Under the SADBRIDGE with GOSAR: QUASAR Gets a Golang Rewrite

Elastic Security Labs share details about the SADBRIDGE loader and GOSAR backdoor, malware used in campaigns targeting Chinese-speaking victims.

30 min read Malware analysis

Under the SADBRIDGE with GOSAR: QUASAR Gets a Golang Rewrite

Introduction

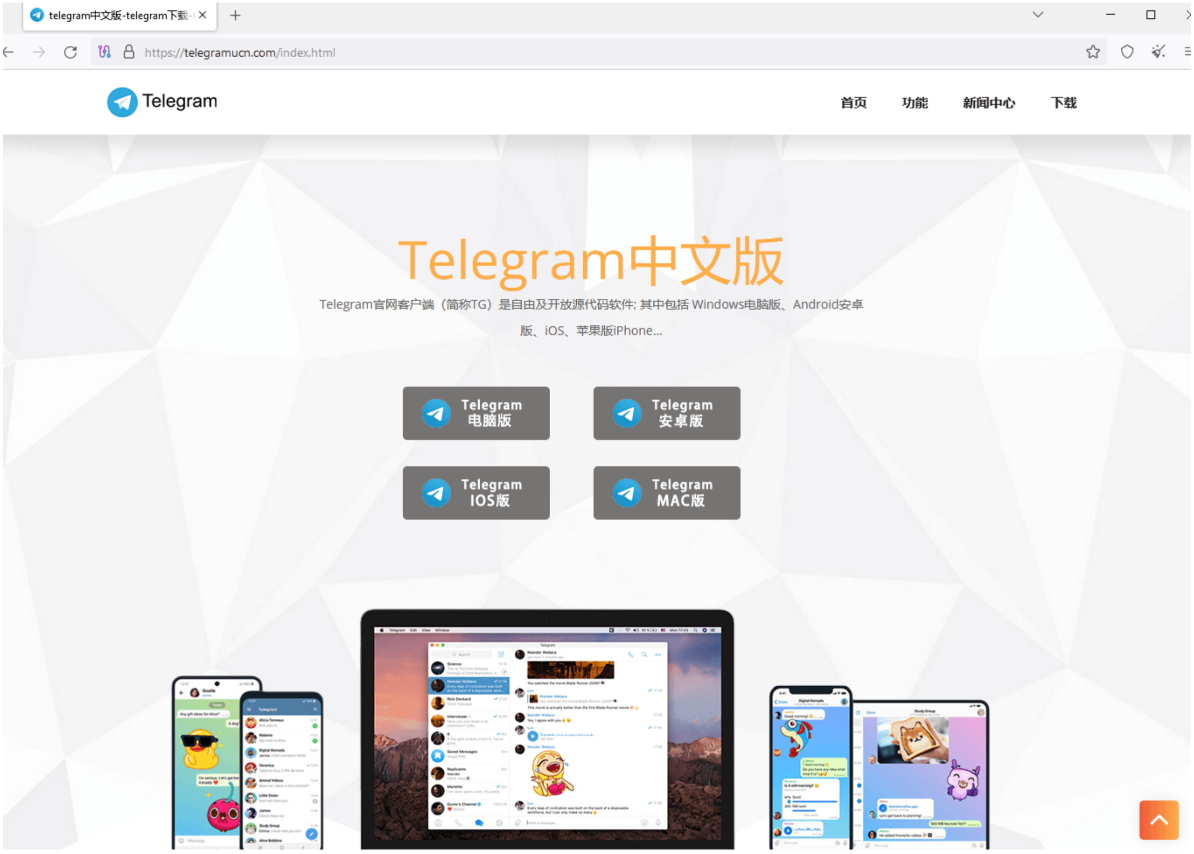
Elastic Security Labs recently observed a new intrusion set targeting Chinese-speaking regions, tracked as REF3864. These organized campaigns target victims by masquerading as legitimate software such as web browsers or social media messaging services. The threat group behind these campaigns shows a moderate degree of versatility in delivering malware across multiple platforms such as Linux, Windows, and Android. During this investigation, our team discovered a unique Windows infection chain with a custom loader we call SADBRIDGE. This loader deploys a Golang-based reimplementation of QUASAR, which we refer to as GOSAR. This is our team's first time observing a rewrite of QUASAR in the Golang programming language.

Key takeaways

- Ongoing campaigns targeting Chinese language speakers with malicious installers masquerading as legitimate software like Telegram and the Opera web browser
- Infection chains employ injection and DLL side-loading using a custom loader (SADBRIDGE)
- SADBRIDGE deploys a newly-discovered variant of the QUASAR backdoor written in Golang (GOSAR)
- GOSAR is a multi-functional backdoor under active development with incomplete features and iterations of improved features observed over time
- Elastic Security provides comprehensive prevention and detection capabilities against this attack chain

REF3864 Campaign Overview

In November, the Elastic Security Labs team observed a unique infection chain when detonating several different samples uploaded to VirusTotal. These different samples were hosted via landing pages masquerading as legitimate software such as Telegram or the Opera GX browser.



Fake Telegram landing page

During this investigation, we uncovered multiple infection chains involving similar techniques:

- Trojanized MSI installers with low detections
- Masquerading using legitimate software bundled with malicious DLLs
- Custom SADBRIDGE loader deployed
- Final stage GOSAR loaded

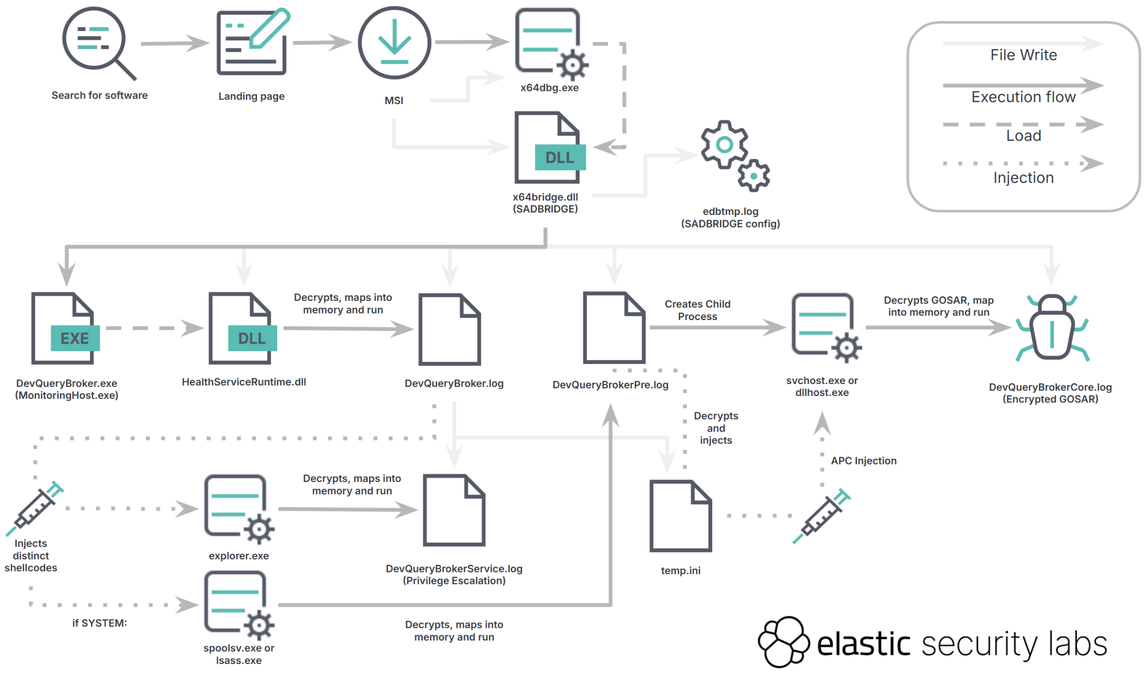
We believe these campaigns have flown under the radar due to multiple levels of abstraction. Typically, the first phase involves opening an archive file (ZIP) that includes an MSI installer. Legitimate software like the Windows `x64dbg.exe` debugging application is used behind-the-scenes to load a malicious, patched DLL (`x64bridge.dll`). This DLL kicks off a new legitimate program (`MonitoringHost.exe`) where it side-loads another malicious DLL (`HealthServiceRuntime.dll`), ultimately performing injection and loading the GOSAR implant in memory via injection.

Malware researchers extracted SADBRIDGE configurations that reveal adversary-designated campaign dates, and indicate operations with similar TTP's have been ongoing since at least December 2023. The command-and-control (C2) infrastructure for GOSAR often masquerades under trusted services or software to appear benign and conform to victim expectations for software installers. Throughout the execution chain, there is a focus centered around enumerating Chinese AV products such as `360tray.exe`, along with firewall rule names and descriptions in Chinese. Due to these customizations we believe this threat is geared towards targeting Chinese language speakers. Additionally, extensive usage of Chinese language logging indicates the attackers are also Chinese language speakers.

QUASAR has previously been used in state-sponsored espionage, non-state hacktivism, and criminal financially motivated attacks since 2017 (Qualys, Evolution of Quasar RAT), including by China-linked APT10. A rewrite in

Golang might capitalize on institutional knowledge gained over this period, allowing for additional capabilities without extensive retraining of previously effective TTPs.

GOSAR extends QUASAR with additional information-gathering capabilities, multi-OS support, and improved evasion against anti-virus products and malware classifiers. However, the generic lure websites, and lack of additional targeting information, or actions on the objective, leave us with insufficient evidence to identify attacker motivation(s).



SADBRIDGE Execution Chain resulting in GOSAR infection

SADBRIDGE Introduction

The SADBRIDGE malware loader is packaged as an MSI executable for delivery and uses DLL side-loading with various injection techniques to execute malicious payloads. SADBRIDGE abuses legitimate applications such as `x64dbg.exe` and `MonitoringHost.exe` to load malicious DLLs like `x64bridge.dll` and `HealthServiceRuntime.dll`, which leads to subsequent stages and shellcodes.

Persistence is achieved through service creation and registry modifications. Privilege escalation to Administrator occurs silently using a UAC bypass technique that abuses the `ICMLuaUtil` COM interface. In addition, SADBRIDGE incorporates a privilege escalation bypass through Windows Task Scheduler to execute its main payload with SYSTEM level privileges.

The SADBRIDGE configuration is encrypted using a simple subtraction of `0x1` on each byte of the configuration string. The encrypted stages are all appended with a `.log` extension, and decrypted during runtime using XOR and the LZNT1 decompression algorithm.

