Статья План обхода ведущей в отрасли защиты конечных точек в 2022 году

Около двух лет назад я перестал быть штатным оператором красной команды. Тем не менее, это по-прежнему область знаний, которая остается очень близкой моему сердцу. Несколько недель назад я искал новый побочный проект и решил заняться своим старым хобби: обход/обход решений для защиты конечных точек. В этом посте я хотел бы изложить набор методов, которые вместе можно использовать для обхода ведущих в отрасли решений для защиты конечных точек предприятия. Это исключительно в образовательных целях для (этичных) красных команд и им подобных, поэтому я решил не публиковать исходный код. Цель этого поста — сделать его доступным для широкой аудитории в индустрии безопасности, а не углубляться в мельчайшие детали каждой техники. Вместо этого я буду ссылаться на записи других людей, которые погружаются глубже, чем я.
При моделировании злоумышленников ключевой проблемой на этапе «начального доступа» является обход возможностей обнаружения и реагирования (EDR) на конечных точках предприятия. Коммерческие системы управления и контроля предоставляют немодифицируемый шелл-код и двоичные файлы оператору красной команды, которые сильно подписаны индустрией защиты конечных точек, и для того, чтобы выполнить этот имплантат, сигнатуры (как статические, так и поведенческие) этого шелл-кода должны быть запутаны.

В этом посте я расскажу о следующих методах, конечной целью которых является выполнение вредоносного шелл-кода, также известного как загрузчик (шелл-кода):

1. Шифрование шелл-кода
2. Снижение энтропии
3. Побег из (локальной) песочницы AV
4. Обфускация таблицы импорта
5. Отключение отслеживания событий для Windows (ETW)
6. Уклонение от распространенных шаблонов вредоносных вызовов API
7. Прямые системные вызовы и уклонение от «знака системного вызова»
8. Удаление крючков в ntdll.dll
9. Подмена стека вызовов потока
10. Шифрование маяка в памяти
11. Пользовательский отражающий загрузчик
12. Конфигурации OpSec в вашем гибком профиле

1. Шифрование шелл-кода

Давайте начнем с простой, но важной темы — обфускации статического шеллкода. В моем загрузчике я использую алгоритм шифрования XOR или RC4, потому что его легко реализовать и он не оставляет много внешних индикаторов операций шифрования, выполняемых загрузчиком. Шифрование AES для запутывания статических подписей шелл-кода оставляет следы в таблице адресов импорта бинарника, что увеличивает подозрительность. У меня был Windows Defender, который специально запускал функции дешифрования AES (например, **CryptDecrypt**, **CryptHashData**, **CryptDeriveKey** и т.д.) в более ранних версиях этого загрузчика.

```
ADVAPI32.dll
         140004000 Import Address Table
         140005930 Import Name Table
                 0 time date stamp
                 0 Index of first forwarder reference

                      C2 CryptAcquireContextW
                      C6 CryptDeriveKey
                      C7 CryptDestroyHash
                      D9 CryptHashData
                      C8 CryptDestroyKey
                      C4 CryptCreateHash
                      C5 CryptDecrypt
                      DC CryptReleaseContext
```

XSS.is

Вывод dumpbin/imports, простая раздача только функций дешифрования AES, используемых в двоичном файле

2. Снижение энтропии

Многие решения AV/EDR учитывают бинарную энтропию при оценке неизвестного бинарного файла. Так как мы шифруем шелл-код, энтропия нашего бинарника довольно высока, что является явным индикатором запутанных частей кода в бинарнике.
Есть несколько способов уменьшить энтропию нашего двоичного файла, два простых, которые работают:

1. Добавление ресурсов с низкой энтропией в двоичный файл, таких как изображения (с низкой энтропией).

2. Добавление строк, таких как английский словарь или "strings C:\Program Files\Google\Chrome\Application\100.0.4896.88\chrome.dll" выход.

Более элегантным решением было бы разработать и реализовать алгоритм, который бы запутывал (кодировал/шифровал) шелл-код в английские слова (низкая энтропия). Это убило бы двух зайцев одним выстрелом.

