Bug hunting in Steam: a journey into the Remote Play protocol

Valentino Ricotta
valentino.ricotta@thalesgroup.com

Thalium

Abstract. Valve, the company behind the widespread videogame platform *Steam*, released in 2019 a feature called *Remote Play Together*. It allows sharing local multiplayer games with friends over the network through streaming.

The protocol associated with the Remote Play technology is elaborate enough to lead to stimulating attack scenarios, and its surface has scarcely been ventured in the past.

This paper covers the reverse engineering of the protocol and its implementations within Steam (client and server). Then, it presents a dedicated fuzzer and a few bugs that have been discovered thanks to it, some of which were exploitable.

1 Introduction

1.1 Context and target

More than a billion people around the world play online videogames on various platforms: Windows, Linux, macOS, Android, iOS, gaming consoles, VR headsets and more.

Online multiplayer games are also massive binaries with a large attack surface (network, gaming logic, graphics, sound, maps...). They are thus great targets for remote hackers, who can seek to exploit vulnerabilities in game clients or servers to fulfill various purposes: cheating, harvesting credentials, spreading malwares, cryptomining, or even targeted surveillance.

Valve is a well-known game developer, editor and publisher. They run a bug bounty program on HackerOne with a lot of public reports [7] that can give great inspiration for attack surfaces and exploitation techniques related to their products. They also developed the *Steam* software, which is the most widely used video game platform.[1] It centralizes and distributes dozens of thousands of games, along with many features (social networking, game integration, marketplaces...).

[1] In 2022, Steam gathered more than 50M daily active users [3].

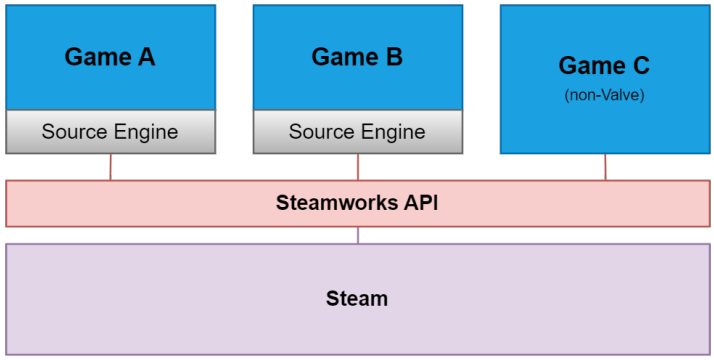

Fig. 1. Layers of attack surfaces within Valve games and the Steam software.

Figure 1 is an overview of the potential attack surface within games on Steam. Many RCEs have already been discovered in Valve games such as Counter-Strike, or inside the game engine they developed, called the *Source Engine*. Likewise, there are public reports of vulnerabilities inside the Steamworks API, a software development kit targeted towards game developers to integrate Steam features into their games.[2]

However, only little research has been conducted on the *Steam* client itself, making it a rather compelling target. More particularly, delving into the public reports, there was never mention of a fairly interesting component in the Steam client: Remote Play.

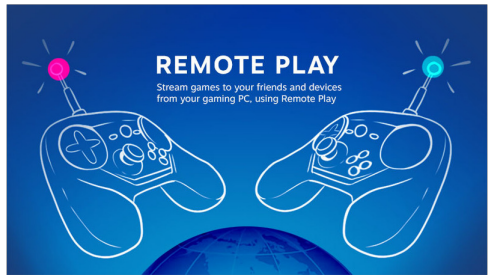

Fig. 2. Steam Remote Play.

[2] For instance, in 2021, user *slidybat* reported a stack buffer overflow in the `DecompressVoice` method that impacted several games with voice communication support, like CS:GO.

1.2 Steam Remote Play

Steam Remote Play began in 2015 with Steam Link.[3] It allows one to stream a game from their computer, usually a gaming rig, to another (typically less powerful) device, like a smartphone, a tablet or a TV.

In 2019, Valve then introduced *Remote Play Together*, allowing players to share local multi-player games with their friends over the network through streaming. The player who streams the game, called the *host*, sends an invite link to another player, the *guest*, who does not need to own the game. The guest can then send inputs (mouse, keyboard, controller...) to play together with the host. This is shown in figure 3.

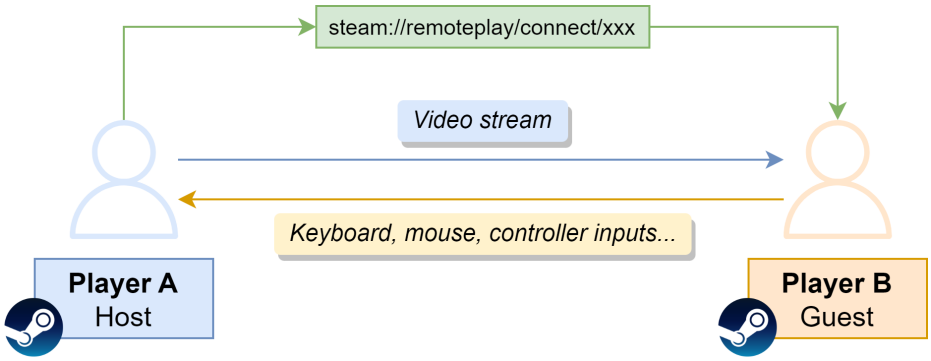

Fig. 3. Interaction between host and guest in Remote Play Together.

Since the protocol behind Steam Link and Remote Play Together is the same, both products can be analyzed in order to reverse it. However, the latter is a more promising target as host and guest are connected through a peer-to-peer link (or a transparent relay). This implies that no third party will verify, filter or alter messages from host to guest and conversely.[4] Based on this information, two main attack scenarios against Remote Play Together can be thought of:

1. escaping the "game sandbox" to gain *graphical* remote access to the host's desktop (client-side attackers only);
2. exploiting low-level bugs (e.g. memory corruption) to achieve RCE or info leak (for both client and server-side attackers).

[3] Initially a set-top box, its hardware version was discontinued in 2018 to make way for a software-based version, also sometimes called *Steam In-Home Streaming*.

[4] Throughout this article, the terms *client* and *guest* will be used interchangeably, as well as the terms *server* and *host*.

The first one was not explored, as reversing the streaming, sandboxing and access control logic sounded more complex than homing in on finding bugs in protocol parsing and message processing. In addition, the latter accommodates better to fuzzing techniques, and can target both the client and server implementations. Hence, this is what this article will focus on. It was also decided to work on the Windows binaries as it is the most popular and thus impactful environment.

