Defeating BazarLoader Anti-Analysis Techniques

- 1,558 people reacted
- 4
- 7 min. read

By Mark Lim

April 25, 2022 at 6:00 AM

Category: Malware

Tags: anti-analysis, BazarLoader

Executive Summary

Malware authors embed multiple anti-analysis techniques in their code to retard the analysis processes of human analysts and sandboxes. However, there are ways defenders can defeat these techniques in turn. This blog post describes two methods for faster analysis of malware that employs two distinctive anti-analysis techniques. The first technique is API function hashing, a known trick to obfuscate which functions are called. The second is opaque predicate, a technique used for control flow obfuscation.

The scripts that we are going to show here can be applied to BazarLoader, as well as other malware families that utilize similar anti-analysis techniques. As an illustration, we will show the IDAPython scripts we created during a recent analysis of BazarLoader with the reverse engineering tool IDA Pro to defeat these anti-analysis techniques. BazarLoader is a Windows backdoor that is used by various ransomware groups.

Palo Alto Networks customers are protected from malware families using similar anti-analysis techniques with Cortex XDR or the Next-Generation Firewall with the WildFire and Threat Prevention security subscriptions.

Primary Malware Discussed	BazarLoader
Related Unit 42 Topics	Malware, anti-analysis techniques

Table of Contents

Reusing Malware Code to Defeat Obfuscated API Calls

Malware compiled as native files has to call Windows API functions to carry out malicious behaviors. The information on which functions are used is usually stored in the Import Address Table (IAT) in the file. Therefore, this table is often a good place to start the analysis process to get an idea of what the malware is trying to do.

To demonstrate, we focused on a BazarLoader sample we recently detected. After peeling away the packer layer of our BazarLoader sample, we saw that it doesn't have an IAT (see Figure 1). Also, there is no IAT constructed during execution, a technique sometimes seen in other malware. BazarLoader obfuscates its function calls to make analysis more difficult and to evade detection techniques that rely on reading the IAT.

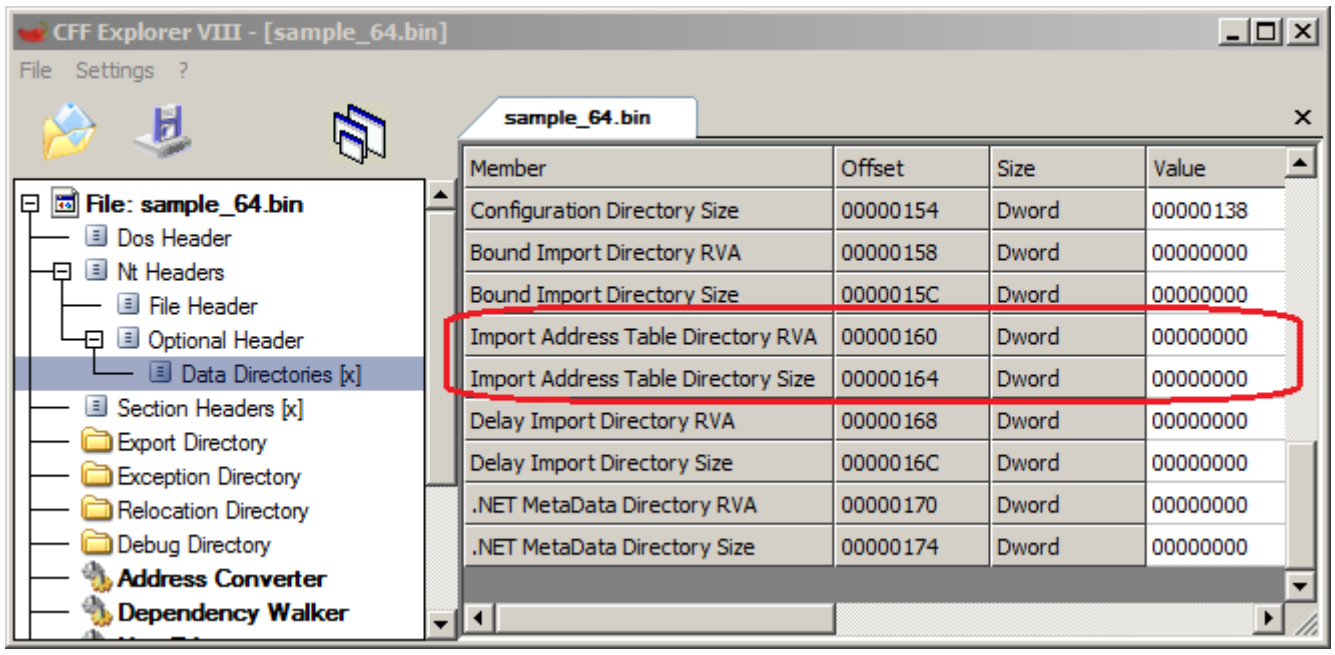

Figure 1. Missing IAT in BazarLoader as seen with CFF Explorer.

