

Como funciona o ddos? syn+ack é um handshake? É a única forma de ddos?

Ótimas perguntas — vou explicar de forma direta e sem enrolação.

Como funciona um DDoS (visão geral)

DDoS = **Distributed Denial of Service**. Objetivo: tornar um serviço (site, API, servidor) indisponível ao sobrecarregá-lo com tráfego ou com pedidos malformados, vindos de muitas fontes (normalmente máquinas comprometidas — *botnet*). Em vez de atacar só um IP a partir de uma máquina, usa-se uma rede de máquinas distribuídas para saturar largura de banda, recursos de conexão ou capacidade de processamento do alvo.

O que é o SYN / SYN-ACK / ACK? (o *handshake* TCP)

O TCP usa um **three-way handshake** para estabelecer uma conexão confiável:

1. **SYN** — cliente envia um pedido de conexão.
2. **SYN-ACK** — servidor responde confirmando o pedido.
3. **ACK** — cliente confirma, a conexão é estabelecida.

Então **SYN + SYN-ACK não é o handshake completo** — faz parte dele. O handshake completo é SYN → SYN-ACK → ACK.

Como um ataque pode explorar esse handshake?

Um exemplo clássico é o **SYN flood**: o atacante envia muitos pacotes SYN com IPs falsificados ou sem completar o passo final (ACK). O servidor aloca recursos (estado) esperando o ACK e, com muitos pedidos incompletos, esgota sua tabela de conexões, ficando incapaz de aceitar conexões legítimas. (Explicação conceitual — **não** instruções para realizar ataque.)

Tipos principais de DDoS (alto nível)

- **Volumétricos (floods de banda)**: geram tráfego em grande volume para saturar a largura de banda do alvo (ex.: UDP flood, ICMP flood, amplificação DNS/NTP).
- **De protocolo (state-exhaustion)**: exploram limites de protocolos/implementações — por exemplo SYN flood (TCP), TCP ACK flood, ou ataques que consomem tabelas de estado em firewalls/load balancers.
- **Aplicação (layer 7)**: parecem tráfego “legítimo” ao nível HTTP/HTTPS (ex.: muitas requisições GET/POST caras), visando esgotar CPU/threads/BD do serviço web. Difíceis de diferenciar do tráfego real.

- **Reflection / Amplification:** o atacante envia uma pequena requisição a um servidor aberto (ex.: servidor DNS) com o IP de vítima como remetente; o servidor responde com uma resposta muito maior para a vítima — amplificando o tráfego.
- **Ataques lentos (Low-and-Slow):** mantêm muitas conexões abertas enviando dados muito lentamente (ex.: Slowloris), exaurindo conexões do servidor.

SYN flood é a *única* forma de DDoS?

Não — é **apenas uma** forma entre muitas (ver lista acima). DDoS é uma família de táticas; adversários escolhem conforme objetivo, ferramentas e alvo.

Como se defender (práticas comuns — alto nível)

- **SYN cookies:** técnica TCP que evita alocar estado até receber o ACK legítimo.
- **Rate limiting** e filtros em borda (firewalls, ACLs).
- **CDNs / Anycast / balanceadores:** distribuem tráfego geograficamente e ajudam absorver picos.
- **Soluções de scrubbing / anti-DDoS comerciais** (Cloudflare, Akamai, etc.) para filtrar tráfego malicioso.
- **Filtragem de spoofing:** implementar BCP 38 / anti-spoofing na rede para reduzir IP spoofing.
- **Dimensionamento e arquitetura resiliente:** escalabilidade, timeouts curtos para conexões pendentes, caching.
- **Monitoração e playbooks:** detecção rápida e ações automatizadas.

Nota legal e ética

DDoS é crime em muitos países. Forneci só explicações conceituais e medidas defensivas. Se queres proteger um serviço teu, posso ajudar com medidas práticas e seguras (ex.: configurar rate limits, explicar SYN cookies, como usar uma CDN). Se a intenção for atacar, não posso ajudar.

Se quiser, eu:

- explico em mais detalhe **como o SYN cookie funciona** (conceito, quando ativar),
- ou faço um **checklist de mitigação** específico para um servidor web (Nginx/Apache) — sem comandos de ataque, só defesa.

Qual destes prefere?

Sim, mas na verdade eu quero proteger um serviço meu que está rodando localmente no ngrok, e quero fazer testes. Você sabe que o ngrok é um túnel seguro para o meu localhost, então quero testar se minha aplicação está segura. Estou usando a ferramenta siege para isso. O siege usa qual desses tipos?

Pensou por 24s >

Perfeito — vamos direto ao ponto.

