Research Project 2: Using TURN servers as Proxies

Sean Liao
sean.liao@os3.nl
University of Amsterdam
Supervisor: Cedric Van Bockhaven

August 19, 2020

Abstract

Traversal Using Relays around NAT (TURN) is a commonly used relay protocol for realtime audio / video communications services. This paper proposes and demonstrates the use of third party TURN relays, relays run by videoconferencing services, as generic TCP and UDP proxies, exposing the functionality through the SOCKS protocol, as well as the use of TURN relays to tunnel connections out of secured networks.

1 Introduction

Traversal Using Relays around NAT (TURN) is a proxy protocol, designed and primarily implemented for use in realtime peer to peer connections as is common in videoconferencing software. It extends Session Traversal Utilities for NAT (STUN) [10]. While STUN provides utilities for clients to establish peer to peer connections through NAT and firewalls with techniques such as hole punching, this is not always successful, in which case TURN relays with public addresses can be used to relay data between the clients. Given its primary use case in audio/video communications such as in WebRTC [1], TURN uses UDP for peer connections. RFC 6062 [11] specifies an extension to TURN to use TCP connections.

SOCKS is a widely supported proxy protocol with a client-server model, providing a circuit-level gateway with a simple interface useful for traversing firewalls and Network Address Translation (NAT) [7]. Despite the capitalization, SOCKS does not appear to be short for anything. [1] SOCKS5 was an update that added support for UDP among other features to the TCP based protocol [4].

Given the nature of proxies, operators of TURN relays need to be careful in the design of their network and in the security policies enforced by the proxy itself. Failure to do so could result in unwanted connections being made into internal networks.

For users of other networks, TURN relays run by public entities, such as those used by videoconferencing software, stand in a privileged position as connections to them are often allowed to pass through both NAT and firewall due to business needs. Using these TURN relays as generic proxies could punch through firewalls for a wider class of applications.

2 Related Work

From early in its design TURN was recognized to stand at a critical juncture between networks. The latest RFC [6] expands on the security considerations when running TURN relays. Additionally, both an authentication and permissions system is built into the protocol, as well as recommendations in configuration.

CloudProxy was a research project that combined SOCKS and TURN for vulnerability scanning with a focus on performance [13]. It uses the TURN protocol for NAT traversal, and a modified client-server pair to deduplicate

[1] https://www.usenix.org/legacy/publications/library/proceedings/sec92/full_papers/koblas.pdf

scanning traffic. This differs from the approach in this paper, which uses unmodified and potentially third party TURN relays for NAT traversal.

The only other notable publicised instance of using TURN relays as a proxy is an April 2020 report by Enable Security outlining misconfiguration of Slack's TURN relays. They used an internal tool to proxy both TCP and UDP connections through Slack's TURN relays [12].

3 Research Question

The aim of this research is to implement and test ideas for using third party TURN relays as proxies, exposing the functionality as a SOCKS proxy. There are two main approaches we will take: a forward translation layer and a tunnelled connection through a TURN relay.

The forwarding approach translates a SOCKS connection request into a TURN connection request, with the TURN relay making the final connection to the destination. This acts mainly as a test of TURN relay capabilities and (mis)configuration. This is very similar to what Enable Security have done, with the main blocker being the lack of public libraries that support the full range of TURN specifications and capabilities implemented by relays.

The tunnelled connection will be of more interest to users inside restricted environments. For practical reasons, the TURN relays operated by major business software services providers are often whitelisted in firewalls. This presents the opportunity to pass connections through these relays, masking their true destination and opening up protected networks to outside connections. This is superficially similar to what CloudProxy did, however, that project only used the protocol and not known endpoints to bypass NATs.

In short:

- What are the technical and practical limitations of a SOCKS to TURN translation layer?
- Which popular public services can be used for tunnelling to bypass network restrictions?
- What are possible methods of preventing abuse for both network and TURN relay operators?

