Vulnerabilities of the Real-Time Transport (RTP) Protocol for Voice over IP (VoIP) Traffic

Mike Adams
Acme Packet
urlington, MA 01803
Email: madams@acmepacket.com

Minseok Kwon
Department of Computer Science
Rochester Institute of Technology
Email: jmk@cs.rit.edu

***Abstract*—Over the past decade, Voice over IP (VoIP) has revolutionalized the telecommunications industry. VoIP has become more prevalent than ever, and consequently more users have switched to IP-based data networks for their telephone use from the analogue Public Switched Telephone Network (PSTN). One challenge, though, is to secure and protect these VoIP connections. In this paper, we investigate a new approach to demonstrate the vulnerability of VoIP connections. Our approach monitors real-time data streams (e.g., RTP), and inserts packets containing fraudulent voice data at expected times estimated from the monitoring. As false packets are well-aligned with original packets, we can maximize the effects of the test while minimizing the number of inserted packets. This minimal number of false packets also helps eschew the detection efforts of denial-of-service defense mechanisms. Our results indicate that the inserted packets at desired times can indeed disrupt the original RTP stream without any noticeable traffic increase.**

***Index Terms*—Voice over IP, RTP, SIP, multimedia networks, security, vulnerability, experiments**

I. Introduction

In recent years, Voice over IP (VoIP) technologies have made significant progress both in research and commercially [1]–[4]. VoIP allows users to make phone calls over IP-based data networks instead of the Public Switched Telephone Network (PSTN) through such technologies as SIP [1], RTP [2], and H.323 [5]. As the technology advances, VoIP can provide a higher quality and yet more affordable phone service than PSTN. The telecommunication industry is without a doubt moving towards using VoIP as their main phone infrastructure.

Despite the advances in VoIP technologies, a vast majority of the VoIP services provided by telecommunication carriers are not secure and vulnerable to a variety of malicious activities. If an attacker sniffs on the network between two end-users in communication, the attacker can see virtually every piece of data sent over the wire. The attacker can even insert its own fraudulent packets into the wire prohibiting the victim from using network resources (like denial-of-service attacks).

Our goal in this paper is to demonstrate that a denial-of-service attack can be crafted specifically targeting the VoIP network, yet is hard to detect and defeat. While the attack can be applied to other real-time data streams, we only focus on the VoIP use of the G.711 (U-law and A-law) codec that employs the RTP protocol to transmit digitized voice data [2], [6]. The attack can also be considered as the *vulnerability test* of RTP. The main idea is to inject false RTP packets at specific times so that the receiver accepts these false RTP packets while dropping legitimate ones. This type of attack is possible because we can anticipate the exact moment when a specific RTP packet is delivered. Moreover, ISPs do not usually inspect RTP payloads due to their limited resources, and can see no clear evidence of any malicious behavior in this attack. We also adopt a similar method to the shrew attack for TCP congestion control [7] to make this attack hard to be detected or defeated. Note that the focus is on demonstrating that such an attack is feasible, not on studying the reaction of VoIP clients to call quality under attack.

Our system comprises two subcomponents: 1) a packet capture engine and 2) a packet injection engine. The packet capture engine monitors the network for any RTP traffic and enables the packet injection engine when a VoIP call is detected. The packet injection engine then synchronizes with the underlying RTP stream and inserts fraudulent RTP packets shortly ahead of the scheduled arrival time. Our results show that the inserted packets at desired times can indeed disrupt the original RTP stream without any noticeable traffic increase. The results also show that this type of attack can be done in a stealthy way and is hard to trace back via typical denial-of-service prevention schemes.

The rest of the paper is organized as follows. In Section II, we discuss the rationale for our vulnerability test, and demonstrate why our scheme is feasible in the context of VoIP. In Section III, we give an overview of related work. In Section IV, we describe the basic design of our scheme and its implementation details. In Section V, we present performance results from our experiments. Finally, we summarize our conclusions and future work in Section VI.

II. Motivation

Our vulnerability test scheme does not technically disrupt RTP or forge any harmful activities. The scheme rather distorts the perception of audio at the end-user by adding bogus duplicate RTP packets. *How is such an attack possible?* The answers are twofold: 1) it is not viable for ISPs to inspect RTP payloads, and 2) there is no clear evidence of malicious behavior except the perception of the end-user.