Resposta curta

Siege é uma ferramenta de *load testing* HTTP/HTTPS — ou seja, testa o nível de aplicação (Layer 7). Ele simula múltiplos clientes fazendo requisições HTTP/HTTPS concorrentes (GET/POST, cookies, auth, etc.). Não é uma ferramenta para gerar floods de baixo nível (SYN floods, UDP floods ou ataques de amplificação); esses são ataques de rede/protocolo (layers 3/4) que normalmente usam spoofing e tráfego bruto.

joedog.org +1

O que isso significa no seu caso com ngrok

- Quando você aponta o **siege** para a URL pública do ngrok, o que ocorre é tráfego HTTP legítimo passando pelo túnel — ou seja, **ataques/simulações de aplicação (Layer 7)**. Siege completa conexões TCP normalmente (handshake) e envia requisições HTTP, então **não faz SYN floods nem spoofing de IP por padrão**.
joedog.org
- **ngrok possui limites e rate limits** (por exemplo: limites de requisições por minuto e banda no plano gratuito). Se você tentar empurrar mais tráfego do que o permitido pelo plano, o throttling ou erros virão do ngrok antes mesmo do seu serviço local sentir a carga. Portanto resultados de teste serão influenciados por esses limites. ngrok.com +1
- Além disso, executar siege a partir de **uma única máquina** simula muitos clientes vistos **a partir de um único IP** (ou poucos IPs). Isso reproduz congestionamento de aplicação (requests/sec, latência, uso de threads/DB), mas **não** reproduz um DDoS distribuído real vindo de milhares de IPs ou ataques de rede/protocolo. [Akamai](https://akamai.com)

Recomendações práticas (para testar com segurança)

1. **Monitore tudo:** CPU, memória, conexões TCP (`ss -tn`), logs da app, latência de banco.
2. **Teste em ramp-up:** comece baixo e aumente concorrência (`-c`) e duração (`-t`) — por exemplo:

```
siege -c50 -t1M 'https://<seu-subdominio-ngrok>.ngrok.io/'
```

Isso simula 50 usuários concorrentes durante 1 minuto. Interprete erros/timeouts como sinais de saturação. [joedog.org](https://www.joedog.org/)

3. **Esteja atento aos limites do ngrok** (free/hobby/pro): não ultrapasse o rate limit do plano ou você testará o throttling do ngrok, não da sua app. ngrok.com +1
4. **Teste localmente (sem túnel)** quando possível — rodar siege direto contra `localhost` (ou contra um servidor em sua rede) evita o efeito de rate-limiting do ngrok e mostra a capacidade real da sua aplicação. [Akamai](https://www.akamai.com/)
5. **Para testar maior escala** (simular DDoS distribuído) use serviços de load-testing distribuído profissionais ou várias máquinas/contêineres controlados — mas faça somente em infraestrutura que você possui e com autorização, e não contra terceiros. (Nota legal/ética.)
6. **Verifique timeouts e limites de conexões do seu servidor** (workers do webserver, pool de DB, keep-alive, timeouts) — Layer 7 esgota esses recursos antes do link de rede em muitos casos.

Resumo rápido

- **Tipo que o siege usa: Layer 7 — application-layer HTTP/HTTPS load testing** (simula muitos clientes fazendo requisições). [joedog.org](https://www.joedog.org/) +1
- **Não é:** SYN flood / UDP flood / amplification (esses são ataques de rede/protocolo).
- **Com ngrok:** cuidado com os rate limits do túnel; teste localmente quando quiser medir a capacidade real da sua app; usar siege através do ngrok pode medir comportamento da aplicação sob carga **mas** será limitado/afetado pelo túnel. ngrok.com +1

Se quiser eu já te passo:

- um **script de testes** com `siege` (ramp-up automático, logging e parsing de resultados),
- comandos para **instrumentar sua app** (métricas simples com `htop` , `ss` , logs), e

- uma **lista curta de parâmetros do ngrok** a observar (requests/min, banda, conexões).

Quer que eu gere o script de `siege` + checklist de monitoramento agora?

Sim, quero tudo isso.

Pensou por 22s >

Perfeito — vou te dar **tudo pronto**: (A) um **script de testes com siege** com ramp-up, logging e parsing dos resultados; (B) comandos e um **checklist de monitoramento** (CPU, conexões TCP, threads/workers, DB, latência); (C) parâmetros e limites do **ngrok** que você deve observar; (D) recomendações de como interpretar os resultados e ajustar a app. Vou ser direto e prático — logo abaixo.

Rápido (resumo)

- **Siege** é um *load tester* HTTP/HTTPS (Layer 7). Ele simula muitos clientes fazendo requisições reais; **não** gera SYN floods/UDP floods/amplificação. joedog.org +1
- **ngrok** aplica limites (requests/min, conexões, banda) dependendo do plano — se você usar o túnel esses limites podem mascarar a capacidade real da sua app. Teste localmente para medir o máximo real. ngrok.com +1