Both host-to-guest and guest-to-host attack scenarios are worth investigating, but it is important to notice that vulnerabilities in Remote Play Together have a stronger impact for the guest players, as a client:

— does not need to own any particular game on Steam;
— does not need to be friends with the attacker on Steam (anyone can open an invite link);
— automatically connects to the Remote Play server upon the link being opened (no further user interaction or confirmation).[5]

2 Study of the Remote Play implementations in Steam

2.1 Software architecture

Remote Play involves two main binaries: `streaming_client.exe`, spawned by Steam upon joining a Remote Play session and which contains all of the client logic, and `SteamUI.dll`, where most of the server's logic is located. Like most of Steam, these are written in C++. Analyzing them will provide answers to questions such as how do client and server communicate, what is the packet format, where in the binaries do packets arrive, or how is audio and video data exchanged.

These rather large binaries (15MB) are exempt from any debug symbol, making the analysis much more laborious. Fortunately, there is another way.

The **Steam Link client for Android** (native library) curiously happens to contain a lot of symbols, and more especially function names. Although this may be some kind of compilation or distribution error from Valve, this is a remarkable asset for reversers. Therefore, even though we target Windows environments, most of the analysis for the protocol can be performed on the Android client. Figure 4 shows a high-level architecture of the client implementation.

[5] The link can even be opened without any user interaction under certain circumstances (for instance, if a `steam://` wrapper URL is hidden inside an iframe on a web page hosted on a trusted domain), which can turn a whole remote code execution zero-click.

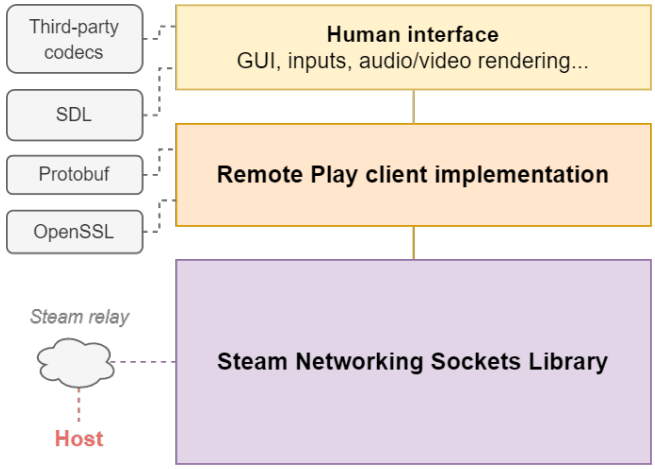

Fig. 4. High-level view of the architecture of the client's implementation.

Some important dependencies are the Protobuf library,[6] used to serialize messages in the Remote Play protocol, and SDL which is mainly used for the GUI, audio/video rendering, and interfacing with input devices (keyboard, controllers...).

At the foundation of the client lies a rather large component, the *Steam Networking Sockets* library, in charge of the P2P transport. It seems to be at least partially based on Valve's Game Networking Sockets (GNS) [14], an open-source UDP connection-oriented transport layer with support of many features such as encryption and P2P.[7] It was decided not to investigate this component further because in the context of Remote Play, attack scenarios involving this library are much more complex.

As for the server implementation, its architecture is quite similar to the client's, without the human interface part. It is also worth noting the server implementation is part of a bigger DLL that contains lots of other Steam-related stuff that are not linked to Remote Play.

[6] Protobuf (or *protocol buffers*) is a serialization mechanism developed by Google [6] that is extensively used within Valve games and Steam.

[7] A brief analysis suggested that Valve uses a heavily modified version of GNS. The P2P part of GNS is based on WebRTC and the ICE protocol, but Remote Play doesn't seem to implement the full WebRTC stack: it mostly consists of TURN/STUN and custom encryption layers with clear deviations from GNS.

2.2 Reverse engineering the protocol

To get started on reverse engineering the protocol, there exists a tremendously useful Protobufs repository on GitHub [12], maintained by the SteamDB project [13]. It tracks many protobuf definitions from Valve products. More particularly, the `steammessages_remoteplay.proto` file is a goldmine to learn about the protocol, as it includes practically all the message types and their fields.

Of course, efforts do not stop there; there is still a lot to unpack by reversing the binaries, and the first step is to understand how packets are received and processed.

Network reception logic and processing. Figure 5 shows a high-level view of the data flow for packets that arrive from the server. Having a clearer overview of this part of the architecture helps a lot to, on the one hand, understand the protocol, and on the other hand, bring out potential attack surfaces. The diagram can be split into three main components.

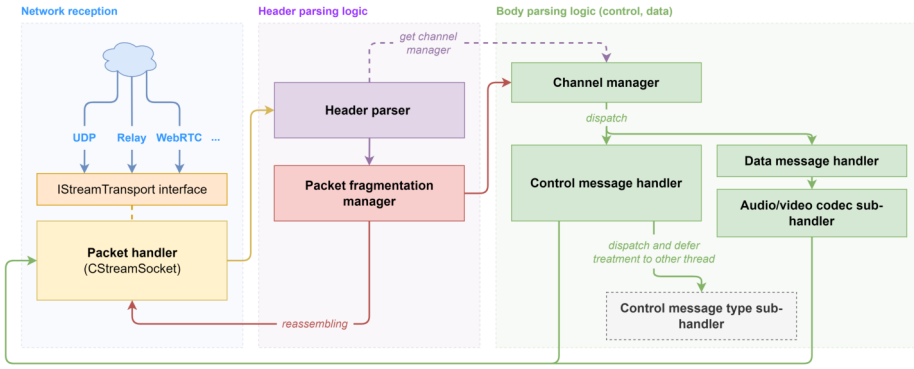

Fig. 5. High-level view of the data flow for incoming network packets in the Android client.

First, there is the network reception logic on the left, which depends on the chosen transport mode. It is characterized by an interface called `IStreamTransport` that implements primitives for sending and receiving data. This way, it does not matter whether direct UDP, SDR relay [8] or WebRTC was used for the P2P link: all packets end up at some point in the `CStreamSocket::HandlePacket` method.

[8] *Steam Datagram Relays* [15] is a feature from Valve that can be used as relay servers for Remote Play sessions.

Next, the purple block implements header parsing logic. The classes in this component reveal a lot of fields, mechanisms and concepts within the protocol: for example, flags, checksum (CRC32C), the existence of different packet types (*Connect*, *Disconnect*, *Data*, *Ack*...) and a system of *channels*.

Finally, after passing through different queues and systems related to reassembling and fragmentation, the packets land in `CStreamClient`'s `OnStreamPacket` method, where they are then handled differently depending on the *channel* they are associated to.

Channel system. Channels are an abstraction layer for the transport of parallel data, and are represented by identifiers between 0 and 31 (figure 6).

