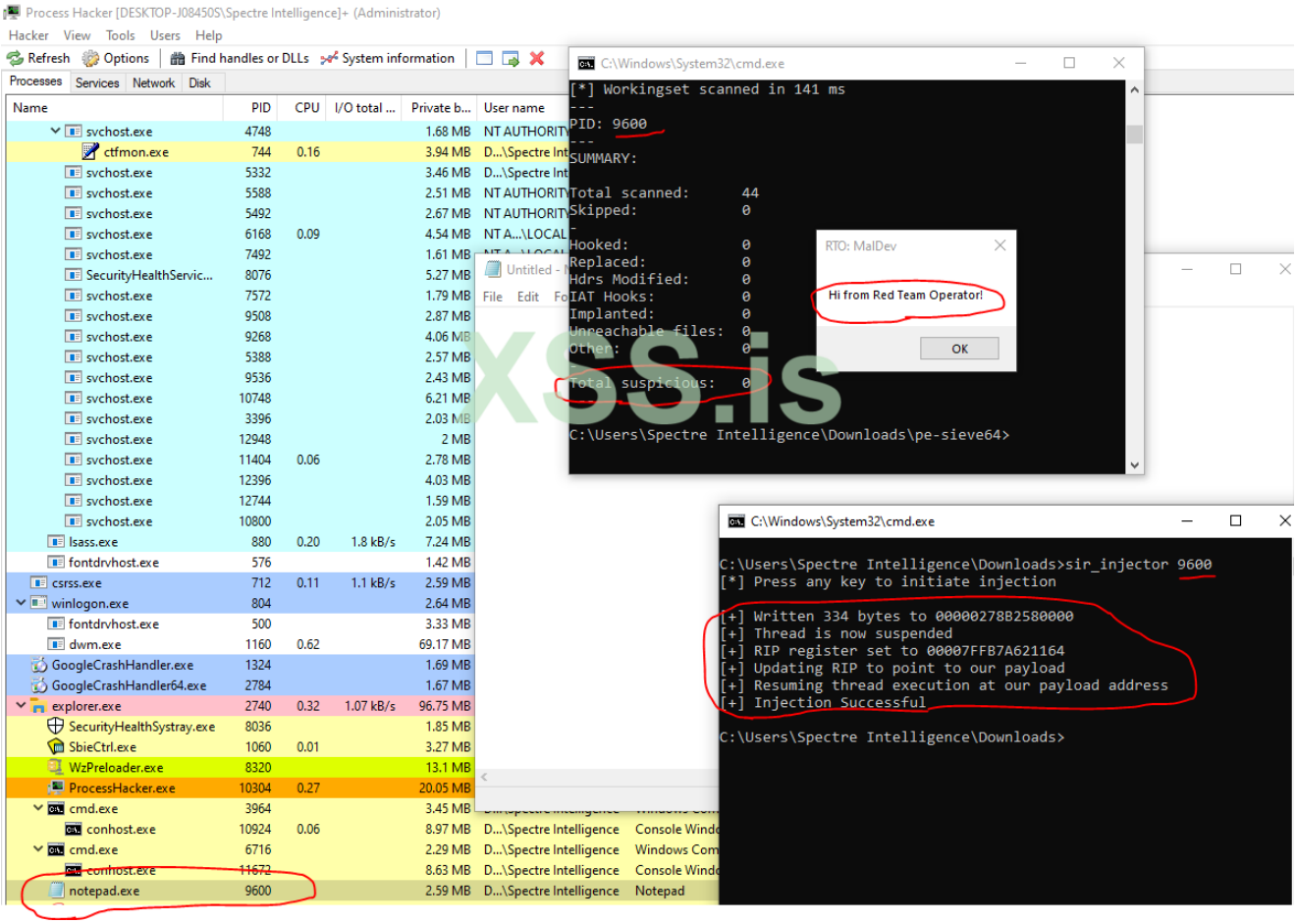
Типичная инжекция кода использует инжекцию в целевой процесс с помощью задокументированного `CreateRemoteThread()` или менее документированного `RtlCreateUserThread()` , или эквивалентный `NtCreateThreadEx()` .

Что происходит в **Thread Injection**, так это то, что в удаленном процессе создается поток, который служит для выполнения нашего вредоносного кода.

