



# Process Injection Unveiled:

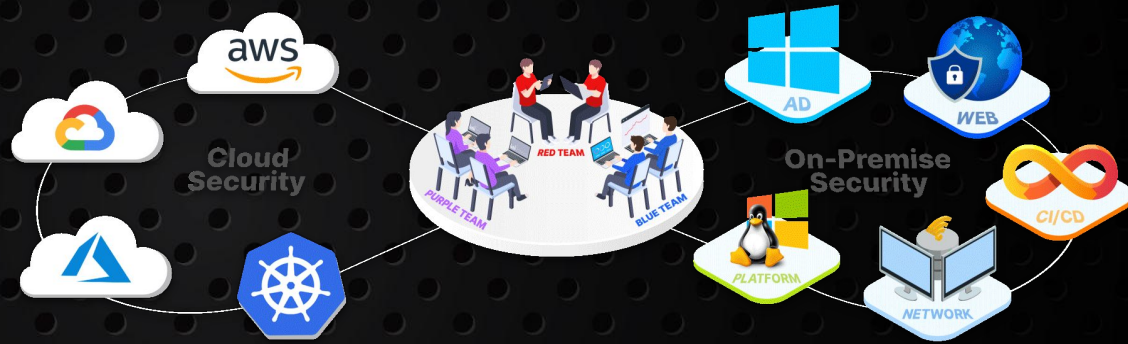
## A Deep Dive into Advanced Strategies



# About CyberWarFare Labs :

CW Labs is a renowned UK based Ed-tech company specializing in cybersecurity cyber range labs. They provide on-demand educational services and recognize the need for continuous adaptation to evolving threats and client requirements. The company has two primary divisions :

1. Cyber Range Labs
2. Up-Skilling Platform



## INFINITE LEARNING EXPERIENCE

# About Speaker :

**John Sherchan**  
**(Security Researcher)**

John Sherchan, Red Team Security Researcher at CyberwarFare Labs.

He has been working in reverse engineering, malware development and analysis, and source code review. He has very good knowledge of Windows Internals (Both User & Kernel Mode). He has reversed several

AV and EDRs to comprehend their architecture. He is currently engaged in AV/EDR evasion projects at his place of employment.

# Advance Process Injection Techniques

1. Process Injection Mindset
2. Classic Process Injection
3. APC Code Injection
4. Section Mapping
5. Module Stomping
6. Process Hollowing
7. Process Doppelganging
8. Transacted Hollowing
9. Process Herpaderping
10. Process Ghosting
11. Dirty Vanity
12. Mockingjay

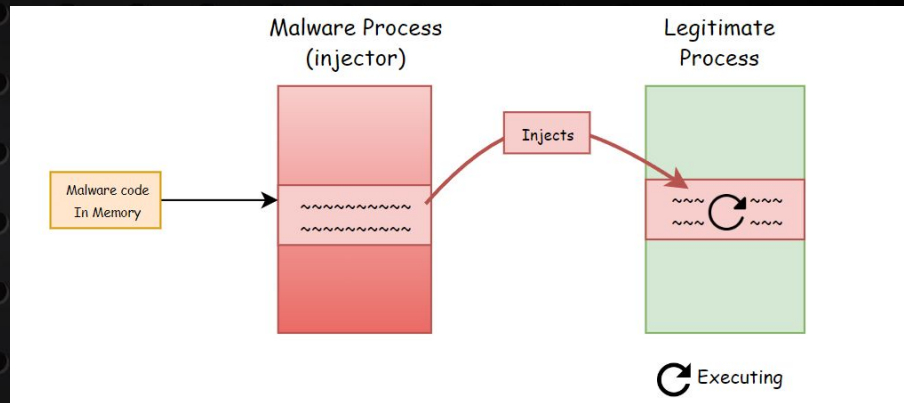
# PRE-REQUISITES

- Vmware/VirtualBox
- Windows 10 x64
  - Lab version: Windows 10 version 22H2 ( x64bit)
- Any IDE or Editor (Visual studio 2022 is preferred)
- [System Informer](#), [PE-bear](#), [CFF Explorer](#), [x64dbg](#)
- Programming Language:
  - C/C++



# Basic Mindset for Process Injection (remote)

- Injecting PE/DLL/shellcode (malicious) into another process' address space
  - ◆ To hide from the AV products
  - ◆ To hide from the naked eye of the analyst
  - ◆ Sometimes, to access the resources (network, memory, files etc.) owned by another process
  
- When performing process injection, we need to have the following queries in our mind
  - ◆ How can we access the remote process?
  - ◆ How can we send our malicious code to the remote process?
  - ◆ How can we execute our malicious code which is inside the remote process?



# Process Injection – Access Remote Process

## → Getting Access to Remote Process

- ◆ Obtain a handle to the remote process
  - Handle is value given to the user-mode processes when they try to access some object (process, thread, file, etc) from user-land

## → Obtaining the handle

- ◆ Opening a executing process
  - Win32 API: **OpenProcess**
  - NT API: **NtOpenProcess**
- ◆ Creating a new legitimate process
  - Win32 API: **CreateProcessA**
  - NT API: **NtCreateProcessEx**, **NtCreateUserProcess**
- ◆ Duplicating existing process handle from another process
  - Win32 API: **DuplicateHandle**
  - NT API: **NtDuplicateObject**

# Process Injection – Sending Malicious Code

- Many ways to send malicious code to remote process, but few queries to have in our mind
- ◆ Do we have enough privilege to write code into the remote process?
    - `PROCESS_VM_OPERATION, PROCESS_VM_WRITE`
  - ◆ Can we locate the address of the malicious code in the remote process that we just sent?
  - ◆ Is the memory region in remote process has enough memory access rights to write & execute code in that memory region?
    - Commonly we look for writable (W) & executable (X) memory region
      - But in modern OS because of security reason memory region is usually either writable or executable (W^X).



# Process Injection – Sending Malicious Code

- Usually sending/injecting malicious code in remote process involves
  - ◆ Allocating new memory region with READ, WRITE & Execute access in remote process
    - Win32 API: **VirtualAllocEx**
    - NT API: **NtAllocateVirtualMemory**
    - Access Rights: PAGE\_READWRITE, (PAGE\_READWRITE | PAGE\_EXECUTE)
  - ◆ Writing payload into the memory
    - Win32 API: **WriteProcessMemory**
    - NT API: **NtWriteVirtualMemory**
- Additionally, changing memory protection also involves in this stage
  - ◆ Usually, memory protection PAGE\_READWRITE is changed to PAGE\_EXECUTE\_READ and vice versa.
    - Win32 API: **VirtualProtectEx**
    - NtAPI: **NtProtectVirtualMemory**

# Process Injection – Execute the Malicious code

## → Common ways to perform execution

- ◆ Create a new thread in target process
  - Win32 API: **CreateRemoteThread**
  - NT API: **NtCreateThreadEx, RtlCreateUserThread**
- ◆ Queuing APC in alertable thread
  - Win32 API: **QueueUserAPC**
  - NT API: **NtQueueUserAPC**
- ◆ Hijacking the executing thread
  - Win32 API: **SetThreadContext**
  - NT API: **NtSetContextThread**

## → Last phase of the injection

- ◆ Some APIs that are used in this stage are heavily monitored by the AV/EDR products

# Process Injection – Common APIs

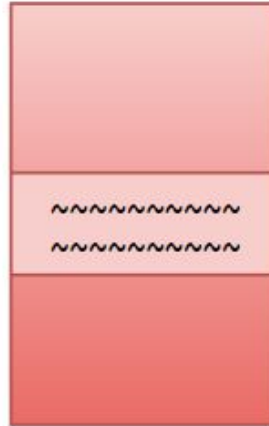
Query Process/Thread	CreateToolhelp32Snapshot, NtQuerySystemInformation, NtQueryInformationProcess, NtQueryInformationThread
Open Process/Thread	NtOpenProcess, NtOpenThread, ZwDuplicateObject
Reading Process Memory	ReadProcessMemory, NtReadVirtualMemory
Write to Process Memory	WriteProcessMemory, NtWriteVirtualMemory, ZwMapViewOfSection
Execute Code	RtlCreateUserThread, CreateRemoteThread, NtCreateThreadEx, QueueUserAPC, NtQueueUserAPC, SetThreadContext

# Classic Process Injection – steps

- Obtain Handle to a target process
  - CreateToolHelp32Snapshot, OpenProcess, NtQuerySystemInformation
- Allocate new memory region at target process
  - VirtualAllocEx, NtAllocateVirtualMemory
- Write payload into newly allocated memory
  - WriteProcessMemory, NtWriteVirtualMemory
- Create new remote thread
  - CreateRemoteThread, NtCreateThreadEx

# Classic Process Injection

Malware Process  
(injector)



Legitimate  
Process

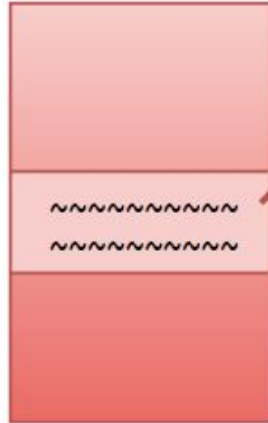


 thread Executing



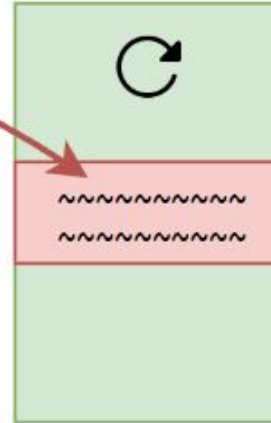
# Classic Process Injection

Malware Process  
(injector)



Injects

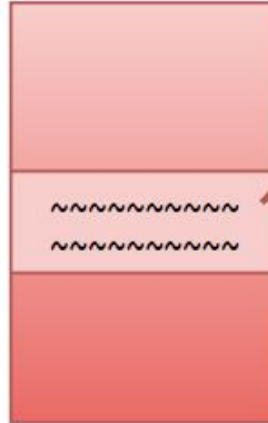
Legitimate  
Process



thread Executing

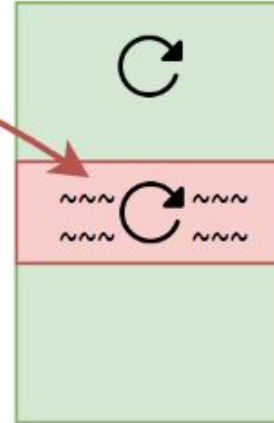
# Classic Process Injection


Malware Process  
(injector)



Injects

Legitimate  
Process



 thread Executing

# Classic Process Injection – API calls

- Kernel32.dll :
  - ◆ *CreateToolHelp32Snapshot, Process32First, Process32Next, Thread32First, Thread32Next, OpenProcess, WriteProcessMemory, VirtualProtectEx, OpenThread*
- Ntdll.dll:
  - ◆ *NtQuerySystemInformation, NtAllocateVirtualMemory, NtWriteVirtualMemory*

# APC Code Injection

- APC stands for Asynchronous Procedure Call
- APC functions execute asynchronously in context of a particular thread
- In this techniques our shellcode is placed in APC Queue of the thread.
- The payload will get executed when the thread goes to alertable state
- Wait routines puts thread in alertable state, such as:
  - ◆ *SleepEx()*
  - ◆ *WaitForSingleObjectEx()*
  - ◆ *WaitForMultipleObjectEx()*

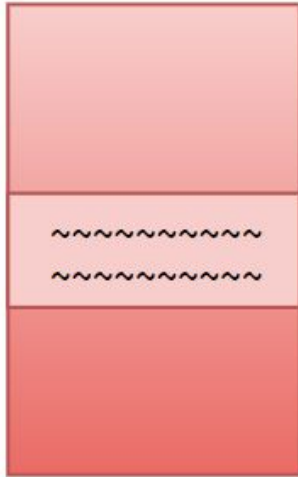
# APC Code Injection - Steps

- Find the process to inject our payload
  - ◆ *CreateToolHelp32Snapshot, NtQuerySystemInformation*
- Find all the threads in that process
  - ◆ *Thread32First, Thread32Next*
- Allocate memory in that process
  - ◆ *VirtualAllocEx, NtAllocateVirtualMemory*
- Write the payload into that allocated memory
  - ◆ *WriteProcessMemory, NtWriteVirtualMemory*
- Put the APC function in the queue for all threads
  - ◆ *QueueUserAPC, NtQueueUserAPC*
- APC function here points to our shellcode

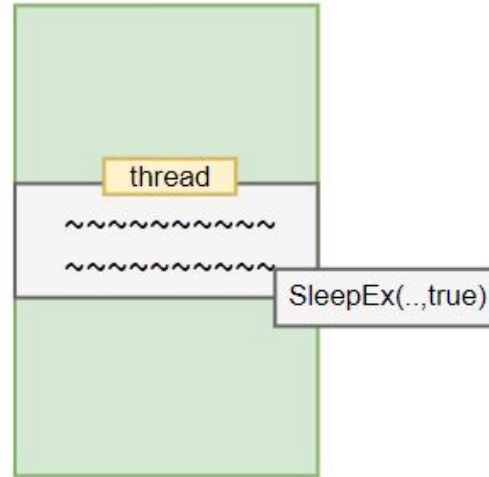


# APC Code Injection

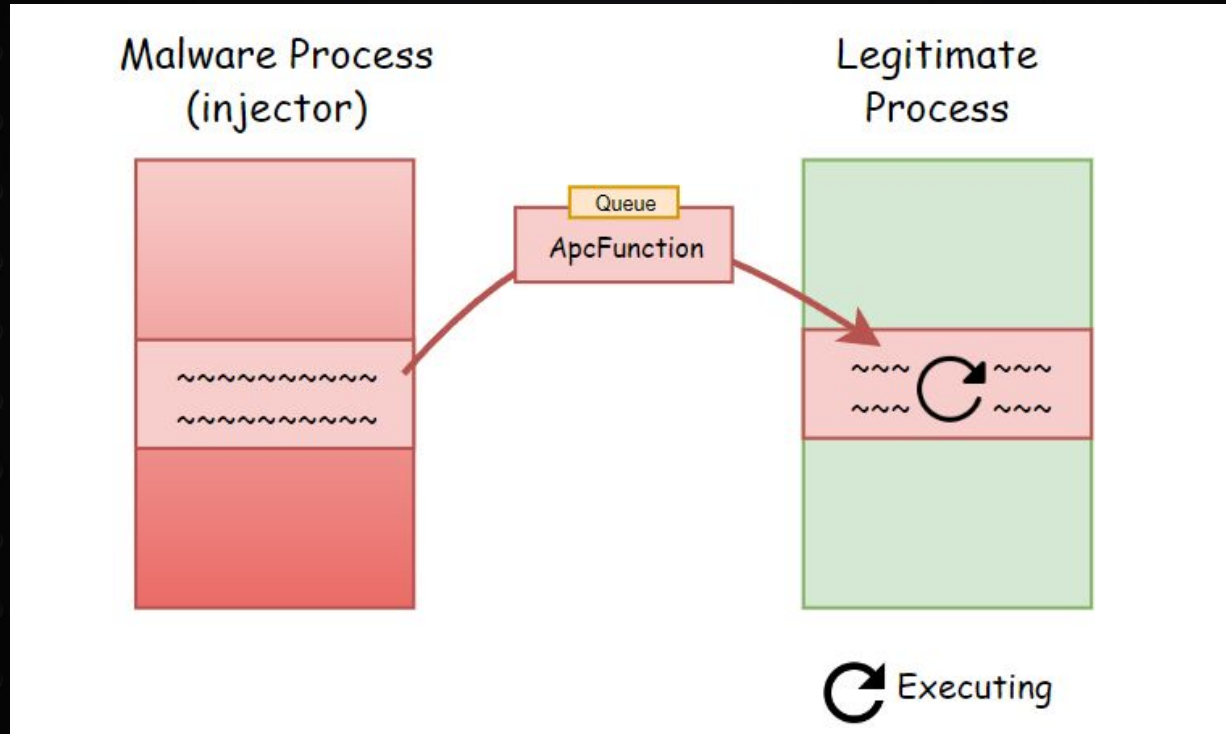
Malware Process  
(injector)



