





# Lecture 10:

## DLL Injection

# Quick Review

Static Linking

Dynamic Linking: Implicit

Dynamic Linking: Explicit

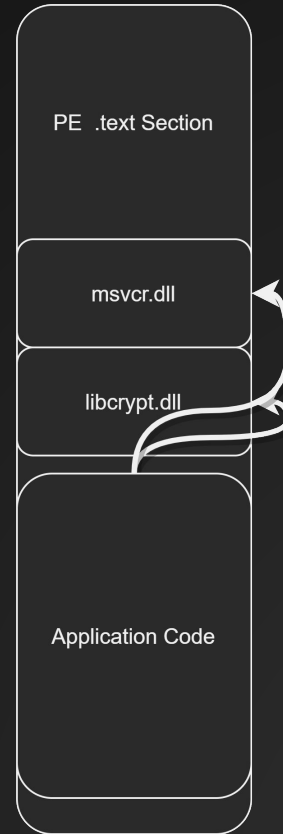
# Static Linking

Statically linking against a library embeds the library inside of the PE

This is inefficient with space, but can solve many portability problems with code.

The main reason to statically link a library is if you are unsure if it will be available on the host machine

This bloats the size of the code, which may or may not be an issue.



Static Linking embeds dependencies inside of the PE

# Dynamic Linking: Implicit

Implicit Linking is declaring the required imports inside of the PE

The PE loader will then resolve these imports before execution is passed to the main thread

If any of the DLLs cannot be found, the PE loader aborts and the process exits with an error.

Most legitimate applications use implicit linking, and will ship with required DLLs packaged together with the application. These DLLs are then loaded at runtime.

# Resolving Imports

When a DLL is loaded by a process, it is not guaranteed to be placed in the same place in memory, nor is it guaranteed to be placed in the same *relative* location

# Resolving Imports

To handle variability in load location, the programmer can declare all imports in the *Import Address Table* (IAT)

This shifts the work to the PE loader to resolve all the imports.

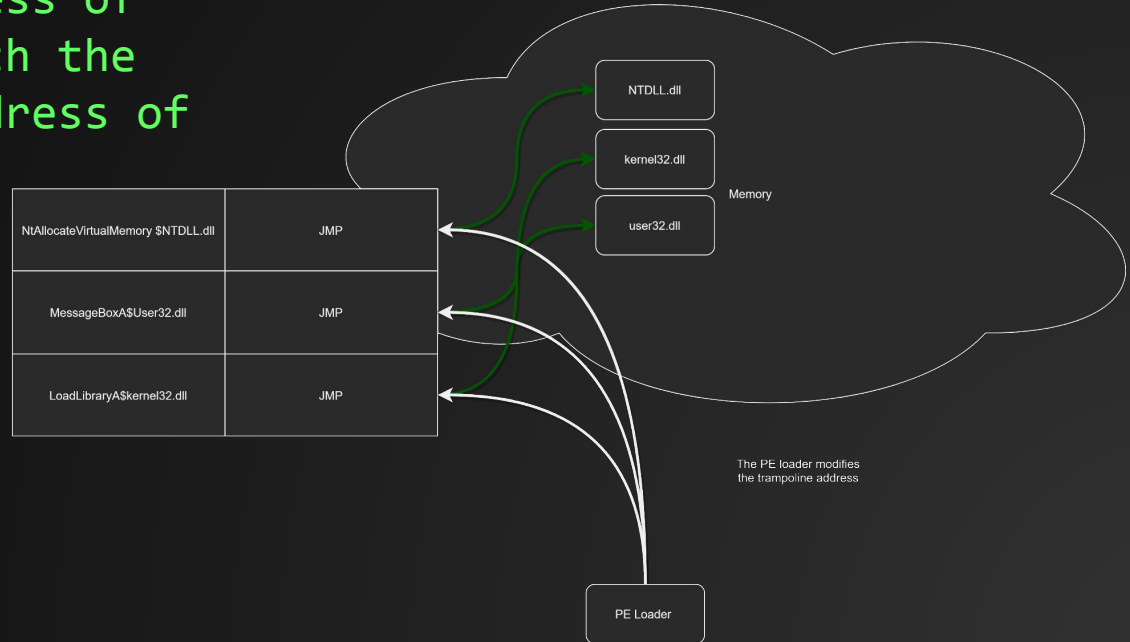


# Resolving Imports

The IAT is effectively a collection of key value pairs, where the key is the name of an imported function, and the value is the address of a small function that jumps to the location of the real function.

