Real-time Steganography with RTP

September, 2007

I)ruid, C^{2}ISSP
<druid@caughq.org>
http://druid.caughq.org

Abstract

Real-time Transfer Protocol (RTP) is used by nearly all Voice-over-IP systems to provide the audio channel for calls. As such, it provides ample opportunity for the creation of a covert communication channel due to its very nature. While use of steganographic techniques with various audio cover-medium has been extensively researched, most applications of such have been limited to audio cover-medium of a static nature such as WAV or MP3 file audio data. This paper details a common technique for the use of steganography with audio data cover-medium, outlines the problem issues that arise when attempting to use such techniques to establish a full-duplex communications channel within audio data transmitted via an unreliable streaming protocol, and documents solutions to these problems. An implementation of the ideas discussed entitled SteganRTP is included in the reference materials.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This paper describes a research effort within the disciplines of steganography, Internet telephony, and data communications.

1.1 Overview

This paper is structured in the following order: The first chapter provides an introduction, describes the motivation for this research, and covers some basic concepts and terminology for the subjects of Voice over IP (VoIP), Real-time Transport Protocol (RTP), Steganography, and, more specifically, the use of steganography with an audio cover-medium. The second chapter defines the concept of real-time steganography, discusses using steganography with RTP, and describes some of the identified problems and challenges. The third chapter details the reference implementation entitled SteganRTP including a description of the project's goals, the implementation's operational architecture, process flow, message data structure, and functional sub-systems. The fourth chapter addresses the identified problems and challenges that were met and describes how they were solved. The fifth and final chapter concludes the paper with observations made as a result of this research effort.

1.2 Voice over IP

The term Voice over IP (VoIP) is nearly synonymous with Internet Telephony. The majority of VoIP systems are designed to utilize separate signaling and media channels to provide calling services to users. The signaling channel is generally used to set-up, manage, and tear-down calls between two or more parties whereas the media

channel is used to transmit the audio, video, or other media that may be associated with the call. A number of competing protocol standards exist for use as the VoIP system's signaling channel which include Session Initiation Protocol[?] (SIP), H.323[?], Skinny[?], and many others. Real-time Transport Protocol[?] (RTP), however, is used almost ubiquitously to provide VoIP systems with the required media channel.

1.3 Real-time Transport Protocol

Real-time Transport Protocol[?] (RTP) is described by the protocol authors as "a transport protocol for real-time applications." RTP provides an end-to-end network transport suitable for applications transmitting real-time data such as audio, video or any other type of streamed data. RTP generally utilizes the User Datagram Protocol[?] (UDP) for its transport and can do so in both multicast or unicast network environments. When employed by a VoIP system, RTP generally handles the media channel of a call. The call's media channel is generally handled independent of the VoIP signaling channel. However, per the RTP specification, there are no default network ports defined. As such, the RTP endpoint network ports must be negotiated between the endpoints via the signaling channel. Other events in the signaling channel may also influence the operation of the media channel as handled by RTP such as requests to change audio encoding, add or remove parties from the call, or tear down the call.

One of RTP's current deficiencies is that it is entirely clear-text while traversing the network. An RTP profile has been defined for encrypting parts of the RTP data packet called Secure Real-time Transport Protocol[?] (SRTP). However, the specification defines no mechanism for negotiating or securely exchanging keying information to be used for the encryption and decryption processes. At the time of this writing, a number of keying mechanisms have been defined but no standard has either been agreed upon by the standards bodies or determined by the free market. As such, most implementations of RTP do not currently use the SRTP profile and instead continue to transmit call media data in the clear. As will be detailed in full in Section 3.2, this property of the media channel provides ample opportunity for multiple types of operational scenarios where unknown third-parties to the legitimate callers may hijack all or part of the call's media traffic for transmission of covert communications. Making use of this blatantly insecure property of RTP is the primary motivation for this research effort.

1.4 Steganography

The term steganography originates from the Greek root words *"steganos"* and *"graphein"* which literally mean "covered writing". As a sub-discipline of the aca-

demic discipline of information hiding, the primary goal of steganography is to hide the fact that communication is taking place[?, ?, ?] by concealing a message within a cover-medium in such a way that an observer can not discern the presence of the hidden message.

Conversely, steganalysis is the act of attempting to detect a concealed message which was hidden via the use of steganographic techniques[?], thus preventing a steganographer from achieving their primary goal. Common steganalysis techniques include statistical analysis of the properties of potential stego-medium, statistical analysis of extracted potential message data for properties of language, and many others such as specific techniques that target known steganographic embedding methods.

1.4.1 Terminology

The following terminology as used in the discipline of steganography and steganalysis has been set forth over many years of compounding research[?, ?, ?]. As such, the following terminology will be used consistently within this research paper:

1. *Cover-medium* - Data within which a message is to be hidden.
2. *Stego-medium* - Data within which a message has been hidden.
3. *Message* - Data that is or will be hidden within a stego-medium or cover-medium, respectively.
4. *Redundant Bits* - Bits of data in a cover-medium that can be modified without compromising that medium's integrity.

1.4.2 Digitally Embedding

Digitally embedding a message into a cover-medium usually involves three basic steps. First, the redundant bits of the target cover-medium must be identified. Second, it must be decided which of the identified redundant bits are to be utilized. Finally, the bits selected for use must be modified to store the message data. In many cases, a cover-medium's redundant bits are likely to be the least-significant bit or bits of each of the encoded data's word values.

1.5 Steganography With Audio

Media formats in general, and audio formats specifically, tend to be very inaccurate data formats simply because they do not need to be accurate; the human ear is

not very adept at differentiating sounds. As an example, an orchestra performance which is recorded with two separate recording devices will produce vastly different recordings when viewed digitally, but will generally sound the same when played back if they were recorded in a similar manner. Due to this inherent inaccuracy, changes to an audio bit-stream can be made so slightly that when played back the human ear won't be able to distinguish the difference between the cover-medium audio and the stego-medium audio.

With many audio formats, the least-significant bit from each audio sample can be used as the medium's redundant bits for the embedding of message data. To illustrate, assume that an audio file encoded with an 8-bit sample encoding has the following 8 bytes of data in it, which will be used as cover-data:

```
0xb4 0xe5 0x8b 0xac 0xd1 0x97 0x15 0x68
```

In binary this would result in the following bit-stream:

```
10110100 11100101 10001011 10101100 11010001 10010111 00010101 01101000
```

In order to hide the message byte value `0xd6`, or `11010110` in binary, each sample word's least-significant bit would be modified to represent all 8 bits of the message byte:

```
10110101 11100101 10001010 10101101 11010000 10010111 00010101 01101000
```

The modifications result in the following 8 bytes of stego-data:

```
0xb5 0xe5 0x8a 0xad 0xd0 0x97 0x15 0x68
```

When compared to the original 8 bytes of cover-data, it is noticeable that on average only half of the bytes of data have actually changed value, however the resulting stego-data's least-significant bits contain the entire message byte. It is also noticeable that when utilizing this embedding method with a cover-medium with these word size properties, the cover-medium must be at least eight times the size of the message in order to successfully embed the entire message.

1.5.1 Previous Research

Audio Steganography

Much research has been done in the field of steganography utilizing an audio cover-medium. Techniques such as using audio to convey messages in both the human

audible and inaudible spectrum as well as various methods for the digital embedding of information into the audio data itself have all been explored; so much in fact that many methods are now considered standard. Many of the most recent implementations cannot be considered to advance the state of research in the area as they generally only implement the standard methods.

It is important to note that the significant majority of previous research in the sub-discipline of audio steganography, however, has focused on static, unchanging audio data files. Tools such as S-Tools[**?**], MP3Stego[**?**], Hide 4 PGP[**?**], and many others, are just such implementations, employing standard embedding methods with WAV, MP3, and VOC audio file cover-mediums, respectively. Very few practical implementations have been developed that utilize audio steganography with a cover-medium that is in a flux state or within streaming or real-time media sessions.

VoIP Steganography

A few previous research efforts have been made to employ steganography with various VoIP technologies. A complete analysis of such efforts identified prior to embarking upon the research presented in this paper has previously been provided[**?**]. In summary, most identified research efforts were utilizing steganographic techniques but not achieving the primary goal of steganography or otherwise employing steganographic techniques to accomplish an otherwise overt goal.

Chapter 2

Real-time Steganography

This paper defines "real-time" use of steganography as the utilization of steganographic techniques to embed message data within an active, or real-time, media stream. The research and reference implementation presented herein focuses on VoIP call audio as the active media stream being targeted as cover-medium.

Nearly all uses of steganography targeting audio cover-medium in general, or VoIP cover-medium specifically, that were evaluated prior to performing this research were found to operate on a target cover-medium as a storage channel and provided separate "hide" and "retrieve" modes. In addition, most cover-medium that were targeted by such implementations were of a static nature such as WAV or MP3 files or were unidirectional such as streaming stego-audio to a recipient.

A few weeks prior to the research contained herein being initially presented[?] at the DEFCON 15[?] hacker conference on August 3rd through 5th 2007, another use of steganography in a real-time fashion was made public via a research effort entitled Vo^{2}IP[?]. An analysis of this research effort and its deficiencies has been included in an updated version of the previously mentioned analysis paper[?].

