Malware Analysis Report: "Practical2.exe"

CAP6137 Malware Reverse Engineering: P0x02

Naman Arora
naman.arora@ufl.edu

March 17, 2021

Contents

1 Executive Summary **3**

2 Static Analysis: Primary Executable **4**
2.1 Basic Identification . . . 4
2.2 Malware Sample Family Identification . . . 4
2.3 PE Sections . . . 4
2.3.1 The *.text*, *.rdata*, *.idata*, *.rsrc* and *.reloc* sections . . . 4
2.3.2 The *.data* Section . . . 5
2.3.3 The *.tls* Section . . . 5
2.3.4 The *.00cfg* Section . . . 5
2.4 A case for Packing . . . 5
2.5 Interesting Imports . . . 7
2.5.1 Imports from *Kernel32.dll* . . . 7
2.5.2 Imports from *user32.dll* . . . 7

3 Dynamic leading to Static Analysis: Unpacked Shell Code **7**
3.1 Basic Identification . . . 7
3.2 Sample Family Identification . . . 7
3.3 Shell Code Sections . . . 8
3.4 Interesting Imports . . . 8

4 Dynamic leading to Static Analysis: Unpacked PE Executable **9**
4.1 Basic Identification . . . 9
4.2 Sample Family Identification . . . 9
4.3 Packing and Further Obfuscation . . . 9
4.4 Interesting Imports . . . 9
4.4.1 Imports from *bcrypt.dll* . . . 9
4.4.2 Import from *urlmon.dll* . . . 9
4.4.3 Imports from *shell32.dll* . . . 9
4.4.4 Imports from *netapi32.dll* . . . 9
4.4.5 Imports from *advapi32.dll* . . . 9

5 Dynamic leading to Static Analysis: Unpacked DLL **10**
5.1 Basic Identification . . . 10
5.2 Sample Family Identification . . . 10
5.3 Sections . . . 10
5.4 Analysis . . . 10

6 Dynamic Analysis **10**
6.1 Network Based Analysis . . . 10
6.2 File System Based Analysis . . . 10
6.2.1 File System Changes . . . 10
6.2.2 Windows Registry Changes . . . 10

7 Indicators of Compromise **12**
7.1 Network Based . . . 12
7.2 Host Based . . . 12
7.3 *YARA* Rule . . . 12

8 Appendix A: Screenshots **13**

1 Executive Summary

The provided binary is a PE executable 32-bit Microsoft Windows platform. This is certainly a malicious executable and has string resemblance to *AveMariaRAT* and *WarZoneRAT* trojan families. The malware is obfuscated to thwart analysis in both static and dynamic phases.

The primary malware on execution unpacks an intermediary *shell-code* stage as well as another PE and writes them to its own memory. The execution then passes on to the said *shell-code* which then unpacks another PE, this time a *Dynamic Linked Library, DLL*, again within its own memory.

The malware [4], largely, is capable of,

- Soliciting remote desktop connections *RDP*
- Bypassing important Windows security features like *UAC*, and *Defender*
- Remotely monitoring using Webcam, KeyLogger, Process manager etc.
- Activating reverse proxy, poking hole in internal network for external access
- Upload, download and execution of files from internet on victim

IOCs mentioned towards the end of this report can be leveraged to detect this malware in transit on network or on file system.

2 Static Analysis: Primary Executable

2.1 Basic Identification

Attribute	Value
Bits	32
Endianess	Little
Operating System	Microsoft Windows
Class	PE32
Subsystem	Windows CUI
Size	1446912
Compiler Timestamp	Thu Dec 10 02:47:43 2020
Compiler	Visual Studio
SHA256 Hash	9633d0564a2b8f1b4c6e718ae7ab48be921d435236a403cf5e7ddfbfd4283382

2.2 Malware Sample Family Identification

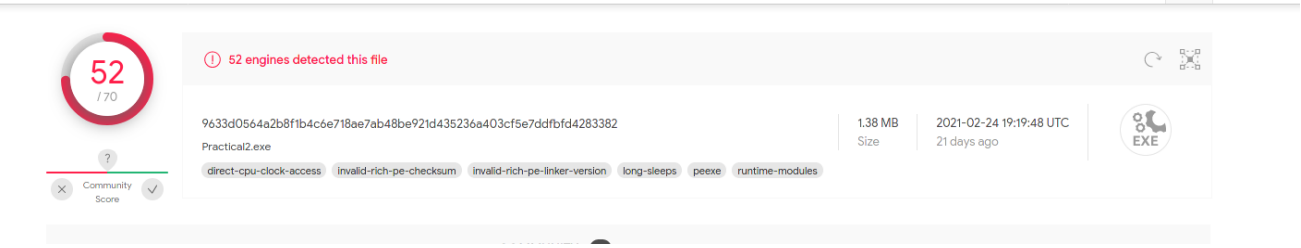

Figure 1: VirusTotal: VirusTotal Scan

The given PE file, on being uploaded to VirusTotal, is identified as a variant of *AveMariaRAT* [2] family (Fig. 1) As seen later in the *dynamic analysis* section, another in-memory PE when dumped and analysed on VirusTotal, is identified to belong to *WarZoneRAT* [4] family.

2.3 PE Sections

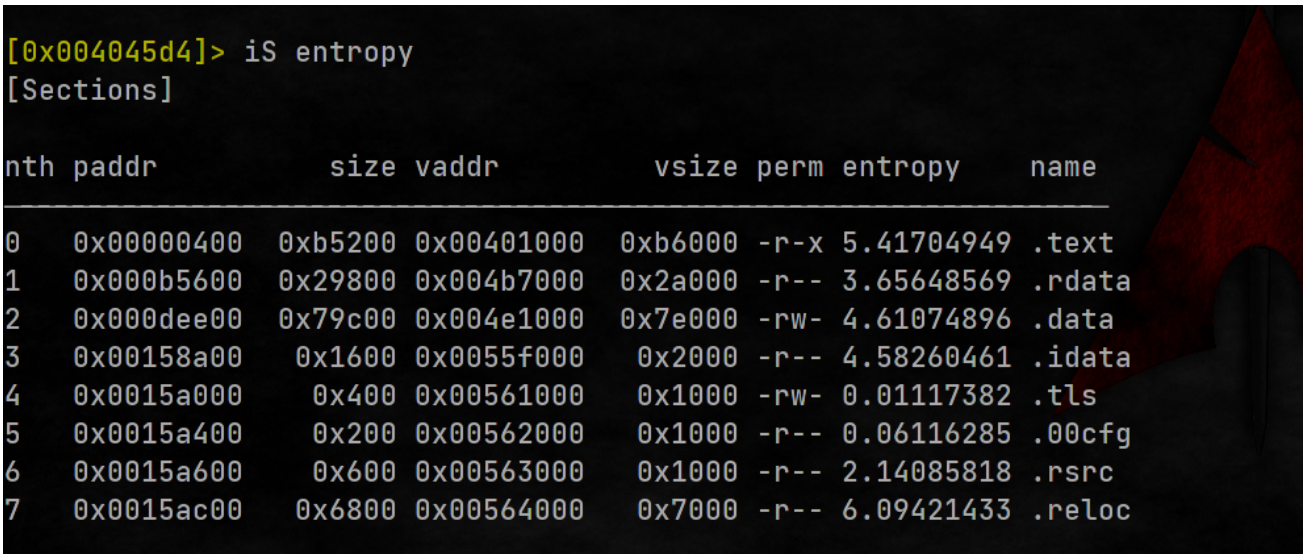

Figure 2: Rizin: Section-wise Entropy

2.3.1 The *.text*, *.rdata*, *.idata*, *.rsrc* and *.reloc* sections

These commonly found PE sections within the executable show no peculiar characteristics in terms of entropy, virtual sizes and permissions.

