Windows API for red teamers

i

يعني إيه Windows API؟

Windows API (أو WinAPI) هو مجموعة functions بتطلعها Microsoft عشان تخلّي البرامج تتفاعل مع نظام التشغيل Windows. الـ APIs دي بتدي الـ user-mode applications إمكانية الوصول لمميزات الـ kernel-level من غير ما تكسر حدود الأمان.

الأهداف الرئيسية لـ Windows API

- إخفاء الوصول المباشر لـ hardware والـ system.
- توفير services لـ applications.
- الحفاظ على التوافق بين إصدارات Windows المختلفة.
- تقديم programming interfaces ثابتة لـ GUI، file systems، memory، threading، وغيرها.

الفئتين الرئيسيتين

- **User-Mode APIs**: Functions زي اللي في kernel32.dll، user32.dll، advapi32.dll، وغيرها.
- **Native APIs (NT API)**: Internal system calls في ntdll.dll، بتتصدّر كـ functions بأسماء Nt* و Zw*.

User Mode vs Kernel Mode

User Mode	Kernel Mode
بيشتغل بصلاحيات محدودة	بيشتغل بأعلى صلاحيات (Ring 0)
ما يقدرش يوصل للـ hardware مباشرة	عنده وصول كامل للـ hardware
لو حصل crash، الـ system ما بيقعش	لو حصل crash، بيطلّع BSOD
بيستخدم APIs زي CreateFile	بينفّذ APIs زي NtCreateFile

في الـ Red Team، فهم الفصل ده مهم جدًا للـ evasion، لأن معظم الـ EDRs بتركز على الـ user-mode hooks، خصوصًا في ntdll.dll.

طبقات الـ API: Win32، NT، وال SysCalls

Win32 API (High-Level)

- الأكتر شهرة عند الـ developers.
- Functions زي VirtualAllocEx، OpenProcess، CreateFileW.
- موجودة في user32.dll، kernel32.dll.

NT API (Low-Level)

- موجودة في ntdll.dll.
- بتبدأ بـ Nt أو Zw (زي NtOpenProcess).
- بتكون wrappers مباشرة للـ syscalls.
- بتستخدم في العمليات الـ stealthy، خصوصًا عشان نعدّي الـ EDRs.

Syscalls

- الـ interface الـ low-level بين ntdll.dll والـ kernel.
- في x64 بيستخدم syscall instruction، وفي x86 القديم كان int 0x2e.
- كل syscall ليه System Service Number (SSN)، زي NtOpenProcess = 0x26.
- ممكن نستدعي syscall يدويًا باستخدام inline assembly أو shellcode، ودة بيستخدم كتير في تقنيات direct syscall evasion.

Internal Kernel Prefixes الـNt، Zw، Ig

في الـ Windows internals، الـ low-level functions بتتبع prefixes موحدة بتوّري إزاي الـ function تابع لأي subsystem في الـ kernel. الـ prefixes دي مهمة جدًا لو بتعمل kernel development، reverse engineering، أو security research، خصوصًا لو بتحلل drivers أو بتعمل syscall stubs يدويًا.

الـ functions دي عادةً بتكون موجودة في kernel modules، وبعضها بيوصل لها مباشرة من kernel mode، والبعض التاني بيستدعى بشكل غير مباشر من user mode عن طريق ntdll.dll.

Kernel Prefixes ومعانيها

Prefix	Kernel Component	مثال Function	الغرض
Cm	Configuration Manager	CmRegisterCallbackEx	التفاعل مع الـ registry وتسجيل callbacks
Ex	Executive Layer	ExAllocatePool	وظايف مساعدة أساسية (تخصيص memory، synchronization، إلخ)
Hal	Hardware Abstraction Layer	HalGetAdapter	طبقة abstraction بين الـ OS والـ hardware (I/O، interrupts)
Io	I/O Manager	IoAllocateIrp	إدارة I/O للأجهزة IRPs (I/O requestg packets)
Ke	Kernel Core	KeSetEvent	وظايف kernel أساسية لجدولة threads، إشارات IRQL، events، إلخ
Mm	Memory Manager	MmUnlockPages	إدارة paging، virtual memory، physicalg memory
Ob	Object Manager	ObReferenceObject	إدارة kernel objects زي handles، events، processes، namespacesg
Po	Power Manager	PoSetPowerState	إدارة حالات الطاقة للأجهزة والـ system (sleep، hibernate، إلخ)
Tm	Transaction Manager	TmCommitTransaction	دعم kernel transactions (مستخدم في TxF، TxR)
Nt	Native API (user-mode interface)	NtCreateFile	موجود في ntdll.dll، بيستدعي system services من user mode
Zw	Kernel-mode Native API	ZwCreateFile	مستخدم داخليًا في kernel، بيعدّي بعض فحوصات الأمان