2.1 Context Terminology

The disciplines of steganography and data networking share some common terminology which have different meanings relative to each discipline. This paper discusses research that lies within the realm of both disciplines, and as such will use terms that may be confusing when taken out of context. The following terms are defined here and used consistently without to prevent confusion when interpreting the content of this paper.

1. *Packet* - Used in the data networking sense; A data packet which is routed through a network, such as an IP/UDP/RTP packet.
2. *Message* - Used in the steganography sense; Data to be hidden or retrieved.

2.2 RTP Payload Redundant Bits

RTP packet payloads are essentially encoded multimedia data. RTP payloads may contain any type of multimedia data. However, this research effort focused entirely on audio. Specifically, audio encoded with the G.711[**?**] Codec. Any number of audio Codecs can be used to encode the RTP payload, the identifier of which is included in the RTP packet's header as the *payload type* (PT) field.

The frequency, locations, and number of redundant bits found within the RTP packet's encoded payload are determined by the Codec that is used to encode the audio transmitted by an individual packet. The Codec focused on during this research, G.711, uses a 1-byte sample encoding and is generally resilient to modifications to the least significant bit[**?**] (LSB) of each sample. Codecs with larger samples may provide for one or more bits per sample to be modified without any discernible audible change in the encoded audio, which is defined as the audio's *audible integrity*.

2.2.1 Audio Word Size

The data value word size, or *sample size* in audio terminology, used by various audio encoding formats is one factor in determining the amount of available space within the cover-medium for embedding a message. Generally only the least significant bit of each word value can be expected to be modifiable without any perceptible impact to audible integrity. Thus, only half the amount of available space in an audio cover-medium encoded in a format with a 16-bit word size will be available in comparison with a cover-medium with an 8-bit word size.

2.2.2 Common VoIP Audio Codecs

For reference, some common VoIP audio Codecs and their encoding and sample properties[**?**] are listed in Table 2.1 below.

Table 2.1: Common VoIP Audio Codecs

Codec	Standard by	Bit Rate (kb/s)	Sample Rate (kHz)	FrameSize (ms)
G.711	ITU-T	64	8	Sampling
G.721	ITU-T	32	8	Sampling
G.722	ITU-T	64	16	Sampling
G.722.1	ITU-T	24/32	16	20
G.723	ITU-T	24/40	8	Sampling
G.723.1	ITU-T	5.6/6.3	8	30
G.726	ITU-T	16/24/32/40	8	Sampling
G.727	ITU-T	variable		Sampling
G.728	ITU-T	16	8	2.5
G.729	ITU-T	8	8	10
GSM 06.10	ETSI	13	8	22.5
LPC10	U.S. Gov	2.4	8	22.5
Speex (NB)		8, 16, 32	2.15 - 24.6	30
Speex (WB)		8, 16, 32	4 - 44.2	34
iLBC		8	13.3	30
DoD CELP	U.S. DoD	4.8		30
EVRC	3GPP2	9.6/4.8/1.2	8	20
DVI	IMA	32	Variable	Sampling
L16		128	Variable	Sampling

2.2.3 G.711 (alaw/ulaw)

The G.711 audio Codec is a fairly straight-forward sample-based encoding. It encodes audio as a linear grouping of 8-bit audio samples arranged in the order in which they were sampled.

Throughput

Utilizing the LSB of every sample in a G.711 encoded RTP payload, which is commonly of 160 bytes in size, a total of 20 bytes of message data can be successfully embedded. Given an average of 50 packets per second unidirectional, this results in approximately 1,000 bytes of full-duplex throughput of message data within the established covert channel.

2.3 Identified Problems and Challenges

Many problems and challenges that arise when considering the use of steganography with RTP stem from properties of the underlying transport mechanism, the nature of real-time audio, or the RTP protocol itself. The following sections outline various problems and challenges that were identified when attempting to use steganography with RTP.

2.3.1 Unreliable Transport

One of the most significant challenges to utilizing RTP packet payloads as cover-medium is that RTP generally employs UDP as its underlying transport protocol. This is appropriate for a streaming multimedia protocol, however it is less than ideal for a reliable covert communications channel. UDP is a datagram messaging protocol which is considered connectionless and unreliable[**?**]. As such, each packet's successful delivery and order of arrival is not guaranteed. Any message data which is split across multiple RTP cover-packets may arrive out of order or not arrive at all.

2.3.2 Cover-Medium Size Limitations

The RTP protocol, being designed for "real-time" transport of media, behaves like a streaming protocol should. RTP datagram packets are relatively small and there are usually tens to hundreds of packets sent per second in the process of relaying audio between two peers. Additionally, different audio Codecs provide for different encoded audio sample sizes, resulting in a variable amount of available space for embedding which is dependent upon which Codec the audio for any individual RTP packet is encoded with. Due to the small size of these packets and the common constraint among many steganographic embedding methods which limits the amount of data that is able to be embedded to a fraction of the size of the cover-medium, a very limited amount of space is actually available for the embedding of message data. As such, large message data will inevitably be required to be split across multiple cover-packets and thus must be reassembled at its destination.

2.3.3 Latency

RTP is, by design, extremely susceptible to media degradation due to packet latency. As such, any processing overhead from the embedding of message data into the cover-medium or delay due to inspection of potential cover-medium packets may have a noticeable impact on the end-user's quality of experience. When manipulating an RTP stream between two endpoints that are expecting packet delivery in

a timely manner, a steganographic system cannot be overly invasive when packets are not needed for embedding and must be efficient at its task when they are.

2.3.4 Tracking of RTP Streams

In normal operation, RTP establishes two packet streams to form a session between two endpoints. Each endpoint uses one stream to send multimedia data to the other, thus achieving full-duplex communication via two unidirectional packet streams. When identifying an RTP session to be utilized as cover-medium for a full-duplex covert communications channel, the two paired streams must be correctly identified and tracked.

2.3.5 Raw vs. Compressed Audio

It is important to consider that audio being transported via RTP may be compressed. To successfully embed message data into a cover-medium, it is generally required that it is performed against the raw data so as to properly identify and utilize the cover-medium's redundant bits. As such, identification of compressed cover-medium, decompression, modification of the raw data, and then re-compression may be required.

Lossy vs. Lossless Compression

When considering the potential use of compression within the cover-medium, it is also important to consider the type of compression used. Most compression methods can be categorized into two types; lossy compression and lossless compression.

If the compression method used is of the lossy type, the integrity of any message data embedded into the cover-medium prior to compression may be compromised when the stego-medium is uncompressed as some of the original audio data may be lost. Due to this property of lossy compression types, audio data compressed in this manner may not be appropriate for use as cover-medium without additional safeguards against this loss.

2.3.6 Media Gateway Audio Modifications

RTP, as a protocol being potentially routed across multiple networks by its underlying transport, network, and data-link protocols, may also be routed or gatewayed along its path by other intermediary telephony devices like Media Gateways or Back-to-Back User Agent (B2BUA) devices. At such transition points, the media being

transported may undergo potential modification. Some of these modifications include translation from one audio Codec to another, down-sampling, normalization, or mixing with other audio streams. Invasive changes such as these can potentially impact the integrity of any message data embedded within the stego-medium.

Audio Codec Conversion

Codec conversion takes place when an intermediary device such as a Media Gateway is providing translation services for two endpoints that support disparate sets of Codecs. For example, one endpoint may support GSM encoding of audio and the other only G.711 or Speex encoding. Unless an intermediary translator is involved, these two devices cannot directly establish an RTP audio channel. The intermediary device essentially translates audio from the Codec being used by one endpoint to a Codec that can be understood by the other. Audio Codec conversion may also take place if the inherent latency or Quality-of-Service (QoS) properties of the transport network on either side of the intermediary device requires a lighter-weight Codec.

Down-sampling and Normalization

Down-sampling and normalization may be performed on an audio payload to bring the properties of the audio such as volume and background white-noise more in line with the other party's audio stream. Occasionally this task is handled by the endpoint devices when playing the media for the user. In that scenario the integrity of the stego-medium will likely remain intact as the audio payload isn't actually modified in transit. However, there are scenarios where an intermediary media device may actually re-sample or otherwise modify the payload of the media stream specifically to alter its audible properties. In these cases, the integrity of the stego-medium may become compromised.

Audio Stream Mixing

When performing conferencing or other types of multi-party calls, it is possible that multiple party's audio streams may be mixed together. Such invasive modification of the audio will almost certainly compromise the integrity of the stego-medium.

2.3.7 Mid-session Audio Codec Change

Most VoIP signaling protocols provide methods for VoIP endpoints to change the audio encoding method on the fly. Due to this functionality an RTP session may begin using one Codec and then switch to a completely different Codec mid-session. This functionality may be used for a variety of reasons including QoS metrics not

being met, inclusion of a new endpoint in the call that does not support the original Codec, or any number of other reasons. Due to this dynamic nature, any steganographic system attempting to embed data into an RTP stream's packets must be able to dynamically adjust its message embedding algorithm to accommodate different Codecs' various sample sizes and layout within the RTP packet payload.

Chapter 3

Reference Implementation: SteganRTP

3.1 Design Goals

The goals set forth for the SteganRTP reference implementation[?] are described in the following subsections.

