fireeye.com

Remote Symbol Resolution

by David Zimmer

8-10 minutes

Introduction

The following blog discusses a couple of common techniques that malware uses to obscure its access to the Windows API. In both forms examined, analysts must calculate the API start address and resolve the symbol from the runtime process in order to determine functionality.

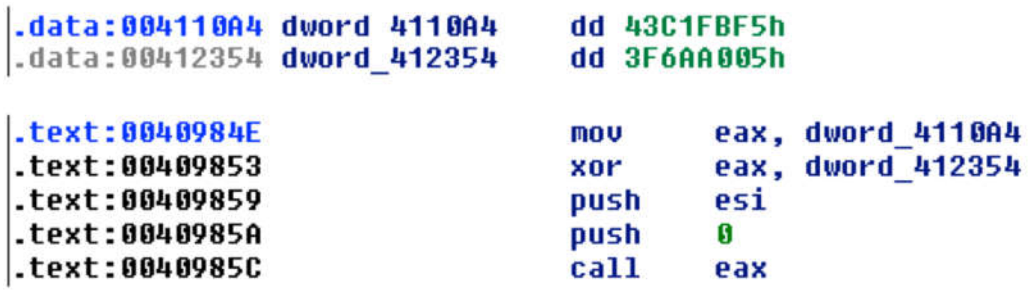
After introducing the techniques, we present an open source tool we developed that can be used to resolve addresses from a process running in a virtual machine by an IDA script. This gives us an efficient way to quickly add readability back into the disassembly.

Techniques

When performing an analysis, it is very common to see malware try to obscure the API it uses. As a malware analyst, determining which API is used is one of the first things we must resolve in order to determine the capabilities of the code.

Two common obfuscations we are going to look at in this blog are encoded function pointer tables and detours style hook stubs. In both of these scenarios the entry point to the API is not directly visible in the binary.

For an example of what we are talking about, consider the code in Figure 1, which was taken from a memory dump of xdata crypto ransomware sample C6A2FB56239614924E2AB3341B1FBBA5.

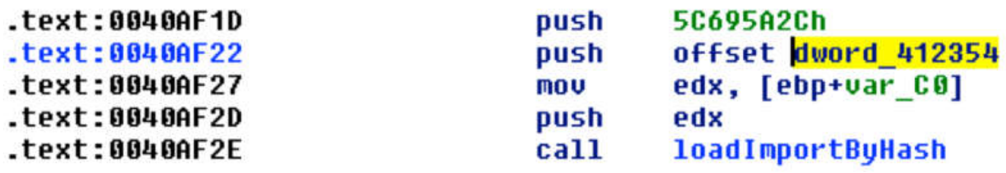


```
.data:004110A4 dword_4110A4     dd 43C1FBF5h
.data:00412354 dword_412354     dd 3F6AA005h

.text:0040984E                  mov     eax, dword_4110A4
.text:00409853                  xor     eax, dword_412354
.text:00409859                  push    esi
.text:0040985A                  push    0
.text:0040985C                  call    eax
```

Figure 1: API obfuscation code from a crypto ransomware sample

In Figure 1, we see one numeric value being loaded into eax, XORed against another, and then being called as a function pointer. These numbers only make sense in the context of a running process. We can calculate the final number from the values contained in the memory dump, but we also need a way to know which API address it resolved to in this particular running process. We also have to take into account that DLLs can be rebased due to conflicts in preferred base address, and systems with ASLR enabled.

Figure 2 shows one other place we can look to see where the values were initially set.