Legitimate  
Process



# APC Code Injection



# APC Code Injection - API calls

## → Kernel32.dll :

- ◆ *CreateToolHelp32Snapshot, Process32First, Process32Next, Thread32First, Thread32Next, OpenProcess, WriteProcessMemory, VirtualProtectEx, OpenThread, QueueUserAPC*

## → Ntdll.dll:

- ◆ *NtQuerySystemInformation, NtAllocateVirtualMemory, NtWriteVirtualMemory*

# Section Mapping

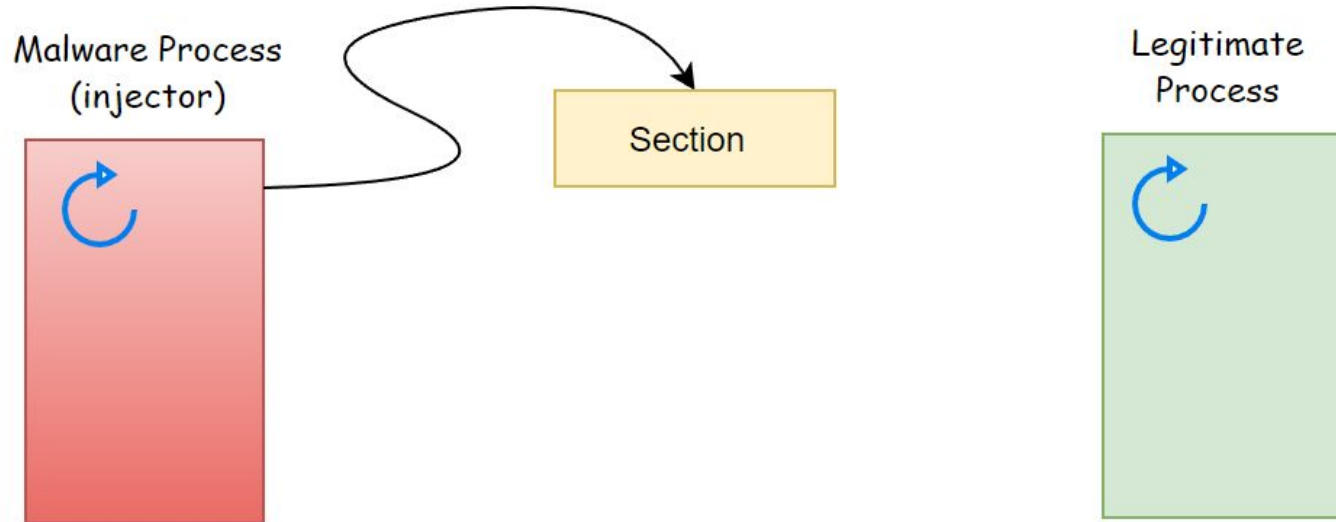
- Block of memory that can be shared between multiple processes [1]
- In memory, each section has corresponding views, which are parts of the section that are visible to processes.
  - Act of creating a view for a section is known as mapping a view of the section [1]
- In this technique a section is created and view of section is mapped to both local & target process with different page protection

# Section Mapping – Steps

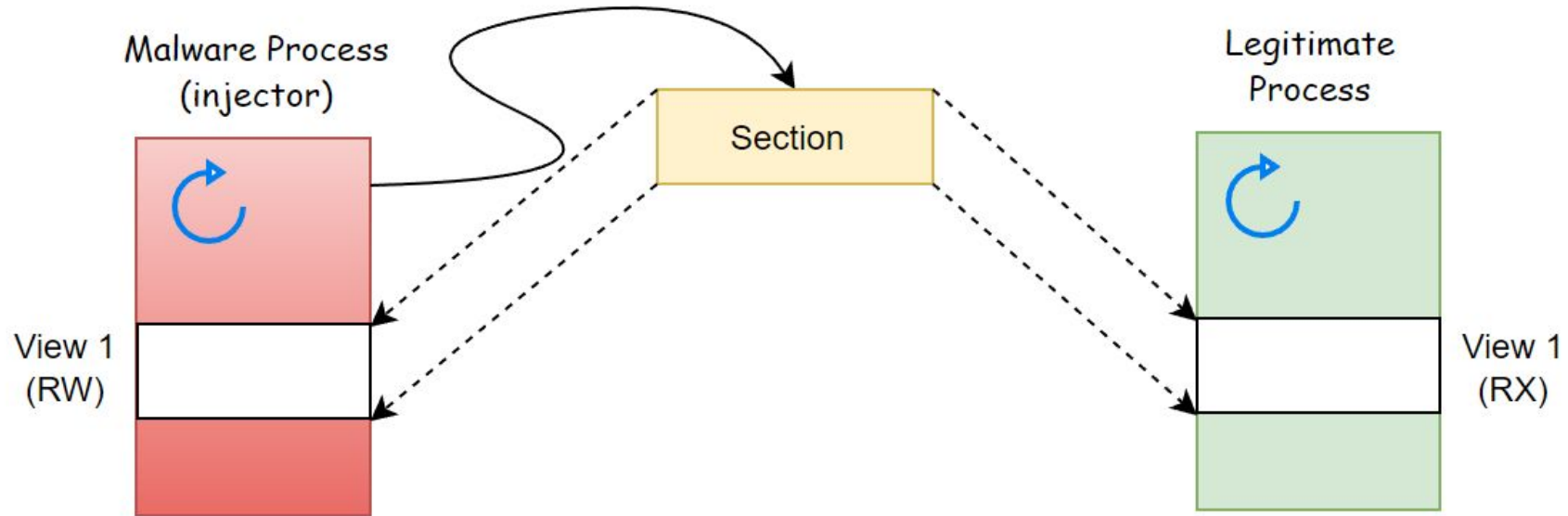
- Create a new section with full RWX page protection
  - ◆ NtCreateSection
- Map a view of section to local process (injector) with RW page protection
  - ◆ NtMapViewOfSection
- Map a view of section to target process with RX page protection
  - ◆ NtMapViewOfSection
- Write a payload to a view mapped to a local process
  - ◆ memcpy
- Create a remote thread with a base address of view mapped to remote process
  - ◆ CreateRemoteThread, NtCreateThreadEx, RtlCreateUserThread



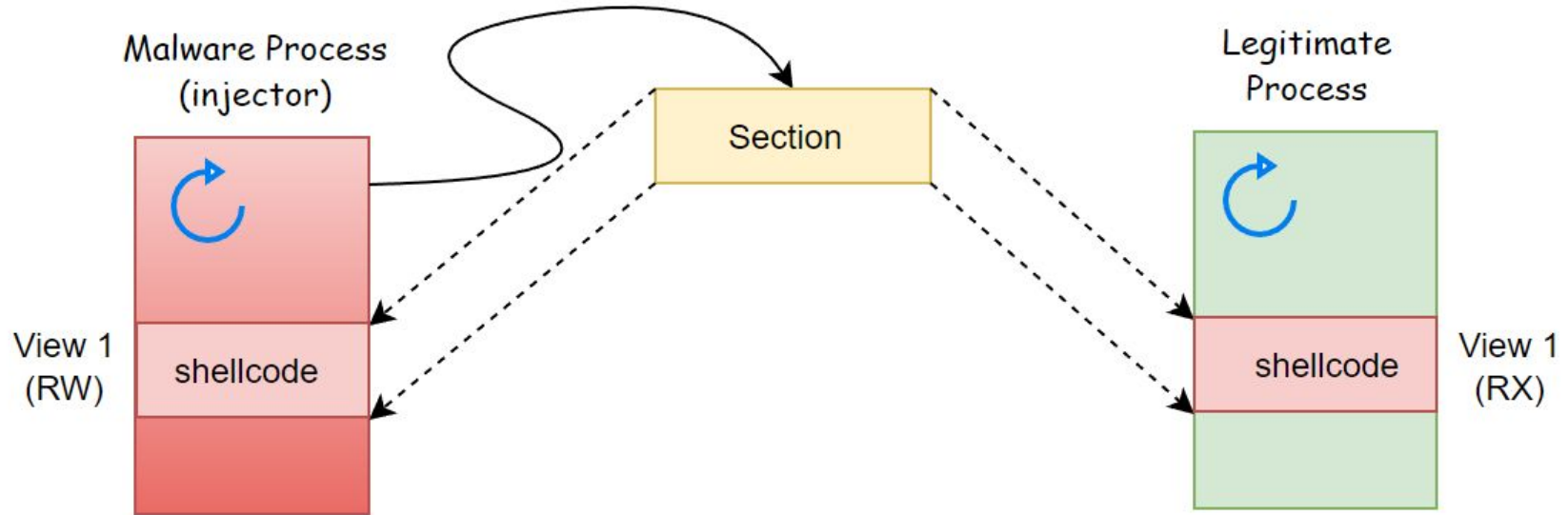
# Section Mapping



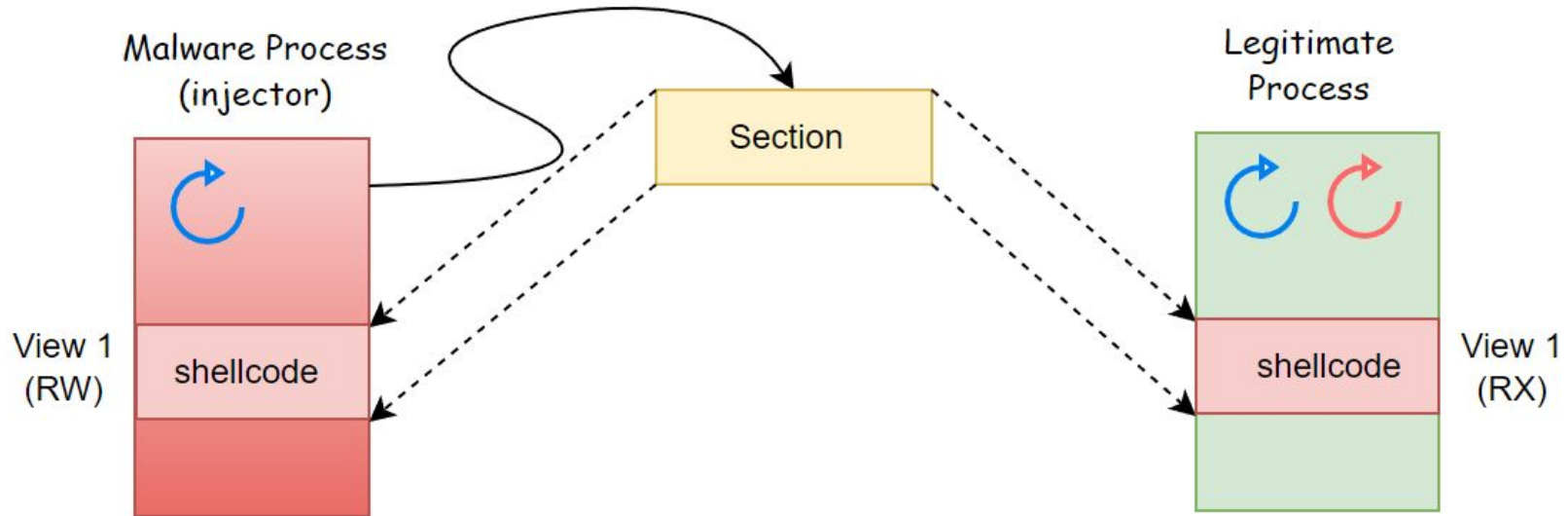
# Section Mapping



# Section Mapping



# Section Mapping



# Section Mapping – API calls

- Kernel32.dll :
  - *OpenProcess, CreateRemoteThread*
- Ntdll.dll:
  - *NtCreateSection, NtMapViewOfSection, NtCreateThreadEx*



# Module Stomping

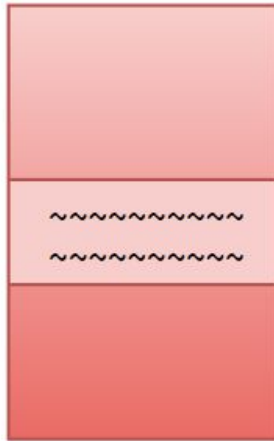
- This is the technique to load fresh dll into the target process and inject the shellcode into it
- No need to change memory protection in target process memory
- Shellcode gets executed from the legitimate dll

# Module Stomping - Steps

- Open a target process and get handle to the target process
  - ◆ *OpenProcess, NtOpenProcess*
- Load the target module in the target process
  - ◆ *VirtualAllocEx, WriteProcessMemory, CreateRemoteThread*
- Write the payload at the entrypoint address of the loaded module
  - ◆ *WriteProcessMemory*
- Create a thread to execute the payload
  - ◆ *CreateRemoteThread*

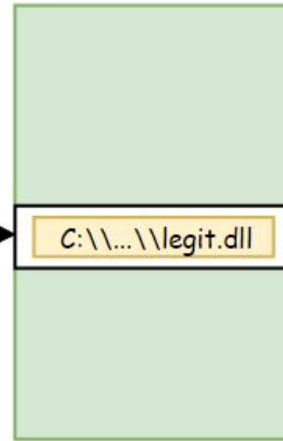
# Module Stomping

Malware Process  
(injector)



