

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

---

 [xss.is/threads/43431](https://xss.is/threads/43431)

*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!) ЦПУ. Который был зарегистрирован как обычный **Print Monitor** для достижения устойчивости на хосте (отсюда и название - **Default Print Monitor**).

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 "перечислить" 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 привелегии, которые доступны и включены в токене:

`SeSystemEnvironmentPrivilege/SE_SYSTEM_ENVIRONMENT_NAME` . Это значит, что наш фреймворк не сможет установиться 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 (ringo), используя такие фреймворки, как *CHIPSEC* или физический доступ к машине с **UEFI**.

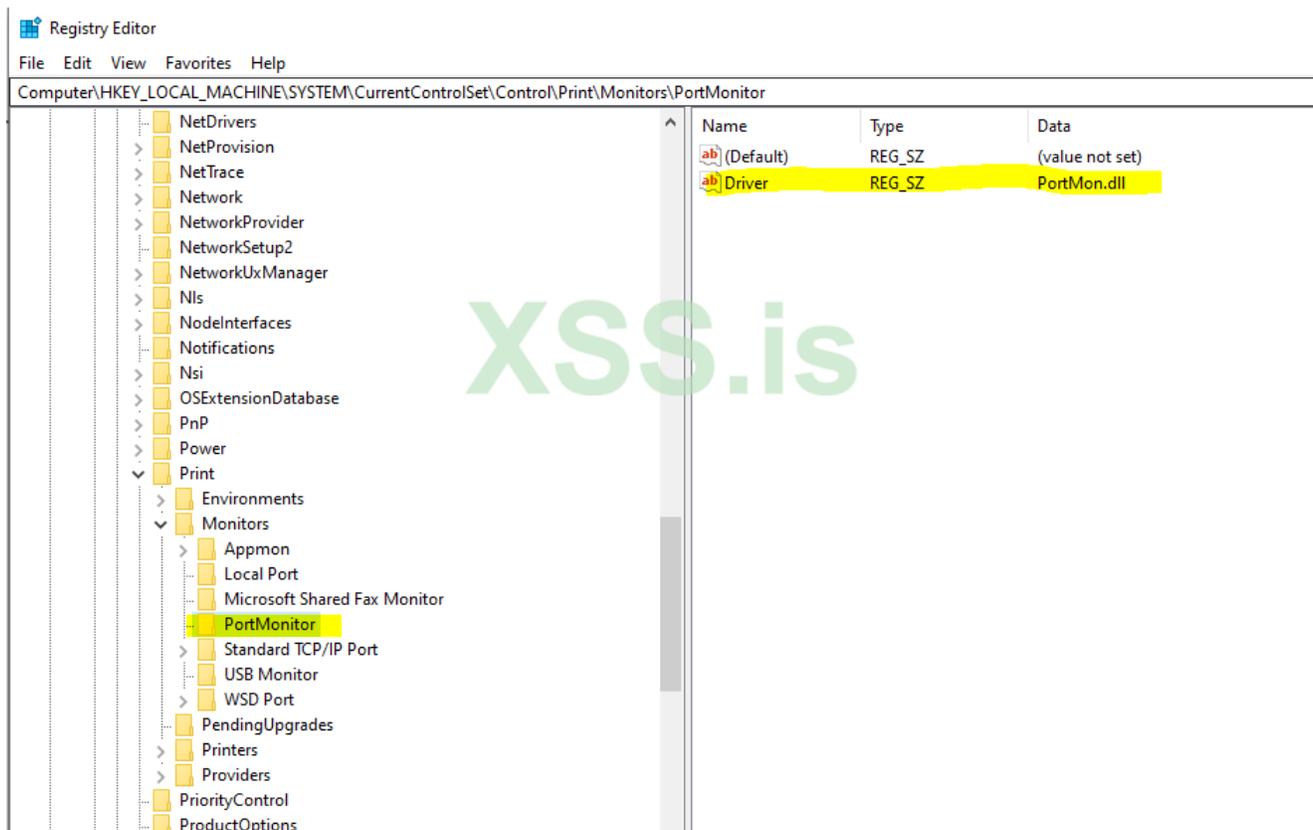
## Port Monitors

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 выглядеть 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 находиться 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 деинсталляция **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