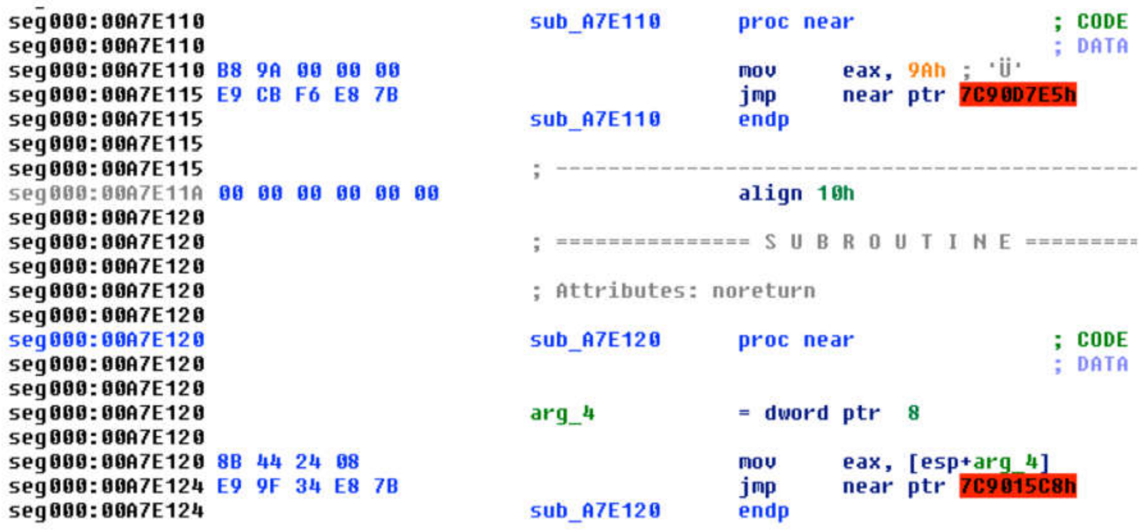


```
.text:0040AF1D                  push    5C695A2Ch
.text:0040AF22                  push    offset dword_412354
.text:0040AF27                  mov     edx, [ebp+var_C0]
.text:0040AF2D                  push    edx
.text:0040AF2E                  call    loadImportByHash
```

Figure 2: Crypto malware setting obfuscated function pointer from API hash

In this case, the initial value is loaded from an API hash lookup – again not of immediate value. Here we have hit a crossroad, with multiple paths we can take to resolve the problem. We can search for a published hash list, extract the hasher and build our own database, or figure out a way to dynamically resolve the decoded API address.

Before we choose which path to take, let us consider another sample. Figure 3 shows code from Andromeda sample, 3C8B018C238AF045F70B38FC27D0D640.



```
seg000:00A7E110                     sub_A7E110      proc near               ; CODE
seg000:00A7E110                                                             ; DATA
seg000:00A7E110 B8 9A 00 00 00                      mov     eax, 9Ah ; 'Ü'
seg000:00A7E115 E9 CB F6 E8 7B                      jmp     near ptr 7C90D7E5h
seg000:00A7E115                     sub_A7E110      endp
seg000:00A7E115
seg000:00A7E115                     ; ---------------------------------------------
seg000:00A7E11A 00 00 00 00 00 00                   align 10h
seg000:00A7E120
seg000:00A7E120                     ; =============== S U B R O U T I N E =========
seg000:00A7E120
seg000:00A7E120                     ; Attributes: noreturn
seg000:00A7E120
seg000:00A7E120                     sub_A7E120      proc near               ; CODE
seg000:00A7E120                                                             ; DATA
seg000:00A7E120
seg000:00A7E120                     arg_4           = dword ptr  8
seg000:00A7E120
seg000:00A7E120 8B 44 24 08                         mov     eax, [esp+arg_4]
seg000:00A7E124 E9 9F 34 E8 7B                      jmp     near ptr 7C9015C8h
seg000:00A7E124                     sub_A7E120      endp
```

Figure 3: API redirection code from an Andromeda sample

This code was found in a memory injection. Here we can see what looks to be a detours style trampoline, where the first instruction was stolen from the actual Windows API and placed in a small stub with an immediate jump taken back to the original API + x bytes.

In this situation, the malware accesses all of the API through these stubs and we have no clear resolution as to which stub points where. From the disassembly we can also see that the stolen instructions are of variable length.

