Статья Разработка вредоносного ПО. Часть 9 - размещение среды CLR и внедрение управляемого кода

Введение

Это 9-й 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. Сегодня мы исследуем методы выполнения управляемого кода из нативного кода.

Примечание: здесь мы, как обычно, работаем с 64-битным кодом.

Неуправляемый и управляемый код

Неуправляемый (собственный) код компилируется непосредственно в ассемблерный код, который интерпретируется процессором. Управляемый код компилируется в некоторое промежуточное представление (байт-код), которое интерпретируется средой выполнения. Среда выполнения также может управлять памятью, собирать мусор и т. д.

Как правило, проще разрабатывать приложения с использованием управляемых языков, таких как Java или .NET, потому что разработчику не нужно беспокоиться о выделении, освобождении памяти и других низкоуровневых вещах. Среда выполнения обеспечивает абстракцию от этих низкоуровневых операций и системного API, иногда даже позволяя кроссплатформенную разработку (например, .NET Core).

Однако иногда разработчики могут извлечь выгоду из прямой интеграции с API операционной системы и памяти, предлагаемой родными языками (например, C/C ++). И, что наиболее важно (с точки зрения разработчика вредоносных программ), нативный код сложнее подвергнуть реверс инжинирингу и предлагает больше возможностей для обфускации.

Хостинг среды выполнения .NET

Управляемая сборка (EXE или DLL) при загрузке через CreateProcess или LoadLibrary интерпретируется загрузчиком Windows, а процедура загрузки инициализирует CLR (Common Language Runtime). В PE-файле есть каталог дескрипторов COM, который содержит метаданные .NET, и его присутствие указывает на тот факт, что PE содержит управляемый код.

Однако можно вручную разместить CLR из собственного приложения с помощью интерфейсов хостинга CLR https://docs.microsoft.com/en-us/dotnet/framework/unmanaged-api/hosting/clr-hosting-interfaces.

mscoree.dll реализует функции, которые мы можем использовать для хостинга CLR. Существует интерфейс ICLRRuntimeHost, который можно использовать для запуска среды выполнения CLR и выполнения кода с диска. Код очень прост:

C:

```
ICLRMetaHost* metaHost = NULL;
CLRCreateInstance(CLSID_CLRMetaHost, IID_ICLRMetaHost, (LPVOID*)&metaHost);
ICLRRuntimeInfo* runtimeInfo = NULL;
metaHost->GetRuntime(L"v4.0.30319", IID_ICLRRuntimeInfo, (LPVOID*)&runtimeInfo);
ICLRRuntimeHost* runtimeHost = NULL;
runtimeInfo->GetInterface(CLSID_CLRRuntimeHost, IID_ICLRRuntimeHost, (LPVOID*)&runtimeHost);
runtimeHost->Start();
DWORD retVal;
CLRRuntimeHost->ExecuteInDefaultAppDomain(L"path_to_assembly", L"Namespace.Class",
L"MethodName", L"argument", &retVal);
```

Согласно документации https://docs.microsoft.com/en-us/do...rruntimehost-executeindefaultappdomain-method, управляемый метод для выполнения должен иметь следующую подпись:

C:

```
static int pwzMethodName (String pwzArgument)
```

Выполнение кода .NET в памяти

Однако не так просто выполнить управляемый код, хранящийся, например, в виде массива байтов в памяти. Для этого мы должны обратиться к устаревшему интерфейсу ICorRuntimeHost. Это обеспечивает больший контроль над управляемой средой выполнения из собственного кода.

Поэтому вместо CLRRuntimeHost нам нужен CorRuntimeHost, и это то, что мы делаем с помощью ICLRRuntimeInfo::GetInterface:

C:

```
ICorRuntimeHost* corRuntimeHost = NULL;
runtimeInfo->GetInterface(CLSID_CorRuntimeHost, IID_ICorRuntimeHost,
(LPVOID*)&corRuntimeHost);
```

Для выполнения управляемым кодом в памяти мы будем использовать управляемые методы и некоторую рефлексию:

C#:

```
Assembly managedAssembly = AppDomain.CurrentDomain.Load(assemblyByteArray);
Type managedType = managedAssembly.GetType("Namespace.Class");
object[] parameters = new object[1] {("Argument_1")};
managedType.InvokeMember("MethodName", BindingFlags.Public | BindingFlags.InvokeMethod |
BindingFlags.Static, null, null, parameters);
```

Но как выполнить этот управляемый код с неуправляемого хоста CLR? Мы будем использовать компонентную объектную модель.

Компоненты .NET можно предоставлять в COM (https://docs.microsoft.com/en-us/dotnet/framework/interop/exposing-dotnet-components-to-com).

Системные сборки .NET framework скомпилированы с включенной функциональной совместимостью COM, поэтому можно вызывать общедоступные методы "вне" среды CLR.