Это остается одним из самых популярных и легких способов реализации инжектирования кода, но это так же имеет некоторые недостатки с точки зрения **OPSEC**. С помощью инструментов, таких как *Get-InjectedThread*, становится очень просто обнаружить поток в удаленном процессе, заметив недостающие `MEM_IMAGE` флаги в сегменте запуска потока.

Можно не инжектиться в поток, а перехватить существующий, приостановив его, затем перенаправив `RIP` регистр на наш вредоносный код, прежде чем возобновить поток снова, чтоб на этот раз запустить наш код.

Это еще любят называть **SiR** (Suspend-Inject-Resume) инъекцией.



Чтоб осуществить **перехват**, нам сперва нужно выполнить следующие шаги (и вызовы **API**):

1. `VirtualAllocEx()` - выделяем память в целевом процессе для нашего шеллкода;
2. `WriteProcessMemory()` - записываем наш шеллкод в выделенную память целевого процесса;
3. `SuspendThread()` - приостанавливаем поток;
4. `GetThreadContext()` - получаем состояния регистров для нашего перехваченного потока;
5. `SetThreadContext()` - устанавливаем обновленное состояние регистров для перехваченного потока. Теперь `RIP` регистр перенаправлен на наш шеллкод;
6. `ResumeThread()` - возобновляем поток.

Можно произвести еще один, дополнительный вызов `VirtualAllocEx()`, что я и смело сделал, так как выделение `RWX` памяти в удаленном процессе не очень хорошо воспринимается *PSP* (прим. переводчика: PSP это поставщики услуг, если я ничего не перепутал).

Этот не хитрый ход сначала выделяет `RW` -пейджи для записи нашего **payload**, затем меняет пейджи-защиты на `RX` до возобновления потока, так что теперь **payload** может быть выполнен.

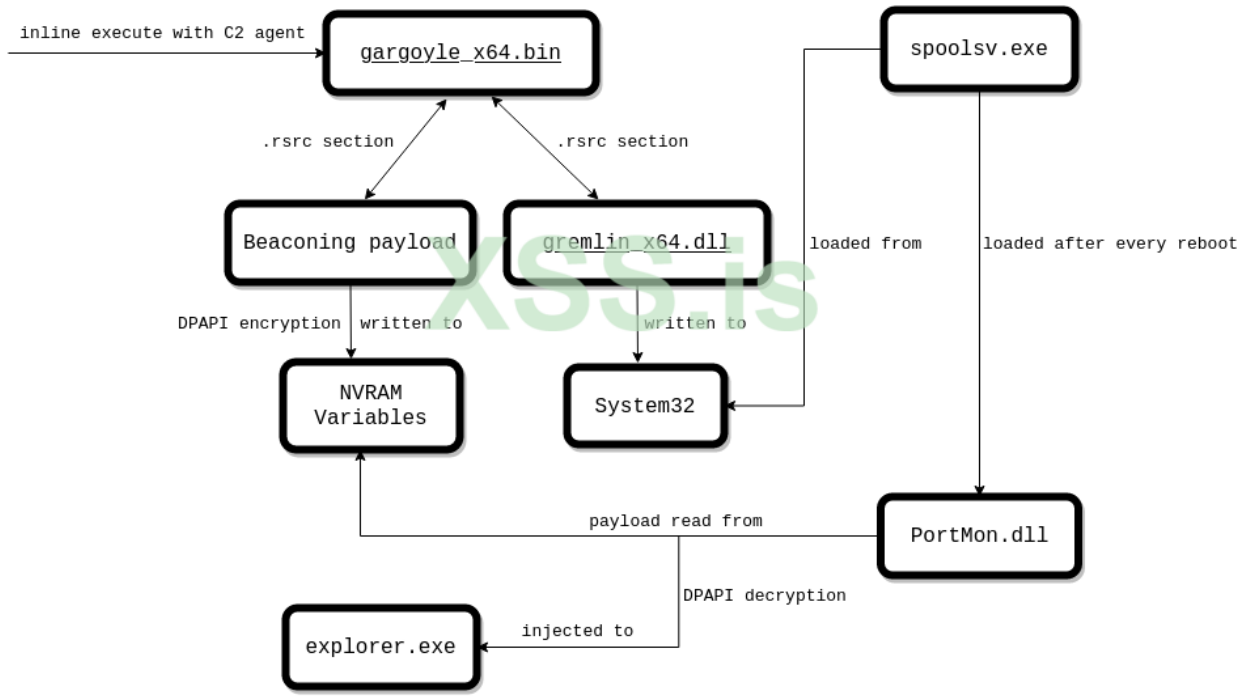
Последнее, что стоит сказать, так это то, что такой метод не очень-то и стабильный, следовательно есть шанс сбоя целевого процесса после завершения нашего вредоносного кода (для фикса требуется клин-ап)