4 Background

4.1 SOCKS5

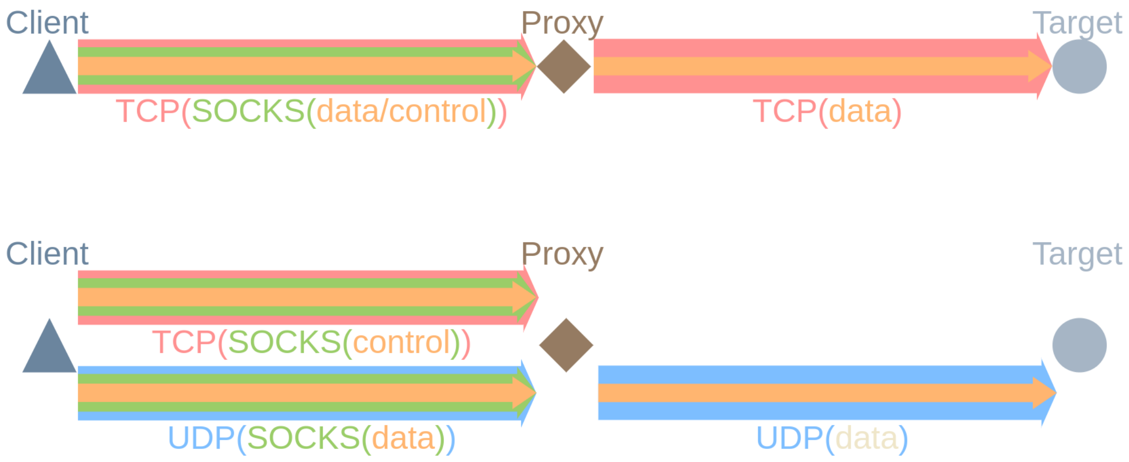


Figure 1: SOCKS: encapsulation

SOCKS is a simple protocol, designed for when the client is behind some NAT or firewall and needs to pass though to a network connected to the proxy.

For TCP connections, clients can establish a connection to a proxy and send request with the intended destination and options inline. The proxy can then create the connection to the final destination and reply with a success response to the client. Any further data on the connections is then relayed between the client and destination without modification.

For UDP, the client first establishes a TCP connection to the proxy to act as a control connection. The proxy replies with the UDP port for the client to connect to. The TCP connection is then used to control the lifetime of the session. Each UDP packet from the client contains a SOCKS header containing the destination, this is removed by the proxy before forwarding to the destination. Incoming packets have a SOCKS header inserted before being sent to the client.

4.2 STUN & TURN

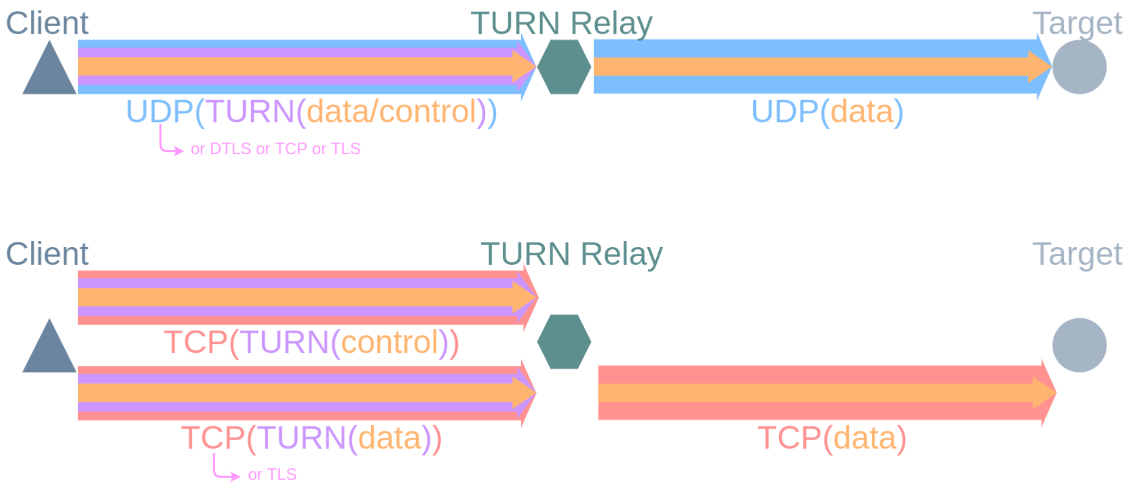


Figure 2: TURN: encapsulation. Note the connection between the client and the TURN relay may be optionally encrypted through TLS or DTLS.

