Quack Quack: Analysing Qakbot's Browser Hooking Module – Part 1

July 24, 2021

Direction	Typ	Address	Text
Do...	p	sub_10008D6F+28	call wrap_stringDecryptA_2; GetForegroundWindow
Do...	p	sub_1000A2D8+27	call wrap_stringDecryptA_2; PR_GetError
Do...	p	sub_1000A2D8+F	call wrap_stringDecryptA_2; PR_GetNameForIdentity
Do...	p	sub_1000A2D8+1B	call wrap_stringDecryptA_2; PR_SetError
Do...	p	sub_1000BB8E+E	call wrap_stringDecryptA_2; RapportGP.DLL
Do...	p	sub_1000BB8E+DB	call wrap_stringDecryptA_2; StackWalk64
Do...	p	DllEntryPoint+1D5	call wrap_stringDecryptA_2; WBJ_IGNORE
Do...	p	sub_1000DA06+F4	call wrap_stringDecryptA_2; Windows10 Edge HttpQueryInfo Bug!!!
Do...	p	sub_10002720+BD	call wrap_stringDecryptA_2; chrome.dll
Do...	p	sub_10002CD0+D	call wrap_stringDecryptA_2; chrome.dll
Do...	p	sub_10002720+CA	call wrap_stringDecryptA_2; chrome_child.dll
Do...	p	sub_10002CD0+41	call wrap_stringDecryptA_2; chrome_child.dll
Do...	p	sub_10007F51+D5	call wrap_stringDecryptA_2; comet.yahoo.com;.hiro.tv;safebrowsing.google.com;geo.query.yahoo.com;googleusercontent....
Do...	p	sub_1000D056+61	call wrap_stringDecryptA_2; data_after
Do...	p	sub_1000D056+54	call wrap_stringDecryptA_2; data_before
Do...	p	sub_1000D056+7B	call wrap_stringDecryptA_2; data_end
Do...	p	sub_1000D056+6E	call wrap_stringDecryptA_2; data_inject
Do...	p	sub_1000BB8E+CE	call wrap_stringDecryptA_2; dbghelp.dll
Do...	p	sub_1000D056+95	call wrap_stringDecryptA_2; exclude_url
Do...	p	sub_1000EC48:loc_1000ECC5	call wrap_stringDecryptA_2; f1
Do...	p	sub_100080D8+E5	call wrap_stringDecryptA_2; h2
Do...	p	sub_10008461+21	call wrap_stringDecryptA_2; h3
Do...	p	sub_100080D8+22	call wrap_stringDecryptA_2; https://en.wikipedia.org/static/apple-touch/wikipedia.png
Do...	p	sub_100080D8+61	call wrap_stringDecryptA_2; i4

- 0verfl0w_
- 24th July 2021
- No Comments

Qakbot is one of the most notorious malware families currently operating, and dates back to around 2007. It is primarily focused around stealing banking information and user credentials, however with the huge jump in ransomware popularity among threat actors, Qakbot has been seen to drop Egregor and the ProLock ransomware. As it is primarily operated with an affiliate based business model, a number of threat actors have used it to target different industry sectors, all with varying tactics, techniques, and procedures.

Qakbot is highly modular, with the core payload acting as a loader for additional modules sent by the command and control server. Modules include a hidden VNC plugin, an email collector, a password grabber, and a browser hooking module, which is the main focus of this post.

I have previously covered Qakbot's browser hooking module, with a focus on the fairly simple Internet Explorer hooking functionality. In the next few posts, we'll be analysing how the module hooks Google Chrome API, and what the malicious replacement functions do in order to modify the contents of a web-page, all while supporting both HTTP and HTTP2 traffic. In this first post, we'll be looking at leveraging IDA Python to speed up our analysis of this binary, by developing 3 main scripts; a string decryptor, an API resolver, and a structure resolver for the target API to be hooked. Want to jump straight in? You can find the scripts **here**!

Browser Hooking Module MD5 Hash: 02ca3e9c06b2a9b2df05c97a8efa03e7

Table of Contents

String Decryption

The string decryption function is fairly simple to replicate, all that entails is a basic XOR algorithm. Our goal however, is not just to replicate it. This module has two core string decryption functions, which appear in four function wrappers. The four function wrappers are called a total of 66 times, which would make manual decryption quite tedious.

```
char *__usercall stringDecryptA@<eax>(
        unsigned int stringOffset@<eax>,
        int dataBlob@<ecx>,
        unsigned int dataBlobSize,
        char *keyBlob)
{
  unsigned int counter1; // ecx
  LPVOID allocatedHeap; // eax MAPDST
  unsigned int counter2; // esi
  unsigned int v10; // edi
  _BYTE *v11; // ecx
  char v12; // al
  unsigned int stringSize; // [esp+10h] [ebp-4h]
  unsigned int v1; // [esp+1Ch] [ebp+8h]

  stringSize = 0;
  counter1 = stringOffset;
  if ( stringOffset < dataBlobSize )
  {
    while ( *(_BYTE *)(counter1 + dataBlob) != (unsigned __int8)keyBlob[counter1 % 0x5A] )
    {
      if ( ++counter1 >= dataBlobSize )
        goto LABEL_6;
    }
    stringSize = counter1 - stringOffset;
  }
LABEL_6:
  allocatedHeap = callHeapAlloc(stringSize + 1);
  counter2 = 0;
  if ( !allocatedHeap )
    return (char *)&unk_10029CCC;
  if ( stringSize )
  {
    v1 = stringOffset - (_DWORD)allocatedHeap;
    v10 = stringOffset + dataBlob;
    do
    {
      v11 = (char *)allocatedHeap + counter2;
      v12 = *(_BYTE *)(v10 + counter2) ^ keyBlob[((unsigned int)allocatedHeap + counter2 + v1) % 0x5A];
      ++counter2;
      *v11 = v12;
    }
    while ( counter2 < stringSize );
  }
  return (char *)allocatedHeap;
}
```

