شناخت آناتومی بدافزارها

« مبهم‌سازی فراخوانی‌ها سیستمی در ویندوز»

تاریخ تالیف: پنج شنبه – ۱ اسفند ۱۳۹۹

تهیه شده توسط تیم فنی آزمایشگاه امنیت کی‌پاد

مقدمه‌ای بر مبهم‌سازی زمان اجرا

در هنگام تحلیل باینری یک بدافزار، در اولین گام تحلیلگر تلاش خواهد کرد تا اطلاعات کافی از رابط‌های برنامه‌نویسی که در باینری فراخوانی شده‌اند، به دست آورد. با مشاهده لیست توابعی که توسط باینری استفاده شده است، می‌توانید حدس بزنید که باینری قرار است بر روی سیستم چه عملیاتی انجام بدهد.

از همین روی، اگر ما به عنوان توسعه‌دهنده نرم‌افزار بتوانیم به شکلی آناتومی بدافزار را طراحی کنیم که تحلیلگران نتوانند به سادگی لیست رابطهایی را که فراخوانی شده‌اند، کشف کنند (الخصوص در فاز تحلیل استاتیک)، می‌توانیم به شکل قابل توجه‌ای از شناسایی عملکرد باینری خود جلوگیری کنیم.

در این مقاله، به یکی از تکنیک‌هایی خواهیم پرداخت که می‌توانیم با استفاده از آن رابطهای برنامه‌نویسی که در باینری استفاده شده است، مبهم‌سازی یا مخفی کنیم.

کلیدواژه:

تحلیل بدافزار، طراحی بدافزار، طراحی حملات APT گریز از شناسایی، بدافزارنویسی

سناریو مبهم‌سازی

شایان ذکر است، رابطهای برنامه‌نویسی ویندوز شامل هزاران توابع قابل فراخوانی هستند که در گروه‌های اصلی زیر تقسیم می‌شوند.

- سرویس‌های پایه[1]
- سرویس‌های کامپوننتی[2]
- سرویس‌های رابط کاربری[3]
- سرویس‌های چندرسانه‌ای و گرافیکی[4]
- سرویس‌های همکاری و پیام‌رسانی[5]
- سرویس‌های شبکه‌ای[6]
- سرویس‌های وب[7]

به عنوان مثال CreateProcess، CreateFile و GetMessage از این جمله توابع هستند. این توابع توسط شرکت مایکروسافت کاملا مستندسازی شده‌اند و از طریق آدرس msdn.microsoft.com در دسترس برنامه‌نویسان و توسعه‌دهندگان نرم‌افزارهای ویندوزی هستند.

[1] Base Services
[2] Component Services
[3] User Interface Services
[4] Graphics and Multimedia Services
[5] Messaging and Collaboration
[6] Networking
[7] Web Services

homeland security kaipod
کیان امنیت کیپاد

در ادامه بررسی خواهیم کرد که چگونه می‌توان از این دست رابطهای برنامه‌نویسی استفاده کرد، بدون اینکه در جدول توابع ایمپورت توابع[1] باینری یا حتی در محیط دیزاسمبلی نام و جزئیات آن‌ها آورده شود.

فراخوانی استاتیک رابط‌های برنامه‌نویسی

تصور کنید ما در برنامه خود قرار است یک رابط برنامه‌نویسی از ویندوز مانند MessageBox را فراخوانی کنیم تا یک پیام در صفحه نمایش نشان بدهیم. برای فراخوانی این تابع به صورت استاتیک کافی است هدر فایل Windows.h را به درون برنامه وارد کنیم، و سپس بعد فراخوانی تابع MessageBox و عبور پارامترهایی که نیاز دارد، برنامه را کامپایل و اجرا کنیم. در تصویر ۱، پیاده‌سازی این برنامه را مشاهده می‌کنید:

```
1  #include <windows.h>
2  #include <iostream>
3
4  int main(int argc, char* argv[])
5  {
6      MessageBox(NULL, L"API Obfuscation", L"Message", MB_OKCANCEL);
7
8      return 0;
9  }
```

تصویر ۱: نمایش یک پیام در صفحه نمایش

همانطور که در تصویر ۲ مشاهده می‌کنید، بعد اجرای برنامه در خروجی پیامی به ما نشان داده شده است که گواه عملکرد صحیح برنامه است. تا اینجا مشکلی وجود ندارد، مسئله اصلی این است که اگر ما برنامه را در یک پارزر باینری‌های PE باز کنیم، نام تابع MessageBoxW و دیگر توابعی که در این برنامه برای انجام کارهای گوناگون استفاده شده است، لیست خواهند شد.