الفرق الرئيسي - Nt vs Zw

*Nt Function	*Zw Function
بتستخدم من user mode	بتستخدم في kernel mode
بتستدعي system services مباشرة عن طريق syscall	بتستدعي نفس الـ service بس ممكن تتخطى فحوصات user-mode
بترجّع NTSTATUS codes بصورة خام	ممكن تعدل أو تغلف قيم الرجوع

ملحوظة: الاتنين غالبًا بيشيروا لنفس عنوان الـ memory، بس بيتصرفوا بشكل مختلف حسب وضع الـ CPU (user vs kernel).

الـ DLLs الرئيسية في استخدام الـ API

DLL	الدور
kernel32.dll	High-level Win32 API (files, memory, threads)
ntdll.dll	NT API / syscall stubs
user32.dll	GUI, input, windows
advapi32.dll	Registry, services, tokens, crypto
gdi32.dll	Graphics Device Interface

ملحوظة: معظم الـ EDRs بتعمل hooks على ntdll.dll، عشان كده الـ bypassing أو استعادة الـ ntdll هي استراتيجية شائعة في الـ Red Team.

الخلاصة

- Windows APIs هي البوابة بين الـ applications والـ OS.
- فهم الفرق بين Win32 API وNative API مهم جدًا للـ stealth.
- الـ Red Teamers بيستخدموا الـ low-level access (ntdll، syscall stubs) عشان يعدّوا الـ detection.
- الـ Nt* functions بتدي تحكم أدق ووصول مباشر للـ kernel objects.
- هنستغل الطبقات دي في الموديولات الجاية باستخدام manual syscall stubs، API hashing، remappingg.

ii

🧠 يعني إيه IAT؟

الـ IAT أو Import Address Table هي structure جوه ملفات الـ PE (يعني ملفات الـ EXE و DLL) بتتخزن فيها عناوين الـ functions اللي البرنامج بيستوردها من DLLs تانية.
يعني مثلًا لو البرنامج بينادي على MessageBoxA أو CreateFileW، هو مش بيروح ينادي على الـ function جوا الـ DLL مباشرة، لأ، هو بيروح للعنوان اللي متخزن جوه الـ IAT.

الفكرة دي اسمها dynamic linking، ودي اللي بتسمح إن العنوان الحقيقي بتاع الـ function يتحدد وقت التشغيل (runtime) بواسطة Windows Loader.

💡 إزاي الـ Loader بيتعامل مع الـ IAT؟

Windows Loader بيعدّي على قايمة الـ imports اللي البرنامج كاتبها في Import Table، ويشوف كل DLL المفروض يستورد منها، زي kernel32.dll أو user32.dll.
بعد كده بيلاقي كل function جوه الـ DLL ويملأ الـ IAT بالعناوين الفعلية للـ functions دي.

لو حد عمل hook للـ IAT (يعني غيّر العناوين دي)، ممكن يوجّه النداءات دي لـ functions تانية خالص — وده اللي بيخلي الـ IAT هدف مشترك للـ debuggers، malware، وحاجات زي API hooking frameworks.

🧠 طب وEAT بقى يعني إيه؟

الـ EAT أو Export Address Table هي اللي بتستخدمها الـ DLLs علشان تعرّض (export) الـ functions بتاعتها للعالم الخارجي — يعني لأي برنامج تاني حابب يستخدمها.

بمعنى تاني: الـ DLL بتقول "أنا عندي functions بالشكل ده، ودي العناوين بتاعتها، اللي عايز يستخدمهم يتفضل".

