# CS-501

Introduction to Malware, Threat Hunting & Offensive Capabilities Development

## Lecture 17: Injection, Packers

-

## Recall: Dll Injection

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

```
CreateRemoteThread(hProcess,
NULL, 0,
(LPTHREAD_START_ROUTINE)
::GetProcAddress(::GetModuleHandleW(L"kernel32"),
"LoadLibraryW"),  dllBuffer, 0, &tid);
```

## Dll Injection: Last time

0

**LoadLibrary()**

Load the DLL from disk, and get a handle to it.

1

**OpenProcess**

Open a handle to a target process with the proper permissions

2

**VirtualAllocEx**

Allocate a buffer large enough to hold the full path to our dll on disk.

3

**Copy Memory**

Copy the contents of the path to our DLL into the remote buffer

4

**Execute**

Call CreateRemoteThread (or some other function) with the start address invoking LoadLibrary

## Building on DLL Injection

Limitations of our previous setup:

1) We need to copy the contents of our DLL to disk

2) Calling LoadLibrary creates an image load callback.

3) Doing anything more complex than running say MessageBoxA in DLL_PROCESS_ATTATCH is going to break stuff. More generally, doing anything complex in DllMain is inadvisable as it has a high chance of deadlocking the process

## Dll Injection ++

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

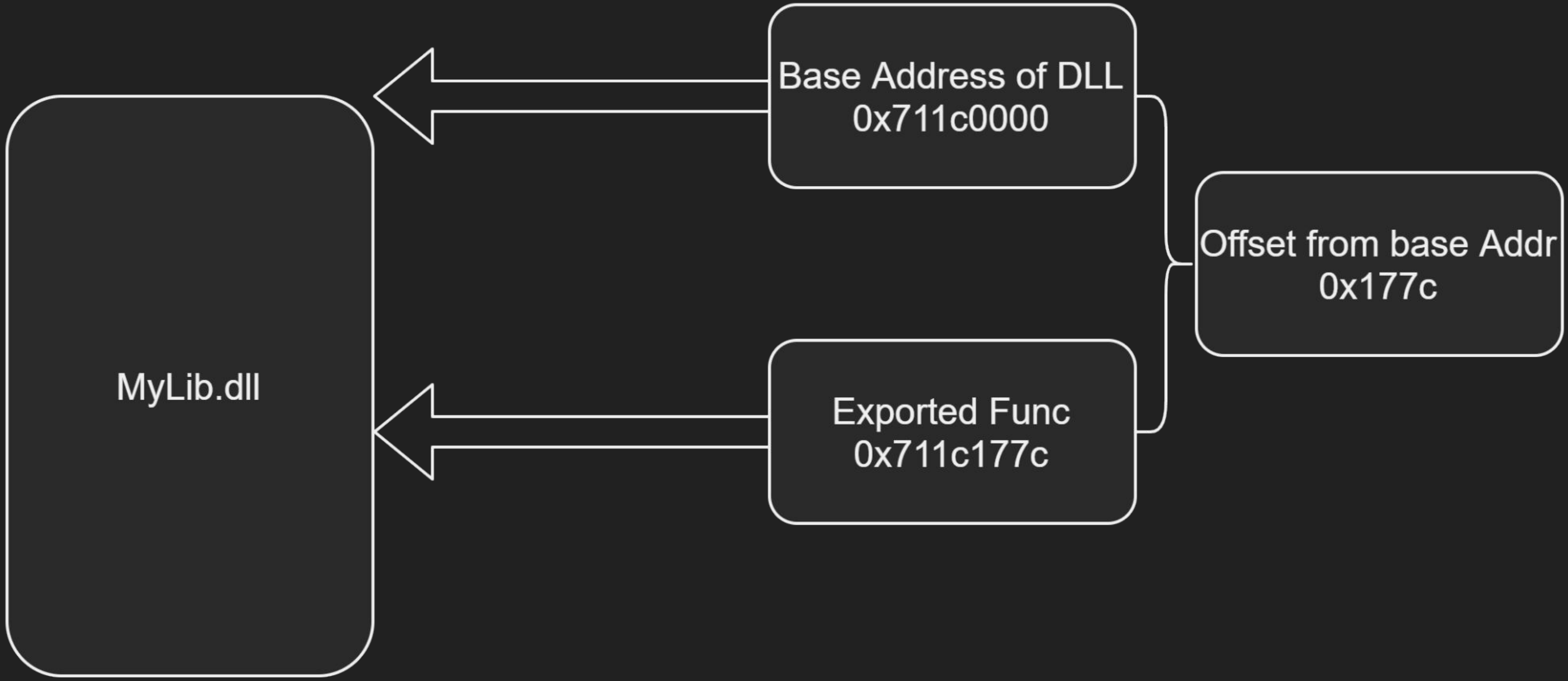
## Steps Involved

1) Inject our DLL into the remote process
2) Wait for the loading thread to terminate, and get the address of the injected DLL’s base Address
3) Load the DLL into our address space.
4) Calculate the offset from the Base address of our exported function
5) Add that offset to the base address of the injected DLL in the remote address
6) Create a thread whos entry point is the remote address