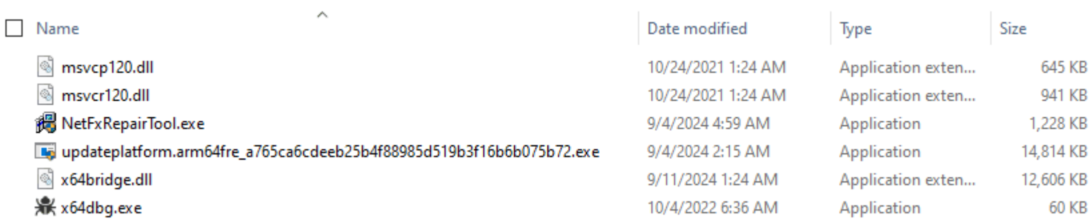
SADBRIDGE employs PoolParty, APC queues, and token manipulation techniques for process injection. To avoid sandbox analysis, it uses long `Sleep` API calls. Another defense evasion technique involves API patching to disable Windows security mechanisms such as the Antimalware Scan Interface (AMSI) and Event Tracing for Windows (ETW).

The following deep dive is structured to explore the execution chain, providing a step-by-step walkthrough of the capabilities and functionalities of significant files and stages, based on the configuration of the analyzed sample. The analysis aims to highlight the interaction between each component and their roles in reaching the final payload.

SADBRIDGE Code Analysis

MSI Analysis

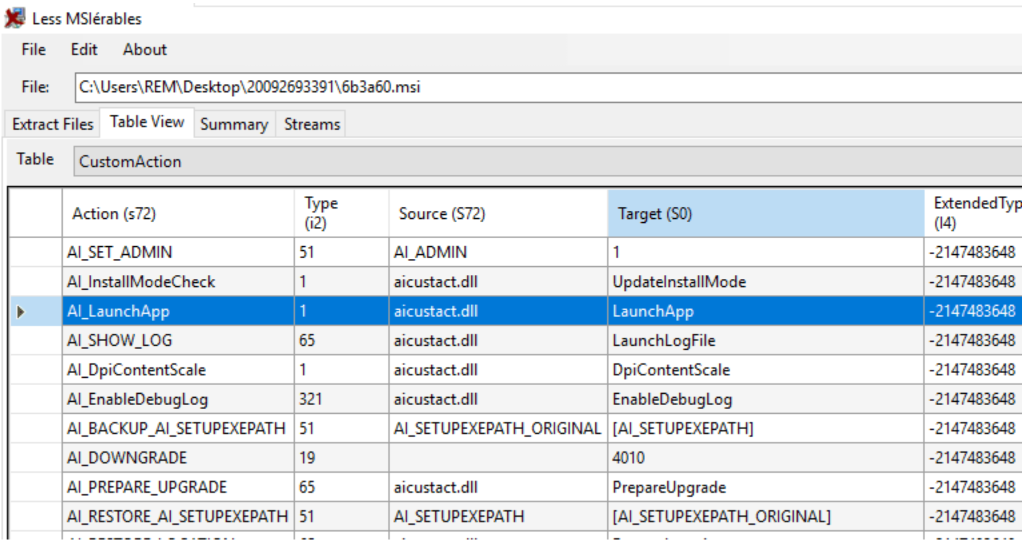
The initial files are packaged in an MSI using Advanced Installer, the main files of interest are `x64dbg.exe` and `x64bridge.dll`.



Name	Date modified	Type	Size
msvcp120.dll	10/24/2021 1:24 AM	Application exten...	645 KB
msvcr120.dll	10/24/2021 1:24 AM	Application exten...	941 KB
NetFxRepairTool.exe	9/4/2024 4:59 AM	Application	1,228 KB
updateplatform.arm64fre_a765ca6cdeeb25b4f88985d519b3f16b6b075b72.exe	9/4/2024 2:15 AM	Application	14,814 KB
x64bridge.dll	9/11/2024 1:24 AM	Application exten...	12,606 KB
x64dbg.exe	10/4/2022 6:36 AM	Application	60 KB

Significant files inside the MSI installer

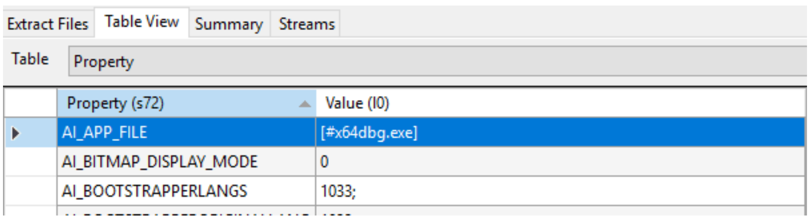
By using MSI tooling (lessmsi), we can see the `LaunchApp` entrypoint in `aicustact.dll` is configured to execute the file path specified in the `AI_APP_FILE` property.



Action (s72)	Type (i2)	Source (S72)	Target (S0)	ExtendedTyp (I4)
AI_SET_ADMIN	51	AI_ADMIN	1	-2147483648
AI_InstallModeCheck	1	aicustact.dll	UpdateInstallMode	-2147483648
AI_LaunchApp	1	aicustact.dll	LaunchApp	-2147483648
AI_SHOW_LOG	65	aicustact.dll	LaunchLogFile	-2147483648
AI_DpiContentScale	1	aicustact.dll	DpiContentScale	-2147483648
AI_EnableDebugLog	321	aicustact.dll	EnableDebugLog	-2147483648
AI_BACKUP_AI_SETUPEXEPATH	51	AI_SETUPEXEPATH_ORIGINAL	[AI_SETUPEXEPATH]	-2147483648
AI_DOWNGRADE	19		4010	-2147483648
AI_PREPARE_UPGRADE	65	aicustact.dll	PrepareUpgrade	-2147483648
AI_RESTORE_AI_SETUPEXEPATH	51	AI_SETUPEXEPATH	[AI_SETUPEXEPATH_ORIGINAL]	-2147483648

Custom actions configured using Advanced Installer

If we navigate to this `AI_APP_FILE` property, we can see the file tied to this configuration is `x64dbg.exe`. This represents the file that will be executed after the installation is completed, the legitimate `NetFxRepairTool.exe` is never executed.

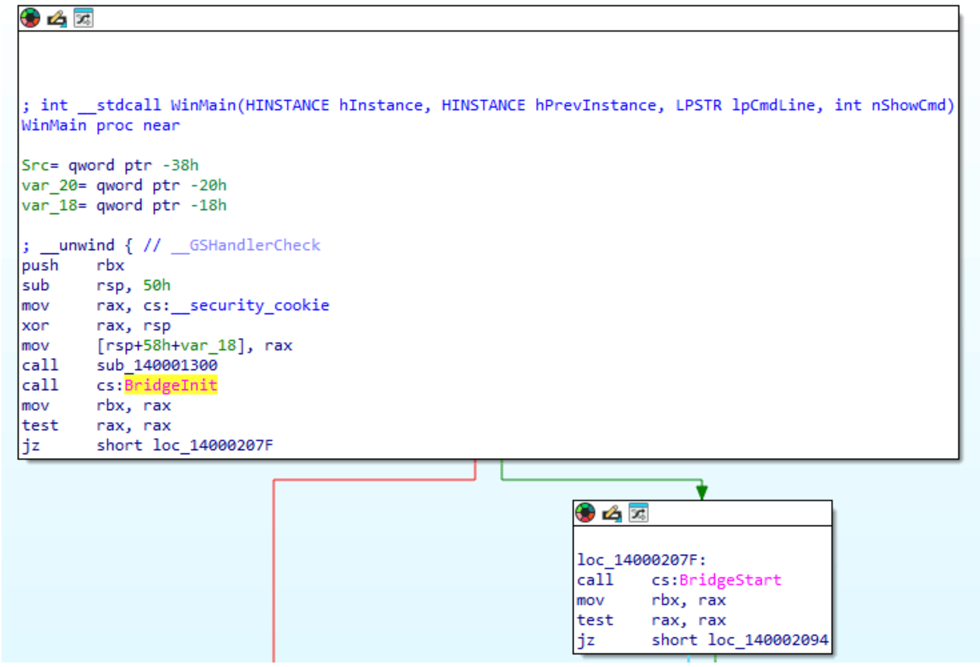


Property (s72)	Value (l0)
AI_APP_FILE	[#x64dbg.exe]
AI_BITMAP_DISPLAY_MODE	0
AI_BOOTSTRAPPERLANGS	1033;

AI_APP_FILE property configured to launch x64dbg.exe

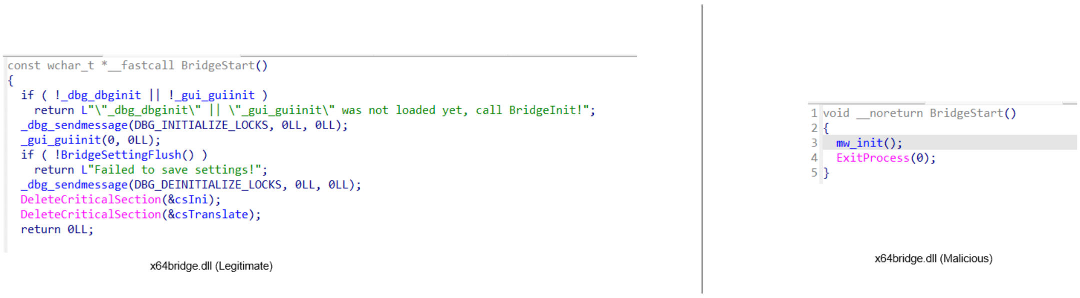
x64bridge.dll Side-loading

When `x64dbg.exe` gets executed, it calls the `BridgeInit` export from `x64bridge.dll`. `BridgeInit` is a wrapper for the `BridgeStart` function.