2.3.2 The *.data* Section

This section, although not peculiar either, on static analysis reveals that it is referenced in the identified main function. On further analysis of the function, a unpacking loop is encountered thus hinting towards the section being the store of packed data.

2.3.3 The *.tls* Section

Presence of this section generally hints towards thread execution before *entrypoint* is reached in the context of malicious binaries. This binary, however, shows no such execution. Thus, the reason for the presence of this section cannot be corroborated during the current analysis.

2.3.4 The *.00cfg* Section

The presence of this unusual section (*Control Flow Guard*) seems to be explained as an artifact of the *Visual studio compiler*. This guess is supported by

- Very small size of the section *0x200*.
- Almost all bytes being zeros.
- All the references to this section (Fig. 3) seem to originate from *Ghidra* identified library functions with exception to one which does not show much promise on followup.

Figure 3: Ghidra: references to the *.00cfg* section

2.4 A case for Packing

A very strong case for packing can be made for this binary given the following observations,

- The identified *main* function exhibits a series of byte operations on data pointed to by the *.data* section.
- Immediately preceding the manipulations, a call to *VirtualAlloc* can be intercepted.

```
i = 0;
while (i <= count + -1) {
  allocatedMem[i] = (code)~(byte)*(undefined2 *)(s_VirtualAddress + ((count + -1) -
                  /* pbstrPath != 0 && ppTypeLib != 0 */
                  /* pbstrPath != 0 && ppTypeLib != 0 */
  MessageBoxA((HWND)0x0,"","",0);
  i = i + 1;
}
j = 0;
while (j < 9600000) {
  k = 0;
  while (k < 0x400) {
    local_90 = count;
    if (count < 0x1f5) {
      local_94 = 0;
    }
    else {
      local_94 = 100;
    }
    k = k + 1;
  }
  j = j + 1;
}
l = 0;
while (l < local_90) {
  allocatedMem[l] = (code)((byte)allocatedMem[l] ^ local_70[l % local_94]);
  l = l + 1;
}
```

Figure 4: Ghidra: Disassembly of unpacking

- The manipulated bytes from *.data* section are stored in the allocated memory section.
- After the said manipulations, the memory section is called as a function.
- The said allocated section, on analysis and after being manipulated, exhibits a presence of *shell code* and a *PE* header preceding code at repeatedly reproducible offsets and sizes.

2.5 Interesting Imports

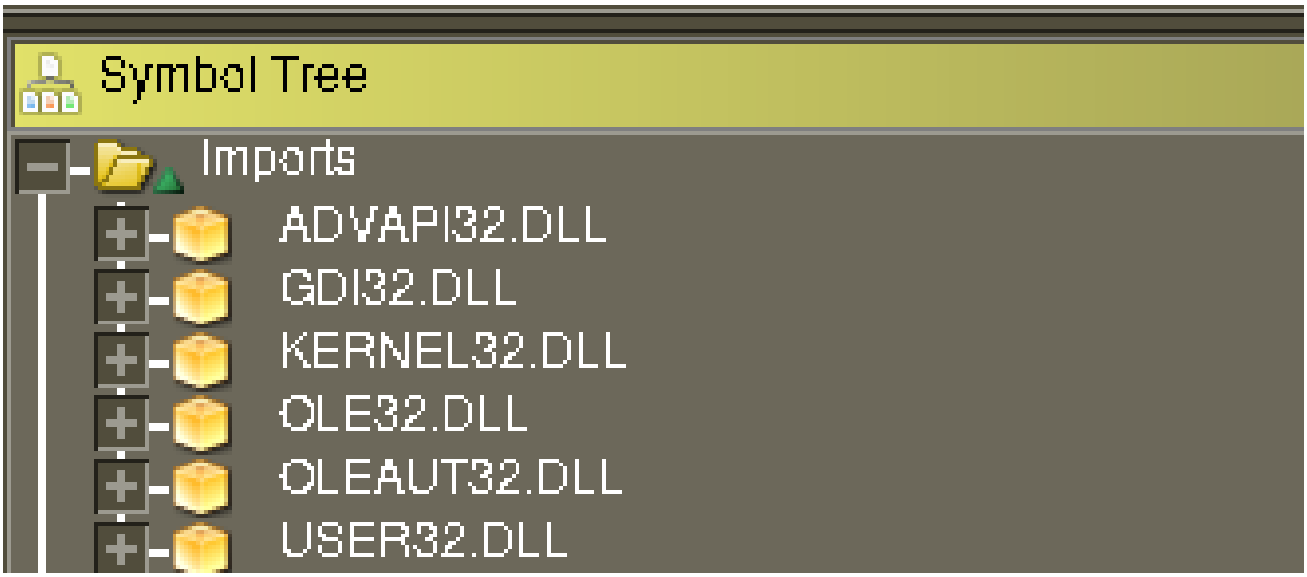

Figure 5: Ghidra: Imports tree

2.5.1 Imports from *Kernel32.dll*

Imports like *VirtualAlloc and VirtualFree* in combination with *VirtualProtect* strongly indicate runtime memory injection preceding change in injected region's permissions to *executable*. Presence of *FreeConsole* seems to corroborate the assumption that this is a *CUI* program, given this function is used to unlink from the parent process.

2.5.2 Imports from *user32.dll*

An import from this library, *viz.*, *MessageBoxA* is peculiar. This is due to the fact that, in *main* function, the permissions of memory containing code for this import is updated from *PAGE_EXECUTE_READ* to *PAGE_EXECUTE_READWRI* and is subsequently the code is replaced with a *return 0x10000* call. This function is then invoked multiple times during the unpacking process and the string *"pbstrPath != 0 && ppTypeLib != 0"* is pushed as twice arguments. The reason behind this could not be identified during this analysis (Fig 4).

3 Dynamic leading to Static Analysis: Unpacked Shell Code

3.1 Basic Identification

Attribute	Value
Bits	32
Endianess	Little
Class	Raw Binary
Size	1343
Compiler	Visual Studio (Likely)
SHA256 Hash	7203a68d0fcbde21f4005f45b14ff9ee625e16dfcf936fd82743d6bf88f76b91

3.2 Sample Family Identification

The *shell code*, on submission to *VirusTotal*, was identified as *generic malicious shell code*. Although, a comment on the submission to the *VirusTotal* from community seems to suggest this *shell code* is generated by *BC-SECURITY/Empire* [3] post exploitation framework. (Fig. 6).