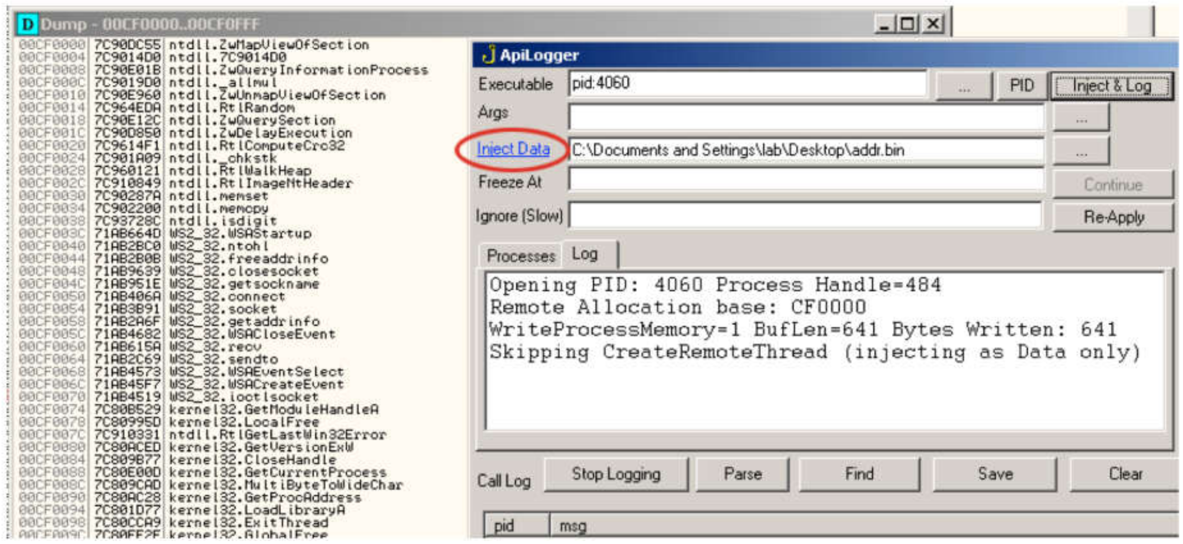
In order to resolve where these functions go, we would have to:

- enumerate all of the stubs
- calculate how many bytes are in the first instruction
- extract the jmp address
- subtract the stolen byte count to find the API entrypoint
- resolve the calculated address for this specific process instance
- rename the stub to a meaningful value

In this sample, looking for cross references on where the value is set does not yield any results.

Here we have two manifestations of essentially the same problem. How do we best resolve calculated API addresses and add this information back into our IDA database?

One of the first techniques used was to calculate all of the final addresses, write them to a binary file, inject the data into the process, and examine the table in the debugger (Figure 4). Since the debugger already has a API address look up table, this gives a crude yet quick method to get the information we need.



```
00CF0000 7C90DC55 ntdll.ZwMapViewOfSection
00CF0004 7C9014D0 ntdll.7C9014D0
00CF0008 7C90E01B ntdll.ZwQueryInformationProcess
00CF000C 7C9019D0 ntdll._allmul
00CF0010 7C90E960 ntdll.ZwUnmapViewOfSection
00CF0014 7C964EDA ntdll.RtlRandom
00CF0018 7C90E12C ntdll.ZwQuerySection
00CF001C 7C90D850 ntdll.ZwDelayExecution
00CF0020 7C9614F1 ntdll.RtlComputeCrc32
00CF0024 7C901A09 ntdll._chkstk
00CF0028 7C960121 ntdll.RtlWalkHeap
00CF002C 7C910849 ntdll.RtlImageNtHeader
00CF0030 7C90287A ntdll.memset
00CF0034 7C902200 ntdll.memcpy
00CF0038 7C93728C ntdll.isdigit
00CF003C 71AB664D WS2_32.WSAStartup
00CF0040 71AB2BC0 WS2_32.ntohl
00CF0044 71AB2B0B WS2_32.freeaddrinfo
00CF0048 71AB9639 WS2_32.closesocket
00CF004C 71AB951E WS2_32.getsockname
00CF0050 71AB406A WS2_32.connect
00CF0054 71AB3B91 WS2_32.socket
00CF0058 71AB2A6F WS2_32.getaddrinfo
00CF005C 71AB4682 WS2_32.WSACloseEvent
00CF0060 71AB615A WS2_32.recv
00CF0064 71AB2C69 WS2_32.sendto
00CF0068 71AB4573 WS2_32.WSAEventSelect
00CF006C 71AB45F7 WS2_32.WSACreateEvent
00CF0070 71AB4519 WS2_32.ioctlsocket
00CF0074 7C80B529 kernel32.GetModuleHandleA
00CF0078 7C80995D kernel32.LocalFree
00CF007C 7C910331 ntdll.RtlGetLastWin32Error
00CF0080 7C80ACED kernel32.GetVersionExW
00CF0084 7C809B77 kernel32.CloseHandle
00CF0088 7C80E00D kernel32.GetCurrentProcess
00CF008C 7C809CAD kernel32.MultiByteToWideChar
00CF0090 7C80AC28 kernel32.GetProcAddress
00CF0094 7C801D77 kernel32.LoadLibraryA
00CF0098 7C80CCA9 kernel32.ExitThread
00CF009C 7C80FF2F kernel32.GlobalFree
```