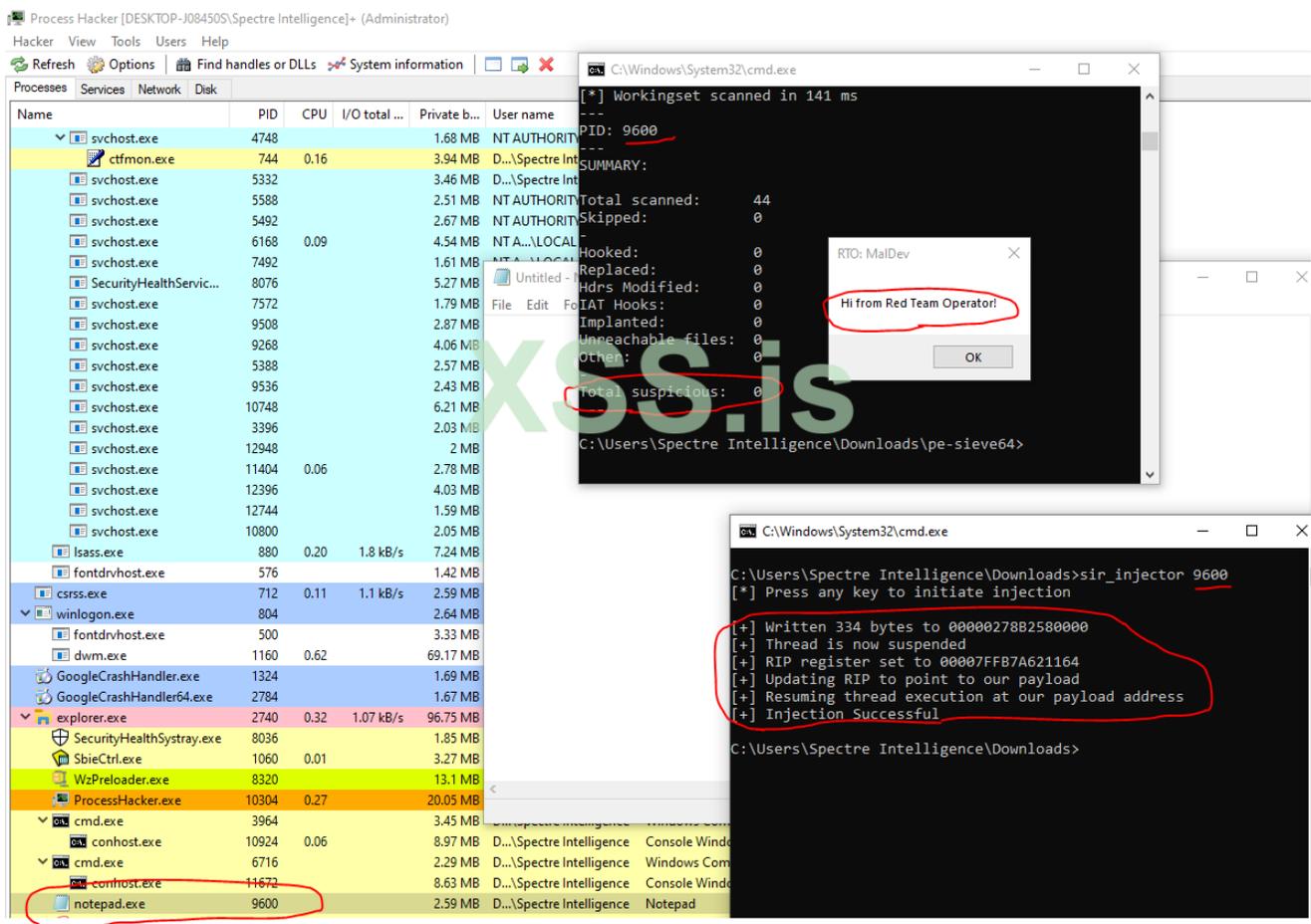
Типичная инъекция кода использует инъекцию в целевой процесс с помощью задокументированного `CreateRemoteThread()` или менее документированного `RtlCreateUserThread()`, или эквивалентный `NtCreateThreadEx()`.