A — Script pronto: `siege_rampup.sh`

Salve em `~/siege_rampup.sh`, torne executável (`chmod +x siege_rampup.sh`) e rode em uma máquina de teste (não contra terceiros).

```
#!/usr/bin/env bash # siege_rampup.sh # Uso: ./siege_rampup.sh
https://meu-subdominio.ngrok.io /path/to/urls.txt URL="$1"
URLS_FILE="$2" # opcional: arquivo com várias URLs (siege -f)
OUTDIR="{3: -./siege_results}" mkdir -p "$OUTDIR" # parâmetros (ajuste conforme necessário)
MAX_CONCURRENCY=200 # máximo concorrente a atingir
STEP=20 # aumenta 20 concorrentes por step
STEP_DURATION="1M" # duração de cada step (1 minute)
DELAY=1 # delay random entre clientes (-d1)
REPS=0 # se >0 usa -r (reps), se 0 usa -t (tempo)
LOGFILE="$OUTDIR/siege.log" # arquivo global (siege --log appends here)
STDOUT_LOG="$OUTDIR/siege_stdout.log" echo "Início: $(date)" | tee -a "$LOGFILE"
current=10 while [ "$current" -le "$MAX_CONCURRENCY" ]; do
echo "=== step: concorrência=$current (duração $STEP) - $(date) ===" |
```

```
tee -a "$LOGFILE" if [ -n "$URLS_FILE" ]; then if [ "$REPS" -gt 0 ];
then siege -c"$current" -r"$REPS" -d"$DELAY" --time="$STEP" --
file="$URLS_FILE" -v --log >>"$STDOUT_LOG" 2>&1 else siege -c"$current"
-d"$DELAY" -t"$STEP" --file="$URLS_FILE" -v --log >>"$STDOUT_LOG" 2>&1
fi else if [ "$REPS" -gt 0 ]; then siege -c"$current" -r"$REPS" -
d"$DELAY" -v --log "$URL" >>"$STDOUT_LOG" 2>&1 else siege -c"$current" -
d"$DELAY" -t"$STEP" -v --log "$URL" >>"$STDOUT_LOG" 2>&1 fi fi # salvar
timestamp do fim do step echo "=== fim step: concurrency=$current -
$(date) ===" | tee -a "$LOGFILE" # Parse rápido: extrai linhas de
estatísticas do stdout log (último bloco) tail -n 200 "$STDOUT_LOG" |
awk '/Transactions:/{p=1} p{print} /Benchmarking complete/{exit}' >>
"$LOGFILE" 2>/dev/null || true # aumentar current=$((current + STEP))
sleep 5 done echo "Fim total: $(date)" | tee -a "$LOGFILE" # resumo
final: mostra as últimas estatísticas do siege echo "=== Resumo final
(últimas estatísticas do stdout) ===" tail -n 200 "$STDOUT_LOG" | sed -n
'/Transactions:/$, $p' || true
```

O que o script faz

- Faz ramp-up de concorrência (10 → MAX_CONCURRENCY em passos de STEP), cada passo por STEP_DURATION.
- Usa --log do siege para append em ~/.siegerc /var/siege.log e redireciona stdout/stderr para arquivo para capturar estatísticas. (Note: o siege --log escreve estatísticas em arquivo; redirecionamento de stdout captura as estatísticas finais impressas no terminal). joedog.org +1

Ajustes práticos

- Se quiser testar várias URLs, passe um arquivo urls.txt e o script usará --file.
- Mude MAX_CONCURRENCY, STEP, STEP_DURATION conforme o que pretende simular.

B — Comandos de monitoramento (rodar durante o teste)

Cole esses comandos em outra janela/terminal (ou crie um monitor.sh).

Resumo rápido do host (a cada 2s)

```
# CPU / memória / load htop # interativo # ou só snapshot: top -b -n1 #
estatísticas do sistema IO/CPU/mem vmstat 1 # estatísticas disk IO
iostat -x 1 # conexões TCP / estados ss -s ss -tn state established ss -
tnp | head -n 50 # mostra PID/programa para conexões TCP # ver número de
conexões por IP (útil para detectar muitos clientes únicos ou um único
IP) ss -nt | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr
| head # uso de disco/loggrow du -sh /var/log/* | sort -h # se for
```

```
nginx: status simples (se tiver stub_status) curl
http://127.0.0.1/nginx_status # se for postgres (exemplo): ver conexões
sudo -u postgres psql -c "SELECT state, count(*) FROM pg_stat_activity
GROUP BY state;"
```

Recomendado para análise de rede (se for possível)

```
# capturar tráfego por X segundos (cuidado com espaço) sudo tcpdump -i
any -w capture.pcap port 80 or port 443 or host <ngrok-ip> -c 10000
```

C — Checklist de métricas para observar (e onde agir se estiver ruim)

1. CPU alta

- Interpretação: handlers/threads saturados → aumentar workers/threads ou otimizar código.
- Ação: se Node/Python/Rails, aumente workers (process manager), use profiling, habilite cache.

2. Memória aumentando sem cair

- Possível memory leak ou objetos retidos. Gere heap dumps (Node/Java) e profile.

3. Número de conexões TCP (established / syn_recv / time_wait)

- Muitos established e server não aceita novas → aumentar max file descriptors, tune `net.core.somaxconn`, `tcp_tw_reuse`, `tcp_fin_timeout` se apropriado.

4. Latency aumentou (p95/p99)

- Indica blocking sync work: DB queries lentas, chamadas externas. Trace com APM (NewRelic, jaeger, etc).

5. Errors 5xx / timeout no app

- Ver logs da aplicação (stack traces), ver limit de threads/worker, conexões ao banco vencendo.

6. DB pool esgotado

- Ação: aumentar pool ou reduzir concorrência, otimizar queries, adicionar réplicas.

7. Keep-alive e timeouts

- Parâmetros: `keepalive_timeout` (nginx), `client_max_body_size`, `worker_connections`. Em Nginx/Apache dimensione `worker_processes` e `worker_connections`.