Executable: pid:4060
Args:
Inject Data: C:\Documents and Settings\lab\Desktop\addr.bin
Freeze At:
Ignore (Slow):

```
Opening PID: 4060 Process Handle=484
Remote Allocation base: CF0000
WriteProcessMemory=1 BufLen=641 Bytes Written: 641
Skipping CreateRemoteThread (injecting as Data only)
```

Figure 4: ApiLogger from iDefense MAP injecting a data file into a process and examining results in debugger

From here we can extract the resolved symbols and write a script to integrate them into our IDB. This works, but it is bulky and involves several steps.

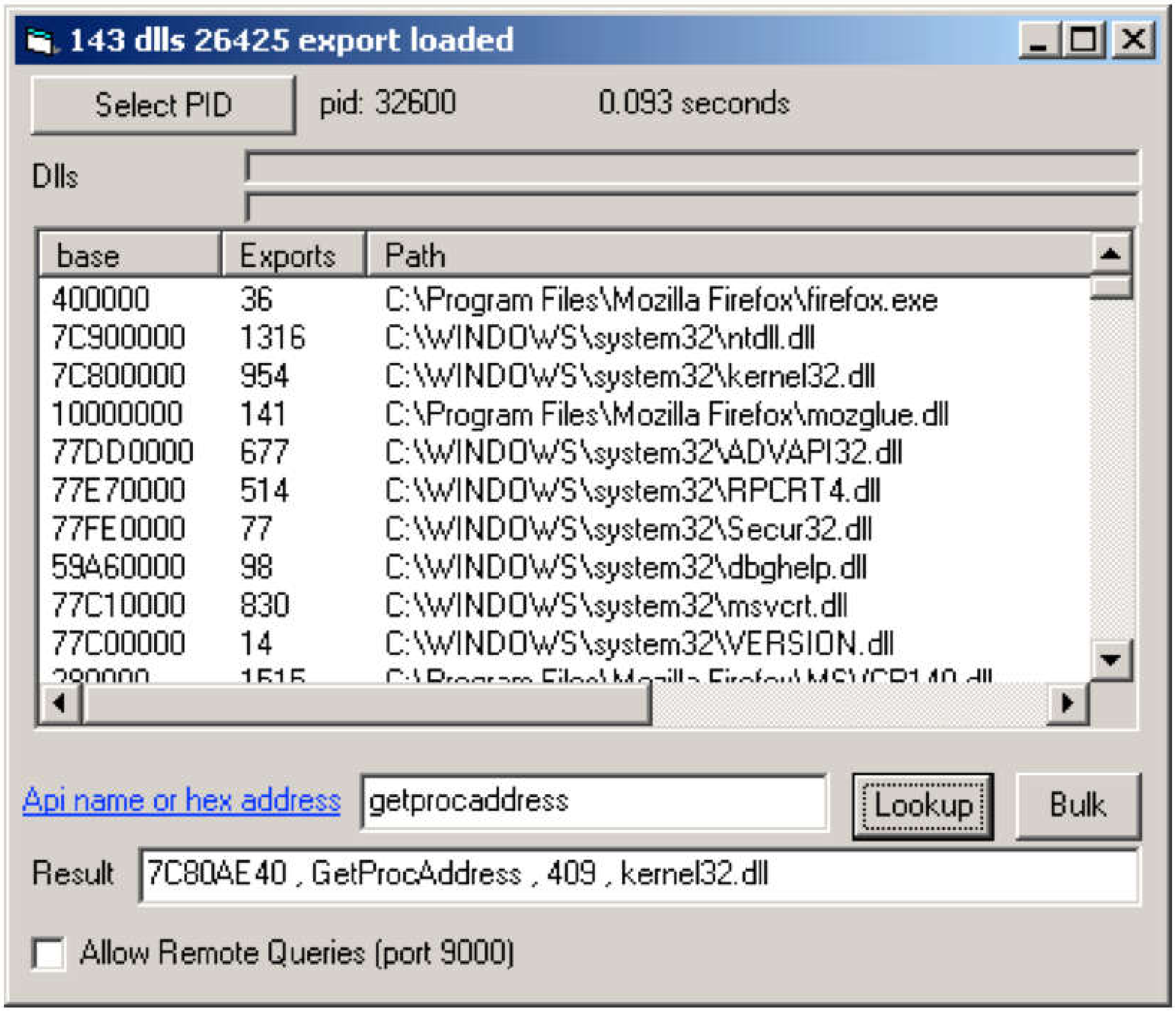
Our Tool

What we really want is to build our own symbol lookup table for a process and create a streamlined way to access it from our scripts.

The first question is: How can we build our own lookup table of API addresses to API names? To resolve this information, we need to follow some steps:

- enumerate all of the DLLs loaded into a process
- for each DLL, walk the export table and extract function name and RVA
- calculate API entrypoint based on DLL base address and export RVA
- build a lookup table based on all of this information

While this sounds like a lot of work, libraries are already available that handle all of the heavy lifting. Figure 5 shows a screenshot of a remote lookup tool we developed for such occasions.