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 инъектирования 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 инъектиться 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.

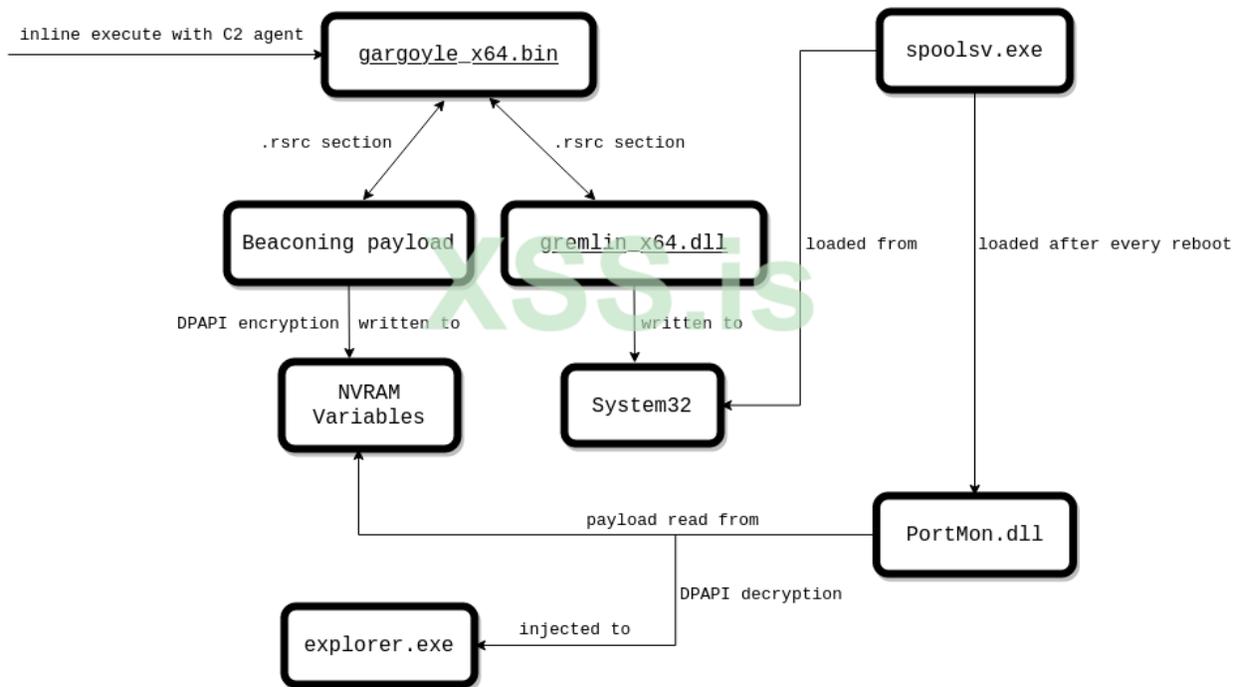
Последнее, что стоит сказать, так это то, что такой метод не очень-то и стабильный, следовательно есть шанс сбоя целевого процесса после завершения нашего вредоносного кода (для фикса требуется клин-ап)

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 твоё "**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 установке 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. Читать 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 буферным 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 буферного 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 лучше? Да, вполне. Один из способов - это использование **Base64URL**, которая является *URL-safe* вариантом **Base64**, еще один плюс, что она опускает символ "=".