Additionally, each of the four wrappers utilise different arguments. The core string decryption function accepts four arguments in total; a string offset, an encrypted data blob, the encrypted data blob size, and a key blob. All but one of the function wrappers accept one argument, which is the string offset – the other wrapper accepts no arguments, and uses a hardcoded offset.

```
.text:10001000 ; char *__usercall sub_10001000@<eax>(unsigned int@<eax>)
.text:10001000 sub_10001000    proc near
.text:10001000
.text:10001000                 push    offset aV
.text:10001005                 push    0BBBh
.text:1000100A                 mov     ecx, offset unk_10026460
.text:1000100F                 call    stringDecryptA
.text:10001014                 pop     ecx
.text:10001015                 pop     ecx
.text:10001016                 retn
.text:10001016 sub_10001000    endp
.text:10001016
.text:10001017
.text:10001017 ; =============== S U B R O U T I N E =======================================
.text:10001017
.text:10001017
.text:10001017 ; LPVOID sub_10001017()
.text:10001017 sub_10001017    proc near               ; CODE XREF: DllEntryPoint+1F0↓p
.text:10001017                 push    esi
.text:10001018                 push    offset aV       ; "v"
.text:1000101D                 push    0BBBh
.text:10001022                 mov     esi, 567h
.text:10001027                 mov     eax, offset unk_10026460
.text:1000102C                 call    stringDecryptB
.text:10001031                 pop     ecx
.text:10001032                 pop     ecx
.text:10001033                 pop     esi
.text:10001034                 retn
.text:10001034 sub_10001017    endp
.text:10001034
.text:10001035
.text:10001035 ; =============== S U B R O U T I N E =======================================
```

Key
Data Blob Length
Data Blob

Hardcoded String Offset

Luckily for us, there are minimal differences across the function wrappers, with the important data pushed to the string decryption function in a similar fashion. In order to get the string blob address, we need to query the address before the call to the string decryption function. We then need to query the address before that, and check for a **push** instruction. If there is not one, we're dealing with the wrapper using a hardcoded offset, so we need to handle it differently. If there is, we can grab the string blob size, and jump back one more address to locate the address of the key blob.

So, we will be writing a script to accept the addresses of the two core string decryption functions, locate the function wrappers, gather the relevant arguments from the wrappers, before finding all cross references to the wrappers, and locating the string offset.

```
.text:100146D8 ; ---------------------------------------------------------------------------
.text:100146D8
.text:100146D8 loc_100146D8:                           ; CODE XREF: sub_1001468F+C↑j
.text:100146D8                 mov     eax, 6DDh
.text:100146DD                 call    wrap_stringDecryptA_1
.text:100146E2                 mov     [ebp+var_4], eax
.text:100146E5                 push    eax             ; lpString
.text:100146E6                 xor     eax, eax
.text:100146E8                 call    strdup
.text:100146ED                 pop     ecx
.text:100146EE                 lea     ecx, [ebp+var_4]
.text:100146F1                 mov     dword_10029C6C, eax
.text:100146F6                 call    sub_10013DE1
.text:100146FB
```

The main IDA API we'll be using for this are:

```
idc.prev_head() # get the previous address
idc.get_operand_value() # get operand value
idc.print_insn_mnem() # print instruction
idc.get_operand_type() # get operand type
idautils.XrefsTo() # get cross references to address
```

Locating the cross references is as simple as returning a list of addresses gathered from the idautils.XrefsTo() function call, as seen below.

```
def locateFunctionCrossReferences(functionAddress):
        return [addr.frm for addr in idautils.XrefsTo(functionAddress)]
```