In fact, BazarLoader resolves every API function to be called individually at run time. After we figured out that the functions are resolved during execution, the following function caught our attention as it was referenced more than 300 times:

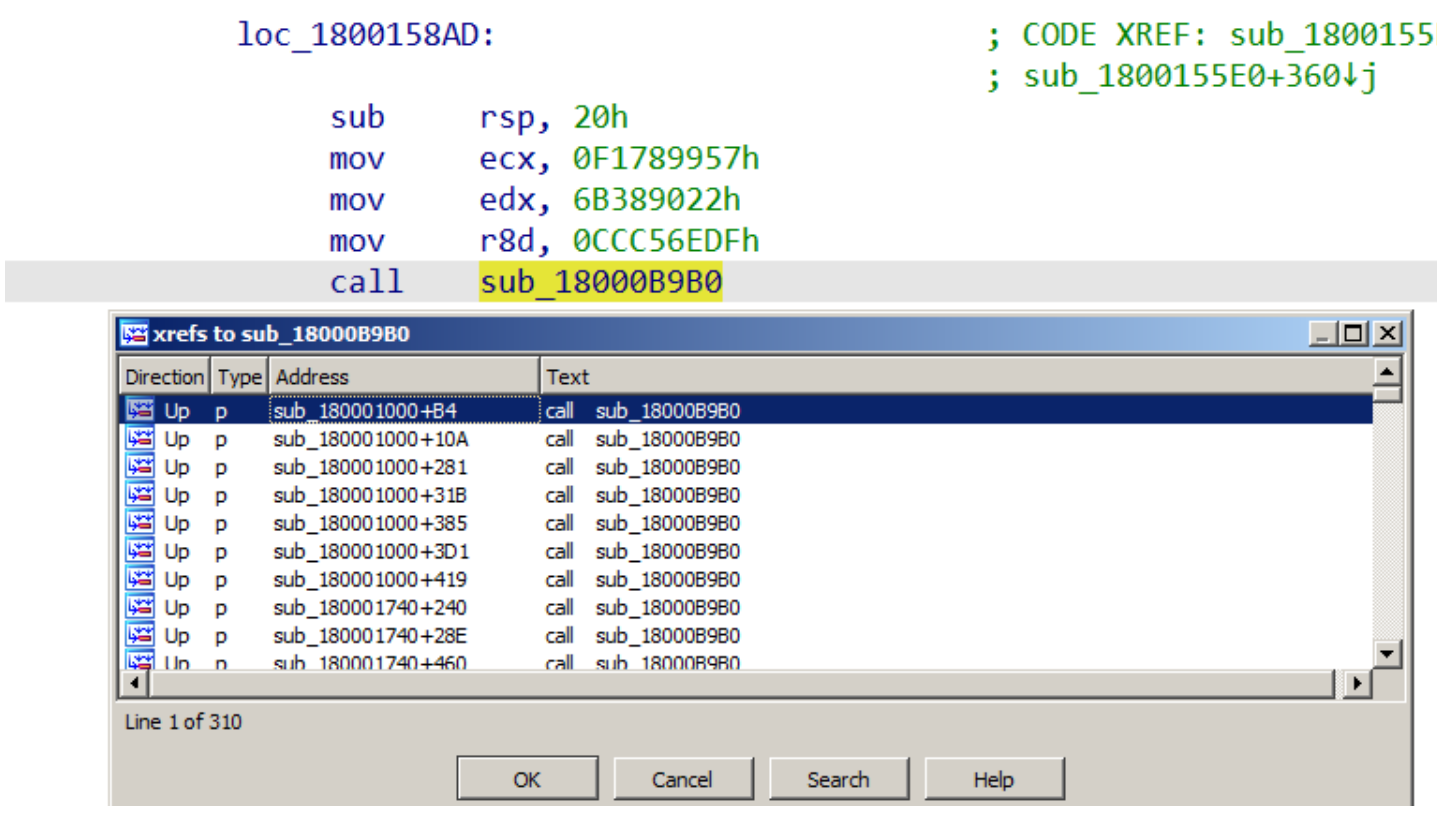

Figure 2. Function for resolving the obfuscated Windows API functions (marked in yellow).

While most pieces of malware rely on publicly known hashing algorithms to resolve the functions' addresses, the one used by BazarLoader is unique. The API function resolution procedure (sub_18000B9B0, labelled as FN_API_Decoder) requires three parameters and returns the address of the requested function.