3. Побег из (локальной) песочницы AV

Многие решения EDR запускают двоичный файл в локальной песочнице на несколько секунд, чтобы проверить его поведение. Чтобы не ставить под угрозу работу конечного пользователя, они не могут позволить себе проверять двоичный файл дольше нескольких секунд (в прошлом я видел, как yAvast занимало до 30 секунд, но это было исключением). Мы можем злоупотреблять этим ограничением, задерживая выполнение нашего шелл-кода. Вычисление большого простого числа — мой личный фаворит. Вы можете пойти немного дальше и детерминистически вычислить простое число и использовать это число как (часть) ключа к вашему зашифрованному шеллкоду.

4. Обфускация таблицы импорта

Вы хотите, чтобы подозрительный Windows API (WINAPI) не попал в нашу IAT (таблицу адресов импорта). Эта таблица содержит обзор всех API-интерфейсов Windows, которые ваш двоичный файл импортирует из других системных библиотек. Список подозрительных (часто поэтому проверяемых решениями EDR) API можно найти здесь . Как правило, это VirtualAlloc, VirtualProtect, WriteProcessMemory, CreateRemoteThread, SetThreadContextи т.д. dumpbin /exports <binary.exe>перечислит весь импорт. По большей части мы будем использовать прямые системные вызовы, чтобы обойти обе перехватчики EDR (см. раздел 7) подозрительных вызовов WINAPI, но для менее подозрительных вызовов API этот метод работает просто отлично.

Добавляем сигнатуру функции вызова WINAPI, получаем адрес WINAPI в ntdll.dllа затем создайте указатель функции на этот адрес:

Code:

```
typedef BOOL (WINAPI * pVirtualProtect)(LPVOID lpAddress, SIZE_T dwSize, DWORD  flNewProtect, 
PDWORD lpflOldProtect);
pVirtualProtect fnVirtualProtect;
unsigned char sVirtualProtect[] = { 'V','i','r','t','u','a','l','P','r','o','t','e','c','t', 
0x0 };
unsigned char sKernel32[] = { 'k','e','r','n','e','l','3','2','.','d','l','l', 0x0 };
fnVirtualProtect = (pVirtualProtect) GetProcAddress(GetModuleHandle((LPCSTR) sKernel32), 
(LPCSTR)sVirtualProtect);
// call VirtualProtect
fnVirtualProtect(address, dwSize, PAGE_READWRITE, &oldProt);
```

Обфускация строк с использованием массива символов разрезает строку на более мелкие части, что затрудняет их извлечение из двоичного файла.
Звонок по-прежнему будет ntdll.dllWINAPI и не будет обходить перехватчики в WINAPI в ntdll.dll, а чисто для удаления подозрительных функций из IAT.

5. Отключение отслеживания событий для Windows (ETW)

Многие решения EDR широко используют трассировку событий для Windows (ETW), в частности Microsoft Defender для конечной точки (ранее известный как Microsoft ATP). ETW позволяет выполнять обширную инструментальную обработку и отслеживать функциональные возможности процесса и вызовы WINAPI. ETW имеет компоненты в ядре, в основном для регистрации обратных вызовов для системных вызовов и других операций ядра, но также состоит из пользовательского компонента, который является частью ntdll.dll(глубокое погружение ETW и векторы атаки). С ntdll.dll— это DLL, загруженная в процесс нашего двоичного файла, мы имеем полный контроль над этой DLL и, следовательно, над функциональностью ETW. существует довольно пользовательском много различных способов обхода ETW, но наиболее распространенным является исправление функции. EtwEventWriteкоторый вызывается для записи/регистрации событий ETW. Мы получаем его адрес в ntdll.dll, и замените его первые инструкции инструкциями по возврату 0 (SUCCESS).

Code:

```
void disableETW(void) {
// return 0
unsigned char patch[] = { 0x48, 0x33, 0xc0, 0xc3};     // xor rax, rax; ret

ULONG oldprotect = 0;
size_t size = sizeof(patch);

HANDLE hCurrentProc = GetCurrentProcess();

unsigned char sEtwEventWrite[] = { 'E','t','w','E','v','e','n','t','W','r','i','t','e', 0x0
};

void *pEventWrite = GetProcAddress(GetModuleHandle((LPCSTR) sNtdll), (LPCSTR)
sEtwEventWrite);

NtProtectVirtualMemory(hCurrentProc, &pEventWrite, (PSIZE_T) &size, PAGE_READWRITE,
&oldprotect);

memcpy(pEventWrite, patch, size / sizeof(patch[0]));

NtProtectVirtualMemory(hCurrentProc, &pEventWrite, (PSIZE_T) &size, oldprotect, &oldprotect);
FlushInstructionCache(hCurrentProc, pEventWrite, size);

}
```

Я обнаружил, что описанный выше метод по-прежнему работает на двух протестированных EDR, но это зашумленный патч ETW.