Then, we need to pass these addresses into a function for retrieving the string blob address, string blob size, and key blob address. Again, this is fairly simple to do. We will get the address before the cross reference, using idc.prev_head(), and use idc.get_operand_value() in order to retrieve the address of the string blob. Then, use idc.prev_head() to get the address before, check if it corresponds to a **push** instruction, and if so grab the string blob size, and the address of the key blob using idc.get_operand_value(). If it doesn't, then we will locate the hardcoded string offset, move the current address pointer back, and then extract the string blob size and key blob address.

```
def retrieveFunctionArguments(functionCrossReference):

    specificStringOffset = 0
    stringBlobSize = 0
    keyBlobAddress = 0
    stringBlobAddress = 0

    currentAddress = functionCrossReference
    functionStart = idc.get_func_attr(functionCrossReference, FUNCATTR_START)

    previousAddress = idc.prev_head(currentAddress)

    stringBlobAddress = idc.get_operand_value(previousAddress, 1)

    previousAddress = idc.prev_head(previousAddress)

    if idc.print_insn_mnem(previousAddress) != "push":

        specificStringOffset = idc.get_operand_value(previousAddress, 1)
        previousAddress = idc.prev_head(previousAddress)

    stringBlobSize = idc.get_operand_value(previousAddress, 0)
    keyBlobAddress = idc.get_operand_value(idc.prev_head(previousAddress), 0)

    return stringBlobAddress, keyBlobAddress, stringBlobSize, specificStringOffset
```

We now have the three arguments, we just need the string offset. The string offset retrieving function will accept an address (cross reference to the function wrappers), and iterate over the addresses before the call, in order to find a **mov** instruction, where the operand type of the second argument is of **idc.o_imm**. As almost all calls to the wrapper functions use a register to hold the string offset, we will have very few errors with this function, and the errors we do have we can "manually" decrypt.

```python
def locateStringOffset(functionAddress):

    stringOffset = 0

    previousAddress = functionAddress

    while True:

        previousAddress = idc.prev_head(previousAddress)

        if previousAddress <= functionAddress - 10:
            break

        if idc.print_insn_mnem(previousAddress) == "mov":
            if idc.get_operand_type(previousAddress, 0) == 1 and idc.get_operand_type(previousAddress, 1) == idc.o_imm:
                stringOffset = idc.get_operand_value(previousAddress, 1)
                return stringOffset
```

At this point, we have now grabbed all of the four values that we need, so now to wrap it together in one function, and implement the string decryption function. I won't be covering the reversing of the string decryption function, as it is already widely documented.

```python
def decryptString(stringOffset, stringBlob, stringBlobSize, keyBlob):

    loopCounter = 0
    offsetStringEnd = stringOffset
    decryptedString = ""

    if stringOffset < stringBlobSize:

        while stringBlob[offsetStringEnd] != keyBlob[offsetStringEnd % 0x5A]:
            offsetStringEnd += 1
        stringBlobSize = offsetStringEnd - stringOffset

        while loopCounter <= stringBlobSize:

            decryptedByte = ord(stringBlob[(stringOffset + loopCounter)]) ^ ord(keyBlob[(stringOffset + loopCounter) % 0x5A])
            decryptedString += chr(decryptedByte)

            loopCounter += 1

        return decryptedString
def locateStringFunctions(listOfCoreFunctions):

    for coreFunction in listOfCoreFunctions:

        functionCrossReferences = locateFunctionCrossReferences(coreFunction)

        for functionReference in functionCrossReferences:

            stringBlobAddress, keyBlobAddress, stringBlobSize, specificStringOffset = retrieveFunctionArguments(functionReference)

            stringBlobData = readBytesFromFile(stringBlobAddress, stringBlobSize)
            keyBlobData = readBytesFromFile(keyBlobAddress, 0x5A)

            if specificStringOffset != 0:
                # decrypt the string off the bat
                pass

            functionStart = idc.get_func_attr(functionReference, FUNCATTR_START)

            nextFunctionCrossReferences = locateFunctionCrossReferences(functionStart)

            for nextFunctionReference in nextFunctionCrossReferences:

                stringOffset = locateStringOffset(nextFunctionReference)

                if stringOffset == -1:
                    continue

                decryptedString = decryptString(stringOffset, stringBlobData, stringBlobSize, keyBlobData)

                addStringComment(decryptedString, nextFunctionReference)
```

With the relevant functions all wrapped into one, we need to add a final function that will add comments to the IDB in the relevant locations. **OALabs** have a brilliant snippet **here** that we will be using, passing in the cross reference to the function wrapper calls in order to add comments at that specific address.

Next, we need to add one more function responsible for reading bytes from the IDB. We will pass the addresses of the string blob and key blob, along with the respective sizes, and have it return the read bytes back to our main function. This is simple to do, and we can use the idaapi.get_bytes() function to do just that.

```
def readBytesFromFile(dataOffset, bytesToRead):
        return idaapi.get_bytes(dataOffset, bytesToRead)
```

From here, all that needs to be done is to add a “main” function that will accept a non-hardcoded amount of offsets, in case we have more than 2 string decryption functions. That’s simple to do as well, all we need is to use an asterisk!

```
def stringAutomation(*coreFunctionList):
        locateStringFunctions(coreFunctionList)
```

And we’re finished! All that we need to do now is import it into IDA, pass the offsets of the core string decryption functions to the “main” function, and hit enter!

If all goes well, you should immediately notice strings have been added as comments next to most of the calls to the function wrappers.