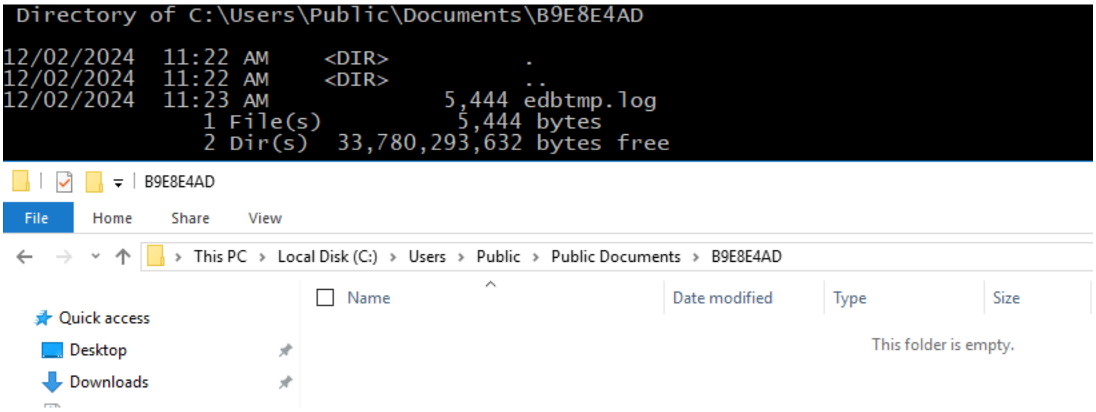
Control flow diagram showing call to BridgeStart

Similar to techniques observed with BLISTER, SADBRIDGE patches the export of a legitimate DLL.



Comparison of BridgeStart export from x64bridge.dll

During the malware initialization routine, SADBRIDGE begins with generating a hash using the hostname and a magic seed `0x4E67C6A7`. This hash is used as a directory name for storing the encrypted configuration file. The encrypted configuration is written to `C:\Users\Public\Documents\<hostname_hash>\edbtmp.log`. This file contains the attributes FILE_ATTRIBUTE_SYSTEM, FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_HIDDEN to hide itself from an ordinary directory listing.



Configuration file hidden from users

Decrypting the configuration is straightforward, the encrypted chunks are separated with null bytes. For each byte within the encrypted chunks, we can increment them by `0x1`.

The configuration consists of:

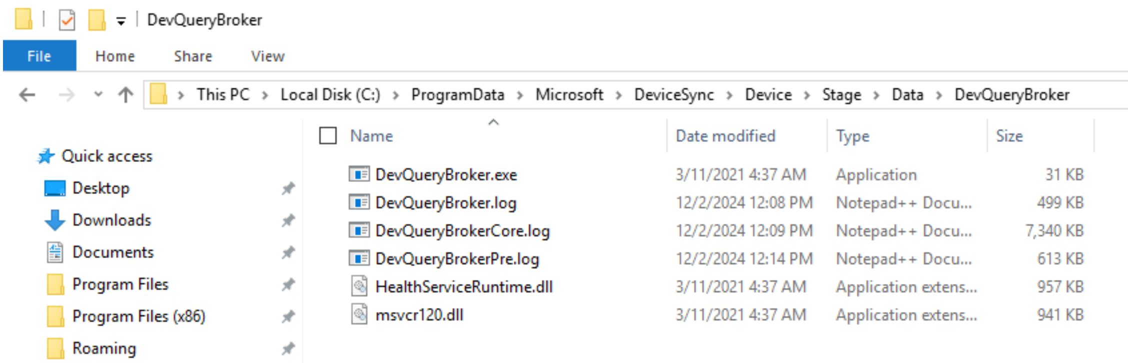
- Possible campaign date
- Strings to be used for creating services
- New name for MonitoringHost.exe (`DevQueryBroker.exe`)
- DLL name for the DLL to be sideloaded by MonitoringHost.exe (`HealthServiceRuntime.dll`)

- Absolute paths for additional stages (`.log` files)
- The primary injection target for hosting GOSAR (`svchost.exe`)

```
20240910
DevQueryBrokerService
DevQuery Background Discovery Broker Service
Enables apps to discover devices with a background task.
C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker
DevQueryBroker.exe
HealthServiceRuntime.dll
C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\DevQueryBroker.log
C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\DevQueryBrokerCore.log
C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\DevQueryBrokerService.log
20240911002404
msvcr120.dll
C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\DevQueryBrokerPre.log
svchost.exe
```

SADBRIDGE configuration

The `DevQueryBroker` directory (`C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\`) contains all of the encrypted stages (`.log` files) that are decrypted at runtime. The file (`DevQueryBroker.exe`) is a renamed copy of Microsoft legitimate application (`MonitoringHost.exe`).



File listing of the DevQueryBroker folder

Finally, it creates a process to run `DevQueryBroker.exe` which side-loads the malicious `HealthServiceRuntime.dll` in the same folder.

HealthServiceRuntime.dll

This module drops both an encrypted and partially decrypted shellcode in the User's `%TEMP%` directory. The file name for the shellcode follows the format: `log<random_string>.tmp`. Each byte of the partially decrypted shellcode is then decremented by `0x10` to fully decrypt. The shellcode is executed in a new thread of the same process.

```
p_PartiallyDecryptedShellcode = 0LL;
Size = 0;
mw_DropEncryptedShellcode(&p_PartiallyDecryptedShellcode, &Size);
v19 = 0;
for ( j = 0; j != 100000000; ++j )
{
  v6 = j + v19;
  v19 = v6;
}
v7 = Size;
v8 = (char *)p_PartiallyDecryptedShellcode;
if ( Size )
{
  v9 = (char *)p_PartiallyDecryptedShellcode + Size;
  do
    *v8++ -= 0x10;
  while ( v9 != v8 );
  v7 = Size;
}
```

Decryption of a shellcode in HealthServiceRuntime.dll

The malware leverages API hashing using the same algorithm in research published by SonicWall, the hashing algorithm is listed in the Appendix section. The shellcode decrypts `DevQueryBroker.log` into a PE file then performs a simple XOR operation with a single byte (`0x42)` in the first third of the file where then it decompresses the result using the LZNT1 algorithm.

```
partial_file_size = file_size / 3;
if ( file_size / 3 )
{
  _dev_query_broker_log = dev_query_broker_log;
  do
  {
    j = 0;
    if ( i != 1 )
      j = i;
    i = j + 1;
    *_dev_query_broker_log++ ^= *(&p_final_uncompressed_size + 2 * j);
    --partial_file_size;
  }
  while ( partial_file_size );
}

p_final_uncompressed_size = 0;
VirtualAlloc = mw::ResolveAPIHashes(kernel32_dll_VirtualAlloc);
p_decompressed = VirtualAlloc(0LL, 4 * file_size, 0x3000LL, 4LL);
RtlDecompressBuffer = mw::ResolveAPIHashes(ntdll_dll_RtlDecompressBuffer);
RtlDecompressBuffer(2LL, p_decompressed, 4 * file_size, dev_query_broker_log, file_size, &p_final_uncompressed_size)
*_p_final_uncompressed_size = p_final_uncompressed_size;
*_p_mem2 = p_decompressed;
VirtualFree = mw::ResolveAPIHashes(kernel32_dll_VirtualFree);
VirtualFree(dev_query_broker_log, 0LL, 0x8000LL);
return 1LL;
```

Shellcode decrypting DevQueryBroker.log file

The shellcode then unmaps any existing mappings at the PE file's preferred base address using `NtUnmapViewOfSection`, ensuring that a call to `VirtualAlloc` will allocate memory starting at the preferred base address. Finally, it maps the decrypted PE file to this allocated memory and transfers execution to its entry point. All shellcodes identified and executed by SADBRIDGE share an identical code structure, differing only in the specific `.log` files they reference for decryption and execution.

DevQueryBroker.log

The malware dynamically loads `amsi.dll` to disable critical security mechanisms in Windows. It patches `AmsiScanBuffer` in `amsi.dll` by inserting instructions to modify the return value to `0x80070057`, the standardized Microsoft error code `E_INVALIDARG` indicating invalid arguments, and returning prematurely, to effectively bypass the scanning logic. Similarly, it patches `AmsiOpenSession` to always return the same error code `E_INVALIDARG`. Additionally, it patches `EtwEventWrite` in `ntdll.dll`, replacing the first instruction with a `ret` instruction to disable Event Tracing for Windows (ETW), suppressing any logging of malicious activity.

```
  CurrentProcess = GetCurrentProcess();
  LibraryA = LoadLibraryA("amsi.dll");
  if ( GetProcAddress(LibraryA, "AmsiScanBuffer") )
  {
    mw_WrapperNtProtectVirtualMemory();
    mw_WrapperNtWriteVirtualMemory();
    mw_WrapperNtProtectVirtualMemory();
  }
  v2 = LoadLibraryA("amsi.dll");
  if ( GetProcAddress(v2, "AmsiOpenSession") )
  {
    mw_WrapperNtProtectVirtualMemory();
    mw_WrapperNtWriteVirtualMemory();
    mw_WrapperNtProtectVirtualMemory();
  }
  ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
  if ( GetProcAddress(ModuleHandleW, "EtwEventWrite") )
  {
    mw_WrapperNtProtectVirtualMemory();
    mw_WrapperNtWriteVirtualMemory();
    mw_WrapperNtProtectVirtualMemory();
  }
  return CloseHandle(CurrentProcess);
}
```

Patching AmsiScanBuffer, AmsiOpenSession and EtwEventWrite APIs

Following the patching, an encrypted shellcode is written to `temp.ini` at path (`C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\temp.ini`).
The malware checks the current process token's group membership to determine its privilege level. It verifies if the process belongs to the LocalSystem account by initializing a SID with the `SECURITY_LOCAL_SYSTEM_RID` and calling `CheckTokenMembership`. If not, it attempts to check for membership in the Administrators group by creating a SID using `SECURITY_BUILTIN_DOMAIN_RID` and `DOMAIN_ALIAS_RID_ADMINS` and performing a similar token membership check.