The ISPs do not inspect RTP payloads, encrypt and authenticate messages [8], or conduct any sort of deep packet analysis

because it is computationally expensive and introduces delay in real-time streams. To verify RTP payloads, an ISP would need to cache some of the current RTP payloads, and then later it can execute a new packet with the cached data through some sort of Digital Signal Processing (DSP). This process entails expensive equipment to run in near real-time. When the end-user perceives the low quality of the call, the end-user simply hangs up and complains to the ISP. This, however, does not help the ISP much identify any erroneous activities. If the ISP examined the phone records (technically without access to the content), the ISP would find out that the call had been started and ended correctly with no problem. It is also illegal for the ISP to wiretap the call without a warrant even for quality control purposes. Our attack will indeed not be detected by denial-of-service prevention mechanisms nor will it be flagged as any kind of anomalous behavior.

In order to verify the feasibility of our idea, we conducted a simple experiment. The testbed consists of two PCs as a caller and callee that run Debian Linux with the kernel 2.6.21. As these two PCs communicate, another PC that also runs the same Linux attempts to disrupt the packets between the caller and callee as an attacker. The attacker detects voice packets, synchronizes to them, and adds its own customized RTP packets in small bursts. The primary goal of this experiment is to test whether RTP streams are predictable. If predictable, the attacker should be able to inject additional RTP packets shortly before the original RTP packets. The callee will then receive the injected packets instead and experience degraded voice quality. This will ultimately disrupt the entire communication without being detected due to its small bursts.

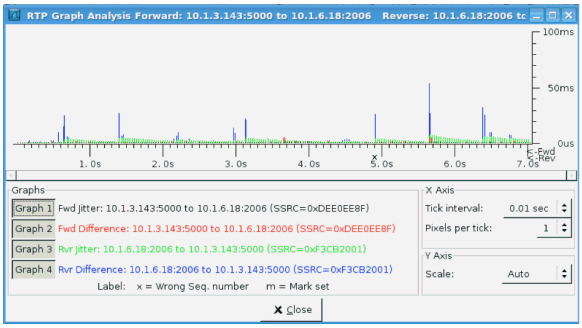


Fig. 1. RTP jitter analysis performed via Wireshark

Figure 1 illustrates a snapshot of RTP jitters in the network. We analyze this RTP jitter data using Wireshark [9]. The horizontal axis denotes the timeline of the RTP stream, and the vertical axis represents the amount of jitter. Out of the four separate statistics at the bottom of the graph, "Graph 1" and "Graph 3" show the jitters in the forward and reverse directions, respectively. The graph indicates that no jitter exists for the majority of the call and the jitters are never longer than 50ms if there is any. In our other experiments, no jitter higher than 1ms was observed for calls on a local and private network. These results strongly suggest that we can add our malicious RTP packets at the times that we wish in a more predictable way.

III. Related Work

VoIP digitizes small time quantums of voice data into IP packets and sends them across the Internet in a stream from one IP phone to another. Since each packet in the stream is valid only for a few milliseconds (usually 20-60ms), it would extremely be hard to recover if a packet gets lost or corrupted. VoIP is a conglomeration of multiple protocols, most notably 1) a signaling protocol and 2) a transport protocol. A signaling protocol initializes both endpoints and relays call channel information as well as transport protocol specifications.

SIP (Session Initiation Protocol) [1] is one of the most popular VoIP signaling protocols that helps establish, modify, and tear down call sessions between two endpoints. To make a call, a SIP user agent first registers itself with a SIP server that resolves names (e.g., e-mail addresses) to IP addresses. After that, the agent contacts the server to obtain the IP address of the receiver, and negotiates with the receiver on the details of the call including the codec, sampling rate, IP address and port to be used, and transport protocol. When negotiating call details, SIP indeed executes another protocol named SDP (Session Description Protocol) [10]. An SDP payload may contain a list of codecs the caller is aware of as well as the acceptable sampling rates. The callee, specifically the UAS (User Agent Server) examines this list, selects one option, and replies back to the caller that is the UAC (User Agent Client).