8. Limitação pelo ngrok

- Se ver respostas estranhas do ngrok ou `throttling/429`, provavelmente bateu limite do plano. Ver logs/usage do ngrok. ngrok.com +1

D — Parâmetros do ngrok que você deve observar (importante)

- **Requests per minute / TCP connection rate:** ngrok tem limites por plano (ex.: Free tem limite baixo; planos pagos aumentam). Se você exceder, verá `throttling` ou bloqueio. Ver página de *Pricing and Limits* para números exatos do seu plano — confira antes do teste. ngrok.com +1
- **Banda (data transfer out):** Free tem limites (ex.: 1GB). Grandes testes podem consumir rápido. ngrok.com
- **Concurrent tunnels / endpoints:** free tem restrições (3 endpoints etc). ngrok.com

Conclusão prática: se quer testar a capacidade da *sua* aplicação sem interferência do túnel, rode `siege` **direto contra** `localhost` (sem ngrok). Se precisa usar ngrok (p.ex. cliente externo acessando), use um plano que suporte o volume (ou use um túnel alternativo/servidor público que você controla). ngrok.com +1

E — Como interpretar resultados (prático)

- **Throughput (transações/s) cai enquanto latência sobe** → servidor saturado (CPU/threads/IO).
- **Erros 502/504** → backend lento / timeout; ver logs do reverse proxy/ngrok.
- **Respostas do ngrok:** se erros vindos do domínio ngrok (404/401/429) → provavelmente limite do túnel.
- **Se você roda siege a partir de 1 VM (mesmo IP)**, resultado mede concorrência de L7 vista de um ou poucos IPs. Um DDoS real é distribuído; para simular isso precisa de múltiplos agentes distribuídos (use serviços de load test distribuído com autorização).

F — Ajustes rápidos que você pode aplicar na app/nginx/stack (orde em prioridade)

1. **Ativar cache HTTP** (Varnish, CDN) para endpoints estáticos.
2. **Timeouts curtos para conexões inativas** (keepalive shorter) para liberar recursos.
3. **Aumentar pool de workers/threads** conforme CPU/cores (para CPU-bound só mais threads não resolve).
4. **Aumentar max_fds / somaxconn** no host (`sysctl net.core.somaxconn=1024, ulimit -n`).
5. **Configurar rate limiting** via nginx ou via ngrok Traffic Policy para proteger endpoints públicos. ngrok.com
6. **SYN cookies / TCP tuning** (se algum tráfego L3/L4 suspeito aparecer — mais relevante para ataques de rede).

G — Par de comandos “pronto pra rodar” (monitor + teste)

Em um terminal 1 (monitor):

```
watch -n2 'date; ss -s; ss -tn state established | wc -l; free -h; vmstat 1 2 | tail -n1'
```

Em terminal 2 (rodar teste):

```
./siege_rampup.sh https://seu-subdominio.ngrok.io # ou para testar local: ./siege_rampup.sh http://127.0.0.1:8080
```

H — Observações finais e segurança

- **Não realize testes de carga contra serviços de terceiros ou sem autorização.**
Testes de carga em serviços de terceiros podem ser considerados ataque.
- Se quiser simular um **DDoS distribuído** para hardening real (apenas em infra que você possui), use plataformas de load-testing distribuído (ex.: k6 Cloud, Gatling Enterprise, AWS distributed load) ou crie várias VMs/containers controlados — mas só na sua infra e com autorização.

- Posso: (1) gerar uma versão compacta do `monitor.sh` para coletar métricas em CSV; (2) ajustar o `siege_rampup.sh` para gerar um CSV final com p95/p99 se quiser. Quer que eu gere o `monitor.sh` + CSV parser agora?

Se quiser, já gero o `monitor.sh` e o parser de logs (CSV) automaticamente — me confirma que gere e eu mando os arquivos prontos aqui.

OK, esse é um ótimo teste para a camada de aplicação, agora elabora testes para as camadas inferiores.

Pensou por 32s >

Perfeito — vou montar **um plano completo, seguro e prático** para testar as camadas inferiores (L3/L4 — rede e transporte) **sem** te ensinar ataques reais que possam ser usados contra terceiros. Vou priorizar métodos legítimos: *ambiente controlado, geradores de tráfego e serviços de simulação autorizados*. Incluo ferramentas recomendadas, cenários de teste, métricas a medir e o que ajustar se algo falhar. Onde fizer sentido, cito referências oficiais.

1 — Regras obrigatórias (antes de qualquer teste)

1. **Só realize testes em infraestrutura que você controla e autorizou explicitamente** (seu servidor local, rede da sua casa/empresa ou um ambiente de teste na nuvem pago por você). Testes contra terceiros são ilegais. [Cloudflare Docs](#) +1
2. **Use um ambiente isolado** quando for gerar tráfego de baixo nível (por ex., uma VLAN ou rede de laboratório), para não congestionar sua rede doméstica ou o túnel ngrok.
3. **Monitore tudo e defina kill-switch**: tenha scripts/alertas que param os geradores se latência/CPU/banda ultrapassarem um limite seguro.
4. **Prefira testbeds e serviços feitos para isso** (DETERLab, TRex, ou serviços comerciais/Cloudflare que permitem simulações). [deter-project.org](#) +1

2 — Ambientes recomendados para testes (ordem de segurança/praticidade)

- **Ambiente local isolado (VMs/containers + rede virtual)** — bom para iniciar; executa geradores numa VM separada e o alvo noutra.
- **Testbed público para pesquisa (DETERLab)** — permite testes em escala maior com controle e permissão apropriados. Útil para testes distribuídos em escala.
deter-project.org
- **Ferramentas de tráfego em hardware/DPDK (TRex)** — geradores de alta performance para simular tráfego L3–L7 realisticamente. Use em seu laboratório ou numa nuvem que você controla. trex-tgn.cisco.com +1
- **Serviços comerciais / Cloud providers que suportam simulações DDoS autorizadas** — se você quer testar mitigação de um provedor (ex.: Cloudflare), use as ferramentas de simulação e processos de onboarding que eles recomendam.
[Cloudflare Docs](#)