Возможная альтернатива может включать создание нового процесса и последующий его перехват, но я бы хотел избежать так называемый **Sysmon Event ID 1**. В идеале, нужно взвесить все "+" и "-", принимая во внимание целевое окружение и другие факторы, а затем отредактировать код, соответствующий вашим потребностям.

Фреймворк MIDNIGHTTRAIN

То, что ты прочитал до сих пор, объясняет твое "**Что?**". Теперь мы имеем более-менее хорошее понимание индивидуальных частиц пазла, давайте двигаться дальше, чтоб собрать все части пазла воедино и попытаемся ответить на ваш вопрос: "**Как?**".

Но сперва, давайте посмотрим на эту блок-схему, она поможет нам визуализировать архитектуру фреймворка:



На схеме мы можем увидеть, что фреймворк состоит из двух **payload'ов**:

1. `Gremlin` - **Port Monitor DLL.**

2. `Gargoyle` - **персистентный установщик** ("Persistence Installer").

Оба из них скомпилированы в DLL, а с `Gargoyle` **payload'ом** был предпринят еще один дополнительный шаг для преобразования **payload'а** в **PIC-блок** (Position Independent Code). Спасибо *Nick Landers* за замечательный **sRDI** (Shellcode Reflective DLL Injection).

Все это сделано для того, чтоб фреймворк мог гарантировать, что **persistence** "доставлен" и установлен с **inline/поточнымисполнением** шеллкода.

Когда `Gargoyle` запущена в памяти в **Elevated Context**, перед ней ставятся 2 задачи:

1. Выявить, установлен ли **persistence** на целевом хосте или нет. Если нет:
   - Извлекаем `Gremlin` implant-DLL в папку `system32` из ресурс-секции перед его установкой в качестве **Port Monitor DLL**.
   - Извлекаем **payload** шеллкода из **ресурс-секции**, шифруем **payload** с помощью **DPAPI** на целевом хосте, `Base64URL` закодирует зашифрованный **payload** и разделит его на фрагменты, прежде чем записать это все во столько **NVRAM переменных**, во сколько это допустимо флэш-чипом.

2. Если наш **persistence** уже установлен, то:
   - Удаляем `Gremlin` из `system32` .
   - Выгружаем его же из `spoolsv.exe` .
   - Удаляем **payload** из **NVRAM переменных**.

Как и было описано ранее, `spoolsv.exe` подгружает `Gremlin` при успешной установки нашего persistence для преследования следующих целей:

1. Кража токена из `winlogon.exe` и представление себя текущим потоком (подробнее об этом позже).
2. Убедиться, что `SeSystemEnvironmentPrivilege/SE_SYSTEM_ENVIRONMENT_NAME` доступен в токене, последующее его включение, если он же доступен.
3. Считать отдельные фрагменты из **NVRAM переменных**, собрать их воедино, чтоб получить зашифрованный **payload** в `Base64URL` -кодировке.
4. Дешифровать блок с помощью **DPAPI**.
5. Перехватить поток `explorer.exe` , чтоб выполнить наш **payload** (`Meterpreter/Beacon/Grunt` и др.).

Соображения по дизайну архитектуры фреймворка и проблемы OPSEC

Если вы уже столько прочитали, то у вас наверняка куча вопросов, относительно фреймворка. Не волнуйтесь, я постараюсь ответить на всех них и обсудить некоторые **OPSEC** проблемы. Надеюсь, это объяснит ваше "**Почему?**".

Сперва рассмотрим проблему с **UEFI переменными**.