أي برنامج بيحمّل الـ DLL دي، يقدر يستخدم GetProcAddress() علشان يجيب عنوان function بعينها، أو يسيب الـ loader ي resolve العناوين دي أوتوماتيك.

📊 مقارنة سريعة: IAT vs EAT

Aspect	IAT (Import Address Table)	EAT (Export Address Table)
Purpose	ي resolve عناوين الـ functions اللي البرنامج بيستوردها	ي expose الـ functions اللي DLL بتقدمها
مين بيستخدمه؟	البرامج اللي بتستورد functions	DLLs أو EXEs اللي بتصدر functions
بيتستخدم إمتى؟	وقت ما الـ process بتبدأ	وقت ما الـ process تانية تـ import أو ت load الـ DLL
مين بيعدّله؟	Windows Loader وقت التشغيل أو الـ hookers	بيتحدد وقت ما الـ DLL تتكتب وت compile
Read/Write؟	Writable – وده اللي بيسمح بالـ IAT hooking	غالبًا read-only، بس ممكن يتجاب منه باستخدام GetProcAddress
ليه علاقة بالأمان؟	بيتم استغلاله للـ patchingg hooking	ممكن يتلعب فيه بالـ DLL Proxying أو EAT patching
إزاي بيتجاب؟	Direct memory dereference أو عن طريق الـ loader	باستخدام GetProcAddress أو ordinal
بيكون فين؟	idata section. في ملف الـ PE	edata section. في ملف الـ PE

🧪 مثال عملي سريع

لو أنت كتبت كود بينادي على MessageBoxA، الـ compiler بيعمل IAT entry بتشاور على MessageBoxA.

وقت التشغيل، الـ loader بيروح يستخدم EAT بتاعت user32.dll علشان يجيب عنوان MessageBoxA الحقيقي، ويحطه جوه الـ IAT entry.

بس لو فيه attacker عدّل العنوان ده وخلاه يشاور على function تانية هو كتبها، البرنامج لسه هينادي على نفس العنوان — بس هيكون تحت تحكم الـ attacker.

تعالى ناخد سيناريو كامل من أول ما تكتب برنامج بيستخدم دالة من DLL (زي MessageBoxA) لحد ما البرنامج يشتغل فعليًا، ونشوف الـ Import Address Table (IAT) بيتعامل إزاي خطوة بخطوة.

🧠 السيناريو: من الكود للتنفيذ

1. إنت كتبت برنامج C++ بسيط

```
<include <windows.h#

} ()int main
;MessageBoxA(NULL, "Hello!", "IAT Example", MB_OK)
;return 0
{
```

أنت هنا استخدمت دالة اسمها MessageBoxA من user32.dll.

2. 🛠️ مرحلة الـ Compilation

- الكومبايلر (زي g++) بيحوّل الكود لـ object file (.obj).
- بس الكود لسة مش كامل، لأنك استخدمت دوال من DLLs تانية.
- فبيسيب مكان فاضي للدوال دي، ويقول للـ linker: "هنا في دالة اسمها MessageBoxA، هاتها من user32.dll".

3. 🔗 مرحلة الـ Linking

- الـ linker (زي ld أو link.exe) بيجهز الملف التنفيذي (.exe).
- وبيعمل حاجة مهمة: بيضيف جدول اسمه Import Table، فيه لستة بكل الـ DLLs والدوال اللي البرنامج محتاجهم.
- كمان بيضيف IAT — جدول فيه مؤشرات (pointers) للدوال دي، بس لسة مش معروف العناوين الحقيقية.

يعني:

- في الـ .exe بقى فيه:
  - اسم الـ DLL (user32.dll)
  - اسم الدالة (MessageBoxA)
  - IAT entry فيها مكان فاضي يتملّا بعدين

4. 🧠 البرنامج لسة ما اشتغلش، بس جاهز للتنفيذ

- لحد دلوقتي، IAT معمول وموجود في قسم `idata.` في الـ PE file.
- بس المؤشرات فيه لسة مش واصلة للوظايف الحقيقية.