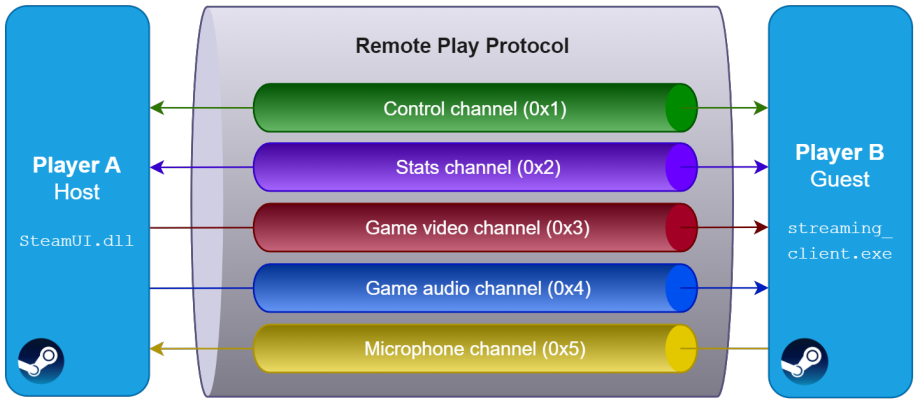

Fig. 6. The different channels in the Remote Play protocol.

A few channels are statically allocated, the most important ones being the *control channel* (0x1) and the *stats channel* (0x2). The stats channel allows to communicate statistics, events, debug logs and screenshots. As for audio and video streams, they are dynamically allocated a data channel (0x3-0x1f) upon request from the server (or the client for microphone audio data).

The control channel (0x1) is the channel that contains the most different types of messages (around a hundred). The enum `EStreamControlMessage` defines the *control* message types, which serve multifarious purposes:

— authenticating and negotiating upon connection;
— setting audio, video or network parameters;

— sending inputs (mouse, keyboard, controller, touchscreen);
— sharing information about the lobby (game, players);
— interacting with remote HID devices;
— editing the client's cursor, icon, window title...

Message format. The analysis of the components involved in header parsing allows to reconstruct the format of the packets that are exchanged. Figure 7 shows a typical example of what a packet looks like.

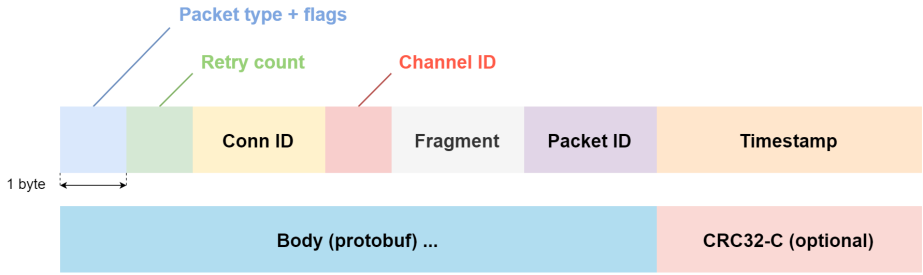

Fig. 7. Structure of a typical message in the Remote Play protocol.

The different fields will not be described in detail as the goal of this article is not to write a specification for the protocol. Some of these fields, such as *Connection ID*, *Fragment* or *Packet ID* are sequentially constrained values that can break the session if not crafted carefully.

Processing of control messages and cryptography. Zooming on the body parsing block from the higher-level view diagram (figure 5) yields the flowchart shown in figure 8. It describes how packets are dispatched in the client based on their channel and how control messages are handled.

Messages from the *control channel* are all encrypted, with the exception of *Authentication Request* (1), *Authentication Response* (2), *Client Handshake* (6) and *Server Handshake* (7). These are sent in plaintext because they are exchanged *before* the client is successfully authenticated.

For encrypted message types, control message bodies consist of 1 byte that indicates the message type (`EStreamControlMessage` enum) followed by the encrypted data:

$$\text{AES-CBC}(\text{S} \,||\, \text{M},\ \text{SessKey},\ \text{IV} = \text{HMAC-MD5}_{\text{SessKey}}(\text{S} \,||\, \text{M}))$$

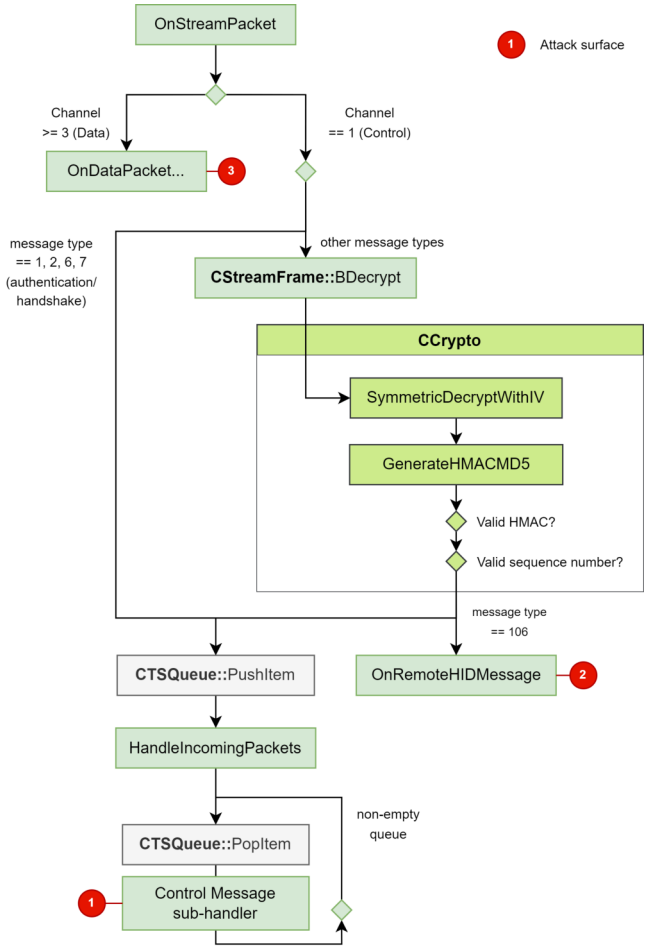

Fig. 8. Body parsing logic for packets received by the client.

M is the actual Protobuf message data. S denotes an 8-byte sequence number that is incremented every new control message. *SessKey* is a secret key shared between the client and server before the Remote Play session.[9]

Upon reception, the session key and the IV are used to decrypt the message. Then, the IV, which also happens to be an HMAC of the message, is used to verify its integrity. Finally, the sequence number is checked. If any of these verifications fail, the packet is discarded.

[9] How this key is agreed upon depends on the network transport mode and was not investigated much further.

There is a special kind of control message called `CRemoteHIDMsg`. It is related to remote HID device interaction and will be detailed further when mentioning attack surfaces.

The treatment of all other control messages is deferred, and later on, they are dispatched one at a time to their corresponding sub-handler.

Connection sequence diagram. Lastly, figure 9 shows a connection sequence diagram that was reconstructed for the protocol.