Direction	Typ	Address	Text
Do...	p	sub_10008D6F+28	call wrap_stringDecryptA_2; GetForegroundWindow
Do...	p	sub_1000A2D8+27	call wrap_stringDecryptA_2; PR_GetError
Do...	p	sub_1000A2D8+F	call wrap_stringDecryptA_2; PR_GetNameForIdentity
Do...	p	sub_1000A2D8+1B	call wrap_stringDecryptA_2; PR_SetError
Do...	p	sub_1000BB8E+E	call wrap_stringDecryptA_2; RapportGP.DLL
Do...	p	sub_1000BB8E+DB	call wrap_stringDecryptA_2; StackWalk64
Do...	p	DllEntryPoint+1D5	call wrap_stringDecryptA_2; WBJ_IGNORE
Do...	p	sub_1000DA06+F4	call wrap_stringDecryptA_2; Windows10 Edge HttpQueryInfo Bug!!!
Do...	p	sub_10002720+BD	call wrap_stringDecryptA_2; chrome.dll
Do...	p	sub_10002CD0+D	call wrap_stringDecryptA_2; chrome.dll
Do...	p	sub_10002720+CA	call wrap_stringDecryptA_2; chrome_child.dll
Do...	p	sub_10002CD0+41	call wrap_stringDecryptA_2; chrome_child.dll
Do...	p	sub_10007F51+D5	call wrap_stringDecryptA_2; comet.yahoo.com;.hiro.tv;safebrowsing.google.com;geo.query.yahoo.com;googleusercontent....
Do...	p	sub_1000D056+61	call wrap_stringDecryptA_2; data_after
Do...	p	sub_1000D056+54	call wrap_stringDecryptA_2; data_before
Do...	p	sub_1000D056+7B	call wrap_stringDecryptA_2; data_end
Do...	p	sub_1000D056+6E	call wrap_stringDecryptA_2; data_inject
Do...	p	sub_1000BB8E+CE	call wrap_stringDecryptA_2; dbghelp.dll
Do...	p	sub_1000D056+95	call wrap_stringDecryptA_2; exclude_url
Do...	p	sub_1000EC48:loc_1000ECC5	call wrap_stringDecryptA_2; f1
Do...	p	sub_100080D8+E5	call wrap_stringDecryptA_2; h2
Do...	p	sub_10008461+21	call wrap_stringDecryptA_2; h3
Do...	p	sub_100080D8+22	call wrap_stringDecryptA_2; https://en.wikipedia.org/static/apple-touch/wikipedia.png
Do...	p	sub_100080D8+61	call wrap_stringDecryptA_2; i4

In order to create a “manual” decrypt function, we will set up a new function that accepts 3 arguments: the offset of the function wrapper in question, the address where the function wrapper is called, and the string offset. We then just pass this into the relevant functions to grab the correct arguments, decrypt the string itself, and then add the comments!

```python
def manualStringDecrypt(stringDecryptWrapper, referenceAddress, targetOffset):

    stringBlobAddress, keyBlobAddress, stringBlobSize,_ = retrieveFunctionArguments(stringDecryptWrapper)

    stringBlobData = readBytesFromFile(stringBlobAddress, stringBlobSize)
    keyBlobData = readBytesFromFile(keyBlobAddress, 0x5A)

    decryptedString = decryptString(targetOffset, stringBlobData, stringBlobSize, keyBlobData)

    addStringComment(decryptedString, referenceAddress)
```

With the automation of the string decryptor complete, it's time to move onto resolving the API calls!

Resolving Hashed API

The browser hooking module uses an interesting method of storing resolved APIs. Rather than resolve all APIs when necessary like Dridex, or resolving at startup and assign each API to a variable, Qakbot uses structures in memory to hold API loaded from different libraries, meaning there will be a kernel32 structure, a wininet structure, and so on. This can cause some issues as it is not as simple as renaming variables, or adding comments next to each call. Instead we will have to recreate these structures, and change the type of the variable responsible for pointing to the structures.

```c
DWORD **__cdecl wResolveAPI(char *hashedAPIList, int hashedAPIListSize, int dllNameOffset)
{
  DWORD **HashedAPI; // esi
  HMODULE ModuleHandleA; // eax
  char *v6; // [esp+4h] [ebp-4h] BYREF

  HashedAPI = 0;
  v6 = wrap_stringDecryptA_1(dllNameOffset);
  if ( dllNameOffset == 2552 )
    ModuleHandleA = GetModuleHandleA(v6);
  else
    ModuleHandleA = (HMODULE)((int (__stdcall *)(char *))*dword_1002714C)(v6);
  if ( ModuleHandleA )
    HashedAPI = (DWORD **)locateAndLoadHashedAPI(hashedAPIListSize, (int)hashedAPIList, (int)ModuleHandleA);
  sub_10013DE1((void **)&v6);
  return HashedAPI;
}
```