3.1.1 Achieve Steganography

As stated in Section 1.4, the primary goal of steganography is to hide the fact that communication is taking place. Therefore, it is the primary goal of this reference implementation to prevent indication to a third-party observer of the RTP audio stream that anything other than the overt communication between the two RTP endpoints is taking place.

3.1.2 Full-Duplex Communications Channel

This reference implementation intends to achieve a full-duplex covert communication channel between the two RTP endpoints, mirroring the utility of RTP itself. This will be accomplished through the use of both RTP streams that comprise an RTP session. By utilizing both RTP streams within the session, either application will be able to both send and receive data simultaneously.

3.1.3 Compensate for Unreliable Transport

This reference implementation intends to compensate for the unreliable transport inherent to RTP. This will be accomplished by providing a data sequencing, tracking, and resending mechanism.

3.1.4 Identical User Experience Regardless of Mode of Operation

This reference implementation intends to provide two distinct modes of operation. The first mode of operation is described as the SteganRTP application running locally on the same host as the RTP endpoint. The second mode of operation is described as the SteganRTP application running on an intermediary host along the route from one RTP endpoint to another. This intermediary host must be forwarding or bridging the RTP traffic as an active man-in-the-middle (MITM). The reference implementation intends for the user experience of running the SteganRTP application to be identical regardless of the mode of operation. This will be accomplished by interfacing directly with the host operating system's network stack in order to hook the desired packet streams.

3.1.5 Multi-type Data Transfer

The reference implementation intends to provide simultaneous transfer of multiple types of data, such as text chat, file transfer, and remote shell access. This will be accomplished by providing type indication and formatting for each type of supported data being transferred.

3.2 Operational Architecture

As mentioned in Section 3.1.4 above, the application will operate in one of two distinct modes: the application running locally on the same host as the RTP endpoint (see Figure 3.1 below) or the application running as an active MITM (see Figure 3.2 below). It is not intended that the two SteganRTP applications which are communicating be operating in the same mode. Thus, a mixed-mode operation such as is described in Figure 3.3 below is entirely possible.

It is important to note that the SteganRTP application is only required to be bridging or forwarding the RTP stream considered *outbound* from the closer RTP endpoint destined for the more remote RTP endpoint. Conversely, the application is only required to be able to observe the *inbound* RTP stream flowing in the other

direction as it does not need to invasively modify any packets from the inbound stream.

3.2.1 Local Operation

In Figure 3.1 below, the gray boxes represent network hosts. The green boxes represent the SteganRTP application. The telephone icons represent the RTP endpoint applications, and the black arrows represent the two distinct RTP streams, one flowing in either direction. In this scenario, both instances of the SteganRTP application are running on the same hosts as the RTP endpoint applications.

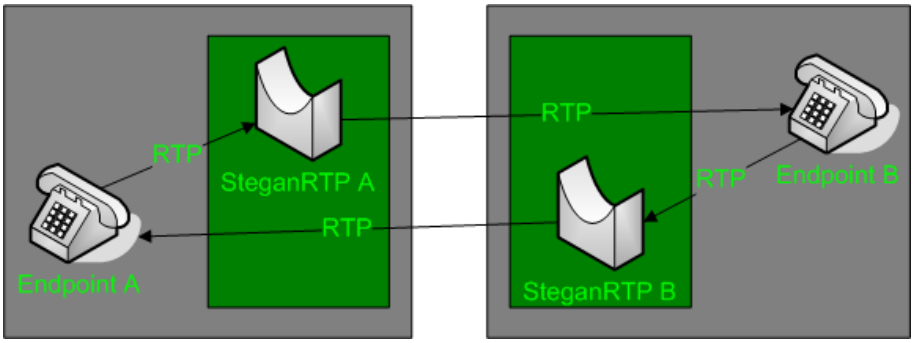

Figure 3.1: SteganRTP running locally.

3.2.2 Man-in-the-Middle Operation

In Figure 3.2 below, the gray boxes represent network hosts. The green boxes represent the SteganRTP application. The telephone icons represent the RTP endpoint applications, and the black arrows represent the two distinct RTP streams, one flowing in either direction. In this scenario, both instances of the SteganRTP application are running on intermediary hosts along the route between the two RTP endpoint hosts.

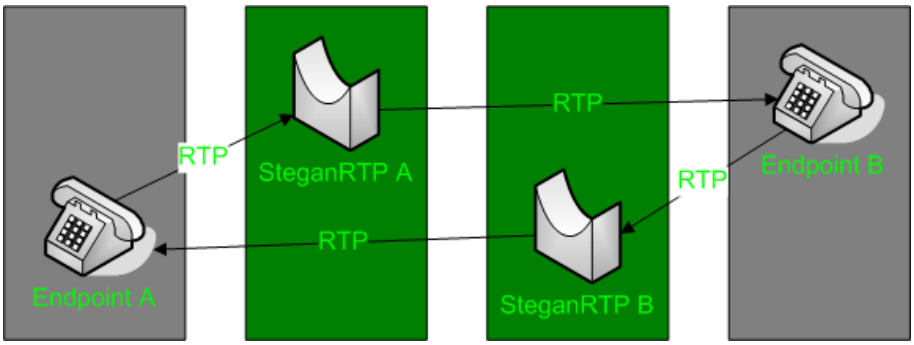

Figure 3.2: SteganRTP running as an active man-in-the-middle.

3.2.3 Mixed Operation

In Figure 3.3 below, the gray boxes represent network hosts. The green boxes represent the SteganRTP application. The telephone icons represent the RTP endpoint applications, and the black arrows represent the two distinct RTP streams, one flowing in either direction. In this scenario, the SteganRTP application on the left of the diagram, SteganRTP A, is running on an intermediary host along the route from its nearer RTP endpoint, Endpoint A, to its remote RTP endpoint, Endpoint B. The SteganRTP application on the right of the diagram, SteganRTP B, is running on the same host as its nearer RTP endpoint, Endpoint B.

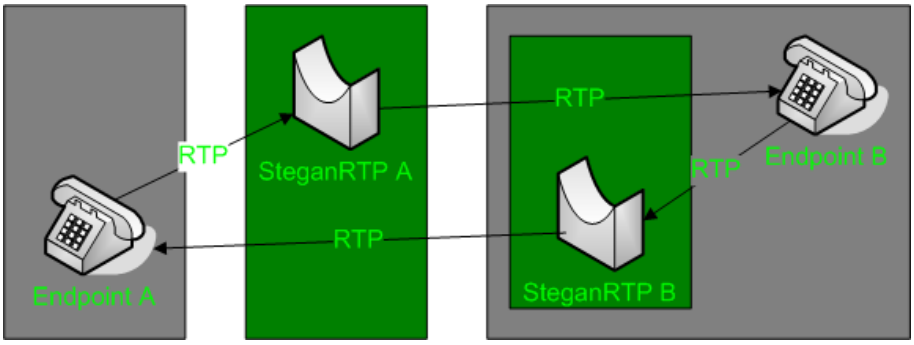

Figure 3.3: SteganRTP running locally on one endpoint host and as an active man-in-the-middle on the other.

3.3 Application Flow

Figure 3.4 below describes the overall SteganRTP application flow.

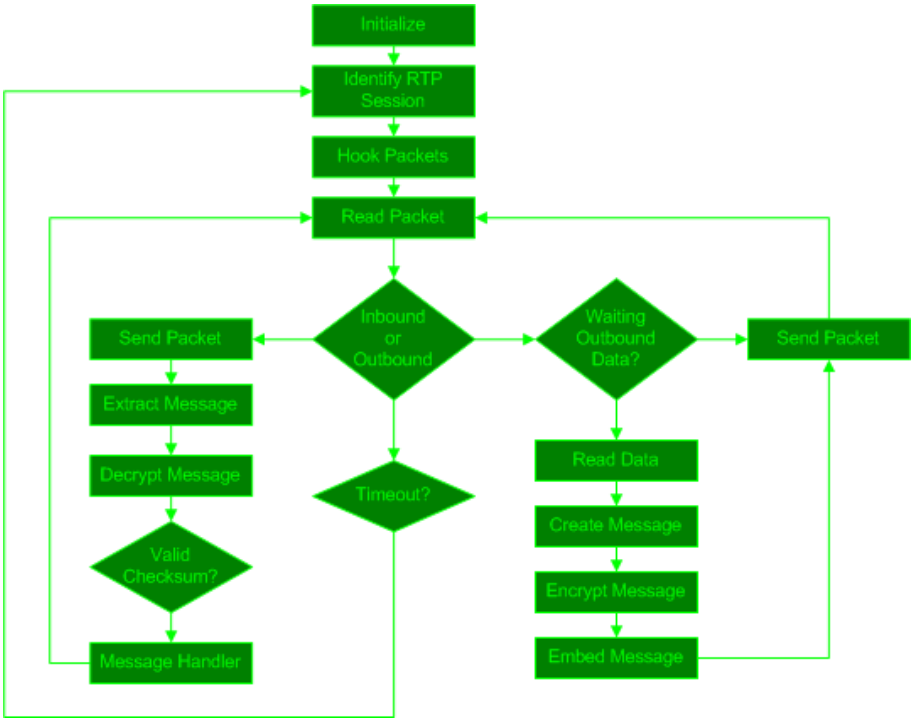

Figure 3.4: SteganRTP application process flowchart.