If the current process does not have LocalSystem or Administrator privileges, privileges are first elevated to Administrator through a UAC bypass mechanism by leveraging the `ICMLuaUtil` COM interface. It crafts a moniker string `"Elevation:Administrator!new:{3E5FC7F9-9A51-4367-9063-A120244FBEC7}"` to create an instance of the `CMSTPLUA` object with Administrator privileges. Once the object is created and the `ICMLuaUtil` interface is obtained, the malware uses the exposed `ShellExec` method of the interface to run `DevQueryBroker.exe`.

```
if ( !SIDCheck )                                  // If process is not running under LocalSystem or Administrator
{
  sub_140003380();
  v59 = CoInitializeEx(0LL, 2u);
  wcscpy(&cmstplua_obj_moniker[1], L"Elevation:Administrator!new:");
  pICMLuaUtil = 0LL;
  memset(&pBindOptions, 0, sizeof(pBindOptions));
  v60 = cmstplua_obj_moniker;
  pBindOptions.cbStruct = 0x30;
  pBindOptions.dwClassContext = 4;
  do
  {
    v6 = v60[1] == 0;
    ++v60;
  }
  while ( !v6 );
  *(_OWORD *)v60 = *(_OWORD *)xmmword_140038960;// CLSID = {3E5FC7F9-9A51-4367-9063367-9063FBEC7}
  *((_OWORD *)v60 + 1) = xmmword_140038970;
  *((_OWORD *)v60 + 2) = xmmword_140038980;
  *((_OWORD *)v60 + 3) = xmmword_140038990;
  *((_QWORD *)v60 + 8) = 0x43004500420046LL;
  *((_DWORD *)v60 + 18) = 0x7D0037;
  v60[38] = 0;
  Object = CoGetObject(&cmstplua_obj_moniker[1], (BIND_OPTS *)&pBindOptions, &icmluautil_iid, (void **)&pICMLuaUtil);
  pICMLuaUtil_2 = pICMLuaUtil;
  if ( Object )
  {
    if ( !pICMLuaUtil )
      goto LABEL_83;
  }
  else
  {
    if ( !pICMLuaUtil )
      goto LABEL_83;
    ((void (__fastcall *)(CCMLuaUtil *, WCHAR *, _QWORD, _QWORD, _DWORD, MACRO_SW))pICMLuaUtil->vtable->ShellExec)(
      pICMLuaUtil,
      str_pathToDevQueryBrokerEXE,
      0LL,
      0LL,
      0,
      SW_SHOW);
  }
  ((void (__fastcall *)(CCMLuaUtil *))pICMLuaUtil_2->vtable->Release)(pICMLuaUtil_2);
```

Privilege Escalation via ICMLuaUtil COM interface

If a task or a service is not created to run `DevQueryBroker.exe` routinely, the malware checks if the Anti-Virus process `360tray.exe` is running. If it is not running, a service is created for privilege escalation to SYSTEM, with the following properties:

- Service name: **DevQueryBrokerService**
  Binary path name:
  "C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\DevQueryBroker.exe -svc".
- Display name: **DevQuery Background Discovery Broker Service**
- Description: **Enables apps to discover devices with a background task.**
- Start type: **Automatically at system boot**
- Privileges: **LocalSystem**

If `360tray.exe` is detected running, the malware writes an encrypted PE file to `DevQueryBrokerService.log`, then maps a next-stage PE file (Stage 1) into the current process memory, transferring execution to it.

Once `DevQueryBroker.exe` is re-triggered with SYSTEM level privileges and reaches this part of the chain, the malware checks the Windows version. For systems running Vista or later (excluding Windows 7), it maps another next-stage (Stage 2) into memory and transfers execution there.

On Windows 7, however, it executes a shellcode, which decrypts and runs the `DevQueryBrokerPre.log` file.

Stage 1 Injection (explorer.exe)

SADBRIDGE utilizes PoolParty Variant 7 to inject shellcode into `explorer.exe` by targeting its thread pool's I/O completion queue. It first duplicates a handle to the target process's I/O completion queue. It then allocates memory within `explorer.exe` to store the shellcode. Additional memory is allocated to store a crafted `TP_DIRECT` structure, which includes the base address of the shellcode as the callback address. Finally, it calls `ZwSetIoCompletion`, passing a pointer to the `TP_DIRECT` structure to queue a packet to the I/O completion queue of the target process's worker factory (worker threads manager), effectively triggering the execution of the injected shellcode.

```
  h_IoCompletion = (__int64)TargetHandle;
  v0 = VirtualAllocEx(::h_proc_explorer_exe, 0LL, nSize, 0x3000u, 0x40u);
  if ( v0 )
  {
    p_baseAddrShellcode = (__int64)v0;
    LODWORD(v0) = WriteProcessMemory(::h_proc_explorer_exe, v0, shellcode, nSize, 0LL);
    if ( (_DWORD)v0 )
    {
      v17 = GetModuleHandleA("ntdll.dll");
      ZwSetIoCompletion = GetProcAddress(v17, "ZwSetIoCompletion");
      *(_QWORD *)&Buffer[56] = p_baseAddrShellcode;
      memset(Buffer, 0, 56);
      ZwSetIoCompletion_1 = (__int64 (__fastcall *)(_QWORD, _QWORD, _QWORD, _QWORD, _QWORD))ZwSetIoCompletion
      *(_QWORD *)&Buffer[64] = 0LL;
      RemoteDirectAddress = VirtualAllocEx(::h_proc_explorer_exe, 0LL, 0x48uLL, 0x3000u, 4u);
      WriteProcessMemory(::h_proc_explorer_exe, RemoteDirectAddress, Buffer, 0x48uLL, 0LL);
      LODWORD(v0) = ZwSetIoCompletion_1(h_IoCompletion, RemoteDirectAddress, 0LL, 0LL, 0LL);
    }
  }
```

I/O Completion Port Shellcode Injection

This shellcode decrypts the `DevQueryBrokerService.log` file, unmaps any memory regions occupying its preferred base address, maps the PE file to that address, and then executes its entry point. This behavior mirrors the previously observed shellcode.

Stage 2 Injection (spoolsv.exe/lsass.exe)

For Stage 2, SADBRIDGE injects shellcode into `spoolsv.exe`, or `lsass.exe` if `spoolsv.exe` is unavailable, using the same injection technique as in Stage 1. The shellcode exhibits similar behavior to the earlier stages: it decrypts `DevQueryBrokerPre.log` into a PE file, unmaps any regions occupying its preferred base address, maps the PE file, and then transfers execution to its entry point.

DevQueryBrokerService.log

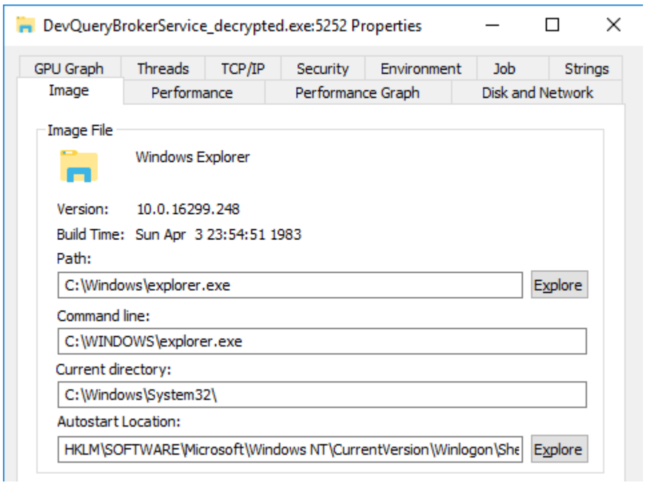
The shellcode decrypted from `DevQueryBrokerService.log` as mentioned in the previous section leverages a privilege escalation technique using the Windows Task Scheduler. SADBRIDGE integrates a public UAC bypass technique using the `IElevatedFactorySever` COM object to indirectly create the scheduled task. This task is configured to run `DevQueryBroker.exe` on a daily basis with SYSTEM level privileges using the task name `DevQueryBrokerService`.

```
}
CLSIDFromString(L"{A6BFEA43-501F-456F-A845-983D3AD7B8F0}", &pclsid);
*&pBindOptions[4] = 0LL;
v113 = 0LL;
v114 = 0LL;
*pBindOptions = 48;
v112 = 4;
ppv = 0LL;

if ( CoGetObject(L"Elevation:Administrator!new:{A6BFEA43-501F-456F-A845-983D3AD7B8F0}", pBindOptions, &riid, &ppv) < 0 )
{
```

GUID in Scheduled Task Creation (Virtual Factory for MaintenanceUI)

In order to cover its tracks, the malware spoofs the image path and command-line by modifying the Process Environment Block (PEB) directly, likely in an attempt to disguise the COM service as coming from `explorer.exe`.



DevQueryBrokerService.log Spoofed Image Command-Line

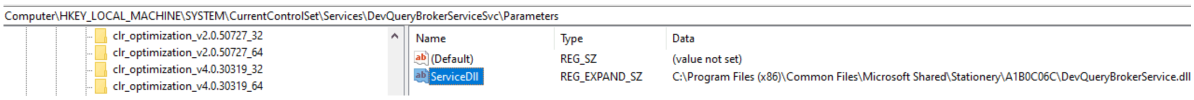
DevQueryBrokerPre.log

SADBRIDGE creates a service named `DevQueryBrokerServiceSvc` under the registry subkey `SYSTEM\CurrentControlSet\Services\DevQueryBrokerServiceSvc` with the following attributes:

- **Description**: Enables apps to discover devices with a background task.
- **DisplayName**: DevQuery Background Discovery Broker Service
- **ErrorControl**: 1
- **ImagePath**: `%systemRoot%\system32\svchost.exe -k netsvcs`
- **ObjectName**: LocalSystem
- **Start**: 2 (auto-start)
- **Type**: 16.
- **Failure Actions**:
    - Resets failure count every 24 hours.
    - Executes three restart attempts: a 20ms delay for the first, and a 1-minute delay for the second and third.

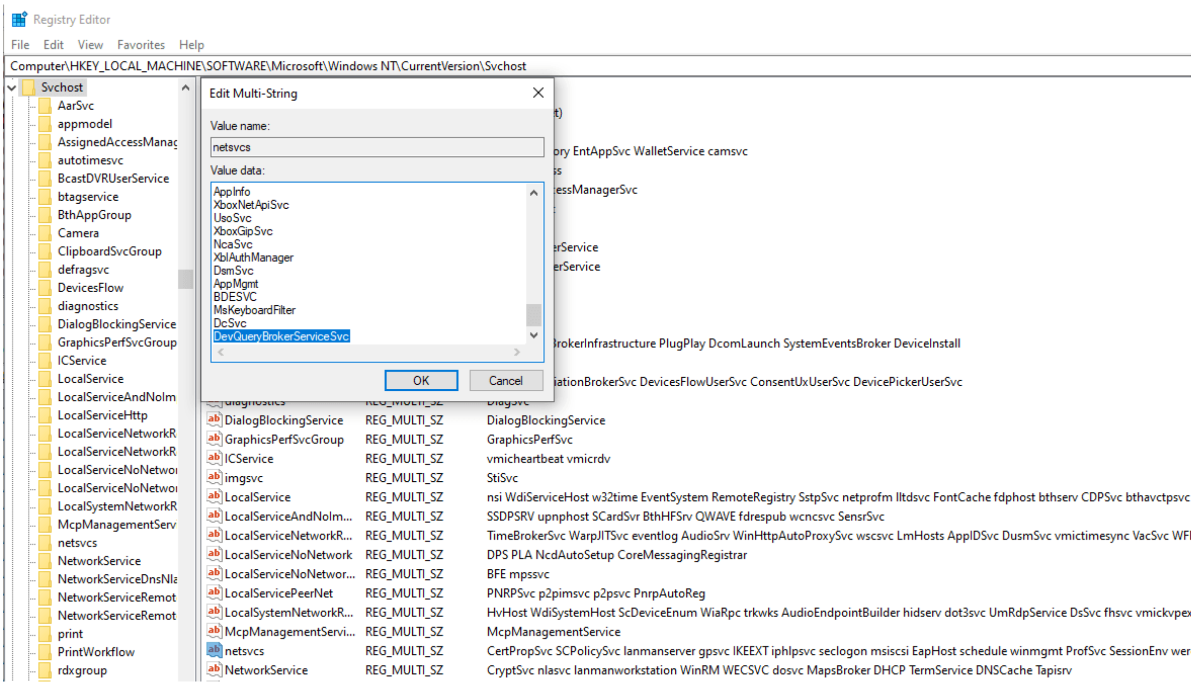
The service parameters specify the `ServiceDll` located at `C:\Program Files (x86)\Common Files\Microsoft Shared\Stationery\<hostname_hash>\DevQueryBrokerService.dll`. If the DLL file does not exist, it will be dropped to disk right after.