Load legitimate module

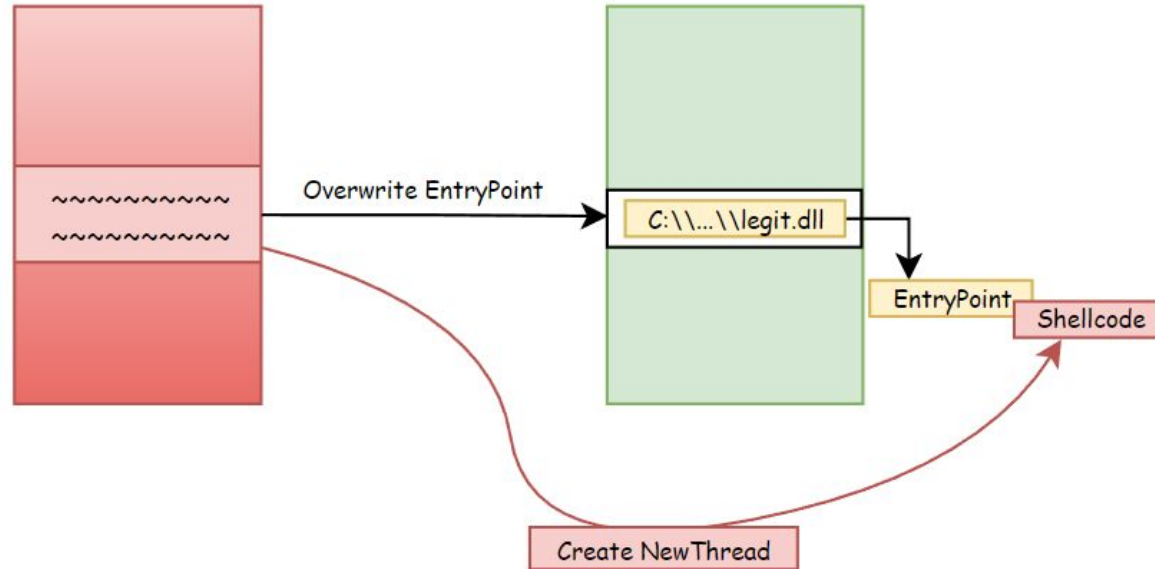
Legitimate  
Process



# Module Stomping

Malware Process  
(injector)

Legitimate  
Process



# Module Stomping – API calls

- Kernel32.dll :
  - ◆ *OpenProcess, ReadProcessMemory, WriteProcessMemory, VirtualAllocEx, VirtualProtectEx, CreateRemoteThread*
- Psapi.dll:
  - ◆ *EnumProcessModules, GetModuleFileNameEx*
- Ntdll.dll:
  - ◆ *NtAllocateVirtualMemory*



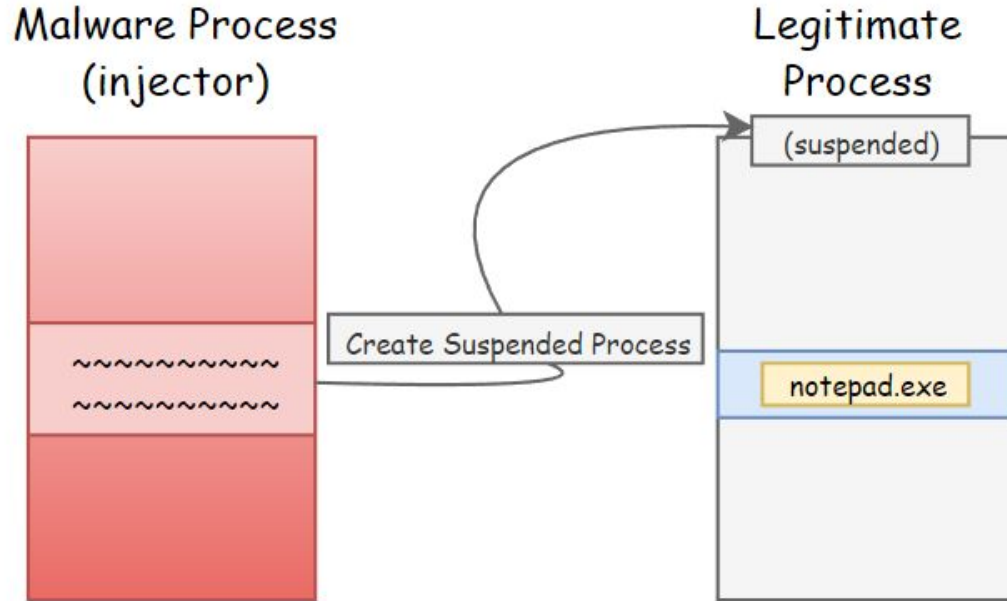
# Process Hollowing

- Replace executable section of the legitimate process with malicious executable.
- Replacement takes place in memory.
- Malicious code executes from inside of the legitimate process thus, it conceals its presence.
- The path of the hollowed process still points to the legitimate executable path.

# Process Hollowing – Steps

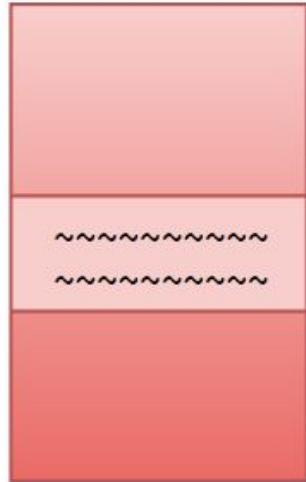
- Create target process in suspended mode
  - ◆ *CreateProcessA*
- Get Image Base Address of the target process
  - ◆ *NtQueryInformationProcess, ReadProcessMemory*
- Hollow/Unmap target image
  - ◆ *ZwUnmapViewOfSection*
- Allocate new memory in target process for the payload
  - ◆ *VirtualAllocEx*
- Copy all the payload section to the allocated memory in target process
  - ◆ *WriteProcessMemory*
- Get Context of target process
  - ◆ *GetThreadContext*
- Set the entrypoint of payload in respective context
  - ◆ *EAX for x86, RCX for x64*
- Apply the Context of target process
  - ◆ *SetThreadContext*
- Resume main thread of target process
  - ◆ *ResumeThread*

# Process Hollowing



# Process Hollowing

Malware Process  
(injector)



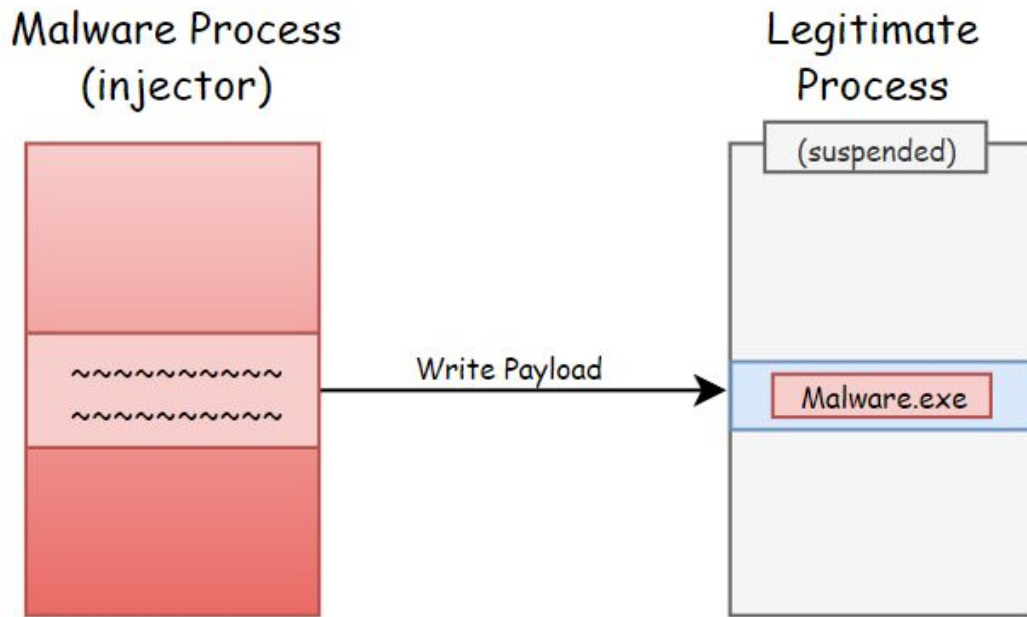
Hollow the target process



Legitimate  
Process

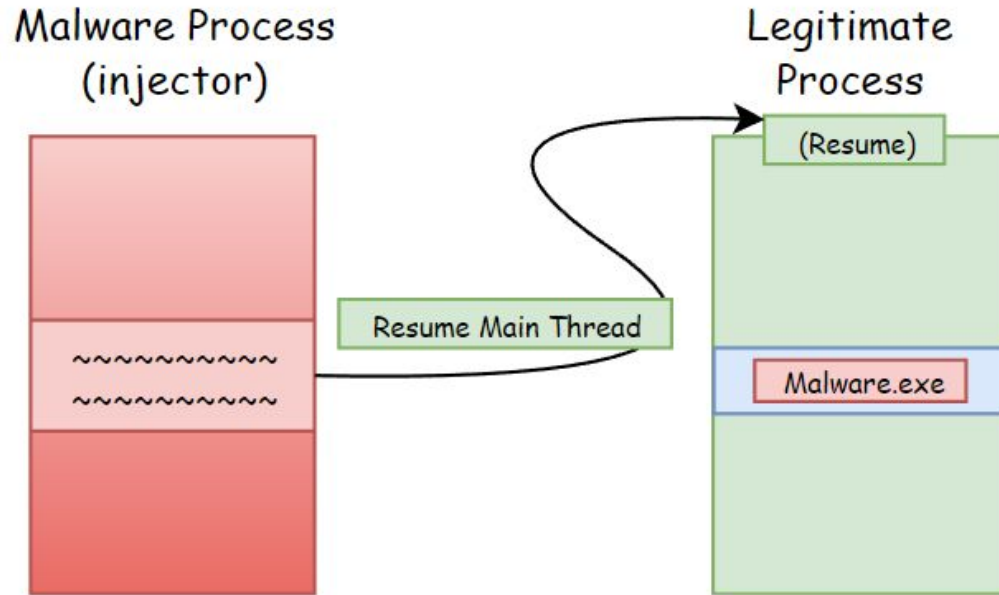


# Process Hollowing





# Process Hollowing



# Process Hollowing - API Calls

- Kernel32.dll:
  - ◆ CreateProcessA, ReadProcessMemory, WriteProcessMemory, GetThreadContext, SetThreadContext, ResumeThread
- Ntdll.dll:
  - ◆ NtQueryInformationProcess, NtUnmapViewOfSection/ZwUnmapViewOfSection

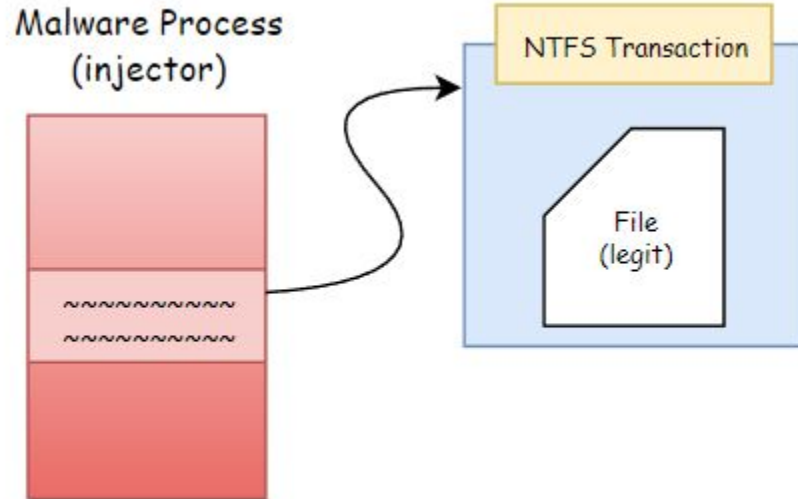
# Process Doppelganging

- Process Doppelganging utilizes the Windows API calls related to the NTFS transactions.
- Transactional NTFS brings the concept of atomic transactions to NTFS file system, which allows app developers and administrators to handle mistakes and maintain data integrity more easily.
- Transactional NTFS allows for files and directories to be created, modified, renamed and deleted atomically.
- In a series of file operations (performed in a transaction), when all operations complete successfully, the operation is committed. If an error occurs, the entire operation is rolled back and fails. This is to preserve integrity of data on disk.

# Process Doppelganging – Steps

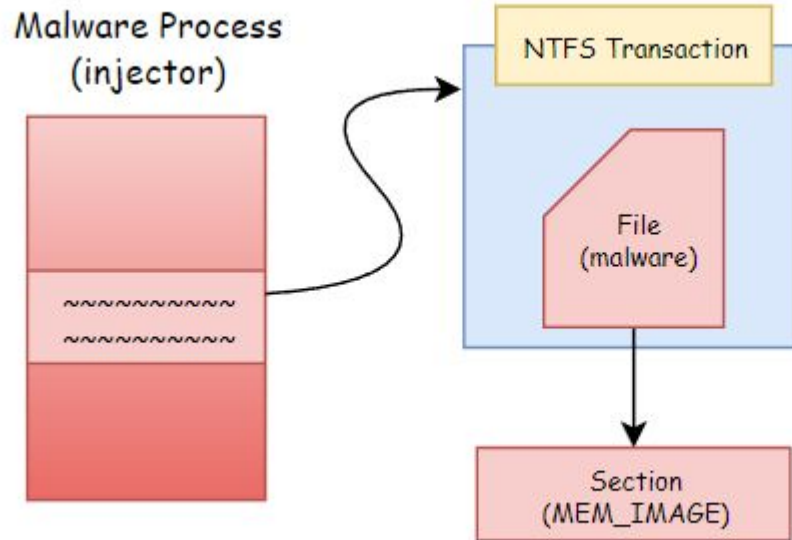
- Steps of Doppelganging can be broken down into 4 steps:
1. Transact : process a legitimate file into the NTFS transaction and then overwrite it with a malicious payload file
    - *CreateTransaction, CreateFileTransactedA*
  2. Load: Create a memory section from the payload and load the malicious code
    - *NtCreateSection*
  3. Rollback: Rollback the transaction i.e., removing malicious code so that no data left on the disk
    - *RollbackTransaction*
  4. Animate: Bringing Doppelganging to life. Create a process from the previously created memory section (step 2). The memory section contains malicious code and never written to the disk.
    - *NtCreateProcessEx, NtCreateThreadEx*