STUN and TURN are designed to facilitate establishing peer to peer connections. STUN encompasses the extensible wire protocol and interactions such as discovering the address of a client after NAT [10]. TURN extends the protocol to support relaying of data in the cases that direct peer connections between clients cannot be established [6].

STUN and TURN are most commonly associated with realtime audio and video communications [1]. These are the cases in which peer connections are advantageous due to lower latency as well as lower processing and bandwidth costs for the service operator. It is also for this reason that authentication is mandatory to support TURN. TURN allocations are time limited (10 minutes by default) with the option to refresh.

For UDP connections to a destination, clients can connect to the relay over UDP or TCP, or their secured variants, DTLS[9] or TLS. Since STUN messages contain a length field, they can be transmitted over a reliable stream without issue. Clients can then send an `Allocate` request, allocating a UDP port on the relay for communications. The client can then send data with `Send`, containing both the destination and the data. Alternatively, clients can request a channel with `ChannelBind` and send data with `ChannelData`, omitting the STUN header and using just a 4 byte ID for a lower overhead method of communications.

For TCP connections, clients first connect to the relay with either TCP or TLS, establishing a control connection. As with UDP, an `Allocate` request allocates a (TCP) port. The client can then send a `Connect` request with the intended destination. If the relay connects successfully, it will reply with a `CONNECTION-ID`. The client can then open a separate data connection to the relay, and associate to the requested connection with a `ConnectionBind` including the `CONNECTION-ID`. All further data on the data connection is then relayed between the client and destination without modification.

Specification wise, RFC 5766[6] for TURN only specifies support for UDP allocations, given the intended use case for audio/video communications. TCP support was added in RFC 6062[11].

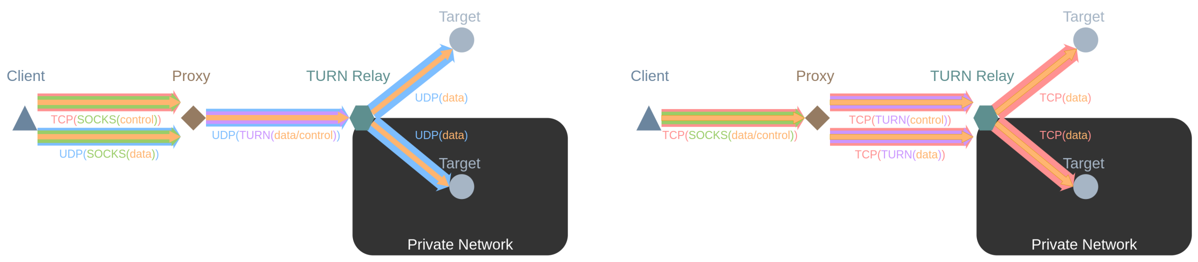
In terms of software, there are various server implementations, a non exhaustive list includes: coturn[2], restund,[3] reTURNServer[4], rfc5766-turn-server[5]. Of these, only coturn supports the full range of protocols, others only support the base protocol for UDP connections. Public client library support is more scarce, for C/C++ the server implementations

[2]`https://github.com/coturn/coturn`
[3]`http://www.creytiv.com/restund.html`
[4]`https://github.com/resiprocate/resiprocate`
[5]`https://github.com/coturn/rfc5766-turn-server`

can be partially reused, Erlang has a `processone/stun`[6] library with TURN support, and Go has `pion/turn`[7] library. Further support can be found in browsers which expose the functionality under WebRTC. Of these client libraries, none support TCP connections, and the only instance of client code we found for TCP was in coturn's test client [8].