`DevQueryBrokerService.dll` has a similar code structure as `HealthServiceRuntime.dll`, which is seen in the earlier stages of the execution chain. It is responsible for decrypting `DevQueryBroker.log` and running it. The `ServiceDll` will be loaded and executed by `svchost.exe` when the service starts.



svchost.exe's malicious ServiceDLL parameter

Additionally, it modifies the `SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost\netsvcs` key to include an entry for `DevQueryBrokerServiceSvc` to integrate the newly created service into the group of services managed by the `netsvcs` service host group.



Modifies the netsvc registry key to add DevQueryBrokerServiceSvc

SADBRIDGE then deletes the scheduled task and service created previously by removing the registry subkeys `SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Schedule\\TaskCache\\Tree\\DevQueryBrokerService` and `SYSTEM\\CurrentControlSet\\Services\\DevQueryBrokerService`.

Finally, it removes the files `DevQueryBroker.exe` and `HealthServiceRuntime.dll` in the `C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker` folder, as the new persistence mechanism is in place.

GOSAR Injection

In the latter half of the code, SADBRIDGE enumerates all active sessions on the local machine using the `WTSEnumerateSessionsA` API.

If sessions are found, it iterates through each session:

- For each session, it attempts to retrieve the username (`WTSUserName`) using `WTSQuerySessionInformationA`. If the query fails, it moves to the next session.
- If `WTSUserName` is not empty, the code targets `svchost.exe`, passing its path, the session ID, and the content of the loader configuration to a subroutine that injects the final stage.
- If `WTSUserName` is empty but the session's `WinStationName` is `"Services"` (indicating a service session), it targets `dllhost.exe` instead, passing the same parameters to the final stage injection subroutine.

If no sessions are found, it enters an infinite loop to repeatedly enumerate sessions and invoke the subroutine for injecting the final stage, while performing checks to avoid redundant injections.

Logged-in sessions target `svchost.exe`, while service sessions or sessions without a logged-in user target `dllhost.exe`.

```
  WTSEnumerateSessionsA(0LL, 0, 1u, &ppSessionInfo, &pCount);
  if ( !pCount )
  {                                             // If no sessions found
LABEL_18:
    WTSFreeMemory(ppSessionInfo);
    while ( 1 )
    {                                           // infinite loop
      mw_nextstage_wrapper(
        (LPCSTR)(p_EDBTmp + 0xF3C),
        str_pathToSvcHostExe,
        str_dllhostExe,
        str_pathToDllhostExe,
        (BYTE *)p_EDBTmp);                      // p_EDBTmp[0xF3C] = svchost.exe
      Sleep(300000u);
    }
  }
  while ( 1 )
  {
    v8 = session_ctr;
    if ( WTSQuerySessionInformationA(0LL, ppSessionInfo[v8].SessionId, WTSUserName, &WTSUserName, &pBytesReturned) )
      break;                                    // fallthrough if query fails
LABEL_17:
    if ( ++session_ctr >= pCount )              // check if all sessions have been processed
      goto LABEL_18;
  }
  if ( lstrcmpiA(WTSUserName, empty_string) )
  {                                             // if WTSUserName is not empty
    Sleep(0xBB8u);
    FileAttributesA = GetFileAttributesA(str_pathToSvcHostExe);
    str_pathToDllhostOrSvchost = str_pathToSvcHostExe;
    SessionId = ppSessionInfo[v8].SessionId;
    if ( FileAttributesA != -1 )
    {
LABEL_15:
      mw_nextstage(str_pathToDllhostOrSvchost, SessionId, (BYTE *)p_EDBTmp);
      goto LABEL_16;
    }
  }
  else
  {                                             // if WTSUserName is empty
    if ( lstrcmpiA(ppSessionInfo[v8].pWinStationName, str_Services) )
    {
LABEL_16:
      WTSFreeMemory(WTSUserName);
      goto LABEL_17;
    }
    Sleep(0xBB8u);
    SessionId = ppSessionInfo[v8].SessionId;
  }
  str_pathToDllhostOrSvchost = str_pathToDllhostExe;
  goto LABEL_15;
}
```

Enumeration of active sessions

If a session ID is available, the code attempts to duplicate the user token for that session and elevate the duplicated token's integrity level to `S-1-16-12288` (System integrity). It then uses the elevated token to create a child process (`svchost.exe` or `dllhost.exe`) via `CreateProcessAsUserA`.

```
      strcpy((char *)&v66, "S-1-16-12288");
      LibraryA = LoadLibraryA("Advapi32.dll");
      ConvertStringSidToSidA = (BOOL (__stdcall *)(LPCSTR, PSID *))GetProcAddress(LibraryA, "ConvertStringSidToSidA");
      if ( ((unsigned int (__fastcall *)(LPVOID *, PSID *))ConvertStringSidToSidA)(&v66, &pSid) )
      {
        v60 = pSid;
        v61 = 32;
        LengthSid = GetLengthSid(pSid);
        SetTokenInformation(TokenHandle, TokenIntegrityLevel, &v60, LengthSid + 16);
        dwCreationFlags = 0x2000034;
        v23 = LoadLibraryA("Userenv.dll");
        CreateEnvironmentBlock = (BOOL (__stdcall *)(LPVOID *, HANDLE, BOOL))GetProcAddress(
                                                                              v23,
                                                                              "CreateEnvironmentBlock");
        if ( ((unsigned int (__fastcall *)(LPVOID *, HANDLE, __int64))CreateEnvironmentBlock)(&v57, TokenHandle, 1LL) )
        {
          lpEnvironment = v57;
          dwCreationFlags = 0x2000434;
        }
        else
        {
          lpEnvironment = 0LL;
          v57 = 0LL;
        }
        ProcessAsUserA = CreateProcessAsUserA(
                           TokenHandle,
                           0LL,
                           str_pathToDllhostOrSvchost,
                           0LL,
                           0LL,
                           0,
                           dwCreationFlags,
                           lpEnvironment,
                           0LL,
                           lpStartupInfo,
                           lpProcessInformation);
```

Duplication of user token and elevating token privileges

If token manipulation fails or no session ID is available (system processes can have a session ID of 0), it falls back to creating a process without a token using `CreateProcessA`.

The encrypted shellcode `C:\ProgramData\Microsoft\DeviceSync\Device\Stage\Data\DevQueryBroker\temp.ini` is decrypted using the same XOR and LZNT1 decompression technique seen previously to decrypt `.log` files, and APC injection is used to queue the shellcode for execution in the newly created process's thread.

```
TokenInformation_1 = (unsigned int)TokenInformation;
TokenInformation_2 = (LPVOID)(unsigned int)TokenInformation;
pSid = 0LL;
v41 = LoadLibraryA("ntdll.dll");
NtAllocateVirtualMemory = GetProcAddress(v41, "NtAllocateVirtualMemory");
((void (__fastcall *)(HANDLE, PSID *, _QWORD, LPVOID *, int, int))NtAllocateVirtualMemory)(
  hProcess,
  &pSid,
  0LL,
  &TokenInformation_2,
  0x3000,
  0x40);
v43 = LoadLibraryA("ntdll.dll");
NtWriteVirtualMemory = GetProcAddress(v43, "NtWriteVirtualMemory");
((void (__fastcall *)(HANDLE, PSID, LPVOID, _QWORD, _QWORD))NtWriteVirtualMemory)(
  hProcess,
  pSid,
  UncompressedTempINI,
  TokenInformation_1,
  0LL);
v45 = LoadLibraryA("ntdll.dll");
NtQueueApcThread = GetProcAddress(v45, "NtQueueApcThread");
((void (__fastcall *)(PLUID, PSID, PSID, _QWORD, _QWORD))NtQueueApcThread)(h_Thread, pSid, pSid, 0LL, 0LL)
v47 = LoadLibraryA("ntdll.dll");
NtResumeThread = GetProcAddress(v47, "NtResumeThread");
((void (__fastcall *)(PLUID, _QWORD))NtResumeThread)(h_Thread, 0LL);
```

APC injection to run GOSAR

Finally, the injected shellcode decrypts `DevQueryBrokerCore.log` to GOSAR and runs it in the newly created process's memory.

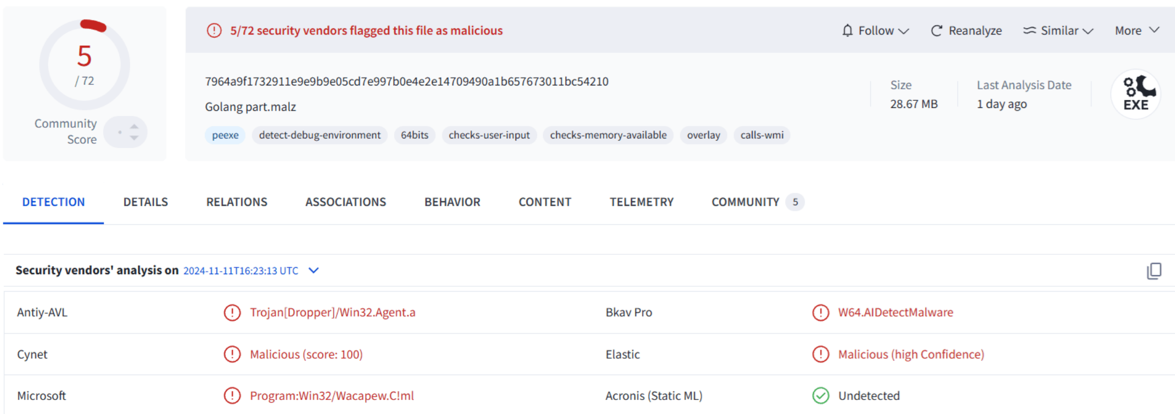
spoolsv.exe	2328		9.07 MB	NT AUTHORITY\SYSTEM	Spooler SubSystem App
dllhost.exe	6724	0.17	59.7 MB	NT AUTHORITY\SYSTEM	COM Surrogate
svchost.exe	6528	0.26	60.03 MB	DESKTOP-OB0D13M\JiaYuChan	Host Process for Windows Ser.