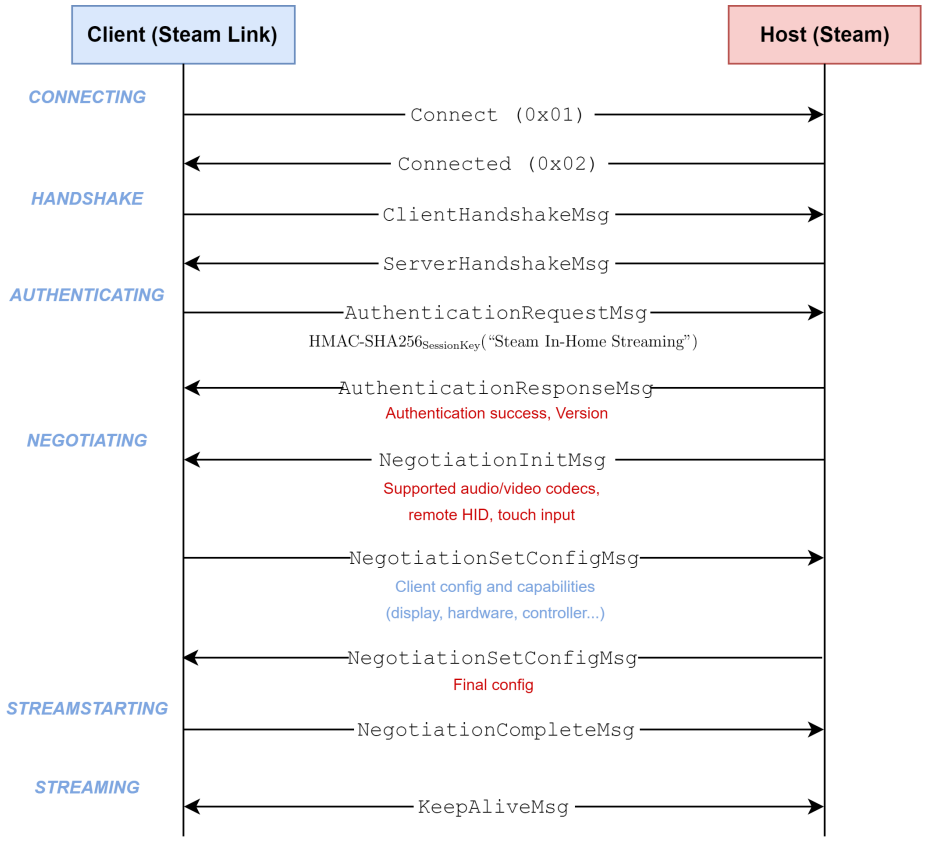

Fig. 9. Connection sequence diagram for the connection phase of the Remote Play protocol.

Both the server and the client rely on a state machine. After connecting and performing a handshake, the client must send an *Authentication Request*. It contains an HMAC of a constant magic string (`"Steam In-Home`

`Streaming"`) using the session key, which ensures the server that the client knows it and will be able to decrypt future messages.

Then, they exchange various settings, such as the audio/video codecs to use or whether to enable certain features, through a negotiation phase.

Once the configuration is done, the client and the server enter the *Streaming* state, where they can freely exchange control packets and audio/video data. Although the setup sequence can be subject to bugs, the *Streaming* state is more interesting for vulnerability research as it handles more complex data and exposes a larger attack surface.

3 Main attack surfaces

Three main attack surfaces, shown in table 1, can be accordingly identified inside the parsing of message bodies. Each one will be covered in more detail.

Attack surface	**Client to server**	**Server to client**
Control messages	$\sim$ 40 message types	$\sim$ 50 message types
Remote HID	5 message types	12 message types
Audio/video data	Microphone	Game audio and video

Table 1. Attack surfaces inside message body parsing.

Note: there are actually two more attack surfaces, which are the connection phase and the parsing of headers (including the channel management system and the packet fragmentation system). These surfaces are not very broad and were subject to manual vulnerability research. Nothing of interest was found.

3.1 Control messages

Control messages are the broadest and perhaps most valuable attack surface in Remote Play: there are almost a hundred different types of messages split between the client and the server. As stated earlier, they are all associated to a Protobuf structure.

While some control messages are rather short and straightforward, others are more intricate and good targets for vulnerability research. For instance, the message type shown in listing 1 features strings, bytes, index fields and an array of nested sub-messages — all of which could hide bugs (out-of-bounds accesses, integer overflows. . .).

Listing 1: The *Remote Play Together Group Update* message type.

```
message CRemotePlayTogetherGroupUpdateMsg {
  message Player {
    optional uint32 accountid = 1;
    optional uint32 guestid = 2;
    optional bool keyboard_enabled = 3;
    optional bool mouse_enabled = 4;
    optional bool controller_enabled = 5;
    repeated uint32 controller_slots = 6;
    optional bytes avatar_hash = 7;
  }

  repeated .CRemotePlayTogetherGroupUpdateMsg.Player players = 1;
  optional int32 player_index = 2;
  optional string miniprofile_location = 3;
  optional string game_name = 4;
  optional string avatar_location = 5;
}
```

3.2 Remote HID

Remote HID is a feature that allows the server to interact with the client's human interface devices, such as USB controllers. The protobuf definition for the `k_EStreamControlRemoteHID` message type is a rather enigmatic structure, shown in listing 2.

Listing 2: The `CRemoteHIDMsg` message type.

```
message CRemoteHIDMsg {
  optional bytes data = 1;
  optional bool active_input = 2;
}
```

Reversing shows that the `data` field is actually nested serialized Protobuf data. More specifically, it is either a serialized `CHIDMessageToRemote` message for client targets, or a serialized `CHIDMessageFromRemote` message for server targets. The definitions for these messages can be found in another file, `steammessages_hiddevices.proto`. These implement a whole sub-protocol.

Messages sent by the server. Listing 3 shows the example of the `CHIDMessageToRemote` message type. Among the 12 different sub-message types that can be nested, the server can ask the client to open a device, read from it, write to it, and obtain various metadata.

Listing 3: Commands in the `CHIDMessageToRemote` message type.

```
message CHIDMessageToRemote {
 optional uint32 request_id = 1;
 oneof command {
  DeviceOpen device_open=2;
  DeviceClose device_close=3;
  DeviceWrite device_write=4;
  DeviceRead device_read=5;
  DeviceSendFeatureReport device_send_feature_report=6;
  DeviceGetFeatureReport device_get_feature_report=7;
  DeviceGetVendorString device_get_vendor_string=8;
  DeviceGetProductString device_get_product_string=9;
  DeviceGetSerialNumberString device_get_serial_number_string=10;
  DeviceStartInputReports device_start_input_reports=11;
  DeviceRequestFullReport device_request_full_report=12;
  DeviceDisconnect device_disconnect=13;
 }
}
```

The actions that are performed and the data that is sent back obviously depend on the device that is plugged in, hence why this list of commands is actually an interface for which there exist multiple implementations, listed in table 2.