5 Methodology

5.1 Forward Connections



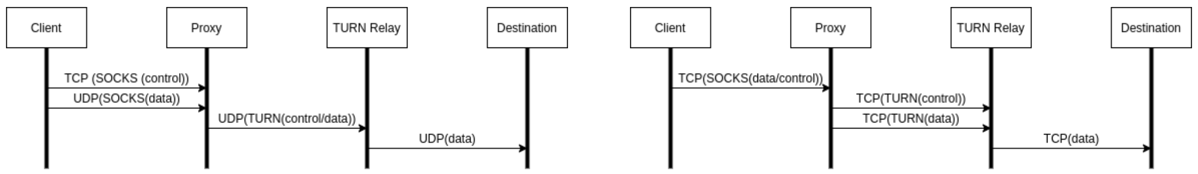
(a) Making a UDP connection (b) Making a TCP connection

Figure 3: Forward connections: encapsulation

This section describes a mode of operation in which a proxy runs as a SOCKS server and TURN client, translating SOCKS into TURN, forwarding the request to a TURN relay. The relay then makes the connection to the final destination.

For UDP, in theory, the time to establish a TURN session from the proxy to the relay is when a client first establishes a TCP control connection to the proxy. However, due to various limitations, such as poorly behaving clients which don't send the source UDP port it intends to use, or inflexible server libraries, it may not be possible to associate client UDP packets with a TCP session. For this reason, a new TURN session is started on demand upon an incoming client UDP packet, keyed by the client's source address. Unfortunately, this may slowly leak sessions as the aforementioned inability to associate with control channels means there is no point at which the session can be safely shut down, meaning they must be held open indefinitely.

For TCP, a TURN session can be started per SOCKS connection, and the connection oriented nature means it is much more straightforward to relay the packets. However, this approach may run into per user session quotas, if the TURN relay has it configured, earlier than UDP as there is absolutely no session reuse.



(a) Making a UDP connection (b) Making a TCP connection

Figure 4: Forward connections: order of events

5.2 TURN TCP

Since many commonly used tools and protocol, such as SSH and HTTP, are built on TCP, gaining access to a relay with TCP support enabled would be very useful. To do so, it is necessary to implement TCP support (RFC 6062). From the libraries mentioned in the previous section, `pion/turn` was selected for extension, based on the existing framework, the availability of other protocols for later parts, and the author's familiarity with the Go language. As a proof of concept, the code is a relatively straightforward implementation of the RFC, reusing large parts of the code already

[6] `https://github.com/processone/stun`
[7] `https://github.com/pion/turn`
[8] `https://github.com/coturn/coturn/wiki/turnutils_uclient`

existing for UDP communications, and is available upstream on the `rfc6062`[9] branch. The diffs with the specific implementation can be viewed online[10]. This should not be confused with a `rfc-6062-client`[11], an earlier attempt[12] from someone else that did not see any significant progress. This extension includes full client and server side support for TCP allocations. However, there remain some minor API design issues, which are a blocker for merging into the main release.