143 dlls 26425 export loaded
Select PID
Dlls
Api name or hex address
Lookup
Bulk
Allow Remote Queries (port 9000)

pid: 32600 0.093 seconds

base	Exports	Path
400000	36	C:\Program Files\Mozilla Firefox\firefox.exe
7C900000	1316	C:\WINDOWS\system32\ntdll.dll
7C800000	954	C:\WINDOWS\system32\kernel32.dll
10000000	141	C:\Program Files\Mozilla Firefox\mozglue.dll
77DD0000	677	C:\WINDOWS\system32\ADVAPI32.dll
77E70000	514	C:\WINDOWS\system32\RPCRT4.dll
77FE0000	77	C:\WINDOWS\system32\Secur32.dll
59A60000	98	C:\WINDOWS\system32\dbghelp.dll
77C10000	830	C:\WINDOWS\system32\msvcrt.dll
77C00000	14	C:\WINDOWS\system32\VERSION.dll

getprocaddress

Result 7C80AE40 , GetProcAddress , 409 , kernel32.dll

Figure 5: Open source remote lookup application

In order to maximize the benefits of this type of tool, the tool must be efficient. What is the best way to interface with this data? There are several factors to consider here, including how the data is submitted, what input formats are accepted, and how well the tool can be integrated with the flow of the analysis process.

The first consideration is how we interface with it. For maximum flexibility, three methods were chosen. Lookups can be submitted:

- individually via textbox
- in bulk by file or
- over the network by a remote client

In terms of input formats, it accepts the following:

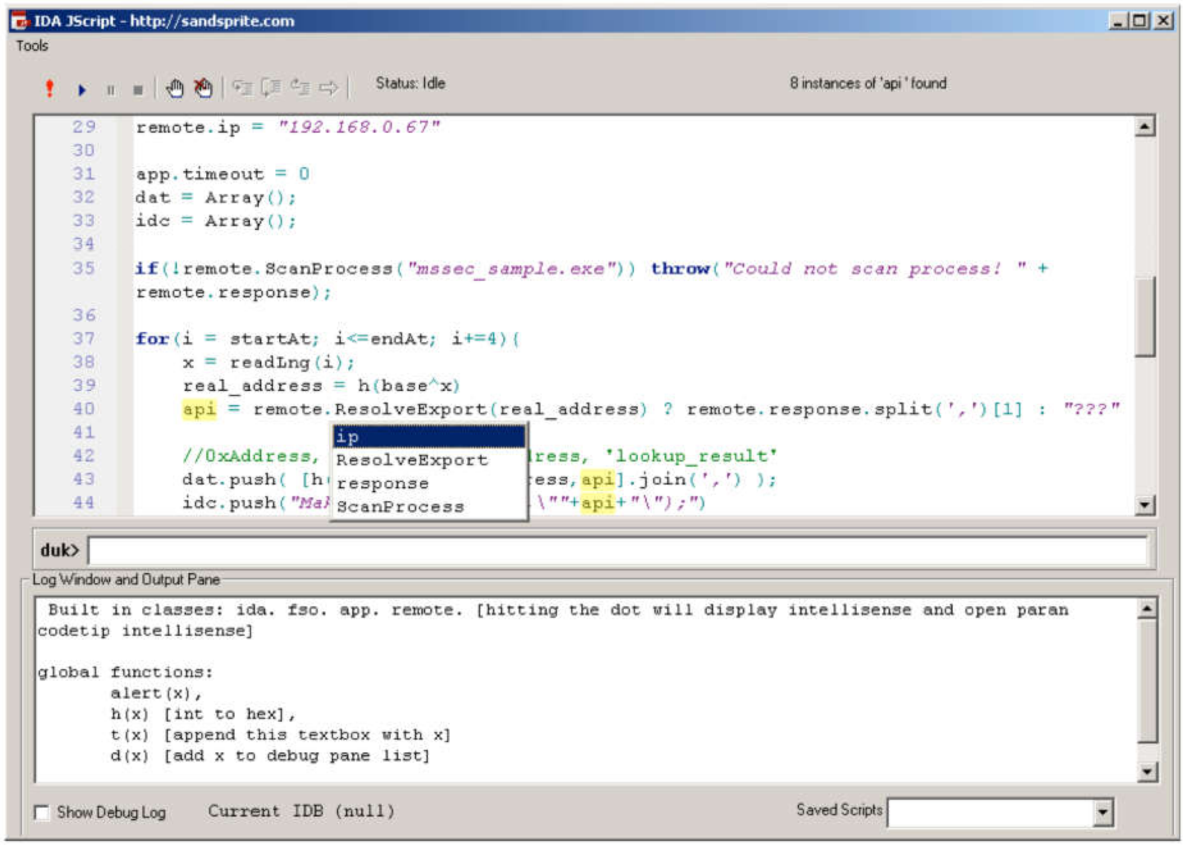
- hex memory address
- case insensitive API name
- dll_name@ordinal
- dll_name.export_name

The tool output is in the form of a CSV list that includes address, name, ordinal, and DLL.

With the base tool capabilities in place, we still need an efficient streamlined way to use it during our analysis. The individual lookups are nice for offhand queries and testing, but not in bulk. The bulk file lookup is nice on occasion, but it still requires data export/import to integrate results with your IDA database.

What is really needed is a way to run a script in IDA, calculate the API address, and then resolve that address inline while running an IDA script. This allows us to rename functions and pointers on the fly as the script runs all in one shot. This is where the network client capability comes in.

Again, there are many approaches to this. Here we chose to integrate a network client into a beta of IDA Jscript (Figure 6). IDA Jscript is an open source IDA scripting tool with IDE that includes syntax highlighting, IntelliSense, function prototype tooltips, and debugger.


