Exposing the Great Firewall's Dynamic Blocking of Fully Encrypted Traffic

Authors: Anonymous, Kevin Bock, Jackson Sippe, Shelikhoo, David Fifield, Eric Wustrow, Dave Levin, Amir Houmansadr

Since as early as November 6, 2021, numerous users reported the blocking of their servers running Shadowsocks and VMess+TCP. Outline lead developer Vinicius also reported "a drop in the opt-in Outline usage metrics in China starting on November 8". The start of this blocking coincides with the Sixth Plenary Session of the 19th CPC Central Committee (中国共产党第十九届中央委员会第六次全体会议), which was held from November 8, 2021 to November 11, 2021.

On November 14, 2021, we confirmed that the Great Firewall (GFW) of China is now able to inspect and dynamically block any seemingly random traffic in real time. This capability potentially affects many censorship circumvention protocols that use encryption to appear random, including (but not limited to) VMess+TCP, Obfs4, and many variants of Shadowsocks. The censor strategically applies this censorship only against connections from China to certain popular VPS providers, possibly to mitigate over-blocking caused by false positives in traffic classification.

In this report, we demonstrate how the GFW identifies and blocks seemingly random traffic and we share code that led to our conclusions. We then discuss the implication of this blocking incident.

To maintain reproducibility and promote transparency, we include code in-line throughout this article to help other researchers reproduce our findings. Note that this code is ***explicitly designed*** to trigger (or subvert) this new censorship system, so understand the risks before running any of the below examples yourself.

## Summary

* The GFW can now dynamically block any seemingly random traffic in real-time based on passive traffic analysis without relying on its well-known active probing infrastructure.
* The blocking only targets connections from China to a few popular VPS providers outside of China, including Vultr, AlibabaCloud (Hong Kong and Singapore), and Digital Ocean (San Francisco, New York City). Amazon Lightsail, EC2, and Oracle Cloud are reportedly not affected.
* In many cases, after the TCP handshake, a single data packet containing only 1 byte of payload from client to server is sufficient to trigger blocking.

* Only TCP traffic can trigger and is affected by the blocking. UDP traffic cannot trigger and is not affected by the blocking.
* Once triggered, the GFW drops all **TCP** packets from the client to the server having the same (client IP, server IP, server port) for 120 to 180 seconds. Packets from the server to the client are not dropped.
* The blocking can happen on any port from 1 to 65535.
* While the traffic classification appears to be deterministic, the blocking is probabilistic.

## Background ##

Tschantz et al. divide approaches to censorship circumvention traffic into two types: *steganographic* and *polymorphic* (see Section V and Table 3 of their paper). The goal of *steganography* is to make circumvention traffic look like allowed traffic; the goal of *polymorphism* is to make circumvention traffic not look like forbidden traffic.

A common way to achieve polymorphism is to fully encrypt the traffic: since fully encrypted traffic presents no plaintext or fixed structures at all, the censor cannot simply identify such traffic with regular expression rules. This is the approach used by Shadowsocks, VMess+TCP, Obfs4 and many other censorship circumvention tools. In comparison, most of the protocols that offer encryption, such as TLS, still leave various framing fields unencrypted and therefore can be easily identified; circumvention tools encrypt or otherwise obfuscate these fields.

Fully encrypted traffic is often referred to as "looks like nothing", or misunderstood as "having no characteristics"; however, a more accurate description would be *"looks like random"*. In fact, such traffic does have many characteristics:

* Fully encrypted traffic is indistinguishable from random. One may therefore refer to it as (seemingly) random traffic.
* The data stream possesses *high entropy, homogeneously throughout the entire connection*, starting from the bytes of the first data packets.
* When no packet length obfuscation is implemented, the length of each circumvention packet is equal to `a fixed header length` + `the length of the payload of the proxied packet`.

## How do we know blocking is happening? ##

We sent traffic between hosts inside and outside of China. We captured and compared traffic on both endpoints to identify any dropped packets. All experiments were conducted between November 14 and December 8, 2021.

At first, we set up our own Shadowsocks client and server based on this tutorial. We then used an automated script to generate traffic and proxy it through Shadowsocks. The traffic was generated by using curl to visit `http://example.com` and `https://example.com` every 5 seconds. We found that the connections got blocked and unblocked periodically soon after the script started.

Aided by these two observations by Alice et al., we quickly narrowed down sufficient conditions to trigger blocking:

* "[T]he byte streams sent between Shadowsocks clients and servers are, by design, indistinguishable from random" (see Section 4.1 of their paper);
* and "[a]fter a TCP handshake, a single data packet from client to server suffices to trigger active probes" (see Section 4.2 of their paper).