RTP (Real-time Transport Protocol) [2] is one of the most popular real-time transport protocols for VoIP. RTP defines a packet structure that incorporates payload identification, a sequence number, time stamp, and flags for a variety of conditions (e.g., padding in the voice data, extensions to the RTP being used, etc.). Additionally, RTP utilizes RTCP (Real-time Transport Control Protocol) [2] that monitors and reports on transfer states. These states include bytes sent, packets sent, lost packets, jitter, and round-trip delay. There also exist different kinds of RTCP packets: 1) SR (Sender Reports), 2) RR (Receiver Reports), 3) SDES (Source Description), 4) BYE, and 5) APP. Based on these reports, an endpoint may renegotiate the call with the other endpoint via a signaling protocol like SIP.

Kuzmanovic *et al.* investigated the shrew attack that can fail TCP congestion control with only a few more packets at specific intervals without being detected by most of the denial-of-service detection schemes [7]. In their approach, the packet bursts incurred by these extra packets continue to force TCP to back off so that the performance will significantly degrade. Rosenberg investigated a potential denial-of-service attack on RTP that enables RTP packets flooding towards a victim [11]. He also proposed a possible prevention mechanism using Interactive Connectivity Establishment (ICE).

IV. Vulnerabilities of RTP

Our system consists of three main components: 1) a controller, 2) a packet capture engine (or thread), and 3) a packet injection engine (as illustrated in Figure 2). We implement the

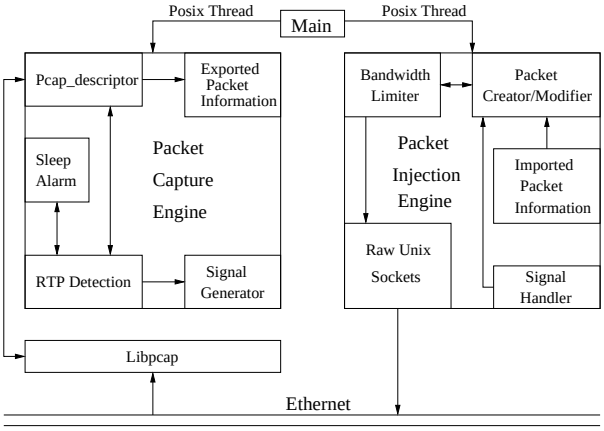


Fig. 2. Overview of our packet capture, injection, and analysis system

system on Debian Linux with the kernel version 2.6.21 using C which facilitates the near real-time processing required by our system. The packet capture and packet injection engines run as a Posix thread [12] and communicate with each other through the controller. We will discuss the details of these engines in this section.

A. Packet Capture Engine

To capture packets from the wire, we use the libpcap library [13] available in Linux. The library is executed in promiscuous mode that grabs all packets destined to the local machine. Specifically, we invoke the pcap_open_live system call as follows: pcap_descriptor = pcap_open_live("eth1", BUFSIZ, PERMISC, 1000, errorbuf). This system call opens the Ethernet device eth1 with a buffer of size BUFSIZ, a timeout of 1000ms, and an error buffer named errorbuf. This call returns a pcap_descriptor that uniquely identifies a connection in libpcap.

In addition to packet capturing, libpcap also provides packet filtering capabilities using the tcpdump [14] command line filter format. Since RTP shows no unique characteristics at the low levels (Ethernet, IP, and UDP), however, we need to develop an RTP fingerprinting method to identify RTP packets. Our observation indicates that RTP packets are 180-220 bytes in length, encapsulated in UDP and not being broadcast. These constraints are expressed as a string and passed to the packet filter compiler as pcap_compile(pcap_descriptor, kern_filter, filter_string, 0, ip) where filter_string is the constraints and kern_filter is the data structure that contains filtered results. We can set this filter in the current connection to libpcap by pcap_setfilter(pcap_descriptor, kern_filter). Although different audio encoding schemes may change packet size and intervals, the size of the RTP header does not change significantly and the packet capture engine synchornizes itself to those packet intervals.

Once the filter is in place, the only remaining step is to set our connection to non-blocking mode in which the system immediately returns if no packet is found to capture. This feature is useful because it allows more control over where time is spent. We invoke pcap_setnonblock(pcap_descriptor, 1, errorbuf) to set the libpcap connection in non-blocking mode.