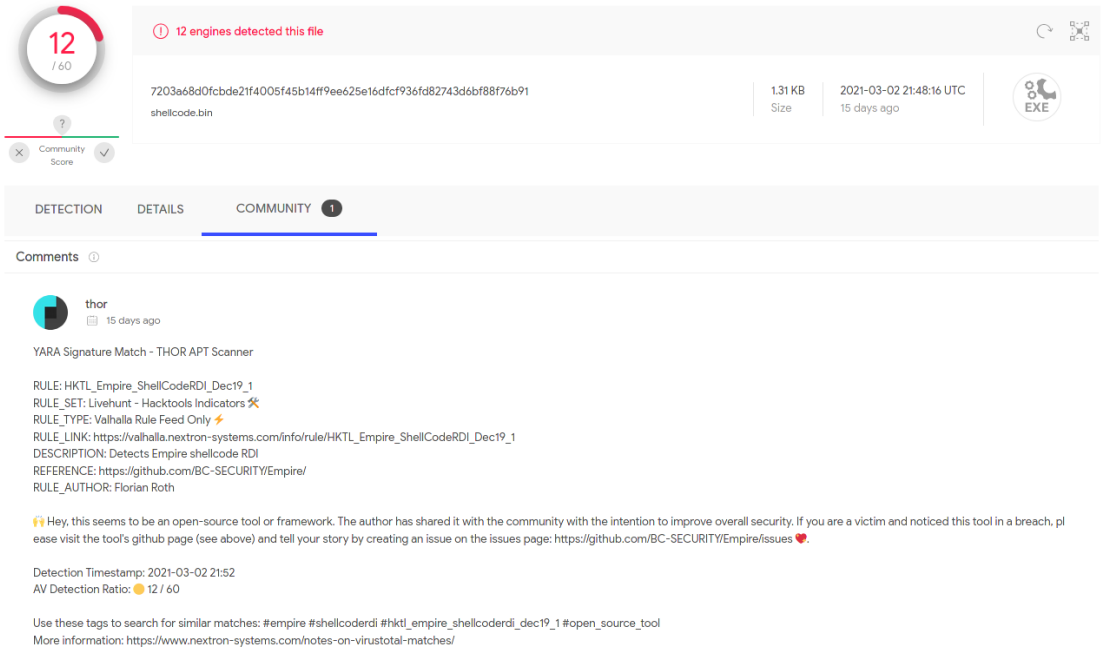

Figure 6: VirusTotal: Shell Code Family

3.3 Shell Code Sections

There are ***four*** distinctly identifiable sections of the shell code, *viz.*

- *Call/Pop* to get *EIP* (Fig 8).
- Call to *build_IAT_and_Jump* function at offset *0x2d* by pushing the pointers to the end of shell code and the *WarZone RAT* binary *Discussed later*, *EIP* and some other as of yet unidentified arguments (Fig. 9).
- The *build_IAT_and_Jump* function at offset *0x2d* which possibly allocates memory for another injection of a *DLL (Discussed later)*, resolves the imports for the *WarZone RAT binary*, executes a function from *DLL* and eventually causes an exception to execute the *WarZone RAT* OEP.
- The *getFuncNames* function at offset *0x467* which possibly un-hashes the imports for the shell code itself and is called *six* times in total.

3.4 Interesting Imports

Although the shell code does not import anything due to it being *Position Independent Code*, it certainly un-hashes ***six*** imports in particular by using possibly the previously mentioned *getFuncNames* function. These imports are,

- *LoadLibraryA*
- *GetProcAddress*
- *VirtualAlloc*
- *VirtualProtect*
- *ZwFlushInstructionCache*
- *GetNativeSystemInfo*

4 Dynamic leading to Static Analysis: Unpacked PE Executable

4.1 Basic Identification

Attribute	Value
Bits	32
Endianess	Little
Operating System	Microsoft Windows
Class	PE32
Subsystem	Windows CUI
Size	1383640
Compiler Timestamp	2020-08-29 07:01:59
Compiler	Visual Studio
SHA256 Hash	37a5c9162c834ecf877a9461e29b5adba92cbcbbe07fe56685e4f7982d1a9bc8

4.2 Sample Family Identification

This is a malicious PE extracted from the memory of primary PE after being unpacked. The submission on *VirusTotal* shows this belongs to the *WarZonRAT* [4] family of trojans. Moreover, presence of the string *warzone160* and others link it to the family to a high degree of confidence (Fig. 10).

4.3 Packing and Further Obfuscation

This dumped binary does not show signs of being obfuscated further in terms of being packed. This assumption is supported by observations,

- Presence of a multitude of imports that generally would be obfuscated.
- Presence of a multitude of ASCII strings and functions.
- Nominal range of entropy of individual sections.

4.4 Interesting Imports

4.4.1 Imports from *bcrypt.dll*

This library shows the imports *BCryptDecrypt, BCryptGenerateSymmetricKey, BCryptOpenAlgorithmProvider and BCryptSetProperty* which hint towards symmetric key generation and decryption of data. Notably, absence of an *encrypt* counterpart along with this being a *RAT* leads to suspicion that something encrypted is received over network activity which then is decrypted.

4.4.2 Import from *urlmon.dll*

An import of *URLDownloadToFile* indicates a downloader like behavior.

4.4.3 Imports from *shell32.dll*

Imports like *SHCreateDirectory, ShellExecuteA, SHGetFolderPath, etc* indicate towards filesystem manipulation behavior as well as executing some other OS command.

4.4.4 Imports from *netapi32.dll*

Imports *NetLocalGroupAddMembers and NetUserAdd* hint towards a backdoor like behavior.

4.4.5 Imports from *advapi32.dll*

Imports like *RegCreateKey, RegSetValue, RegCloseKey, AdjustTokenPrivileges and GetTokenInformation* hint towards registry action *(also later corroborated by dynamic analysis)* as well as privilege escalation.

5 Dynamic leading to Static Analysis: Unpacked DLL

5.1 Basic Identification

Attribute	Value
Bits	32
Endianess	Little
Operating System	Microsoft Windows
Class	PE32
Subsystem	Windows CUI
Size	1383640
Compiler Timestamp	2020-08-29 07:01:59
Compiler	Visual Studio
SHA256 Hash	a0e0bdb288eb7bf5585cbe101c30b892e0d5d916fa9f2a90d2059d6c8382be3e

5.2 Sample Family Identification

The extracted binary is a *DLL* linked to both *AveMariaRAT* [2] as well as *WarZoneRAT* [4], as illustrated by submission to *VirusTotal* (Fig 7).

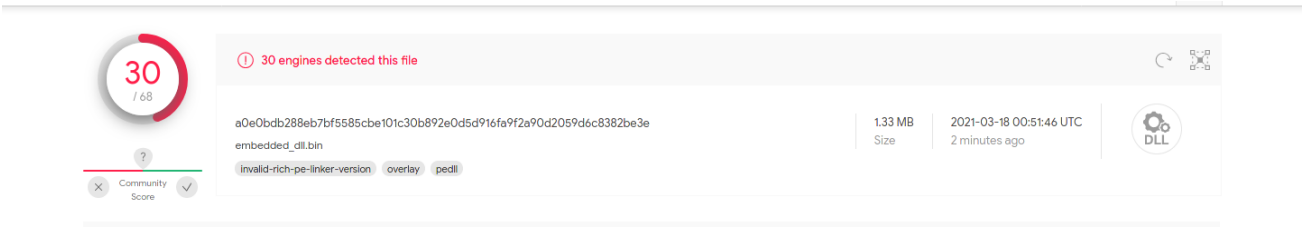

Figure 7: VirusTotal: Embedded *DLL* family

5.3 Sections

The sections *.text, .rdata, .data, .rsrc, .reloc and .bss* do not show any significant deviation from the ordinary in terms of virtual sizes, entropy as well as novelty.

5.4 Analysis

Much of the static analysis of the *DLL* is thwarted by some anti-disassembly technique since *Ghidra and Rizin/Radare2* could not corroborate with *x64_dbg* during debugging in terms of instruction alignment. Due to this, not much static analysis could be performed during the current analysis.

Although, outputs from *strings* command extracts some illuminating information nevertheless. Screenshots in appendix of the strings illustrate that the *DLL* is associated with the *WarZone* too. This is evident from the presence of the string *warzone160* as well as multiple imports that overlap with the previously analyzed binary.

