Projeto: Captura de Consumo de Alarmes FTAE via Injeção COM

# Objetivo

O projeto visa analisar em profundidade como o FactoryTalk Alarms and Events (FTAE) do FactoryTalk Studio v13 (ambiente distribuído) consome e manipula eventos de alarme. A ideia é interceptar as chamadas COM feitas pelo cliente e servidor, permitindo investigar:

* Como ocorre a escuta de alarmes.
* O comportamento interno ao receber eventos.
* Como cada alarme é configurado e instanciado.

# Ambiente de Teste

* SO: Windows 10
* IDE: Visual Studio 2022
* Linguagem: C# 7.3
* Framework: .NET Framework 4.8
* Ferramentas auxiliares: PowerShell 5.1 para execução de testes e automação.
* Execução: Cliente e Servidor FTAE rodando na mesma máquina.

# Estrutura da Solução

A solução (Banner.sln) contém dois projetos principais:

* InjectorConsole (EXE): localiza PID, injeta a DLL e inicializa o logging.
* ComHookLib (DLL): implementa RemoteEntry, hooks de UI (UiHook) e hooks de COM.

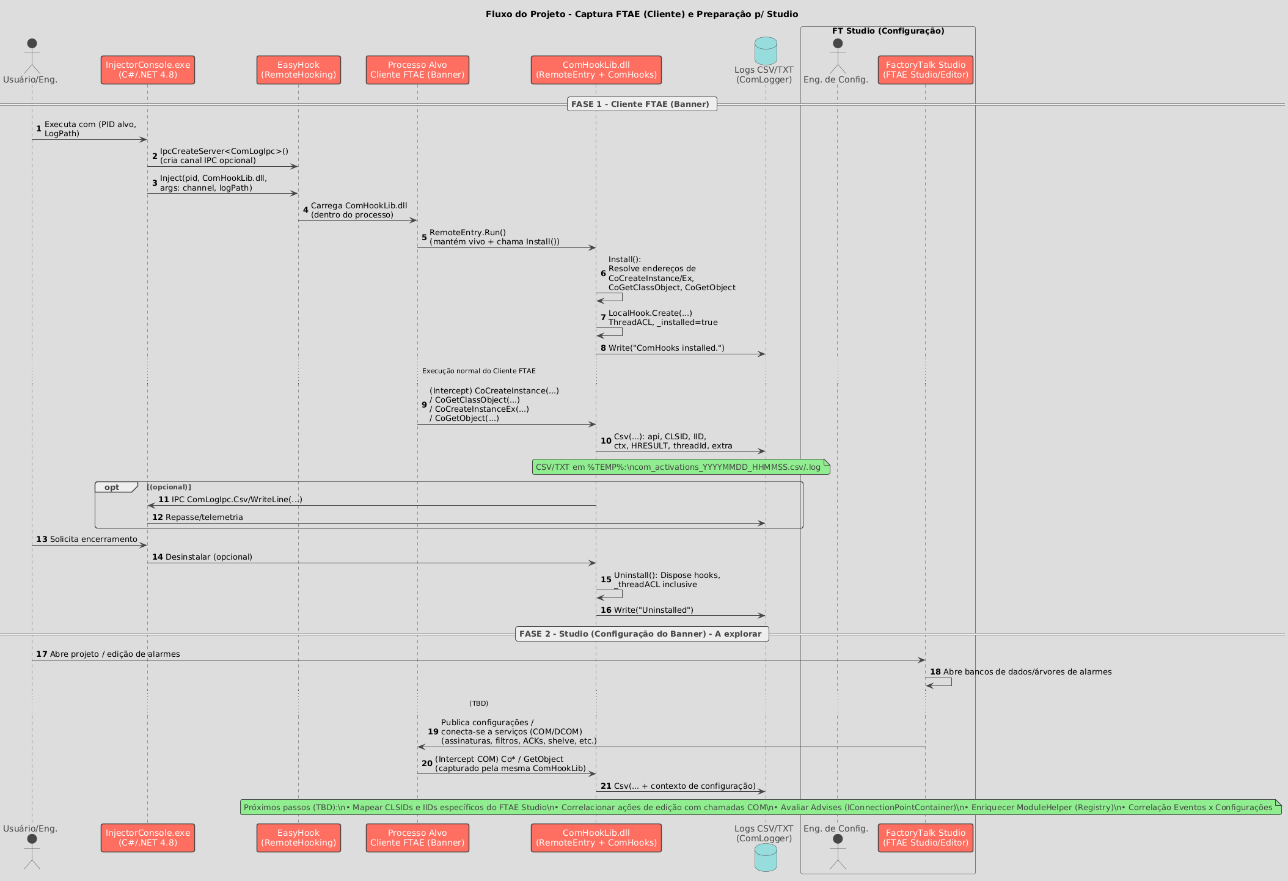


FIGURA 1 - FLUXO DO PROGRAMA (placeholder)

# Etapa 1/4 – Base de Injeção e Placeholders

* InjectorConsole implementado (Program.cs) com suporte a parâmetros PID, LogPath e DLL.
* RemoteEntry (ComHookLib) carregando hooks de UI (UiHook) e inicializando COM em modo placeholder.
* UiHook captura eventos de janela (window.create, window.show, window.foreground).
* ComHooks apenas inicializado como placeholder (sem interceptar CoCreateInstance ainda).
* Logging estruturado ativo: Jsonl, ComLogger, ComLogIpc (NamedPipe + fallback para arquivo).

Status atual:

* ✔️ Injeção confirmada
* ✔️ Hooks de UI ativos
* ✔️ Placeholder COM inicializado
* ⏸️ Hooks reais de COM ainda não instalados (aguardam etapa 2/4).