When the SteganRTP application begins it performs an initialization phase by setting up internal memory structures and configuration information from the command-line. Next, it observes network traffic until it identifies an RTP session which falls within the constraints specified by the user on the command-line. These constraints are how the user controls selection of the RTP sessions between specific RTP endpoints to utilize as cover-medium and, by virtue, which remote SteganRTP application to communicate with. After identifying an RTP session, SteganRTP inserts hooks into the host's network stack in order to receive the desired packets upon transmission or arrival, or both if the SteganRTP application is operating in the active MITM scenario. From these hooks a packet queue is created which the application then reads individual packets from. Whether the packet is considered *inbound* or *outbound* determines the further course of the application. Whether a packet is considered inbound or outbound is determined by which RTP endpoint network address and port is defined as "local" or "remote", which in the case of

the active MITM operation can be inferred as "near" or "far", respectively.

When an inbound RTP packet is read from the queue, it is copied for the application's use and the original packet is immediately sent as the SteganRTP application does not need to invasively modify it. All received inbound packets are assumed to be potential cover-medium for the covert channel, so potential message data is then extracted from each inbound packet. The potential message data is then decrypted, and the result is checked for a valid checksum value in the potential message's header. If the checksum is valid, the message data is sent to the message handler component for processing.

When an outbound RTP packet is read from the queue, the SteganRTP application immediately polls its outbound data queues for any message data waiting to be sent. If there is no data waiting to be sent, the packet is immediately sent unmodified. If there is message data waiting to be sent, as much of that data as will fit into the cover-medium packet's payload is read from its file descriptor, packaged as a formatted message, encrypted, and then steganographically embedded into the RTP packet's payload. The modified RTP packet is then sent in place of the original RTP packet.

3.3.1 Initialization

Upon start-up, SteganRTP first initializes various memory structures such as message caches, configuration settings, and an RTP session context structure.

The most notable task performed during the initialization phase is the computation of keying information used by various components. The method chosen for creation of this keying information is to create a 20-byte SHA-1[?] hash of a user-supplied shared secret text string. Due to the result of this operation being used as keying information by various components of the overall SteganRTP system, this shared secret must be provided to both SteganRTP applications that wish to communicate with each other.

The 20-byte result of the SHA-1 hash function against the user-supplied shared secret is defined here as the *keyhash* and described by Equation 3.1 below where f represents the SHA-1 hash function.

$$keyhash = f(sharedsecret) \tag{3.1}$$

SHA-1 Collision Irrelevance

In February of 2005, a group of Chinese researchers developed an algorithm for finding SHA-1 hash collisions faster than brute force[?]. They proved it possible to find collisions in the full 80-step SHA-1 in less than 2^{69} hash operations, about 2,000 times faster than brute force of the 2^{80} hash operation theoretical bound. The

paper also includes search attacks for finding collisions in the 58-step SHA-1 in 2^{33} hash operations and SHA-0 in 2^{39} hash operations. The biggest impact that this discovery has pertains to use of SHA-1 hashes in digital signatures and technologies where one of the pre-images is known. By searching for a second pre-image which hashes to the same value as the original, a digital signature for the original may theoretically be used to authenticate a forgery.

The use of SHA-1 by the SteganRTP reference implementation is solely to compute a bit-pad of keying information with a longer, seemingly more random bit distribution than what is likely provided directly by user input as the shared secret. The result of the SHA-1 hash of the user's shared secret is used directly as keying information. In order to launch a collision attack against the hash used as the bit-pad, the attacker would have to either obtain the original user-supplied shared secret or the hash itself. Due to the hash being used directly as keying information, the possession of it by an attacker has already compromised the security of the data being obfuscated with it; computing one or more additional pre-images which hash to a collision provides no additional value for the attacker.

3.3.2 RTP Session Identification

RTP session identification is performed using libfindrtp[**?**]. libfindrtp is a C library that identifies sessions between two endpoints by observing VoIP signaling traffic and watching for call set-up. Constraints can be passed to the library to limit session identification to a single endpoint, specific multiple endpoints, or even specific multiple endpoints using specific UDP ports. These constraints are passed through to libfindrtp from the input provided to the SteganRTP application via the command-line. At the time of this writing, libfindrtp supports session identification via the Session Initiation Protocol[**?**] (SIP) and Cisco Skinny Call Control Protocol[**?**] (SCCP) VoIP signaling protocols.

3.3.3 Hooking Packets

The SteganRTP application makes use of NetFilter[**?**] hook points in order to receive both inbound and outbound RTP session packets. The Linux kernel is instructed to pass specific packets to an application by inserting an iptables rule describing the packets with a target of QUEUE. Packets which match a rule with a target of QUEUE are queued to be read by a registered NetFilter user-space queuing agent. Access to this queue is provided to the SteganRTP application via an API provided by the NetFilter C library libipq. An iptables rule used to hook packets via this interface may be inserted at any of the NetFilter hook points as indicated by the circle icons in Figure 3.5 below.

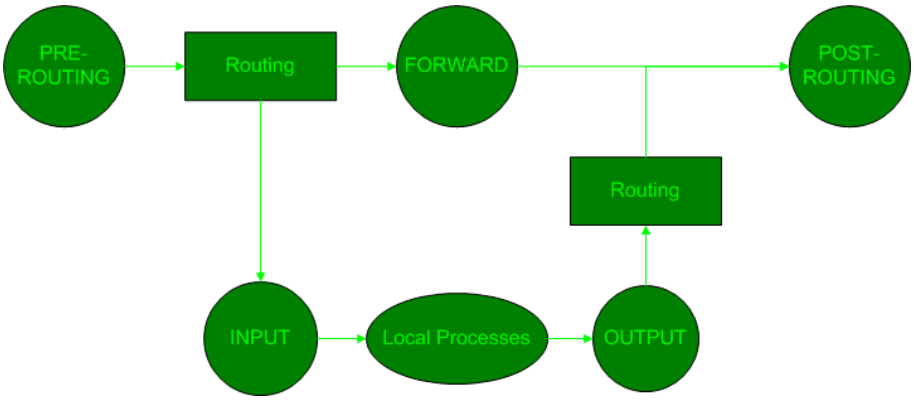

Figure 3.5: NetFilter hook points.

For the most beneficial use by the SteganRTP application, packets must be hooked at points where their integrity as stego-medium is maintained. Thus, inbound packets are hooked at the *PRE-ROUTING* hook point and outbound packets are hooked at the *POST-ROUTING* hook point. In this manner, incoming packets are able to be processed by the SteganRTP application prior to any potential modification by the local system and outbound packets are able to be modified by SteganRTP after the local system is essentially finished with them.

SteganRTP registers itself as a user-space queuing agent for NetFilter via libipq. SteganRTP then creates two iptables rules in the NetFilter engine with targets of QUEUE. The first rule matches the inbound RTP stream at the PRE-ROUTING hook point. The second rule matches the outbound RTP stream at the POST-ROUTING hook point.

3.3.4 Reading Packets

Using the packet hooks described in the previous section, SteganRTP is then able to read packets from the provided packet queue, determine if they are considered *inbound* or *outbound* packets, and pass them to the appropriate processing functions. The processing functions may then analyze them, modify them if needed, place modified versions back into the queue in place of the original, and instruct the queue to accept the packet for further routing.

3.3.5 Inbound Processing

As outlined in Figure 3.4 above, the basic steps for inbound packet processing are as follows:

1. Immediately accept the packet for routing.
2. Extract potential message data.
3. Decrypt potential message data.
4. Verify the potential message header's checksum.
5. Send valid messages to the message handler.

3.3.6 Outbound Processing

As outlined in Figure 3.4 above, the basic steps for outbound packet processing are as follows:

1. Poll for message data waiting to be sent.
2. If there is no message data waiting, immediately send the packet and return.
3. Create a new formatted message with header based on the properties of the RTP packet who's payload is being used as cover-medium.
4. Read as much of the waiting data as will fit in the formatted message.
5. Encrypt the message.
6. Embed the message into the RTP payload cover-medium.
7. Send the modified RTP packet in place of the original via the NetFilter user-space queue.

3.3.7 Session Timeout

In the event that no RTP packets are available in the NetFilter queue for a period of time, all session information is dropped and process flow returns to the RTP session identification phase to locate a new session for use.

In the event that RTP packets are being received but no valid messages have been received for a period of time, the SteganRTP application attempts to solicit a response from the remote application. If these solicitations have failed by the timeout period, all session information is dropped and process flow returns to the RTP session identification phase to locate a new session for use.

3.4 Communication Protocol Specification

The SteganRTP communication protocol makes use of formatted messages which are steganographically embedded into the payloads of individual RTP packets. This steganographic embedding creates the covert channel within which the communication protocol described in the following sections operates.

3.4.1 The cover medium: RTP Packet

Figure 3.6 below, reproduced verbatim from the RTP specification[?], describes the RTP packet header. Of special interest are the *payload type* (PT), *sequence number*, and *timestamp* fields, all of which will become relevant when building, encrypting, and steganographically embedding the message data into the packet's payload. The remainder of the packet contains an optional number of header extensions which are irrelevant to the SteganRTP communication protocol, and finally the encoded media data, otherwise known as the RTP packet's payload, which will be utilized by SteganRTP as cover-medium.