IDA JScript - http://sandsprite.com
Tools
Status: Idle
8 instances of 'api' found
ip
ResolveExport
response
ScanProcess
duk>
Log Window and Output Pane
Show Debug Log
Saved Scripts

```
remote.ip = "192.168.0.67"

app.timeout = 0
dat = Array();
idc = Array();

if(!remote.ScanProcess("mssec_sample.exe")) throw("Could not scan process! " +
remote.response);

for(i = startAt; i<=endAt; i+=4){
    x = readLng(i);
    real_address = h(base^x)
    api = remote.ResolveExport(real_address) ? remote.response.split(',')[1] : "???"

    //0xAddress,                ress, 'lookup_result'
    dat.push( [h                ess,api].join(',') );
    idc.push("Ma                \""+api+"\");")
```

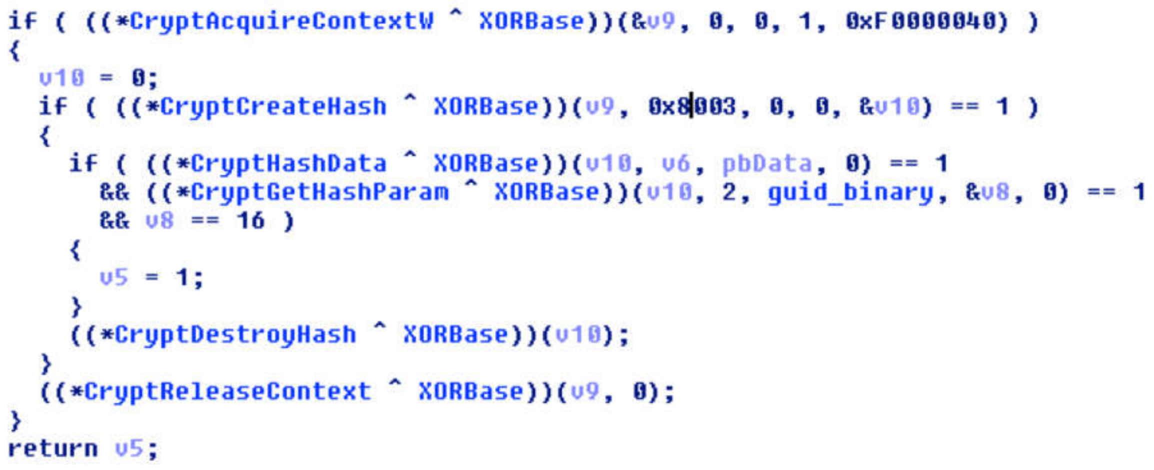
```
 Built in classes: ida. fso. app. remote. [hitting the dot will display intellisense and open paran
codetip intellisense]

global functions:
        alert(x),
        h(x) [int to hex],
        t(x) [append this textbox with x]
        d(x) [add x to debug pane list]
```

```
Current IDB (null)
```

Figure 6: Open source IDA Jscript decoding and resolving API addresses

In this example we see a script that decodes the xdata pointer table, resolves the API address over the network, and then generates an IDC script to rename the pointers in IDA.

After running this script and applying the results, the decompiler output becomes plainly readable (Figure 7).