5.3 Tunnelled Connections

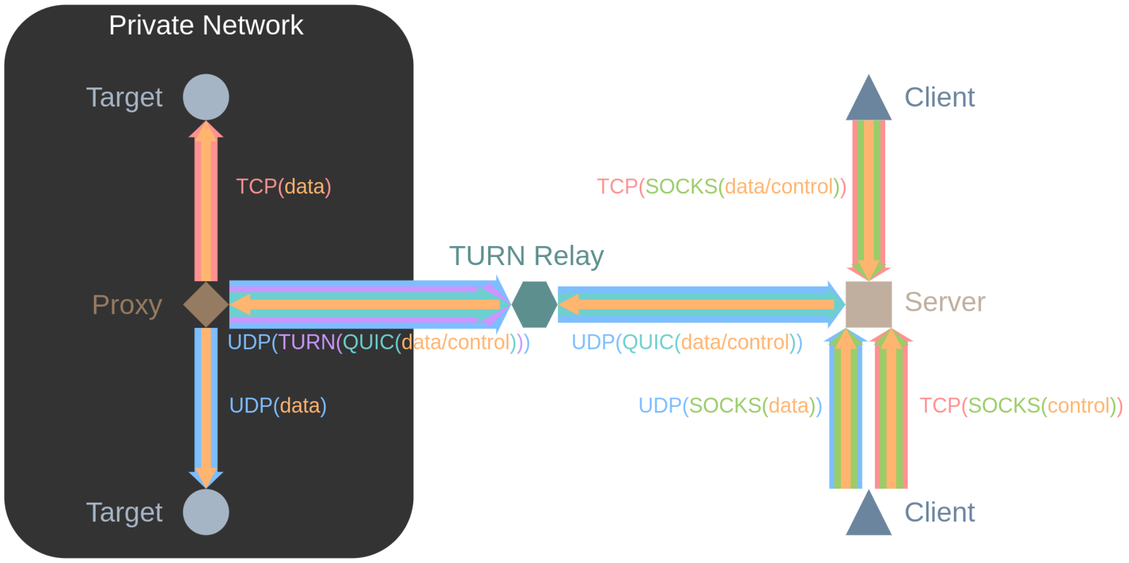


Figure 5: Tunnelled connection: encapsulation

This section describes using a TURN relay as a known whitelisted endpoint to establish a tunnel through a NAT or firewall. The initiator "proxy" connects to the relay from within a restricted network, the relay then forwards the connection to a waiting static endpoint "server". This avoids the need for separate signalling server as commonly used in peer to peer connections, such as with the Interactive Connectivity Establishment (ICE)[3] The server can then start a SOCKS server and tunnel all connections back to the proxy, which makes the outgoing connections from within the restricted network. This is analogous to `ssh -R`, opening up targets from the initiator's side to incoming connections.

There is more flexibility within this design, as we control both ends of the connection with the TURN relay. We can limit the TURN protocol to just UDP, as this is the most widely available. To multiplex multiple connections over a single TURN session/connection, we use QUIC[2]. While QUIC would not be necessary for just relaying UDP data, most common tooling works with TCP, so a reliable stream over the underlying UDP/TURN connection is desirable. Additionally, since clients cannot easily initiate new connections, multiplexing multiple connections over a single existing one is also a desirable feature.

QUIC is a transport protocol on top of UDP, originally designed by Google[13] and currently undergoing standardization at the IETF. With QUIC implementations already running for browsers (Firefox[14], Chrome[15]), servers (NGINX[16]), and cloud providers (Google, Akamai[17], Cloudflare[18]) as part of HTTP/3[5], QUIC is likely to remain well supported for some time to come. This gives us multiplexed, bidirectional streams between the proxy and server.

[9]`https://github.com/pion/turn/tree/rfc6062`
[10]`https://github.com/pion/turn/pull/148/files`
[11]`https://github.com/pion/turn/tree/rfc-6062-client`
[12]`https://github.com/pion/turn/issues/118`
[13]`https://www.chromium.org/quic`
[14]`https://blog.cloudflare.com/how-to-test-http-3-and-quic-with-firefox-nightly/`
[15]`https://chromiumcodereview.appspot.com/11125002`
[16]`https://www.nginx.com/blog/introducing-technology-preview-nginx-support-for-quic-http-3/`
[17]`https://developer.akamai.com/blog/2018/10/10/introducing-quic-web-content`
[18]`https://blog.cloudflare.com/head-start-with-quic/`

On top of QUIC, a lightweight messaging protocol is used. A message consists of a length as a 4 byte unsigned integer in big endian, followed by the message body. TCP and UDP connections open with the first message containing `udp` or `tcp`. The second message contains the destination address. Afterwards, for TCP connections the contents of the stream are directly relayed to the destination without further interference. For UDP, the packet boundaries are preserved by encapsulating them in a message as described above. In the future, if the datagram extension to QUIC is standardized and implemented, UDP packets can use that instead.