6. Уклонение от распространенных шаблонов вредоносных вызовов API

Большая часть поведенческого обнаружения в конечном итоге основана на обнаружении вредоносных шаблонов. Одним из таких шаблонов является порядок конкретных вызовов WINAPI в короткие сроки. Подозрительные вызовы WINAPI, кратко упомянутые в разделе 4, обычно используются для выполнения шелл-кода и поэтому тщательно отслеживаются. Однако эти призывы также используются для доброкачественной деятельности (т. VirtualAlloc, WriteProcess, CreateThread шаблон в сочетании с выделением памяти и записью ~250 КБ шелл-кода), поэтому задача решений EDR состоит в том, чтобы отличить безопасные вызовы от вредоносных. Филип Ольшак написал отличный пост в блоге, используя задержки и меньшие фрагменты выделения и записи памяти, чтобы гармонировать с благоприятным поведением вызовов WINAPI. Короче говоря, его метод корректирует следующее поведение типичного загрузчика шелл-кода:

1. Вместо того, чтобы выделять один большой кусок памяти и напрямую записывать в эту память шелл-код импланта ~250 КБ, выделите небольшие непрерывные блоки памяти, например, <64 КБ, и пометьте их как **NO_ACCES**S. Затем напишите шелл-код размером, аналогичным размеру выделенных страниц памяти.
2. Введите задержки между каждой из вышеупомянутых операций. Это увеличит время, необходимое для выполнения шелл-кода, но также сделает последовательный шаблон выполнения гораздо менее заметным.

Одна загвоздка с этой техникой заключается в том, чтобы убедиться, что вы нашли место в памяти, которое может поместить весь ваш шелл-код в последовательных страницах памяти. от Filip DripLoader реализует эту концепцию.
Созданный мной загрузчик не внедряет шелл-код в другой процесс, а вместо этого запускает шелл-код в потоке в своем собственном пространстве процесса, используя NtCreateThread. Неизвестный процесс (наш двоичный файл де-факто будет иметь низкую распространенность) в другие процессы (как правило, собственные процессы Windows) является подозрительной активностью, которая выделяется (рекомендуется прочитать «Fork&Run — вы в истории»). Гораздо легче раствориться в шуме спокойного выполнения потоков и операций с памятью внутри процесса, когда мы запускаем шелл-код внутри потока в пространстве процесса загрузчика. Недостатком, однако, является то, что любые сбои модулей постэксплуатации также приводят к сбою процесса загрузчика. Методы сохранения, а также запуск стабильных и надежных BOF могут помочь преодолеть этот недостаток.

7. Прямые системные вызовы и уклонение от «отметки системного вызова»

Загрузчик использует прямые системные вызовы для обхода любых встроенных ловушек. ntdll.dllпо EDR. Я не хочу вдаваться в подробности о том, как работают прямые системные вызовы, так как это не является целью этого поста, и об этом было написано много замечательных постов (например , Outflank).
Короче говоря, прямой системный вызов — это прямой вызов WINAPI к эквиваленту системного вызова ядра. Вместо того, чтобы позвонить в ntdll.dll **VirtualAlloc** мы называем его ядерным эквивалентом NtAlocateVirtualMemory определены в ядре Windows. Это здорово, потому что мы обходим любые перехватчики EDR, используемые для мониторинга вызовов (в этом примере) VirtualAlloc определено в ntdll.dll.