# Process Doppelganging

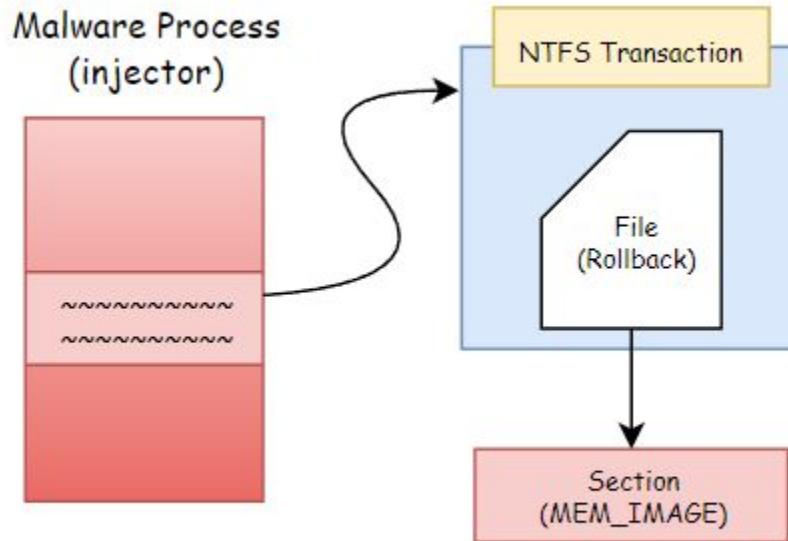




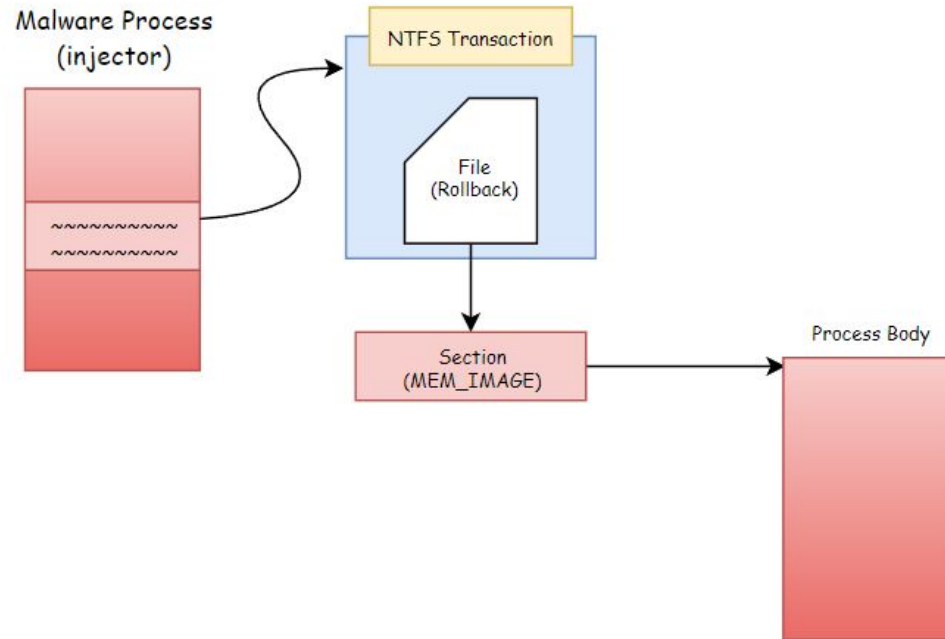
# Process Doppelganging



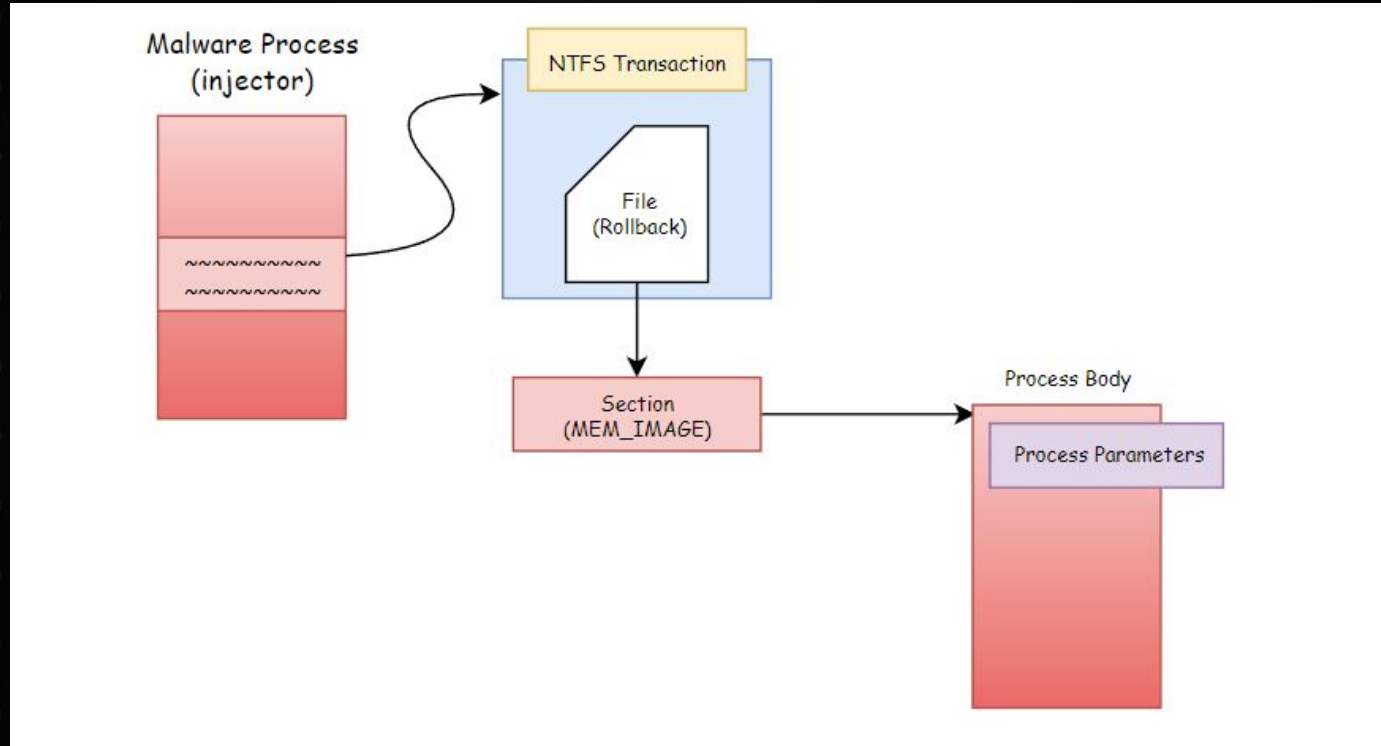
# Process Doppelganging



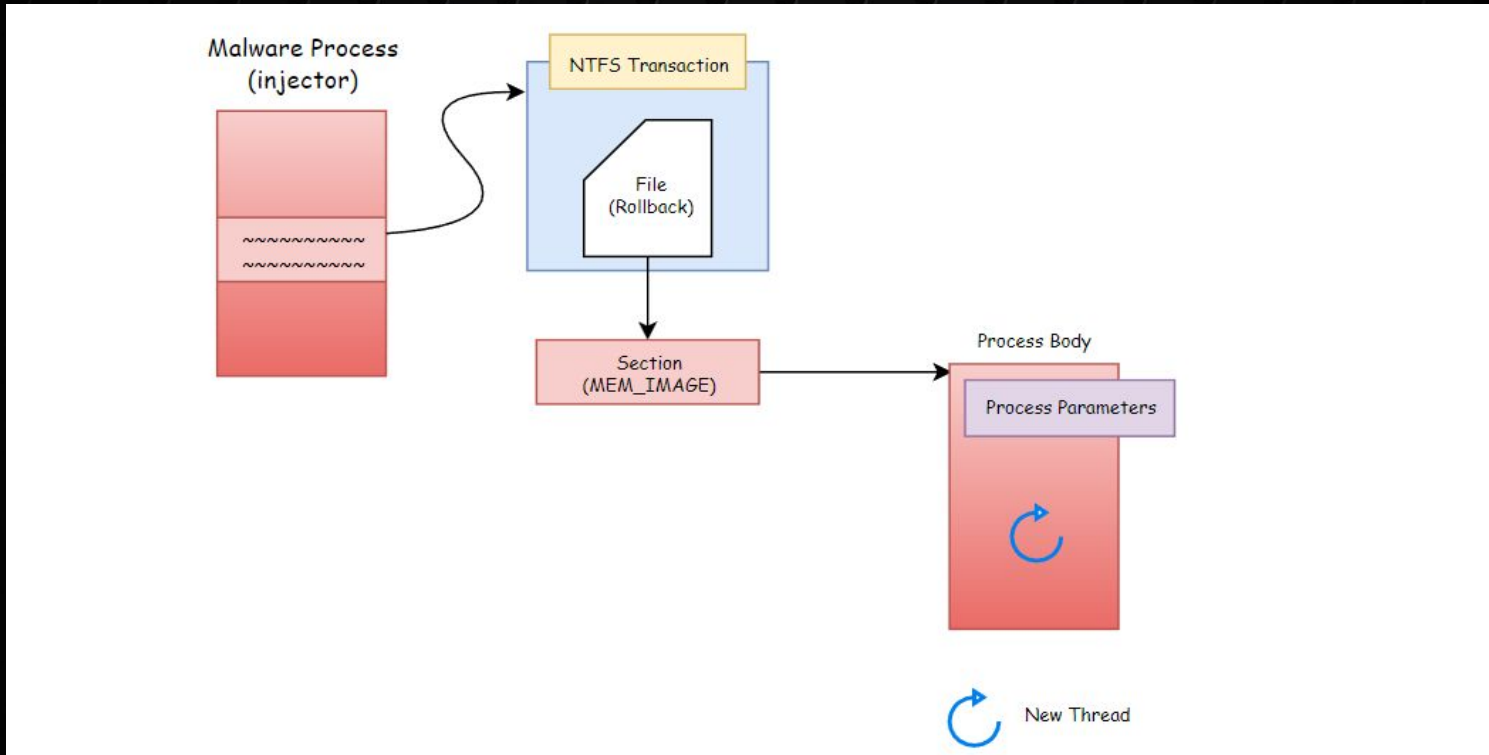
# Process Doppelganging



# Process Doppelganging



# Process Doppelganging



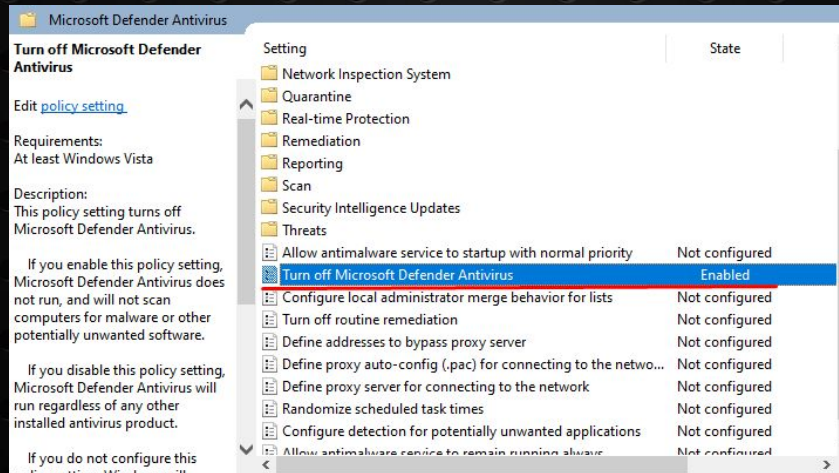


# Process Doppelganging - API Calls

- KtmW32.dll:
  - ◆ *CreateTransaction, RollbackTransaction*
- Kernel32.dll:
  - ◆ *CreateFileTransactedA, WriteFile*
- Ntdll.dll:
  - ◆ *NtCreateSection, NtCreateProcessEx, NtCreateThreadEx*

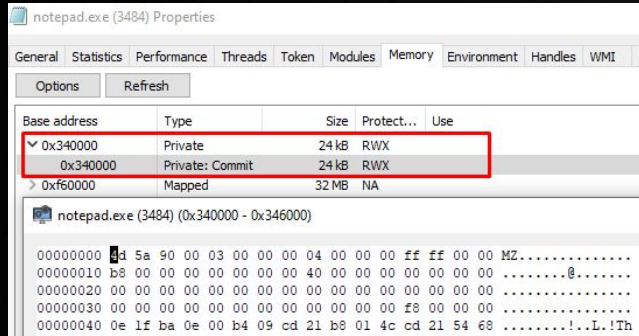
# Process Doppelganging – Issue

- Microsoft Windows Defender is monitoring the creation of remote thread using the routine PsSetCreateThreadNotifyRoutine.
- Disable Microsoft Windows Defender from Group policy.
- Computer Configuration > Administrative Templates > Windows Components > Windows Defender Antivirus > Turn off Microsoft Defender Antivirus (Enabled)

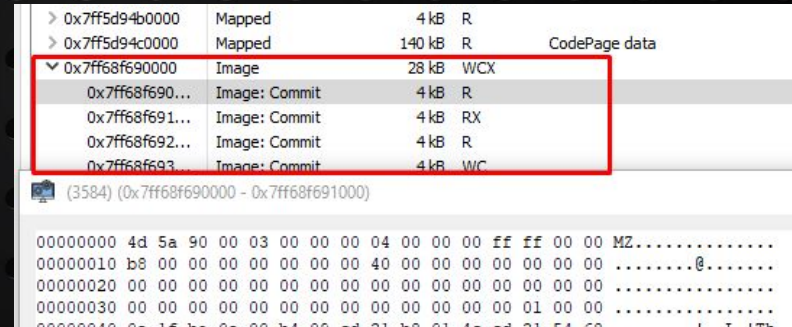


# Transacted Hollowing

- Hybrid of Process Hollowing and Process Doppelganging
- This technique solves the issues of both techniques



Process Hollowing



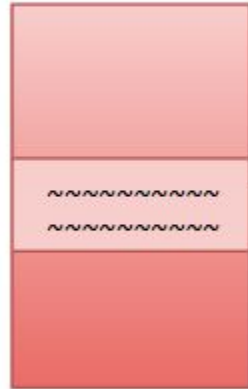
Normal/ Process Doppelganging

# Transacted Hollowing - Steps

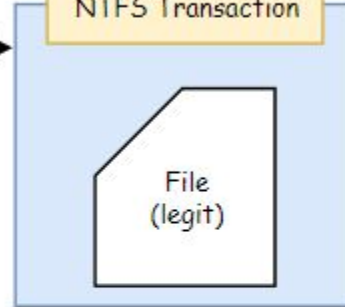
- Create NTFS transaction object
  - ◆ *CreateTransaction*
- Open/Create target file for transaction
  - ◆ *CreateFileTransactedA*
- Create an image section from transacted file
  - ◆ *NtCreateSection*
- Rollback the transaction
  - ◆ *RollbackTransaction*
- Create a new target process in suspended mode
  - ◆ *CreateProcessA*
- Map an image section into the target process
  - ◆ *NtMapViewOfSection*
- Update entrypoint in target process with payload entrypoint
  - ◆ *GetThreadContext, SetThreadContext*
- Update image base address at target process PEB with newly mapped image base address
  - ◆ *NtQueryInformationProcess, WriteProcessMemory*
- Resume the thread
  - ◆ *NtResumeThread*

# Transacted Hollowing

Malware Process  
(injector)