# Resolving Imports

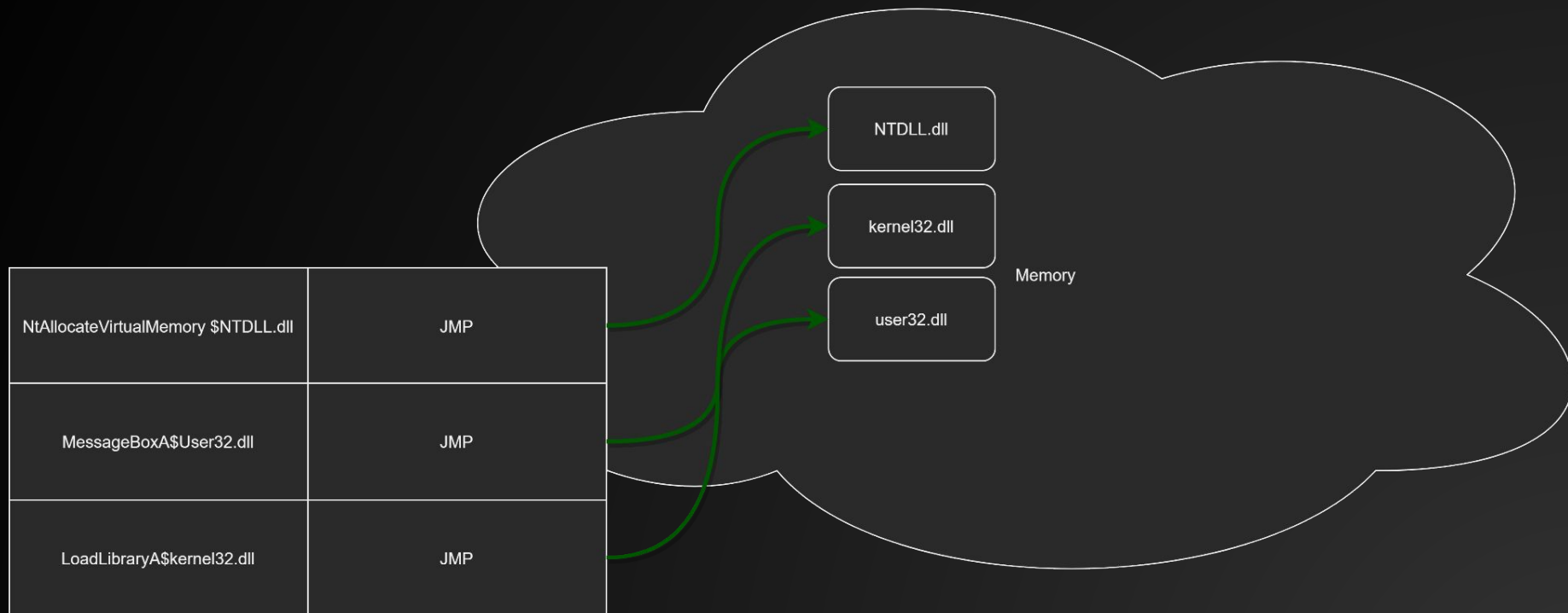
When resolving imports, the PE loader will load all required DLLs, identify the address of the loaded DLL, and patch the IAT with the correct address of the requested function



# Resolving Imports

- The application code simply calls the function referenced in the IAT, which is simply a *trampoline* to the real code.
- I.e., it is a static location relative to the PE's base address that application code can reference
  - I.e., it is simply an unconditional jump to the memory mapped location of the target function.

# Visual: IAT



# Dynamic Loading: Explicit

- The programmer explicitly calls LoadLibrary, and has the proper function prototypes inside of their code.
- The legitimate reason to do this, is to allow the execution of the program to continue even if a DLL does not exist.
- If an import for an implicitly loaded DLL is missing, the program stops.
- The application using an explicitly loaded DLL can choose how to handle a missing DLL.

# Dealing with Relative Addresses

A relative address is an address that is defined as a particular offset from a *base address* (the start of the PE's image in memory)

When the application is loaded into memory, same as the DLLs referenced in the IAT, its location might not align with the preferred location

In this case, the PE loader needs to modify the Base Location to handle relative locations.