3 — Ferramentas úteis (seguras, legítimas)

- **TRex (Cisco)** — gerador de tráfego stateful/stateless com suporte a TCP/UDP/ICMP e perfis realistas; adequado para L3/L4 stress testing em laboratório. (bom para simular SYN load *de forma controlada*). trex-tgn.cisco.com +1
- **tcpreplay** — reproduzir tráfego capturado (útil para validar comportamento com tráfego real previamente capturado). Use apenas sobre tráfego legal e em laboratório.
- **netem / tc (Linux)** — para injetar perda/latência/reordenação e testar resiliência da aplicação/roteadores.
- **DETERLab / testbeds acadêmicos** — executar experimentos distribuídos de forma ética e controlada. deter-project.org
- **Serviços de simulação autorizada (Cloudflare, fornecedores de scrubbing)** — permitem simular ataques contra zonas que você possui. [Cloudflare Docs](#) +1

Observação: eu *não* vou te passar comandos para gerar floods (SYN/UDP/ICMP) na Internet pública — isso seria potencialmente danoso. Em vez disso vou descrever como simular cada cenário **de forma segura** usando as ferramentas acima.

4 — Cenários de teste (L3/L4) — o que testar e por quê

Para cada cenário indico *objetivo*, *onde executar (ambiente)*, *ferramenta sugerida (legítima)* e *métricas a coletar*.

4.1 SYN-flood / conexão-exhaustion (simulação controlada)

- **Objetivo:** avaliar como seu servidor/firewall/load-balancer reage a muitos handshakes TCP incompletos e a tabela de conexões `syn_recv` cheia.
- **Ambiente:** laboratório isolado (VM geradora → VM alvo). Não use Internet pública/ngrok para isso.
- **Ferramenta sugerida:** TRex (ASTF/stateful) para gerar muitas tentativas de conexão de modo controlado e mensurável. trex-tgn.cisco.com
- **Métricas:** `ss -s`, contagem `SYN_RECV`, uso de memória para estruturas TCP, filas `syn backlog` (`net.core.somaxconn`), latência de novas conexões, logs do servidor, contagem de sockets.
- **O que observar:** se o servidor começa a rejeitar conexões legítimas, se timeouts aumentam, se o sistema ativa SYN cookies.

4.2 UDP flood / tráfego em pacotes por segundo (PPS)

- **Objetivo:** medir saturação de CPU/throughput de link e impacto em capacidade de processamento por pacote.
- **Ambiente:** laboratório com gerador e receptor; atenção à interface física (banda).
- **Ferramenta sugerida:** TRex (stateless) ou geradores de pacotes DPDK para PPS elevados. trex-tgn.cisco.com
- **Métricas:** PPS, utilização de CPU por interrupção, drops por kernel, `ifconfig` / `ethtool stats`, uso da fila do NIC, perda de pacotes.
- **O que observar:** se o NIC/CPU não conseguem acompanhar — pode necessitar offload (SR-IOV), limits de IRQ, ou rate-limiting.

4.3 ICMP / Echo floods e resposta de infraestrutura

- **Objetivo:** confirmar impacto de tráfego de controle (ICMP) na rede/hosts.
- **Ambiente:** laboratório; curto e controlado.
- **Ferramenta:** TRex ou replay de pacotes (tcpreplay) com PCAPs de ICMP.
- **Métricas:** perda de pacotes, latência, uso de CPU, logs de firewall, uso de banda.

4.4 Amplification / Reflection (simulação — não uso de servidores terceiros)

- **Objetivo:** entender como uma vítima veria tráfego amplificado e testar mitigação (filtragem por padrões de resposta grandes).
- **Ambiente: SIMULAÇÃO EM LAB** — não provoque servidores públicos. Em vez disso, crie servidores DNS/NTP “refletores” controlados no seu laboratório que

respondem com payload maior ao ver uma requisição pequena, e direcionam as respostas para a vítima (tudo dentro da sua rede de testes).

- **Ferramenta:** TRex + instâncias controladas de serviços reflexivos (DNS com respostas grandes em sua rede).
- **Métricas:** volume de bytes recebido, número de pacotes por segundo, utilização da banda e logs de aplicação.

5 — Métricas e como medir (o que coletar durante os testes)

- **Rede / kernel:** `ss -s`, `ss -tn`, `ip -s link`, `ethtool -S <iface>`, `sar -n DEV`, `ifstat`
- **CPU / Interrupts:** `top` / `htop`, `mpstat`, `vmstat 1`, `cat /proc/interrupts`
- **NIC/driver:** `ethtool -S`, verificar offload (GRO/TSO), número de queues, rx/tx drops
- **Aplicação:** latências p50/p95/p99, erros 5xx, threads/processes ocupados, pool DB (conexões abertas)
- **Firewall / ACLs:** logs do iptables/nftables, counters de regras de rate-limit
- **Sistema:** `dmesg` para mensagens do kernel sobre drops/overruns

6 — O que ajustar após observar um problema (prioridade prática)

- **Se tabelas de conexão se encham (SYN_RECV):** habilite/ajuste **SYN cookies**, aumente `somaxconn`, tune `tcp_max_syn_backlog`, e avalie timeouts. (Firewall também pode filtrar por taxa.)
- **Se PPS alto satura CPU:** mover processamento para NIC offload, aumentar capacidade da máquina, reduzir trabalho por pacote (filtrar na borda).
- **Se banda saturada:** usar rate-limiting na borda, CDN, anycast, scrubbing providers. cloudflare.com
- **Se reflexão/amplificação é o vetor esperado:** bloquear/filtrar portas UDP desnecessárias (ex.: NTP/DNS) na borda e aplicar ACLs de resposta.
- **Sempre:** implementar monitoração e alertas (thresholds p95 latency, %CPU, drops).