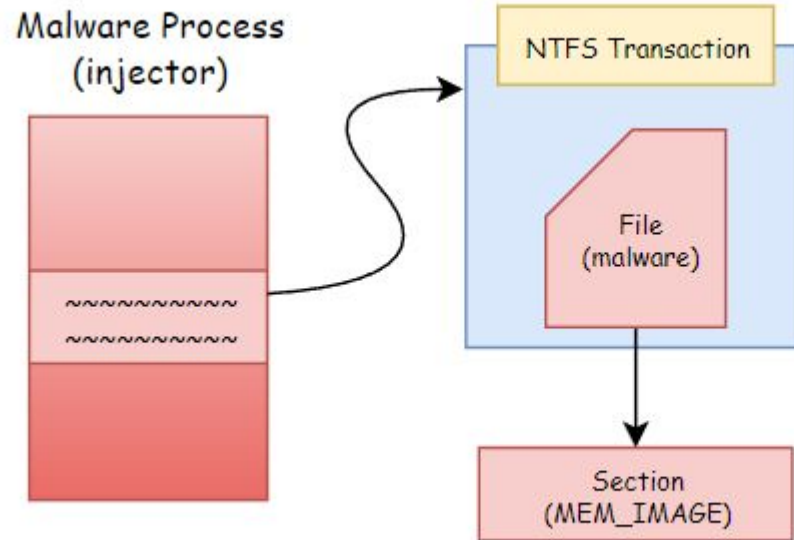


NTFS Transaction



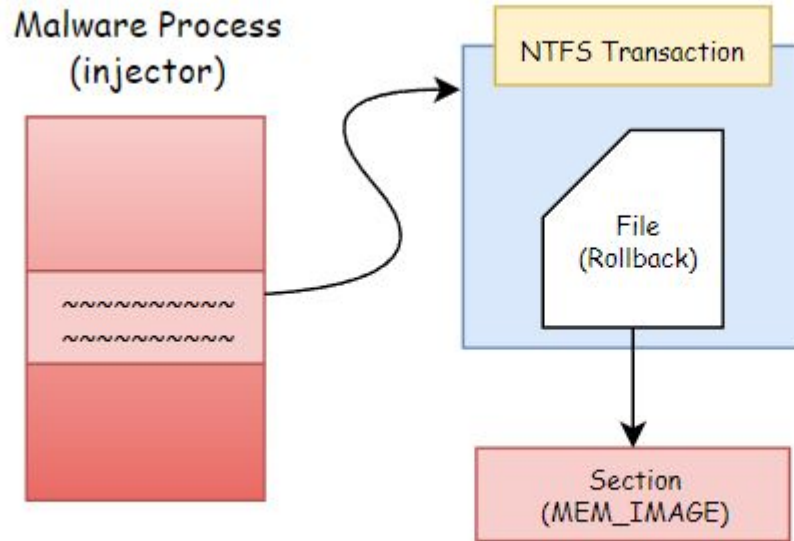


# Transacted Hollowing

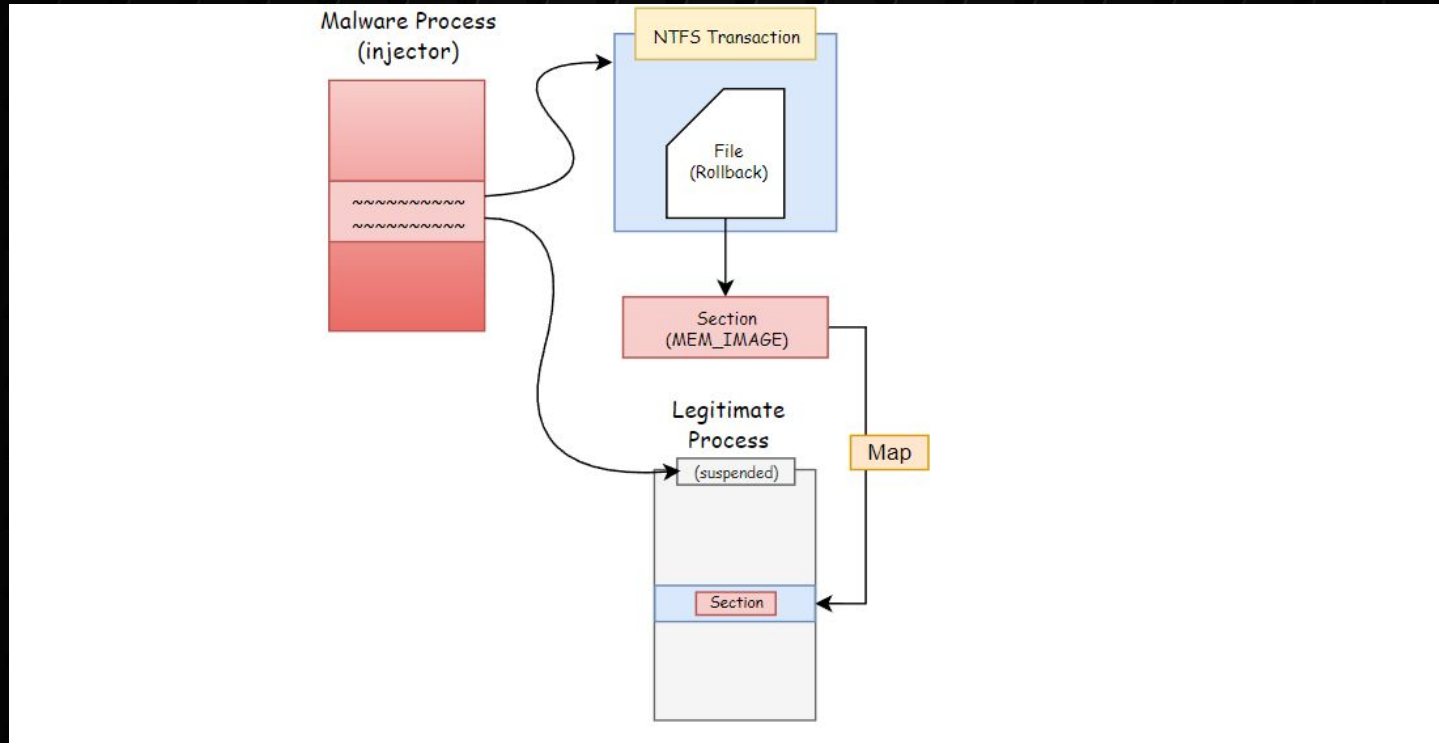




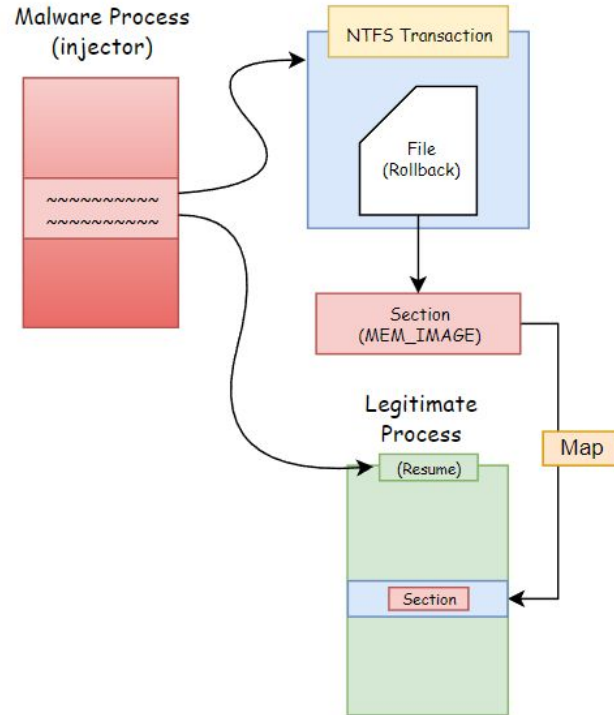
# Transacted Hollowing



# Transacted Hollowing



# Transacted Hollowing



# Transacted Hollowing - API Calls

## → Kernel32.dll:

- ◆ CreateFileTransactedW, WriteFile, CreateProcessW, ResumeThread, GetThreadContext, SetThreadContext, ResumeThread

## → Ntdll.dll:

- ◆ NtQueryInformationProcess, NtCreateTransaction, NtCreateSection, NtRollbackTransaction, NtMapViewOfSection

# Process Herpaderping

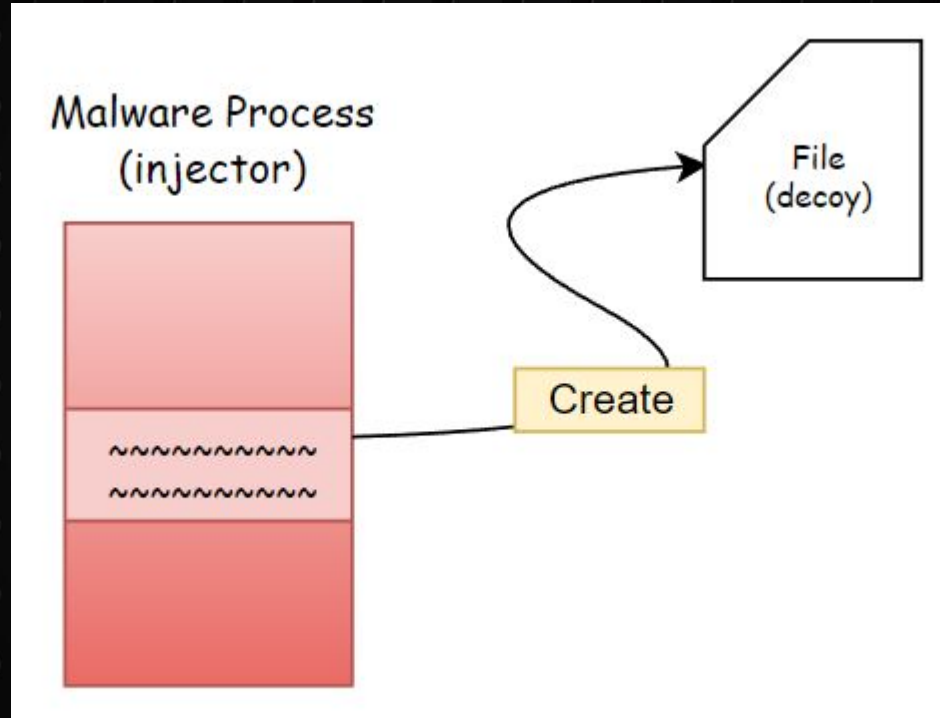
- In this technique the file on-disk is modified after the image has been mapped
- The modification is done before creating an initial thread
- Temporary file on-disk act as a decoy
- At this point the file on-disk is different from the one executed in-memory

# Process Herpaderping - Steps

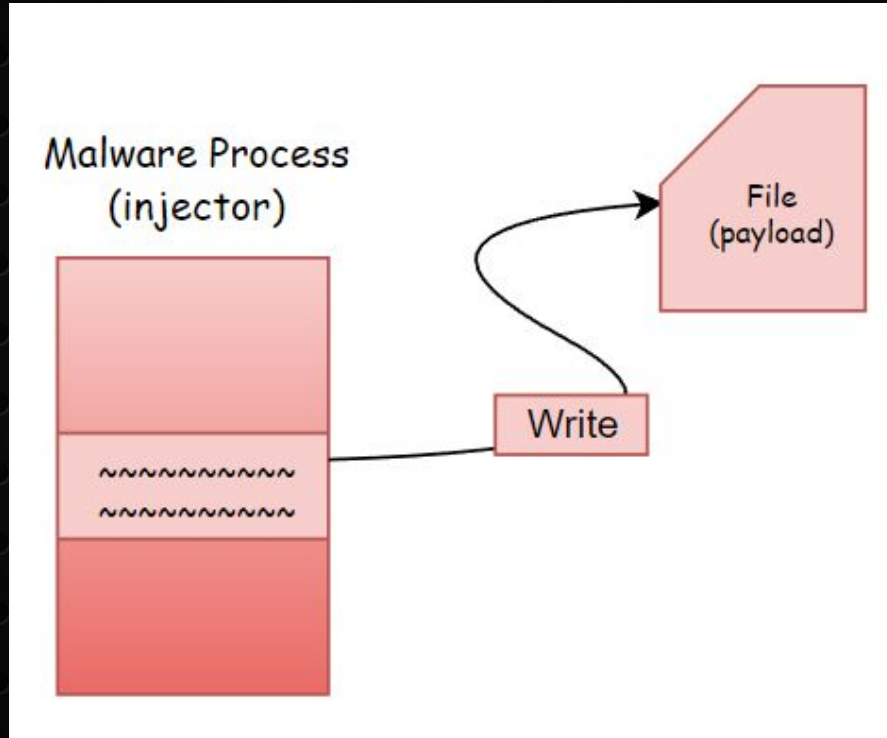
- Create a temp/decoy file
  - ◆ *CreateFileA*
- Write payload into that file (do not close the temp file handle after writing payload into it)
  - ◆ *WriteFile*
- Create an image section from that file
  - ◆ *NtCreateSection*
- Create a process using the newly created section
  - ◆ *NtCreateProcessEx*
- Modify the temp file
  - ◆ *SetFilePointer, WriteFile*
- Setup process parameters
  - ◆ *RtlCreateProcessParametersEx*
- Create new thread
  - ◆ *NtCreateThreadEx*
- Close temp file handle
  - ◆ *CloseHandle*