## Calculating the Offset of the exported function

- Call LoadLibraryW 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, simple take the address of the exported function and subtract the base address

## 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 exitstatus!

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 Load Library.
- The shellcode loads our library, and finds its offset
- Finally, the shellcode invokes the exported function
- The shellcode required to do this is beyond the scope of the class from a coding perspective, but it is possible that Ch0nky bear 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 imaginanable 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?

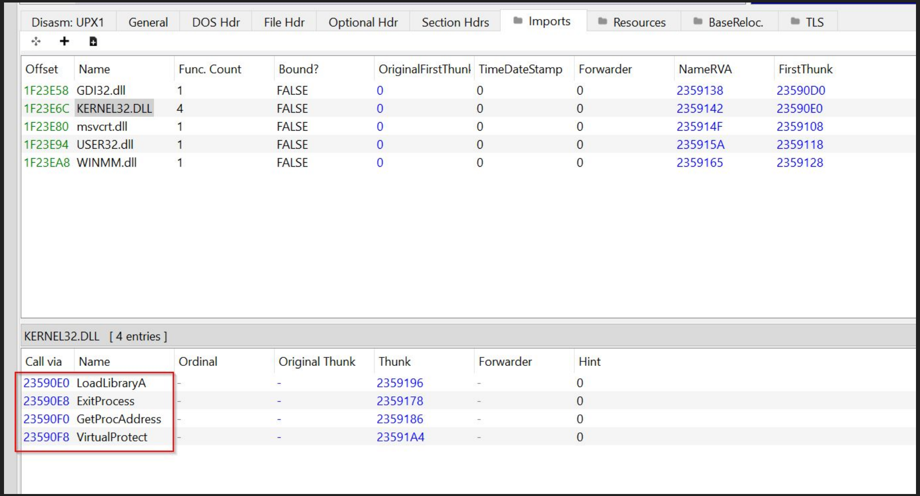
- We can download payloads entirely in memory, and execute them without touching disk
- Executables that can’t defeat static indicators can instead store the bytes in an encrypted format, and load it into memory at runtime
- You can inject into benign processes (explorer.exe, svchost...etc)
- This make 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.
- An example you have already seen thus far is the final payload from homework 1

# 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



| 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 |

KERNEL32.DLL [ 4 entries ]

| 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.

## Unpacking 101

Most packers will do some combination of the following:

1) Start with a PE and compress it
2) Encrypt the compressed PE
3) Embed the encrypted, compressed data in an executable file
4) The Executable file consists of a stub that unpacks the raw bytes of the executable, and (usually) executes the image in memory

If we can find the point right before the executable is decrypted in memory, we can dump it

## 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
-