This is especially important when exploitation mitigations such as ASLR are enabled, as the PE in this case should (probabilistically) never be loaded in the preferred address.

Applications that have only relative addresses are called Position independent. Generating Position Independent Code (PIC) is handled by the compiler, and can be enabled with `-fPIC` for GCC. This means you will see no unconditional jumps in application code, and instead will only see relative Jumps

# Dealing with Absolute Addresses

An absolute address is a reference to a static location. Think `JMP <addr>`

If this address references a specific spot in memory, this could be tricky.

The base relocation table handles absolute addresses by creating a table of pointers to absolute addresses.

If the process is not loaded into its preferred address, the PE loader will modify all the absolute addresses to work with the new base address.

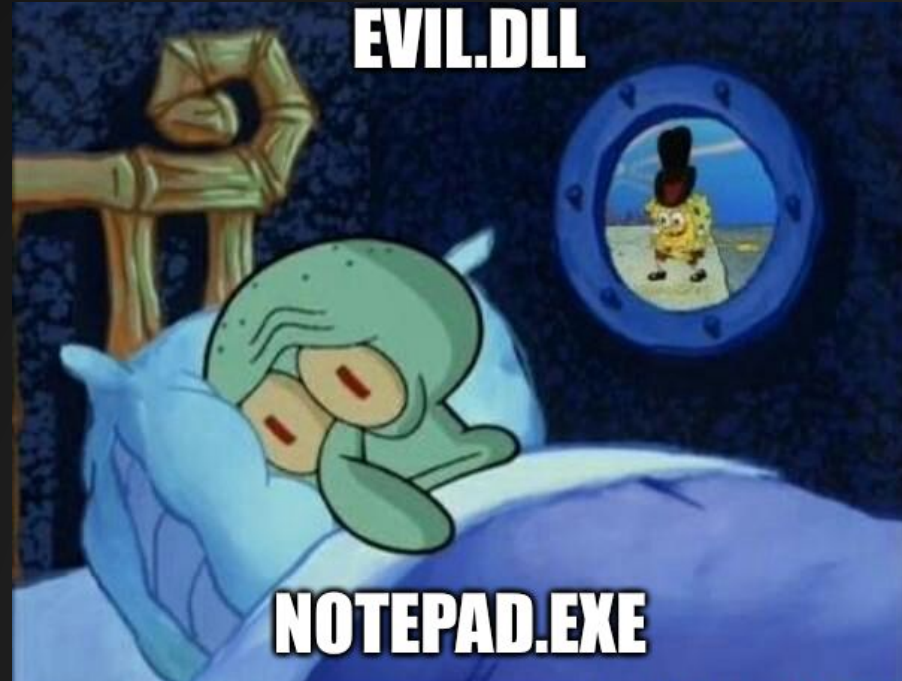
We generally don't have to worry about this in our class, as all code will be PIC

# DLL Injection

Forcing a remote process to load a library (DLL)

The DLL upon being loaded executes whatever code is contained in its attach process code.

There are a lot of reasons to do this, but general it is used for intercepting function calls, interacting with remote processes and hiding code in a remote process





## Example one: Anti Malware

Many anti-malware executables have all the hallmarks of malware themselves!

An AV might inject a DLL into every process that a user spawns. It can then use this DLL to intercept commands to WinAPI functions to detect suspicious sequences of Function calls

## Example 2: Cheat Engines

Running code in the context of a remote process gives unfettered access to resources used by the target process

This can be used to monitor application behavior, modify data, and inject functionality.

For example, the attach function in the injected DLL can modify values in memory associated with player's health bar.