Для начала нам понадобится неуправляемая ссылка (указатель) на AppDomain. Домены приложений - это контейнеры, которые обеспечивают границу изоляции в пределах одного хоста среды выполнения. Вот здесь и пригодится интерфейс CorRuntimeHost - он позволяет получить AppDomain по умолчанию для текущего процесса:

C++:

```
IUnknown* appDomainThunk;
pCorRuntimeHost->GetDefaultDomain(&appDomainThunk);
_AppDomain* defaultAppDomain = NULL;
appDomainThunk->QueryInterface(&defaultAppDomain);
```

На этом этапе мы можем переписать код .NET, используя C++ и COM.

Доступ к mscorlib из C++ с помощью COM

Весь необходимый нам код находится в mscorlib.dll, которая представляет собой управляемую библиотеку общих типов и методов .NET.

Прежде всего, нам нужно каким-то образом «перевести» COM-интерфейсы, определенные в библиотеке mscorlib, в формат, понятный компилятору C++, например, заголовок, определяющий имена, сигнатуры и типы внешних функций (например, импорт, смещение памяти, выравнивание стека для вызовов функций и т.д.)).

Существует инструмент Type Library Exporter (tlbexp.exe) (https://docs.microsoft.com/en-us/dotnet/framework/tools/tlbexp-exe-type-library-exporter) , который мы можем использовать для создания библиотеки типов из управляемого кода. Файлы библиотеки типов должны присутствовать после установки рабочей нагрузки Visual Studio и .NET, но давайте создадим и проанализируем их вручную:

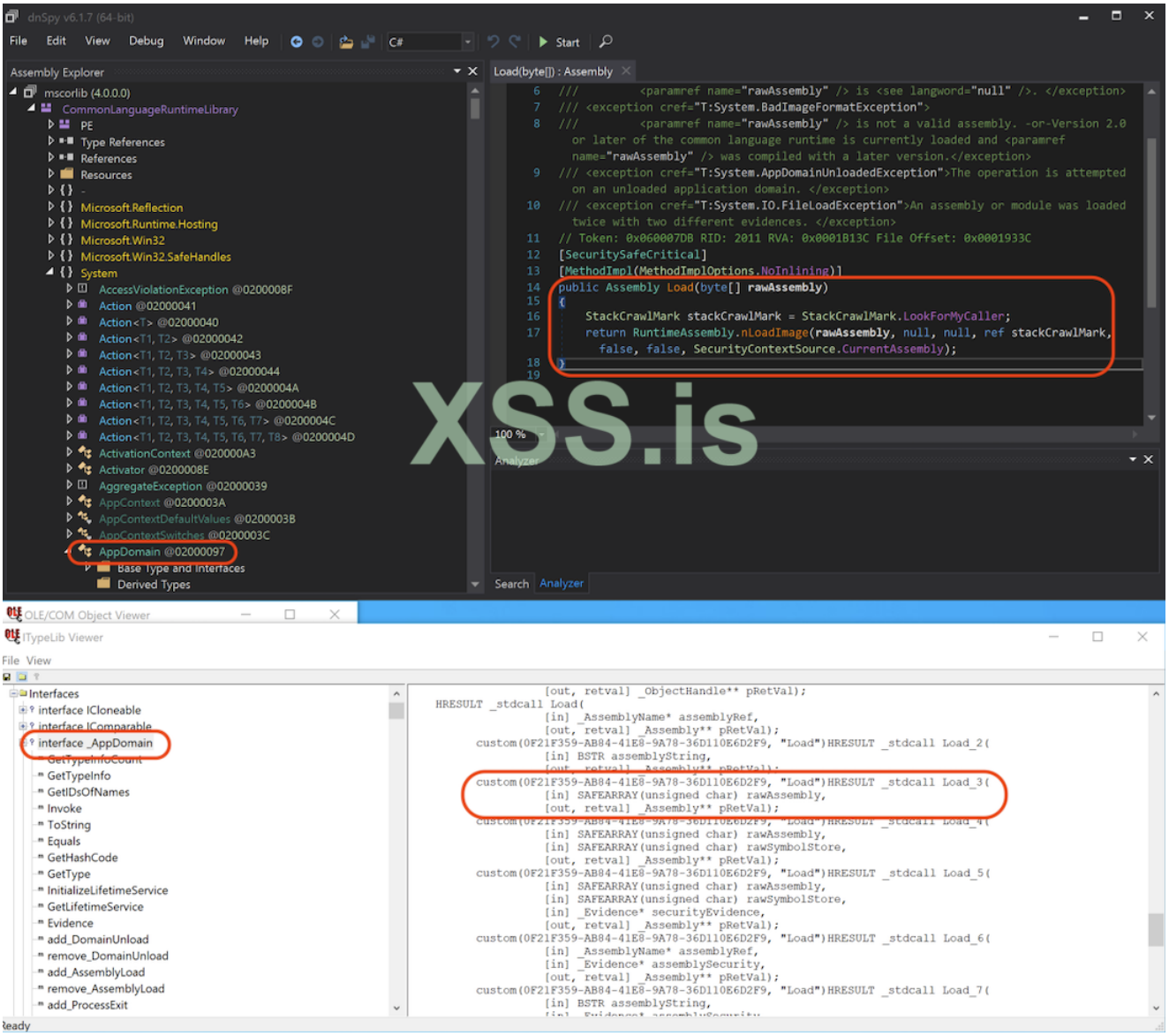
tlbexp.exe mscorlib.dll

.tlb файл можно просмотреть с помощью OLE-COM Object Viewer (oleview.exe, часть Visual C++ SDK)

Давайте посмотрим на код .NET для загрузки кода в памяти:

Assembly managedAssembly =
AppDomain.CurrentDomain.Load(assemblyByteArray);

В нашем случае CurrentDomain - это указатель _AppDomain * defaultAppDomain. Итак, нам нужно найти функцию Load (byte []), экспортированную как интерфейс COM.