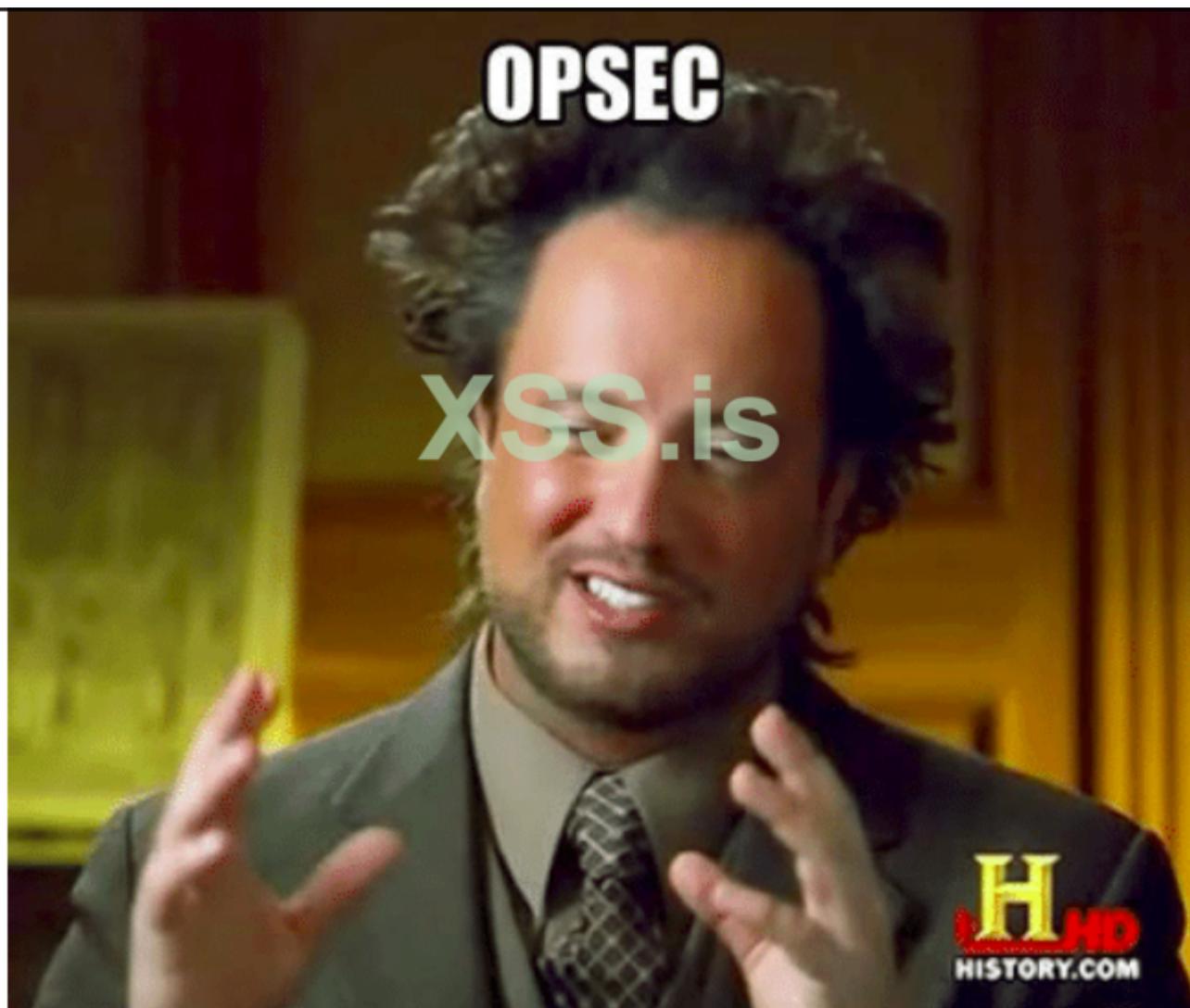
Так какой максимальный размер хранения **payload** в **NVRAM переменных**? Где-то **~36КБ**. Сразу же понятно, что с таким размером не может и идти речи о **stageless payload**, генерируемой как "out-of-the-box".

Так чем мы можем воспользоваться? **Staged payload** генерируемый 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 инжектировать его снова, так как наш implant уже будет находиться в адресном пространстве процесса, откуда сетевая активность уже не распознается 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, **persistence** должен 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 находиться 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 память **egress 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 привелегий, 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 привелегий 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 привелегии уже в своем токене (хоть и в отключенном состоянии) для вызывающего потока, и только после этого попытаться включить нужную привелегию.

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

## Скриншоты

Время скриншотов!