Сейчас уже понятно, что мы мало что сможем сделать с буфферным пространством одной **NVRAM переменной**. Поэтому мы и разбивали **payload** на максимально допустимые фрагменты и затем записывали их в **NVRAM переменные** настолько, насколько нам это позволялось. Как уже было сказано мной ранее, опытным путем было выявлено, что можно создать максимум 50 таких переменных по 1000 символов каждую. Следующая задача это выяснить, какую схему кодирования нам выбрать для хранения байт-блока **payload**. Она должна быть "эффективной", чтоб мы могли выжать максимум из буфферного пространства, данным нам **NVRAM переменными**. `HEX` -кодировка займет 2 символа на 1 байт, в то время как `Base64` займет 3 символа на 4 байта, так что она более эффективна, чем `HEX` -кодировка. Но сможем ли мы сделать что-то получше? Да, вполне. Один из способов - это использование `Base64URL` , которая является *URL-safe* вариантом `Base64` , еще один плюс, что она опускает символ "=".

Так какой максимальный размер хранения payload в **NVRAM переменных**? Где-то `~36КБ` . Сразу же понятно, что с таким размером не может и идти речи о **stageless payload**, генерируемой как "out-of-the-box".

Так чем мы можем воспользоваться? **Staged payload** генериуемый как "out-of-the-box", должен на ура справиться с этим фреймворком. Но тут поднимается **OPSEC** со своими проблемами, так как не рекомендуется использовать стандартный `beacon` **stager**. Но вот как разобраться с **payload**, превышающий размер в `~36КБ` ?

Принимая во внимание все факторы, я рекомендую сделать свой загрузчик payload'a, который будет подгружать конечный payload и выполнять его локально. В этом случае нет необходимости инжектить его снова, так как наш implant уже будет находиться в адрессном пространстве процесса, откуда сетевая активность уже не разпознается как подозрительная для PSP (Поставщики Услуг).

Во-вторых, некоторые из вас могут задаться таким вопросом: "Если мы в любом случае затрагиваем дисковое пространство `Gremlin` 'ом, то зачем нам все эти **NVRAM переменные** и отдельный **payload**? Разве persistence не должен быть частью *Stage-1* и *Stage-2* RAT?".

Краткий ответ: **OPSEC**.

Полный ответ: Конечно, **persistenceдолжен затрагивать дисковое пространство, но мы всегда в силе минимизировать его "влияние", контролируя то, что затрагивает дисковое пространство и что остается только в памяти**. *Stage-1* (Beaconing) и *Stage-2* (Post-Exploitation) RAT **на диске будут просто задетекчены анти-вирусами и EDR** (Endpoint Detection and Response). **На диске делать им нечего** и они **обязаны находится только в памяти**. Но возникает одна проблемка. Если они находятся в памяти, то как мы можем достигнуть "персистентности" с ними? Ответ на этот вопрос будет **"относительно безвредный" (англ: "relatively-benign") persistence implant, который автоматически загружается при запуске машины, что в свою очередь загружает в памятьegress implant**. Каким же образом `Gremlin` получит **egress implant**? Тут 2 пути:

1. Либо по сети, либо что-то получше.

2. Еще какое-то довольно скрытое место для хранения ваших данных в Windows. Одним из таких как раз таки являются **NVRAM переменные**. Другими такими местами могут быть: **NTFS ADS** (Alternate Data Streams), **различные скрытые файловые системы**, **ключи реестра Windows**, **Event Logs** и т.д.

Я просто выбрал **UEFI переменные**, это показалось мне более забавным, нежели чем все остальные места хранения. Поскольку использование их всех в любом случае требует определенных привелигий, я решил использовать **Port Monitor DLL** как **persistence implant**, который загружается через `spoolsv.exe`, который, кстати, является процессом `SYSTEM`, так что это все прекрасно сочетается друг с другом