6 Dynamic Analysis

6.1 Network Based Analysis

Attempted TCP connection to address *195.140.214.82:6703* (Fig. 24).

6.2 File System Based Analysis

6.2.1 File System Changes

Opened file *":Zone.Identifier"*

6.2.2 Windows Registry Changes

- Registry Key Set *"Software\Microsoft\Windows\CurrentVersion\Internet\Settings\MaxConnectionsPer1_0Server"* to *4*

- Registry Key Set *"Software\Microsoft\Windows\CurrentVersion\Internet\Settings\MaxConnectionsPerServer"* to *4*
- Registry Add (expected) *"SOFTWARE\Microsoft\Windows\NT\CurrentVersion\Winlogon\SpecialAccounts\UserList"* (Fig 26)
- Registry Add (expected) *"Software\Classes\Folder\shell\open\command"* (Fig. 27)

7 Indicators of Compromise

7.1 Network Based

Attempted TCP connection to address *195.140.214.82:6703*.

7.2 Host Based

- Registry Key Set *"Software\Microsoft\Windows\CurrentVersion\Internet\Settings\MaxConnectionsPer1_0Server"* to *4*
- Registry Key Set *"Software\Microsoft\Windows\CurrentVersion\Internet\Settings\MaxConnectionsPerServer"* to *4*
- Registry Add (expected) *"SOFTWARE\Microsoft\Windows\NT\CurrentVersion\Winlogon\SpecialAccounts\UserList"* (Fig 26)
- Registry Add (expected) *"Software\Classes\Folder\shell\open\command"* (Fig. 27)
- File Opened *":Zone.Identifier"* (Fig. 28)

7.3 *YARA* Rule

Visit [1] for rule file if copying fails.

```
rule practical2_rat {
        meta:
                description = "Detect Practical2.exe RAT"
                author = "Naman Arora"
                date = "2021-03-17"
                hash = "9633d0564a2b8f1b4c6e718ae7ab48be921d435236a403cf5e7ddfbfd4283382"
        strings:
                $pdb = "C:\\Users\\W7H64\\Desktop\\VCSamples-master\\VC2010Samples\\ATL\\General\\AtlCon\\bitcoin coinjoin op.pdb" fullword ascii
                $ops = {c6 04 0a c2 b8 01 00 00 00 c1 e0 00 8b 4d 84 c6 04 01 10 b8 01 00 00 00 d1 e0 8b 4d 84 c6 04 01 00 b8 01 00 00 00 6b c8 03 8b 55 84 c6 04 0a 90}
        condition:
                uint16(0) == 0x5a4d and filesize < 1500MB and all of them
}
```