After the libpcap connection is fully initialized, the packet capture engine can start the main loop to capture packets. Each time the loop determines whether or not the capture thread should end. If the thread should end, the packet capture thread informs the packet injection thread and joins with the main thread; otherwise, it attempts to grab a packet from libpcap. If the capture is unsuccessful in the latter case, there may or may not be a call underway. If there is an ongoing call, the thread should restart the loop since there will be a packet shortly; the thread goes to sleep otherwise. After successfully capturing a packet, the thread processes the packet and even copies the packet into its memory space if necessary. As the thread starts, it also begins an alarm with a 200ms timer that is reset whenever a new packet arrives. When the alarm expires, a signal is sent to the capture thread informing that no packet is arrived for the past 200ms. This in turn instructs the injection thread to go to sleep and the main loop of the capture thread restarts. The capture thread installs a signal handler to deal with the signals sent from the alarm. This handler informs the capture thread when the RTP stream is no longer available, sets the finish time of the call to the current time, and signals the injection thread to go to sleep.

B. Packet Injection Engine

As discussed earlier, the packet injection thread is largely controlled by the packet capture engine through signals. The packet capture thread issues the SIGALRM signal to the packet injection engine to synchronize the injection of falsified packets to the underlying RTP stream. One challenge is that it is hard to determine which thread receives the signal since both the capture and injection threads use the same signal. To avoid this ambiguity, we use the nanosleep() system call that uses the processor clock for scheduling instead of usleep(). Both the injection and capture threads are spawned as POSIX threads and can access each other's memory space. When the injection thread receives the signal, it copies the most recent packet from the memory space of the capture thread to its own memory space.

Although we can inject packets via libpcap, we instead use raw Unix sockets to avoid the extra overhead incurred with libpcap. Through raw Unix sockets, we can construct an Ethernet frame, an IP datagram, and a UDP packet in the user space. To create a raw socket, we call socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)). PF_PACKET allows the user to create a protocol stack in the user space, SOCK_RAW ensures that the operating system does not modify the packet, and ETH_P_ALL indicates that any high-level protocols are allowed to use this socket. We then bind the created raw socket to an Ethernet device as follows:

```
int rawsock = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));

struct sockaddr_ll sll;
struct ifreq ifr;
bzero(&sll, szieof(sll));
bzero(&ifr, szieof(ifr));
strncpy((char*)ifr.ifr_name, device, IFNAMSIZ);
```

```
if ((ioctl(rawsock, SIOCGIFINDEX, &ifr)) == -1) {
    printf("Error getting Interface index!\n");
    exit(-1);
}
sll.sll_family = AF_PACKET;
sll.sll_ifindex = ifr.ifr_ifindex;
sll.sll_protocol = htons(protocol);

if ((bind(rawsock,(struct sockaddr *)&sll,sizeof(sll)))==-1)
{
    perror("Error binding raw socket to interface\n");
    exit(-1);
}
```

To bind the socket to the interface, we first create and fill a socket address structure (sockaddr_ll) and an interface request structure (ifreq). Note that we use an I/O control to the driver (ioctl) to determine the index of the network interface device. After we fill sockaddr_ll with the index and the type of socket, we can bind the raw socket to the network driver. The packet injection engine can now send packets through this raw socket and the network device as follows:

```
if ((sent = write(rawsock, pkt, pkt_len)) != pkt_len) {
    printf("%d bytes of packet of length %d sent\n",
            sent, pkt_len);
    return 0;
}
```

The injection thread modifies a captured packet, especially the RTP header and payload, and creates a falsified packet. In order to be accepted at the receiver, the falsified packet needs to have the correct sequence number and time stamp in the RTP header. Since RTP creates predictable sequence numbers and time stamps, we can easily set the sequence number and time stamp of the falsified packet. The injection thread simply changes the content of the payload to whatever bogus data it wants to have. In our experiment, we overwrite the original payload with all As.