Мы видим, что для каждой перегрузки метода AppDomain.Load объявляется отдельная функция интерфейса COM. Мы специально нацелены на перегрузку, которая использует массив байтов, хранящий необработанный код как единственный параметр. Итак, Assembly AppDomain.Load (byte [] rawAssembly) становится HRESULT _stdcall Load_3 ([in] SAFEARRAY (unsigned char) rawAssembly, [out, retval] _Assembly ** pRetVal) ;. Значения возвращаются по ссылке с использованием последнего параметра.

Другое дело, что мы не можем просто предоставить этой функции неуправляемый массив байтов, нам нужно создать SAFEARRAY(https://docs.microsoft.com/en-us/ar...arch/introducing-the-safearray-data-structure). Это еще одна часть сложной автоматизации COM/OLE. Примечание: VT_UI1, используемый в SafeArrayCreate, - это VARTYPE, соответствующий массиву символов/байтов без знака.

C++:

```
SAFEARRAYBOUND bounds[1];
bounds[0].cElements = sizeof (rawAssemblyByteArray);
bounds[0].lLbound = 0;
SAFEARRAY* safeArray = SafeArrayCreate(VT_UI1, 1, bounds);
SafeArrayLock(safeArray);
memcpy(safeArray->pvData, rawAssemblyByteArray, sizeof (rawAssemblyByteArray));
SafeArrayUnlock(safeArray);
_AssemblyPtr managedAssembly = NULL;
```

Наконец, загрузите код в AppDomain:

defaultAppDomain->Load_3(safeArray, &managedAssembly)

Следующим шагом является получение ссылки на тип, определенный в коде. Эта строка:

Type managedType = managedAssembly.GetType("Namespace.Class");

транслируется в:

_TypePtr managedType = NULL;
_bstr_t managedClassName("ManagedApp.Program");
managedAssembly->GetType_2(managedClassName, &managedType);

Теперь давайте создадим массив аргументов. Итак, чтобы сделать это:

object[] parameters = new object[1] {("Argument_1")};

нам нужно снова использовать SAFEARRAY с VARTYPE, соответствующим VARIANT, который является другой структурой данных, специфичной для COM/OLE, используемой, например, для хранения строк:

C++:

```
SAFEARRAY* managedArguments = SafeArrayCreateVector(VT_VARIANT, 0, 1);
_variant_t argument(L"Argument_1");
LONG index = 0;
SafeArrayPutElement(managedArguments, &index, &argument);
```

И последнее - вызвать функцию по ее имени:

managedType.InvokeMember("EntryPoint", BindingFlags.Public | BindingFlags.InvokeMethod | BindingFlags.Static, null, null, parameters);

C++:

```
_bstr_t managedMethodName(L"EntryPoint");
_variant_t managedReturnValue;
_variant_t empty;
managedType->InvokeMember_3(
    managedMethodName,
    static_cast<BindingFlags>(BindingFlags_InvokeMethod | BindingFlags_Static |
BindingFlags_Public),
    NULL, empty, managedArguments, &managedReturnValue);
```

Резюме

Фрагмент кода, который я описал для загрузки кода .NET исключительно в память, известен уже много лет, но мне не удалось найти хорошего объяснения того, как он работает. Надеюсь, кому-нибудь это будет интересно.

Я также потратил некоторое время на попытки использовать ICLRRuntimeHost вместо устаревшего ICorRuntimeHost, но мне не удалось найти простой и элегантный способ выполнить код в памяти, особенно для получения указателя на AppDomain. Мне кажется, что единственно возможным решением было бы загрузить небольшой код, который возвращал бы неуправляемый указатель на управляемую функцию с помощью GetFunctionPointerForDelegate, а затем вызывал бы ее с помощью метода ICLRRuntimeHost::ExecuteInAppDomain(https://docs.microsoft.com/en-us/do...ing/iclrruntimehost-executeinappdomain-method), который вообще плохо документирован. В любом случае, вот пример на stackoverflow.com (https://docs.microsoft.com/en-us/do...ing/iclrruntimehost-executeinappdomain-method), который я не проверял.

Еще одна вещь: не забудьте исправить AMSI, если вы внедряете какую-то "вредоносный" код, потому что AppDomain.Load (byte []) (или _AppDomain :: Load_3 ()) использует AMSI для сканирования двоичного файла на наличие индикаторов вредоносных намерений. Подробнее смотрите (https://github.com/dotnet/runtime/blob/main/src/coreclr/vm/peimagelayout.cpp#L347) этот фрагмент исходного кода среды выполнения .NET.