Чтобы вызвать системный вызов напрямую, мы получаем идентификатор системного вызова , из которого мы хотим вызвать. ntdll.dll, используйте сигнатуру функции, чтобы поместить в стек правильный порядок и типы аргументов функции, и вызовите syscall <id> инструкцию. Есть несколько инструментов, которые устраивают все это за нас, SysWhispers2 и SysWhisper3 — два отличных примера. С точки зрения уклонения есть две проблемы с прямыми системными вызовами:

1. Ваш двоичный файл заканчивается тем, что syscall инструкция, которую легко обнаружить статически (она же «метка системного вызова», подробнее в « SysWhispers мертв, да здравствует SysWhispers! »).
2. В отличие от безобидного использования системного вызова, который вызывается через его ntdll.dllэквивалентно, обратный адрес системного вызова не указывает на ntdll.dll. Вместо этого он указывает на наш код, из которого мы вызвали системный вызов, который находится в областях памяти за пределами ntdll.dll. Это индикатор системного вызова, который не вызывается через ntdll.dll, что подозрительно.

Чтобы преодолеть эти проблемы, мы можем сделать следующее:

1. Реализовать механизм охотника за яйцами. Заменить syscall инструкции с egg (какой-то случайный уникальный идентифицируемый шаблон) и во время выполнения выполните поиск этого egg в памяти и заменить его на syscall инструкцию с использованием ReadProcessMemoryа также WriteProcessMemory вызовы WINAPI. После этого мы можем использовать прямые системные вызовы в обычном режиме. Этот метод был реализован klezVirus .
2. Вместо того чтобы вызывать инструкцию syscall из собственного кода, мы ищем инструкцию syscall в ntdll.dll и переходим по этому адресу памяти после подготовки стека к вызову системного вызова. В результате в RIP появится адрес возврата, указывающий на области памяти ntdll.dll.
Оба метода являются частью SysWhisper3 .

8. Removing hooks in ntdll.dll

Еще одна хорошая техника для уклонения от хуков EDR. ntdll.dll перезаписать загруженный ntdll.dll который загружается по умолчанию (и перехватывается EDR) со свежей копией из ntdll.dll. ntdll.dll это первая DLL, которая загружается любым процессом Windows. Решения EDR обеспечивают загрузку их DLL вскоре после этого, что ставит все hook's на место в загруженном файле ntdll.dll прежде чем наш собственный код будет выполнен. Если наш код загружает новую копию ntdll.dll впоследствии в памяти эти хуки EDR будут перезаписаны. RefleXXion — это библиотека C++, которая реализует исследования, проведенные MDSec. RelfeXXion

использует прямые системные вызовы NtOpenSection а также NtMapViewOfSection почитстить ntdll.dll в \KnownDlls\ntdll.dll (путь реестра с ранее загруженными DLL). Затем он перезаписывает .TEXT участок загруженного ntdll.dll, который сбрасывает перехватчики EDR.

Я рекомендую настроить библиотеку RefleXXion, чтобы использовать тот же трюк, который описан выше в разделе 7.

9. Подмена стека вызовов потока

Следующие два раздела посвящены двум методам, позволяющим избежать обнаружения нашего шелл-кода в памяти. Из-за маячкового поведения имплантата большую часть времени он спит, ожидая входящих задач от своего оператора. В это время имплантат уязвим для методов сканирования памяти из EDR. Первый из двух способов обхода, описанных в этом посте, — подмена стека вызовов потока.

