

Forum:[Hack Forums](#)

Thread: [Bypassing user-mode API hooks](#)

AmericanExpress - June 25, 2017, 3:43 pm

Hello.

This thread is just going to be a short basic tutorial however it should successfully get my point across.

For those who are unaware, API hooking is the practice of altering the execution flow within a process; it allows you to control the memory of the process. The most common method of API hooking (user-mode) would be through altering the instructions at the start of the target function in memory to be hooked so once code execution reaches the function you hooked, it will end up executing your own code instead (e.g. place a JMP instruction to the address of your own function which is present within that affected process). There are many other types of API hooking which affect other factors too: IAT and EAT hooking techniques for example.

If you use a direct (manual) Native API system call then you should be able to successfully bypass API hooks (in theory at least). If you are interested in performing system calls then you will require to do lots of extensive testing to ensure it is compatible with all versions of Windows, and with different patch updates; for example, you need to make sure the byte instructions you are copying over to the newly allocated memory is correct for that environment. I cannot remember the website (hopefully another member can post it in the comments), however there is actually an online system call table which is extremely beneficial when developing a system call wrapper.

The purpose of this thread isn't to teach the reader how to perform NTAPI system calls, I'll save that for another thread. I just want to outline a little trick I know of to replicate the same effect, although I will explain briefly how you can perform a system call:

1. Setup an array of bytes which holds the byte representation of the ASM instructions for the NTAPI function you wish to perform the system call with.
2. Allocate some memory within your program which will be used later on to store the function instructions (e.g. VirtualAlloc/NtAllocateVirtualMemory).
3. Copy the function to the newly allocated memory (the byte representations) (e.g. via memcpy).
4. Setup the function definition (e.g. typedef) of the NTAPI function you wish to call via the system call and then apply it to the address of the memory you allocated which now contains the instructions for that function.
5. Now call the function using the address which you made which points to the memory you created and stored the function bytes in.

The above is pretty much what NTDLL.DLL does, it is essentially a system call wrapper. When you call Win32 API functions like kernel32.dll!CreateRemoteThread it will end up at the NTAPI equivalent (which in this case would be NtCreateThreadEx) which is exported by ntdll.dll. Eventually, a system call will be made once code execution reaches the stub within ntdll.dll for that function. The system ID for that function is moved into the EAX register and the Windows Kernel will retrieve that information once execution has been moved to the kernel, then the retrieved information from the EAX register is looked up against the SSDT (System Service Dispatch Table) to retrieve the address of the real function which is being called in memory and then code execution moves to the destination of that pointer address found within the SSDT.

The System Service Dispatch Table is basically a massive table of all the accessible and exported NTAPI functions. The pointer addresses assigned to each table entry points to where the real NTAPI function resides in memory; in the case of a kernel-mode rootkit which makes use of kernel-mode patching mechanisms such as SSDT hooking, these pointer addresses become manipulated so all code execution for the targeted functions pass through the rootkit's callback function first.

Enough of that theory for now, we'll discuss it more when I make the tutorial on system calls. For now, the trick I would like to share is the following:

- Make a copy of ntdll.dll and store it somewhere else (e.g. your application's installation directory).
- Now load the copy of ntdll.dll into your process (e.g. via LoadLibraryA/W although the better way would be through manual mapping it so it is hidden from the PEB ModuleList which is stealthier and more silent).
- Now call the NTAPI functions the same way you would when using the original NTDLL.DLL module, although retrieve the addresses from the duplicate instead of "ntdll.dll".

To elaborate a bit more, with a code example, let's assume that I created a duplicate of ntdll.dll and it resides in the start-up path of my C++ program. Let's pretend that the duplicated DLL file name is "bypass.dll". Now the below code will call NtTerminateProcess, however it will retrieve the address of the function from "bypass.dll" instead of "ntdll.dll". This means that if a rootkit (or just general security software) injects code into your process and hooks a bunch of functions exported by "ntdll.dll", it will not affect you since you are using the duplicated DLL which was not affected by the hooking!