GOSAR injected into dllhost.exe and svchost.exe

GOSAR Introduction

GOSAR is a multi-functional remote access trojan found targeting Windows and Linux systems. This backdoor includes capabilities such as retrieving system information, taking screenshots, executing commands, keylogging, and much more. The GOSAR backdoor retains much of QUASAR's core functionality and behavior, while incorporating several modifications that differentiate it from the original version.

By rewriting malware in modern languages like Go, this can offer reduced detection rates as many antivirus solutions and malware classifiers struggle to identify malicious strings/characteristics under these new programming constructs. Below is a good example of an unpacked GOSAR receiving only 5 detections upon upload.



Low detection rate on GOSAR VT upload

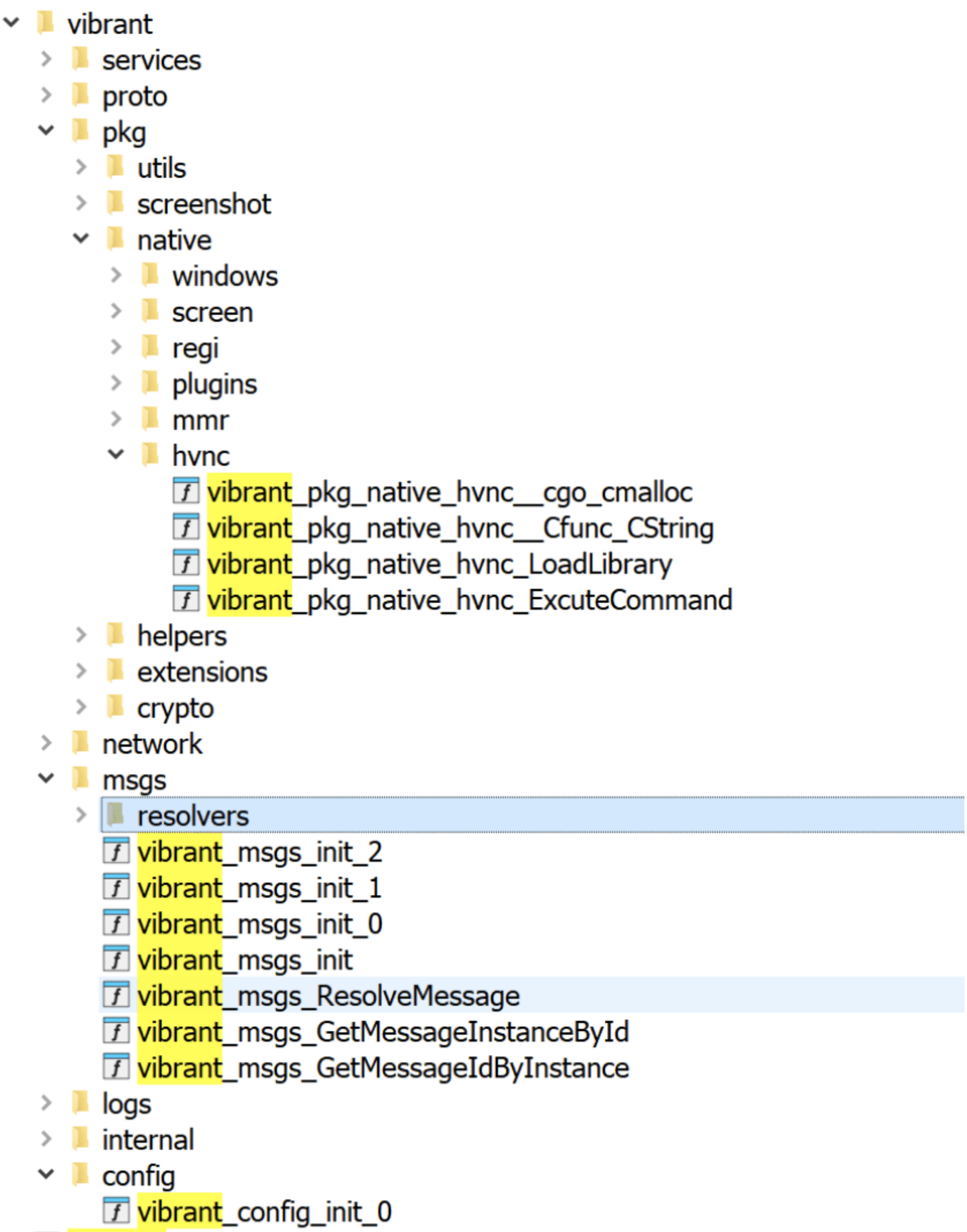
Notably, this variant supports multiple platforms, including ELF binaries for Linux systems and traditional PE files for Windows. This cross-platform capability aligns with the adaptability of Go, making it more versatile than the original

.NET-based QUASAR. Within the following section, we will focus on highlighting GOSAR's code structure, new features and additions compared to the open-source version (QUASAR).

GOSAR Code Analysis Overview

Code structure of GOSAR

As the binary retained all its symbols, we were able to reconstruct the source code structure, which was extracted from a sample of version `0.12.01`



GOSAR code structure

- **vibrant/config**: Contains the configuration files for the malware.
- **vibrant/proto**: Houses all the Google Protocol Buffers (proto) declarations.
- **vibrant/network**: Includes functions related to networking, such as the main connection loop, proxy handling and also thread to configure the firewall and setting up a listener
- **vibrant/msgs/resolvers**: Defines the commands handled by the malware. These commands are assigned to an object within the `vibrant_msgs_init*` functions.
- **vibrant/msgs/services**: Introduces new functionality, such as running services like keyloggers, clipboard logger, these services are started in the `vibrant_network._ptr_Connection.Start` function.

- **vibrant/logs**: Responsible for logging the malware's execution. The logs are encrypted with an AES key stored in the configuration. The malware decrypts the logs in chunks using AES.
- **vibrant/pkg/helpers**: Contains helper functions used across various malware commands and services.
- **vibrant/pkg/screenshot**: Handles the screenshot capture functionality on the infected system.
- **vibrant/pkg/utils**: Includes utility functions, such as generating random values.
- **vibrant/pkg/native**: Provides functions for calling Windows API (WINAPI) functions.

New Additions to GOSAR

Communication and information gathering

This new variant continues to use the same communication method as the original, based on **TCP TLS**. Upon connection, it first sends system information to the C2, with 4 new fields added:

- IPAddress
- AntiVirus
- ClipboardSettings
- Wallets

The list of AntiViruses and digital wallets are initialized in the function `vibrant_pkg_helpers_init` and can be found at the bottom of this document.

Services

The malware handles 3 services that are started during the initial connection of the client to the C2:

- vibrant_services_KeyLogger
- vibrant_services_ClipboardLogger
- vibrant_services_TickWriteFile

```
v20.cap = 1LL;
github_com_sirupsen_logrus__ptr_Logger_Log(off_7FF7D59290E8, 4u, v20);
runtime_newproc((unsigned int)&w_vibrant_services_KeyLoggervibrant_services_KeyLogger);
runtime_newproc((unsigned int)&w_vibrant_services_ClipboardLogger);
runtime_newproc((unsigned int)&w_vibrant_services_TickWriteFile);
```

GOSAR services

KeyLogger

The keylogging functionality in GOSAR is implemented in the `vibrant_services_KeyLogger` function. This feature relies on Windows APIs to intercept and record keystrokes on the infected system by setting a global Windows hook with `SetWindowsHookEx` with the parameter `WH_KEYBOARD_LL` to monitor low-level keyboard events. The hook function is named `vibrant_services_KeyLogger_func1`.

```
HHOOK = vibrant_pkg_native_windows_SetWindowsHookEx(WH_KEYBOARD_LL, (_DWORD)hook_function, 0, 0);
while ( 1 )
{
  p__4_uintptr = (_4_uintptr *)runtime_newobject(&RTYPE__4_uintptr);
  (*p__4_uintptr)[0] = (uintptr)p_win32_MSG;
  (*p__4_uintptr)[1] = 0LL;
  *(_OWORD *)&(*p__4_uintptr)[2] = v5;
  v16 = (int)p__4_uintptr;
  if ( !((__int64 (__golang *)(_DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, _DWORD, __int64))golang_org_x_sys_windows__ptr_LazyProc_Call)
          (_DWORD)GetMessage,                // GetMessage
          (_DWORD)p__4_uintptr,
          4,
          4,
```