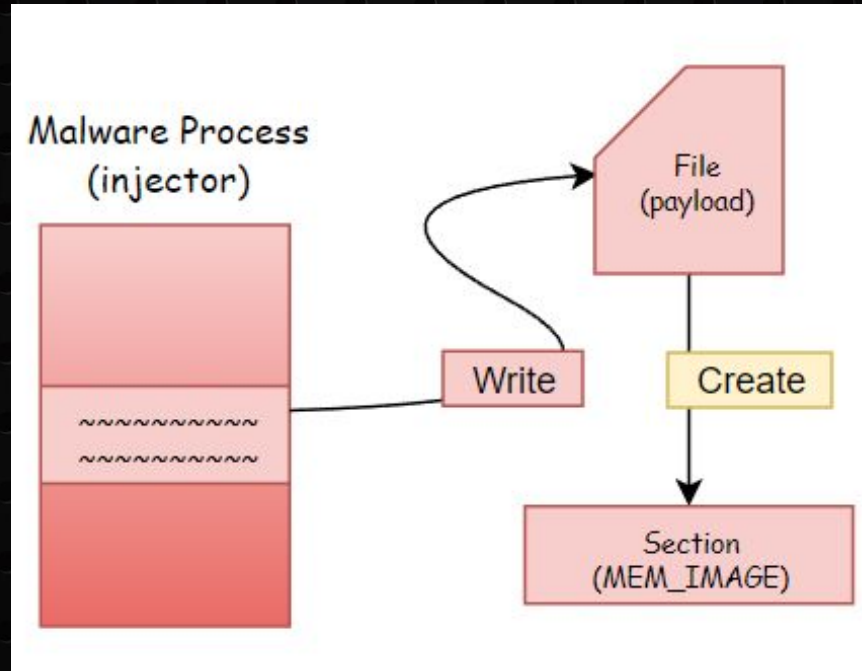
# Process Herpaderping



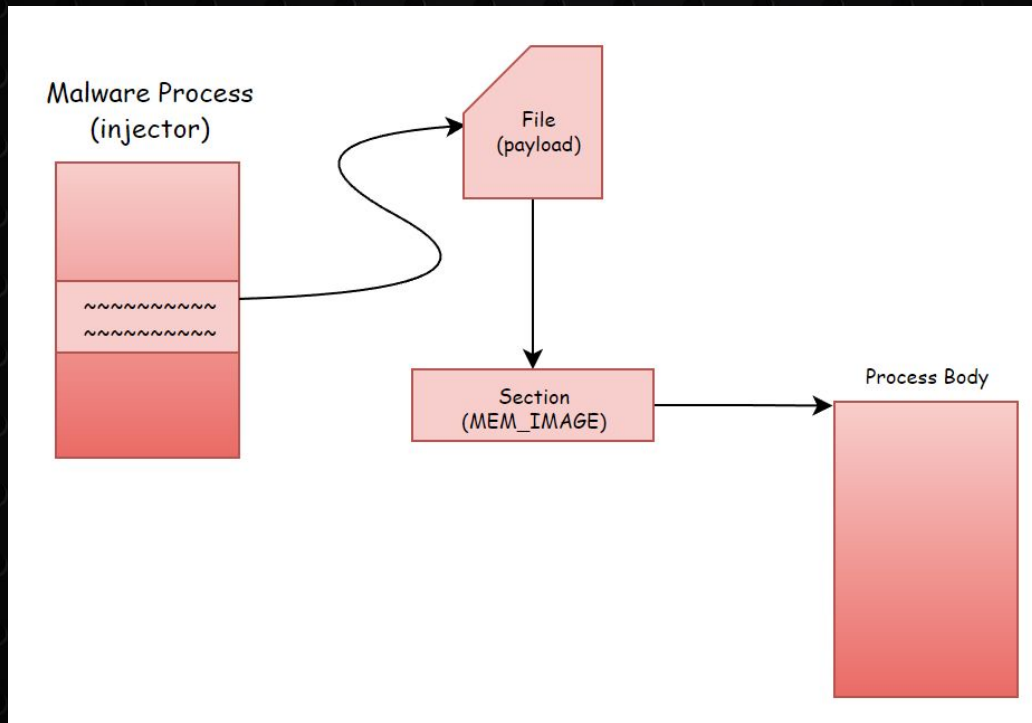
# Process Herpaderping



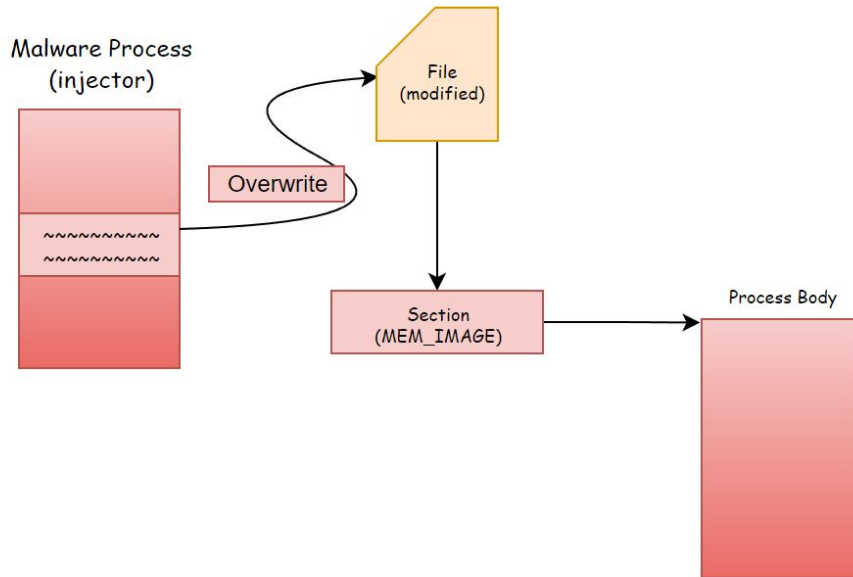
# Process Herpaderping



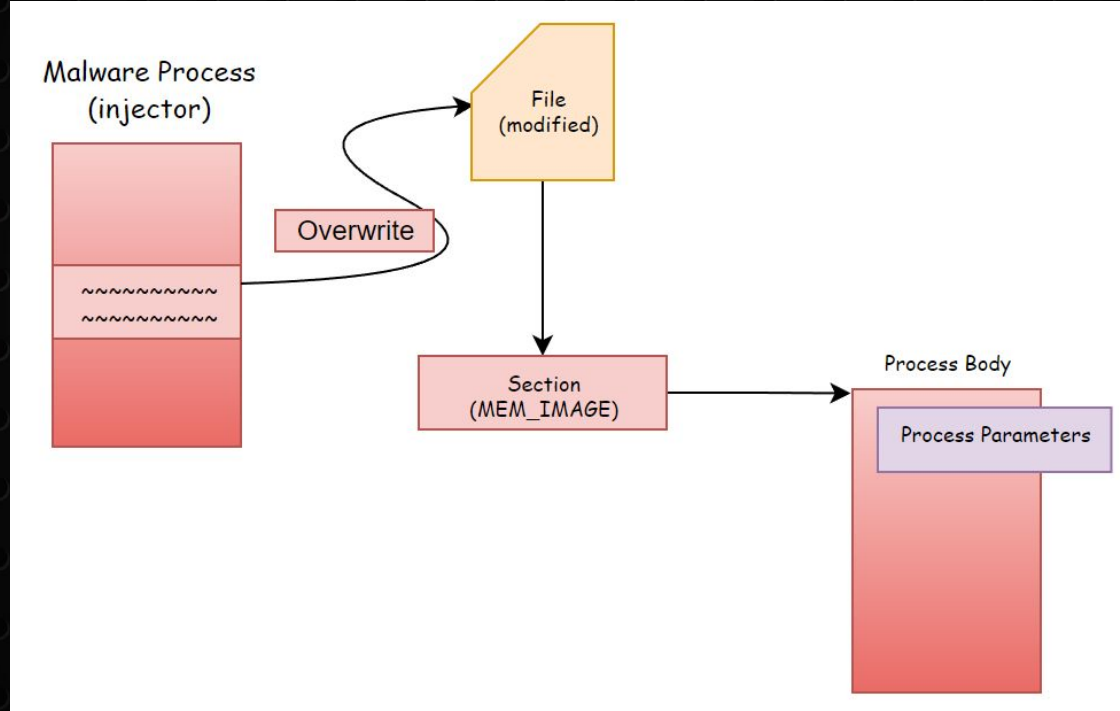
# Process Herpaderping



# Process Herpaderping

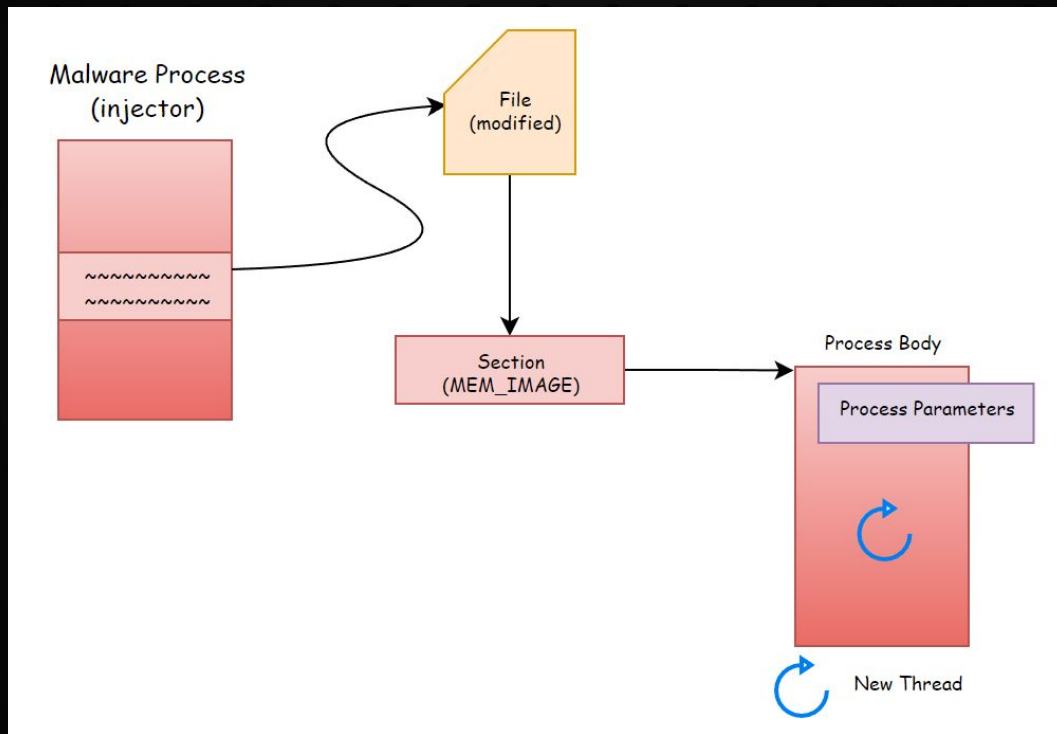


# Process Herpaderping





# Process Herpaderping



# Process Herpaderping – API Calls

→ Kernel32.dll:

- ◆ CreateFileW, WriteFile, SetFilePointer, CloseHandle

→ Ntdll.dll:

- ◆ NtOpenFile, NtSetInformationFile, NtCreateSection, NtCreateProcessEx, NtCreateProcessParametersEx, NtCreateThreadEx

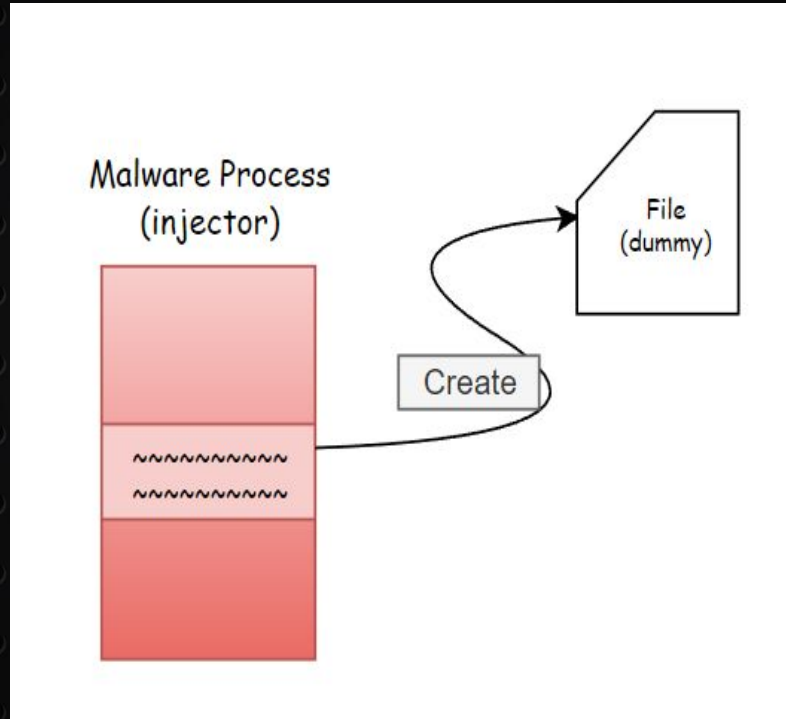
# Process Ghosting

- Similar to process doppelganging
- However, the section is created using delete-pending file instead of transaction.
- Puts a file in delete-pending state which makes antivirus tools difficult to scan or delete it.
- Before creating the process the file is completely vanished and we're left out with file-less section.

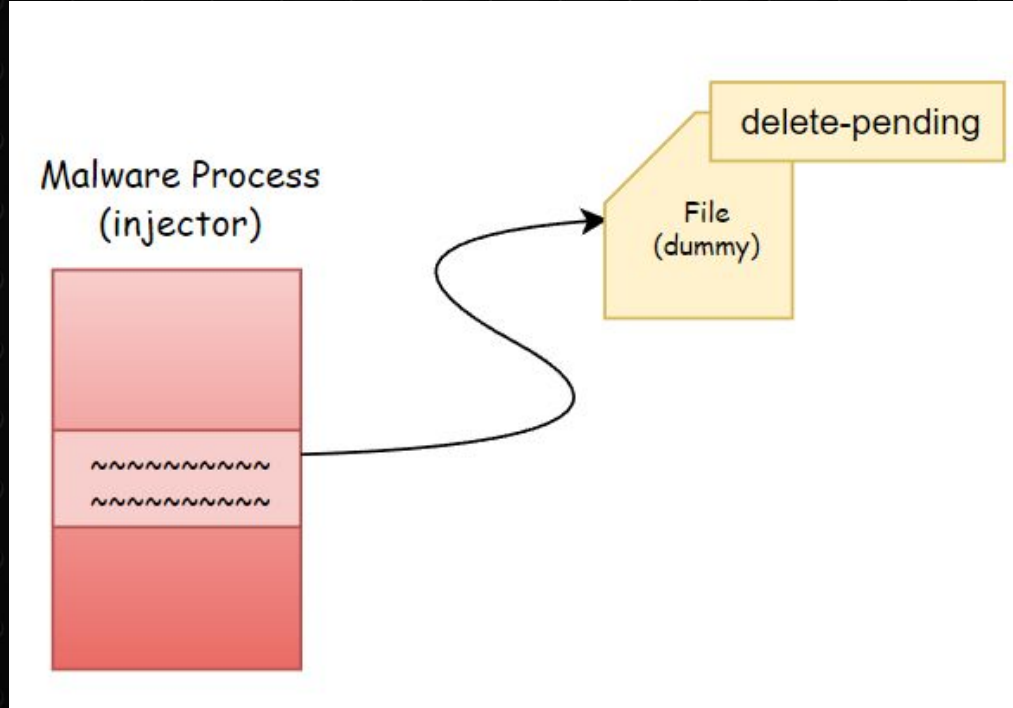
# Process Ghosting – Steps

- Open/Create new dummy file
  - ◆ *CreateFileA*
- Put the file into delete-pending state using API *NtSetInformationFile*
  - ◆ **FileDispositionInformation** information class is used here
- Write payload buffer into delete-pending file
  - ◆ *WriteFile*
- Create an image section with the delete-pending file
  - ◆ *NtCreateSection*
- Close delete-pending file handle
  - ◆ *CloseHandle*
- Create a process with newly created image section using API *NtCreateProcessEx*
- Update/fix process parameters
  - ◆ *RtlCreateProcessParametersEx*
- Create a new thread
  - ◆ *NtCreateThreadEx*