Name RVA	Name	OriginalFirstThunk	TimeDate Stamp	ForwarderChain	FirstThunk	Description (Read from file)
0000272A	USER32.dll	0000268C	00000000	00000000	00002034	Multi-User Windows USER API Client DLL
0000278E	VCRUNTIME140.dll	00002694	00000000	00000000	0000203C	Microsoft® C Runtime Library
00002962	api-ms-win-crt-runtime-l1-1-0.dll	000026C0	00000000	00000000	00002068	
00002984	api-ms-win-crt-math-l1-1-0.dll	000026B8	00000000	00000000	00002060	
000029A4	api-ms-win-crt-stdio-l1-1-0.dll	00002710	00000000	00000000	000020B8	
000029C4	api-ms-win-crt-locale-l1-1-0.dll	000026B0	00000000	00000000	00002058	
000029E6	api-ms-win-crt-heap-l1-1-0.dll	000026A8	00000000	00000000	00002050	
00002B22	KERNEL32.dll	00002658	00000000	00000000	00002000	Windows NT BASE API Client DLL

Type to filter...

OFT	FT	Hint	Name	Ordinal
0000271C	0000271C	0288	MessageBoxW	

Message
API Obfuscation
OK Cancel

تصویر ۲: نمایش پیام و لیست رابطهای ایمپورت شده به باینری

به عنوان مثال، در تصویر ۲ مشاهده می‌کنید که از کتابخانه User32.dll ویندوز، رابط برنامه‌نویسی MessageBox فراخوانی شده است. با مشاهده این اطلاعات یک تحلیلگر باینری می‌تواند تشخیص بدهد که در جایی از برنامه، به کاربر یک پیام نمایش داده خواهد شد.

به هر صورت، هنگامیکه رابطهای برنامه‌نویسی ویندوز را به صورت استاتیک وارد برنامه می‌کنیم و مورد استفاده قرار می‌دهیم، به سادگی می‌توان لیست آن‌ها را به دست آورد و عملکرد باینری را به صورت تقریبی حدس زد.

[1] Import Address Table

homeland security kaipod
کـــیـــان امنیت کیاپاد

فراخوانی دینامیک رابط‌های برنامه‌نویسی

در این قسمت به این مسئله خواهیم پرداخت که چگونه می‌توانیم به صورت دینامیک رابط‌های برنامه‌نویسی ویندوز را فراخوانی کنیم، به شکلی که اطلاعات آن‌ها در جدول توابع ایمپورت شده باینری قابل مشاهده نباشد.

در قسمت قبل مشاهده کردیم که یک تحلیلگر باینری می‌تواند به سادگی با باز کردن یک باینری در پارزرهای PE اطلاعات زیادی از جمله رابط‌های برنامه‌نویسی که استفاده شده‌اند، استخراج کند.

در این رویکرد، به جای اینکه رابط‌های برنامه‌نویسی را به صورت استاتیک به درون باینری خود وارد کنیم و مورد استفاده قرار بدهیم، ابتدا کتابخانه‌ای را که در آن رابط مورد نظر ما اکسپورت شده است، به درون باینری خود بارگزاری خواهیم کرد، و سپس با به دست آوردن آدرس رابط مد نظر خود آن را در فضای باینری فراخوانی خواهیم کرد. در تصویر ۳، ساختار این برنامه نمایش داده شده است.

```
typedef int(__cdecl* fMessageBox)(HWND, LPCTSTR, LPCTSTR, UINT);

int main(int argc, char* argv[])
{
    HINSTANCE handle_user32;
    fMessageBox DyMessageBox;
    BOOL FreeResult;

    handle_user32 = LoadLibrary(TEXT("user32.dll"));
    // If the handle is valid, try to get the function address.
    if (handle_user32 != NULL)
    {
        DyMessageBox = (fMessageBox)GetProcAddress(handle_user32, "MessageBoxW");
        // If the function address is valid, call the function.
        if (NULL != DyMessageBox)
        {
            (DyMessageBox)(NULL, L"Message sent to the DLL function\n", L"Dynamic Linking", MB_OKCANCEL);
        }
        // Free the DLL module.
        FreeResult = FreeLibrary(handle_user32);
    }
    return 0;
}
```

تصویر ۳: بارگزاری کتابخانه User32 و فراخوانی MessageBox به صورت دینامیک

حال اگر برنامه را اجرا کنیم، در خروجی پیام Message send to the DLL function را مشاهده خواهیم کرد ولی اگر باینری را در یک پارزر PE مانند CFFExplorer باز کنیم، همانطور که در تصویر ۴ نمایش داده شده است، اثری از User32.dll و فراخوانی تابع MessageBox در جدول توابع ایمپورت شده باینری یا به اختصار IAT مشاهده نخواهیم کرد.