8 Appendix A: Screenshots

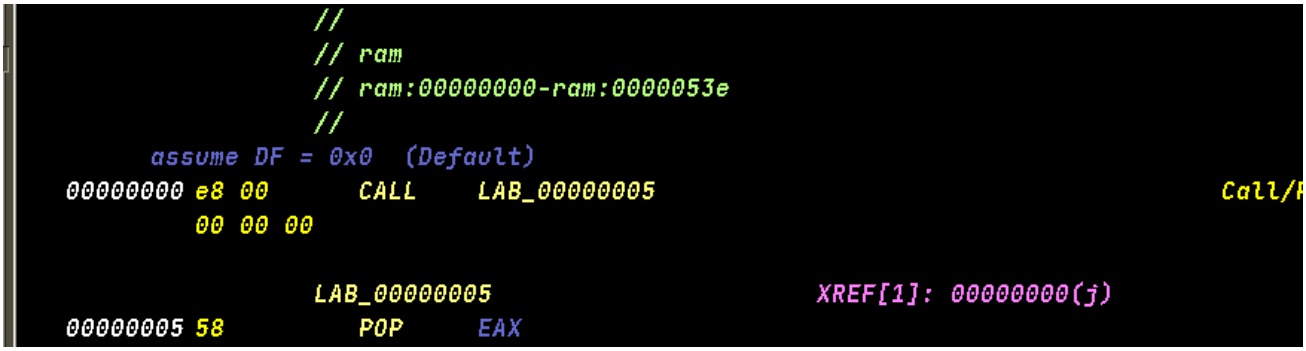

Figure 8: Ghidra: Call/Pop technique to get *EIP*

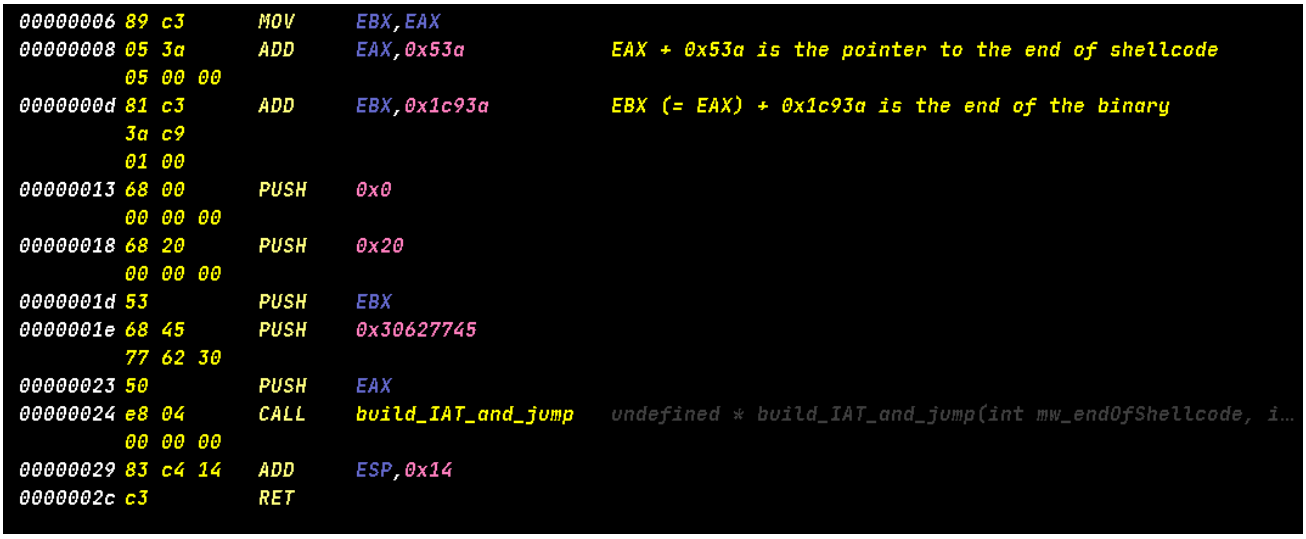

Figure 9: Ghidra: Shell Code call to *build_IAT_and_jump*

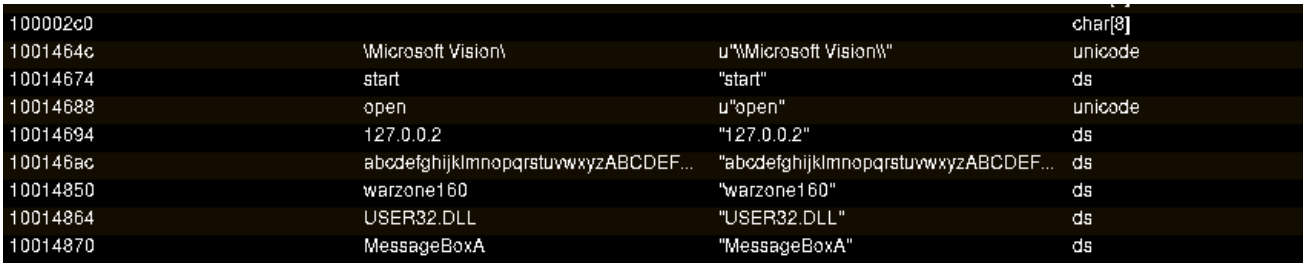

Figure 10: Ghidra: *warzone160* string

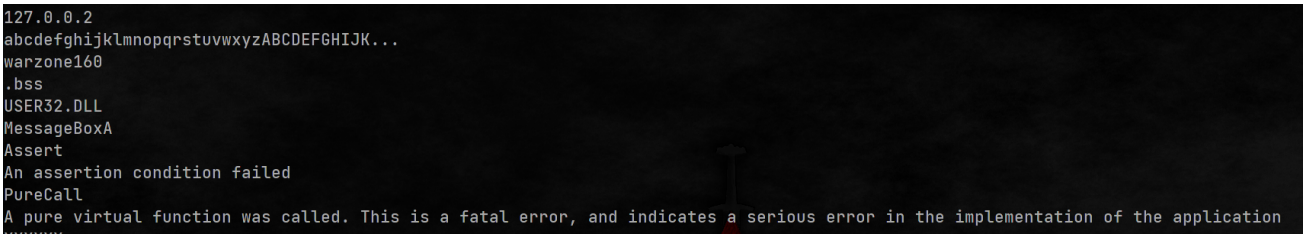

Figure 11: Strings: Embedded *DLL* strings 1

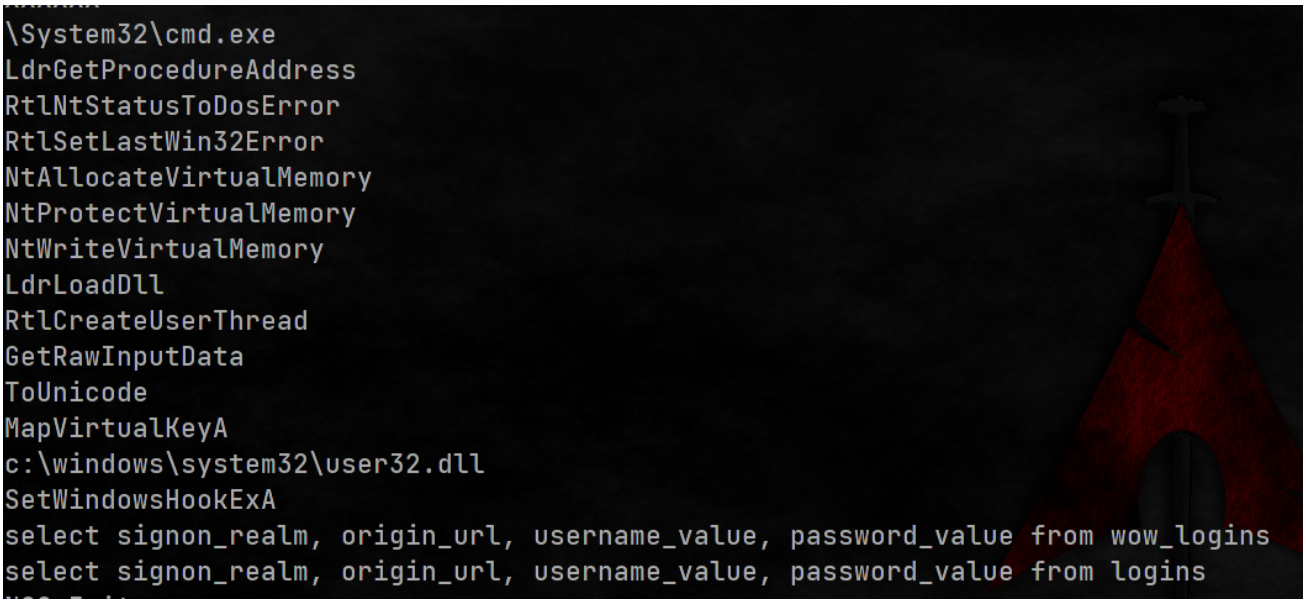

Figure 12: Strings: Embedded *DLL* strings 2

References

[1] Naman Arora. *YARA Rule for Practical2.exe RAT*. https://gist.github.com/r0ck3r008/988ea4d76cc0673db0 [Online; accessed 17-Mar-2021]. 2021.

[2] MalPedia. *AveMaria RAT*. https://malpedia.caad.fkie.fraunhofer.de/details/win.ave_maria. [Online; accessed 17-Mar-2021]. 2021.

[3] BC-SECURITY. *BC-SECURITY/Empire*. https://github.com/BC-SECURITY/Empire. [Online; accessed 17-Mar-2021]. 2021.

[4] WarZone. *WarZone RAT*. https://www.warzone.pw/. [Online; accessed 17-Mar-2021]. 2021.

```
NSS_Init
PK11_GetInternalKeySlot
PK11_Authenticate
PK11SDR_Decrypt
NSSBase64_DecodeBuffer
PK11_CheckUserPassword
NSS_Shutdown
PK11_FreeSlot
PR_GetError
vaultcli.dll
VaultOpenVault
VaultCloseVault
VaultEnumerateItems
VaultGetItem
VaultFree
encryptedUsername
hostname
encryptedPassword
```