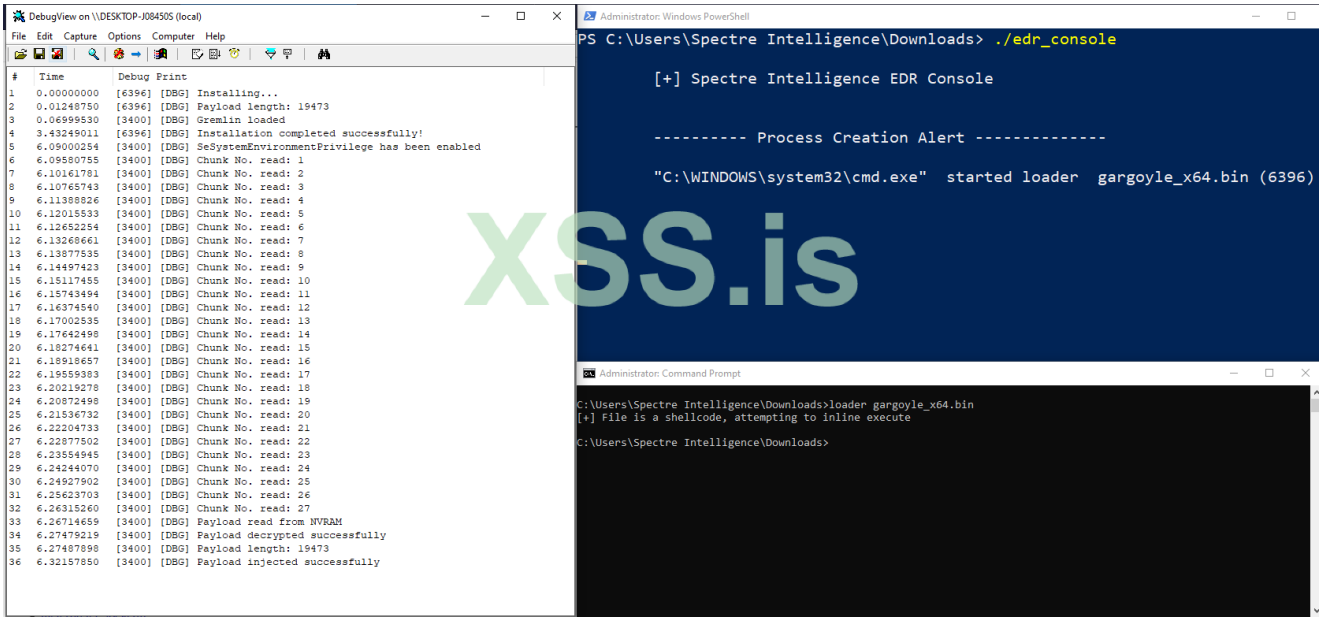
Последняя деталь, на которую стоит обратить внимание, так это то, что хотя `spoolsv.exe` работает как `SYSTEM`, у него все равно нет привелигий для использования **NVRAM переменных**. Следовательно, мы должны осуществить кражу токена, aka **Token Impersonation**, т.е украсть и выдать себя за токен процесса `winlogon.exe`, который имеет необходимые нам привелигии уже в своем токене (хоть и в отключенном состоянии) для вызывающего потока, и только после этого попытаться включить нужную привелигию.

Надеюсь, этим я смог объяснить мотивацию за каждым дизайн-решением.

Скриншоты

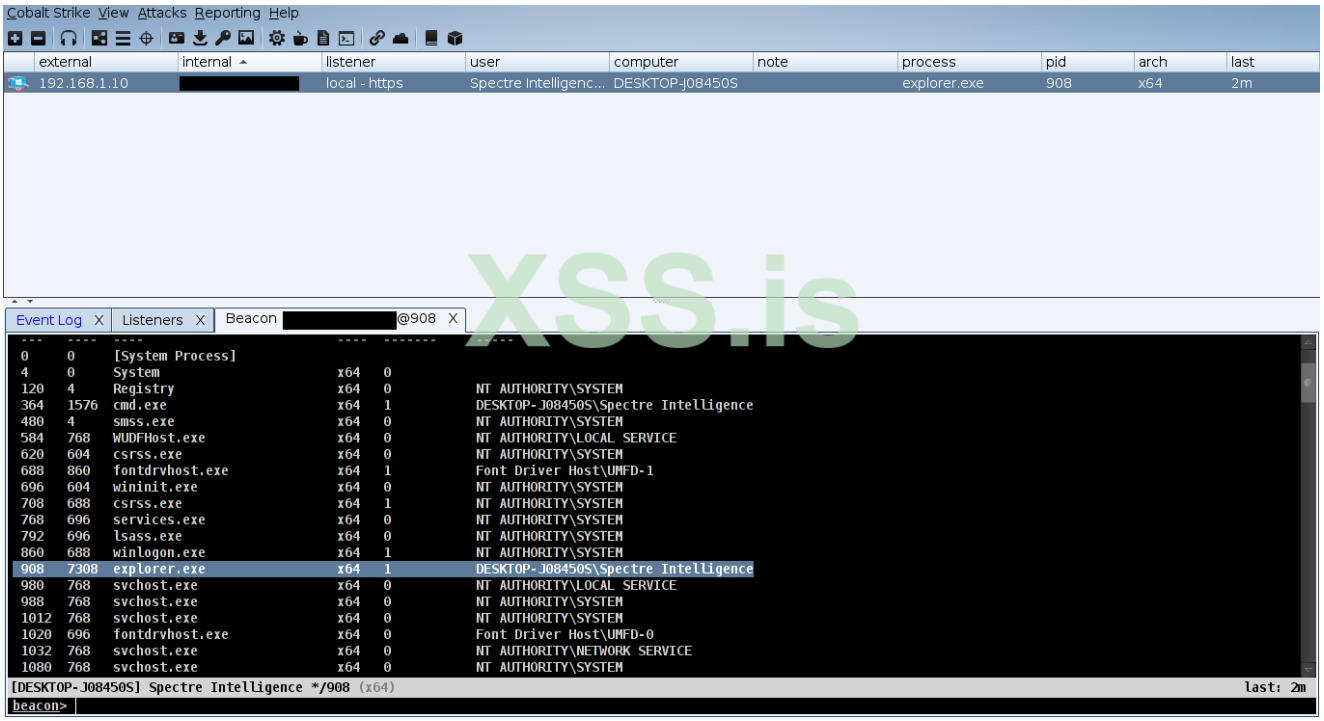
Время скриншотов!