Name RVA	Name	OriginalFirstThunk	TimeDate Stamp	ForwarderChain	FirstThunk	Description (Read from file)
0000277C	KERNEL32.dll	00002684	00000000	00000000	00002000	Windows NT BASE API Client DLL
000027E2	VCRUNTIME140.dll	000026C4	00000000	00000000	00002040	Microsoft® C Runtime Library
000029B6	api-ms-win-crt-runtime-l1-1-0.dll	000026F0	00000000	00000000	0000206C	
000029D8	api-ms-win-crt-math-l1-1-0.dll	000026E8	00000000	00000000	00002064	
000029F8	api-ms-win-crt-stdio-l1-1-0.dll	00002740	00000000	00000000	000020BC	
00002A18	api-ms-win-crt-locale-l1-1-0.dll	000026E0	00000000	00000000	0000205C	
00002A3A	api-ms-win-crt-heap-l1-1-0.dll	000026D8	00000000	00000000	00002054	

Dynamic Linking
Message sent to the DLL function
OK Cancel

تصویر ۴: مشاهده جزئیات کتابخانه‌های لینک شده به باینری

با اینکه نام کتابخانه و توابعی که به صورت دینامیک به باینری ما لینک شده‌اند، درون جدول IAT قابل مشاهده نیستند، با این حال اگر ما باینری را دیزاسمبل کنیم، و موقعیت‌هایی را که توابع LoadLibrary و GetProcAddress فراخوانی شده‌اند، مورد بررسی قرار بدهیم، می‌توانیم جزئیات کتابخانه‌ها و رابط‌هایی که به درون باینری لینک شده‌اند، به دست آوریم.

به عنوان مثال، در تصویر ۵ مشاهده می‌کنید که با فراخوانی تابع LoadLibrary کتابخانه User32 به درون باینری بارگزاری شده است و در ادامه با فراخوانی

homeland security kaipod
کیــــان امنیت کایپد

GetProcAddress آدرس تابع MessageBox از طریق ثبات EAX محاسبه و بازگشت داده شده است.

```
push    esi
push    offset LibFileName ; "user32.dll"
call    ds:LoadLibraryW
mov     esi, eax
test    esi, esi
jz      short loc_40103C

push    offset ProcName ; "MessageBoxW"
push    esi             ; hModule
call    ds:GetProcAddress
test    eax, eax
jz      short loc_401035

push    1
push    offset aDynamicLinking ; "Dynamic Linking"
push    offset aMessageSentToT ; "Message sent to the DLL function\n"
push    0
call    eax
add     esp, 10h

loc_401035:             ; hLibModule
push    esi
call    ds:FreeLibrary
```

تصویر ۵: خروجی دیزاسمبلی باینری

با اینکه دیگر شخص تحلیلگر به سادگی نمی‌تواند این اطلاعات را استخراج کند، اما به هر صورت دشواری زیادی هم پیش روی خود ندارد. در تصویر ۵، به سادگی قابل فهم است که برنامه‌نویس کتابخانه User32.dll را به درون باینری وارد کرده است، در ادامه با فراخوانی تابع GetProcAddress آدرس تابع مورد نظر خود را محاسبه کرده است و از طریق ثبات EAX آدرس آن را بازگشت داده است.

[1] Encoding

در نهایت با عبور پارامترهای مورد نیاز تابع MessageBox به درون پشته، دستور call eax اجرا خواهد شد که در ثبات EAX آدرس تابع MessageBox قرار دارد.

پنهان‌سازی رابط‌های برنامه‌نویسی

در دو قسمت قبل، ما دو رویکرد استفاده از رابط‌های برنامه‌نویسی ویندوز به صورت استاتیک و دینامیک را مورد بررسی قرار دادیم. همچنین مشاهده کردیم که در هر دو روش، شخص تحلیلگر باینری به سادگی می‌تواند به اطلاعات مورد نظر خود دسترسی پیدا کند.