7 — Plano de testes sugerido (passos concretos — alto nível)

1. Preparação

- Crie 2 VMs/containers em uma rede isolada: gerador e alvo .
- Configure coleta de métricas (prometheus/exporters, ou scripts que colem ss , vmstat , ethtool).

2. Teste baseline

- Meça tráfego normal (sem stress).

3. SYN test (controlado)

- Use TRex para gerar conexões TCP incrementais (ramp-up) e observe syn_recv , latência de conexão, e logs.

4. UDP/PPS test

- Gere pacotes UDP em PPS crescente e observe drops, CPU e NIC stats.

5. ICMP / controle

- Envie rajadas de ICMP curtas e monitore impacto.

6. Amplification (SIMULADO)

- Configure um “reflektor” DNS/NTP na sua rede de teste que amplifica respostas e direcione as respostas para alvo para observar impacto. **Não** use servidores públicos.

7. Escalonamento e repeat

- Aumente gradualmente até ver saturação; documente thresholds e comportamentos.

8. Relatório

- Registrar throughput máximo, PPS limite, pontos de falha e recomendações.

8 — Recursos e referências (para leitura e uso seguro)

- TRex — open-source traffic generator (docs & GitHub). trex-tgn.cisco.com +1
- DETERLab — testbed para experimentação controlada em cibersegurança. deter-project.org
- OWASP — cheat sheet / guia sobre DoS (classificação por camadas e vetores). [cheatsheetserie...](https://cheatsheetseries.owasp.org/)
- Cloudflare — como simular DDoS de forma autorizada e guias de proteção. [Cloudflare Docs](https://cloudflare.com/docs) +1

9 — Posso automatizar isto para você (o que eu ofereço agora)

Posso gerar **algumas entregas concretas** (dentro da linha segura/ética):

1. **Diagrama de rede + playbook de montagem do laboratório** (lista de VMs, interfaces, limites).
2. **Perfil TRex (ASTF) de exemplo em formato seguro** para simular conexões TCP/UDP *numa rede de laboratório* (não conterá instruções para atacar Internet pública).
3. **Templates de coleta de métricas** (scripts que coletam `ss`, `vmstat`, `ethtool` periodicamente e geram CSV).
4. **Checklist de interpretação e relatório em PDF/Markdown** com os principais indicadores e thresholds.

Diz qual(is) desses itens quer que eu gere agora (vou colocar tudo pronto e seguro — por exemplo, o perfil TRex será marcado explicitamente para uso **apenas** em rede de testes que você controla). Não preciso de mais informações para começar — eu já preparo as coisas com parâmetros razoáveis que você poderá ajustar.

No momento a aplicação está num ambiente Linux, e eu só tenho acesso ao Ubuntu para realizar os testes, então forneça opções com ferramentas de linha de comando. Podem ser também para a camada 7, mas outras formas, para que eu possa testar tudo que estiver ao meu alcance.

Perfeito — vou entregar **um kit prático CLI** (tudo para rodar num Ubuntu) para você testar camadas 7, 4 e 3 *no seu ambiente controlado*. Vou incluir:

- comandos e **scripts prontos** para executar testes (ramp-up, bursts, throughput, PPS, SYN-rate) usando só ferramentas de linha de comando;
- comandos de **monitoramento** para rodar em paralelo e coletar métricas;
- **como usar tc/netem** para emular rede ruim;
- **táticas de proteção imediata** (iptables/nft/sysctl) para experimentar mitigação;
- avisos/boas práticas de segurança (rodar só em infra que você controla).

Importante: tudo aqui pressupõe que você tem autorização para testar o alvo (seu host local/VM). Não use estas técnicas contra terceiros. Vou sugerir ferramentas que já existem nos repositórios do Ubuntu e outras fáceis de instalar.

1 — Instalação rápida das ferramentas úteis

```
sudo apt update sudo apt install -y iperf3 hping3 nmap nping tcpreplay
tcptraceroute iproute2 iptables nftables conntrack tcpdump tshark iperf
net-tools ethtool sysstat bc jq curl moreutils # ferramentas L7 sudo apt
install -y apache2-utils # ab # wrk, siege e vegeta (possivelmente via
release binary) # Siege: sudo apt install -y siege # wrk: build from
source or download binary; vegeta: go install or apt
```

2 — Monitoramento (rodar em outra janela/terminal)