Implementation	**Can be triggered via...**
`CVirtualController`	Virtual touch device in the client settings
`CHIDDeviceSDLGamepad`	USB controller, handled by SDL
`CHIDDeviceSDLJoystick`	USB joystick, handled by SDL
`CHIDDeviceLocal`	Manually open device via SDL API or raw IOCTL

Table 2. Remote HID class implementations in the client depending on local device.

In terms of attack surfaces, only the first three implementations are of interest, as the local device one is entirely device-specific (no client logic). The caveat is that since each implementation is specific to a type of device, it is harder to look for bugs without owning or emulating them, and bugs themselves can highly depend on the client device itself which is not under control of the attacker.

Messages sent by the client. The client can send several `CHIDMessageFromRemote` messages, shown in listing 4. They can notably

announce a list of available devices (gamepad, joysticks...) through the `UpdateDeviceList` message. They can also answer a read request or a feature report request with specific data.

Listing 4: Remote HID commands sent by the client to the host.

```
oneof command {
  .CHIDMessageFromRemote.UpdateDeviceList update_device_list = 1;
  .CHIDMessageFromRemote.RequestResponse response = 2;
  .CHIDMessageFromRemote.DeviceInputReports reports = 3;
  .CHIDMessageFromRemote.CloseDevice close_device = 4;
  .CHIDMessageFromRemote.CloseAllDevices close_all_devices = 5;
}
```

Remote HID is therefore a tempting attack surface: it adds 17 new message types in total, and as their purpose is to interface with devices, they operate at a slightly lower level.

3.3 Audio/video data

In data channels, the sub-handling logic primarily depends on the codec that was selected by the channel opener. The tree diagram from figure 10 shows the different codecs and formats that are implemented in Remote Play.

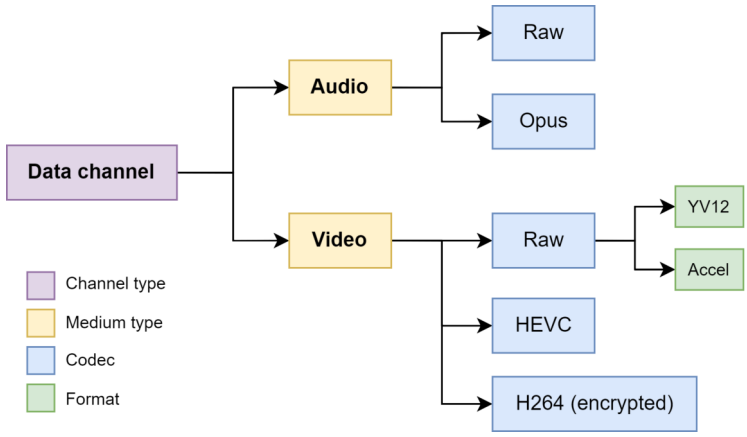

Fig. 10. Codecs and formats available in Remote Play for audio/video data.

The most interesting codecs are the raw ones, because they implement custom logic, unlike other codecs that usually leverage third-party libraries

(e.g. libopus). It is also worth noting most codecs actually do not implement encryption (which is rather odd since audio or video communications can carry sensitive data).

Data packets embed a whole new layer of data, encapsulated within an additional header. Although all the fields were not clearly figured out, the information listed in table 3 is enough to, as a server, be able to send audio or video data that the client understands and renders.

Field	**Size (bytes)**
Data message type	1
Sequence number 1	2
Timestamp	4
Unknown	6
Sequence number 2	2
Flags	1
Unknown	4

Table 3. Nested header structure for audio/video messages.

Timestamps have to be non-null and increasing. Some observations showed that the unknown fields were always null and that the sequence number fields were usually equal, except for audio data where the second one is null.

This new knowledge allows to trigger new paths in the binary related to decoding audio and video data for each codec/format. These are an interesting surface because such functions may be more prone to memory corruption bugs.

4 Implementing a custom client and server

This section explains how a client and a server for the Remote Play protocol were reimplemented in Python. Their first purpose was to easily play around with the protocol and send custom messages manually. These implementations eventually grew into an *ad hoc* fuzzer, which section 5 will be dedicated to.

4.1 Choice of transport mode

Section 2.2 briefly mentioned that Remote Play implements several modes of transport, given by the `EStreamTransport` enum. Some examples are UDP, relay UDP, WebRTC and SDR (*Steam Datagram Relays* [15]).

A few tests showed that the preferred network transport mode that was automatically used by the Remote Play client and server in Steam was SDR relaying. However, the most simple transport mode is direct UDP (`k_EStreamTransportUDP`), for clients and hosts that can communicate directly without need for a peer-to-peer setup or relays.

Using direct UDP allows to focus on the Remote Play protocol itself by circumventing any potential SDR or WebRTC abstraction, making it much easier to carry out tests and develop a custom client or server implementation that works locally.

4.2 Server reimplementation

Reimplementing a server for the Remote Play protocol allowed to interface with the official streaming client in Steam. The latter can be started from the command line by specifying the transport mode (UDP) along with the server's IP address and port. There is, however, a trick: by running the client from the command line directly, the key exchange scheme is somewhat bypassed and once the *Authenticating* state is reached, the client crashes because it cannot find any session key to load.

Listing 5: First method in which the client uses the session key.

```
int CStreamClient::StartAuthentication(CStreamClient *this) {
  CAuthenticationRequestMsg Msg; // [sp+8h] [bp-58h] BYREF
  _BYTE hmac[32]; // [sp+34h] [bp-2Ch] BYREF

  CStreamClient::SetSessionState(this, AUTHENTICATING);
  CAuthenticationRequestMsg::CAuthenticationRequestMsg(Msg);
  CCrypto::GenerateHMAC256(
    "Steam In-Home Streaming", strlen("Steam In-Home Streaming"),
    this->SessionKey, this->SessionKeySize, hmac
  );
  CAuthenticationRequestMsg::set_token(Msg, hmac, 0x20);
  CStreamClient::SendControlMessage(
    this, k_EStreamControlAuthenticationRequest, Msg
  );
}
```

To address this issue, one can look for the first time in the client where the session key is supposed to be used: the `StartAuthentication` method,

shown in listing 5. Indeed, as seen earlier in the protocol's connection sequence diagram (figure 9), the client needs in this state to authenticate to the host, which involves computing an HMAC using the session key.

At this point in time, just before `CCrypto::GenerateHMAC256` is called, a custom session key can be injected inside the `CStreamClient` structure. To this purpose, using x32dbg's scripting engine [16], it was possible to inject a 32-byte null key in the client on start-up.[10] This is shown in listing 8 in appendix.

Once this issue has been addressed, the whole protocol can be reimplemented by leveraging the Protobuf definitions at disposal. It requires going through the whole connection phase and implementing a few basic messages (such as *Keep Alive*), before being able to send custom control messages to the client.