حال موضوعی که مطرح می‌شود این است که آیا امکان این وجود دارد که ما به شکلی این ساختار را تغییر بدهیم که تحلیلگران به سادگی نتوانند جزئیات رابط‌هایی را که درون برنامه استفاده شده‌اند، شناسایی کنند؟

هش کردن نام توابع

ما برای اینکه از تحلیل باینری خود جلوگیری کنیم، نیازمند استفاده از رویکردهای رمزگذاری[1] در برنامه خود هستیم. همانطور که در قسمت گذشته مشاهده کردید، دیزاسمبلر IDA Pro به سادگی می‌تواند با تحلیل باینری ما در خروجی مشخص کند که با استفاده از LoadLibrary و GetProcAddress چه توابعی را به درون باینری وارد شده است.

برای اینکه استخراج این دست اطلاعات را در محیط دیزاسمبلی توسط تحلیلگر سخت کنیم، باید از هش کردن نام توابع استفاده کنیم. در تصویر ۶، سورس کد

homeland security kaipod
کیان امنیت کیاپاد

برنامه‌ای را مشاهده می‌کنید که با دریافت نام یک تابع مقدار هش آن را به ما بازگشت خواهد داد.

```
UINT encoder::api_call_hashed(PCHAR arg_input)
{
    INT counter = NULL;
    UINT hash_value = 0;
    UINT N = 0;
    while (counter = *arg_input++)
    {
        hash_value ^= ((N++ & 1) == NULL) ? ((hash_value << 5) ^ counter ^ (hash_value >> 1)) :
            (~((hash_value << 9) ^ counter ^ (hash_value >> 3)));

    }

    return (hash_value & 0x7FFFFFFF);
}
```

تصویر ۶: تابع هش‌کننده نام رابطهای برنامه‌نویسی

این تابع با دریافت یک رشته کاراکتر به عنوان ورودی، مقدار هش آن را محاسبه می‌کند و در خروجی به ما نمایش خواهد داد. به عنوان مثال، اگر نام رابط برنامه‌نویسی MessageBoxW را دریافت کند، مقدار ۱۹۰۳۴۲۵۱۲۹ را به عنوان هش نام تابع MessageBoxW ارائه خواهد کرد.

در ادامه به این مسئله خواهیم پرداخت که چگونه می‌توانیم با مقدار هش نام رابطهای برنامه‌نویسی، توابع مورد نظر خود را بارگزاری و فراخوانی کنیم بدون اینکه در محیط حتی IDA Pro قابل شناسایی باشند.

پردازش جدول توابع اکسپورت شده کتابخانه‌ها

برای اینکه بتوانیم تابع مورد نظر خود را به صورت دینامیک و هش شده فراخوانی کنیم و در ادامه مورد استفاده قرار بدهیم، ابتدا باید مقدار هش تابع MessageBoxW را در برنامه به صورت یک مقدار ثابت تعریف کنیم که در ادامه با جستجوی و مقایسه نام توابع اکسپورت شده کتابخانه مورد نظر خود، آدرس تابع هدف را به دست بیاوریم. در تصویر ۷، مشاهده می‌کنید که مقدار هش نام تابع MessageBoxW به صورت یک ثابت در برنامه تعریف شده است:

```
typedef int (WINAPI* fnMessageBoxW)(HWND, LPCTSTR, LPCTSTR, UINT);
#define HashMessageBoxW 1903425129

#define RtlOffsetToPointer(Module, Pointer) PBYTE(PBYTE(Module) + DWORD(Pointer))

struct BASE {
    HMODULE User32;
    fnMessageBoxW _MessageBoxW;
};
```

تصویر ۷: تعریف هش تابع MessageBoxW به صورت یک عدد ثابت

در گام بعد، باید ابتدا با فراخوانی LoadLibrary کتابخانه مورد نظر خود را به درون باینری بارگزاری کنیم، سپس به صورت دستی تمامی توابع اکسپورت شده توسط کتابخانه مورد نظر را پردازش کنیم. به منظور پردازش نام توابع اکسپورت شده کتابخانه باید نام تمامی آن توابع را هش کنیم، و در نتیجه مقدار هش آن را با هش MessageBoxW مقایسه کنیم.