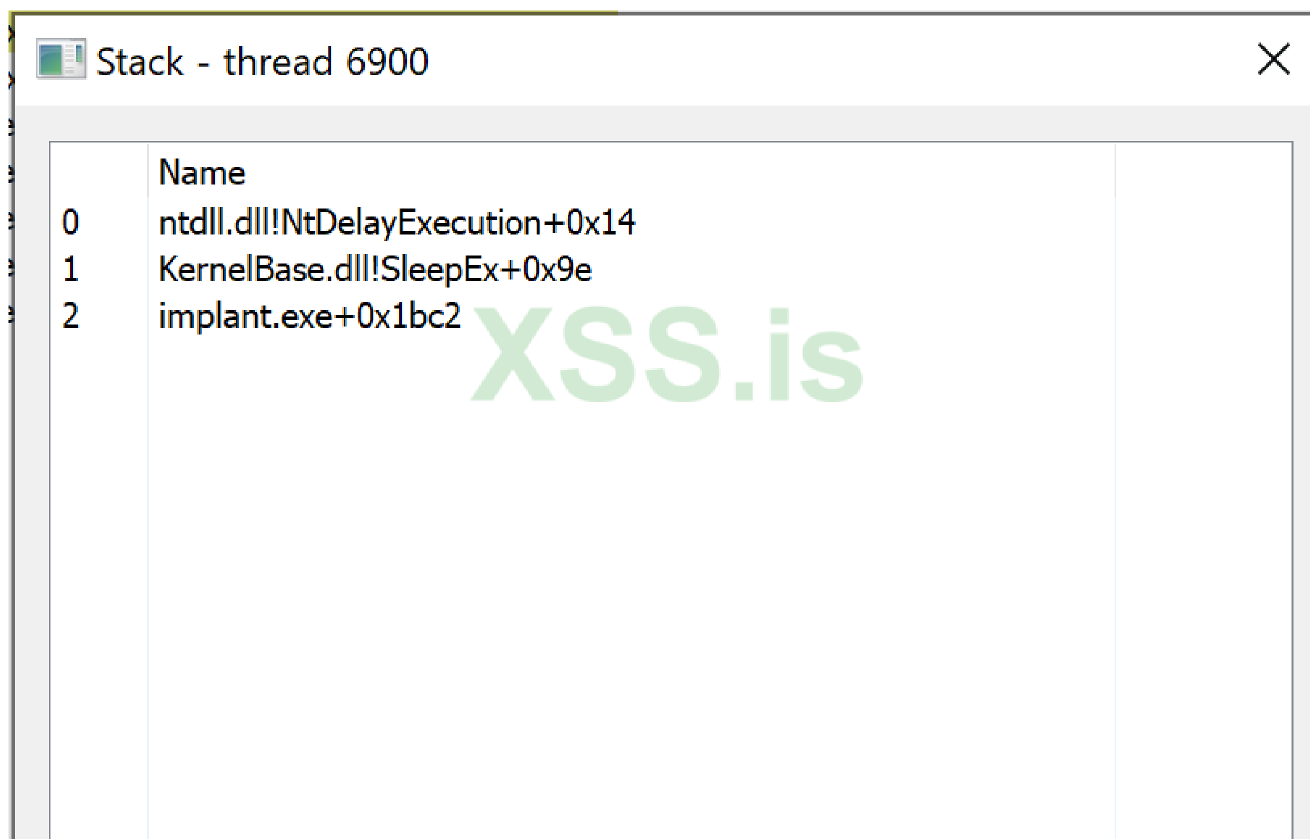
Когда имплант спит, его адрес возврата потока указывает на наш шелл-код, находящийся в памяти. Изучив адреса возврата потоков в подозрительном процессе, наш шелл-код импланта можно легко идентифицировать. Чтобы этого избежать, нужно разорвать эту связь между обратным адресом и шеллкодом. Мы можем сделать это, подключив Sleep() функцию. Когда этот hook вызывается, мы перезаписываем адрес возврата на 0x0 и вызываем Sleep() функцию. Когда Sleep() возвращается, мы возвращаем исходный адрес возврата на место, чтобы поток возвращался по правильному адресу для продолжения выполнения. Мариуш Банах реализовал эту технику в своем ThreadStackSpoofer . В этом репозитории содержится гораздо больше подробностей о технике, а также излагаются некоторые предостережения. Мы можем наблюдать результат подмены стека вызовов потока на двух снимках экрана ниже, где стек вызовов без подделки указывает на неподдерживаемые области памяти, а стек вызовов подделки указывает на наш перехваченный Sleep (MySleep) и «отрезает» остальную часть стека вызовов.

Stack - thread 9444

	Name
0	ntdll.dll!NtWaitForSingleObject+0x14
1	KernelBase.dll!WaitForSingleObjectEx+0x8e
2	wininet.dll!InternetFindNextFileW+0xe5c8
3	wininet.dll!InternetFindNextFileW+0x921f
4	wininet.dll!UrlCacheServer+0x3887c
5	wininet.dll!HttpSendRequestA+0xe5
6	wininet.dll!HttpSendRequestA+0x58
7	0x152f203cdda
8	0xcc000c
9	0x31d7bff7e0

XSS.is

Стек вызовов потока маяка по умолчанию.



Поддельный стек вызовов потока маяка.