Устанавливаем **persistence**:

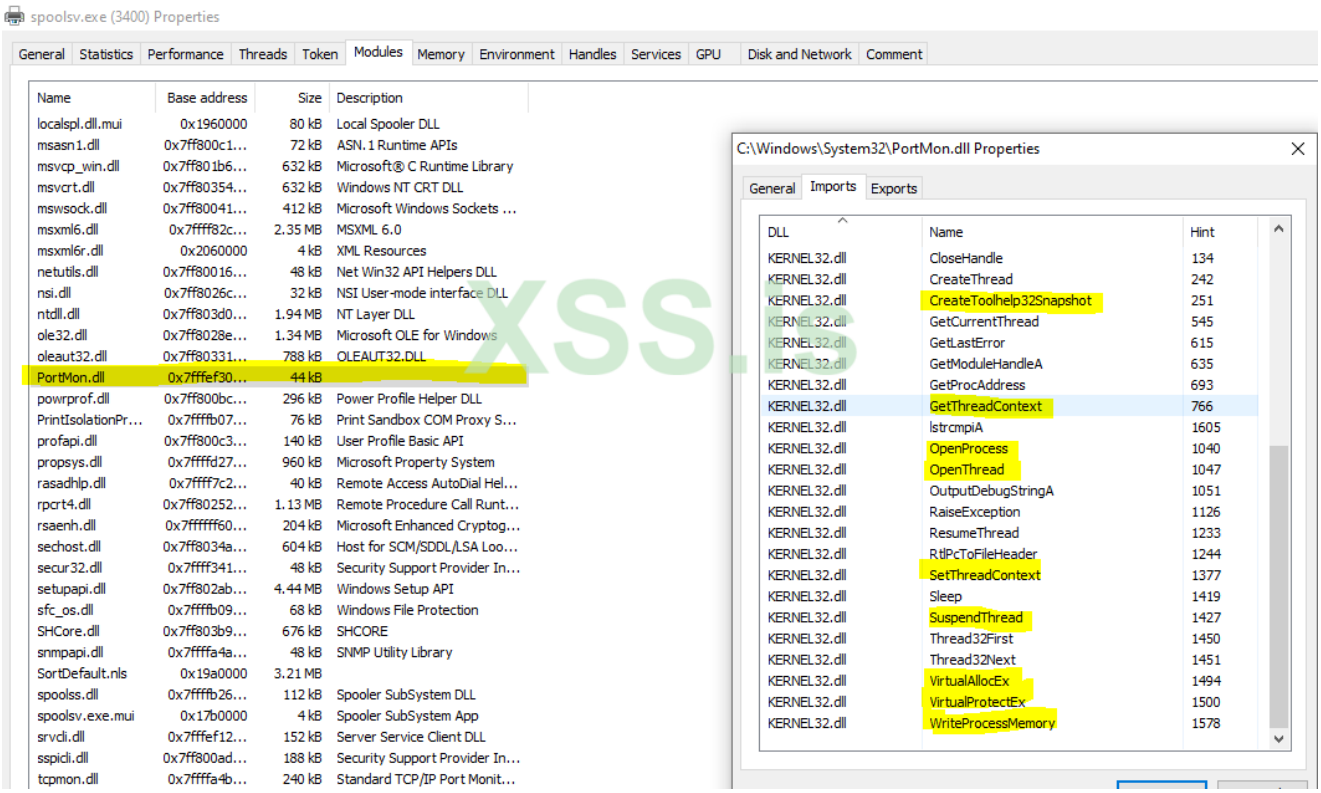


Если вы заинтересовались `edr_console`, то это просто модифицированный парсер **Sysmon