To test this, we set up “sink servers” that complete TCP handshakes, but never send data back to the client. We then repeatedly opened connections, sent random data to the server, and closed the connection. Within a few connections, we observed blocking.

## How does the blocking work? ##

### How can we trigger the blocking?

We observe that a single data packet from client to newly deployed server is sometimes sufficient to trigger the GFW. Often though, we do not observe blocking unless we repeatedly make connections and send random-looking data.

Reproduce it:

One can try reproducing the blocking by sending traffic from China to open ports to a server within an affected data center:

1. In one terminal, start TCP-pinging the port with Nping:

```sh
nping -4 -c 0 --tcp-connect $SERVER_IP -p $PORT
```

2. In another terminal, send 180 bytes of random data to the port (it may not trigger blocking on the first try; you may need to repeat this step a few times to get the port blocked):

```sh
head -c180 /dev/urandom | nc -vn $SERVER_IP $PORT
```

3. The `nping` log at the client shows that the port was blocked right after sending the random data at 147s:

~~~
```
...
SENT (144.3590s) Starting TCP Handshake > REDACTED
RCVD (144.5365s) Handshake with REDACTED completed
SENT (145.3615s) Starting TCP Handshake > REDACTED
RCVD (145.5318s) Handshake with REDACTED completed
SENT (146.3639s) Starting TCP Handshake > REDACTED
RCVD (146.5410s) Handshake with REDACTED completed
SENT (147.3660s) Starting TCP Handshake > REDACTED
SENT (148.3671s) Starting TCP Handshake > REDACTED
SENT (149.3693s) Starting TCP Handshake > REDACTED
SENT (150.3715s) Starting TCP Handshake > REDACTED
SENT (151.3736s) Starting TCP Handshake > REDACTED
```
~~~

### No active probing is required before blocking

The GFW makes its blocking decision based purely on passive traffic analysis, without relying on its well-known active probing infrastructure. We know this because the GFW had not sent any active probes to the server before the connection was already blocked.

We want to emphasize that this finding does not mean that defenses against active probing are not necessary or not important anymore. On the contrary, it is because of efforts from Shadowsocks-libev, Outline, and many other censorship circumvention tool developers who improved defenses against active probing that forces the censor to strategically take this approach.

Despite this new censorship system, we confirm that the GFW still sends active probes to servers. This evidence warns us that the censor still attempts to accurately identify circumvention servers using active probing when possible.

### The same payload does not always trigger the blocking on the first try

We find that while the traffic classification appears to be deterministic, the blocking is probabilistic. That is, sending a payload that once triggered the blocking does not always trigger the blocking every time; however, repetitively sending it will eventually trigger the blocking. On the contrary, sending a payload that *never* triggers the blocking for hundreds of times will not trigger the blocking.

Reproduce it:

~~~
```sh
# This command triggered blocking. Record the payload in payload file:
head -c180 /dev/urandom | tee payload | netcat -vn $SERVER_IP
$SERVER_PORT_1
~~~

~~~
# Send the same payload again, but it does not always trigger blocking
after being replayed:
cat payload | netcat -vn $SERVER_IP $SERVER_PORT_2
```
~~~

### Blocking is done by dropping packets from client to server

We captured and compared the packets sent and received on both endpoints. We found that blocking is implemented by dropping packets from the client to the servers. Packets sent by the server are not blocked and are still received at the client.

### Only connections from China to a few well-known hosting services outside of China are affected

We have only been able to trigger the blocking when the servers are hosted by one of a few well-known hosting services outside of China. This finding aligns with many user reports, where users could use Shadowsocks servers deployed on Amazon Lightsail, EC2, Oracle Cloud and some "off-brand" VPS providers, but not on AlibabaCloud (Hong Kong and Singapore), or Digital Ocean (San Francisco, New York City).

In particular, we conducted the following testing:

1. Sending random data from a host inside of China to a well-known host provider outside of China: blocking triggered.
2. Using exactly the same pair of hosts in 1, but sending random data from outside-in this time: blocking not triggered.
3. Sending random data from a host inside of China to the open ports of some foreign websites: blocking not triggered.
4. Sending random data from a well-known host provider outside of China to the open ports of some Chinese websites: blocking not triggered.

We also conducted tests from different vantage points within China to different foreign datacenters to confirm our findings.

### A complete TCP handshake is necessary to trigger the blocking

Sending a `SYN` packet followed by a `PSH+ACK` packet containing random data (without the server completing its end of the handshake) is not sufficient to trigger blocking. The blocking is thus harder to exploit for residual censorship attacks.