5. 🧬 مرحلة الـ Loading (أول ما تشغل الـ EXE)

- Windows Loader (اللي بيحمّل البرامج في الذاكرة) يبدأ يشتغل.
- يقرأ Import Table ويشوف إن فيه DLLs محتاجة تتحمّل.
- يحمل user32.dll في الذاكرة (لو مش محمّلة أصلاً).
- يروح لـ EAT (Export Address Table) جوه user32.dll ويدور على MessageBoxA.
- يلاقي عنوانها الحقيقي في الذاكرة.
- يكتب العنوان ده جوه الـ IAT بتاع البرنامج.

يعني:

- IAT entry الخاصة بـ MessageBoxA = دلوقتي فيها pointer حقيقي بيوصل للدالة فعلاً.

6. ▶️ البرنامج يبدأ يشتغل

- لما الكود يوصل للسطر ده:

```
MessageBoxA(...);
```

- الكود بيروح للعنوان اللي في الـ IAT (مش بيكلم الـ DLL مباشرة).
- العنوان ده هو اللي الـ Loader حطه.
- فالبرنامج يشتغل طبيعي ويظهر الـ MessageBox.

🎯 ليه ده مهم؟

- أي حد يغيّر الـ IAT بعد الـ Loader (زي malware أو red team) يقدر يخلي البرنامج يندّه دالة تانية.
- دي اسمها IAT Hooking — أداة قوية جدًا في الهجوم أو الـ debugging.

اعتبارات أمنية

- IAT hooking: دي من أكتر الطرق اللي الـ malware و الـ red teamers بيستخدموها علشان ي redirect نداءات حقيقية (زي ReadFile) ل code خبيث.
- EAT patching أو DLL Proxying: ممكن يتحقن code مخصص عن طريق تعديل الـ Export Table أو تحميل DLL تانية بنفس الـ exports.

ملخص الكلام

- IAT = المستورد بيستخدمه علشان ينادي على functions من DLLs.
- EAT = المصدّر بيستخدمه علشان ي expose الـ functions بتاعته.
- الـ IAT بيعتمد على EAT علشان ي resolve العناوين.
- الاتنين بيشتغلوا مع بعض علشان يخلوا الـ dynamic linking شغال، وعلشان كده هما targets مشهورة جدًا في التحليل الأمني وعمليات الـ red teaming.

iii

إيه هو الـ Syscall؟

التعريف

الـ syscall ده زي جسر بيسمح لبرنامج بيشتغل في الـ user-mode إنه يطلب خدمات من الـ kernel-mode بتاع نظام التشغيل. ليه؟ عشان البرامج اللي في الـ user-mode مينفعش تدخل على ذاكرة محمية أو هاردوير بشكل مباشر. فبيستخدموا الـ syscalls عشان يعملوا حاجات زي:

- تخصيص ذاكرة (Allocate memory)
- فتح ملفات (Open files)
- إنشاء بروسيس أو ثريدز (Create processes or threads)
- الدخول على الـ Windows Registry

في ويندوز، الـ syscall functions بتبقى عادةً بتبدأ بـ Nt أو Zw، زي:

- NtAllocateVirtualMemory
- NtReadVirtualMemory
- NtCreateThreadEx
- ZwCreateFile

يعني فكر في الـ syscall زي إنك بترن على تليفون داخلي في شركة، بتطلب من الكيرنل إنه ينفذلك حاجة معينة.

التحويل من User إلى Kernel

الـ syscalls دي بتشتغل زي بوابات بين Ring 3 (الـ user-mode) وRing 0 (الـ kernel-mode). في ويندوز على أنظمة x64 الحديثة، الأمر اللي بيستخدم للتحويل ده اسمه syscall.

إيه هو الـ Syscall Stub؟

التعريف

الـ syscall stub ده عبارة عن دالة صغيرة - غالبًا موجودة في ملف اسمه ntdll.dll - بتحضّر الـ registers (السجلات) وبعدين بتنفذ الأمر syscall. يعني هي زي وسيط بيسهّل التحويل للـ kernel-mode.

الغرض منه

الـ stub ده بيعمل كام حاجة:

- بيحط رقم الـ syscall number (SSN) في الـ EAX register.
- بينقل أول argument من الـ RCX للـ R10.
- بينفذ الأمر syscall عشان يتنقل من الـ user-mode للـ kernel-mode.