## Example 3: Malware and Hooking

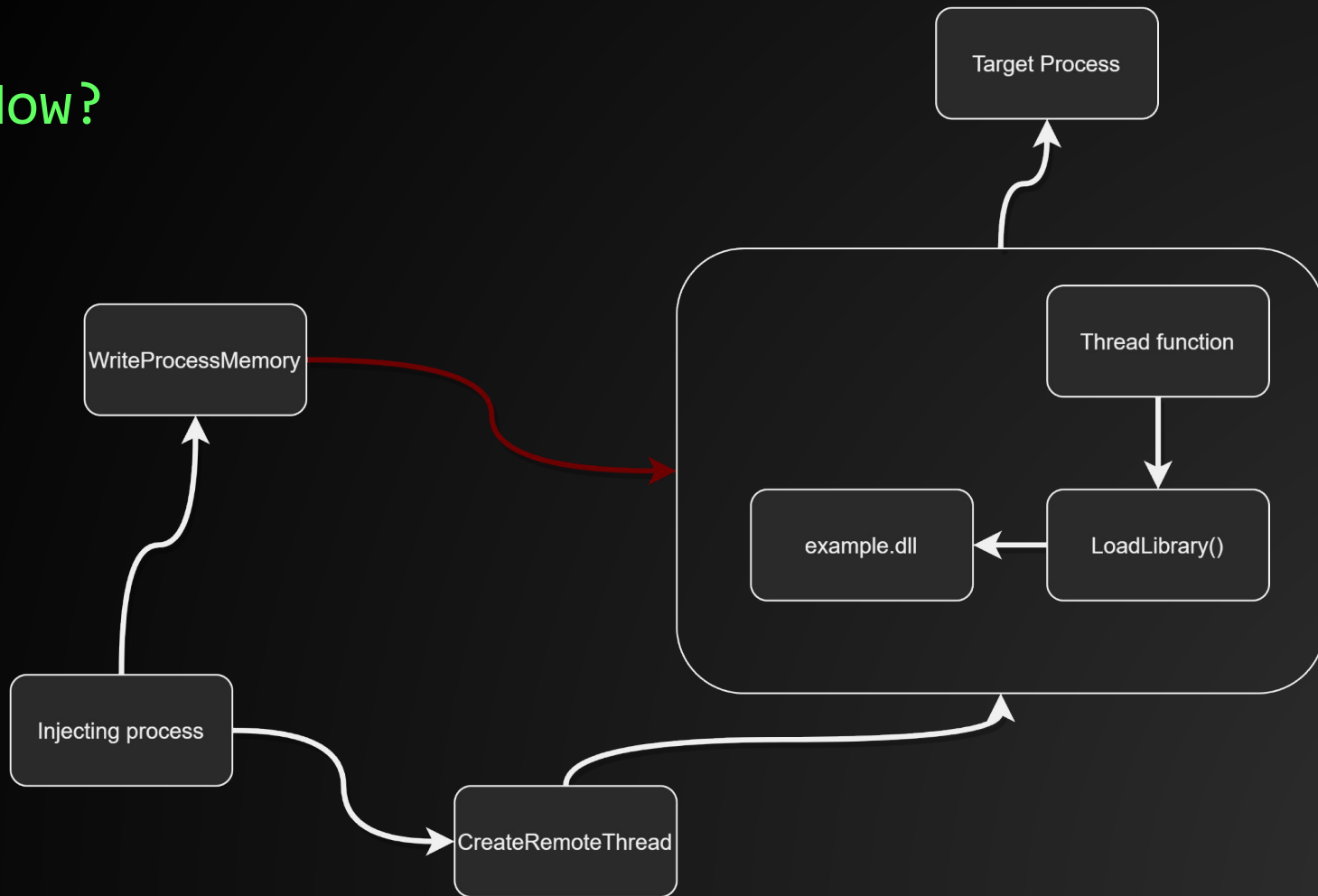
Imagine a user running Google Chrome logging into a website

If malware injects a malicious DLL into a chrome instance, it can read it's process memory, modify functions, and steal data

For example, if the chrome function that sends an HTTP request is known, the DLL can patch references to that function with a small trampoline to code controlled by the malware

The malicious code can then inspect, and modify the arguments and forward them along.

# How?



# Steps to Inject a DLL

- 1) Get a Handle to the remote process
- 2) Get the **\*\*FULL\*\*** path to the DLL on disk
- 3) Copy the path of the DLL to the remote Process
- 4) *Politely* ask the remote process to call LoadLibrary

OK but how?