Reproduce it:

We tested this with the `test_handshake.py` script, located in the `handshake_tests` folder:

~~~
```python3
#!/usr/bin/env python3
"""
A helper script to test if the censorship system can be triggered
with or without a 3-way handshake.
"""

import sys
import os
import time

from scapy.all import *

dst_ip = sys.argv[1]
dport = int(sys.argv[2])
need_syn_ack = int(sys.argv[3])

seqno = random.randint(1000, 100000)
sport = random.randint(10000, 20000)
syn = IP(dst=dst_ip)/TCP(dport=dport, flags="S", seq=seqno, sport=sport)

ackno = random.randint(1000, 100000)
if need_syn_ack:
    synack = sr1(syn)
    ackno = synack[TCP].seq
else:
    send(syn)
    time.sleep(0.5)

pshack = IP(dst=dst_ip)/TCP(dport=dport, flags="PA", seq=seqno + 1,
ack=ackno+1, sport=sport)/Raw(os.urandom(200))
send(pshack)
~~~

~~~
```
~~~

Specifically, we use the following command to 1) send a SYN to an *open* port of the server; 2) wait until we receive a `SYN+ACK` from the server; 3) and then send a `PSH+ACK` data packet with 200 bytes random payload. We confirm that it could successfully trigger the blocking:

~~~
```sh
sudo ./test_handshake.py $DST_IP $DST_PORT 1
```
~~~

We then use the following command to 1) send a SYN to *filtered* port of the server; 2) since the filtered port will not send any `SYN+ACK` or `RST` packet, we simply wait for 0.5 seconds; 3) and then send a `PSH+ACK` data packet with 200 bytes random payload. We confirm that it could **not** trigger the blocking:

~~~
```sh
sudo ./test_handshake.py $DST_IP $DST_PORT 0
```
~~~

### The blocking happens on all ports

Blocking can happen on all server ports from 1 to 65535. Therefore, running circumvention servers on an unusual port cannot mitigate the blocking.

We tested it by having a sink server listen on all ports from 1 to 65535; then having a small Go program continuously send random traffic to each port until the port became blocked.

Reproduce it:

Having a server listen on all ports is not always feasible, since some ports may already be taken. We find the following small trick especially simple and helpful:

1. Let the sink server listen on only one port, for example `12345`.
2. Use iptables to redirect all traffic from `$CLIENT_IP` to the port `12345`: `iptables -t nat -A PREROUTING -i eth0 -p tcp -s "$CLIENT_IP" --dport 1:65535 -j REDIRECT --to-port 12345`.
3. Increase the server process's maximum number of file descriptors: `ulimit -n 655350`.

### UDP traffic does not trigger the blocking

At this time, it appears that this new censorship system is limited to TCP. Because of the absence of UDP blocking, users may experience something interesting when using

Shadowsocks: they can still access websites or use apps that rely on UDP (for example, QUIC), but cannot access websites that use TCP. This is because:

1. Shadowsocks proxies TCP traffic with TCP, and proxies UDP traffic with UDP; and
2. Even when the server's port is blocked, UDP packets to or from the same port is not affected.

Reproduce it:

To test this, start `nping` in the background:

~~~~
```sh
nping -4 -c 0 --tcp-connect $SERVER_IP -p $SERVER_PORT
nping -4 -c 0 --udp $SERVER_IP -p $SERVER_PORT
```
~~~~

Then, run the following simple Python script:

~~~~
```python
import os
import socket
import sys
import time
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.settimeout(1.0)
while True:
    s.sendto(os.urandom(200), (sys.argv[1], int(sys.argv[2])))
    time.sleep(0.001)
```
~~~~

Alternatively, one can run:

~~~~
```sh
while true; do head -c200 /dev/urandom | nc -vn -u $SERVER_IP $SERVER_PORT
& done
```
~~~~

We observed no disruption of `nping` and saw that the UDP packets made it to the server.

### Residual censorship is present

Once the GFW blocks a connection, it continues to drop all TCP packets having the same `(client IP, server IP, server Port)` for 120 or 180 seconds. We tested it as follows:

1. Let both Chinese hosts `C1` and `C2` TCP-ping and UDP-ping port `1000` of a foreign host `F1`;
2. Let Chinese host `C1` also TCP-ping and UDP-ping port `1001` of `F1`;
3. Send random data from `C1` to port `1000` of `F1`.

We observed that only TCP connections, not UDP connections, from `C1` to port `1000` of `F1` got blocked. All other pings could still reach the server or get the proper response.