# Etapa 2/4 – Instalação dos Hooks COM

* Hooks reais instalados para interceptação das funções CoCreateInstance, CoCreateInstanceEx e CoGetClassObject.
* Logs capturam:
* CLSID e IID utilizados em cada chamada.
* Tradução de CLSID para ProgID quando disponível (Native.ProgIdFromClsidSafe).
* Flags de contexto (CLSCTX) em formato legível.
* Resultado da chamada (HRESULT) e tempo decorrido (elapsed\_ms).

Exemplos observados em ambiente de teste:

* RnaClientCore.RnaNamespace.1
* RSHMI.ParSrv.1
* HMIDiagnostics.CHMIDiagClt.1
* HMI.Connection.ConnectionServer.1
* FTAlarmSummary.FTAlarmBannerCtl.1 ✅ (relacionado diretamente ao Alarm Banner).

Status atual:

* ✔️ Injeção confirmada
* ✔️ Hooks de UI ativos (UiHook)
* ✔️ Hooks COM instalados e capturando chamadas reais
* ⏸️ Ainda sem enriquecimento semântico (tradução detalhada de HRESULTs, IIDs e classificação de objetos).

# Etapa 3/4 – Enriquecimento da Captura (CONCLUÍDA)

Objetivo: enriquecer os eventos COM com metadados legíveis e normalizados para análise.

Implementações:

* Tradução de HRESULTs para nomes legíveis (ex.: 0x80004002 → E\_NOINTERFACE; 0x00000000 → S\_OK) via ComDictionary.TryResolveHResult.
* Mapeamento de IIDs para nomes conhecidos (IUnknown, IDispatch, etc.) via ComDictionary.TryResolveIid.
* Suporte a múltiplos IIDs em CoCreateInstanceEx: campo adicional iid\_names (array) preenchido a partir de iids.
* Resolução de ProgID (T3 + T4) combinando entrada/CLSID e fallback de dicionário.
* Normalização de CLSCTX (T5) para formato canônico (ex.: INPROC\_SERVER|LOCAL\_SERVER|REMOTE\_SERVER).
* Campo kind derivado de CLSID/ProgID para classificar objetos críticos (ex.: Alarm Banner, Alarm Summary, Subscriptions).
* Enriquecimento tolerante a tipos anônimos (sem setter): somente grava quando a propriedade é gravável; caso contrário, mantém o DTO inalterado.