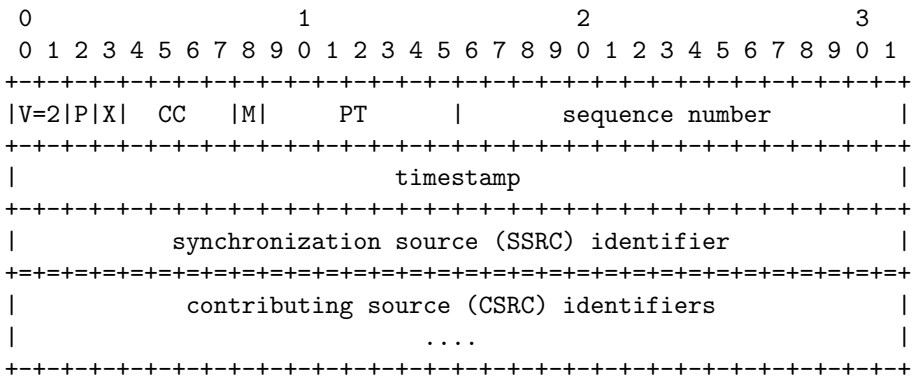

Figure 3.6: RTP packet header, from RFC-1889.

The 7-bit *payload type* field indicates the audio Codec used to encode the payload. The 16-bit *sequence number* field is a standard incrementing sequence number. The 32-bit *timestamp* field describes the sampling instant of the first sample in the payload, and the remaining packet data is the audio payload as encoded by the indicated Codec.

3.4.2 Message Format

The format of the messages that the SteganRTP applications use to communicate with each other is described in the following sections. Figure 3.7 below describes the core message format of all types of SteganRTP formatted messages. This format consists of two fields, the *Checksum / ID* and *Sequence* fields followed by a standard Type-Length-Value[?] (TLV) structure. The *Checksum / ID*, *Sequence*, *Type*, and *Length* fields comprise the message header, while the *Value* field is considered the message body, or payload.

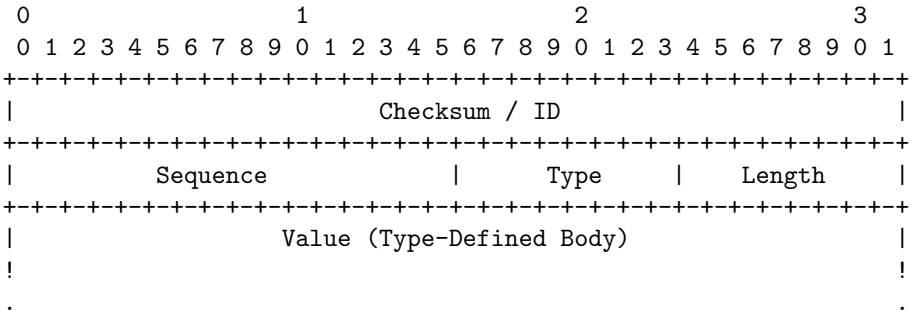

Figure 3.7: SteganRTP message format.

The 32-bit *Checksum / ID* field contains a hash value which is used to identify whether or not a potential message that is extracted from the payload of an inbound RTP packet is indeed a valid SteganRTP communication protocol message. The hashword[?] function is used to compute this hash. The function's two primary operands consist of the keying information defined as *keyhash* in Section 3.3.1 and the sum of the message's *Sequence*, *Type*, and *Length* header fields. This value is defined as *checksumid* and is described by Equation 3.2 below.

$$checksumid = hashword(keyhash, (Sequence + Type + Length)) \qquad (3.2)$$

The verification of extracted potential messages is required due to the fact that some packets in the inbound RTP stream may not contain SteganRTP messages if there was no outbound data waiting to be sent by the remote application when the RTP packet in question traversed it. The hash function used to compute this checksum value incorporates the keyhash so as not to be computable solely from message data, which would allow an observer to also verify that a message is embedded within the RTP payload.

The 16-bit *Sequence* field is a standard incrementing sequence number, the 8-bit *Type* field indicates what type of message it is, and the 8-bit *Length* field indicates the length, in bytes, of the *Value* field. The *Value* field contains the message's payload.

3.4.3 Message Types

The currently defined message types are listed in Table 3.1 below.

Table 3.1: SteganRTP Message Types

ID	Type
0	Reserved
1	Control
10	Chat Data
11	File Data
12	Shell Input Data
13	Shell Output Data

Control Messages

Figure 3.8 below describes the format of SteganRTP control messages. Control messages are used to send non-user data to the remote SteganRTP application to convey operational information such as requesting a message resend or indicating that a file is about to be sent and providing that file's context information. Control messages consist of one or more stacked TLV structures and are not required to be 32-bit aligned.

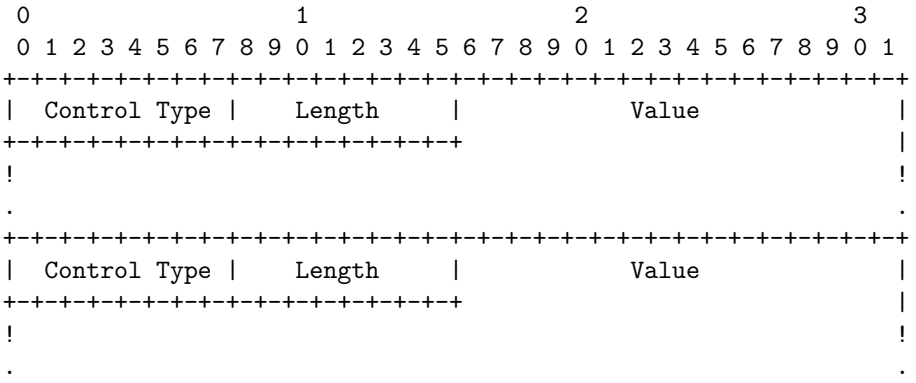

Figure 3.8: SteganRTP control message format.

The 8-bit *Control Type* field indicates the type of control data contained in the TLV structure whereas the 8-bit *Length* field indicates the size, in bytes, of the *Value* field. The *Value* field contains the control data of the indicated type.

Control Message Types

The currently defined control message types are listed in Table 3.2 below.

Table 3.2: SteganRTP Control Message Types

ID	Type
0	Reserved
1	Echo Request
2	Echo Reply
3	Resend
4	Start File
5	End File

Type 1: Echo Request

The Echo Request control message is used to prompt the remote SteganRTP application for a response, allowing the local application making the request to determine if the remote application is still present and communicating. This message is sent when a session inactivity timeout limit is approaching. Figure 3.9 below describes the format of an Echo Request control message.

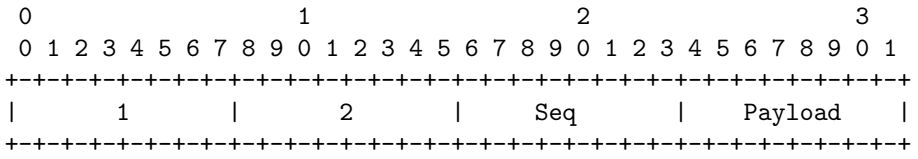

Figure 3.9: SteganRTP Echo Request control message.

The *Control Type* field's value is 1, indicating that it is an Echo Request control message, and the *Length* field's value is 2, indicating the 2-byte control message payload. The control message payload consists of an 8-bit *Seq* field which contains a standard incrementing sequence number specific to Echo Requests, and an 8-bit Echo Request *Payload*, which contains a random bit-string. The *Seq* value is used to correlate sent Echo Request messages with received Echo Reply messages and the *Payload* field received in an Echo Reply message must match the random bit-string sent in its corresponding Echo Request message.

Type 2: Echo Reply

The Echo Reply control message is used to respond to the remote SteganRTP application's Echo Request message. The format of the Echo Reply message is identical to the Echo Request message as described in 3.9 above, however the *Control Type* field's value is 2 rather than 1.

Type 3: Resend

The Resend control message is used to request the resending of a specified message by the remote SteganRTP application, allowing the local application to request missing or corrupted messages. This message is sent when the application begins to receive messages which contain sequence numbers beyond the next sequence number that is expected. Figure 3.10 below describes the format of a Resend control message.

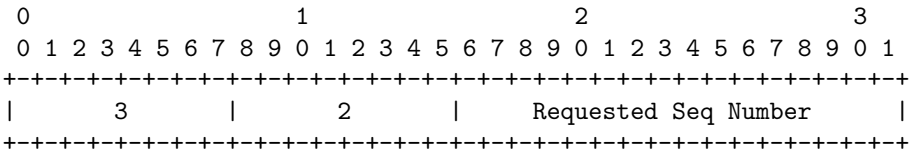

Figure 3.10: SteganRTP Resend control message.

The *Control Type* field's value is 3, indicating that it is a Resend control message, and the *Length* field's value is 2, indicating the 2-byte control message payload. The control message payload consists of a 16-bit *Requested Seq Number* field which indicates the sequence number of the message to be resent.

Type 4: Start File

The Start File control message is used to indicate to the remote application that that local application will begin sending file data for a new file transfer. This message is sent when the user executes the command to transfer a file. Figure 3.11 below describes the format of a Start File control message.

