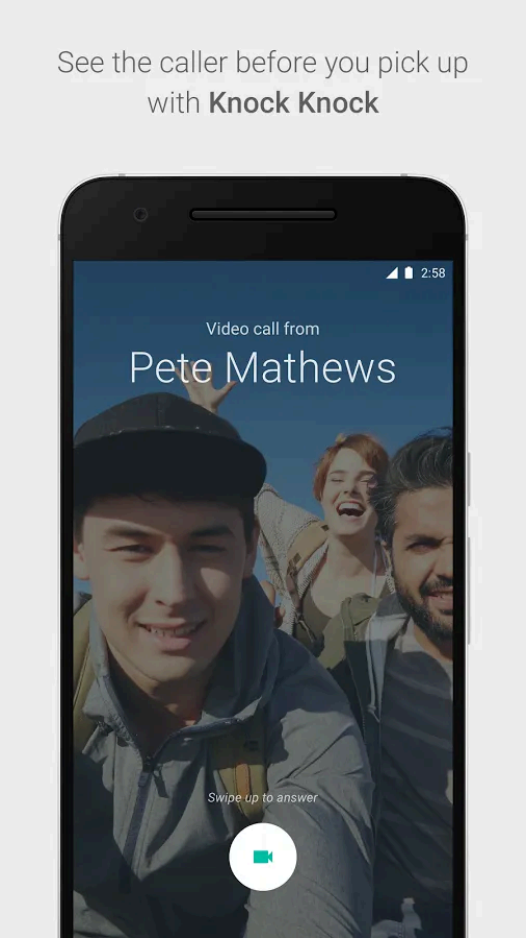
Google Duo Race Condition

Written by Swayam Parida & Benjamin Xie

Introduction

Google Duo was the Android equivalent of FaceTime that let users make cross-platform video calls.

When it was introduced, it had a quirky feature called “Knock Knock” that allowed a callee to view the caller’s video when receiving the call and allowed the callee to decide whether to answer or not.



In 2020, security researcher Natalie Silvanovich discovered a race condition bug in this application that allowed a caller to receive a few video frames of a callee even if the callee didn't answer the call.

Background

Many modern video call applications, including Google Duo, make use of an open-source framework called WebRTC[1]. This framework provides APIs for media capture – i.e. access to microphones and cameras – as well as peer-to-peer (P2P) connectivity.

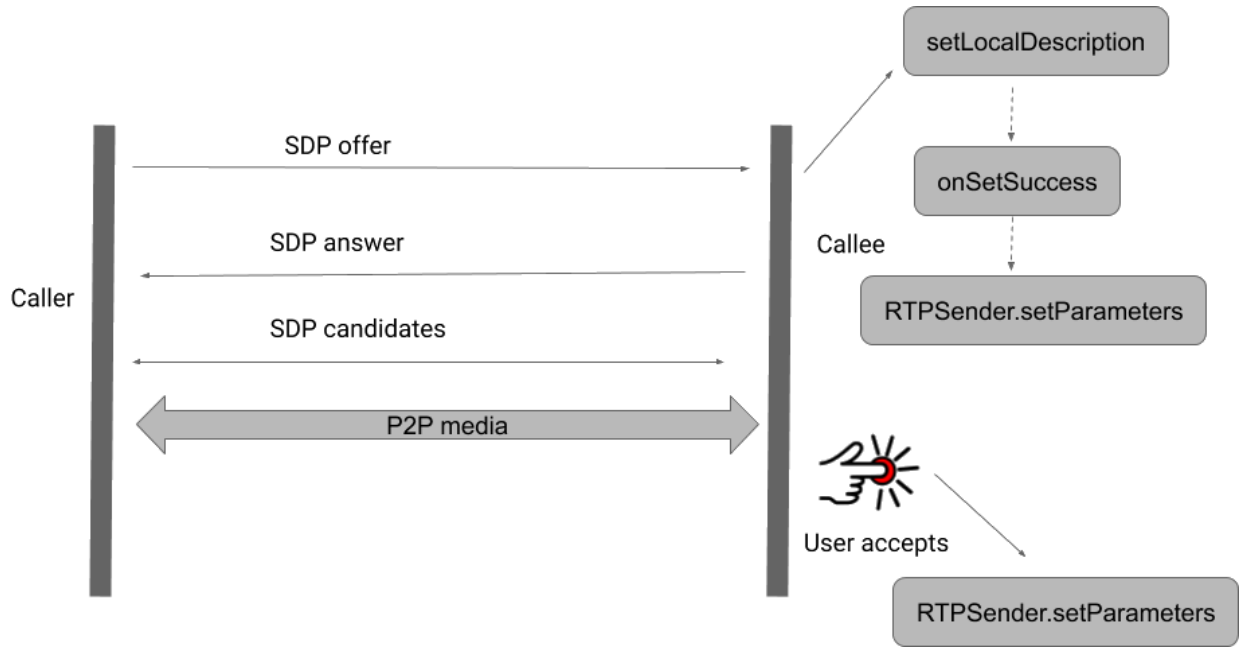
P2P connectivity is desirable for applications where latency is of utmost importance. For services such as text messaging, latency is not particularly important and so when User A sends a WhatsApp message to User B, the message goes through Meta servers. By contrast, latency is very important for audio and video calls and thus these applications will want to send data directly between User A and User B's devices without having to go through an intermediary server.