# Process Ghosting

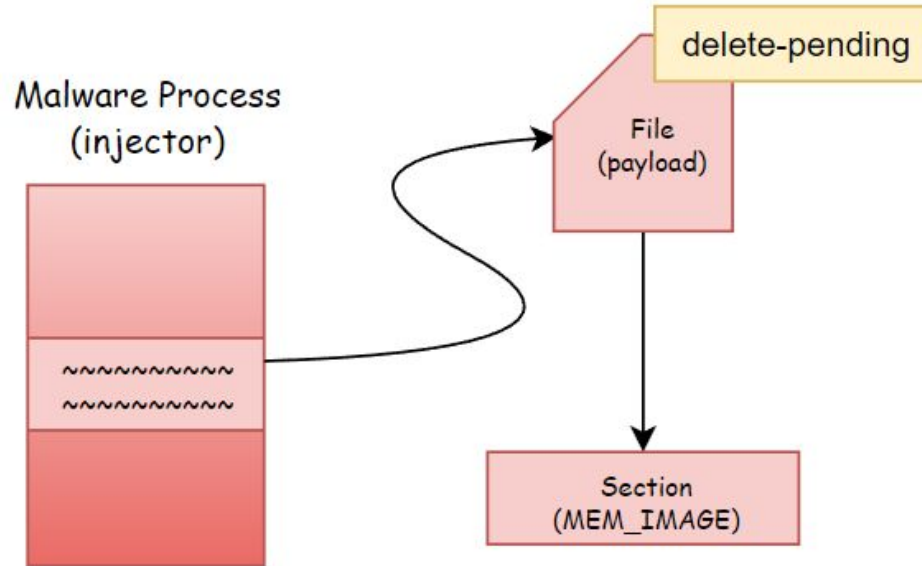


# Process Ghosting

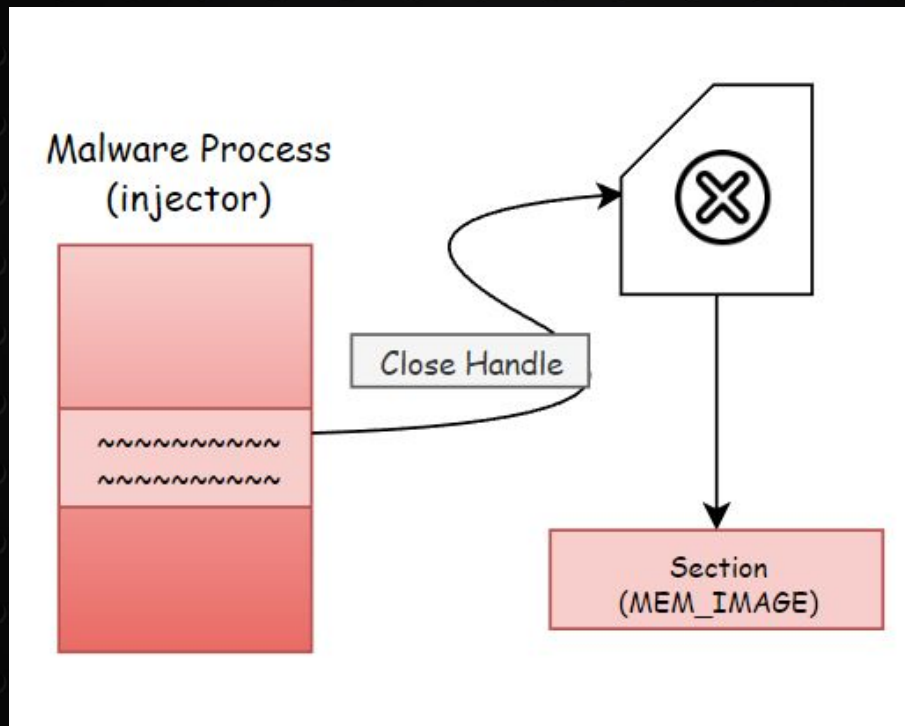




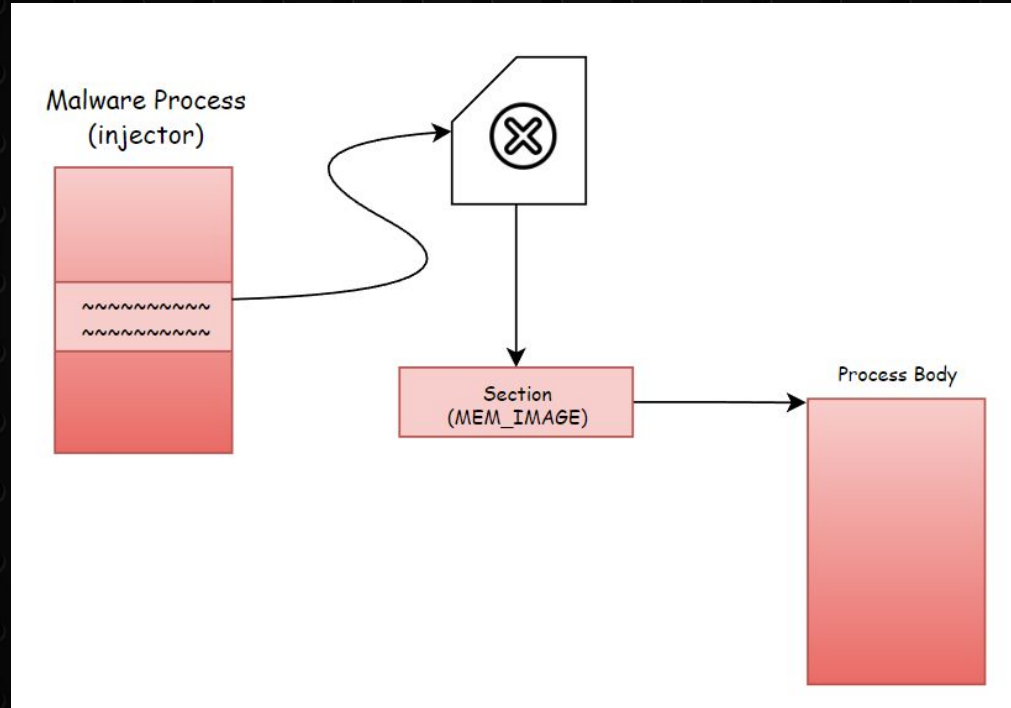
# Process Ghosting



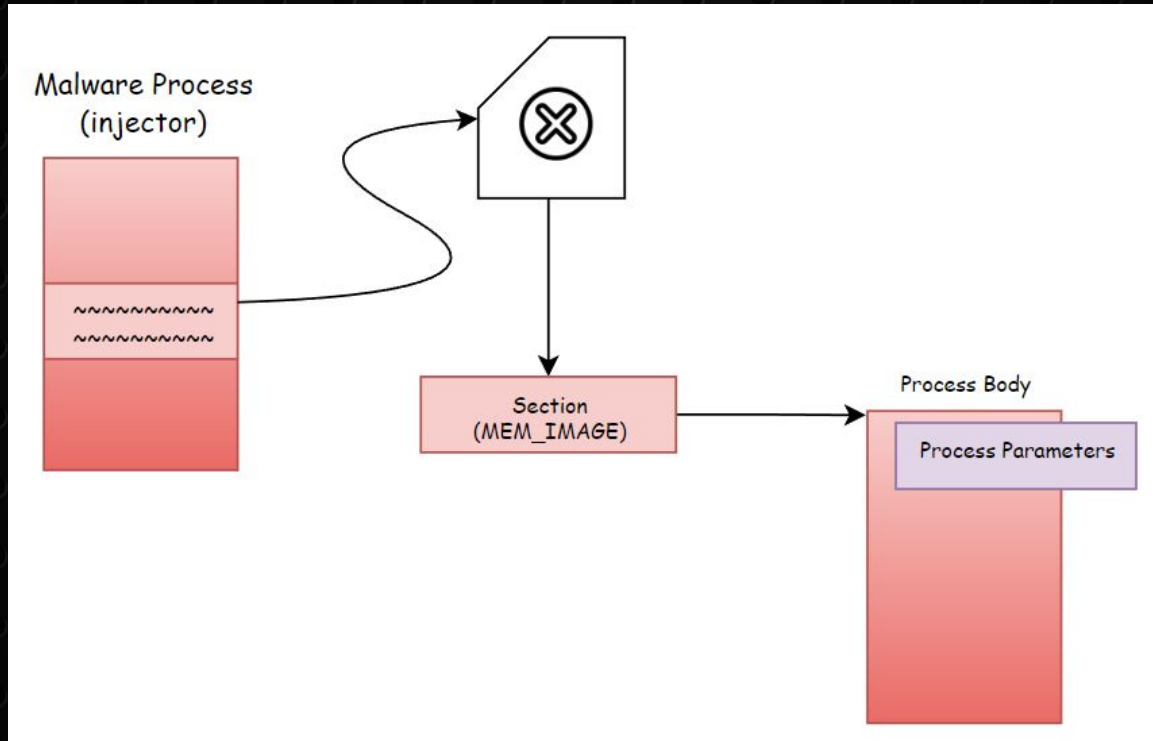
# Process Ghosting



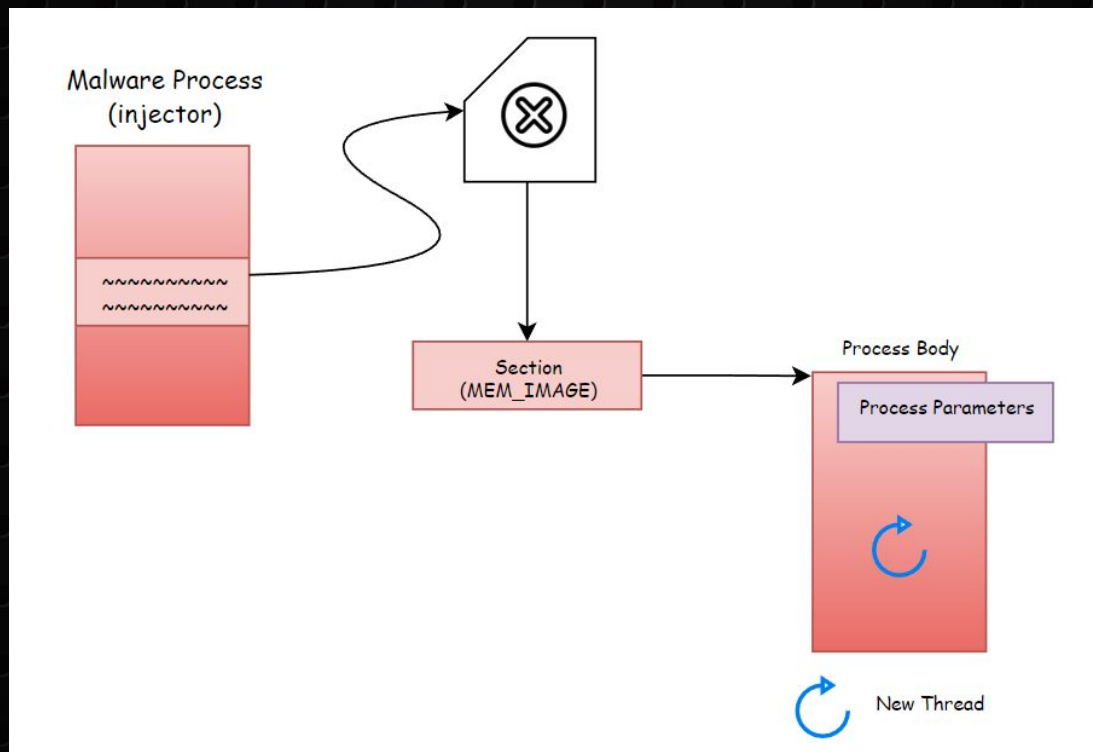
# Process Ghosting



# Process Ghosting



# Process Ghosting



# Process Ghosting – API Calls

- Kernel32.dll:
  - ◆ WriteFile, CloseHandle
- Ntdll.dll:
  - ◆ NtOpenFile, NtSetInformationFile, NtCreateSection, NtCreateProcessEx, NtCreateProcessParametersEx, NtCreateThreadEx



# Why Process Doppelganging was detected?

```

3: kd> dt _FILE_OBJECT fffffde84`d18ee450 // transacted file
ntdll!_FILE_OBJECT
+0x000 Type           : 0n5
+0x002 Size           : 0n216
...
+0x050 Flags          : 0x44042
+0x058 FileName       : _UNICODE_STRING "\temp\mynotes.txt"
...
+0x0c0 IrpList         : _LIST_ENTRY [ 0xffffde84`d18ee510 - 0xffffde84`d18ee510 ]
+0x0d0 FileObjectExtension : 0xffffde84`d2cc8840 Void

3: kd> dt 0xffffde84`d2cc8840 L 4
ffffde84`d2cc8840  00000000`00000000 fffffde84`c8ff07b0
ffffde84`d2cc8850  00000000`00000000 00000000`00000000

3: kd> dt _TXN_PARAMETER_BLOCK fffffde84`c8ff07b0
ntdll!_TXN_PARAMETER_BLOCK
+0x000 Length         : 0x10
+0x002 TxFsContext     : 0xffffe
+0x008 TransactionObject : 0xffffde84`ce346730 Void

0: kd> dt _FILE_OBJECT 0xffffde84d7084370 // normal file
ntdll!_FILE_OBJECT
+0x000 Type           : 0n5
+0x002 Size           : 0n216
...
+0x058 FileName       : _UNICODE_STRING "\Users\STEALT~1\AppData\Local\Temp\PG82BB.tmp"
...
+0x0c0 IrpList         : _LIST_ENTRY [ 0xffffde84`d7084430 - 0xffffde84`d7084430 ]
+0x0d0 FileObjectExtension : (null)

```

# Dirty Vanity

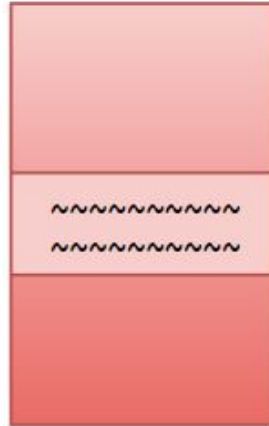
- Similar to Classic Process Injection
- However, instead of creating a new thread in the target process, it clones the target process executing our injected shellcode.
- This defeats the security products which relies on classic injection pattern i.e.
  - ◆ Allocation, remote code injection, and remote thread creation

# Dirty Vanity - Steps

- Obtain Handle to a target process
  - CreateToolHelp32Snapshot, OpenProcess, NtQuerySystemInformation
- Allocate new memory region at target process
  - VirtualAllocEx, NtAllocateVirtualMemory
- Write payload into newly allocated memory
  - WriteProcessMemory, NtWriteVirtualMemory
- Clone Process
  - [RtlCreateProcessReflection](#)

# Dirty Vanity

Malware Process  
(injector)



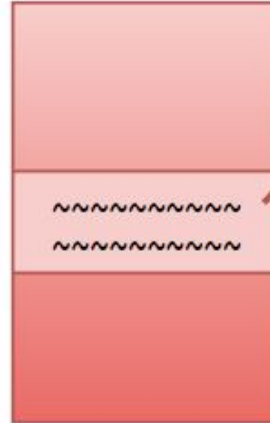
Legitimate  
Process



 thread Executing

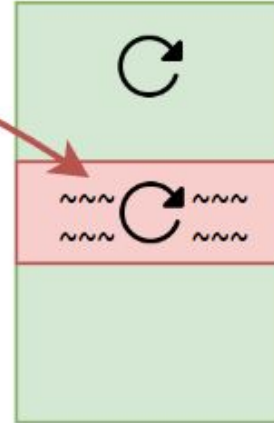
# Dirty Vanity

Malware Process  
(injector)