Устанавливаем **persistence**:

```
File Edit Capture Options Computer Help
# Time Debug Print
1 0.00000000 [6396] [DBG] Installing...
2 0.01248750 [6396] [DBG] Payload length: 19473
3 0.06999530 [3400] [DBG] Gremlin loaded
4 3.43249011 [6396] [DBG] Installation completed successfully!
5 6.09002254 [3400] [DBG] SeSystemEnvironmentPrivilege has been enabled
6 6.09580755 [3400] [DBG] Chunk No. read: 1
7 6.10161701 [3400] [DBG] Chunk No. read: 2
8 6.10765743 [3400] [DBG] Chunk No. read: 3
9 6.11388826 [3400] [DBG] Chunk No. read: 4
10 6.12018533 [3400] [DBG] Chunk No. read: 5
11 6.12652254 [3400] [DBG] Chunk No. read: 6
12 6.13266661 [3400] [DBG] Chunk No. read: 7
13 6.13877535 [3400] [DBG] Chunk No. read: 8
14 6.14487423 [3400] [DBG] Chunk No. read: 9
15 6.15117455 [3400] [DBG] Chunk No. read: 10
16 6.15743494 [3400] [DBG] Chunk No. read: 11
17 6.16374540 [3400] [DBG] Chunk No. read: 12
18 6.17002535 [3400] [DBG] Chunk No. read: 13
19 6.17642498 [3400] [DBG] Chunk No. read: 14
20 6.18274641 [3400] [DBG] Chunk No. read: 15
21 6.18918657 [3400] [DBG] Chunk No. read: 16
22 6.19559303 [3400] [DBG] Chunk No. read: 17
23 6.20219278 [3400] [DBG] Chunk No. read: 18
24 6.20872498 [3400] [DBG] Chunk No. read: 19
25 6.21536732 [3400] [DBG] Chunk No. read: 20
26 6.22204733 [3400] [DBG] Chunk No. read: 21
27 6.22877502 [3400] [DBG] Chunk No. read: 22
28 6.23564945 [3400] [DBG] Chunk No. read: 23
29 6.24244070 [3400] [DBG] Chunk No. read: 24
30 6.24927902 [3400] [DBG] Chunk No. read: 25
31 6.25623703 [3400] [DBG] Chunk No. read: 26
32 6.26315260 [3400] [DBG] Chunk No. read: 27
33 6.26714659 [3400] [DBG] Payload read from NVRAM
34 6.27479219 [3400] [DBG] Payload decrypted successfully
35 6.27487898 [3400] [DBG] Payload length: 19473
36 6.32187850 [3400] [DBG] Payload injected successfully
```

```
Administrator: Windows PowerShell
PS C:\Users\Spectre Intelligence\Downloads> ./edr_console

[+] Spectre Intelligence EDR Console

----- Process Creation Alert -----

"C:\WINDOWS\system32\cmd.exe" started loader gargoyle_x64.bin (6396)
```

```
Administrator: Command Prompt
C:\Users\Spectre Intelligence\Downloads> loader gargoyle_x64.bin
[+] File is a shellcode, attempting to inline execute
C:\Users\Spectre Intelligence\Downloads>
```

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

Мы успешно словили входящий `beacon`:

The image shows a screenshot of the Cobalt Strike interface. At the top, there is a menu bar with 'Cobalt Strike', 'View', 'Attacks', 'Reporting', and 'Help'. Below the menu is a toolbar with various icons. The main window displays a table of active listeners:

external	internal	listener	user	computer	note	process	pid	arch	last
192.168.1.10		local - https	Spectre Intelligenc...	DESKTOP-J08450S		explorer.exe	908	x64	2m