Code example (C++):

Code #include <Windows.h>

#include <winternl.h>

```
typedef NTSTATUS(NTAPI *fNtTerminateProcess)(HANDLE ProcessHandle, NTSTATUS  
ExitStatus);
```

```
void bypass()  
{  
    LPVOID lpFunctionAddr = (LPVOID)GetProcAddress(LoadLibraryA("bypass.dll"),  
"NtTerminateProcess");  
    if (lpFunctionAddr)  
    {  
        HANDLE ProcessHandle = OpenProcess(PROCESS_TERMINATE, FALSE, 12444);  
        if (ProcessHandle)  
        {  
            __try  
            {  
                fNtTerminateProcess NtTerminateProcess = (fNtTerminateProcess)lpFunctionAddr;  
                NtTerminateProcess(ProcessHandle, 0);  
            }  
            __except (EXCEPTION_EXECUTE_HANDLER)  
            {  
            }  
        }  
    }  
}
```

```
int main()
```

```
{  
    bypass();  
    return 0;  
}
```

Just make sure you pick the correct ntdll.dll for the duplicate, you may notice in some circumstances (e.g. X86-X64 process) that there are multiple instances of ntdll.dll modules loaded within the same process but reside in different directories found under Windows folder (e.g SysWow64). This method works best on 32-bit processes on a 32-bit OS or a 64-bit process on a 64-bit OS since in those cases these days the system calls are made directly (in the past on 32-bit it did actually pass through a function called KiFastSystemCall - even these days on X86-X64 processes the NTAPI calls pass through the Wow64 layer which does open up interception opportunities on functions like X86SwitchTo64BitMode, Wow64Transition, etc.).

Let me know if you found this thread educational & if you've got any suggestions for future threads just lmk in comments or PM.

TouchMyMalware - June 25, 2017, 3:53 pm

Edit: I spoke to someone else I suppose.

AmericanExpress - June 25, 2017, 3:57 pm

(Jun 25, 2017 03:53 PM)TouchMyMalware Wrote: [\[CITE\]](#) Didn't I just discuss how to do this with you on discord? Because I'm pretty sure this is mostly a copy and paste of everything I wrote. I have never spoken to you before off-site, I don't even know who you are. I think you've mistaken me for someone else?

With that in mind, I am pretty sure this is something I have literally just typed up on the thread editing window... I'd prefer not getting trolled since after all I am trying to provide useful content to the community to help educate people. Hopefully that is alright?

maldev0 - June 25, 2017, 11:28 pm

(Jun 25, 2017 03:57 PM)AmericanExpress Wrote: [\[CITE\]](#) I have never spoken to you before off-site, I don't even know who you are. I think you've mistaken me for someone else?

With that in mind, I am pretty sure this is something I have literally just typed up on the thread editing window... I'd prefer not getting trolled since after all I am trying to provide useful content to the community to help educate people. Hopefully that is alright?

It's just a weird coincidence because I had a talk with TouchMyTralala and you've literally made a post about it rofl. It's fun because we've been talking about the exact same routines described in your post.

HFDone - June 26, 2017, 12:22 am

Windows System Call Table

x32 - <http://j00ru.vexillium.org/ntapi>

x64 - http://j00ru.vexillium.org/ntapi_64

apis hook is old as windows
the api hook is not global
manual mapping of ntdll

AmericanExpress - June 26, 2017, 4:52 am

(Jun 26, 2017 12:22 AM)HFDone Wrote: [\[CITE\]](#) Windows System Call Table

x32 - <http://j00ru.vexillium.org/ntapi>

x64 - http://j00ru.vexillium.org/ntapi_64 Yeah, that was the website I was referring to! Thank you!

(Jun 25, 2017 04:07 PM)Yani Wrote: [\[CITE\]](#) Interesting read, thanks for sharing.

Consider writing blog posts :)Thank you for support, more content coming soon ?

TouchMyMalware - June 26, 2017, 5:16 am

(Jun 25, 2017 11:28 PM)maldev0 Wrote: [\[CITE\]](#) It's just a weird coincidence because I had a talk with TouchMyTralala and you've literally made a post about it rofl. It's fun because we've been talking about the exact same routines described in your post.

Yes. There are still flaws in his design though because he's assuming some functions are left unhooked. Like NtReadFile, and NtOpenFile.

AmericanExpress - June 26, 2017, 5:21 am

(Jun 26, 2017 05:16 AM)TouchMyMalware Wrote: [\[CITE\]](#) Yes. There are still flaws in his design though because he's assuming some functions are left unhooked. Like NtReadFile, and NtOpenFile. Could you elaborate a bit more?

The point of the method is to bypass user-mode hooks on ntdll.dll by using a duplicated copy. If the hook code is more sophisticated and scans through the loaded modules to detect other copies of ntdll.dll and then hooks them too then it won't work unless the duplicate is manual mapped (harder to detect).

I'm not really catching on with what you are saying; if NtReadFile/NtOpenFile are hooked in the ntdll.dll module then use them from the duplicated copy. Bear in mind for some functions exported by NTDLL.DLL it doesn't perform a system call right away (e.g. RtlCreateUserThread won't and neither will RtlSetProcessIsCritical) so you'd need to resolve addresses and all that, but for most functions like NtOpenProcess, NtTerminateProcess, etc, should work fine.