10. Шифрование маяка в памяти

Другим способом уклонения от обнаружения в памяти является шифрование исполняемых областей памяти импланта во время Sleep(). Используя тот же хук сна, как описано в разделе выше, мы можем получить сегмент памяти шелл-кода, изучив адрес вызывающего абонента (код-маяк, который вызывает Sleep() и поэтому наш MySleep() hook. Если область памяти вызывающего абонента MEM_PRIVATE а также EXECUTABLE и примерно как размер нашего шеллкода, то сегмент памяти шифруется с помощью функции XOR и Sleep(), затем Sleep() возвращается, он расшифровывает сегмент памяти и возвращается к нему.

Другой метод заключается в регистрации векторного обработчика исключений (VEH), который обрабатывает NO_ACCESS исключения расшифровывает сегменты памяти и изменяет разрешения на RX. Затем непосредственно перед Sleep() пометьте сегменты памяти как NO_ACCESS, так что когда Sleep() возвращает, выдает исключение нарушения доступа к памяти. Поскольку мы зарегистрировали VEH, исключение обрабатывается в контексте этого потока и может быть возобновлено точно в том же

месте, где возникло исключение. VEH может просто расшифровать и изменить разрешения обратно на RX, а имплант может продолжить выполнение. Этот метод предотвращает обнаружение Sleep() hook на месте, когда имплантат sleep()

Мариуш Банах также реализовал эту технику в ShellcodeFluctuation .

11. Пользовательский отражающий загрузчик

Шелл-код маяка, который мы выполняем в этом загрузчике, в конечном итоге представляет собой DLL, которую необходимо выполнить в памяти. Многие C2-фреймворки используют ReflectiveLoader Стивена Фьюера. Существует много хорошо написанных объяснений того, как именно работает отражающий загрузчик DLL, и код Стивена Фьюера также хорошо документирован, но вкратце отражающий загрузчик делает следующее:

1. Преобразовывает адреса в kernel32.dllWINAPI, необходимые для загрузки DLL (например, VirtualAlloc, LoadLibraryA так далее.)
2. Записать DLL и ее разделы в память
3. Создайте таблицу импорта DLL, чтобы DLL могла вызывать ntdll.dll а также kernel32.dllWINAPI
4. Загрузите любые дополнительные библиотеки и разрешите их соответствующие импортированные адреса функций.
5. Вызов точки входа DLL

В Cobalt Strike добавлена поддержка пользовательского способа рефлективной загрузки DLL в память, что позволяет оператору красной команды настраивать способ загрузки DLL-маяка и добавлять методы уклонения. Bobby Cooke и Santiago P создали скрытый загрузчик (BokuLoader), используя UDRL от Cobalt Strike, который я использовал в своем загрузчике. BokuLoader реализует несколько техник уклонения:

- Ограничить вызовы на GetProcAddress() (обычно EDR перехватывает вызов WINAPI для разрешения адреса функции, как мы делаем в разделе 4)
- Обход AMSI и ETW
- Используйте только прямые системные вызовы
- Использовать только RWили RX, и нет RWX(EXECUTE_READWRITE) разрешения
- Удаляет заголовки DLL маяка из памяти

Обязательно раскомментируйте два определения, чтобы использовать прямые системные вызовы через HellsGate и HalosGate и обойти ETW и AMSI (на самом деле

это не обязательно, так как мы уже отключили ETW и не внедряем загрузчик в другой процесс).

12. Конфигурации OpSec в вашем гибком профиле

Убедитесь, что в вашем профиле Malleable C2 настроены следующие параметры, которые ограничивают использование RWXотмеченная память (подозрительная и легко обнаруживаемая) и очистить шелл-код после запуска маяка.