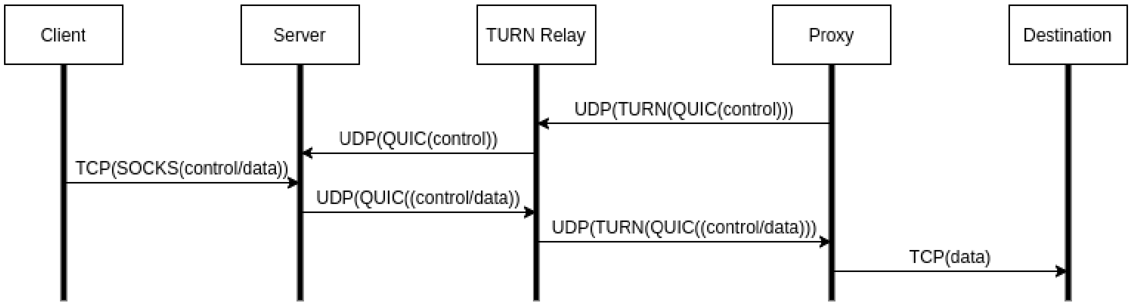


Figure 6: Tunnelled connection: The proxy and destination are located in a protected network. The proxy initiates the first connection. Clients can connect to the server once it has started a SOCKS server (after it receives an incoming QUIC connection). UDP connections follow a similar path.

5.4 Testing

To test our implementation, we selected a handful of popular hosted videoconferencing software to attempt to connect to. To do so, we signed up for the various services, using credentials associated with our own accounts for testing. Despite being called "long-term credentials", those used in TURN authentication are in reality short lived and supplied on demand. There was a draft recommendation on the requesting credentials over HTTP/JSON, however it was never standardized and each product uses their own way.

To avoid the tedium of reverse engineering the credential exchange mechanisms of multiple services, we patched Chromium to output the credentials it received and used to connect to TURN relays. Obtaining the credentials was then a matter of making a call through the web browser, and copying the credentials from the debug log. It should be noted that reverse engineering the credential exchange mechanisms would be unavoidable if stable long term credentials are desired.

6 Results & Discussion

6.1 Forward & Tunnelled

Implementation can be found on GitHub[19]. Code related to forwarding is in *forward.go*, for tunnelled connections the proxy is in *reverse_client.go* and the server is in *reverse_server.go*.

For forward connections, the core of it is translating the SOCKS destination to a TURN `XOR-PEER-ADDRESS`, which both have to be `ip:port` pairs. The remainder revolves around managing sessions and keeping track of UDP connections. For tunnelled connections, the work centers around getting SOCKS and native connections into and out of QUIC streams.

Given TURN relays that support the protocols and no other restrictions, forwarding connections works as expected for both TCP and UDP. Tunnelled connections also work as expected, successfully relaying both TCP and UDP connections through the TURN relay (which only needs UDP).

6.2 Limitations

There are several issues with the implementation, such as the lack of session reuse due to the desire to allow multiple connections to the same host/port destination and the allocation leakage stemming from unclosed SOCKS UDP sessions mentioned in section 5.3. as well as a lack of graceful error handling.

[19]`https://github.com/seankhliao/uva-rp2/tree/master/cmd/proxy`

Another limitation is with domain names. In the forwarding configuration, this is limited by both the choice of SOCKS library and the TURN protocol, which does not support addressing hosts through domain names. In the tunnelled configuration, this is only limited by the SOCKS library, and a different implementation would not have this restriction. As a result of the above, only raw IP addresses or names resolvable from the public network can be used to address the destination.

As a practical issue for smuggling the proxy into secured environments, the statically linked final binary weighs in at 16MiB, this can be reduced to 4.1MiB by stripping out debug symbols and packing with `upx`[20], but it is still a large executable. This is a general limitation with Go programs in general.

6.3 Third Party TURN Relays

Service	UDP	TCP
Zoom	No TURN used	
Google Meet	No TURN used	
Cisco Webex	Drops after allocate	
GoToMeeting (Citrix)	Wrong Transport Field	
Slack	Forbidden IP	X
Microsoft Teams / Skype	V	X
Jitsi Meet	V	X
Riot.im (Matrix)	V	X
BlueJeans	V	X
Facebook Messenger	V	X

6.3.1 No TURN Relay

In the course of our testing, both Zoom and Google Meet do not appear to use TURN relays for their videoconferencing products. As such, there is nothing to test.

