プロキシサーバーの禁止に関する分析

プロキシサーバーの API は GFW のブロックを回避できるか？

私は Shadowsocks インスタンス上で以下のコードを使って簡単なサーバーを実行しています：

```
from flask import Flask, jsonify
from flask_cors import CORS
import subprocess

app = Flask(__name__)
CORS(app)  # すべてのルートで CORS を有効にする

@app.route('/bandwidth', methods=['GET'])
def get_bandwidth():
    # vnstat コマンドを実行して eth0 の 5 分間隔のトラフィック統計を取得
    result = subprocess.run(['vnstat', '-i', 'eth0', '-5', '--json'], capture_output=True, text=True)
    data = result.stdout

    # 取得したデータを JSON レスポンスとして返す
    return jsonify(data)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

そして、以下のように nginx を使ってポート 443 を提供しています：

```
server {
    listen 443 ssl;
    server_name www.some-domain.xyz;

    ssl_certificate /etc/letsencrypt/live/www.some-domain.xyz/fullchain.pem; # managed by
    # ...
    location / {

        proxy_pass http://127.0.0.1:5000/;
        # ...
```

```
    }
}
```

このサーバープログラムはネットワークデータを提供し、私はこのサーバーをプロキシサーバーとして使用し、ネットワークデータを使ってブログに自分のオンラインステータスを表示しています。

興味深いことに、このサーバーは数日間、Great Firewall（GFW）やその他のネットワーク制御システムによってブロックされていません。通常、私が設定したプロキシサーバーは 1 日か 2 日でブロックされます。このサーバーは 51939 のようなポートで Shadowsocks プログラムを実行しているため、Shadowsocks のトラフィックと通常の API トラフィックが混在しています。この混在により、GFW はこのサーバーが専用のプロキシではなく通常のサーバーであると認識し、IP をブロックしないようです。

この観察は興味深いものです。GFW は特定のロジックを使ってプロキシトラフィックと通常のトラフィックを区別しているようです。Twitter や YouTube のような多くのウェブサイトは中国でブロックされていますが、多くの外国のウェブサイト—例えば国際的な大学や企業のウェブサイト—はアクセス可能です。

これは、GFW が通常の HTTP/HTTPS トラフィックとプロキシ関連のトラフィックを区別するルールに基づいて動作していることを示唆しています。両方のタイプのトラフィックを処理するサーバーはブロックを回避するようですが、プロキシトラフィックのみを処理するサーバーはブロックされる可能性が高いです。

一つ疑問なのは、GFW がブロックのためにデータを蓄積する時間範囲が 1 日なのか 1 時間なのかということです。この時間範囲内で、トラフィックがプロキシからのものであるかどうかを検出します。もしそうであれば、サーバーの IP がブロックされます。

私はよく自分のブログを訪れて書いた内容を確認しますが、今後数週間はブログ記事を書くのではなく他のタスクに集中する予定です。これにより、ポート 443 を通じて `bandwidth` API にアクセスする回数が減ります。もし再びブロックされることがわかったら、この API に定期的にアクセスするプログラムを書いて GFW を欺くべきでしょう。

以下は、テキストの構造と明瞭さを改善したバージョンです：

Great Firewall（GFW）の仕組み

ステップ 1: リクエストのロギング

```
import time
```

```
# リクエストデータを保存するデータベース
request_log = []

# リクエストをログに記録する関数
def log_request(source_ip, target_ip, target_port, body):
    request_log.append({
        'source_ip': source_ip,
        'target_ip': target_ip,
        'target_port': target_port,
        'body': body,
        'timestamp': time.time()
    })
```

`log_request` 関数は、送信元 IP、ターゲット IP、ターゲットポート、リクエストボディ、タイムスタンプなどの重要な情報を含む着信リクエストを記録します。

ステップ 2: IP のチェックとブロック

```
# リクエストをチェックし、IP をブロックする関数
def check_and_ban_ips():
    banned_ips = set()

    # すべてのログに記録されたリクエストを繰り返し処理
    for request in request_log:
        if is_illegal(request):
            banned_ips.add(request['target_ip'])
        else:
            banned_ips.discard(request['target_ip'])