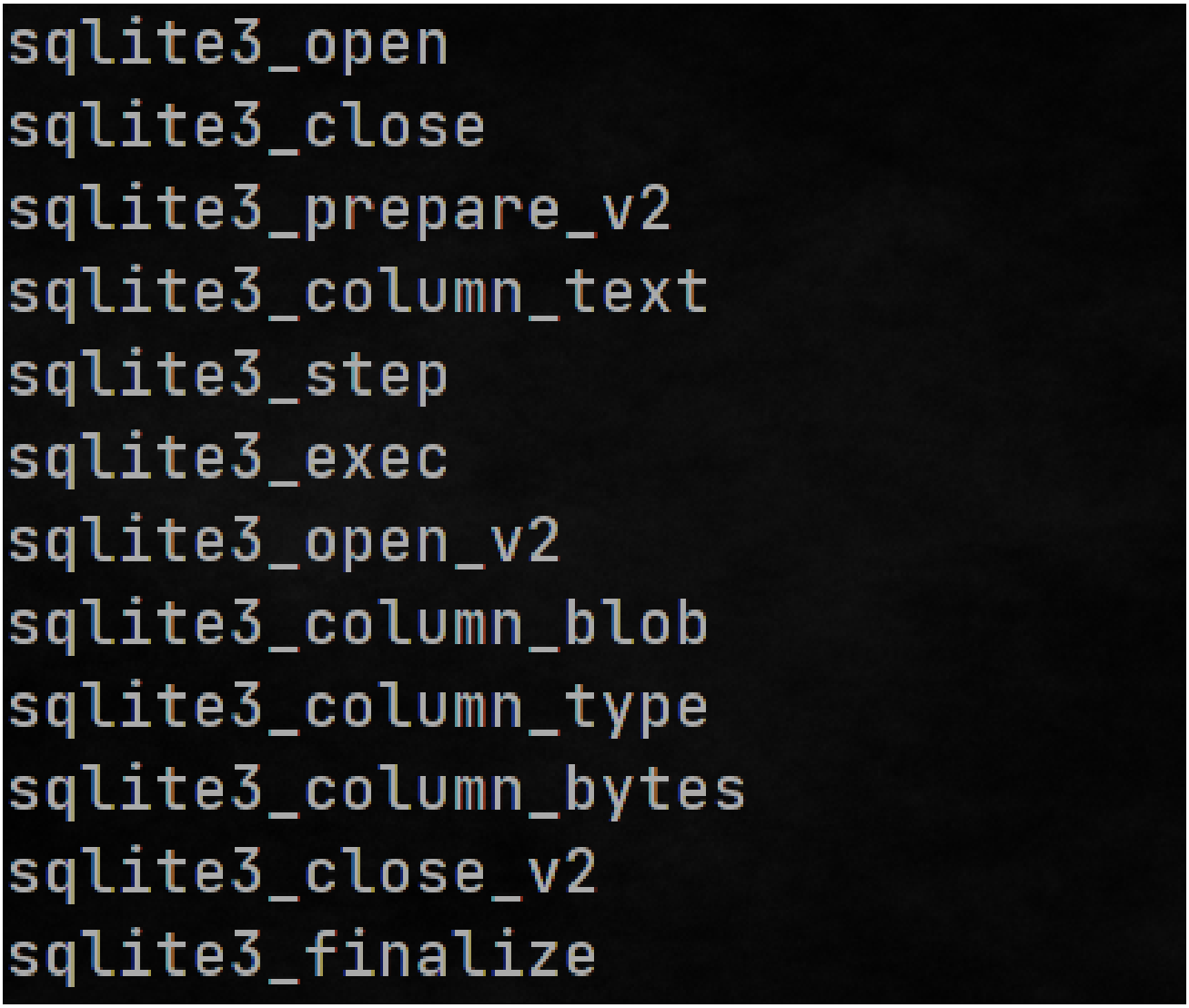

Figure 14: Strings: Embedded *DLL* strings 4

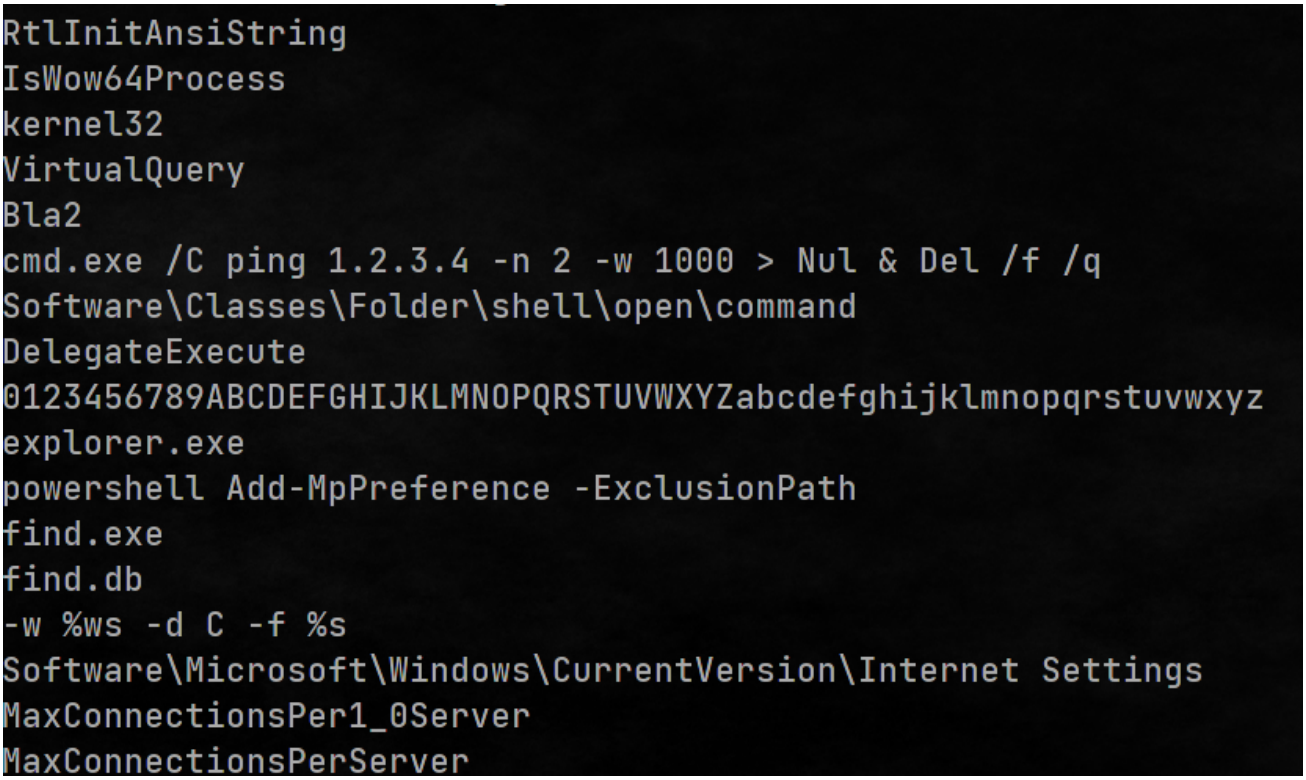

Figure 15: Strings: Embedded *DLL* strings 5

```
BCryptDecrypt
BCryptOpenAlgorithmProvider
BCryptSetProperty
BCryptGenerateSymmetricKey
bcrypt.dll
CreateDirectoryW
GetModuleFileNameA
SetLastError
VirtualFree
GetLastError
lstrcatW
CloseHandle
lstrlenW
ExpandEnvironmentStringsW
lstrlenA
lstrcmpA
lstrcatA
MultiByteToWideChar
Sleep
lstrcpyA
WideCharToMultiByte
lstrcpyW
```

```
GetCommandLineA
GetStartupInfoA
HeapFree
VirtualAlloc
HeapReAlloc
VirtualQuery
TerminateThread
CreateThread
WriteProcessMemory
GetCurrentProcess
OpenProcess
GetWindowsDirectoryA
VirtualProtectEx
VirtualAllocEx
CreateRemoteThread
CreateProcessA
GetModuleHandleW
IsWow64Process
WriteFile
CreateFileW
LoadLibraryW
GetLocalTime
GetCurrentThreadId
GetCurrentProcessId
ReadFile
FindFirstFileA
```

```
SetCurrentDirectoryW
GetFileSize
FreeLibrary
SetDllDirectoryW
GetFileSizeEx
LocalAlloc
LocalFree
WaitForSingleObject
WaitForMultipleObjects
CreatePipe
PeekNamedPipe
DuplicateHandle
SetEvent
CreateProcessW
CreateEventA
GetModuleFileNameW
LoadResource
FindResourceW
GetComputerNameW
GlobalMemoryStatusEx
LoadLibraryExW
FindFirstFileW
FindNextFileW
SetFilePointer
GetLogicalDriveStringsW
DeleteFileW
CopyFileW
GetDriveTypeW
```

```
TerminateProcess
CreateToolhelp32Snapshot
Process32NextW
Process32FirstW
SizeofResource
VirtualProtect
GetSystemDirectoryW
LockResource
GetWindowsDirectoryW
Process32First
Process32Next
WinExec
GetTempPathA
KERNEL32.dll
wsprintfW
wsprintfA
GetWindowTextW
```