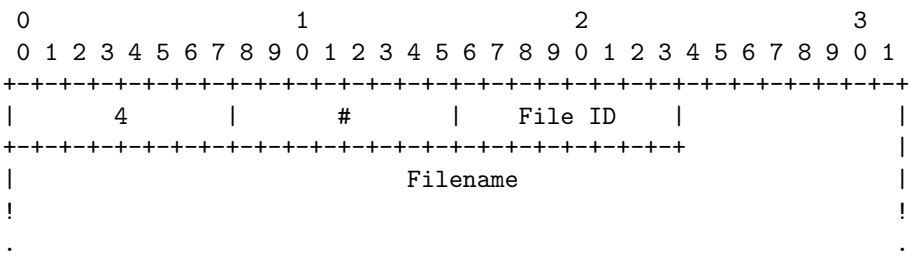

Figure 3.11: SteganRTP Start File control message.

The *Control Type* field's value is 4, indicating that it is a Start File control message, and the *Length* field's value is 1 plus the string length, in bytes, of the filename of the file being sent, indicating the total size of the control message payload. The control message payload consists of an 8-bit *File ID* field which indicates the sending application's unique ID value for the file, and the *Filename* field is the name of the file being sent in ASCII.

Type 5: End File

The End File control message is used to indicate to the remote application that that local application is finished sending file data for a particular file transfer. This message is sent when the local application has finished sending all data related to the open file descriptor being used to send data from a file. Figure 3.12 below describes the format of a End File control message.

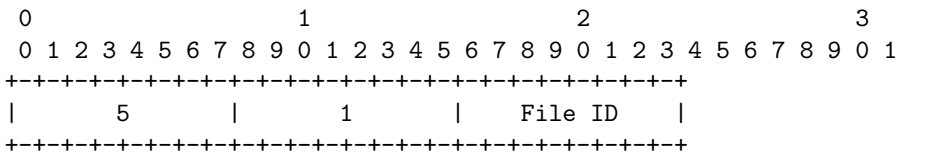

Figure 3.12: SteganRTP End File control message.

The *Control Type* field's value is 5, indicating that it is a End File control message, and the *Length* field's value is 1, indicating the 1-byte control message payload. The control message payload consists of an 8-bit *File ID* field which indicates the sending application's unique ID value for the file who's transfer is now complete.

Data Messages

Non-control messages are considered data messages and contain some form of actual data for the user, whether it be text chat data, incoming file data, a command for the local shell service, or a response from the remote shell service. These various types of data are differentiated by the value of the message header's *Type* field.

Chat Data Messages

The Chat Data Message is used to transmit text chat data between SteganRTP applications. This type of data requires no context information, thus the message payload contains only a single field, *Chat Data*, as described by Figure 3.13 below.

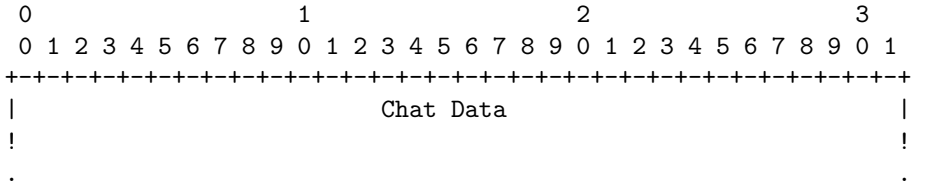

Figure 3.13: SteganRTP Chat Data Message.

File Data Messages

The File Data Message is used to transmit data file contents between SteganRTP applications. Because multiple file transfers may be in progress at any given time, this type of data must be accompanied with context information indicating which file transfer the chunk of data belongs to. Figure 3.14 below describes the format of a File Data message.

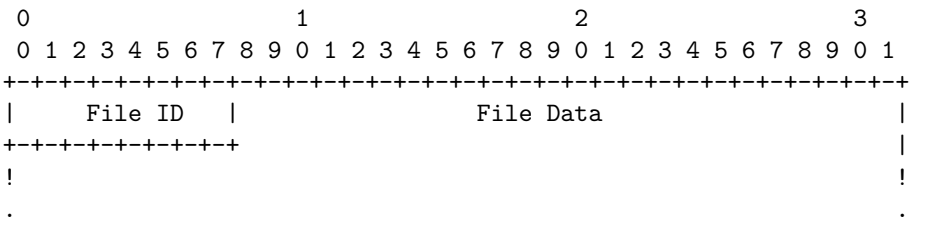

Figure 3.14: SteganRTP File Data Message.

The *File ID* field's value is a unique file ID number chosen for the particular file transfer taking place and is used to indicate which file transfer the chunk of data contained in the *File Data* field belongs to. The *File Data* field is a chunk of data from the file being transferred. The proper order for reconstruction of the file chunks transferred by these messages is ensured by the message header's sequence number.

Shell Data Messages

The Shell Input Data and Shell Output Data Messages are used to transmit shell input to, and receive shell output from, a remote SteganRTP shell service, respectively. This type of data requires no context information, thus the message payload contains only a single field, *Shell Data*, as described by Figure 3.15 below.

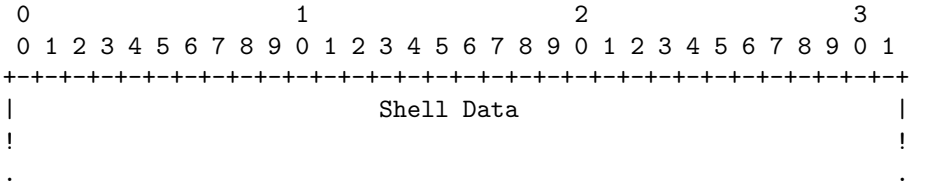

Figure 3.15: SteganRTP Shell Data Message.

3.5 Functional Components

3.5.1 File Descriptor Lists

Two separate file descriptor lists are maintained; destinations for inbound data and sources of outbound data. The data structure for storage of a file descriptor and its data for inclusion in either list is defined in Figure 3.16 below.

```
/* Structure used for file descriptor information */
typedef struct file_info_t {
   u_int8_t id;
   char *name;
   u_int8_t type;
   int fd;
   struct file_info_t *next;
   struct file_info_t *prev;
} file_info;
```

Figure 3.16: C structure for File Descriptor List elements.

The independence of the file descriptor lists from the outbound data polling and message handler components provides for a flexible and versatile environment within which to expand functionality. In order to include new data types for transfer, all that is required is to define a new data type ID for both applications to correlate messages upon, open a file descriptor to the appropriate place to read or write the data, and include the file descriptor in the appropriate list.

Inbound File Descriptors

Inbound File Descriptors are a list of file descriptors for various destinations that inbound data may be directed to. The order of these file descriptors as included in the list is irrelevant as which file descriptor data is destined for is looked up by matching the message type and properties with the file descriptor's type and properties.

Chat Interface

Inbound chat data is written to this file descriptor. This file descriptor is tied to the chat window of the SteganRTP ncurses interface.

Remote Shell Interface

Inbound shell data from the remote application's shell service is written to this file descriptor. This file descriptor is tied to the shell window of the SteganRTP ncurses

interface.

Local Shell Service

Inbound shell data to the local application's shell service is written to this file descriptor. This file descriptor is tied to the local process providing shell access. This file descriptor does not exist in the list if the local shell service is disabled.

File Transfers

Any number of file descriptors for data files being actively received may be appended or removed from the end of the inbound file descriptors list.

Outbound File Descriptors

Outbound File Descriptors are polled, in order, for data waiting to be sent. Due to being polled in order, they are essentially prioritized in that order and data waiting to be sent from a prior descriptor in the list will have precedence over data waiting to be sent from a latter descriptor. The file descriptors included in the outbound list are as follows:

Raw Message Interface

Entire, unencrypted outbound messages are written to this file descriptor. This file descriptor is used for the replaying of entire messages in response to a Resend control message as described in Section 3.4.3.

Control Message Interface

Outbound control messages as described in Section 3.4.3 are written to this file descriptor after creation.

Chat Interface

Outbound chat data is written to this file descriptor. This file descriptor is tied to the command window of the SteganRTP ncurses interface. All non-command text entered into the command window while in chat mode is considered chat data.

Remote Shell Interface

Outbound shell data is written to this file descriptor. This file descriptor is tied to the command window of the SteganRTP ncurses interface. All non-command text entered into the command window while in shell mode is considered shell data.

Local Shell Service

Outbound shell data from the local shell service is written to this file descriptor. This file descriptor is tied to the local process providing shell access. This file descriptor does not exist in the list if the local shell service is disabled.

File Transfers

Any number of file descriptors for data files being actively sent may be appended or removed from the end of the outbound file descriptors list.

3.5.2 Message Handler

The SteganRTP application's message handler receives all valid incoming messages as verified by the RTP packet receiving system for inbound packets. This component performs all internal state changes and administrative tasks in response to control messages. It also handles the routing of inbound data message payloads to the appropriate file descriptor in the inbound file descriptors list.

Administrative Tasks

Echo Reply

If an Echo Request control message is received from the remote application, the message handler constructs an appropriate Echo Reply control message as described in Section 3.4.3 and writes it to the Control Message Interface file descriptor in the outbound file descriptor list.

Start File Transfer

If a Start File control message is received from the remote application, the message handler opens a new file descriptor using the file's context information contained in the control message and appends the file descriptor to the inbound file descriptors list.