Cole isto num arquivo `monitor.sh` e rode em paralelo (`chmod +x monitor.sh && ./monitor.sh`):

```
#!/usr/bin/env bash OUTDIR=./monitor_$(date +%Y%m%d_%H%M%S) mkdir -p
"$OUTDIR" echo "Logging to $OUTDIR" # snapshot inicial date >
"$OUTDIR"/start.txt uname -a >> "$OUTDIR"/start.txt # loop de coleta
(ajuste duration/interval) for i in {1..3600}; do ts=$(date +%s) echo
"=== $ts ===" >> "$OUTDIR"/sysstats.log # ss summary ss -s >>
"$OUTDIR"/sysstats.log ss -tn state established | wc -l >>
"$OUTDIR"/conn_count.log ss -tn | awk '{print $1,$2,$3,$4,$5}' >>
"$OUTDIR"/ss_snapshot.log # CPU/mem vmstat 1 2 | tail -n1 >>
"$OUTDIR"/vmstat.log top -b -n1 | head -n20 >> "$OUTDIR"/top.log free -h
>> "$OUTDIR"/mem.log # ifstat for network throughput ifstat -t 1 1 >>
"$OUTDIR"/ifstat.log # ethtool stats (iface eth0 or adjust) ip -o -4
addr show | awk '{print $2}' | head -n1 > /tmp/ifname IFACE=$(cat
/tmp/ifname) if [ -n "$IFACE" ]; then ethtool -S "$IFACE" 2>/dev/null |
head -n50 >> "$OUTDIR"/ethtool.log || true fi # interrupts cat
/proc/interrupts | head -n20 >> "$OUTDIR"/interrupts.log sleep 5 done
```

3 — Testes de camada 7 (HTTP / aplicação) — alternativas a Siege

1. ab (ApacheBench) — simples:

```
ab -n 10000 -c 200 https://127.0.0.1:8443/endpoint # -n total requests,
-c concurrency
```

2. wrk (mais moderno, suporte a scripts Lua):

Exemplo de ramp-up simples (necessita instalar wrk):

```
wrk -t4 -c200 -d60s https://127.0.0.1:8443/endpoint # -t threads, -c
```



```
connections, -d duration
```

3. **vegeta** (attack + report):

```
# instalar vegeta (ex.: go install or binary) echo "GET
https://127.0.0.1:8443/endpoint" > targets.txt vegeta attack -
duration=1m -rate=200 -connections=50 -targets=targets.txt | vegeta
report vegeta attack -duration=2m -rate=500 -targets=targets.txt |
vegeta report -type=json > vegeta_report.json
```

4. **curl em loop** (para testes básicos):

```
for i in $(seq 1 1000); do curl -s -o /dev/null -w "%{http_code} %
{time_total}\n" https://127.0.0.1:8443/endpoint & done wait
```

4 — Testes de camada 4 / 3 — ferramentas e scripts (Ubuntu-only)

4.1 Throughput (banda) — **iperf3**

Rode no alvo:

```
iperf3 -s
```

No gerador (mesma máquina ou outra VM na sua LAN):

```
iperf3 -c <ALVO_IP> -P 10 -t 60 # -P parallel streams, -t seconds
```

Para UDP (teste de banda e perda):

```
iperf3 -c <ALVO_IP> -u -b 100M -t 60 # -u UDP, -b bandwidth target
```

4.2 PPS / pacotes por segundo e testes TCP SYN (controlado) — **nping** / **hping3**

nping (do pacote nmap) pode gerar TCP SYNs ou TCP connections sem spoofing:

```
# SYN flood-ish controlado (não use na Internet pública sem permissão)
nping --tcp -p 80 --rate 1000 --flags SYN <ALVO_IP> # --rate em pps
```

hping3 (muito poderoso — use apenas em lab):

- SYNs a uma taxa controlada:

```
sudo hping3 -S -p 80 --flood --rand-source <ALVO_IP> # NÃO recomendado
com --rand-source fora de lab (spoof). Use sem --rand-source: sudo
```

```
hping3 -S -p 80 --flood <ALVO_IP> # Para taxa controlada: sudo hping3 -S
-p 80 --interval u1000 <ALVO_IP> # --interval u1000 envia 1000
microsegundos entre pacotes (~1000pps)
```

Notas: `--flood` ignorará delays — cuidado. Prefira usar `--interval` para ramp-up controlado. Execute em VM separada que você controla.

4.3 Reproduzir tráfego capturado — tcpreplay

Se você tem um pcap de tráfego legítimo:

```
sudo tcpreplay --intf1=eth0 capture.pcap # controle de speed: sudo
tcpreplay --intf1=eth0 --pps=1000 capture.pcap
```

4.4 ICMP bursts

```
# ping flood controlado (use local target) ping -f -s 1200 <ALVO_IP> #
flood (root) # ou taxa controlada: ping -i 0.01 -s 1200 <ALVO_IP> #
0.01s interval
```

4.5 Emular rede ruim (loss, delay, re-ordering) — tc/netem

Testar comportamento sob perda/latência:

```
# aplicar para interface eth0 (faça em VM/namespace) sudo tc qdisc add
dev eth0 root netem delay 100ms loss 2% duplicate 0.1% # remover: sudo
tc qdisc del dev eth0 root netem
```

5 — Scripts prontos para ramp-up (Layer4 SYN-rate) — syn_ramp.sh

Salve e rode em máquina geradora (ajuste ALVO_IP):

```
#!/usr/bin/env bash ALVO="$1" MAX_PPS="${2:-2000}" # máximo pps
STEP="${3:-200}" # incremento pps DURATION="${4:-30}" # segundos por
step if [ -z "$ALVO" ]; then echo "Uso: $0 <ALVO_IP> [MAX_PPS] [STEP]
[DURATION]" exit 1 fi pps=200 while [ "$pps" -le "$MAX_PPS" ]; do echo
"Rodando hping3: pps=$pps duration=${DURATION}s" # usa intervalo
calculado: interval_us = 1e6/pps interval_us=$((1000000/pps)) sudo
timeout "${DURATION}s" hping3 -S -p 80 --interval u${interval_us}
"$ALVO" > /dev/null 2>&1 || true echo "Fim step pps=$pps" pps=$((pps +
STEP)) sleep 3 done echo "Terminado"
```

Aviso: use este script **somente** em VM/host que você controla e não em rede pública sem autorização.