Below this table, there is a large green watermark that reads 'XSS.is'. At the bottom of the interface, there is a window titled 'Event Log X' with tabs for 'Listeners X' and 'Beacon [redacted] @908 X'. The 'Beacon' tab is active, showing a Sysmon EventLog window with the following content:

```
0 0 [System Process]
4 0 System x64 0
120 4 Registry x64 0 NT AUTHORITY\SYSTEM
364 1576 cmd.exe x64 1 DESKTOP-J08450S\Spectre Intelligence
480 4 smss.exe x64 0 NT AUTHORITY\SYSTEM
584 768 WUDFHost.exe x64 0 NT AUTHORITY\LOCAL SERVICE
620 604 csrss.exe x64 0 NT AUTHORITY\SYSTEM
688 860 fontdrvhost.exe x64 1 Font Driver Host\UMFD-1
696 604 wininit.exe x64 0 NT AUTHORITY\SYSTEM
708 688 csrss.exe x64 1 NT AUTHORITY\SYSTEM
768 696 services.exe x64 0 NT AUTHORITY\SYSTEM
792 696 lsass.exe x64 0 NT AUTHORITY\SYSTEM
860 688 winlogon.exe x64 1 NT AUTHORITY\SYSTEM
908 7308 explorer.exe x64 1 DESKTOP-J08450S\Spectre Intelligence
980 768 svchost.exe x64 0 NT AUTHORITY\LOCAL SERVICE
988 768 svchost.exe x64 0 NT AUTHORITY\SYSTEM
1012 768 svchost.exe x64 0 NT AUTHORITY\SYSTEM
1020 696 fontdrvhost.exe x64 0 Font Driver Host\UMFD-0
1032 768 svchost.exe x64 0 NT AUTHORITY\NETWORK SERVICE
1080 768 svchost.exe x64 0 NT AUTHORITY\SYSTEM
```

At the bottom of the Event Log window, it says '[DESKTOP-J08450S] Spectre Intelligence \*/908 (x64)' and 'Last: 2m'. The command prompt shows 'beacon>'.

Проверка подгруженных модулей в `spoolsv.exe`:

spoolsv.exe (3400) Properties

General Statistics Performance Threads Token Modules Memory Environment Handles Services GPU Disk and Network Comment

Name	Base address	Size	Description
localspl.dll.mui	0x1960000	80 kB	Local Spooler DLL
msasn1.dll	0x7f800c1...	72 kB	ASN.1 Runtime APIs
msvcp_win.dll	0x7ff801b6...	632 kB	Microsoft® C Runtime Library
msvcrt.dll	0x7ff80354...	632 kB	Windows NT CRT DLL
mswsock.dll	0x7ff80041...	412 kB	Microsoft Windows Sockets ...
msxml6.dll	0x7ffff82c...	2.35 MB	MSXML 6.0
msxml6r.dll	0x2060000	4 kB	XML Resources
netutils.dll	0x7ff80016...	48 kB	Net Win32 API Helpers DLL
nsi.dll	0x7ff8026c...	32 kB	NSI User-mode interface DLL
ntdll.dll	0x7ff803d0...	1.94 MB	NT Layer DLL
ole32.dll	0x7ff8028e...	1.34 MB	Microsoft OLE for Windows
oleaut32.dll	0x7ff80331...	788 kB	OLEAUT32.DLL
PortMon.dll	0x7ffef30...	44 kB	
powrprof.dll	0x7ff800bc...	296 kB	Power Profile Helper DLL
PrintIsolationPr...	0x7ffffb07...	76 kB	Print Sandbox COM Proxy S...
profapi.dll	0x7ff800c3...	140 kB	User Profile Basic API
propsys.dll	0x7ffffd27...	960 kB	Microsoft Property System
rasadhlp.dll	0x7ffff7c2...	40 kB	Remote Access AutoDial Hel...
rprct4.dll	0x7ff80252...	1.13 MB	Remote Procedure Call Runt...
rsaenh.dll	0x7fffff60...	204 kB	Microsoft Enhanced Cryptog...
sechost.dll	0x7ff8034a...	604 kB	Host for SCM/SDDL/LSA Loo...
secur32.dll	0x7ffff341...	48 kB	Security Support Provider In...
setupapi.dll	0x7ff802ab...	4.44 MB	Windows Setup API
sfc_os.dll	0x7ffffb09...	68 kB	Windows File Protection
SHCore.dll	0x7ff803b9...	676 kB	SHCORE
snmpapi.dll	0x7ffffa4a...	48 kB	SNMP Utility Library
SortDefault.nls	0x19a0000	3.21 MB	
spoolss.dll	0x7ffff26...	112 kB	Spooler SubSystem DLL
spoolsv.exe.mui	0x17b0000	4 kB	Spooler SubSystem App
srvcli.dll	0x7ffef12...	152 kB	Server Service Client DLL
sspicli.dll	0x7ff800ad...	188 kB	Security Support Provider In...
tcpmon.dll	0x7ffffa4b...	240 kB	Standard TCP/IP Port Monit...