4.3 Client reimplementation

In order to target the server implementation in Steam, reimplementing a client required a little bit more work. Interfacing with a Remote Play server that was directly started through Steam's invite mechanism inside a game is rather difficult as the session will be built over SDR or WebRTC. There is, however, a way to circumvent this issue: forcing a direct UDP connection by hijacking a local *Steam Link* key.

Steam Link uses a separate protocol, called the *Steam In-Home Streaming Discovery Protocol*, for clients to discover devices that are available for streaming on a local network.[11] Through this protocol, a client can specify a list of connection transport modes that they support: this way, one can ensure that the connection will use direct UDP. This article will not go through the extent of detailing the whole discovery protocol. However, this protocol had to be reversed and reimplemented as well in order to be authenticated on a local Steam instance.

Indeed, when a client discovers and connects to a machine, the device gets *paired* to the machine and shares a dedicated *discovery key*. It is possible to borrow a discovery key and plug it inside the discovery protocol to go through. The server eventually answers a message that contains the port to connect to for the Remote Play session, as well as a randomly generated session key that can be decrypted using the client discovery key.

The rest of the client implementation process is quite similar to the server one.

[10] This could also very well be achieved through other techniques such as patching or DLL injection.

[11] Or remotely, through a PIN code system.

5 Implementing a dedicated fuzzer

5.1 `rpfuzz`: a fuzzer for the Remote Play protocol

The client and server reimplementations that were developed and detailed in the previous section were extensively used to play around with the protocol, and naturally evolved into a basic fuzzer, which was given the name `rpfuzz` (for *remote play fuzzer*).

The idea was to keep on playing around with the protocol by writing a little fuzzer on top of the existing code from scratch, to see if we could stumble upon "quick wins" by randomly mutating Protobuf messages. Figure 11 describes `rpfuzz`'s software architecture.

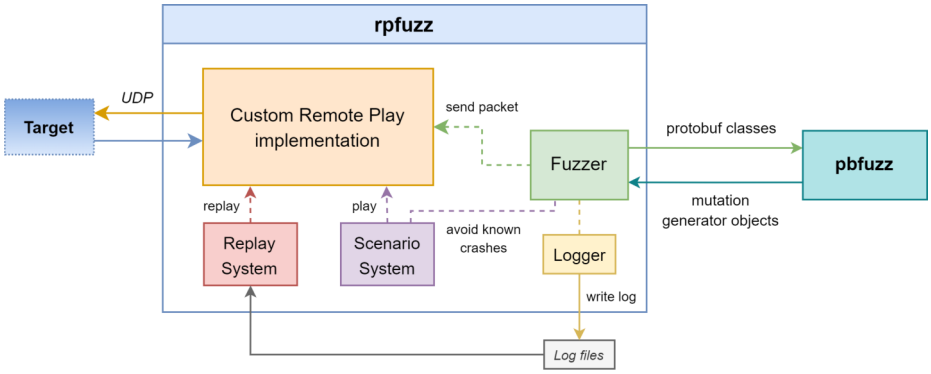

Fig. 11. `rpfuzz`'s software architecture.

Network component. The network component (orange block) is interchangeable: depending on the target, it can be replaced by a server implementation, or by a client implementation (coupled with the discovery protocol). It is in charge of communicating with the target.

Fuzzer component. The *Fuzzer* component runs on a separate thread. It supports both control messages and audio/video channels.

Control message fuzzing is essentially stateless. It consists of a loop that randomly chooses a message type and passes on the associated protobuf class over to a mutation engine: `pbfuzz`, which will be covered more in depth in the next sub-section. `pbfuzz` sends back a Python object to generate an endless amount of mutations, which are assembled into messages and then sent to the target through the network implementation.

On the other hand, fuzzing remote HID messages requires stateful actions, such as sending a request to open a device. In the same way, fuzzing audio/video channels requires opening a new dynamic channel.

Replay, scenario and logging systems. A few other nice features were implemented, namely a replay system, a scenario system and a logging system.

The *Logger* basically just saves all the sent mutations to a file for a fuzzing session. It helps keeping a fuzzing history. This is useful for debugging and analyzing crashes.

These logs can also be fed back to the *Replay System*, which will replay all the messages from a session one at a time. This can prove useful to try and reproduce a (deterministic enough) crash, by hopefully bringing the target to a state that has already been reached before.

Finally, a *Scenario System* was designed to write specific scenarios and play them at any time. It was especially useful to reproduce bugs and write proofs of concept. Besides, each bug scenario can specify a condition that should be necessarily verified by messages that trigger the associated bug. Thanks to this, the fuzzer knows when to avoid specific messages, and is not slowed down by already-found crashes.

5.2 `pbfuzz`: a custom Protobuf mutation engine

`pbfuzz` is — you guessed it — another unconventional name standing for *protobuf fuzzer*. One of the challenges usually brought by fuzzing network state machines is that of *grammatical awareness*. In our case, existing mutational engines inside fuzzing frameworks can definitely not be adopted out-of-the-box, as they would break the messages' structures. Even worse, they would totally disfigure all Protobuf serialized data, hence the need for a Protobuf-aware mutational engine.

The choice was made to write a custom Protobuf mutation engine from scratch for more flexibility, better integration and educational purposes. Other contenders for this component include `libprotobuf-mutator` [5], which was not investigated and could constitute a valid alternative, and ProtoFuzz [11], a Python library that ended up lacking flexibility for our use case, and in which there were too many bugs (such as broken support of repeated fields).

At its core, `pbfuzz` relies on playing with inner objects and attributes of Google's protobuf module, in order to walk through message descriptors, types, labels. Although the fuzzer is model-based and does not require input

seeds (the Protobuf definitions are known in advance), several mutation strategies were implemented for each field type, taking inspiration from traditional model-less mutation engines (as described in *The Art, Science, and Engineering of Fuzzing*, section 5.2 [8]).

Strings and bytes fields can undergo bit flips, byte substitutions, trimming, or insertion of random or "interesting" data like string formatters (`%x`, `%s`, `%n`), paths, URLs, XML, JSON... of random length. These could trigger buffer overflows, format string vulnerabilities, logic bugs, or other kinds of more higher-level bugs.

Integer fields and floats are also mutated with interesting values, depending on bit size (32, 64) and signedness, opening up for integer overflows or out-of-bounds accesses.

Repeated fields (lists) can go through single mutations (only one element of the list is mutated), random trims or random insertions of random lengths.

Finally, nested message fields are mutated recursively, and fields marked as optional can be deliberately omitted at random.[12]

5.3 Fuzzing results and crash analysis

Performance and surface reached. The fuzzing speed was limited by the target's packet processing speed; in other words, the target acted as a bottleneck and the fuzzing speed had to be adjusted manually not to cause an overload. Still, the fuzzer was able to send around 100 messages per second without overworking the target too much.