Now, we could reverse engineer the algorithm used in FN_API_Decoder and reimplement it in Python to get all functions resolved. However, this would take a lot of time and we would have to repeat the whole process for every piece of malware that uses a different hashing algorithm.

Instead, the approach we used is independent from the hashing algorithm as it makes use of the hashing function itself. For this, we used the Appcall feature with IDAPython in IDA Pro to call FN_API_Decoder and pass it the required parameters. The result from Appcall would be the resolved address of the Windows API function. The Appcall feature used while debugging the malware allows us to execute any function from the sample as if it were a built-in function.

Using the following code, we can run FN_API_Decoder to resolve Windows API function addresses while debugging the malware process.

```
def get_arg(ea):
    """
    Extract the 3 values of the parameters (rcx,rdx,r8) for a given effective address
    :param ea: effective address
    :return: Tuple of operands
    """
    if print_insn_mnem(ea) == "mov":
        operand = print_operand(ea, 0)
        if operand in ["r8", "r8d", "rcx", "ecx", "rdx", "edx"]:
            op1 = get_operand_value(ea, 1) & 0xFFFFFFFF
            return operand, op1

    return None, None
```

Figure 3. Using Appcall with IDAPython.

Next, we gathered all the required parameters by looking up all the cross references to FN_API_Decoder. The following code will search and extract the required parameters for resolving the API function calls.

```
def get_args_of_func(addr, args_count, lookup_limit=0x20):
    """
    search for the argument of a function.
    The search will be up-to "lookup_limit" bytes before addr
    :param addr: address of an xref to function
    :param args_count: number of arguments to look for
    :param lookup_limit: the search limit in bytes, before addr
    :return: dictionary of arguments
    """
    args = {}
    curr_addr = addr
    found_args = 0
    # While loop to locate the args_count parameters for addr
    while curr_addr > (addr - lookup_limit) and found_args < args_count:
        curr_addr = idc.prev_head(curr_addr)
        try:
            register, operand_value = get_arg(curr_addr)
            if register is not None and operand_value is not None:
                found_args += 1
                args[register] = operand_value
                # adding the 8-byte regs for ease of use
                if register == 'ecx':
                    args['rcx'] = operand_value
                elif register == 'edx':
                    args['rdx'] = operand_value
                elif register == 'r8d':
                    args['r8'] = operand_value
                elif register == 'r9d':
                    args['r9'] = operand_value

        except TypeError:
            pass

    return args
```

Figure 4. IDAPython code to search and extract the three parameters.

Finally, by using the returned value from Appcall we are able to rename all the dynamic calls to the APIs to their corresponding names and apply comments:

```python
def check_instruction_call_reg(ea):
    """
    Check if the instruction at the current address is "call rax/eax/rdi/edi"
    :param ea:  effective address
    :return:
    """
    if print_insn_mnem(ea) == "call":
        if print_operand(ea, 0) in ["rax", "eax","rdi", "edi","r14"]:
            return True

    return False
```

Figure 5. IDAPython code to locate dynamic calls.

Putting the above steps together, we deobfuscated the API function calls:

```
0000000180019448D loc_18001944D:                          ; CODE XREF: sub_180019120+320↑j
0000000180019448D                                         ; sub_180019120+406↓j ...
0000000180019448D       mov     ecx, 8B4B5059h
0000000180019452       mov     edx, 0E1BC2710h
0000000180019457       mov     r8d, 0B6F6A7D1h
0000000180019457       call    sub_18000B9B0
0000000180019462       call    rax
0000000180019464       mov     [r14+818h], eax
```

Figure 6. Before executing the above IDAPython scripts.

```
0000000180019448D
0000000180019448D loc_18001944D:                          ; CODE XREF: sub_180019120+320↑j
0000000180019448D                                         ; sub_180019120+406↓j ...
0000000180019448D       mov     ecx, 8B4B5059h
0000000180019452       mov     edx, 0E1BC2710h
0000000180019457       mov     r8d, 0B6F6A7D1h
0000000180019457       call    FN_API_Decoder            ; kernel32_GetTickCount
0000000180019462       call    GetTickCount
0000000180019464       mov     [r14+818h], eax
```

Figure 7. Renamed API function call with added comment.

After all the API function calls are renamed, we can now easily locate other interesting functions in the malware. For example, sub_1800155E0 is the procedure in BazarLoader that carries out code injection.