The API structures are resolved on startup, and use 3 pieces of information; a pointer to the list of hashed APIs, the size of the list in bytes, and a string offset corresponding to the target DLL. This string offset is passed into a string decryption function, which luckily we have already implemented, so we are already 35% done.

```
.text:1000B84E                 push    9F8h            ; dllNameOffset
.text:1000B853                 push    104h            ; hashedAPIListSize
.text:1000B858                 push    offset byte_1001F9A0 ; hashedAPIList
.text:1000B85D                 call    wResolveAPI
.text:1000B862                 add     esp, 0Ch
.text:1000B865                 push    ebx             ; dllNameOffset
.text:1000B866                 push    28h ; '('       ; hashedAPIListSize
.text:1000B868                 push    offset byte_1001FAA8 ; hashedAPIList
.text:1000B86D                 mov     dword_1002714C, eax
.text:1000B872                 call    wResolveAPI
.text:1000B877                 add     esp, 0Ch
.text:1000B87A                 push    655h            ; dllNameOffset
.text:1000B87F                 push    38h ; '8'       ; hashedAPIListSize
.text:1000B881                 push    offset byte_1001FAD4 ; hashedAPIList
.text:1000B886                 mov     dword_10027130, eax
.text:1000B88B                 call    wResolveAPI
.text:1000B890                 add     esp, 0Ch
.text:1000B893                 push    0AD4h           ; dllNameOffset
.text:1000B898                 push    18h             ; hashedAPIListSize
.text:1000B89A                 push    offset byte_1001FB10 ; hashedAPIList
.text:1000B89F                 mov     dword_1002712C, eax
.text:1000B8A4                 call    wResolveAPI
```

The arguments we need are pushed to the stack immediately before calling the API resolving function, so extracting them will be fairly simple. All we need to search for are 2 integers and an offset in memory that are pushed to the stack. It will be structured very similarly to our string decryption function, except we will use idc.is_off0() and idc.get_full_flags() to locate the offset.

```python
while True:

    previousAddress = idc.prev_head(currentAddress)

    if previousAddress <= functionStart:
        break

    if idc.print_insn_mnem(previousAddress) == "push":
        if idc.get_operand_type(previousAddress, 0) == idc.o_imm and idc.is_off0(idc.get_full_flags(previousAddress)):
            hashDataAddress = idc.get_operand_value(previousAddress, 0)

            previousAddress = idc.prev_head(previousAddress)

            if idc.print_insn_mnem(previousAddress) == "push":
                if idc.get_operand_type(previousAddress, 0) == idc.o_imm and not idc.is_off0(idc.get_full_flags(previousAddress)):
                    hashListSize = idc.get_operand_value(previousAddress, 0)

                    previousAddress = idc.prev_head(previousAddress)

                    if idc.print_insn_mnem(previousAddress) == "push":
                        if idc.get_operand_type(previousAddress, 0) == idc.o_imm and not idc.is_off0(idc.get_full_flags(previousAddress)):
                            dllStringOffset = idc.get_operand_value(previousAddress, 0)

                        else:
                            dllStringOffset = 0

                        break

    currentAddress = previousAddress
```

Now we have the 3 arguments, there is one more offset we need to locate: the offset in memory that will point to the API structure. The return value will be stored in **EAX**, and in every instance it is moved into a variable. This variable is then referenced whenever an API is called, so we will set up the automation to change the type of the variable to a pointer to the API structure, saving us some time.

```
else
{
  v2 = (void *)((int (__stdcall *)(_DWORD, _DWORD, _DWORD, char *))dword_1002714C[47])(0, 0, 0, v7);
}
hObject = v2;
LastError = GetLastError();
if ( !hObject )
  return 0;
if ( LastError == 183 )
{
  if ( !((int (__stdcall *)(HANDLE))dword_1002714C[46])(hObject) )
  {
```

In order to do this, we will loop through every address **after** the call to the API resolving function, and check for a **mov** instruction that has **EAX** as the second operand, and an offset in memory as the first operand. Once we have located a valid instruction, we can return all 4 of the discovered values.

```
newAddress = functionCrossReference

while True:

    nextAddress = idc.next_head(newAddress)

    if nextAddress >= functionEnd:
        break

    if idc.print_insn_mnem(nextAddress) == "mov" and idc.print_operand(nextAddress, 1) == "eax" and idc.is_off0(idc.get_full_flags(nextAddress)):

        dwordPointer = idc.get_operand_value(nextAddress, 0)

        break

    newAddress = nextAddress
```

With the address of the hashed APIs list, and the size, we need to read the list from the IDB, and split it up into chunks of 4 bytes, before converting each chunk to a 32 bit integer using the **struct** module. Each chunk will be XORed with a 32 bit integer, as can be seen in the code below. In this case, that integer is **0x218FE95B**.

```
while ( 1 )
{
  apiName = (const CHAR *)(v2 + *(_DWORD *)(v6 + 4 * exportCounter));
  calculatedHashOfAPI = lstrlenA(apiName);
  if ( calculatedHashOfAPI )
    calculatedHashOfAPI = calculateCRC32(0, (int)apiName, calculatedHashOfAPI);
  if ( (calculatedHashOfAPI ^ 0x218FE95B) == targetHash )
    break;
  if ( (unsigned int)++exportCounter >= *((_DWORD *)v4 + 6) )
    return 0;
  v2 = a1;
}
```