In terms of surface, all the control messages were successfully reached, with a few exceptions being obsolete or unimplemented message types. Audio/video codecs were also all reached, except for the raw accelerated graphics format and the HEVC codec, which channels could not be opened.

Areas for improvement. The fuzzer can benefit from multiple improvements: for instance, it does not feature any dynamic instrumentation ability or code coverage, and it is not able to synchronize with the target either. But even though `rpfuzz` is rather naive and black-box driven, it proved to be largely sufficient to uncover several bugs, as discussed in the next section.

`pbfuzz` also comes with its own set of limitations. Namely: it only supports Protobuf 2, does not implement some concepts (like unions, maps

[12] Indeed, a program could try accessing fields from a deserialized object without verifying whether they are actually present, leading to potentially unexpected behavior.

or extensions that are practically non-existent in Remote Play), and could also feature better string mutators. However, it is rather efficient and malleable, and could be reused to fuzz other targets that feature Protobuf communications.

Analyzing crashes. A debugger was attached to the target in order to intercept crashes and analyze them. *PageHeap* [9] was also enabled on the target, which is a must-have to keep track of any out-of-bounds read or write access in the heap.

Another nice tool to have in the toolbox to analyze certain crashes, or bigger schemes that involve more convoluted control flows, is *Time Travel Debugging* [10]. More specifically, the *ttddbg* [2] plugin for IDA is neat: it supports loading a TTD trace and debugging it.

6 Bugs found

Table 4 lists some of the bugs that were found thanks to `rpfuzz`, along with a brief description and their impact. They affect Remote Play Together in the Steam client, and also the Steam Link product.

Most of these bugs should be reproducible on other platforms (Linux, Android, iOS), although it was not verified for all of them. In the *scenario* column, H2C means host-to-client (client victim) and C2H means client-to-host (host victim).

Scn.	**Description**	**Impact**
H2C	`CRPTogetherGroupUpdateMsg` format string	Remote memory leak
H2C	`CRPTogetherGroupUpdateMsg` request forgery	Info leak (at least)
H2C	Integer overflow	Unexploitable
H2C	Heap overflow	Heap leak
H2C	Heap overflow	Local heap leak
H2C	Heap overflow	Local heap leak
H2C	Heap overflow	Unexploitable
C2H	Format string	Local memory leak

Table 4. List of bugs found in the Remote Play client and server implementations.

A lot of other insignificant bugs (such as DoS from null pointer dereferences or arbitrary malloc) were not included.[13]

[13] Likewise, all the listed bugs also usually lead to DoS, but crashing the streaming client or the Steam client doesn't really have any security impact.

Because of responsible disclosure policy, only the first two bugs from the list will be detailed further; at the time of writing this article, the other bugs cannot yet be communicated on.

6.1 Format string bugs in `CRemotePlayTogetherGroupUpdateMsg`

In section 3.1 was given the example of the *Remote Play Together Group Update Message* (listing 1) as a rather complex message type that could conceal many bugs. Of course, this was done on purpose, because it is really full of bugs. This message is basically sent by the session host to notify the guest of various elements, such as who the players in the current session are and where their avatars are stored, as shown in figure 12.

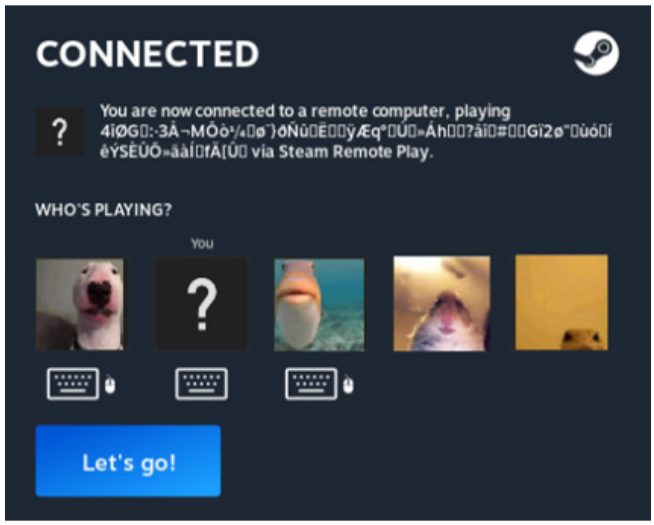

Fig. 12. Pop-up window with the *group update* information sent by the server.

There are two distinct format string vulnerabilities in the code that handles this message. In the `CMiniProfileLoader::LoadProfiles` function, a loop iterates over a list of player objects (listing 6).

Listing 6: Loop on players in the *Mini Profile Loader* component.

```
while (n_players--) {
  Player = *Players++;
  if (Player->accountid) {
    CMiniProfileLoader::LoadAccountProfile(
      this,
      RPTogetherGroupUpdateMsg->miniprofile_location,
      Player->accountid
    );
  } else if (Player->guestid) {
    binarytohex(Player->avatar_hash, avatar_hash_size, hash_hex);
    CUtlString::Format(
      url, RPTogetherGroupUpdateMsg->avatar_location, hash_hex
    );
    CMiniProfileLoader::LoadGuestProfile(
      this,
      url,
      Player->guestid
    );
  }
}
```

When an `accountid` field is provided in the current player object in the list, the `LoadAccountProfile` method eventually calls:

```
CUtlFmtString::CUtlFmtString(url, miniprofile_location, accountid);
```

The attacker-controlled `miniprofile_location` field is naively used as a string formatter. They also control the first argument to the format string (`accountid`). Therefore, the host can leak arbitrary memory from the process, using formatters such as `%x` and `%s`.[14] Then, the formatted string is used as a URL and a CURL request is performed.

There are two ways the attacker can retrieve back the leaks: by exfiltrating over HTTP (e.g. set `miniprofile_location` to `http://evil/%x` and log the request), or by reading received debug strings over the *Stats* channel. The second option is the easiest one to carry out because it happens automatically. Indeed, by setting the `miniprofile_location` field to `"Leak: %08x.%08x.%08x.%08x"`, the CURL request fails and a debug string that looks like the following is output:

```
Web request Leak: 13374242.11fe0ff0.11fe0fec.13374242 failed, CURL
↪  error code 3, HTTP error code 0
```

[14] Unfortunately, the `%n` formatter is disabled by default, thus no write primitive.

In the *Stats* channel exists a type of message used to send over logs, called `CLogMsg` (listing 7). It happens that the streaming client always sends automatically all its debug strings over to the host via this message type. Therefore, the attacker retrieves the leaks without effort.

Listing 7: *CLogMsg* message sent by the client.

```
message CLogMsg {
  optional int32 type = 1;
  optional string message = 2;
}
```

The second format string vulnerability is highly similar to the first one, and thus will not be detailed further. It is found in the `avatar_location` field defined for guest players.

