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.

```assembly
.data: 00410A4  dw 410A4  dd 43C1FBF5h
.data: 00412354  dw 412354  dd 3F6AA05h

.text: 00409B4E
    mov  eax, dword_410A4
    xor  eax, dword_412354
    push  esi
    push  0
    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.

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

.text: 0040AF22
    push  offset dword_412354
    mov  edx, [ebp+var_C0]
    push  edx
    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.

![Figure 3: API redirection code from an Andromeda sample](https://www.fireeye.com/blog/threat-research/2017/06/...)

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.

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.
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.
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).

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.

```python
def main():
    remote_ip = "192.168.116.128"
    if not remote.ScanProcess("328.exe"):
        throw("failed to scan VM process is tool running? ip right?");

    cnt = ida.funcCount();
    for i in range(0, cnt):
        api = 'Lookup failed';
        ea = ida.functionStart(i);
        if ea >= 0x4A652F && ea <= 0x4A7E45C:
            firstSz = ida.insSize(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.