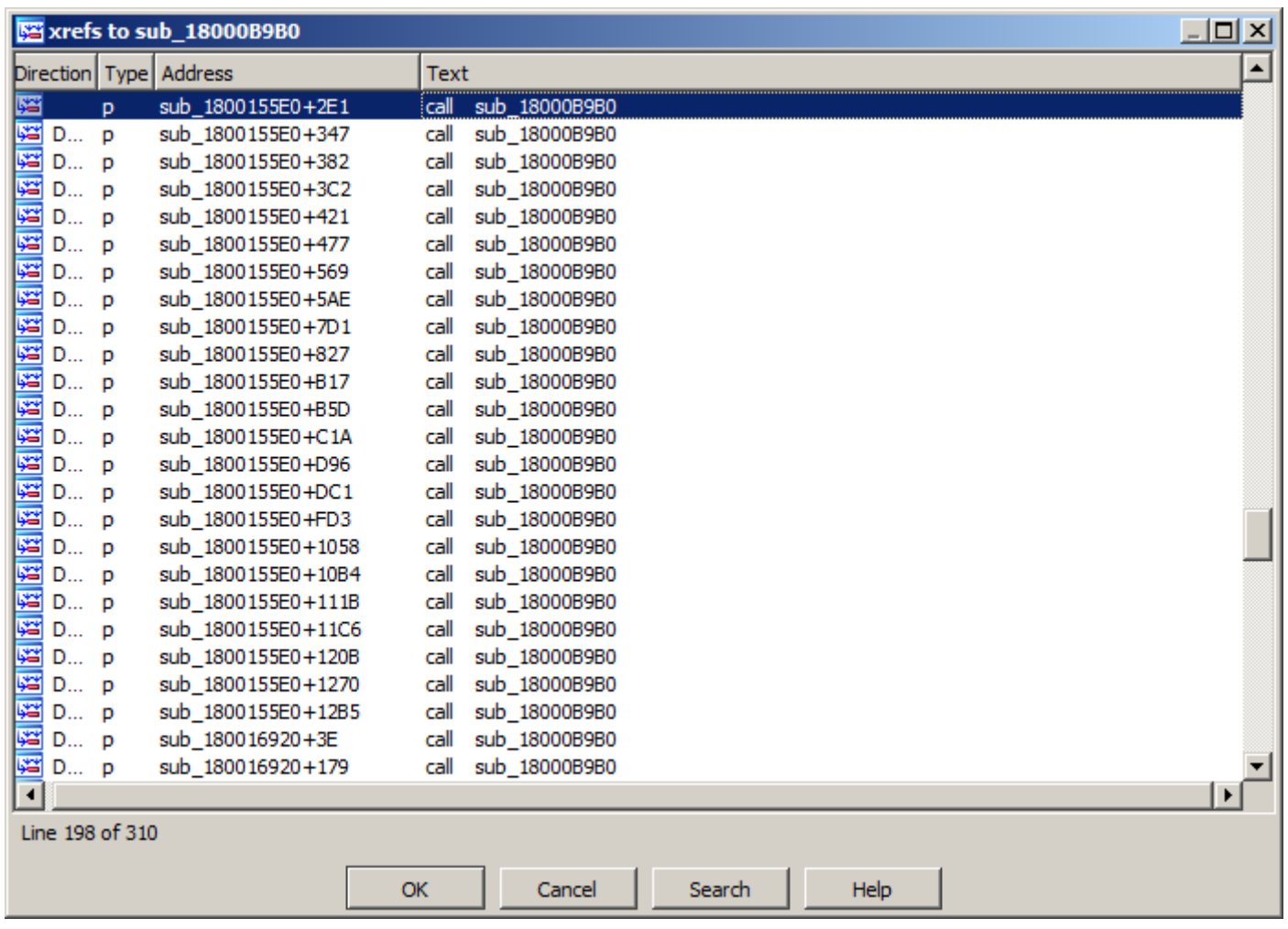

Figure 8. Before renaming API calls.