The issue is that the modern internet does not support easy P2P connectivity between consumer devices due to the pervasiveness of something called Network Address Translation (NAT). Hence, for media to be exchanged over a P2P connection, the two communicating devices must first engage in something called the Session Description Protocol (SDP) to establish the parameters for the P2P connection.

[1] RTC is an acronym for Real Time Communication

Describing the bug

The diagram below illustrates the sequence of events that takes place when a Google Duo call is made and a one-way video stream is initiated for the Knock Knock.



When the Duo application receives a phone call, it concurrently does two things:

- It creates a WebRTC connection – by calling `setLocalDescription` on the SDP offer received.
- It disables outgoing video traffic from the callee's device – by calling `RTPSender.setParameters`.

One may wonder why the P2P connection even needs to be created before the callee answers the phone call. The reason for this is to support the Knock Knock feature. Without creating a P2P connection, it would not be possible to transmit the caller's video feed to the callee's device.

An important point to note is that the caller's SDP offer contains a property (`a=sendonly`) which configures the connection such that the callee will receive the caller's video stream but not vice versa. However, since it is the caller that creates the

SDP offer, it is in their power to maliciously change the property to be `a=sendrecv` such that the client begins transmitting their video by default as soon as the SDP offer is processed and the P2P connection is established. Google was aware that such a situation was possible and thus added in another precaution: when the callee accepts the SDP offer, they schedule a thread that explicitly disables outgoing traffic.

Normally, disabling outgoing video traffic is a very quick process and is completed before the WebRTC connection is set up and thus there is no risk of video traffic being transmitted from the callee's device by the time the WebRTC connection is created.

However, since these tasks are scheduled to occur concurrently, in the absence of formal synchronization mechanisms such as locks and condition variables to prevent the threads from making progress, the OS is free to schedule these threads in whatever manner it pleases.

An attacker with knowledge of the OS's scheduling policy can take advantage of this setup by sending a lot of network traffic to the callee's device. The OS prioritizes processing these packets ahead of the thread that is responsible for disabling outgoing traffic. This is similar to how we've put a `sleep(500)` call at different points in a thread to surface race conditions.

Hence, if the thread that disables outgoing traffic ends up being run after the connection setup thread, the caller is able to receive the callee's video feed in the interim period, i.e. the period between when the connection has been set up but the outgoing traffic has not yet been disabled.

References

Project Zero Blog Post:
https://googleprojectzero.blogspot.com/2021/01/the-state-of-state-machines.html#h.8htob0rs10i7

Project Zero Bug Disclosure:
https://bugs.chromium.org/p/project-zero/issues/detail?id=2085&q=duo&can=1