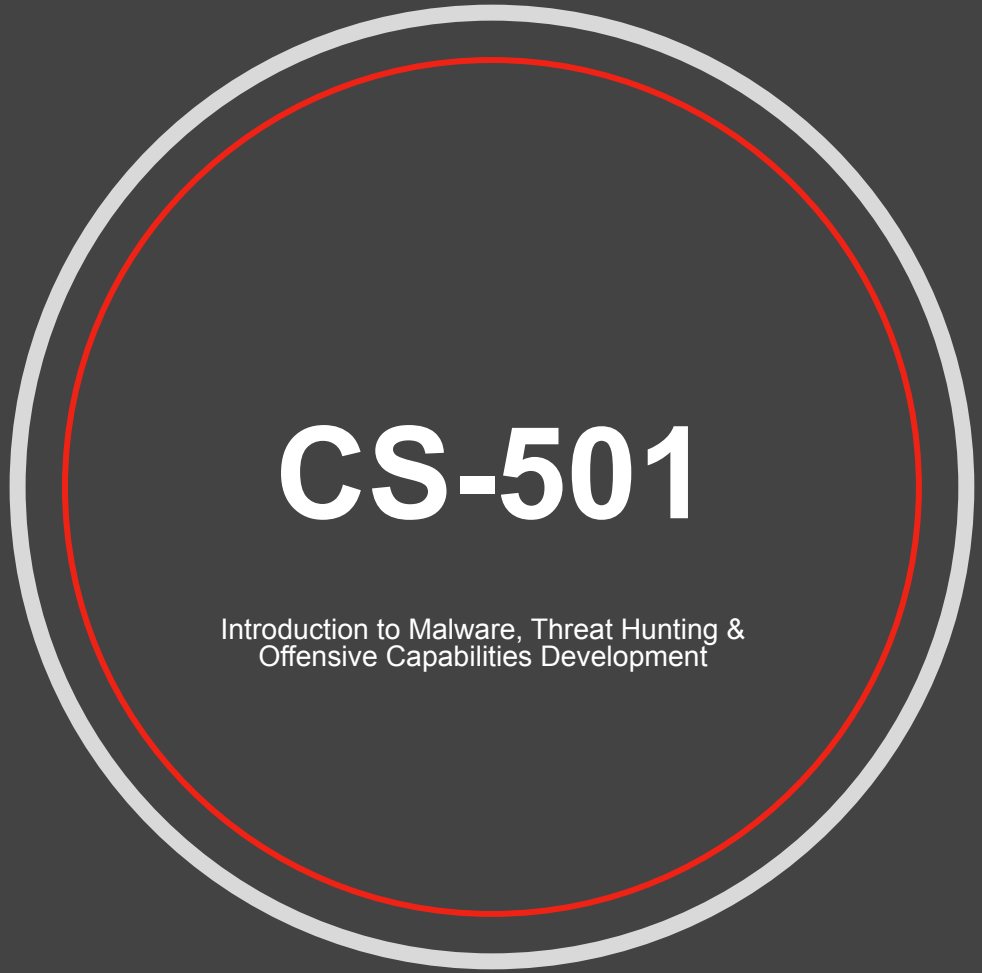

CS-501

Introduction to Malware, Threat Hunting &
Offensive Capabilities Development

Lecture 20: Packers, Remote Loading, Intro to PI

Last Time

- PE Loading
- Parsing DOS headers, Nt Headers, Optional Headers
- Walking DataDirectory

Clarification on RVA

- Recall that the virtual address of an object is the address in memory that the address starts at
- Most objects referenced in the PE file format provide an attribute called VirtualAddress which corresponds to the Relative Virtual Address (RVA)
- Recall that the RVA is the offset of from some other Virtual Address
- While *most* PEs will the RVA be the offset from the Base Address of the image, in some cases it is actually the offset from the section!