C:\Windows\System32\PortMon.dll Properties

General Imports Exports

DLL	Name	Hint
KERNEL32.dll	CloseHandle	134
KERNEL32.dll	CreateThread	242
KERNEL32.dll	CreateToolhelp32Snapshot	251
KERNEL32.dll	GetCurrentThread	545
KERNEL32.dll	GetLastError	615
KERNEL32.dll	GetModuleHandleA	635
KERNEL32.dll	GetProcAddress	693
KERNEL32.dll	GetThreadContext	766
KERNEL32.dll	IstrcmpiA	1605
KERNEL32.dll	OpenProcess	1040
KERNEL32.dll	OpenThread	1047
KERNEL32.dll	OutputDebugStringA	1051
KERNEL32.dll	RaiseException	1126
KERNEL32.dll	ResumeThread	1233
KERNEL32.dll	RtlPcToFileHeader	1244
KERNEL32.dll	SetThreadContext	1377
KERNEL32.dll	Sleep	1419
KERNEL32.dll	SuspendThread	1427
KERNEL32.dll	Thread32First	1450
KERNEL32.dll	Thread32Next	1451
KERNEL32.dll	VirtualAllocEx	1494
KERNEL32.dll	VirtualProtectEx	1500
KERNEL32.dll	WriteProcessMemory	1578

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

Фактический используемый шеллкод:

The image shows two windows from a Windows system. The left window is the 'explorer.exe (908) Properties' dialog, specifically the 'Memory' tab. It displays a table of memory regions with columns for Base address, Type, Size, Protection, Use, Total WS, Private WS, and Shareable WS. The entry for 0x2970000 is highlighted in yellow, showing a size of 20 KB and protection of RW. The right window is a hex dump of memory at address 0x2970000, showing a sequence of bytes that appears to be a shellcode payload.