# Steps to Inject a DLL (Usual way)

- 1) Get the PID of the remote process we want to inject into
- 2) Get the FULL path to the DLL on disk
- 3) Open a handle to the remote process with the proper permissions
- 4) Get the address of LoadLibraryA in kernel32.dll in our process
- 5) Allocate a buffer in the remote process
- 6) Copy the path of the DLL to the remote process
- 7) Force the remote process to execute LoadLibraryA

# Steps to Inject a DLL (Usual way)

- 1) Get the PID of the remote process we want to inject into
  - a) Fork & run OR pass as an argument
- 2) Get the FULL path to the DLL on disk
  - a) Hardcode it
- 3) Open a handle to the remote process with the proper permissions
  - a) `OpenProcess(...)`
- 4) Get the address of `LoadLibraryA` in `kernel32.dll` in our process
  - a) This will be the same addr in the remote process! (Why?)
- 5) Allocate a buffer in the remote process
  - a) `VirtualAllocEx(...)`
- 6) Copy the path of the DLL to the remote process
  - a) `WriteProcessMemory`
- 7) Force the remote process to execute `LoadLibraryA`
  - a) `CreateRemoteThread(...)`

# Execution dependent: CreateRemoteThread

```
HANDLE CreateRemoteThread(  
    [in] HANDLE hProcess,  
    [in] LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    [in] SIZE_T dwStackSize,  
    [in] LPTHREAD_START_ROUTINE lpStartAddress,  
    [in] LPVOID lpParameter,  
    [in] DWORD dwCreationFlags,  
    [out] LPDWORD lpThreadId  
);
```



# LPTHREAD\_START\_ROUTINE

LPTHREAD\_START\_ROUTINE lpStartAddress → function pointer!

LPVOID lpParameter → pointer to start of arguments

# Getting the Arguments into the remote process

- Open a handle to the target Process with `OpenProcess`
- Allocate a page of memory in the remote process with `VirtualAllocEx`
- Copy the required arguments (the full path to our DLL) into the remote buffer
- At this point, the thread arguments are in the remote process. We simply need to create a remote thread

# Calling a function in the remote process

- We need to know the start Virtual Address of the function in the remote processes' address space!
- For functions in Kernelbase, kernel32 and ntdll, these will be set randomly on boot, but will be the same from process to process!
- For the simple case, we need only calculate the address of `kernel32.dll$LoadLibraryA`

# Calling a function in the remote process

- The code:

```
o (LPTHREAD_START_ROUTINE)::GetProcAddress(::GetModuleHandleA("kernel32"), "LoadLibraryA")
```

- The above code calculates the address of kernel32.dll\$LoadLibraryA at runtime and casts it to an LPTHREAD\_START\_ROUTINE
- Use CreateRemoteThread with the thread arguments set to the contents of the remote buffer containing our DLL!

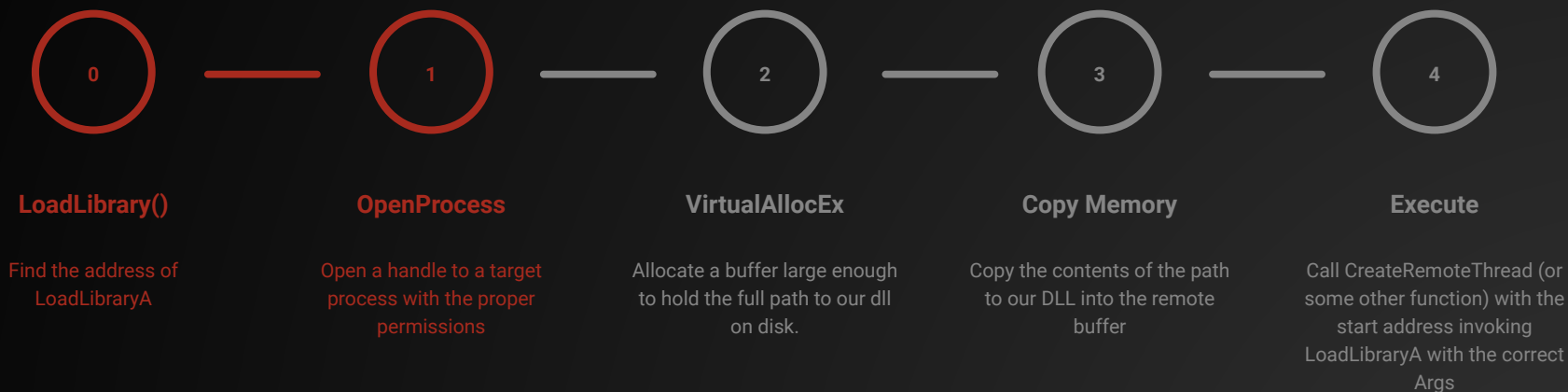
# End Result

The remote process creates a new thread who's code consists of a single call to LoadLibraryA, and whatever code is present in

`DLL_PROCESS_ATTACH`

We will talk more next week about how malware can use this to “hook” important functions

# Dll Injection: Summary



# DLL injection Summary

...Politely asking a remote process to load a DLL that exists on disk.