Then, we just need to figure out what DLL is being targeted, which we can find by passing the string offset to a string decryption function. With the list of hashes in hand, and the target DLL, we can now start “brute forcing” the APIs.

```
resolvedAPIList = []

stringBlobAddress, keyBlobAddress, stringBlobSize = retrieveStringFunctionArguments(internalStringFunction)

stringBlobData = readBytesFromFile(stringBlobAddress, stringBlobSize)
keyBlobData = readBytesFromFile(keyBlobAddress, 0x5A)

listOfHashes, hashListSize, dllStringOffset, dwordPointer = retrieveAPIFunctionArguments(functionReference)

listOfConvertedHashes = extractListOfHashes(xorValue, listOfHashes, hashListSize)

dllName = decryptString(dllStringOffset, stringBlobData, stringBlobSize, keyBlobData).strip("\x00")

for convertedHash in listOfConvertedHashes:
    resolvedAPIString = bruteForceCRC32Hash(dllName, convertedHash)
    resolvedAPIList.append(resolvedAPIString)
```

Essentially what we will do is open the target DLL using the **pefile** module, parse the exports from the export directory, and proceed to CRC32 hash each export using the **zlib** module, to locate a match. Once a match has been discovered, we will return a string similar to the one below:

```
kernel32::CreateProcessW
```

We will then pass this string into a local type we create in IDA, before assigning it to the correct variable. Before doing so, we need to create a local type first. Both of these processes can be done with the following functions:

```
// create struct and add struct members
idc.add_struc()
idc.add_struc_member()
// assign to variable
idc.set_name()
idc.SetType()
```

Putting these calls into a few functions, we get the code that you can see below.

```
def generateAPIStructure(dllName, resolvedAPIList):

    structureName = dllName + "_array"
    structID = idc.add_struc(-1, structureName, 0)

    for resolvedAPI in resolvedAPIList:
        idc.add_struc_member(structID, resolvedAPI, -1, FF_DWORD, -1, 4)

    return structureName
structureName = generateAPIStructure(dllName.replace(".", "_"), resolvedAPIList)

idc.set_name(dwordPointer, structureName + "_ptr")
idc.SetType(dwordPointer, structureName + "*")
```

And that’s pretty much all the important functions we need to write! All we need to do is set up a main function that accepts the **xorValue** we found in the API resolver, the address of the string decryption function used inside the API resolver, and the address of the API resolving routine. We then pass this into a function that will find all cross references to the API resolving function, retrieve the required arguments for the string decryption function, and then locate the arguments needed to resolve the API. This is then passed into the respective functions, and once the correct API has been found, we add it to a local type structure, and assign the filled structure to the correct variable.

```
def locateAPIFunctions(xorValue, internalStringFunction, listOfCoreFunctions):

    for coreFunction in listOfCoreFunctions:

        functionCrossReferences = locateFunctionCrossReferences(coreFunction)

        for functionReference in functionCrossReferences:

            resolvedAPIList = []

            stringBlobAddress, keyBlobAddress, stringBlobSize = retrieveStringFunctionArguments(internalStringFunction)

            stringBlobData = readBytesFromFile(stringBlobAddress, stringBlobSize)
            keyBlobData = readBytesFromFile(keyBlobAddress, 0x5A)

            listOfHashes, hashListSize, dllStringOffset, dwordPointer = retrieveAPIFunctionArguments(functionReference)

            listOfConvertedHashes = extractListOfHashes(xorValue, listOfHashes, hashListSize)

            dllName = decryptString(dllStringOffset, stringBlobData, stringBlobSize, keyBlobData).strip("\x00")

            for convertedHash in listOfConvertedHashes:
                resolvedAPIString = bruteForceCRC32Hash(dllName, convertedHash)
                resolvedAPIList.append(resolvedAPIString)


            structureName = generateAPIStructure(dllName.replace(".", "_"), resolvedAPIList)

            idc.set_name(dwordPointer, structureName + "_ptr")
            idc.SetType(dwordPointer, structureName + "*")

def apiAutomation(xorValue, internalStringFunction, *coreFunctionList):

    locateAPIFunctions(xorValue, internalStringFunction, coreFunctionList)
```