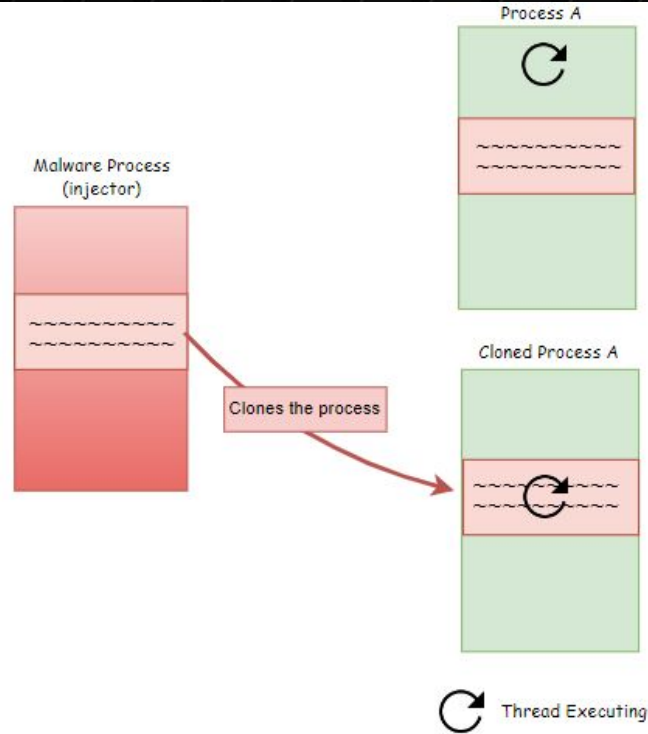
Injects

Legitimate  
Process



thread Executing

# Dirty Vanity





# Dirty Vanity - API calls

→ Kernel32.dll :

- ◆ *CreateToolHelp32Snapshot, Process32First, Process32Next, Thread32First, Thread32Next, OpenProcess, WriteProcessMemory, VirtualProtectEx, OpenThread*

→ Ntdll.dll:

- ◆ *NtQuerySystemInformation, NtAllocateVirtualMemory, NtWriteVirtualMemory, RtlCreateProcessReflection*

# Mockingjay

- This technique abuses the DLL which already has the section with the RWX (Read, Write, Execute) protection
- The main purpose of this technique is to avoid the usage of heavily monitored APIs that does
  - ◆ New memory allocation (NtAllocateVirtualMemory)
  - ◆ Changes the memory protection (NtProtectVirtualMemory)

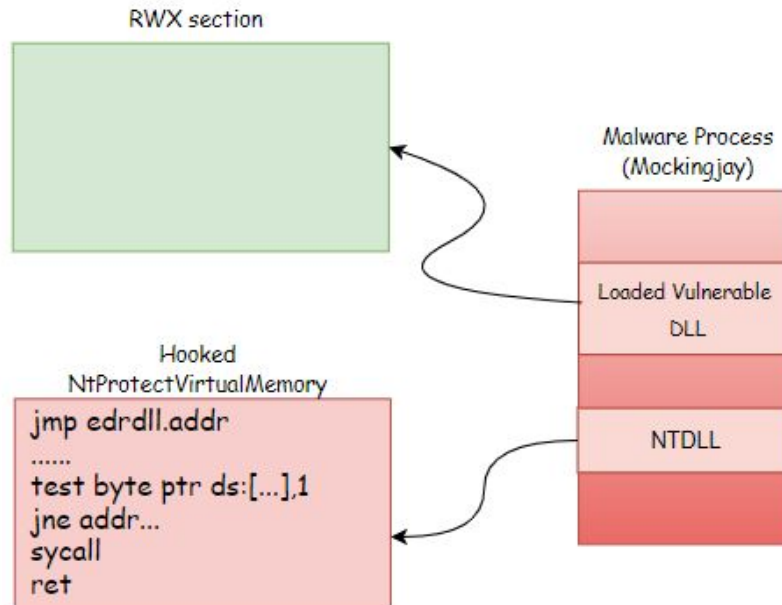
# Mockingjay - Steps

- Finding DLL that contains RWX section
  - <https://gist.github.com/thefLink/5054f3cd35ff05e0ad028459d2ef57b9>
  - <https://github.com/oldboy21/JayFinder>
- Various steps can be performed now
  - Self injection and evasion
  - Remote injection to RWX region
  - Remotely loading “targeted dll”, writing payload in RWX region and creating new thread

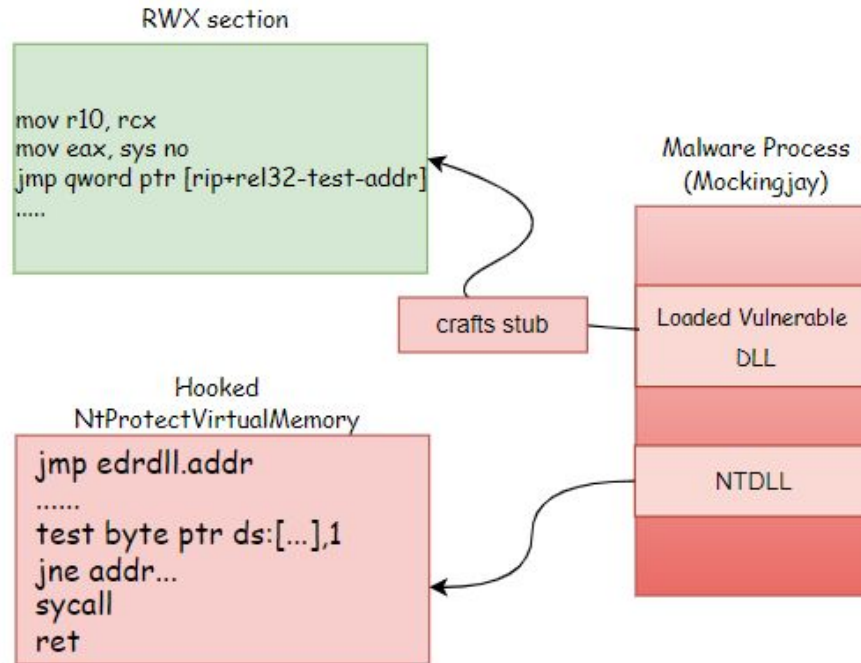
# Mockingjay - Steps

- Self injection and evasion
  - Author of this technique;
    - loads the dll into local process memory
    - Creates a stub where specified syscall number and test stub address at ntdll are resolved
    - Copies the stub to RWX region
    - When the specified syscall needs to be performed the execution is redirected to the crafted stub in DLL
    - Stub moves the syscall number to "rax" register and immediately jumps to the location of the syscall test stub at ntdll skipping the jump hook

# Mockingjay – Self Injection & evasion

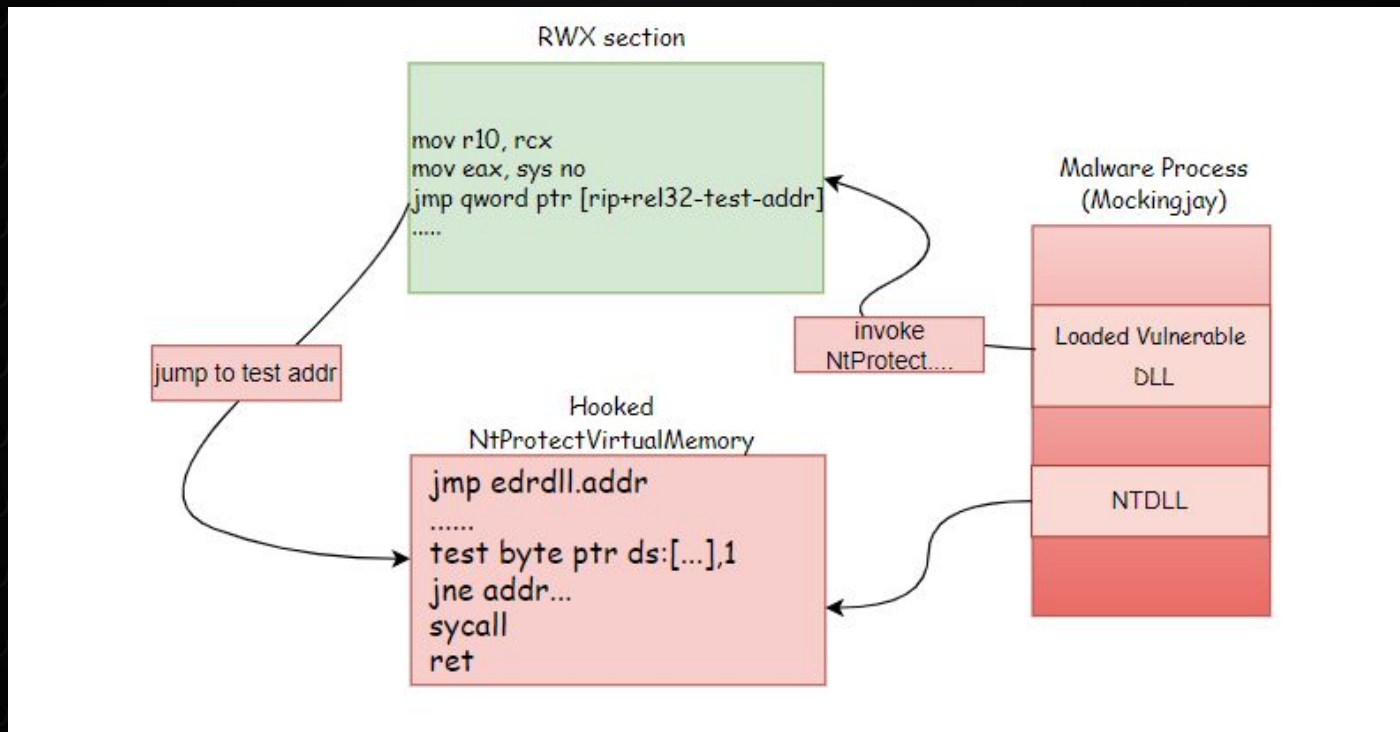


# Mockingjay – Self Injection & evasion





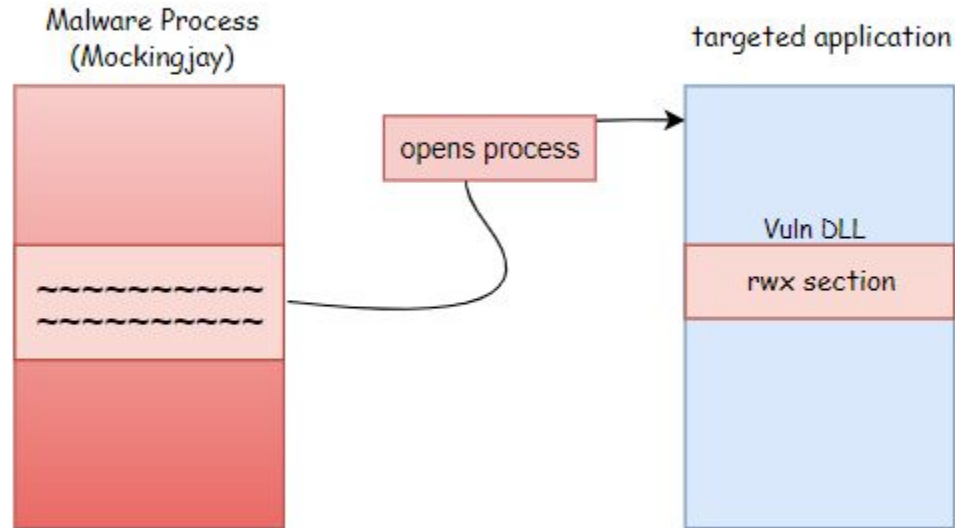
# Mockingjay - Self Injection & evasion



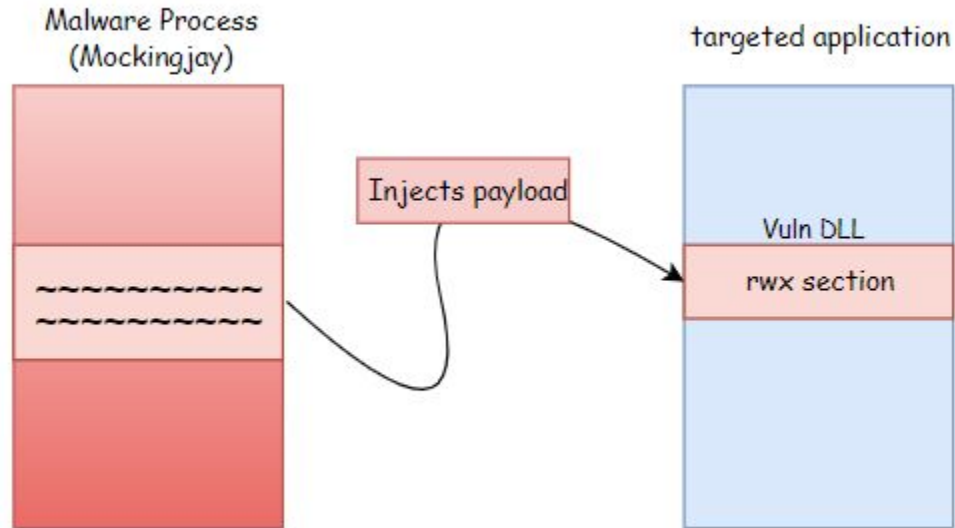
# Mockingjay - Steps

- Remote injection to RWX region
  - Author of this technique;
    - Finds and load the application that automatically loads the target dll
    - Injects the shellcode at RWX region
    - Waits for the application to execute the shellcode

# Mockingjay - Remote Injection



# Mockingjay - Remote Injection



# Mockingjay - Remote Injection

Waits for the application to execute the payload from  
RWX region

# Mockingjay - Our Approach

- Remote injection to RWX region
  - Find and load the application that automatically loads the target dll
  - Inject the shellcode at RWX region
  - Create a remote thread



# Mockjackingly - API calls

- Kernel32.dll :
  - ◆ *OpenProcess, WriteProcessMemory, CreateThread*
- Ntdll.dll:
  - ◆ *NtQuerySystemInformation, NtWriteVirtualMemory, NtCreateThreadEx*

# References

<https://learn.microsoft.com/en-us/windows-hardware/drivers/kernel/section-objects-and-views>  
<https://www.ired.team/>  
<https://blog.f-secure.com/hiding-malicious-code-with-module-stomping/>  
<https://github.com/m0n0ph1/Process-Hollowing>  
<https://www.youtube.com/watch?v=XmWOj-cfixs>  
[https://github.com/hasherezade/transacted\\_hollowing](https://github.com/hasherezade/transacted_hollowing)  
<https://jxy-s.github.io/herpaderping/>  
[https://github.com/hasherezade/process\\_ghosting](https://github.com/hasherezade/process_ghosting)  
<https://www.deepinstinct.com/blog/dirty-vanity-a-new-approach-to-code-injection-edr-bypass>  
<https://www.securityjoes.com/post/process-mockingjay-echoing-rwx-in-userland-to-achieve-code-execution>  
<https://cyberwarfare.live/the-final-curtain-for-process-doppelganging-unmasking-the-defender/>

# Thank You

For Professional Red Team / Blue Team / Purple Team /

Cloud Cyber Range labs / Trainings, please contact

[support@cyberwarfare.live](mailto:support@cyberwarfare.live)

To know more about our offerings, please visit: <https://cyberwarfare.live>