Evidências (Smoke de 28/08/2025):

* Ausência de erros de reflexão: não há mais “Falha no EnrichIfComEvent”.
* Eventos mostram hr\_name="S\_OK", iid\_name preenchido e iid\_names quando aplicável.
* ProgIDs resolvidos (ex.: WScript.Shell.1, Scripting.Dictionary).
* CLSCTX normalizado nos registros.

Status da etapa 3/4:

* ✔️ Enriquecimento aplicado e validado em Smoke.
* ✔️ JSONL estruturado com campos adicionais (hr\_name, iid\_name, iid\_names, progId resolvido, clsctx normalizado, kind).

# Próxima etapa 4/4 – Validação funcional no FTView

* Executar interações reais no Banner do FTView: reconhecer alarmes ativos (Ack) e executar Refresh.
* Gerar carga COM leve para avaliar estabilidade do enriquecimento.

Roteiro sugerido (PowerShell):

1..50 | % { $null = New-Object -ComObject WScript.Shell; Start-Sleep -m 30 }

1..50 | % { $null = New-Object -ComObject Scripting.Dictionary; Start-Sleep -m 30 }

# Mix opcional

1..20 | % { $null = New-Object -ComObject WScript.Shell; $d = New-Object -ComObject Scripting.Dictionary; Start-Sleep -m 50 }

Critérios de aceite:

* FTView segue responsivo durante Ack/Refresh (sem janelas modais inesperadas).
* Log JSONL sem “warn”/“error” do enriquecimento.
* Campos enriquecidos presentes e consistentes durante a carga (iid\_names, hr\_name, progId, clsctx, kind).

# Via paralela — OcxProbe (WinForms x86)

Motivação: após a etapa 4/4 inicial (Attach bem-sucedido e hooks instalados), observamos baixa emissão de novos eventos nos logs sob operações simples. Para isolar variáveis e inspecionar profundamente os tipos COM expostos pelos controles do FTAE (sem depender do cliente FTView), criamos um host WinForms x86 chamado OcxProbe.

## Objetivos da via paralela

* Explorar as interfaces, classes e eventos expostos pelo OCX (sem UI/FTView).
* Gerar RCW (Runtime Callable Wrapper) estático via aximp para inspeção offline.
* Mapear propriedades/métodos/dispids e IIDs de eventos (\_IFTAlarmSummaryEvents, \_IFTASEvents).
* Validar nomes de ProgID/CLSID e apoiar a classificação (kind) usada no enriquecimento dos logs.

## Como montamos

1. Projeto Windows Forms App (.NET Framework) — \*\*4.8\*\*, plataforma \*\*x86\*\*.

Geração do RCW com aximp:

$ocx = "C:\Program Files (x86)\Common Files\Rockwell\FTAlarmSummary.ocx"

$out = "C:\Projetos\VisualStudio\OcxProbe\libs"

"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\aximp.exe" `

"$ocx" /out:"$out\FTAlarmEventSummary"

Resultado: \*\*FTAlarmEventSummary.dll\*\* (referenciada no projeto, Embed Interop Types = False, Copy Local = True).

Hospedagem do OCX por \*\*AxHost\*\*; a UI exibe a faixa de licença (comportamento esperado fora do FTView).