اگر دو مقدار هش با یکدیگر برابر بودند، می‌توانیم اطمینان حاصل کنیم که آدرس بازگشت داده شده برای تابع MessageBoxW است، بدون اینکه رشته یا کاراکتری در باینری وجود داشته باشد که مشخص کند ما به دنبال استفاده از تابع MessageBoxW هستیم. تصویر ۸، شیوه پردازش توابع اکسپورت شده توسط تابع get_proc_address را نمایش می‌دهد که یک نمونه سفارشی‌سازی شده از تابع GetProcAddress است.

```
HMODULE obfuscation::get_proc_address(HMODULE arg_module_base, DWORD arg_hash, DWORD arg_data_directory)
{
    PIMAGE_DOS_HEADER image_dos_header = PIMAGE_DOS_HEADER(arg_module_base);
    if (image_dos_header->e_magic == IMAGE_DOS_SIGNATURE) {
        PIMAGE_NT_HEADERS nt_headers = PIMAGE_NT_HEADERS(RtlOffsetToPointer(arg_module_base, image_dos_header->e_lfanew));
        if (nt_headers->Signature == IMAGE_NT_SIGNATURE) {
            if (nt_headers->OptionalHeader.DataDirectory[arg_data_directory].VirtualAddress && arg_data_directory < nt_headers-
             >OptionalHeader.NumberOfRvaAndSizes) {
                PIMAGE_EXPORT_DIRECTORY image_export = PIMAGE_EXPORT_DIRECTORY(PBYTE(RtlOffsetToPointer(arg_module_base, nt_headers-
                 >OptionalHeader.DataDirectory[arg_data_directory].VirtualAddress)));
                if (image_export != ERROR) {
                    PDWORD address_of_names = PDWORD(RtlOffsetToPointer(arg_module_base, image_export->AddressOfNames));
                    for (DWORD n = NULL; n < image_export->NumberOfNames; ++n) {
                        LPSTR hashed_returned_names = LPSTR(RtlOffsetToPointer(arg_module_base, address_of_names[n]));
                        if (encoder::api_call_hashed(hashed_returned_names) == arg_hash)
                        {
                            PDWORD address_of_function = PDWORD(RtlOffsetToPointer(arg_module_base, image_export->AddressOfFunctions));
                            PWORD address_of_ordinal = PWORD(RtlOffsetToPointer(arg_module_base, image_export->AddressOfNameOrdinals));
                            return HMODULE(RtlOffsetToPointer(arg_module_base, address_of_function[address_of_ordinal[n]]));
                        }
                    }
                }
            }
        }
    }
    return ERROR;
}
```

تصویر ۸: پردازش توابع اکسپورت شده کتابخانه

همانطور که در تصویر ۸ مشاهده می کنید، در یک حلقه for از ابتدا تا انتها لیست توابع اکسپورت شده کتابخانه مورد نظر خود را پویش می‌کنیم (در اینجا کتابخانه مد نظر User32.dll است) و نام تمامی توابع اکسپورت شده توسط این کتابخانه را به تابع محاسبه‌گر هش خود عبور خواهیم داد.

اگر مقدار هش بازگشت داده شده توسط api_call_hashed با مقداری که ما توسط پارامتر arg_hash به تابع عبور داریم، برابر باشد، گواه این است که تابع اکسپورت شده، همان تابعی است که ما به دنبال آن هستیم. در نهایت آدرس آن تابع را بازگشت خواهیم داد.

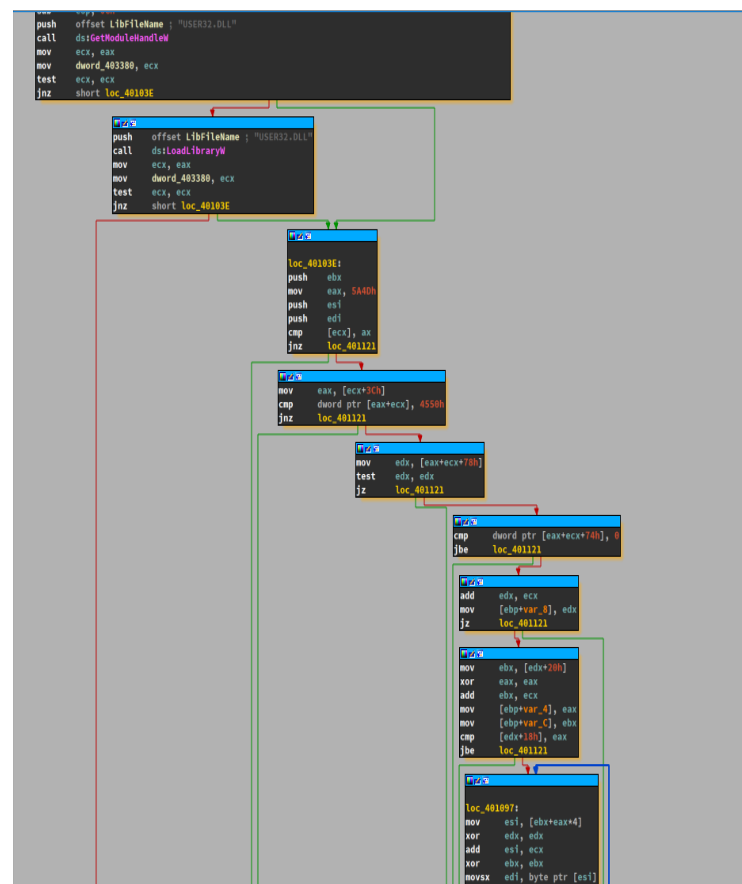
```
VOID obfuscation::load(BASE arg_api)
{
    if ((g_CallGate.User32 = GetModuleHandleW(L"USER32.DLL")) == ERROR) {
        g_CallGate.User32 = LoadLibraryW(L"USER32.DLL");
    }

    if (g_CallGate.User32) {
        g_CallGate._MessageBoxW = fnMessageBoxW(resolved_functions(g_CallGate.User32, HashMessageBoxW));
    }
}
```