GOSAR setting the keylogger

ClipboardLogger

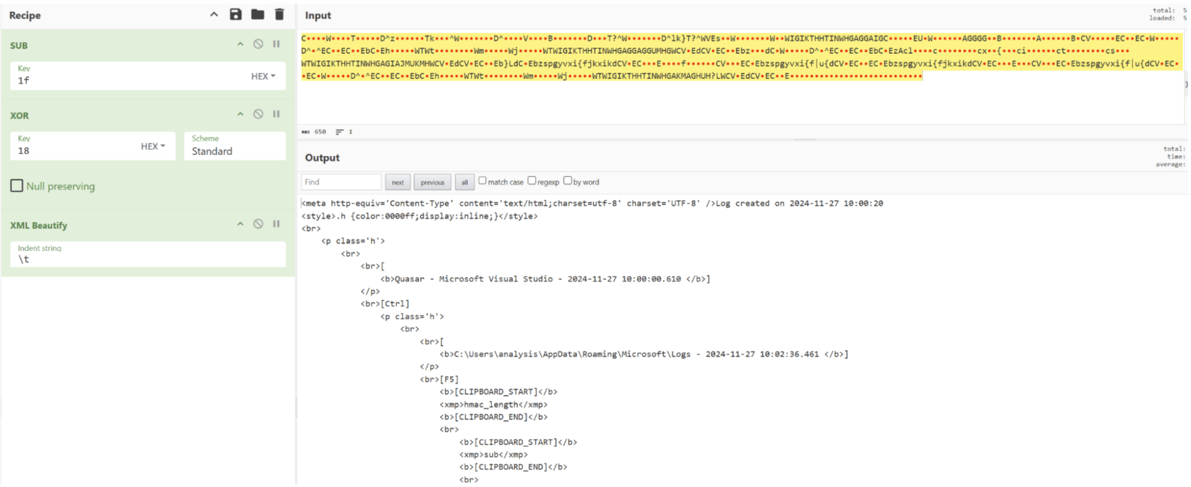
The clipboard logging functionality is straightforward and relies on Windows APIs. It first checks for the availability of clipboard data using `IsClipboardFormatAvailable` then retrieves it using `GetClipboardData` API.

```
if ( golang_org_x_sys_windows__ptr_LazyProc_Call(
        (windows_LazyProc *)IsClipboardFormatAvailable,
        (int)p__1_uintptr,
        1,
        1,
        a5) )
{
  v7 = vibrant_pkg_helpers_waitOpenClipboard();
  if ( v7 )
  {
    runtime_UnlockOSThread(v7);
    return v5;
  }
  v8 = (_1_uintptr *)runtime_newobject(&RTYPE__1_uintptr);
  (*v8)[0] = 13LL;
  v9 = golang_org_x_sys_windows__ptr_LazyProc_Call((windows_LazyProc *)GetClipboardData, (int)v8, 1, 1, a5);
  if ( !v9
```

GOSAR clipboard logging

TickWriteFile

Both `ClipboardLogger` and `KeyLogger` services collect data that is written by the `TickWriteFile` periodically to directory (`C:\ProgramData\Microsoft\Windows\Start Menu\Programs\diagnostics`) under a file of the current date, example `2024-11-27`.
It can be decrypted by first subtracting the value `0x1f` then xoring it with the value `0x18` as shown in the CyberChef recipe.



CyberChef recipe used to decrypt keylogger logs

Networking setup

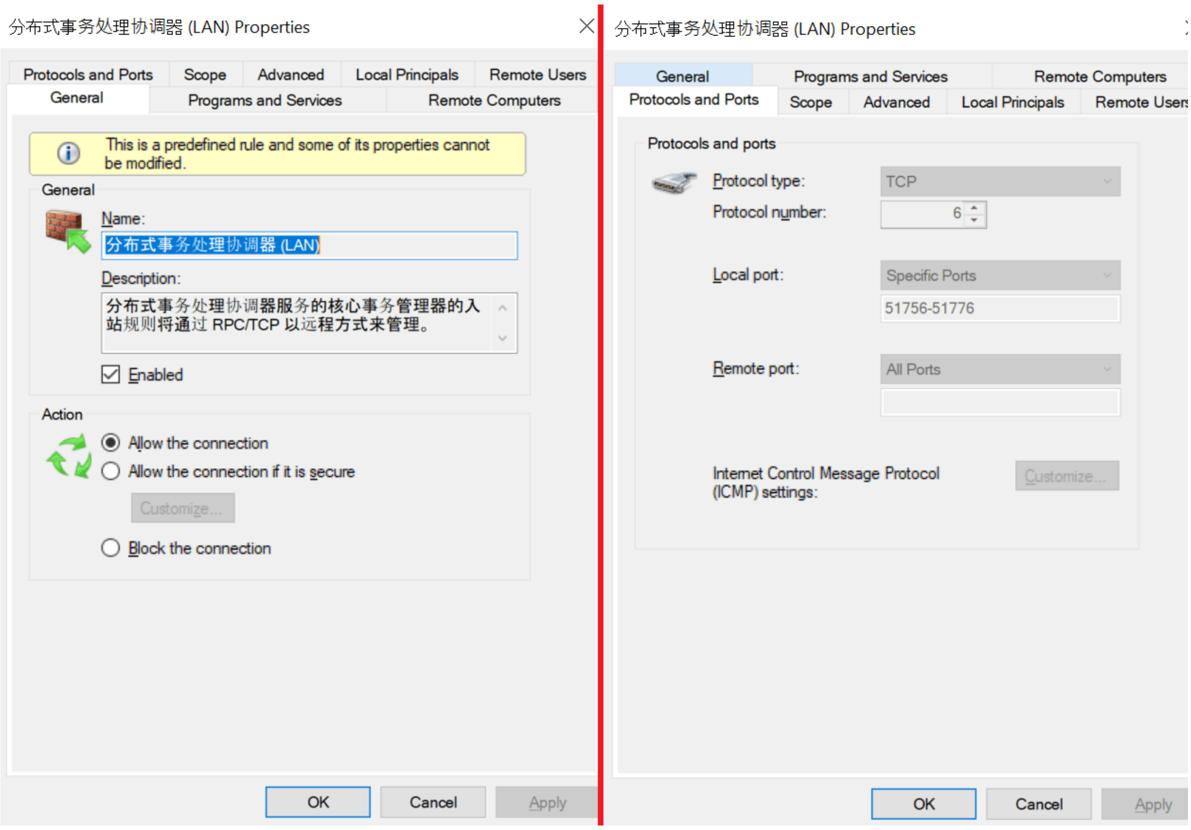
After initializing its services, the malware spawns **three threads** dedicated to its networking setup.

- vibrant_network_ConfigFirewallRule
- vibrant_network_ConfigHosts
- vibrant_network_ConfigAutoListener

Threads handling networking setup

ConfigFirewallRule

The malware creates an inbound firewall rule for the ports range `51756-51776` under a Chinese name that is translated to `Distributed Transaction Coordinator (LAN)` it allows all programs and IP addresses inbound the description is set to :`Inbound rules for the core transaction manager of the Distributed Transaction Coordinator service are managed remotely through RPC/TCP.`



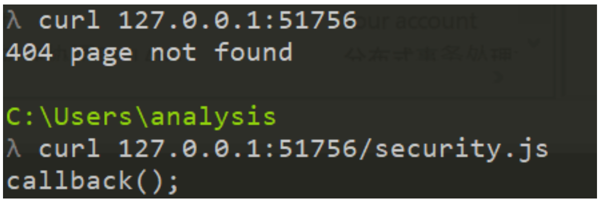
Added firewall rule

ConfigHosts

This function adds an entry to `c:\Windows\System32\Drivers\etc\hosts` the following `127.0.0.1 micrornetworks.com`. The reason for adding this entry is unclear, but it is likely due to missing functionalities or incomplete features in the malware's current development stage.

ConfigAutoListener

This functionality of the malware runs an HTTP server listener on the first available port within the range `51756-51776`, which was previously allowed by a firewall rule. Interestingly, the server does not handle any commands, which proves that the malware is still under development. The current version we have only processes a `GET` request to the URI `/security.js`, responding with the string `callback();`, any other request returns a 404 error code. This minimal response could indicate that the server is a placeholder or part of an early development stage, with the potential for more complex functionalities to be added later



Callback handled by GOSAR

Logs

The malware saves its runtime logs in the directory: `%APPDATA%\Roaming\Microsoft\Logs` under the filename formatted as: `windows-update-log-<YearMonthDay>.log`.
Each log entry is encrypted with HMAC-AES algorithm; the key is hardcoded in the `vibrant_config` function, the following is an example:

```
[2024-11-27 11:54:29.024] [INFO] [main.go:28] whether to use mutexes: 24533250adfe
[2024-11-27 11:54:29.035] [INFO] [main.go:38] create mutex success, the handle is 488
[2024-11-27 11:54:29.035] [INFO] [conn.go:78] starting services ...
[2024-11-27 11:54:53.473] [INFO] [conn.go:102] connecting to server ...
[2024-11-27 11:54:53.480] [INFO] [fish_helper_windows.go:196] all hosts are already in the file.
[2024-11-27 11:54:53.576] [INFO] [fish_helper_windows.go:249] ready to start auto listener, listening port is 51756
[2024-11-27 11:54:54.784] [INFO] [conn.go:186] connect host 127.0.0.1:1080 success
[2024-11-27 11:54:58.229] [INFO] [sysinfo_helper.go:138] create user identify file
[2024-11-27 11:54:58.230] [INFO] [sysinfo_helper.go:145] user identify file exist, read it
[2024-11-27 11:54:58.231] [INFO] [sysinfo_helper.go:151] user identify file content is 5f1da1ff-d264-4ac9-98a0-ca031285c73
```
Logs example generated by GOSAR

The attacker can remotely retrieve the malware's runtime logs by issuing the command `ResolveGetRunLogs`.

Plugins

The malware has the capability to execute plugins, which are PE files downloaded from the C2 and stored on disk encrypted with an XOR algorithm. These plugins are saved at the path: `C:\ProgramData\policy-err.log`. To execute a plugin, the command `ResolveDoExecutePlugin` is called, it first checks if a plugin is available.

```
v5 = vibrant_pkg_utils_FileExist((int)"C:\\ProgramData\\policy-err.log", 29);
if ( (_BYTE)v5 )
{
```

GOSAR checking for existence of a plugin to execute

It then loads a native DLL reflectively that is stored in base64 format in the binary named `plugins.dll` and executes its export function `ExecPlugin`.

```
ProcAddress = vibrant_pkg_native_mmr_memoryGetProcAddress(DLL, "ExecPlugin", 0xAuLL, (int)"plugins.dll", 11);
if ( dword_7FF76E024B60 )
{
  ProcAddress = (mmr_Proc *)runtime_gcWriteBarrier2(ProcAddress);
  *v22 = ProcAddress;
  v22[1] = (mmr_Proc *)plugins_dll_ExecPlugin;
}
plugins_dll_ExecPlugin = (__int64)ProcAddress;
if ( "ExecPlugin" )
{
```

GOSAR loading plugins.dlll and calling ExecPlugin

`ExecPlugin` creates a suspended process of `C:\Windows\System32\msiexec.exe` with the arguments `/package /quiet`. It then queues Asynchronous Procedure Calls (APC) to the process's main thread. When the thread is resumed, the queued shellcode is executed.

```
LibraryA = LoadLibraryA("ntdll.dll");
NtAllocateVirtualMemory = GetProcAddress(LibraryA, "NtAllocateVirtualMemory");
NtQueueApcThread = GetProcAddress(LibraryA, "NtQueueApcThread");
StartupInfo.cb = 104;
NtResumeThread = GetProcAddress(LibraryA, "NtResumeThread");
memset(&StartupInfo.cb + 1, 0, 100);
v4 = (char *)operator new(0x104uLL);
strcpy(v4, " /package /quiet");
if ( !CreateProcessA(
        "C:\\Windows\\System32\\msiexec.exe",
        v4,
        0LL,
        0LL,
        0,
        0x2000034u,
        0LL,
        0LL,
        &StartupInfo,
        &ProcessInformation) )
  return 0;
```

GOSAR plugin module injecting a PE in msiexec.exe

The shellcode reads the encrypted plugin stored at `C:\ProgramData\policy-err.log`, decrypts it using a hardcoded 1-byte XOR key, and reflectively loads and executes it.