6.3.2 Restricted TURN relays

Cisco provides an online test instance for their Webex videoconferencing solution [21]. From this we were able to extract a hardcoded set of TURN credentials: `ciscoThinClient / 1234abcd`. Connecting to and allocating a port completed successfully for both TCP and UDP, however, any further requests would be dropped, resulting in timeouts and no usable connection.

Citrix offers GoToMeeting as their web conferencing product [22]. This also uses a set of hardcoded credentials: `citrixturnuser / turnpassword`. Connecting succeeds, but allocations fail with an error stating `Wrong Transport Field` for both TCP and UDP. More investigation would be needed to determine the transport field it uses, but even so it would be considerably less useful for proxying arbitrary connections.

Slack has calls within its product [23]. Enable Security had previously successfully connected to Slack's TURN relays[12], however, since then, they have presumably fixed the issue and announced a migration[14] to Amazon Chime[24], Amazon's hosted communications service. At the time of testing, the TURN servers exposed were Amazon Chime servers, which only allowed UDP allocations, but restricted making connections to outside addresses. Further testing would be needed to identify unrestricted address ranges.

[20]`https://upx.github.io/`
[21]`https://www.webex.com/test-meeting.html`
[22]`https://www.gotomeeting.com/`
[23]`https://slack.com/intl/en-nl/help/articles/115003498363-Slack-calls--the-basics`
[24]`https://aws.amazon.com/chime/`

6.3.3 TURN UDP Relay

Microsoft Teams[25], Jitsi Meet[26], BlueJeans[27], Facebook Messenger[28], and Riot[29] (a Matrix[30] client) all had TURN relays that allowed UDP connections to the public internet. None of them had TCP support enabled, and all used credentials generated on demand with short validity periods. These services used various methods to convey the generated credentials to clients, such as in cookies for Microsoft Teams and in XMPP messages over WebSockets for Jitsi Meet.

6.3.4 TURN TCP Relay

None of the major services tested enabled TCP allocations for their TURN relays. Given the lack of library support and the single server implementation, this leads us to believe that despite its presence in a standard, there are few valid use cases for it and that most relays which have it enabled are misconfigured. While not directly in scope for this research, if the need for TCP relays arises, candidates could be identified with a combination of port scans and user/password lists.

6.4 Defense

For the operator of a secured network, there is unfortunately not a lot that can be done with regards to blocking unwanted connections. Deep packet inspection may flag the multiple either the multiple encapsulation layers or the lack of expected audio/video data as suspicious, while network flow analysis could in theory identify abnormal streams, such as bursts common to one off commands as opposed to a constant video stream. though self signed and pinned certificates in many legitimate applications will limit their usefulness, showing only an encrypted data stream. Previously, RFC 8155[8] did specify the possibility of discovering local STUN/TURN relays through DNS, giving operators control over outgoing connections, though is is unclear if any clients actually support it.

In light of the above, operators could put more work into segmenting parts of their network which do and do not have access to certain destinations, or they could also move towards a Zero Trust architecture, trusting their network even less.

For the operator of a TURN relay, there are many options in building a multilayered defense against potential abuse.

First would be hiding the relays. A simple method is running the relays with non standard ports, something that may already to done to bypass firewalls. More involved would be only running the secured variants of the underlying transport, DTLS or TLS, and using Server Name Indication (SNI) to route traffic. This would prevent casual scans based on the IP address from learning the existence of relays, but would do nothing against a more targeted approach, such as being selected for being a well known public service, as was done for testing against the popular videoconferencing services. In that case, legitimate traffic can be observed to identify the endpoints.

The next line of defense is authentication. Despite its name, the "long-term credentials" in TURN are often only valid for a short period and generated on demand. While there were recommendations on how to relay these generated credentials to clients, there is no standard and each service does it in their own way, requiring reverse engineering of the credential exchange mechanism for each service. In the future, OAuth tokens may be used, standardizing authentication flows [31].