## Resultados obtidos

* Dump de tipos (RCW) bem-sucedido para: Interfaces \*\*AlarmEventSummary\*\*, \*\*AlarmEventBanner\*\* e Classes \*\*AlarmEventSummaryClass\*\*, \*\*AlarmEventBannerClass\*\*.
* GUIDs confirmados: AlarmEventSummary = 1bde1371-449f-47a1-9249-5e24dd769acc; AlarmEventBanner = 42680752-632a-4535-b43d-bd85484e5184.
* Listagem completa de propriedades/métodos/eventos (com DispId) — ex.: AckSelected, LoadMessages, SelectedEvents, EventSubscriptions.
* Eventos mapeados: \*\*\_IFTAlarmSummaryEvents\*\* (Error, EventClick, EventDblClick) e \*\*\_IFTASEvents\*\* (MessageLoadBegin/Complete).
* Form WinForms abre com ambos os controles — mensagem “Failed to load control. Control can run only in Rockwell Automation products.” é esperada.
* Tentativa de assinatura genérica de todos os eventos gerou \*\*COMException 0x80040200\*\* em alguns casos; foi implementado tratamento e filtro por eventos suportados.
* Artefatos gerados pelo host para análise: \*\*rcw\_dump.txt\*\*, \*\*rcw\_run.txt\*\* (leituras simples), \*\*rcw\_events.txt\*\* (assinaturas realizadas).

## O que ainda falta nesta via

* A UI dos controles segue bloqueada por licença fora do FTView — via mantém foco em \*\*reflexão e mapeamento\*\*.
* Assinatura de eventos: restringir a conexão aos eventos presentes nas interfaces de eventos e lidar com ambientes sem publisher.
* Opcional: leitura do TypeInfo (ITypeInfo) para correlacionar DISPIDs→nomes durante a instrumentação no cliente real.

## Integração com as Etapas 1/4–4/4

\*\*Etapa 1/4 – Base de Injeção\*\*

* Usar lista de ProgIDs/CLSIDs confirmados pelo OcxProbe para validar carga inicial dos hooks.
* Adicionar verificação automática pós-injeção: presença das chaves/CLSIDs dos controles de alarme no processo.

\*\*Etapa 2/4 – Hooks COM\*\*

* Priorizar captura de \*\*CoCreateInstance/Ex\*\* e \*\*CoGetClassObject\*\* para CLSIDs do FTAlarmEventSummary.
* Ampliar para \*\*IConnectionPointContainer::FindConnectionPoint\*\* e \*\*IConnectionPoint::Advise/Unadvise\*\*, usando os IIDs mapeados (\_IFTAlarmSummaryEvents, \_IFTASEvents) para reconhecer assinaturas de eventos do banner/summary.

\*\*Etapa 3/4 – Enriquecimento (já feito)\*\*

* Usar a taxonomia do OcxProbe (propriedades/métodos/eventos) para alimentar o dicionário de nomes (hr\_name, iid\_name, kind).
* Classificar objetos críticos: \*\*Alarm Banner\*\*, \*\*Alarm Summary\*\*, \*\*Subscriptions\*\*.

\*\*Etapa 4/4 – Validação funcional no FTView\*\*

* Durante \*\*Ack/Refresh\*\* no cliente real, registrar sequências de \*\*Advise/Invoke/Unadvise\*\* e correlacionar com IIDs/DispIds conhecidos.
* Critérios: cliente responsivo; JSONL sem erros de enriquecimento; presença consistente de `iid\_names`, `hr\_name`, `progId`, `clsctx`, `kind`.

## Benefício final

Com o OcxProbe validando a superfície COM e os hooks instrumentando criação, assinatura e (opcionalmente) invocações, alcançamos \*\*controle lógico completo\*\* do fluxo entre servidor de alarmes e cliente: podemos observar quando e como o cliente se inscreve, quais eventos chegam, e como comandos (Ack, Shelve, Filter) são disparados — utilizando o Banner do FTView ou não.

## Comandos úteis (reprodução)

# Gerar RCW

$ocx = "C:\Program Files (x86)\Common Files\Rockwell\FTAlarmSummary.ocx"

$out = "C:\Projetos\VisualStudio\OcxProbe\libs"

"C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.8 Tools\aximp.exe" "$ocx" /out:"$out\FTAlarmEventSummary"

# Referenciar libs\rockwell\FTAlarmEventSummary.dll no projeto OcxProbe (Embed Interop Types=False, Copy Local=True)