6 — Coleta de pcap durante o teste

```
sudo tcpdump -i any host <ALVO_IP> -w run_capture_$(date +%s).pcap #
limitar por pacotes: sudo tcpdump -i any host <ALVO_IP> -c 100000 -w
capture.pcap
```

Analise com `tshark` / `wireshark`.

7 — Proteção e mitigação (comandos rápidos que pode ativar durante testes)

Ativar SYN cookies e tunar backlog

```
# habilitar SYN cookies sudo sysctl -w net.ipv4.tcp_syncookies=1 #
aumentar backlog sudo sysctl -w net.core.somaxconn=4096 sudo sysctl -w
net.ipv4.tcp_max_syn_backlog=4096 # persistir em /etc/sysctl.conf se
quiser
```

Limitar conexões por IP com nftables (exemplo)

```
# exemplo simples nftables (ajuste para nft-based systems) sudo nft add
table inet myfilter sudo nft 'add chain inet myfilter input { type
filter hook input priority 0; }' # limitar 50 conexões por minuto por IP
sudo nft add rule inet myfilter input ip saddr . tcp dport 80 meter
flood { ip saddr limit rate over 50/minute } drop # alternativa usando
iptables connlimit: sudo iptables -A INPUT -p tcp --dport 80 -m
connlimit --connlimit-above 100 -j REJECT --reject-with tcp-reset
```

Rate-limit por IP com iptables (exemplo de token bucket)

```
sudo iptables -N RATE_LIMIT sudo iptables -A INPUT -p tcp --dport 80 -m
conntrack --ctstate NEW -j RATE_LIMIT sudo iptables -A RATE_LIMIT -m
limit --limit 25/minute --limit-burst 100 -j RETURN sudo iptables -A
RATE_LIMIT -j DROP
```

8 — Métricas alvo e indicadores de falha (o que observar)

- **TCP state:** `ss -s` e contagem de `SYN_RECV`, `TIME-WAIT`, `ESTABLISHED`.
- **Conexões por IP:** `ss -nt | awk '{print $5}' | cut -d: -f1 | sort | uniq -c | sort -nr | head`.
- **Latência:** p50/p95/p99 de respostas HTTP (`vegeta report / wrk`).

- **PPS e drops NIC:** `cat /sys/class/net/eth0/statistics/rx_dropped` e `ethtool stats`.
- **CPU por softirq/irq:** `cat /proc/interrupts` e `mpstat -P ALL 1`.
- **Kernel messages:** `dmesg` por overruns/drops.

9 — Testes adicionais de aplicação em condições de rede ruim

- Combine `tc netem` com `siege / wrk` para avaliar UX sob latência e perda:

```
sudo tc qdisc add dev lo root netem delay 200ms loss 5% # rodar  
siege/wrk # depois remover sudo tc qdisc del dev lo root netem
```

10 — Relatórios simples (parse rápido de outputs)

- `iperf3` já produz resumo.
- `vegeta report -type=json` gera JSON que você pode processar com `jq` para p95/p99.
- Exemplo parse p95 com `vegeta`:

```
cat vegeta_report.json | jq '.latencies | {p50:.p50, p95:.p95,  
p99:.p99}'
```

11 — Plano de execução sugerido (passo-a-passo)

1. Execute `monitor.sh` (coleta contínua).
2. Testes L7: rode `siege / wrk / vegeta` localmente contra `localhost` (sem ngrok) para medir limite real da app. Analise CPU, latência, erros 5xx.
3. Testes banda: rode `iperf3` entre duas VMs (mesma máquina se necessário, usando loopback/bridge).
4. Testes PPS / SYN-rate: rode `syn_ramp.sh` em VM separada apontando para sua app; observe `ss -s` e `dmesg`. Comece com PPS baixos (200) e suba aos poucos. Não use `--rand-source /spoofing`.
5. Testes com `tc` para ver comportamento sob perda/latência.
6. Ative contramedidas (SYN cookies, nftables/iptables) e repita os testes para ver ganhos.

12 — Exemplos práticos de análise (o que te diz que precisa ajustar)

- Se `ss -s` mostra `syn_recv` alto e novos clientes não conseguem conectar → aumentar `tcp_max_syn_backlog` e habilitar `tcp_syncookies`.
- Se NIC drops aumentam com PPS alto → verificar IRQ/CPU per-core, RSS, ajustar `ethtool -L` (RX/TX queues) e offloads.
- Se latência p95 explode antes do máximo de CPU → problema de blocking IO / DB; profile a app.

Se quiser eu já:

- **Gero os scripts** `syn_ramp.sh`, `monitor.sh` (já tive acima) e um `l7_ramp.sh` com parâmetros padrões prontos para executar (posso colar aqui novamente), ou
- **Gero um checklist em Markdown** pronto para imprimir, ou
- **Crio um pequeno playbook** com comandos de mitigação / rollback rápido.

Qual formato prefere agora? Posso também colar aqui um `l7_ramp.sh` finalizado e um `collect_csv.sh` que transforma os logs do `monitor.sh` em CSVs simples.