A final line of defense rests with authorization, the TURN operator should limit access to destinations as necessary, including limiting the protocol, address ranges, and possibly port ranges, as in normal use clients should only be connecting to and from ephemeral ports.

As a side note, TURN relays are used primarily as a fallback to peer to peer connections. Due to bandwidth and other concerns, this typically limits the service to approximately a dozen peers. Scaling beyond this would require central media servers such as MCUs (Multipoint Conferencing Unit) or SFUs (Selective Forwarding Unit). If this approach is taken, a TURN relay should not be necessary.

[25]`https://www.microsoft.com/en/microsoft-365/microsoft-teams/group-chat-software`
[26]`https://jitsi.org/jitsi-meet/`
[27]`https://www.bluejeans.com/`
[28]`https://www.messenger.com/`
[29]`https://about.riot.im/`
[30]`https://matrix.org/`
[31]`https://github.com/coturn/coturn/wiki/turnserver`

7 Conclusion

This paper has demonstrated the viability of using third party TURN relays as a proxy server for SOCKS clients through a translation layer. While both UDP and TCP work on a technical level, servers with TCP support enabled are much rarer in the wild. Additionally, we have shown the possibility of tunnelling both TCP and UDP connections through a TURN relay (which only speaks UDP) into secured networks.

8 Future Work

The research in this paper has all been conducted with IPv4. IPv6 should work in theory but is untested. For using the tunnelled connection in adverse environments, it would be interesting to integrate the proxy as part frameworks such Metasploit, perhaps with a more stripped down and minimal version. Finally, an interesting option would be to further cloak the connections to resemble audio or video streams, blending in better with native WebRTC traffic.

References

[1] C. Holmberg, S. Hakansson, and G. Eriksson. *Web Real-Time Communication Use Cases and Requirements*. Tech. rep. RFC 7478. Mar. 2015.

[2] Ed. J. Iyengar and Ed. M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Tech. rep. draft-ietf-quic-transport-29. June 2020.

[3] A. Keranen, C. Holmberg, and J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. Tech. rep. RFC 8445. July 2018.

[4] M. Leech et al. *SOCKS Protocol Version 5*. Tech. rep. RFC 1928. Mar. 1996.

[5] Ed. M. Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Tech. rep. draft-ietf-quic-http-27. Feb. 2020.

[6] R. Mahy et al. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. Tech. rep. RFC 5766. Apr. 2010.

[7] Rolf Oppliger. *Security Technologies for the World Wide Web*. 2nd ed. 2003. ISBN: 1580533485.

[8] P. Patil, T. Reddy, and D. Wing. *Traversal Using Relays around NAT (TURN) Server Auto Discovery*. Tech. rep. RFC 8155. Apr. 2017.

[9] M. Petit-Huguenin and G. Salgueiro. *Datagram Transport Layer Security (DTLS) as Transport for Session Traversal Utilities for NAT (STUN)*. Tech. rep. RFC 7350. Aug. 2014.

[10] J. Rosenberg et al. *Session Traversal Utilities for NAT (STUN)*. Tech. rep. RFC 5389. Oct. 2008.

[11] Ed. S. Perreault and J. Rosenberg. *Traversal Using Relays around NAT (TURN) Extensions for TCP Allocations*. Tech. rep. RFC 6062. Nov. 2010.

[12] Enable Security. *How we abused Slack's TURN servers to gain access to internal services*. 2020. URL: `https://www.rtcsec.com/2020/04/01-slack-webrtc-turn-compromise/` (visited on 06/04/2020).

[13] Yulong Wang and Jiakun Shen. "CloudProxy: A NAPT Proxy for Vulnerability Scanners based on Cloud Computing". In: *J. Networks* 8 (2013), pp. 607–615.

[14] Tom Warren. *Slack partners with Amazon to take on Microsoft Teams*. June 2020. URL: `https://www.theverge.com/2020/6/4/21280829/slack-amazon-aws-partnership-amazon-chime-voice-video-calls` (visited on 06/30/2020).