    # 識別されたすべての IP にブロックを適用
    ban_ips(banned_ips)
```

`check_and_ban_ips` 関数は、すべてのログに記録されたリクエストを繰り返し処理し、違法な活動に関連する IP を識別してブロックします。

ステップ 3: リクエストが違法かどうかを定義

```
# リクエストが違法かどうかをチェックする関数
def is_illegal(request):
    # 実際の違法リクエストチェックロジックのプレースホルダー
    # 例えば、リクエストボディやターゲットをチェック
    return "illegal" in request['body']
```

ここで、`is_illegal` はリクエストボディに「illegal」という単語が含まれているかどうかをチェックします。これは、違法な活動を構成するものに応じて、より洗練されたロジックに拡張できます。

ステップ 4: 識別された IP をブロック

```
# IP リストをブロックする関数
def ban_ips(ip_set):
    for ip in ip_set:
        print(f"Banning IP: {ip}")
```

違法な IP が識別されると、`ban_ips` 関数はそれらの IP をブロックします（実際のシステムでは、それらをブロックする可能性があります）。

ステップ 5: 80% の違法リクエストに基づく IP のチェックとブロックの代替方法

```
# 80% 以上の違法リクエストに基づいて IP をチェックし、ブロックする関数
def check_and_ban_ips():
    banned_ips = set()
    illegal_count = 0
    total_requests = 0

    # すべてのログに記録されたリクエストを繰り返し処理
    for request in request_log:
        total_requests += 1
        if is_illegal(request):
            illegal_count += 1

    # リクエストの 80% 以上が違法であれば、それらの IP をブロック
```

```
    if total_requests > 0 and (illegal_count / total_requests) >= 0.8:
        for request in request_log:
            if is_illegal(request):
                banned_ips.add(request['target_ip'])

    # 識別されたすべての IP にブロックを適用
    ban_ips(banned_ips)
```

この代替方法は、IP がブロックされるべきかどうかを違法リクエストの割合に基づいて評価します。IP からのリクエストの 80% 以上が違法であれば、その IP はブロックされます。

ステップ 6: 違法リクエストチェックの強化（例：Shadowsocks と Trojan プロトコルの検出）

```
def is_illegal(request):
    # リクエストが Shadowsocks プロトコルを使用しているかチェック（ボディにバイナリのようなデータが含
    if request['target_port'] == 443:
        if is_trojan(request):
            return True
    elif is_shadowsocks(request):
        return True
    return False
```

`is_illegal` 関数は、Shadowsocks や Trojan のような特定のプロトコルもチェックするようになりました：- **Shadowsocks**: リクエストボディに暗号化されたデータやバイナリのようなデータが含まれているかどうかをチェックするかもしれません。- **Trojan**: リクエストがポート 443（HTTPS）経由で送信され、特定のパターン（例：Trojan トラフィックの特性）に一致する場合、違法としてフラグが立てられます。

ステップ 7: 合法リクエストの例

例えば、`GET https://some-domain.xyz/bandwidth` のようなリクエストは確実に合法であり、ブロックメカニズムをトリガーしません。

ステップ 8: プロキシサーバーのトラフィック特性

プロキシサーバーは、通常のウェブサーバーや API サーバーとは非常に異なるトラフィック特性を持っています。GFW は、通常のウェブサーバートラフィックとプロキシサーバートラフィッ

クを区別する必要があり、これらは完全に異なるように見えることがあります。

ステップ 9: スマート検出のための機械学習と AI モデル

インターネットを通過するさまざまなリクエストとレスポンスの範囲を考えると、GFW は AI や機械学習モデルを使用してトラフィックパターンを分析し、違法な行動をインテリジェントに検出する可能性があります。システムをさまざまなタイプのトラフィックでトレーニングし、高度な技術を使用することで、観察されたパターンに基づいてトラフィックをより効果的にブロックまたはフィルタリングできます。

更新

私の努力にもかかわらず、私のプロキシサーバーは引き続きブロックされています。これを緩和するために、Digital Ocean の逆 IP 機能を使用して、ブロックが発生するたびに迅速に新しい IP アドレスを割り当てる回避策を実装しました。