Packers: Creating a Stub

- The stub extracts the packed bytes from the PE, unpacks them, and loads the PE
- There are many ways to create a stub, but they all will more or less look the same.
- What will look different is where the PE bytes are stored, and how they are packed
- One option is to for a stub is to create an EXE that tries to load packed bytes that are stored in a resource
- We can then simply pack any PE, and embed it in the stub

Example Stub:

- Stub loads packed PE bytes from a resource in the .rsrc data section
- The Packed PE bytes are compressed using windows LZMS (Windows version of LZMA)
- The stub needs to decompress the PE bytes
- Finally, the stub needs to load the PE file

Note that we could also store the bytes in the .data, .rdata, or a custom section.

Extracting Resources

- Call `FindResource` with the appropriate resource ID to get a handle to the resource
- Get a pointer to the raw bytes associated with the resource by calling `LoadResource`
- Optionally, lock the resource
- Get the resource size
- Decompress the bytes
- Call LoadPE
- Pass Execution to the entry point

Manual Demo

Working our way to Reflective DLL Injection

- We currently have method for loading a PE in a local process
- The goal is to extend this to work for remote processes
- I.e., the same way that DLL injection forces a remote process to load a DLL from disk, we want to force a remote process to run a PE
    - Further, we to eliminate the requirement of the PE exiting on disk

Reflective DLL injection

- Force a remote process to load a special dll that can load itself!
- In particular, this DLL will have an exported function that is referred to as the reflective loader
- The reflective loader will resolve imports, perform base relocations, and pass execution as needed.
- For now, lets focus on how we can make a remote process to execute code

Process Injection Basics

- Forcing a remote process to execute code.
- We have already done this with DLL injection
- The code we forced the remote process to execute was a call to LoadLibraryA and GetProcAddress.
- From there, we calculated the offset of an exported function and called it via CreateRemoteThread
- There are many, many many ways to inject into a remote process

Key Ingredients

Most process injection requires the injector to

1) Allocate Read/Write/Executable memory to be allocated in a remote processes' address space
2) Copy the contents of the code into the remote buffer
3) Pass execution to the remote buffer

Allocating Memory

```
LPVOID VirtualAllocEx(
  [in]           HANDLE hProcess,
  [in, optional] LPVOID lpAddress,
  [in]           SIZE_T dwSize,
  [in]           DWORD  flAllocationType,
  [in]           DWORD  flProtect
);
```

The only difference between VirtualAlloc and VirtualAllocEx is we can now pass a handle to a process. When the function is called, the allocated memory will live in the remote process!

Copying Code

We can’t use memcpy, RtlMoveMemory, or something similar. We need to ask the kernel nicely to copy data from our Virtual Address space to the remote processes virtual Address space

```
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

Note that we need a pointer to the start of our code, and the size of it!

Passing Execution

- This is where you can get creative. The usual way of passing execution is to invoke CreateRemoteThread (as we did with DLL injection)
- Or you could use the native API call RtlCreateUserThread
- A great talk from Black Hat lists multiple techniques for Process Injection
- https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All.pdf

Demo: Shellcode Injection 1

Demo: Shellcode Injection 2

- *Fork and Run*: Spawn a sacrificial process
- The parent process gets a handle to the child process
  - In many situations, it is better to create the processes in a suspended state
- From there, the shellcode/PE is copied into the child processes' virtual address space from the parent process
- Execution is passed to the injected code
  - Optionally, if the Process was created in a suspended state, you resume the process

Why do we bother with Injecting DLLs

- Writing shellcode is (in my opinion) difficult
- Injecting shellcode is easy
- Injecting PEs is a little harder

See the following for solutions to convert PEs into shellcode!

- https://github.com/TheWover/donut
- https://github.com/monoxgas/sRDI

Outstanding items

Before we are ready for rDLL injection, we need to to figure out how to call LoadLibraryA and GetProcAddress in a remote process for DLLs that might not exist on disk!

There are many ways to accomplish this. If you’r code only relies on Kernel32.dll you could patch the IAT before injecting as the base address of Kernel32.dll will be the same for all processes!

Because of ASLR though, this will not work for DLLs that have been relocated

Exploring