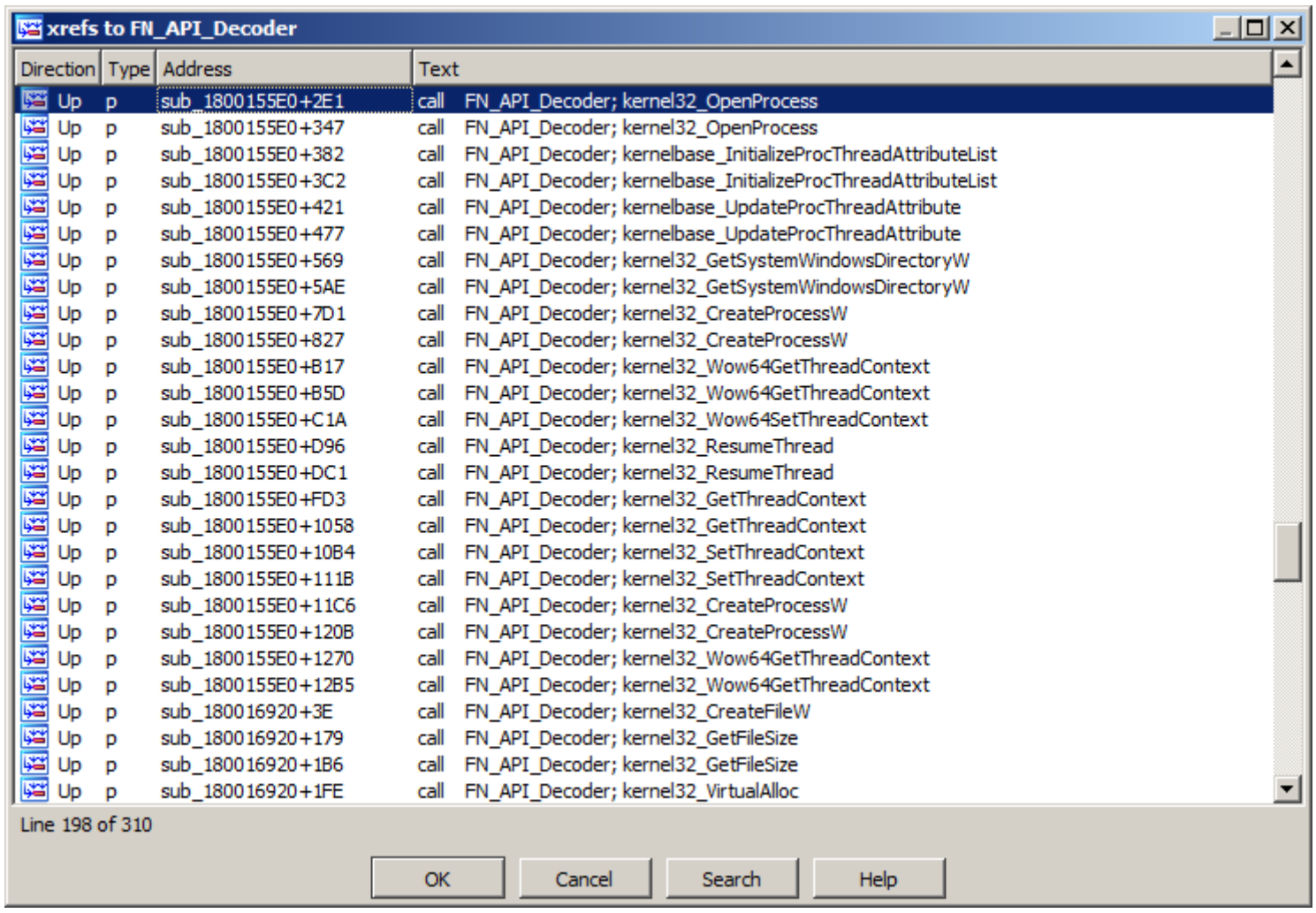

Figure 9. Obfuscated API calls labeled with APIs related to code injection.

With the help of our IDAPython scripts, we are now able to faster assess which functionality this BazarLoader sample contains.

Automating Opaque Predicate Removal

Opaque Predicate (OP) is used in BazarLoader to protect it from reverse engineering tools. OP is an expression that evaluates to either true or false at runtime. Malware authors make use of multiple OPs together with unexecuted code blocks to add complexities that static analysis tools have to deal with.

The following disassembled code shows one of the OPs in Bazarloader:

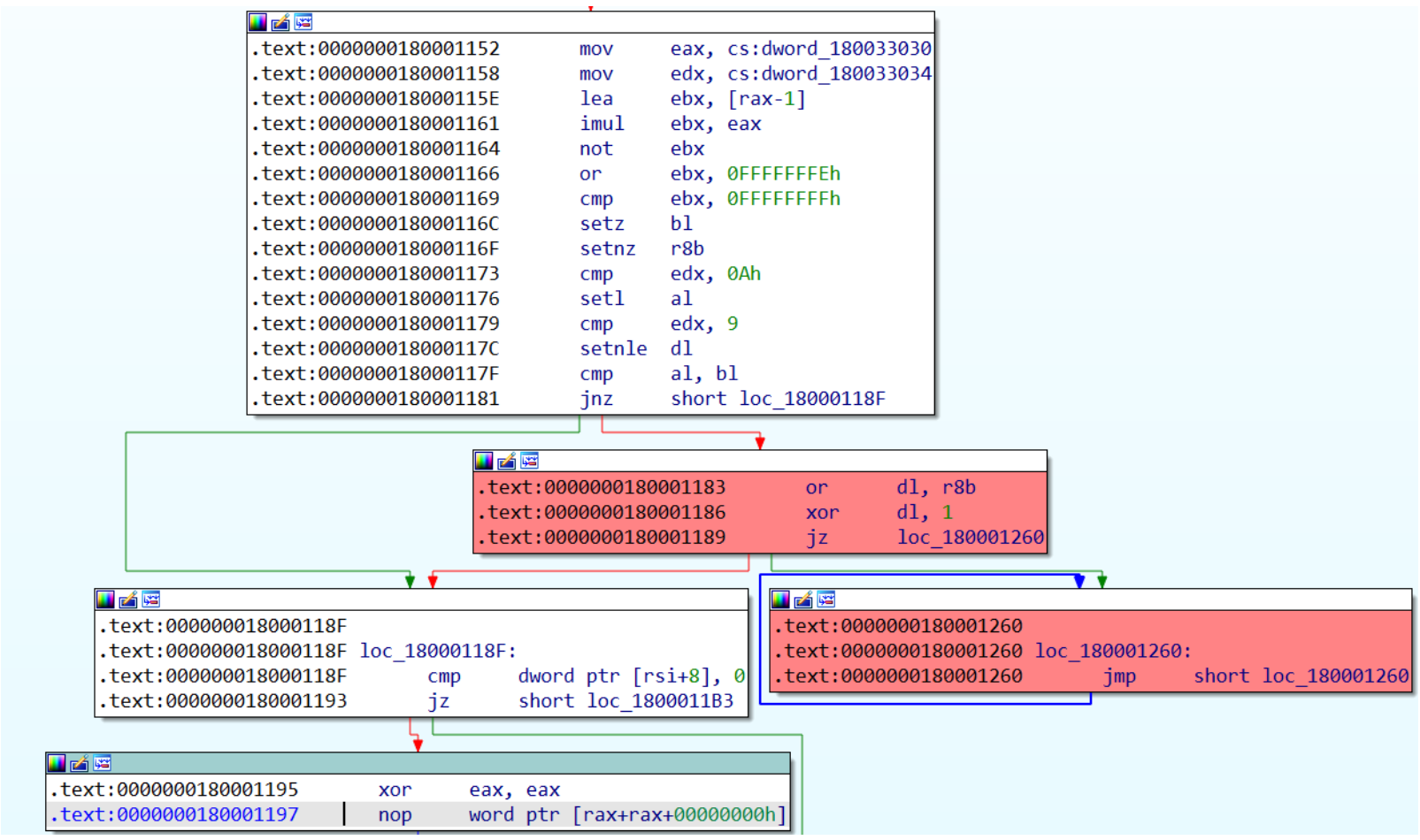

Figure 10. One example of OP in BazarLoader.

From the above control flow graph (CFG), the code flow won't end up in infinite loops (Figure 10, red code blocks). Therefore, the above OP will be evaluated to avoid the infinite loop.

We can demonstrate the extent of the challenge OPs pose to malware analysts. The following CFG shows the unexecuted code blocks (Figure 11, red code blocks) in one of the smaller functions (sub_18000F640) in the sample.