Similar to a past finding on the GFW's censorship on ESNI traffic, we observed duration of residual censorship to be either 120 or 180 seconds. We do not understand why we observed different durations from different vantage points. Unlike other residual censorship systems, the residual censorship timer does not reset when additional packets are sent.

## How does the GFW identify random traffic?

In this section, we describe our understanding of how the GFW identifies fully encrypted traffic. Although we do not fully understand how the traffic analysis algorithm decides whether to block a connection, our experiments led to interesting observations.

### Blocking is not based on entropy

Alice et al. found that the GFW's active probing system likely used entropy to identify Shadowsocks traffic (see Figure 9 of their paper). However, with this new form of blocking, we find that entropy does not tell the whole story. Some payloads with very high entropy never trigger blocking, whereas other payloads with very low entropy can trigger blocking immediately.

In particular, we observed the following phenomena:

* High-entropy payloads consisting of bytes from `/dev/urandom` triggered blocking quickly.
* Low-entropy payloads consisting of repeated printable characters (`AAAA...`) did not trigger blocking.
* However, some (but not all) low-entropy payloads consisting of *non-printable* characters (`\x15\x15\x15\x15...`) *did* trigger blocking.

This led us to conclude that entropy is not the primary determining factor, but rather whether characters are printable.

## Discussions

This blocking incident reveals many interesting details of the nature of the censor. Understanding this nature will help us 1) better understand the censor's capabilities and limitations, and 2) predict its possible future moves.

### The censor's active probing may have been rendered ineffective

The rollout of this new censorship system may be an indication that the GFW's active probing infrastructure has become less effective due to recent community efforts. The censor has been using a combination of passive traffic analysis, active probing, and potentially human decisions to block Shadowsocks. Such an approach cannot now effectively identify Shadowsocks, thanks to suggestions by Frolov et al. and the patches by developers in the community.

### The censor still attempts to avoid over-blocking

A key insight shared by Tschantz et al., after summarizing a large number of real-world censorship incidents, is that "[c]ensors use exploits for which packet loss results in under-blocking instead of over-blocking" (see Table V and Recommendation 5 of their paper).

This conclusion still holds for the current blocking incident, where the censor limits its blocking only to a few popular VPS providers.

### The censor may still be testing this new system

Some of our results also suggest that the censor is not yet fully confident in the new system:

1. The censor limits its blocking to connections from China to a few popular VPS providers. If the censor were confident that its traffic analysis has very few false positives, they could have applied it to *all* connections to the outside of China.
2. The censor does only short-term dynamic port blocking, rather than long term IP blocking. When using this approach, the censor only blocks the port for 120 seconds or 180 seconds. If the censor were confident in having few false positives, it could have blocked the entire IP address of servers for weeks, like what it had done when using its active probing approach.
3. The censor's blocking rule includes the client IP. If the censor were confident that the server was really a circumvention server, it could have blocked the port or IP of the server for all client IP addresses.

### Censor may tolerate more false positives during politically sensitive times

The start of this less accurate blocking coincides with the Sixth Plenary Session of the 19th CPC Central Committee (中国共产党第十九届中央委员会第六次全体会议) , which may show that censor is willing to tolerate more false positives during politically sensitive times.

## FAQ

### I am a user running a Shadowsocks server. What should I do?

At this time, our recommendation in the short term is to use other data centers to host the server. We have informed the circumvention tool developers on how to mitigate this blocking.

### If my Shadowsocks server is updated to protect from active probing, am I still at risk?

Although the GFW still sends active probes to suspected servers, it can now conduct dynamic and real-time blocking merely based on passive traffic analysis, without using any information from active probing. In other words, even if the server can defend against active probing attacks, it is still vulnerable to blocking. We want to emphasize that this does not mean that [defenses against active probing](https://gfw.report/blog/ss_advise/en) are not necessary anymore.

If your shadowsocks server is updated and accepting connections successfully, you may be running with an IP address the GFW is not yet targeting.

## How can I help?

The current blocking only targets some famous VPS providers, including Vultr, AlibabaCloud, and Digital Ocean. But it possibly does not influence Amazon Lightsail, Oracle Cloud, and others.

Please feel free to let us know of any other cloud vendor that does or does not work.

## Acknowledgement

We thank the project lead of Outline Vinicius Fortuna, and many developers from Shadowsocks and V2Ray community for their discussions on this blocking. We are also grateful to three brave anonymous circumvention users who reported the blocking of their servers to us, allowing us to respond to this incident promptly.