Impact-wise, these vulnerabilities allow an attacker to reliably break ASLR on the victim's machine, which is often the first step to an RCE exploit. More particularly on Windows, breaking ASLR for various Steam-related DLLs (like `steamclient.dll`) can greatly help to further compromise the system in any other attack targeting the Steam client. An attacker could also practically leak anything in the process' memory, including potentially sensitive data (environment variables, paths, tokens...).

Valve patched these vulnerabilities by denying URLs that contain the character `%`. Although this does fix the issue, this doesn't address the fact that having a user-controlled string formatter is a bad programming habit.

6.2 Request forgery in `CRemotePlayTogetherGroupUpdateMsg`

This vulnerability is a direct follow-up to the previous one: since the `miniprofile_location` field is a fully host-controlled URL, an attacker can make the client perform an arbitrary HTTP(S) GET request.[15]

At this point, the client expects the CURL response to be a valid JSON file, which will in turn be parsed. However, if the response is *not* a valid JSON string, the following debug string will be output and sent back to the attacker, again, through the *Stats* channel:

```
Couldn't parse profile data: syntax error near: <RESPONSE CONTENTS>
```

Therefore, an attacker can exfiltrate the response to any HTTP GET request performed client-side (as long as the output is not JSON). This gives a so-called *Client Side Request Forgery* primitive, which could be

[15] (Un)fortunately, other wrappers like `file://` are disabled by CURL's configuration.

leveraged to leak potentially sensitive data hosted on local web pages or over an internal network. An attacker could also scan the victim's internal network for recon (e.g. port scanning). If a vulnerable internal service is found, they could even pivot by exploiting it through GET requests (e.g. SQL injection in GET parameter). The possibilities are endless as the attacker can send as many payloads as they want silently.

Valve patched this vulnerability by introducing a whitelist domain filter. No trivial bypass was found in the patch.

7 Reporting to Valve

Two distinct reports were sent to Valve through the HackerOne platform. The first one reported the format string and request forgery vulnerabilities in `CRemotePlayTogetherGroupUpdateMsg`, along with a reliable proof-of-concept. The second one reported all the other bugs with a lesser impact, without proofs-of-concept.

Valve quickly validated the first report (perhaps contrary to popular belief), assessing the vulnerabilities as *Critical* and paying the corresponding bounty. However, it took several months without news and multiple follow-up messages for them to deploy a working fix. In other words, they didn't seem in too much of a hurry to patch vulnerabilities that they classified as *Critical*.

Regarding the second report, Valve have yet to reach a final decision. For the time being, they won't fix the bugs because of the lack of reproducible proofs-of-concept, and although many follow-up messages have been sent, they still won't answer regarding disclosure, hence why not including them in this article was the preferred choice. We believe that the HackerOne triaging process makes it a lot harder to reach the team and does not promote transparency in the best way.

8 Conclusion

In this article, we have covered several captivating aspects of vulnerability research:

— choosing a target and delimiting an attack surface;
— reverse engineering a product to bring out its software architecture;
— analyzing a protocol and constructing a partial specification;
— deducing a minimalistic implementation to talk to the target;
— building a fuzzer upon all this work;
— investigating crashes and exploiting bugs;

— assessing risk and reporting to the editor.

Starting from zero and being able to progressively disentangle so much hidden knowledge is always a very satisfying feeling.

This particular work also highlights that a simplistic homemade fuzzer is sometimes enough to quickly get encouraging results. Of course, the fuzzer could be enhanced further by integrating tools such as *Frida* [4] to feature dynamic instrumentation, or even directly by leveraging existing black-box fuzzers (such as *WinAFL* [18], *Jackalope* [17], *what the fuzz* [1]...).

Remote Play is still an evolving product with frequent updates, so keep an eye out for more!

References

1. Axel '0vercl0k' Souchet. what the fuzz: a distributed, code-coverage guided, customizable, cross-platform snapshot-based fuzzer designed for attacking user and / or kernel-mode targets running on Microsoft Windows. `https://github.com/0vercl0k/wtf`.
2. Airbus CERT. ttddbg: Time Travel Debugging IDA plugin. `https://github.com/airbus-cert/ttddbg`.
3. Brian Dean. Steam Usage and Catalog Stats for 2022. `https://backlinko.com/steam-users#steam-daily-active-users`.
4. Frida. Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. `https://frida.re/`.
5. Google. libprotobuf-mutator: Library for structured fuzzing with protobuffers. `https://github.com/google/libprotobuf-mutator`.
6. Google. Protocol Buffers Documentation. `https://protobuf.dev/`.
7. HackerOne. Hacktivity on Valve's Bug Bounty Program. `https://hackerone.com/valve/hacktivity`.
8. Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018. http://arxiv.org/abs/1812.00140.
9. Microsoft. GFlags and PageHeap. `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags-and-pageheap`.
10. Microsoft. Time Travel Debugging - Overview. `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/time-travel-debugging-overview`.
11. Trail of Bits. ProtoFuzz: A Protobuf Fuzzer. `https://blog.trailofbits.com/2016/05/18/protofuzz-a-protobuf-fuzzer/`, 2016.
12. SteamDB project. Protobufs: Automatically tracked protobufs for Steam and Valve's games. `https://github.com/SteamDatabase/Protobufs`.
13. SteamDB project. SteamDB: database of everything on Steam. `https://steamdb.info/`.

14. Valve. Game Networking Sockets: Reliable & unreliable messages over UDP. Robust message fragmentation & reassembly. P2P networking / NAT traversal. Encryption. https://github.com/ValveSoftware/GameNetworkingSockets.
15. Valve. Steamworks Documentation: Steam Datagram Relay. https://partner.steamgames.com/doc/features/multiplayer/steamdatagramrelay.
16. x64dbg. x64dbg Documentation: Scripts Commands. https://help.x64dbg.com/en/latest/commands/script/.
17. Ivan Fratric (Google Project Zero). Jackalope: Binary, coverage-guided fuzzer for Windows and macOS. https://github.com/googleprojectzero/Jackalope.
18. Ivan Fratric (Google Project Zero). WinAFL: a fork of AFL for fuzzing Windows binaries. https://github.com/googleprojectzero/winafl.

A Appendix

Listing 8: x32dbg script to inject a session key in the client.

```
erun

// Reach CStreamClient::StartAuthentication, just before the call
// to CCrypto::GenerateHMAC256. Offset depends on Steam version.
// ebx needs to point the CStreamClient structure.
bp streaming_client:<offset>
erun

// New key (full null bytes)
alloc 32
$key = $result
fill $key, 0, 32

// Copy new key addr
mov dword:[ebx + 0x24], $key

// Copy key size (0x20)
set (ebx + 0x34), #20 00 00 00#

// Resume execution
erun
```