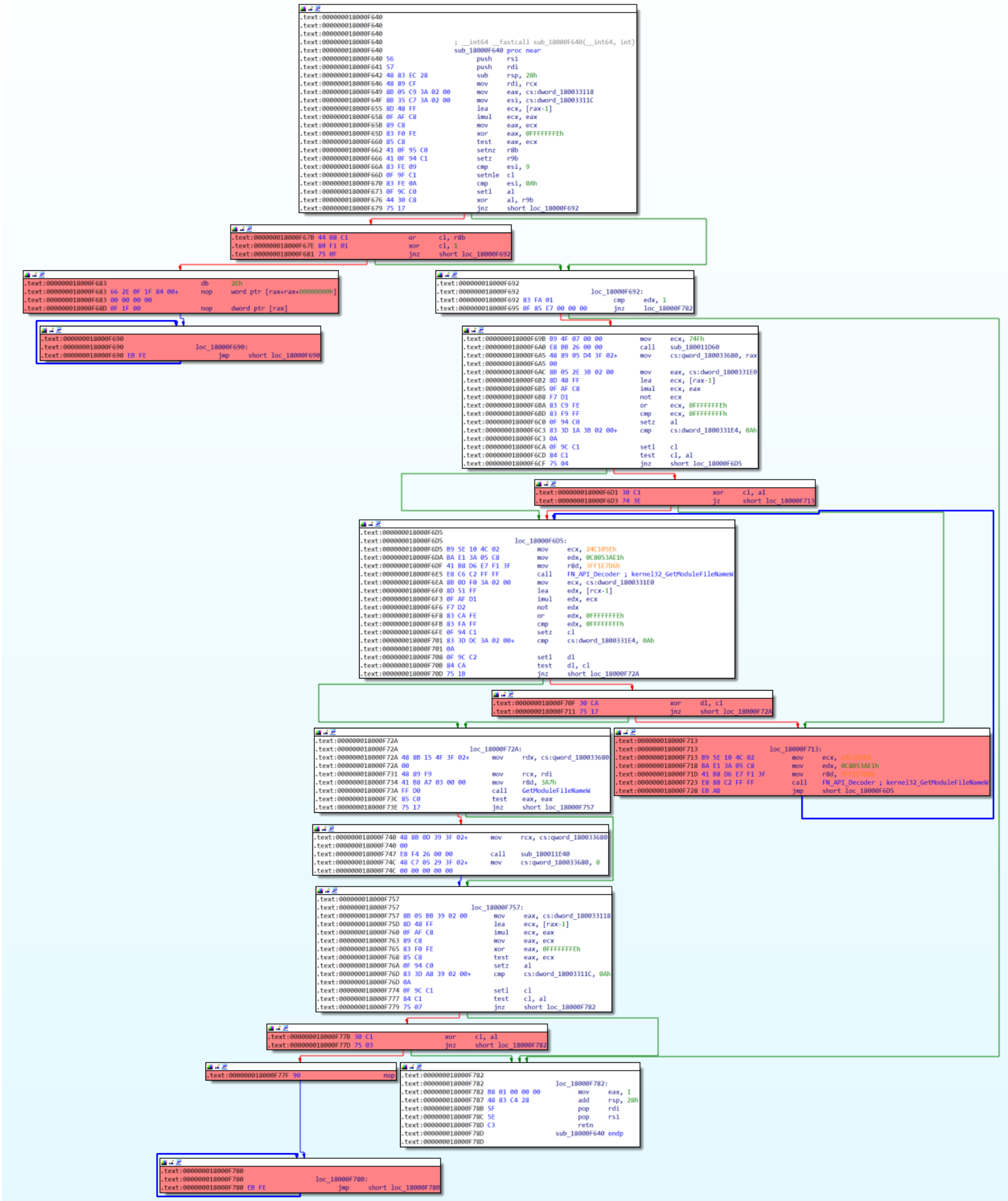

Figure 11. sub_18000F640 function in BazarLoader with unexecuted code blocks colored in red.

We could manually patch away the code blocks that are not executed as we analyze each function in the sample, but this is not very practical and takes a lot of time. Instead, we will choose a smarter way by doing it automatically.

First, we have to locate all the OPs. The most common way to do this is to make use of the binary search mechanism in IDA Pro to find all the byte sequences of the OPs. This turns out not to be possible, as the OPs were likely generated by a compiler during the build process of the malware sample. There are just too many variants of the OPs that could be covered using the byte sequence.

Not only do we need to locate the OPs, we also have to know the exact point when the malware sample decides to avoid the unexecuted code blocks.

Using the following code, we locate the OPs in a function:

```python
def search_jz_or_jnz(ea, lookup_limit=0x10):
    """
    searches for both jz and jnz instructions, upto lookup_limit
    :param ea: start address for search jz or jnz
    :param lookup_limit: the search limit in bytes, after addr
    :return: address of "jz / jnz" instruction, or None if not found
    """
    ea_search_limit = ea + lookup_limit
    while ea < ea_search_limit:
        try:
            instr = idautils.DecodeInstruction(ea)
            if instr.itype == idaapi.NN_jnz or instr.itype == idaapi.NN_jz:
                return instr
        except Exception:
            pass
        ea = idc.next_head(ea)

    return None
```

Figure 12. IDAPython code to locate the OPs in a function.

Next, we have to patch the instructions in OPs to force the code flow away from the unexecuted code blocks.

Using the following code, we patch the OPs in a function:

```python
def locate_and_patch_opaque(ea):
    """
    search for:
        - cmp reg,0xA
        - whatever instructions
        - jnz pattern
    patches the found jz/jnz instruction to NOPs
    :param ea: effective address to check
    :return:
    """
    instr = idautils.DecodeInstruction(ea)
    # check if this is a CMP instrucion and the operand is 0xA, as can be found in our sample
    if instr.itype == idaapi.NN_cmp and get_operand_value(ea, 1) == 0xA:
        cmp_ea = ea
        # locate the point when OP decide to avoid unexecuted code blocks
        j_instr = search_jz_or_jnz(ea)
        if j_instr is not None:
            print(f'0x{cmp_ea:X} {idc.generate_disasm_line(cmp_ea, 0)}')
            print(f'0x{ea:X} {idc.generate_disasm_line(j_instr.ea, 0)}')
            # actually patching the instructions
            if j_instr.itype == idaapi.NN_jnz:
                patch_bytes(j_instr.ea, PATCH_INSTRUCTIONS_JNZ)
            elif j_instr.itype == idaapi.NN_jz:
                patch_bytes(j_instr.ea, PATCH_INSTRUCTIONS_JZ)

            idc.set_cmt(j_instr.ea, f"{j_instr.get_canon_mnem()}_patched!!", 0)
```

Figure 13. IDAPython code to patch the OPs.