بس في نقطة مهمة: الـ stub ده ممكن يتراقب أو يتعدل بواسطة برامج زي EDRs (Endpoint Detection and Response)، ودي برامج أمان بتحاول تكتشف أي استخدام مشبوه للـ syscalls.

تشريح الـ Syscall Stub

خلينا نشوف مثال لكود الـ syscall stub لدالة NtClose على ويندوز x64، وهنشرحه خطوة خطوة:

```
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
```

الخطوات بالتفصيل:

1. mov eax, 0x12:

- هنا بنحط الـ System Service Number (SSN) بتاع الدالة في الـ EAX.
- الكيرنل بيستخدم الرقم ده عشان يلاقي الدالة الصح في الـ System Service Dispatch Table (SSDT).

2. mov r10, rcx:

- في ويندوز x64، عشان الأمر syscall يشتغل صح، لازم أول أرجومنت يتحط في R10 مش RCX. دي قاعدة عشان التوافق.

3. syscall:

- الأمر ده هو instruction سريعة في الـ CPU، بتحوّل من Ring 3 (اليوزر) لـ Ring 0 (الكيرنل). بتستخدم سجلات خاصة زي IA32_LSTAR.

4. ret:

- لما الـ syscall يخلّص (يعني الكيرنل ينفذ اللي مطلوب)، الأمر ده بيرجّع التحكم للكود اللي في الـ user-mode.

إيه هو الـ SSN (System Service Number)؟

التعريف

الـ SSN ده رقم unique integer بيتربط بكل syscall. الرقم ده بيقول للكيرنل إنه ينفذ أي دالة من الـ System Service Dispatch Table (SSDT).

مثال:

- في إصدار معين من ويندوز، الدالة NtAllocateVirtualMemory ممكن يكون الـ SSN بتاعها 0x18.
- في إصدار تاني، ممكن يكون 0x1A.

يعني الـ SSNs دي بتتغير حسب إصدار ويندوز، وده تفصيلة *مهمة* جدًا لو بتعمل direct syscalls، عشان لو استخدمت رقم غلط، البرنامج هيفشل.

جدول توضيحي

الدالة	SSN (مثال)	الغرض
NtAllocateVirtualMemory	0x18	تخصيص ذاكرة افتراضية
NtReadVirtualMemory	0x3F	قراءة من ذاكرة افتراضية
NtCreateThreadEx	0xB9	إنشاء ثريد جديد
ZwCreateFile	0x55	إنشاء أو فتح ملف

ملاحظة: الـ SSNs دي مجرد أمثلة، لازم تتأكد من الرقم الصحيح حسب إصدار ويندوز اللي بتشتغل عليه.

فهم الـ Syscalls في ويندوز

إيه هو الـ Syscall؟

الـ syscall في ويندوز هو زي وسيلة بتخلّي البرامج اللي شغالة في الـ user-mode تطلب خدمات من الـ kernel. يعني لو عايز تعمل حاجة زي فتح ملف، إدارة الذاكرة، أو التحكم في بروسيس، لازم تتحوّل من الـ user-mode للـ kernel-mode.

عادةً، البرامج بتستخدم high-level APIs زي اللي موجودة في kernel32.dll. الـ APIs دي بتروح تندە على native APIs في ntdll.dll، زي مثلاً NtAllocateVirtualMemory. الـ native APIs دي فيها حاجة اسمها syscall stub، ودە عبارة عن كود صغير بينفّذ تعليمة الـ syscall عشان ينقل التنفيذ للـ kernel.

(Intro to Syscalls & Windows internals for malware development Pt.1)

كل syscall عندە رقم مميز اسمه System Service Number (SSN)، بيتحط في الـ EAX register قبل ما تنفّذ الـ syscall. الكيرنل بيستخدم الرقم دە عشان يعرف إيه الخدمة اللي المفروض ينفّذها.

الـ Direct Syscalls

إيه هي؟

الـ Direct syscalls يعني إنك بتتخطّى الـ API layers العادية وبتنفّذ تعليمة الـ syscall مباشرة من الكود بتاعك. بدل ما تندە على دالة في ntdll.dll، بتعمل syscall stub بنفسك وبتحط فيه الـ SSN المناسب.