EventLog**. Его можно получить здесь.

Мы успешно словили входящий `beacon`:

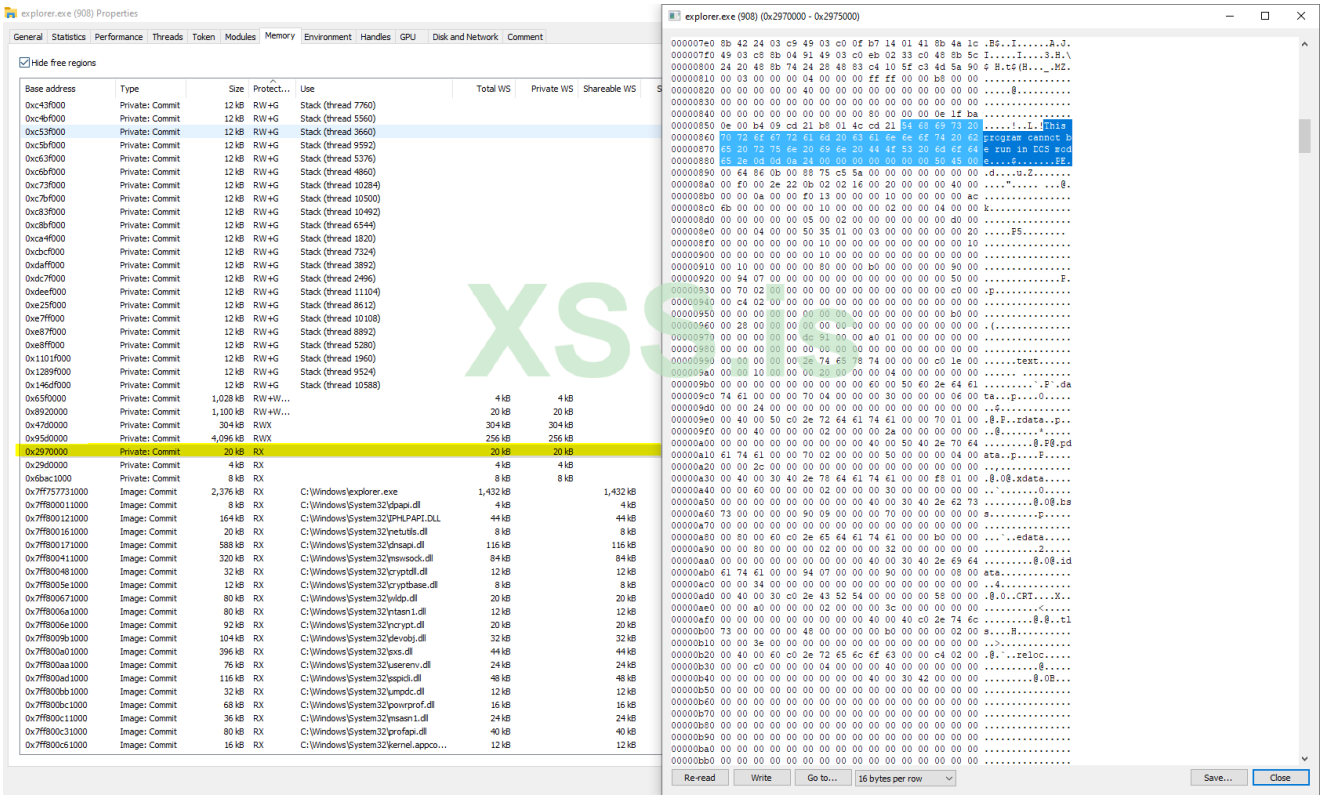


Проверка подгруженных модулей в `spoolsv.exe`:



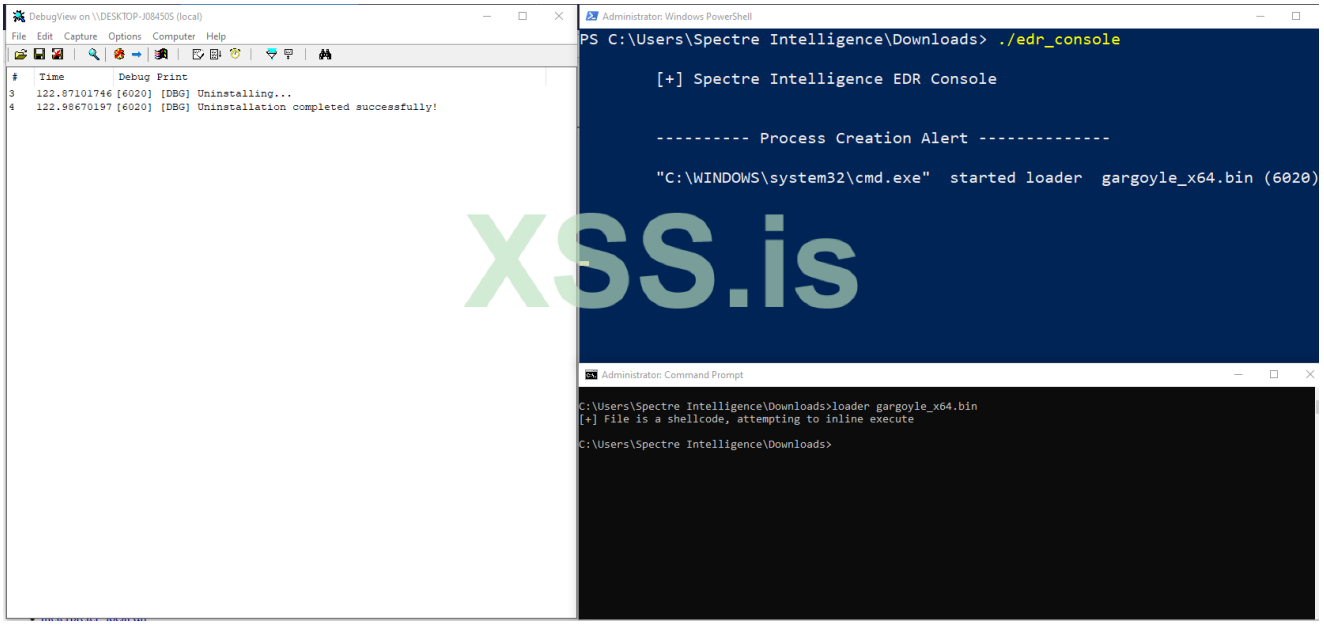
В таблице импорта есть довольно *подозрительные* функции. Интересно, что это за модуль?)

Фактический используемый шеллкод:



Надо же, знакомый PE DOS-stub!

Деинсталируем persistence:



Заключение

Если вы еще читаете, то я хочу поблагодарить вас за прочтение всей статьи. Надеюсь, вы с удовольствием читали её так же, как и я работал над фреймворком и параллельным написанием этого поста в блоге.

Имейте в виду, что фреймворк был спроектирован в **модульной структуре**, это значит, что желающие могут без затруднений **смешивать и сопоставлять другие техники**, **сохраняя архитектуру без изменений**, ну или просто относится к фреймворку **как к модулю и использовать его в своих проектах**.

Если вы считаете, что что-то можно улучшить, или у вас есть некоторые идеи или вопросы, то просто напишите мне!

Bene vale operator!

Ссылки:
Фреймворк: https://github.com/slaeryan/MIDNIGHTTRAIN
Оригинальная статья: https://slaeryan.github.io/posts/midnighttrain.html
Автор статьи: https://twitter.com/slaeryan/
Блог автора: https://slaeryan.github.io/

Перевод:
atavism, *специально для xss.is*
Это мой первый перевод. Пожалуйста, напишите в тему или мне в ЛС по поводу замечаний и предложений.