DispatchMessageA
GetMessageA
GetKeyState
USER32.dll
RegQueryValueExW
RegOpenKeyExW
RegOpenKeyExA
RegEnumKeyExW
RegQueryValueExA
RegQueryInfoKeyW
RegCloseKey
OpenServiceW
ChangeServiceConfigW
QueryServiceConfigW
EnumServicesStatusExW
StartServiceW
RegSetValueExW
RegCreateKeyExA
OpenSCManagerW
CloseServiceHandle
GetTokenInformation

```
ShellExecuteW
SHGetFolderPathW
SHCreateDirectoryExW
SHGetSpecialFolderPathW
SHGetKnownFolderPath
ShellExecuteExW
ShellExecuteExA
SHELL32.dll
URLDownloadToFileW
urlmon.dll
getaddrinfo
freeaddrinfo
InetNtopW
WS2_32.dll
CoInitialize
CoCreateInstance
CoInitializeSecurity
CoUninitialize
CoTaskMemFree
ole32.dll
PathFindExtensionW
PathFindFileNameW
```

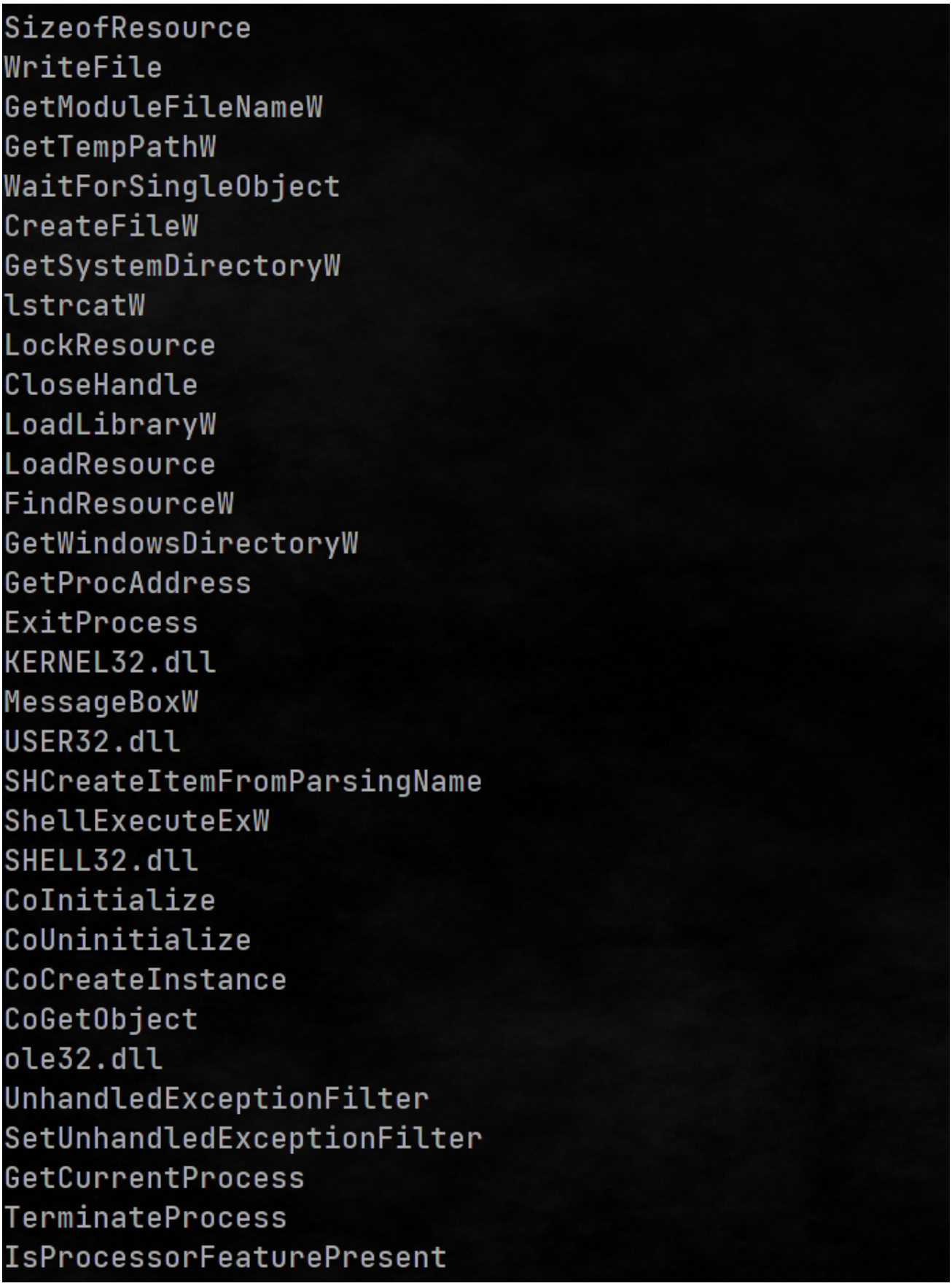

Figure 22: Strings: Embedded *DLL* strings 12

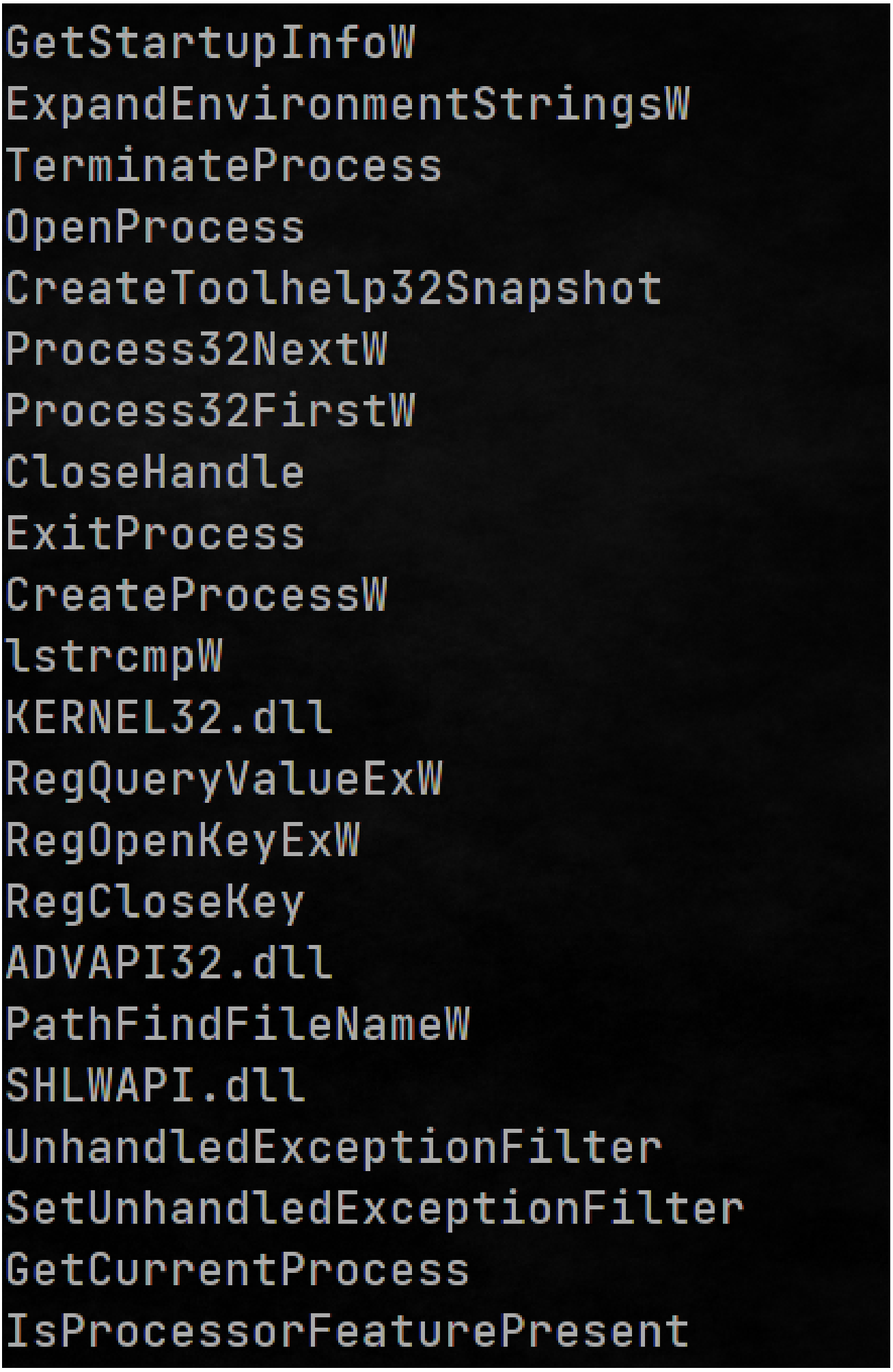

Figure 23: Strings: Embedded *DLL* strings 13

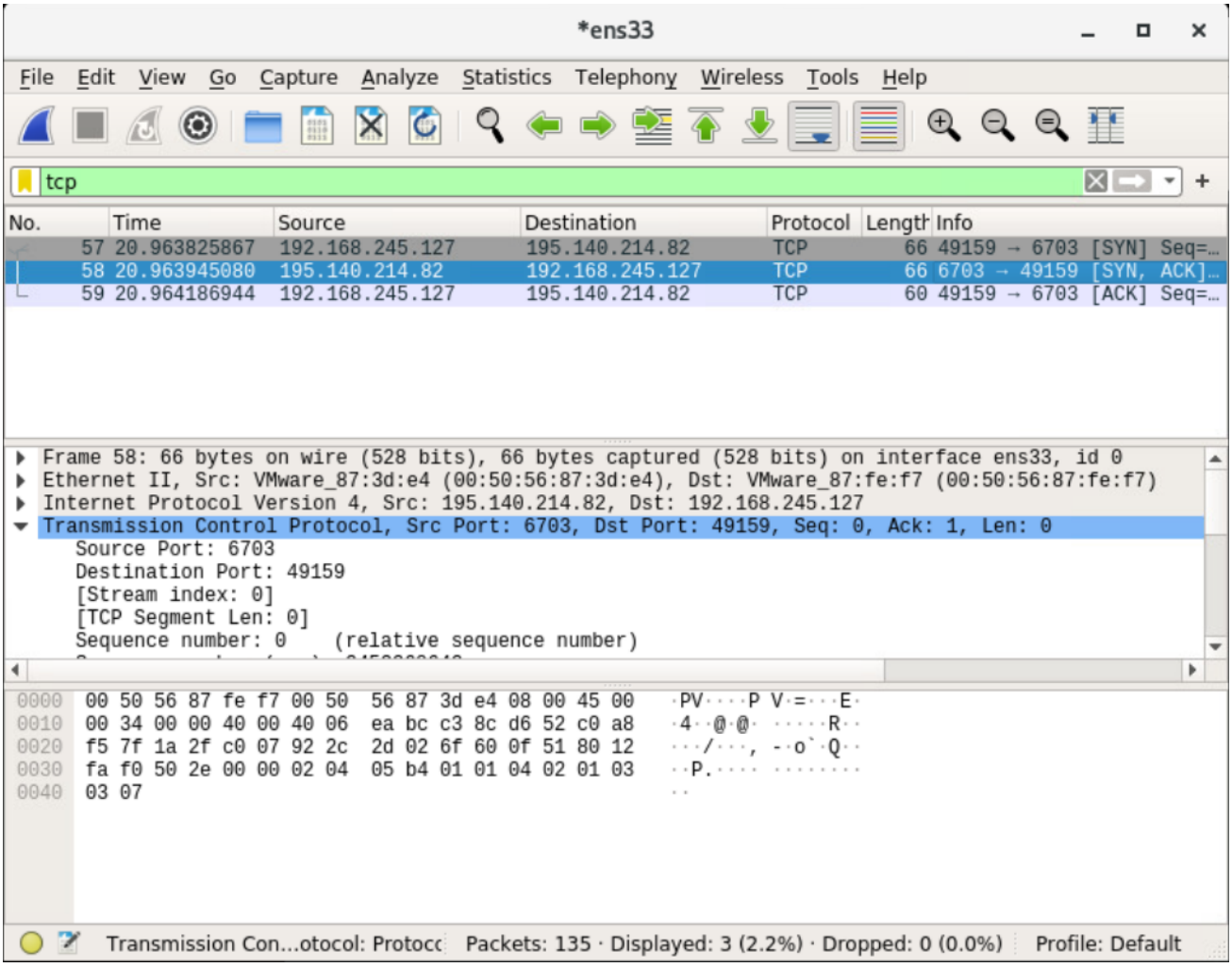

Figure 24: Wireshark: Connection to IP *195.140.214.82* at port 6703

```
RegCreateKeyExA((HKEY)0x80000001,"Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings"