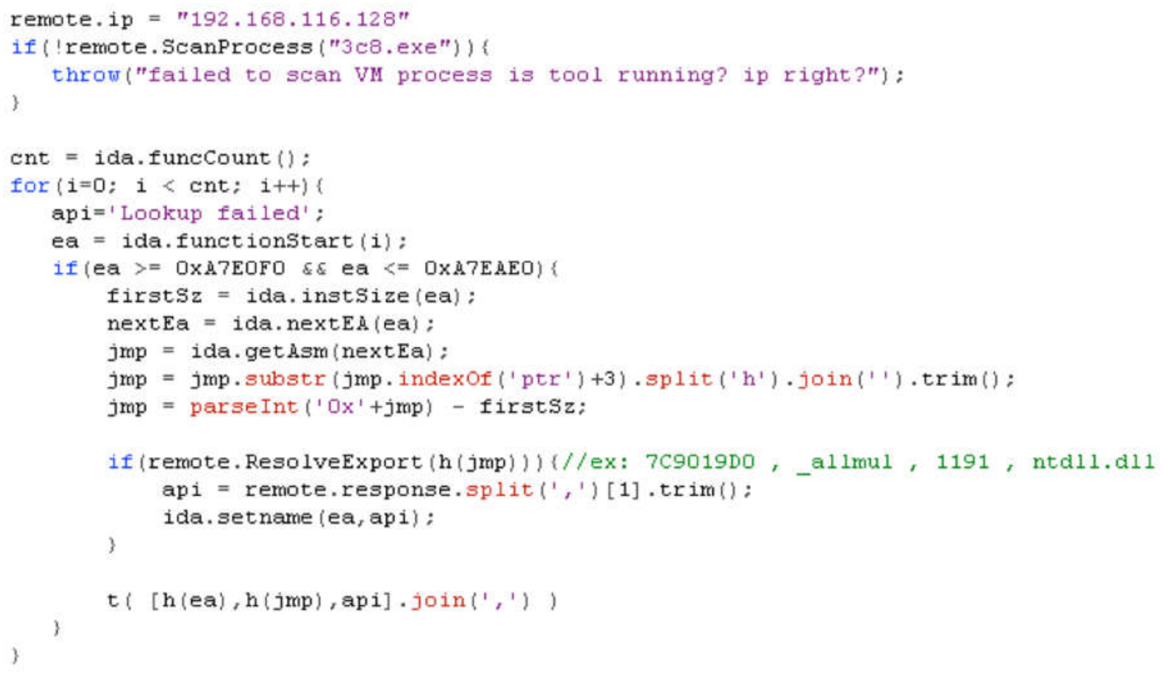


```
if ( ((*CryptAcquireContextW ^ XORBase))(&v9, 0, 0, 1, 0xF0000040) )
{
  v10 = 0;
  if ( ((*CryptCreateHash ^ XORBase))(v9, 0x8003, 0, 0, &v10) == 1 )
  {
    if ( ((*CryptHashData ^ XORBase))(v10, v6, pbData, 0) == 1
      && ((*CryptGetHashParam ^ XORBase))(v10, 2, guid_binary, &v8, 0) == 1
      && v8 == 16 )
    {
      v5 = 1;
    }
    ((*CryptDestroyHash ^ XORBase))(v10);
  }
  ((*CryptReleaseContext ^ XORBase))(v9, 0);
}
return v5;
```

Figure 7: Decompiler output from the xdata sample after symbol

resolution

Going back to the Andromeda sample, the API information can be restored with the brief idajs script shown in Figure 8.



```
remote.ip = "192.168.116.128"
if(!remote.ScanProcess("3c8.exe")){
   throw("failed to scan VM process is tool running? ip right?");
}

cnt = ida.funcCount();
for(i=0; i < cnt; i++){
   api='Lookup failed';
   ea = ida.functionStart(i);
   if(ea >= 0xA7E0F0 && ea <= 0xA7EAE0){
       firstSz = ida.instSize(ea);
       nextEa = ida.nextEA(ea);
       jmp = ida.getAsm(nextEa);
       jmp = jmp.substr(jmp.indexOf('ptr')+3).split('h').join('').trim();
       jmp = parseInt('0x'+jmp) - firstSz;

       if(remote.ResolveExport(h(jmp))){//ex: 7C9019D0 , _allmul , 1191 , ntdll.dll
           api = remote.response.split(',')[1].trim();
           ida.setname(ea,api);
       }

       t( [h(ea),h(jmp),api].join(',') )
   }
}
```

Figure 8: small idajs script to remotely resolve and rename Andromeda API hook stubs

For IDAPython users, a python remote lookup client is also available.

Conclusion

It is common for malware to use techniques that mask the Windows API being used. These techniques force malware analysts to have to extract data from runtime data, calculate entry point addresses, and then resolve their meaning within the context of a particular running process.

In previous techniques, several manual stages were involved that were bulky and time intensive.

This blog introduces a small simple open source tool that can integrate well into multiple IDA scripting languages. This combination allows analysts streamlined access to the data required to quickly bypass these types of obfuscations and continue on with their analysis.

We are happy to be able to open source the remote lookup application so that others may benefit and adapt it to their own needs. Sample network clients have been provided for Python, C#, D, and VB6.

Download a copy of the tool today.