# Limitations of the previous setup

Limitations of our previous setup:

- 1) The DLL needs to exist on disk!
- 2) Calling LoadLibraryA creates an image load callback.
- 3) Doing anything complex (i.e more than running say MessageBoxA) in DLL\_PROCESS\_ATTACH is going to break stuff.



## Limitations of the previous setup

Doing anything complex (i.e more than running say MessageBoxA) in DLL\_PROCESS\_ATTACH is going to break stuff.

More generally, doing anything complex in DllMain is inadvisable as it has a high chance of deadlocking the process and either crashing or making it unresponsive

## Dll Injection ++

- Instead of forcing a remote process to just call to LoadLibraryA, we can also invoke an exported function in a remote process!
- This allows us to execute more complex payloads
- The processes for this is going to come in handy for when we implement a more advanced version of this technique

# Steps Involved

- 1) Inject our DLL into the remote process (Same steps as last time)
- 2) Wait for the loading thread to terminate, and get the address of the injected DLL's base Address
- 3) Calculate the relative virtual address the target exported function
- 4) Add that offset to the base address of the injected DLL in the remote address
- 5) Create a thread whos entry point is the remote address of the exported function

# Calculating the Offset of the exported function

Option 1:

- Calculate it offline

# Calculating the Offset of the exported function

## Option 2:

- Call `LoadLibraryA` to get a handle to the module.
- This handle is actually a pointer to memory! In particular, for a 64 bit machine, under the hood this is an unsigned 64 bit integer
- Recall that memory grows downward from lower order addresses to higher order addresses
- To compute the offset, simply take the address of the exported function and subtract the base address

# Calculating the Offset of the exported function

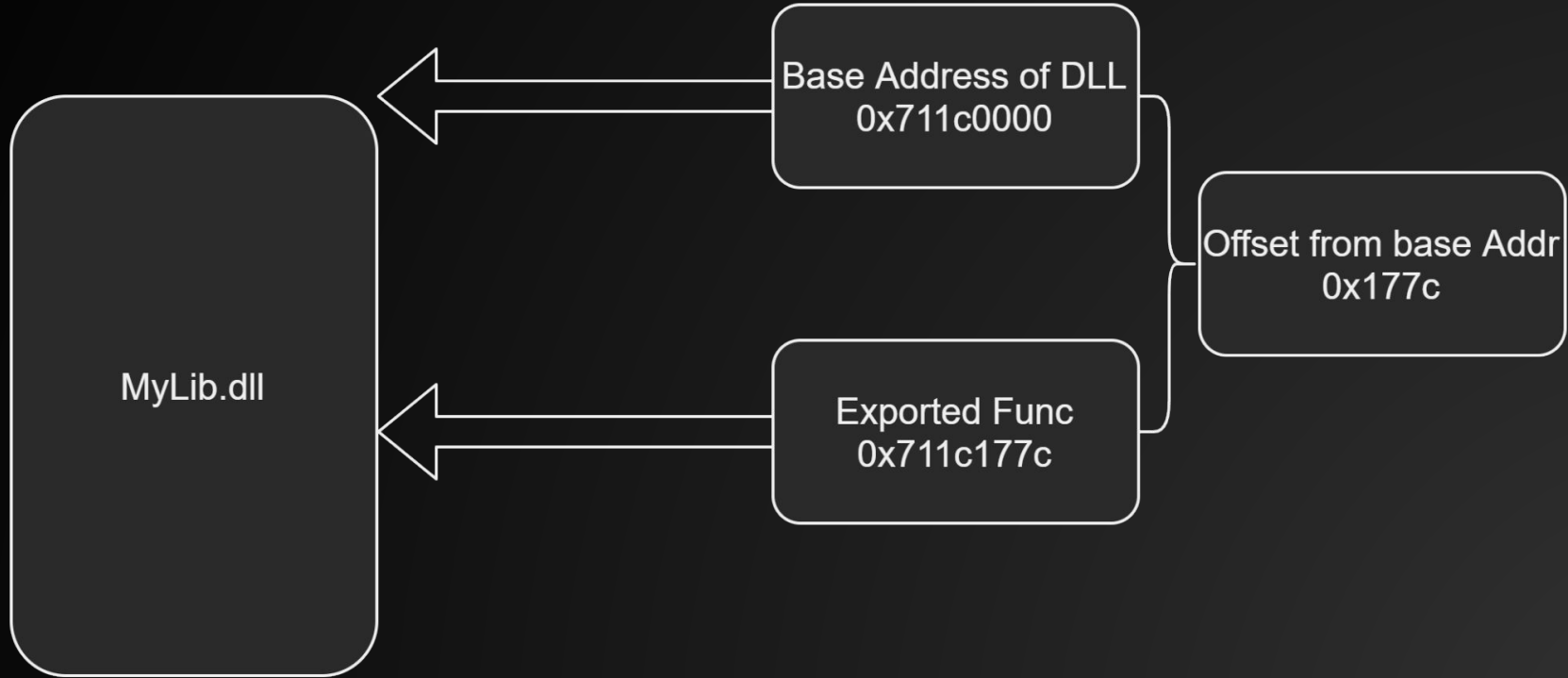
Option 3:

- Walk the export table, and calculate the RVA

How do we get the addr of the exported func?

`GetProcAddress(...)`

# Computing the Offset





# Calling our exported function

- Once we have the offset of our exported function from DLL, we can compute the address of the exported function inside of the remote process by adding the offset to the Base address!
- But how do we get the base address of the DLL in the remote process?

## Method 1

We can enumerate loaded DLLs in a remote process, and find ours.  
This is a little tedious.

## Method 2

Because we injected our DLL via `CreateRemoteThread`, the return value of the `LP_THREAD_START_ROUTINE` will be stored in the exit status! However, we only get access to the bottom 32 bits.

That is, if we wait for the loading to finish, we can read the `HMODULE` (the base address of the library in the remote process) from the exit status via

For more on this, see

<https://stackoverflow.com/questions/7100441/how-can-you-get-the-return-value-of-a-windows-thread>

## Alternative:

- Write a small bootstrap shellcode that reads the base address of Kernel32.dll, and parses GetProcAddress, and LoadLibrary.
- The shellcode loads our library, and finds its offset
- Finally, the shellcode invokes the exported function
- Developing the shellcode required to do this is beyond the scope of the class, but it is possible that Ch0nky Racoon might use this technique...hint hint nudge nudge :)



## Some Details

- The Injected DLL performs more complicated actions than just popping a message box
- The payload loads a resource from the DLL, in this case two WAV files, and plays them!
- Let's look at some common gotchas

# Getting a Handle to our DLL from our DLL

- This is a parameter in DLL Main!
- The way to handle this is to set a global variable to the handle
- This way we can reference it inside of our function
- For example, LoadResource requires a handle to the executable image to parse the resource from! We would pass the global variable set to the loaded DLL's handle as the param

## Taking it One step Further

- A current opsec failure of the current approach is the DLL is required to be on disk. LoadLibrary doesn't work with raw bytes.
- A few years ago, someone had the bright idea of reverse engineering LoadLibrary, and figuring out exactly how the Image loader works
- They then implemented their own version of it, LoadLibraryR, that doesn't require the DLL to be on disk!



# Reflective DLL injection

- Originally (publicly) published by Stephen Fewer
- Allows for the injecting of special special DLLs that have a special exported function
- Any DLL imaginable can be compiled to have this special function, and in effect allows for loading of DLLs from memory without ever touching disk
- The process of loading a DLL from memory by calling this “special” exported function is called Reflective DLL injection
- We call this special exported function the “Reflective Loader”

# Reflective DLL Injection

- Copy the Bytes of the DLL into the remote Process
- Pass execution to the Reflective Loader via CreateRemoteThread (or a small bootstrap Shellcode)
- The reflective loader Does a bunch of magic.
- Note this technique can be extended to PE injection

# The Reflective Loader

- Parse PE headers
- Find the base Address of kernel32.dll, and find LoadLibraryA, GetProcAddress, and VirtualAlloc
- Copy the sections and headers into a newly allocated region of memory
- Perform base relocations, resolve imports
- Pass execution back (either the shellcode, or the thread just exits)