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable WS
0xc43f000	Private: Commit	12 KB	RW+G	Stack (thread 7760)			
0xc4b6f00	Private: Commit	12 KB	RW+G	Stack (thread 5560)			
0xc53f000	Private: Commit	12 KB	RW+G	Stack (thread 3660)			
0xc5b6f00	Private: Commit	12 KB	RW+G	Stack (thread 5992)			
0xc63f000	Private: Commit	12 KB	RW+G	Stack (thread 5376)			
0xc6b6f00	Private: Commit	12 KB	RW+G	Stack (thread 4860)			
0xc73f000	Private: Commit	12 KB	RW+G	Stack (thread 10284)			
0xc7b6f00	Private: Commit	12 KB	RW+G	Stack (thread 10500)			
0xc83f000	Private: Commit	12 KB	RW+G	Stack (thread 10492)			
0xc8b6f00	Private: Commit	12 KB	RW+G	Stack (thread 6344)			
0xc93f000	Private: Commit	12 KB	RW+G	Stack (thread 10108)			
0xc9b6f00	Private: Commit	12 KB	RW+G	Stack (thread 7324)			
0xda3f000	Private: Commit	12 KB	RW+G	Stack (thread 3892)			
0xdab6f00	Private: Commit	12 KB	RW+G	Stack (thread 2496)			
0xdbeef00	Private: Commit	12 KB	RW+G	Stack (thread 11104)			
0xdceef00	Private: Commit	12 KB	RW+G	Stack (thread 8612)			
0xe29f000	Private: Commit	12 KB	RW+G	Stack (thread 10108)			
0xe29f000	Private: Commit	12 KB	RW+G	Stack (thread 8892)			
0xe29f000	Private: Commit	12 KB	RW+G	Stack (thread 5280)			
0x1101000	Private: Commit	12 KB	RW+G	Stack (thread 1960)			
0x1289f00	Private: Commit	12 KB	RW+G	Stack (thread 9524)			
0x146f000	Private: Commit	12 KB	RW+G	Stack (thread 9524)			
0x5f000	Private: Commit	1,028 KB	RW+H...		4 KB	4 KB	
0x8920000	Private: Commit	1,100 KB	RW+H...		20 KB	20 KB	
0x4700000	Private: Commit	304 KB	RWX		304 KB	304 KB	
0x9500000	Private: Commit	4,096 KB	RWX		256 KB	256 KB	
0x2970000	Private: Commit	20 KB	RW		20 KB	20 KB	
0x2900000	Private: Commit	4 KB	RX		4 KB	4 KB	
0x5bac1000	Private: Commit	8 KB	RX		8 KB	8 KB	
0x7f7573731000	Image: Commit	2,376 KB	RX	C:\Windows\explorer.exe	1,432 KB		1,432 KB
0x7f800011000	Image: Commit	8 KB	RX	C:\Windows\System32\ipapi.dll	4 KB		4 KB
0x7f8000121000	Image: Commit	164 KB	RX	C:\Windows\System32\PHPAPI.DLL	44 KB		44 KB
0x7f8000161000	Image: Commit	20 KB	RX	C:\Windows\System32\netutils.dll	8 KB		8 KB
0x7f8000171000	Image: Commit	588 KB	RX	C:\Windows\System32\chrsapi.dll	116 KB		116 KB
0x7f8000411000	Image: Commit	320 KB	RX	C:\Windows\System32\mwssock.dll	84 KB		84 KB
0x7f8000481000	Image: Commit	32 KB	RX	C:\Windows\System32\cryptui.dll	12 KB		12 KB
0x7f80005e1000	Image: Commit	12 KB	RX	C:\Windows\System32\cryptbase.dll	8 KB		8 KB
0x7f8000671000	Image: Commit	80 KB	RX	C:\Windows\System32\wldap.dll	20 KB		20 KB
0x7f80006a1000	Image: Commit	80 KB	RX	C:\Windows\System32\ntasn1.dll	12 KB		12 KB
0x7f80006e1000	Image: Commit	92 KB	RX	C:\Windows\System32\ncrypt.dll	20 KB		20 KB
0x7f80009b1000	Image: Commit	104 KB	RX	C:\Windows\System32\devobj.dll	32 KB		32 KB
0x7f8000aa1000	Image: Commit	396 KB	RX	C:\Windows\System32\psapi.dll	44 KB		44 KB
0x7f8000aa1000	Image: Commit	76 KB	RX	C:\Windows\System32\userenv.dll	24 KB		24 KB
0x7f8000ad1000	Image: Commit	116 KB	RX	C:\Windows\System32\espapi.dll	48 KB		48 KB
0x7f8000bb1000	Image: Commit	32 KB	RX	C:\Windows\System32\lumpdc.dll	12 KB		12 KB
0x7f8000bc1000	Image: Commit	68 KB	RX	C:\Windows\System32\powrprof.dll	16 KB		16 KB
0x7f8000c11000	Image: Commit	38 KB	RX	C:\Windows\System32\msasn1.dll	24 KB		24 KB
0x7f8000c31000	Image: Commit	80 KB	RX	C:\Windows\System32\profapi.dll	40 KB		40 KB
0x7f8000c51000	Image: Commit	16 KB	RX	C:\Windows\System32\kernel.appo...	12 KB		12 KB

Надо же, знакомый PE DOS-stub!

Деинсталируем persistence:

The image shows a Windows command prompt window with the following text:

```

PS C:\Users\Spectre Intelligence\Downloads> ./edr_console

[+] Spectre Intelligence EDR Console

----- Process Creation Alert -----

"C:\WINDOWS\system32\cmd.exe" started loader gargoyle_x64.bin (6020)
  
```

Below the command prompt, there is a screenshot of a DebugView window showing the following log entries:

```

# Time Debug Print
3 122.87101746 [6020] [DBG] Uninstalling...
4 122.98670197 [6020] [DBG] Uninstallation completed successfully!
  
```

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 относиться 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!

**Vene 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.