End File Transfer

If an End File control message is received from the remote application, the message handler closes the file descriptor for the file transfer indicated and removes it from the inbound file descriptors list.

Data Routing

Chat Data

Inbound text chat data is buffered until a complete line of text is received and then is written to the Chat Interface file descriptor in the inbound file descriptors list. A complete line of text is defined as being terminated by a new-line character.

File Data

Inbound file transfer data is written to the appropriate file descriptor in the inbound file descriptors list for the file transfer that the data belongs to.

Shell Input Data

Shell Input Data messages contain input data for the local application's shell service and is written to the Local Shell Service file descriptor in the inbound file descriptors list.

Shell Output Data

Shell Output Data messages contain response data from the remote application's shell service and is written to the Remote Shell Interface file descriptor in the inbound file descriptors list.

3.5.3 Encryption System

The encryption method chosen for use in the SteganRTP reference implementation is not really encryption at all. In favor of light-weight and speed, a simple bit-wise exclusive-or[?] (XOR) obfuscation method was chosen as a symmetric cipher. The choice of encryption method here does not indicate that another, more robust type of encryption could not be used; rather, the modular design of the reference implementation promotes drop-in replacement of the current encryption system entirely, assuming that the replacement encryption method does not have a noticeable impact upon the latency of the overt RTP stream being used as cover-medium.

The author does not claim that the obfuscation method used by the SteganRTP reference implementation to be cryptographically secure. Rather, it is well documented in the literature that XOR against a repeating keystream is insecure[?, ?, ?]. The obfuscation of message data is merely meant to provide some rudimentary protections against statistical steganalysis which focuses upon perceptible properties of language within the stego-medium.

The XOR obfuscation method employed by the SteganRTP reference implementation consists of the following steps:

1. Create a bit-pad for use as keying information.
2. Choose an offset into the bit pad to begin using the keying information.
3. XOR the message against the bit pad, byte by byte.

Bit-pad Creation

The method chosen for creation of the bit-pad is simply to duplicate the bit-string found in *keyhash*, the creation of which is described in detail in Section 3.3.1.

Choose a Bit-pad Offset

To help protect against some forms of statistical analysis that have proved effective against XOR obfuscation using repeated static keying information, it was decided against beginning every XOR loop at the same position within keyhash. To avoid this, a new offset into keyhash for each message must be chosen. The method that the SteganRTP reference implementation employs to determine this offset is to use the hashword[?] function to create a 32-bit hash of *keyhash* and the sum of the RTP packet being embedded into's *Seq* and *Timestamp* header fields. The resultant hash is then interpreted as a 32-bit integer. The integer modulus 20 is the chosen offset into keyhash.

The integer which is the result of the offset choosing operation and is within the range of 0 through 19 is defined here as *keyhash_offset* and described by Equation 3.3 below.

$$keyhash\_offset = hashword(keyhash, (RTP\_Seq + RTP\_TS)) \bmod 20 \quad (3.3)$$

The keyhash_offset equation incorporates keyhash so as to not be entirely computable from observable information in the RTP packet header.

XOR Loop

When used as a bit-pad for the XOR operation loop, keyhash is used 8-bits, or 1-byte, at a time. The XOR loop begins with the first byte of the message to be obfuscated and the byte located at index *keyhash_offset* within *keyhash*. The two bytes are XORed to produce a result byte. This result byte is placed into the obfuscated message buffer at the same byte index as the original message byte. If the end of the bit-pad is reached, the position of the next byte in the bit-pad returns to the beginning of the bit-pad. When the end of the original message is reached, the obfuscated message buffer should be of equal length to the original message and have one corresponding obfuscated byte for each original byte in the message.

It is important to note that within the scope of steganography terminology, whether or not message data is obfuscated or encrypted is irrelevant. As such, further reference to the obfuscated message will still be referred to as the message, or message data.

3.5.4 Embedding System

The embedding system that was developed for the SteganRTP reference implementation is a generalized least-significant-bit (LSB) steganographic data embedding method. It is generalized such that when provided with a cover-medium buffer, its length, the size of each word value within the cover-medium buffer, and the message

buffer to be embedded, it is then able to perform the LSB embedding operation. In this way, any audio Codec which uses a linear grouping of fixed-length audio samples should be able to be utilized as cover-medium by the embedding system.

For the purpose of discussion of the SteganRTP embedding system, the term *word value* used in this context is equivalent to *audio sample*. The example used here, as well as the only Codec currently supported by the reference implementation, is G.711. G.711 is a Codec which encodes audio as a linear grouping of 8-bit audio samples. This encoded data is transported by RTP packets as their payload and will serve as cover-medium.

Using the generalized LSB embedding method, the LSB of each word value in the cover-medium is modified to be equivalent to a single bit from the message data buffer, in order. The properties of the RTP packet, such as its payload length and *payload type* header value, determine how much message data can be embedded into the packet's payload. The RTP packet's payload size is determined by subtracting the size of the RTP packet's header from the value of the UDP packet header's *Length* field[**?**]. The wordsize is equivalent to the sample size used by the RTP packet's Codec, indicated by the RTP packet header's *payload type* field. Modifying 1 bit from each word value requires 8 word values to embed a single byte of message data. Thus, the amount of available space within an RTP packet's payload for embedding is found by multiplying the word value size by 8, then dividing the RTP packet payload size by the result.

The resultant value is defined here as the RTP packet's *available_space* for embedding and is described by Equation 3.4 below.

$$available\_space = RTP\_payload\_size/(wordsize \cdot 8) \tag{3.4}$$

The space available for user data after prepending the SteganRTP communication protocol's message header is defined here as the SteganRTP message's *payload_size* and is described by Equation 3.5 below.

$$payload\_size = available\_space - sizeof(message\_header) \tag{3.5}$$

Thus, *payload_size* bytes of user data can be packaged as a SteganRTP message and embedded into an RTP packet payload cover-medium of *available_space* bytes. If an RTP packet is too small to contain a valid message, it is passed along unmodified.

If a message being embedded is smaller than the available space in the cover-medium, the message is padded out to the available size with random data. This ensures a more uniform distribution of modified values throughout the cover-medium.

3.5.5 Extraction System

All inbound RTP packets are sent to the extraction system where potential message data is extracted, decrypted, and then verified. The extraction system is essentially

a reverse of the embedding system described in Section 3.5.4 and then a pass through the symmetric encryption system described in Section 3.5.3. This results in an decrypted potential message where the message's *Checksum / ID* header field value can be verified to determine whether or not the extracted potential message is valid.

If an extracted potential message is found to be valid, it is passed to the message handler component.

3.5.6 Outbound Data Polling System

File descriptors in the outbound file descriptors list are polled, in order, for data waiting to be sent. When a file descriptor is found to have data, a new formatted message is created if needed and data is read to fill the payload of that message from the file descriptor. The message type is indicated by the file descriptor's record in the outbound file descriptors list. The result of this operation is a formatted SteganRTP message ready for encryption and embedding into the cover-medium.

3.5.7 Message Caching System

All inbound and outbound SteganRTP messages are cached. The outbound message cache provides a mechanism for retrieval of any given message in the event that the remote application issues a Resend control message requesting that the message be resent. The inbound message cache provides a mechanism for storage of messages received that are beyond the expected sequence number. Once the expected message is received, the others may be read back from the cache rather than requesting that the remote application resend them.

3.5.8 Shell Service

The local application's shell service is essentially a child process executing a shell. This process's standard input and output file descriptors are replaced with file descriptors which are stored in the inbound and outbound file descriptors lists, respectively. The local shell service is disabled by default in the SteganRTP reference implementation and must be enabled via the command-line.

3.6 Use

3.6.1 Command-line

The SteganRTP application provides a number of command-line arguments allowing for control and configuration of various components. The following sections describe each in detail.

Usage Output Overview

The following usage output was copied verbatim from the most recent version of the reference implementation, SteganRTP 0.3b.

```
Usage: steganrtp [general options] -t <host> -k <keyphrase>
  required options:
    at least one of:
      -a <host>       The "source" of the RTP session, or, host
treated as the "close" endpoint (host A)
      -b <host>       The "destination" of the RTP session, or,
host treated as the "remote" endpoint (host B)
    -k <keyphrase>  Shared secret used as a key to obfuscate
communications
  general options:
    -c <port>       Host A's RTP port
    -d <port>       Host B's RTP port
    -i <interface>  Interface device (defaults to eth0)
    -s              Enable the shell service (DANGEROUS)
    -v              Increase verbosity (repeat for additional
verbosity)
  help and documentation:
    -V              Print version information and exit
    -e              Show usage examples and exit
    -h              Print help message and exit
```

Command-line Arguments

The following command-line arguments are available from the SteganRTP application's command-line.

-a <host>

<host> is the name or IP address of the closest side of the RTP session desired to be utilized as cover-medium (Host A).

-b <host>

<host> is the name or IP address of the remote size of the RTP session desired to be utilized as cover-medium (Host B).

-k <keyphrase>

<keyphrase> is a shared secret between the users of the two SteganRTP instances which will be communicating. In some cases, a single user may be running both instances. The keyphrase is used to generate a bit-pad via the SHA-1 hash function which will later be used to obfuscate the data being steganographically embedded into the RTP audio cover-data.