If all goes smoothly, you should have something similar to below!

```
dword_10029C64 = HeapCreate(0, 0x80000u, 0);
FF_AsciiTableLower();
kernel32_dll_array_ptr = (kernel32_dll_array *)wResolveAPI(&byte_1001F9A0, 260, 2552);
ntdll_dll_array_ptr = (ntdll_dll_array *)wResolveAPI(&byte_1001FAA8, 40, 0);
user32_dll_array_ptr = (user32_dll_array *)wResolveAPI(&byte_1001FAD4, 56, 1621);
netapi32_dll_array_ptr = (netapi32_dll_array *)wResolveAPI(&byte_1001FB10, 24, 2772);
advapi32_dll_array_ptr = (advapi32_dll_array *)wResolveAPI(&byte_1001FB30, 100, 2295);
shlwapi_dll_array_ptr = (shlwapi_dll_array *)wResolveAPI(&byte_1001FB98, 44, 739);
shell32_dll_array_ptr = (shell32_dll_array *)wResolveAPI(byte_1001FBC8, 8, 2308);
ws2_32_dll_array_ptr = (ws2_32_dll_array *)wResolveAPI(&byte_1001FBD4, 8, 1596);
v4 = sub_10015A7A((int)hinstDLL);
```

```
Name
  _RTL_CRITICAL_SECTION
  _MEMORY_BASIC_INFORMA
  CONTEXT
  FLOATING_SAVE_AREA
  THREADENTRY32
  _FILETIME
  _SYSTEM_INFO
  _SYSTEM_INFO::$A707B7
  _SYSTEM_INFO::$A707B7
  kernel32_dll_array
  ntdll_dll_array
  user32_dll_array
  netapi32_dll_array
  advapi32_dll_array
  shlwapi_dll_array
  shell32_dll_array
  ws2_32_dll_array
  urlmon_dll_array
```

```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 kernel32_dll_array struc ; (sizeof=0x104, mappedto_30)
00000000 kernel32_dll::LoadLibraryA dd ?
00000004 kernel32_dll::GetProcAddress dd ?
00000008 kernel32_dll::GetModuleHandleA dd ?
0000000C kernel32_dll::CreateToolhelp32Snapshot dd ?
00000010 kernel32_dll::Module32First dd ?
00000014 kernel32_dll::Module32Next dd ?
00000018 kernel32_dll::WriteProcessMemory dd ?
0000001C kernel32_dll::OpenProcess dd ?
00000020 kernel32_dll::VirtualFreeEx dd ?
00000024 kernel32_dll::WaitForSingleObject dd ?
00000028 kernel32_dll::CloseHandle dd ?
0000002C kernel32_dll::LocalFree dd ?
00000030 kernel32_dll::CreateProcessW dd ?
00000034 kernel32_dll::ReadProcessMemory dd ?
00000038 kernel32_dll::Process32First dd ?
0000003C kernel32_dll::Process32Next dd ?
00000040 kernel32_dll::Process32FirstW dd ?
00000044 kernel32_dll::Process32NextW dd ?
00000048 kernel32_dll::CreateProcessAsUserW dd ?
0000004C kernel32_dll::VirtualAllocEx dd ?
00000050 kernel32_dll::VirtualAlloc dd ?
```

Internal Hooking Structures

As this post is already quite long, I won't be going into much depth on the hooking structures, so we will only focus on the basics of it and how the IDA Python script works.

With the string decryption and API resolving functions automated, we can now move our focus to the structures used by the module with regards to hooking. These structures contain four pieces of information; two string offsets linked to the target DLL and the target API, a pointer to the function that will be executed whenever the target API is called (replacement function), and a variable that will point to the trampoline setup during hooking.

```
.data:100220B8 winSockStruct   dd 915h              ; DATA XREF: replaceLdrLoadDll+11D↑o
.data:100220B8                                      ; sub_10002A39+3↑o ...
.data:100220BC                 dd 994h
.data:100220C0                 dd offset replaceLdrLoadDll
.data:100220C4                 dd offset originalLdrLoadDll
.data:100220C8                 db    0
.data:100220C9                 db    0
.data:100220CA                 db    0
.data:100220CB                 db    0
.data:100220CC                 db    0
.data:100220CD                 dd 9E6h
.data:100220D1                 dd 0A97h
.data:100220D5                 dd offset replaceWSAConnect
.data:100220D9                 dd offset originalWSAConnect
.data:100220DD                 db    0
.data:100220DE                 db    0
.data:100220DF                 db    0
.data:100220E0                 db    0
.data:100220E1                 db    0
.data:100220E2                 dd 9E6h
.data:100220E6                 dd 77Ah
.data:100220EA                 dd offset replaceConnect
.data:100220EE                 dd offset originalConnect
.data:100220F2                 db    0
.data:100220F3                 db    0
```

Outside of the structure, we need to locate two more values; the address of the list of structures (obviously), and the number of structures in the list. In most calls to the function responsible for parsing these structures and setting up the hooks, the value corresponding to the amount of structures in the list is pushed onto the stack, before being popped off into **EAX**. The list address is then moved into **ECX**, so extracting this information will be quite simple to do.

```
.text:10002A39 ; =============== S U B R O U T I N E =======================================
.text:10002A39
.text:10002A39
.text:10002A39 ; int loadAndHookWinSockAPI(void)
.text:10002A39 loadAndHookWinSockAPI proc near         ; CODE XREF: DllEntryPoint+24C↓p
.text:10002A39                 push    0Ah
.text:10002A3B                 pop     eax
.text:10002A3C                 mov     ecx, offset winSockStruct
.text:10002A41                 jmp     importAndHookTargetAPI
.text:10002A41 loadAndHookWinSockAPI endp
.text:10002A41
.text:10002A46
.text:10002A46 ; =============== S U B R O U T I N E =======================================
.text:10002A46
.text:10002A46
.text:10002A46 ; int loadAndHookWinInetAPI(void)
.text:10002A46 loadAndHookWinInetAPI proc near         ; CODE XREF: DllEntryPoint+251↓p
.text:10002A46                 push    0Ah
.text:10002A48                 pop     eax
.text:10002A49                 mov     ecx, offset winInetStruct
.text:10002A4E                 jmp     importAndHookTargetAPI
.text:10002A4E loadAndHookWinInetAPI endp
.text:10002A4E
.text:10002A53
.text:10002A53 ; =============== S U B R O U T I N E =======================================
```