TouchMyMalware - June 26, 2017, 6:07 am

(Jun 26, 2017 05:21 AM)AmericanExpress Wrote: [\[CITE\]](#) Could you elaborate a bit more?

The point of the method is to bypass user-mode hooks on ntdll.dll by using a duplicated copy. If the hook code is more sophisticated and scans through the loaded modules to detect other copies of ntdll.dll and then hooks them too then it won't work unless the duplicate is manual mapped (harder to detect).

I'm not really catching on with what you are saying; if NtReadFile/NtOpenFile are hooked in the ntdll.dll module then use them from the duplicated copy. Bear in mind for some functions exported by NTDLL.DLL it doesn't perform a system call right away (e.g. RtlCreateUserThread won't and neither will RtlSetProcessIsCritical) so you'd need to resolve addresses and all that, but for most functions like NtOpenProcess, NtTerminateProcess, etc, should work fine.

LoadLibrary calls LdrLoadDll which is just a giant wrapper for mapping an executable into memory relying on NtReadFile and NtOpenFile. I just can't remember if it calls the two functions at the usermode level or the kernel level. If it calls them in the kernel then it will bypass the usermode hooks. Likewise you're also assuming LdrLoadDll or LoadLibrary aren't hooked either. If someone were to hook the call then check NT ordinals passed into the gate they could essentially hook every system call function. It's not uncommon for some malware/anticheats to do this.

AmericanExpress - June 26, 2017, 6:11 am

(Jun 26, 2017 06:07 AM)TouchMyMalware Wrote: [\[CITE\]](#) LoadLibrary calls LdrLoadDll which is just a giant wrapper for mapping an executable into memory relying on NtReadFile and NtOpenFile. I just can't remember if it calls the two functions at the usermode level or the kernel level. If it calls them in

the kernel then it will bypass the usermode hooks. Likewise you're also assuming LdrLoadDll or LoadLibrary aren't hooked either. If someone were to hook the call gate then check NT ordinals passed into the gate they could essentially hook every system call function. It's not uncommon for some malware/anticheats to do this. I understand what you are talking about now, your other post wasn't clear enough to me... Yes, you do have a point.

About the LdrLoadDll hook, that one is an easy solution... Just make a hook detection mechanism for that function and repair the function if it was hooked (e.g. run-time byte patch hook just restore the function prologue). But it depends on the situation I guess.

At the end of the day, this is just a trick which is a bit useful and does work to a reasonable extent. As I've said on previous threads before, nothing is full-proof. I myself prefer manual system calls through a custom wrapper but that is just my opinion.

Good post though. ?

EDIT: Maybe make a custom wrapper for LdrLoadDll. Reverse the function and make a custom wrapper where any other NTAPI calls go through the duplicated DLL. ?

LevelUp - July 4, 2017, 4:23 am

(Jun 26, 2017 06:07 AM)TouchMyMalware Wrote: [\[CITE\]](#) LoadLibrary calls LdrLoadDll which is just a giant wrapper for mapping an executable into memory relying on NtReadFile and NtOpenFile. I just can't remember if it calls the two functions at the usermode level or the kernel level. If it calls them in the kernel then it will bypass the usermode hooks. Likewise you're also assuming LdrLoadDll or LoadLibrary aren't hooked either. If someone were to hook the call then check NT ordinals passed into the gate they could essentially hook every system call function. It's not uncommon for some malware/anticheats to do this.

The functions are called at usermode level i believe.

AmericanExpress - July 4, 2017, 5:07 am

(Jul 4, 2017 04:23 AM)LevelUp Wrote: [\[CITE\]](#) The functions are called at usermode level i believe. I believe it is user-mode too since I doubt Microsoft would have been thinking about "anti-hook" for NtReadFile/NtOpenFile... It'll pass through NTDLL.DLL stub and land in kernel-mode anyway thanks to the system call. ?

Anyway what I said in my previous response was IMO pretty silly about an "easy solution"... For example, you could hook NtProtectVirtualMemory and keep the memory protected at the address of LdrLoadDll to prevent people from unhooking, or even target functions like NtWriteVirtualMemory/memcpy to prevent copying of bytes across to the address of LdrLoadDll and/or bytes representing a system call... etc.

@TouchMyMalware Good post and thank you for contributing, OOI how would you approach bypassing a hook on LdrLoadDll? ?

LevelUp - July 4, 2017, 6:39 am

(Jul 4, 2017 05:07 AM)AmericanExpress Wrote: [\[CITE\]](#) @TouchMyMalware Good post and thank you for contributing, OOI how would you approach bypassing a hook on LdrLoadDll? ?

How about direct syscalls with shellcode or instead of your way of copying ntdll and using loadlibrary, use NtOpenSection and then map it into the current process with NtMapViewOfSection.