-c <port>

<port> is the RTP port used by Host A.

-d <port>

<port> is the RTP port used by Host B.

-i <interface>

<interface> is the interface to use on the local host. This parameter defaults to "eth0".

-s

This argument enables the command shell service. If the command shell service is enabled, the user of the remote instance of SteganRTP will be able to execute commands on the local system as the user running SteganRTP. You likely don't want this unless you are the user running both instances of SteganRTP and intend to use the remote instance as an interface for a remote shell on that host. This feature can be useful for remote administration of a system without direct access to the system, assuming that RTP is allowed to traverse traffic policy enforcement points.

-v

This argument increases the verbosity level. Repeat for higher levels of verbosity.

-V

This argument prints SteganRTP's version information and exits.

-e

This argument prints a quick examples reference.

-h

This argument prints the usage (help) information and exits.

Usage Examples

You can print a quick reference of the following examples from the SteganRTP command-line by using the -e command-line argument.

The simplest command-line you can execute to successfully run SteganRTP is:

```
steganrtp -k <keyphrase> -b <host>
```

This will begin a session utilizing any RTP session involving <host-b> as the destination endpoint.

```
steganrtp -k <keyphrase> -a <host-a> -b <host-b> -i <interface>
```

This will begin a session utilizing any RTP session between <host-a> and <host-b> using interface <interface>

```
steganrtp -k <keyphrase> -a <host-a> -b <host-b> -i <interface> -s
```

This is the same as the previous example but will enable the command shell service:

```
steganrtp -k <keyphrase> -a <host-a> -b <host-b> -c <a-port> -d
<b-port>
```

This will begin a session utilizing a specific RTP session between <host-a> on port <a-port> and <host-b> on <b-port>. Note, this will effectively disable RTP session auto-identification and will attempt to use an RTP session as described whether it exists or not. This is useful for when an RTP session that is desirable for utilization is already in progress as the other examples rely on libfindrtp to identify the RTP session as it is being set up by VoIP signaling and thus must be waiting for the call-setup.

3.6.2 User Interface

SteganRTP provides a curses user interface featuring four windows; the Command window at the bottom of the screen, the large Main window in the middle of the screen, and the Input and Output Status windows at the top of the screen.

Windows

Command Window

All keyboard input, if accepted, is displayed in the Command window. Lines of input that are not prefixed with a slash ('/') character are treated as chat text and are sent to the remote instance of SteganRTP as such. Lines of input that begin with a slash are considered commands and are processed by the local instance of SteganRTP.

Main Window

When in Chat mode, chat text and general SteganRTP information messages and events are displayed in the Main window. When in shell mode, this window is overloaded with the input to and output of the shell service provided by the remote instance of SteganRTP.

Input Status Window

Events related to incoming RTP packets or SteganRTP communication messages are displayed in the Input Status window.

Output Status Window

Events related to output RTP packets or SteganRTP communication messages are displayed in the Output Status window.

Commands

The following commands can be executed from within the Command window:

/chat

The "chat" command puts the interface into Chat Mode.

/sendfile <filename>

The "sendfile" command queues a file for transmission to the remote instance of SteganRTP. <filename> is the path location and filename of the local file to be sent.

/shell

The "shell" command puts the interface into Shell Mode.

/quit /exit

The "quit" and "exit" commands exit the program.

/help /?

The "help" and "?" commands print an available command list.

Chapter 4

Solutions to Problems and Challenges

The following sections describe this research effort's approach to solving many of the problems and challenges that were identified in Section 2.3, as implemented via the SteganRTP reference implementation. Most of the solutions that have been devised during this research effort involved the creation of a communications protocol to operate within the covert channel established within the cover-medium. This protocol, detailed in Section **??** employs a formatted message header which is prepended to user message data before being embedded in the cover-medium, providing various utility to the application making use of the protocol.

4.1 Unreliable Transport

To mitigate the unreliable properties of the underlying transport protocols used to transmit the cover-medium, the message header contains a sequence number. This sequence number coupled with the message caching system allows the recipient to both identify when an expected message is missing as well as request a resend of a particular message via a control message. This property also provides the added benefit of detecting erroneously or maliciously replayed messages.

When considering potential solutions for this problem, various types of Forward Error Correction[**?**] (FEC) were considered. Due to the limited space available for message data as a result of the size of cover-medium available, the additional space required for redundant data by most algorithms considered deemed them to be unfit for purpose within this research effort's context.

4.2 Cover-Medium Size Limitations

The same property of RTP which restricts the size of available cover-medium in each packet is luckily the same property which ensures that there are an abundance of packets being sent between RTP endpoints every second. User data can be spread over multiple messages and cover-packets and then reassembled at their destination. For this research effort's purposes and goals, namely the timely transfer of user text chat, interactive shell access, and transfer of small files, an achieved throughput of 1,000 bytes per second as described in Section 2.2.3 was found to be more than adequate.

4.3 Latency

To prevent against unintended impact on RTP packet latency, care was taken to efficiently perform a number of operations:

4.3.1 Inbound Packet Processing

When receiving inbound RTP packets for processing, the receiving system does not require making any modifications to the received packet. In the SteganRTP reference implementation, the packet is received and immediately accepted for continued routing by the packet queue prior to extracting, decrypting, and verifying any potential message data found within the payload.

4.3.2 Outbound Packet Processing

When receiving outbound RTP packets for processing, the fewest number of operations possible must be performed in order to make a decision on whether or not the packet should be immediately accepted for continued routing or if it must be held for modification. In the SteganRTP reference implementation, the packet is received and then all active outbound file descriptors are polled for data waiting to be sent. If no data is waiting to be sent, the packet is then accepted for continued routing by the packet queue.

4.3.3 Encryption Overhead

When encrypting the raw message prior to embedding into the cover-medium, a low-overhead algorithm was used. The SteganRTP reference implementation employs an XOR against a SHA-1 hash of a user-supplied shared-secret.

4.4 Tracking of RTP Streams

Identification and tracking of RTP streams is handled by the libfindrtp C library paired with the NetFilter libipq C library for tracking and hooking packets. Both libraries were evaluated during this research effort's initial requirements phase and were deemed fit for purpose.

4.5 Media Gateway Audio Modifications

4.5.1 Audio Codec Conversion

Due to the nature of VoIP, it is not always possible to detect whether or not an audio session such as RTP is terminating at the actual recipient of the call audio or at an intermediary. As such, it is not possible to reliably transmit stego-medium from end to end unless the actual network addresses of each endpoint are known. Due to this limitation, the SteganRTP reference implementation assumes that there are no intermediary devices along the media path making changes to the RTP payload. The reference implementation makes this assumption by also assuming that the sending and receiving applications are either running on the same hosts as the RTP endpoint applications or are along the network path between the two visible RTP endpoints which may or may not be intermediaries. The reference implementation requires that these endpoint network addresses are specified by the user or identified by the RTP session identification component.

4.6 Mid-session Audio Codec Change

The SteganRTP reference implementation's embedding component addresses the issue of mid-session audio Codec change by determining the audio sample word size dynamically based on the Codec value supplied by the RTP packet's header. Thus, the embedding system's parameters are derived from each individual RTP packet that will be embedded into as cover-medium. If the RTP session were to change Codecs mid-session, or even to change Codecs for every other packet, the embedding system will only operate on RTP packets who's payloads are encoded with a Codec that the embedding system recognizes and has parameters defined for. If the embedding system does not recognize and support a particular packet's Codec, that packet is passed unmodified.

Chapter 5

Conclusion

5.1 Design Goals

It is the author's belief that all of the design goals set forth in Section 3.1 for the SteganRTP reference implementation were met. The primary goal of steganography, establishment of a full-duplex communications channel, compensation for the unreliable transport mechanism, identical user experience regardless of mode of operation, and multi-type data transfer were all accomplished.

5.2 Identified Challenges

It is the author's belief that all but two of the identified problems and challenges identified in Section 2.3 were fully addressed. The two challenges that were not addressed were the various types of media gateway audio modifications outlined in Section 2.3.6 due to scope and the issue of compressed audio outlined in Section 2.3.5 due to time limitations of the research effort.

5.3 Secure Real-time Transfer Protocol

It is important to note that use of the Secure Real-time Transfer Protocol (SRTP) RTP profile may prevent specific operational scenarios such as the active MITM scenario described in Section 3.2.2. Encrypting various parts of the RTP header and RTP payload will prevent invasive modification of the payload by an external entity to the RTP session. SRTP, however, won't protect against steganographic embedding of message data prior to the application of the SRTP encryption methods,

such as may be performed within the RTP endpoint application itself.

5.4 Future Research

It is the author's intention to continue this research effort at a later time. The identified areas for continued research include:

1. Replacement of the generalized LSB embedding system with Codec specific embedding algorithms. Utilizing Codec-specific properties, more intelligent embedding methods such as the inclusion of silence and voice detection can be performed as well as a wider variety of Codecs can be supported.
2. Creation of embedding algorithms for video Codecs.
3. Replacement of the XOR obfuscation system with real encryption.
4. Addition of support for fragmentation of larger formatted messages across multiple RTP packet payload cover-mediums.
5. Expansion of the shell service functionality into a more generalized services framework.