As we already have the string decryption function, all we need to do is extract the list of structures, parse it into individual structures, decrypt the relevant strings, and then rename the respective pointers. For example, if the hook structure is targeting PR_Read, we would rename the replacement function to **replacePR_Read**, and the trampoline pointer to **originalPR_Read**, making it easier to pick out in a function.

```python
def retrieveInjectStructFunctionArguments(functionCrossReference):

    pointerToStructure = 0
    structureItemCount = 0

    newAddress = functionCrossReference
    functionStart = idc.get_func_attr(functionCrossReference, FUNCATTR_START)

    while True:

        previousAddress = idc.prev_head(newAddress)

        if previousAddress < functionStart:
            break

        if idc.print_insn_mnem(previousAddress) == "mov" and idc.get_operand_type(previousAddress, 1) == idc.o_imm:
            pointerToStructure = idc.get_operand_value(previousAddress, 1)

            tempAddress = idc.prev_head(previousAddress)

            if idc.print_insn_mnem(tempAddress) == "inc":
                structureItemCount = 1
                break

        if idc.print_insn_mnem(previousAddress) == "push" and idc.get_operand_type(previousAddress, 0) == idc.o_imm:
                structureItemCount = idc.get_operand_value(previousAddress, 0)
                break

        newAddress = previousAddress

    return pointerToStructure, structureItemCount
```

With the relevant functions setup, we just need to wrap it together, import into IDA, pass the address of the hooking function, and run it!

```python
functionCrossReferences = locateFunctionCrossReferences(coreFunction)

stringBlobAddress, keyBlobAddress, stringBlobSize = retrieveStringFunctionArguments(coreFunction)
stringBlobData = readBytesFromFile(stringBlobAddress, stringBlobSize)
keyBlobData = readBytesFromFile(keyBlobAddress, 0x5A)

for functionReference in functionCrossReferences:

    pointerToStructure, structureItemCount = retrieveInjectStructFunctionArguments(functionReference)
    structureData = readBytesFromFile(pointerToStructure, structureItemCount * 21)

    splitStructures = [structureData[i:i + 21] for i in range(0, 21 * structureItemCount, 21)]

    for hookStructure in splitStructures:
        print Len(hookStructure)
        dllNameOffset = struct.unpack("I", hookStructure[0:4])[0]
        apiNameOffset = struct.unpack("I", hookStructure[4:8])[0]
        replaceOffset = struct.unpack("I", hookStructure[8:12])[0]
        originaOffset = struct.unpack("I", hookStructure[12:16])[0]

        decryptedDll = decryptString(dllNameOffset, stringBlobData, stringBlobSize, keyBlobData).strip("\x00")
        decryptedAPI = decryptString(apiNameOffset, stringBlobData, stringBlobSize, keyBlobData)

        print decryptedDll + "::" + decryptedAPI


        addStringComment(decryptedDll, pointerToStructure + (i * 21))
        addStringComment(decryptedAPI, pointerToStructure + (i * 21) + 4)

        idc.set_name(replaceOffset, "replace" + decryptedAPI)
        idc.set_name(originaOffset, "original" + decryptedAPI)
```

Unfortunately this does not work with the Google Chrome API, as we will discover in the next post, but this is basically due to Chrome not exposing it’s internal API in the export table of the core DLLs. As a result, the browser hooking module will have to do some pretty inventive parsing of the internal Chrome libraries in order to locate the target APIs, but that is something we will explore in the next part!

```
int parseAndHookChromeLinkedDLLs()
{
  int v0; // edi
  char *v1; // eax
  HMODULE ModuleHandleA; // eax
  HMODULE v3; // eax
  char *v5; // [esp+8h] [ebp-4h] BYREF

  v0 = 0;
  v1 = wrap_stringDecryptA_2(0x755u);           // chrome.dll
  v5 = v1;
  if ( !v1 )
    goto LABEL_5;
  ModuleHandleA = GetModuleHandleA(v1);
  if ( ModuleHandleA )
    v0 = parseBinaryAndPlaceHooks(ModuleHandleA);
  w_HeapFreeStructures((void **)&v5);
  if ( v0 < 1 )
  {
LABEL_5:
    v5 = wrap_stringDecryptA_2(0x970u);         // chrome_child.dll
    v3 = GetModuleHandleA(v5);
    if ( v3 )
      v0 = parseBinaryAndPlaceHooks(v3);
    w_HeapFreeStructures((void **)&v5);
  }
  return v0;
}
```

For now though, that brings an end to this fairly long post. We've developed the 3 main scripts that will allow us to analyse the module a lot easier, and in the next post we will start exploring the parsing of the Chrome DLLs, and most likely analyse the replacement function for the Chrome equivalent of HTTPSendRequest and PR_Write. That post is currently under works, so you should expect it to come out very soon!

Any questions? Feel free to drop a comment with your question, or you can drop me a DM via Twitter (@0verfl0w_)!