One problem with the packet creator and modifier is the packet injection rate. Creating a legitimate falsified packet takes much less than the time window for sending packets (20ms). Without controlling the packet injection rate, therefore, the packet injector will forge some type of denial of service attacks rather than more intelligently crafted attacks. To this end, we again use nanosleep() and limit the packet injection rate. One difficulty lies in the inaccuracy of nanosleep(). Specifically, 0-10ms delay is introduced for packet injection after sleeping. To mitigate this error, we need to re-evaluate the sequence numbers and timestamps after a round of test packets (around 15 packets) are sent. In addition, the packet capture engine sleeps for an extended period of time (roughly 500ms). To resynchronize the injection thread to the underlying RTP stream, the sequence numbers and timestamps are increased as if there have been 24 packets when in fact 25 packets were sent (500ms/20ms per packet = 25 packets). The injection thread loses one time quantum every 800ms (15 packet burst * 20ms per packet = 300ms + 500ms for sleep = 800ms). This is an unfortunate side effect of the process schedule; however, we can easily avoid this error.

V. Performance Evaluation

A. Experimental Setup

We have implemented our vulnerability test scheme and conducted experiments on a local area network (LAN) environment. Our LAN is connected to several Linux PC's and an asterisk server that runs as the router for VoIP calls. The server maintains a call list, and forwards a call to the correct receiver when it receives any request. In our test, this server replays call requests using pre-recorded calls. In addition, two PC's are equipped with soft phones and SIPp which gather live calls in cooperation with the asterisk server. These recorded calls are utilized as a testing benchmark. Figure 3 illustrates the LAN configuration for our experiments.

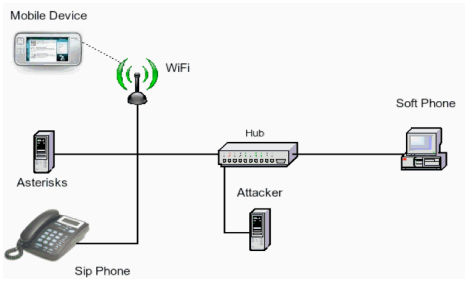


Fig. 3. LAN configuration used for our experiments

Once these calls are recorded in the proper format, we emulate real-world VoIP networks on a single Linux PC via local socket connections. As the testing scheme realizes that an actual call occurs, it starts injecting packets into the RTP stream. The scheme initially sends ten packets for 20ms, stops for 500ms, and then sends another ten packets. This process repeats until the call is terminated. All of the pre-recorded calls made between the asterisk server and any of the phones last approximately eight seconds. These calls were recorded using Wireshark [9] and replayed through SIPp across local sockets on a single machine.

To evaluate the effectiveness of the vulnerability test scheme, we analyze the sound clips produced during the calls. The baseline sound clip used for this testing contains "one two three four five six seven eight nine ten hello world" said by a woman for eight seconds. When the user dials extension one, the aforementioned sound clip is played and the call ends at the asterisk server. To recreate the sound, all of the RTP packets that constitute the call are recorded with Wireshark [9] in .au file format [15], and the rest of the packets are dropped from the packet capture. We visualize, modify, and save the sounds files using sounds mastering software called Audacity [16]. We modify the sound clip only when sound is captured in mp3 format as the sound is compressed.

B. Results

Figure 4(a) illustrates the normal speech sound clip. The bubbles in the figure represent the sound in which the first is "one", the second is "two", and so on. The last bubble is "hello world." As shown in the figure, no major interruptions

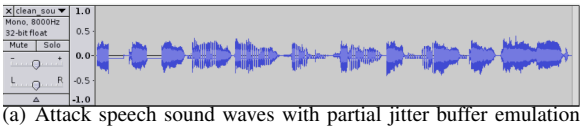
(a) Normal speech sound waves

(b) Attack speech sound waves

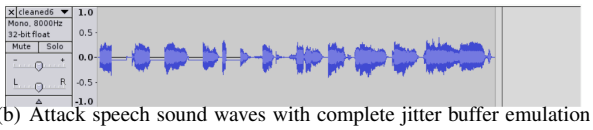
Fig. 4. Analysis of sound clip results without jitter buffer emulation

or spikes are observed in the speech bubbles, but all the sound is fairly clear and distinguishable. Compared to the normal one, Figure 4(b) shows the speech pattern observed during our vulnerability attacks. The initial part shows that the audio is interrupted and disturbed with several disconnected spikes. This clearly indicates that some fraudulent sound packets are injected into the middle of the normal sound.