تصویر ۹: به دست آوردن آدرس تابع مورد نظر

در تصویر ۹، مشاهده می‌کنید که با فراخوانی تابع resolved_functions که در بطن خود تابع get_proc_address را فراخوانی می‌کند، آدرس کتابخانه User32.dll و همچنین هش تابع MessageBoxW عبور داده شده است. وقتی تابع مورد نظر ما شناسایی شد، آدرس آن در g_CallGate._MessageBoxW قرار خواهد گرفت.

حال ما می‌توانیم در تابع main یا هر تابع دیگری از رابط MessageBoxW استفاده کنیم، بدون اینکه اطلاعات آن در IAT وجود داشته باشد و یا حتی در محیط دیزاسمبلی به سادگی مشخص شود که در باینری چه تابعی از User32 بارگزاری و فراخوانی شده است. در تصویر ۱۰، خروجی دیزاسمبلی را مشاهده می‌کنید که نسبت به خروجی دیزاسمبلی تصویر ۵ تفاوت شایانی دارد.

homeland security kaipod
کیان امنیت کایپد



تصویر ۱۰: مبهم‌سازی فراخوانی رابطهای سیستمی در محیط دیزاسمبلی

همانطور که در تصویر ۱۰ قابل مشاهده است، دیگر اثری از فراخوانی MessageBoxW قابل مشاهده نیست. از همین روی، شخصی که قصد تحلیل این باینری را اکنون دارد، به سختی می‌تواند هدف و منطق آن را استخراج کند چون رابطهایی که برای پیاده‌سازی این برنامه مورد استفاده قرار گرفته‌اند، قابل درک و شناسایی نیستند. اگر هم شخص تحلیلگر بخواهد لیست توابع را به دست بیاورد، باید الگوریتم هش ما را شناسایی کند که آن هم یک عمل زمان‌بر و دشوار است. در این مقاله، صرفا به استفاده از MessageBoxW از کتابخانه User32.dll پرداخته شد، اما با این حال شما می‌توانید با این تکنیک فراخوانی تمامی توابع را پنهان کنید.

نتیجه‌گیری

در این مقاله، بررسی کردیم که وقتی در نرم‌افزار خود از رابطهای برنامه‌نویسی ویندوز استفاده می‌کنیم که یک سری از اعمال را بر روی سیستم انجام بدهیم، مانند ایجاد آبجکت‌های کرنل از جمله پروسه، ترد، سوکت شبکه و ... تحلیلگران باینری به سادگی می‌توانند با مشاهده لیست رابطهایی که توسط باینری مورد استفاده قرار گرفته‌اند، از عملیات و هدف نهایی باینری ما پرده‌برداری کنند.

از همین روی، در این مقاله توضیح دادیم که چطور می‌توان با هش کردن نام توابع مورد نظر خود و همچنین پردازش دستی جدول اکسپورت توابع کتابخانه‌ها، تابع مورد نظر خود را شناسایی، بارگزاری و مورد استفاده قرار بدهیم بدون اینکه جزئیاتی از آن در محیط پارزرهای PE یا حتی محیط دیزاسمبلی قابل نمایش و شناسایی باشد.

برای مطالعه دقیق‌تر بر روی این تکنیک می‌توانید از طریق مخزن https://github.com/miladkahsarialhadi/rao به کدهایی که در این مقاله بررسی شدند، دسترسی بگیرید.

homeland security kaipod
کـــیـان امنیت کایپد