The OPs also messed with the output of the HexRays decompiler. This is how the function (sub_18000F640) looks before the OPs are patched:

```
__int64 __fastcall sub_18000F640(__int64 a1, int a2)
{
  int v3; // ecx
  bool v4; // al
  unsigned int (__fastcall *v5)(__int64, __int64, __int64); // rax
  bool v6; // cl
  __int64 v7; // rdx
  __int64 v8; // r8
  bool v9; // al

  v3 = dword_180033118 * (dword_180033118 - 1);
  if ( ((v3 & (v3 ^ 0xFFFFFFFE)) == 0) == dword_18003311C < 10 && ((v3 & (v3 ^ 0xFFFFFFFE)) != 0 || dword_18003311C > 9) )
  {
    while ( 1 )
      ;
  }
  if ( a2 == 1 )
  {
    qword_180033680 = sub_180011D60(1871i64);
    v4 = (~((_BYTE)dword_1800331E0 * ((_BYTE)dword_1800331E0 - 1)) | 0xFFFFFFFE) == -1;
    if ( (!v4 || dword_1800331E4 >= 10) && v4 == dword_1800331E4 < 10 )
      goto LABEL_9;
    while ( 1 )
    {
      v5 = (unsigned int (__fastcall *)(__int64, __int64, __int64))sub_18000B9B0(38539358, -939181343, 1072818134);
      v6 = (~((_BYTE)dword_1800331E0 * ((_BYTE)dword_1800331E0 - 1)) | 0xFFFFFFFE) == -1;
      if ( v6 && dword_1800331E4 < 10 || v6 != dword_1800331E4 < 10 )
        break;
LABEL_9:
      sub_18000B9B0(38539358, -939181343, 1072818134);
    }
    if ( !v5(a1, qword_180033680, 935i64) )
    {
      sub_180011E40(qword_180033680, v7, v8);
      qword_180033680 = 0i64;
    }
    v9 = ((dword_180033118 * (dword_180033118 - 1)) & ((dword_180033118 * (dword_180033118 - 1)) ^ 0xFFFFFFFE)) == 0;
    if ( (!v9 || dword_18003311C >= 10) && v9 == dword_18003311C < 10 )
    {
      while ( 1 )
        ;
    }
  }
  return 1i64;
}
```

Figure 14. Decompiled sub_18000F640 function.

After applying the two techniques above, we have decompiled pseudocode that is much easier to read and understand.

After patching all the OPs and renaming the obfuscated API calls, we could then tell that the function (sub_18000F640) is just a wrapper function for GetModuleFileNameW().

```
void __fastcall sub_18000F640(HMODULE hModule, __int64 Enable_Flag)
{
  DWORD (__stdcall *GetModuleFileNameW)(HMODULE, LPWSTR, DWORD); // rax

  if ( (_DWORD)Enable_Flag == 1 )
  {
    moduleFileName_strW = (void *)FN_RtlAllocateHeap_wrapper(1871i64);
    GetModuleFileNameW = (DWORD (__stdcall *)(HMODULE, LPWSTR, DWORD))FN_API_Decoder(38539358, -939181343, 1072818134);
    if ( !GetModuleFileNameW(hModule, (LPWSTR)moduleFileName_strW, 0x3A7u) )
    {
      FN_Heap_cleanup(moduleFileName_strW);
      moduleFileName_strW = 0i64;
    }
  }
}
```

Figure 15. Decompiled sub_18000F640 function after removing the OPs.

Malware Analysts vs Malware Authors

Malware authors often include anti-analysis techniques with the hope that they will increase the time and resources taken for malware analysts. With the above script snippets showing how to defeat these techniques for BazarLoader, you can reduce the time needed to analyze malware samples of other families that use similar techniques.

Palo Alto Networks customers are further protected from malware families using similar anti-analysis techniques with Cortex XDR or the Next-Generation Firewall with the WildFire and Threat Prevention cloud-delivered security subscriptions.

Indicators of Compromise

BazarLoader Sample ce5ee2fd8aa4acda24baf6221b5de66220172da0eb312705936adc5b164cc052

Additional Resources

Complete IDAPython script to rename or resolve obfuscation API calls is available on GitHub.

Complete IDAPython script to search and patch Opaque Predicates in a function is available on GitHub.