ليه نستخدمها؟

برامج الأمان زي Endpoint Detection and Response (EDRs) بتعمل حاجة اسمها user-mode API hooking، يعني بتراقب وبتعترض الـ API calls. لما تستخدم direct syscalls، بتفوّت ال hooks دي لأنك مش بتستخدم الـ APIs اللي هما حاطين عليها مراقبة.

(Direct Syscalls: A journey from high to low - RedOps - English)

تفاصيل التنفيذ

عشان تعمل direct syscall، لازم تعمل الخطوات دي:

1. تحط الـ SSN في الـ EAX register.
2. تحضّر الـ registers زي RCX وRDX بالبارامترز بتاعة الـ syscall.
3. تنفّذ تعليمة الـ syscall.

(Direct System Call - SysWhispers - Im0s)

(A Syscall Journey in the Windows Kernel - Alice Climent-Pommeret)

بس في مشكلة: الـ SSNs بتتغيّر من إصدار ويندوز للتاني. عشان كده، في تقنيات زي Hell's Gate وHalo's Gate بتساعدك تجيب الـ SSNs ديناميكيًا وقت التنفيذ، وده بيزوّد التوافق والتخفّي.

(Exploring Hell's Gate - RedOps - English)

العيوب

- الـ syscall لو اتعمل برا ntdll.dll، ده سلوك غريب وممكن يتكشف.
- عنوان الرجوع (return address) بعد الـ syscall بيبقى في مكان غير معتاد في الذاكرة.

الـ EDRs المتقدمة ممكن تلاحظ الحاجات دي وتكشفك.

الـ Indirect Syscalls

إيه هي؟

الـ Indirect syscalls بتحاول تستفيد من مميزات الـ direct syscalls بس بطريقة أقرب للسلوك الطبيعي. بدل ما تنفّذ الـ syscall مباشرة، بتخلّي الكود بتاعك ينط لتعليمة الـ syscall اللي موجودة جوا ntdll.dll.

ليه نستخدمها؟

لما تنفّذ الـ syscall جوا ntdll.dll:

- بتفوّت الـ user-mode API hooks لأنك مش بتنده على الـ API مباشرة.
- الـ call stack بيبقى شكله أكتر طبيعية، فالكشف عنه بيبقى أصعب.

تفاصيل التنفيذ

عشان تعمل indirect syscall:

1. تدور على عنوان الدالة اللي عايزها في ntdll.dll.
2. تحسب المسافة (offset) لتعليمة الـ syscall جوا الدالة دي.
3. تحضّر الـ registers بالباراميترز بتاعة الـ syscall.
4. تنط لتعليمة الـ syscall في ntdll.dll.

الطريقة دي بتخلّي الـ syscall و الرجوع منه يحصل جوا ntdll.dll، وده بيبقى شكله زي السلوك المتوقع.

العيوب

رغم إنها أكتر تخفيًا من الـ direct syscalls، الـ EDRs المتقدمة ممكن تكشفها لو حلّلت الـ call stack كامل أو لو لاحظت أنماط تحكم غريبة.

الـ EDR Evasion Considerations

الـ direct و indirect syscalls هما تقنيات بيستخدموها عشان يفوّتوا كشف الـ EDRs:

- **Direct syscalls**: بتفوّت الـ user-mode hooks بس ممكن تتكشف عشان أنماط التنفيذ الغريبة.
- **Indirect syscalls**: أكتر تخفيًا لأنها بتقلّد السلوك الطبيعي، بس الـ EDRs المتقدمة ممكن تلاحظها لو راقبت الـ call stack أو استخدمت مراقبة في الـ kernel-mode.

الموضوع زي لعبة قط وفار بين المخترقين وبرامج الأمان، كل ما الـ EDRs تتطوّر، المخترقين بيجيبوا تقنيات جديدة.

المراجع

- Direct Syscalls vs Indirect Syscalls - RedOps
- Exploring Hell's Gate - RedOps
- A Syscall Journey in the Windows Kernel - Alice Climent-Pommeret
- Resolving System Service Numbers using the Exception Directory - MDSEC

تم بحمد الله

Contact

For questions, feedback, or support:

- X (Formly Twitter): @00xmora
- Linkedin: @00xmora