# What can you do with RDI?

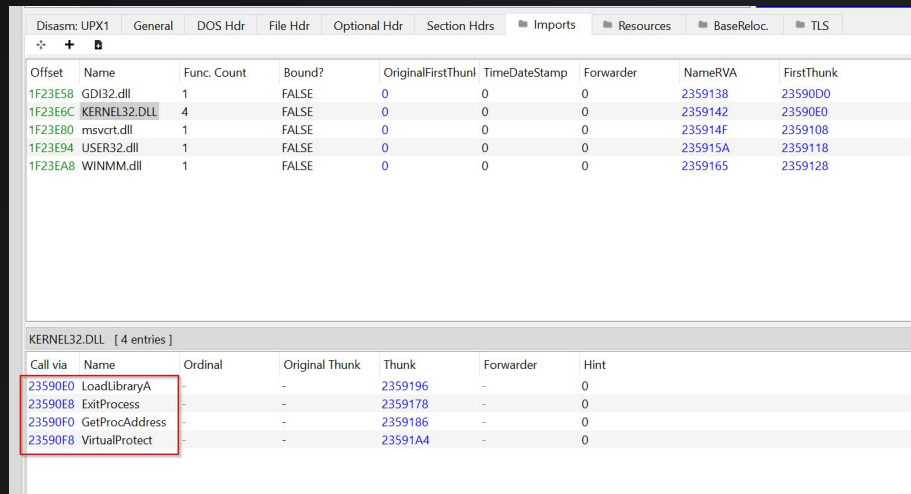
- We can download payloads entirely in memory, and execute them without touching disk
- Executables that can't defeat static indicators can instead store the bytes in an encrypted format, and load it into memory at runtime
- You can inject into benign processes (explorer.exe, svchost...etc)
- This make detecting the payload from just looking at the task manager difficult
- Packing

# Packing

- YARA rules are incredibly effective at detecting static content
- AVs catch countless threats by observing static content
- To combat this, many malware authors will “Pack” their malware to prevent those static signatures from being visible before run time.

# Indications that something is packed

- Non-standard sections (Eg UPX)
- The number of imports is small.
  - Note the exe could just be explicitly resolving imports.
- Sections with high entropy



Disasm: UPX1 | General | DOS Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports | Resources | BaseReloc. | TLS

Offset	Name	Func. Count	Bound?	OriginalFirstThunk	TimeDateStamp	Forwarder	NameRVA	FirstThunk
1F23E58	GDI32.dll	1	FALSE	0	0	0	2359138	23590D0
1F23E6C	KERNEL32.DLL	4	FALSE	0	0	0	2359142	23590E0
1F23E80	msvcrt.dll	1	FALSE	0	0	0	235914F	2359108
1F23E94	USER32.dll	1	FALSE	0	0	0	235915A	2359118
1F23EA8	WINMM.dll	1	FALSE	0	0	0	2359165	2359128

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder	Hint
23590E0	LoadLibraryA	-	-	2359196	-	0
23590E8	ExitProcess	-	-	2359178	-	0
23590F0	GetProcAddress	-	-	2359186	-	0
23590F8	VirtualProtect	-	-	23591A4	-	0

# Packers:

- Consist of packed data, and an execution stub
- The stub is used to unpack the packed data, resolve imports, and pass execution to the true entry point
- Many packers (sometimes referred to as crypters) will combine compression, obfuscation, injection, and encryption to slow down the reverse engineer
- Some can use Virtualization to make reversing very difficult.
  - Thankfully, this usually isn't the case as de-virtualization is a major pain.

# Execution

Most packers execute code by

- 1) Allocating memory (either local or remote)
- 2) Decrypting/deobfuscating the executable image bytes.
- 3) Copying the raw bytes of the executable image into that memory (this is where we want to catch it!)
- 4) Injecting the bytes (either a DLL, PE (exe), raw shellcode or other exotic formats) into either a local or remote process
- 5) The flow of execution is somehow redirected to the bytes (possibly after performing relocations, dependency resolution...etc)



## Next time

- Reflective DLL injection: the details of the reflective Loader
- Packing, and unpacking
- Detecting Reflectively Loaded DLLs