                ,0,(LPSTR)0x0,0,0xf003f,(LPSECURITY_ATTRIBUTES)0x0,(PHKEY)&local_650,&local_63c);
RegSetValueExA(local_650,"MaxConnectionsPer1_0Server",0,4,(BYTE *)&local_644,4);
RegSetValueExA(local_650,"MaxConnectionsPerServer",0,4,(BYTE *)&local_644,4);
```

Figure 25: Ghidra: Registry Entry 1

```
uVar9 = 0;
RegCreateKeyExA((HKEY)0x80000002,

                "SOFTWARE\\Microsoft\\Windows
                NT\\CurrentVersion\\Winlogon\\SpecialAccounts\\UserList"
                ,0,(LPSTR)0x0,0,0xf013f,(LPSECURITY_ATTRIBUTES)0x0,(PHKEY)local_c,&local_14);
local_10 = 0;
```

Figure 26: Ghidra: Registry Entry 2

```
dwErrCode = RegCreateKeyExA((HKEY)0x80000001,"Software\\Classes\\Folder\\shell\\open\\command",0
                            ,(LPSTR)0x0,0,0x20006,(LPSECURITY_ATTRIBUTES)&local_18,
                            (PHKEY)&local_8,&local_c);
if (dwErrCode == 0) {
```

Figure 27: Ghidra: Registry Entry 3

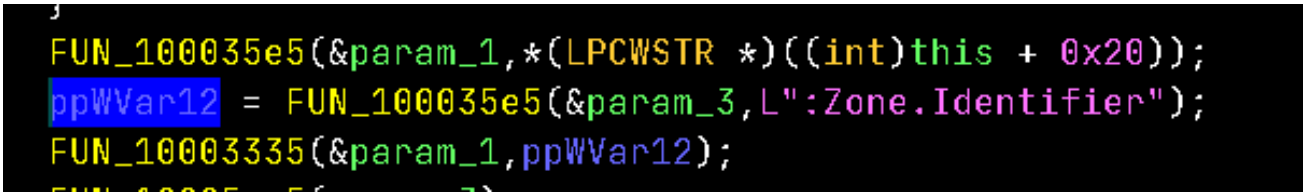

Figure 28: Ghidra: *":Zone.Identifier"*