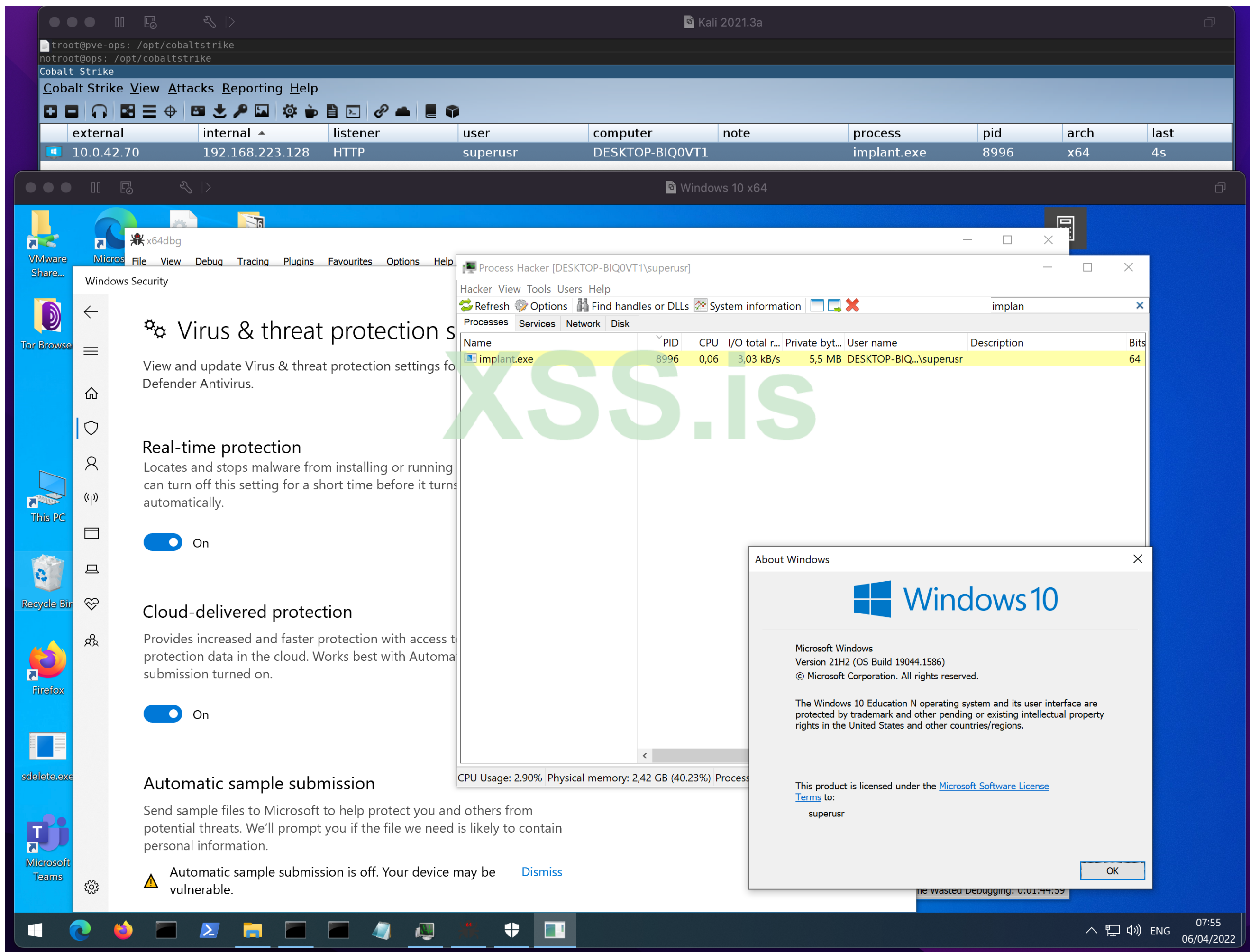
Code:

```
set startrwx        "false";
set userwx          "false";
set cleanup         "true";
set stomppe         "true";
set obfuscate       "true";
set sleep_mask      "true";
set smartinject     "true";
```

Выводы

Сочетание этих методов позволяет вам обойти (среди прочего) Microsoft Defender for Endpoint и CrowdStrike Falcon с 0 обнаружениями (проверено в середине апреля 2022 г.), которые вместе с SentinelOne лидируют в отрасли защиты конечных точек.

CrowdStrike Falcon с 0 оповещениями.



Конечно, это только один первый шаг к полной компрометации конечной точки, и это не означает, что «игра окончена» для решения EDR. В зависимости от того, какие действия/модули после эксплуатации выберет оператор красной команды, для имплантата все еще может быть «игра окончена». В общем, либо запускайте BOF, либо туннелируйте post-ex инструменты через прокси-функцию SOCKS имплантата. Также подумайте о том, чтобы вернуть исправления перехватчиков EDR на место в нашем Sleep()hook, чтобы избежать обнаружения отсоединения, а также удаление исправлений ETW/AMSI.

Это игра в кошки-мышки, и кошке, несомненно, становится лучше.

Перевод вот ЭТОЙ статьи.

-