HVNC

The malware supports hidden VNC(HVNC) through the existing socket, it exposes 5 commands

- ResolveHVNCCommand
- ResolveGetHVNCScreen
- ResolveStopHVNC
- ResolveDoHVNCKeyboardEvent
- ResolveDoHVNCMouseEvent

The first command that is executed is `ResolveGetHVNCScreen` which will first initialise it and set up a view, it uses an embedded native DLL `HiddenDesktop.dll` in base64 format, the DLL is reflectively loaded into memory and executed.

The DLL is responsible for executing low level APIs to setup the HVNC, with a total of 7 exported functions:

- ExcuteCommand
- DoMouseScroll
- DoMouseRightClick
- DoMouseMove
- DoMouseLeftClick
- DoKeyPress
- CaptureScreen

The first export function called is `Initialise` to initialise a desktop with `CreateDesktopA` API. This HVNC implementation handles 17 commands in total that can be found in `ExcuteCommand` export, as noted it does have a typo in the name, the command ID is forwarded from the malware's command `ResolveHVNCCommand` that will call `ExcuteCommand`.

Command ID	Description
0x401	The function first disables taskbar button grouping by setting the `TaskbarGlomLevel` registry key to `2` under `Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced`. Next, it ensures the taskbar is always visible and on top by using `SHAppBarMessage` with the `ABM_SETSTATE` command, setting the state to `ABS_ALWAYSONTOP`.
0x402	Spawns a RUN dialog box by executing the 61th export function of `shell32.dll. C:\Windows\system32\rundll32.exe shell32.dll,#61`
0x403	Runs an instance of `powershell.exe`
0x404	Executes a PE file stored in `C:\\ProgramData\\shell.log`
0x405	Runs an instance of `chrome.exe`
0x406	Runs an instance of `msedge.exe`
0x407	Runs an instance of `firefox.exe`
0x408	Runs an instance of `iexplore.exe`
0x409	Runs an instance of `360se.exe`
0x40A	Runs an instance of `360ChromeX.exe`.
0x40B	Runs an instance of `SogouExplorer.exe`
0x40C	Close current window
0x40D	Minimizes the specified window
0x40E	Activates the window and displays it as a maximized window
0x40F	Kills the process of a window
0x410	Sets the clipboard
0x411	Clears the Clipboard

Screenshot

The malware loads reflectively the third and last PE DLL embedded in base64 format named `Capture.dll`, it has 5 export functions:

- CaptureFirstScreen
- CaptureNextScreen
- GetBitmapInfo
- GetBitmapInfoSize
- SetQuality

The library is first initialized by calling `resolvers_ResolveGetBitmapInfo` that reflectively loads and executes its `DllEntryPoint` which will setup the screen capture structures using common Windows APIs like `CreateCompatibleDC`, `CreateCompatibleBitmap` and `CreateDIBSection`. The 2 export functions `CaptureFirstScreen` and `CaptureNextScreen` are used to capture a screenshot of the victim's desktop as a JPEG image.

Observation

Interestingly, the original .NET QUASAR server can still be used to receive beaconing from GOSAR samples, as they have retained the same communication protocol. However, operational use of it would require significant modifications to support GOSAR functionalities.

It is unclear whether the authors updated or extended the open source .NET QUASAR server, or developed a completely new one. It is worth mentioning that they have retained the default listening port, 1080, consistent with the original implementation.

New functionality

The following table provides a description of all the newly added commands:

New commands	
ResolveDoRoboCopy	Executes `RoboCopy` command to copy files
ResolveDoCompressFiles	Compress files in a zip format

New commands	
ResolveDoExtractFile	Extract a zip file
ResolveDoCopyFiles	Copies a directory or file in the infected machine
ResolveGetRunLogs	Get available logs
ResolveHVNCCommand	Execute a HVNC command
ResolveGetHVNCScreen	Initiate HVNC
ResolveStopHVNC	Stop the HVNC session
ResolveDoHVNCKeyboardEvent	Send keyboard event to the HVNC
ResolveDoHVNCMouseEvent	Send mouse event to the HVNC
ResolveDoExecutePlugin	Execute a plugin
ResolveGetProcesses	Get a list of running processes
ResolveDoProcessStart	Start a process
ResolveDoProcessEnd	Kill a process
ResolveGetBitmapInfo	Retrieve the **BITMAPINFO** structure for the current screen's display settings
ResolveGetMonitors	Enumerate victim's display monitors with `EnumDisplayMonitors` API
ResolveGetDesktop	Start screen capture functionality
ResolveStopGetDesktop	Stop the screen capture functionality
ResolveNewShellExecute	Opens pipes to a spawned cmd.exe process and send commands to it
ResolveGetSchTasks	Get scheduled tasks by running the command `schtasks /query /fo list /v`
ResolveGetScreenshot	Capture a screenshot of the victim's desktop
ResolveGetServices	Get the list of services with a **WMI** query: `select * from Win32_Service`
ResolveDoServiceOperation	Start or stop a service
ResolveDoDisableMultiLogon	Disable multiple session by user by setting the value `fSingleSessionPerUser` to 1 under the key `HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\TerminalServer`
ResolveDoRestoreNLA	Restores the security settings for Remote Desktop Protocol (RDP), enabling **Network Level Authentication** (NLA) and enforcing **SSL/TLS** encryption for secure communication.
ResolveGetRemoteClientInformation	Get a list of all local users that are enabled, the **RDP port** and **LAN IP** and **OS specific information**: **DisplayVersion**, **SystemRoot** and **CurrentBuildNumber** extracted from the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion`
ResolveDoInstallWrapper	Setup a Hidden Remote Desktop Protocol (**HRDP**)
ResolveDoUninstallWrapper	Uninstall **HRDP**
ResolveDoRecoverPrivileges	Restores the original **`HKEY_LOCAL_MACHINE\\SAM\\SAM`** registry before changes were made during the installation of the **HRDP**
ResolveGetRemoteSessions	Retrieve information about the RDP sessions on the machine.
ResolveDoLogoffSession	Logoff RDP session with **`WTSLogoffSession`** API
ResolveGetSystemInfo	Get system information
ResolveGetConnections	Get all the connections in the machine
ResolveDoCloseConnection	Not implemented

Malware and MITRE ATT&CK

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that threats use against enterprise networks.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

Mitigating REF3864

Detection

Prevention

YARA

Elastic Security has created YARA rules to identify this activity.

Observations

The following observables were discussed in this research:

Observable	Type	Name	Reference
opera-x[.]net	domain-name		Landing page
teledown-cn[.]com	domain-name		Landing page
15af8c34e25268b79022d3434aa4b823ad9d34f3efc6a8124ecf0276700ecc39	SHA-256	`NetFxRepairTools.msi`	MSI
accd651f58dd3f7eaaa06df051e4c09d2edac67bb046a2dcb262aa6db4291de7	SHA-256	`x64bridge.dll`	SADBRIDGE
7964a9f1732911e9e9b9e05cd7e997b0e4e2e14709490a1b657673011bc54210	SHA-256		GOSAR
ferp.googledns[.]io	domain-name		GOSAR C2 Server
hk-dns.secssl[.]com	domain-name		GOSAR C2 Server
hk-dns.winsiked[.]com	domain-name		GOSAR C2 Server
hk-dns.wkossclsaleklddeff[.]is	domain-name		GOSAR C2 Server
hk-dns.wkossclsaleklddeff[.]io	domain-name		GOSAR C2 Server

References

The following were referenced throughout the above research:

Appendix

Hashing algorithm (SADBRIDGE)

```
def ror(x, n, max_bits=32) -> int:
    """Rotate right within a max bit limit, default 32-bit."""
    n %= max_bits
    return ((x >> n) | (x << (max_bits - n))) & (2**max_bits - 1)

def ror_13(data) -> int:
    data = data.encode('ascii')
    hash_value = 0

    for byte in data:
        hash_value = ror(hash_value, 13)

        if byte >= 0x61:
            byte -= 32  # Convert to uppercase
        hash_value = (hash_value + byte) & 0xFFFFFFFF

    return hash_value


def generate_hash(data, dll) -> int:
    dll_hash = ror_13(dll)
    result = (dll_hash + ror_13(data)) & 0xFFFFFFFF

    return hex(result)
```

AV products checked in GOSAR

360sd.exe	kswebshield.exe
360tray.exe	kvmonxp.exe
a2guard.exe	kxetray.exe
ad-watch.exe	mcshield.exe
arcatasksservice.exe	mcshield.exe
ashdisp.exe	miner.exe
avcenter.exe	mongoosagui.exe
avg.exe	mpmon.exe
avgaurd.exe	msmpeng.exe
avgwdsvc.exe	mssecess.exe
avk.exe	nspupsvc.exe
avp.exe	ntrtscan.exe
avp.exe	patray.exe
avwatchservice.exe	pccntmon.exe
ayagent.aye	psafesystray.exe
baidusdsvc.exe	qqpcrtp.exe
bkavservice.exe	quhlpsvc.EXE
ccapp.exe	ravmond.exe
ccSetMgr.exe	remupd.exe
ccsvchst.exe	rfwmain.exe
cksoftshiedantivirus4.exe	rtvscan.exe
cleaner8.exe	safedog.exe
cmctrayicon.exe	savprogress.exe
coranticontrolcenter32.exe	sbamsvc.exe
cpf.exe	spidernt.exe
egui.exe	spywareterminatorshield.exe
f-prot.EXE	tmbmsrv.exe
f-prot.exe	unthreat.exe
f-secure.exe	usysdiag.exe
fortitray.exe	v3svc.exe
hipstray.exe	vba32lder.exe
iptray.exe	vsmon.exe
k7tsecurity.exe	vsserv.exe
knsdtray.exe	wsctrl.exe
kpfwtray.exe	yunsuo_agent_daemon.exe
ksafe.exe	yunsuo_agent_service.exe