Another interesting result is that the sound clip is indeed longer than the original one. This is because fake RTP packets generated by our test scheme are simply added instead of replacing correct packets in the RTP jitter buffer. All of the duplicates thus exist in the buffer and are passed up to the host for sound processing. We found that the way that Wireshark records contributes to this phenomenon. Wireshark performs no jitter buffer emulation for RTP when storing RTP streams into an audio file, but simply puts all of the sound bits together in the order that the bits are received. Hence, this result cannot be a representative one.



(a) Attack speech sound waves with partial jitter buffer emulation



(b) Attack speech sound waves with complete jitter buffer emulation

Fig. 5. Analysis of sound clip results with jitter buffer emulation

To address this problem, we modify the sound clip such that the packets arriving first in the RTP stream are kept while those arriving later are eliminated. This allows Wireshark to create the audio file that is indeed identical to the one heard by the listener. The sound clip after this process is depicted in Figure 5.

The first figure shows the interrupted speech with partially complete jitter buffer emulation; the second one shows the interrupted speech with complete jitter buffer emulation. We can clearly see a number of cutoff sounds in the middle of the first figure. These cutoff lines are the injected RTP payload produced by the testing scheme which in fact contains silence. In the second result, we can also observe a number of distorted sounds although there are not as many cutoff lines as the first figure. In this experiment, the listener actually hears many interrupted phonemes (the building blocks of speech) and has difficulty in understanding what the other person speaks.

This type of attack introduces another complexity as falsified packets are added to the RTP stream. Given that the underlying RTP stream arrives at a rate of one packet per 20ms, there will be additional 10 packets every 700ms. That is, over the course of a 1 minute phone call, there will be 3000 normal RTP packets and 857 falsified packets. The overhead of these additional packets is approximately 28.5%. We believe that this overhead is acceptable.

VI. Conclusions and Future Work

We have demonstrated that VoIP streams using RTP is susceptible to a simple packet injection attack. The attack inserts only a few additional fraudulent packets at expected times in RTP streams. The receiver accepts these fake packets instead and drops the original packets following the RTP protocol. Although the attack may corrupt the call, it is not easy to identify such an attack due to the following three reasons. First, denial-of-service prevention mechanisms are not effective because only a small number of packets are inserted; second, the ISPs are not able to inspect all of the packets, especially their payloads; third, the call appears perfectly legitimate from its header information without knowing the context. The results ensure our hypothesis. We can extend our study by collecting more statistical information that will help understand how difficult to detect our attack as the packet injection patterns vary. We can also investigate more comprehensively the reaction of denial-of-service attack detection mechanisms against our attack.

References

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261, June 2002.
[2] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 3550, July 2003.
[3] "Voice over Internet Protocol (VoIP)," http://www.fcc.gov/voip.
[4] "Skype," http://www.skype.com/.
[5] ITU, "ITU-T Recommendation H.323, Packet-based Multimedia Communications Systems," June 2006, http://www.itu.int.
[6] ——, "ITU-T Recommendation G.711, Pulse Code Modulation (PCM)," November 2007, http://www.itu.int.
[7] A. Kuzmanovic and E. Knightly, "Low-Rate TCP-Targeted Denial of Service Attacks (The Shrew vs. The Mice and Elephants)," in *Proc. of ACM SIGCOMM*, August 2003.
[8] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The Secure Real-time Transport Protocol," RFC 3711, March 2004.
[9] "Wireshark: The Network Protocol Analyzer for Windows and Unix," June 2008, http://www.wireshark.org/.
[10] M. Handley and V. Jacobson, "SDP: Session Description Protocol," RFC 2327, April 1998.
[11] J. Rosenberg, "The Real Time Transport Protocol (RTP) Denial of Service (Dos) Attack and its Prevention," Internet-Draft, June 2003, draft-rosenberg-mmusic-rtp-denialofservice-00.
[12] D. Butenhof, *Programming with POSIX Threads*. Addison Wesley, May 1997.
[13] "libpcap: the Packet Capture Library," http://www.tcpdump.org/.
[14] "tcpdump - dump traffic on a network," http://www.tcpdump.org/.
[15] "AU File Format," February 2008, http://en.wikipedia.org/wiki/Au_file_format.
[16] "Audacity: The Free, Cross-Platform Sound Editor," 2008, http://audacity.sourceforge.net/.