Lightweight Wireless Security Protocol

Paul Cychosz
cs640
May 2005

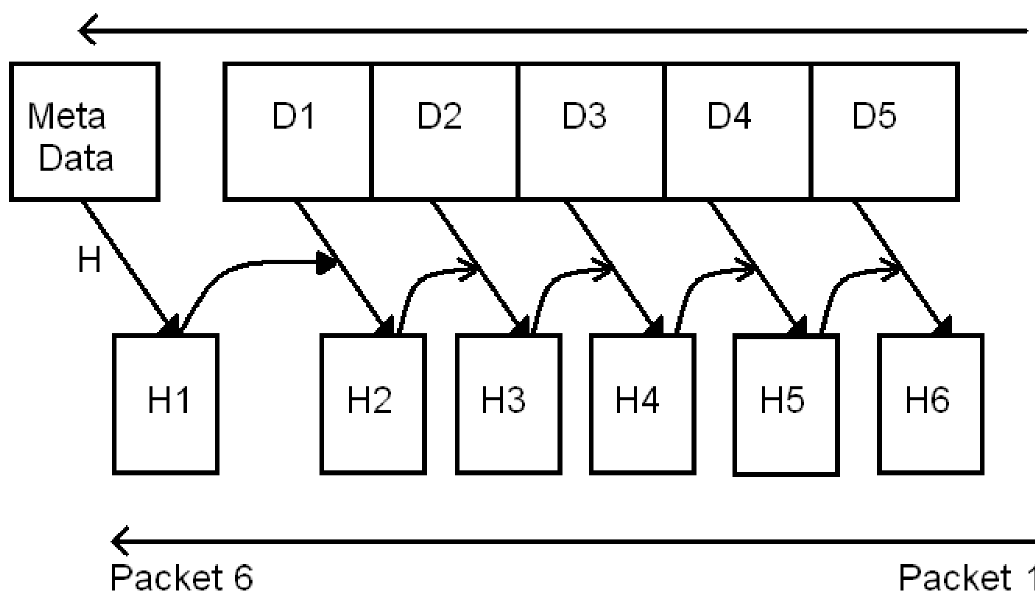
While there are many wireless protocols that involve encryption, many people do not care if their network traffic is encrypted, but they do care if it is changed, or if someone adds to it. This is a security protocol that utilizes hash chaining to ensure injection attacks cannot occur by associating each packet with the true source of the packet before it. The idea is that each hash sent along as part of the message is unverifiable until the next packet is received, thus building a dependence between packets. To correctly verify the data sent, every packet has to be unmodified, else an attack or other corruption problem can be detected and circumvented.

1 Basic Idea

The idea behind hash chaining:

Generate hash chain: $H(\, d1 \,||\, H(\, d2 \,||\, H(\, \ldots\, dN \,||\, H(\, x\,)\, \ldots)))$

This will give a set of N hashes that are all related. In this, the "||" operator means concatenation. The innermost hash is computed first, and it is computed on data, or in this case, some "special" information. In this protocol, the "x" above is a nonce, source MAC address, and sequence number. Any amount of metadata can be added though to ensure better security. Working outwards, each hash is then hashed with the packet data. This is repeated until there is no more data. See the figure below which simulates a chunk of data split over five pieces and then sent.



Once the hashes are obtained, they are paired up with the correct data and sent over the network to the target host (D5 with H6, D4 with H5….metadata with H1). Since H_n is needed to compute H_{n+1} we send all the data packets in the reverse order. This way, every time a packet is received, we can be sure all previous packets came from the same host. This protects from injection attacks as long as the hash is secure enough.

2 Implementation & Protocol

2.1 User-level simulation

To simulate that this method does indeed detect message injection, this was implemented over UDP (although it could be over any transport layer protocol). So, just in itself, it is a lightweight security protocol also applicable to non-wireless devices. The basic outline is as follows. The sending host (herein referred to as the client, to match up with the source code, client.c) wishes to send a file (or any arbitrary data) to a target host (herein referred to as the server; server.c). The client splits up the data based on the MTU. For simplicity sake, the MTU was assumed to be 1500 bytes, although the actual MTU could be retrieved via an ioctl() call (see getmac.c, a line just needs to be uncommented for this). Since the header which will be inserted at the start of 802.11 payload is 36 bytes, the data is split into 1500 – 36 = 1464 byte chunks. The client then computes the first hash which is purely metadata. This hash consists of a nonce, the MAC address, and starting sequence number, although, the hash of the entire data could easily be added ensuring better integrity overall. The hash algorithm used in this case was MD5 and was pulled from RFC 1321 with minor modifications. Using the hash consists of three parts:

```
struct MD5Context md5ctx;
MD5Init(md5ctx)
MD5Update(md5ctx, dataToHash, length1);
. . .
MD5Update(md5ctx, moreDataToHash, lengthN);
MD5Final (outbuffer, md5ctx);
```

The MD5Init sets up the buffers and resets the shift tables MD5 uses. From here, there may be any number of MD5Update calls to do compute the hash on several objects. When more than one object needs to be included in the same hash, this is the way to do it, with one MD5Update call on each object. Then, calling MD5Final fills the buffer with a 128-bit hash.

Each subsequent hash in the chain is computed over a data block, and the previously computed 128-bit hash. If a message header is modified by an attack, it will not correctly compute when the metadata block is received. While it is not any more secure to add the sequence number in with this hash, if it is added, quicker detection and faster retransmission recovery is possible. This was not added due to time constraints though. The packet sent from client → server is shown below:

Hash Chain	Length	Seq. No.	MAC addr	"double ACK"	data . . .
16 bytes	2 bytes	4 bytes	6 bytes	4 bytes	(MTU - header size)

The hash here is 16 bytes as shown, but can be reduced to 8 bytes, or whatever is wished, to reduce the header size. The length field is simply the length of the header plus data (Usually 1500 bytes except the first and last packets). The sequence number and MAC address is self explanatory. The "double-ACK" will be explained in a moment. It is used for retransmissions, but for each initial chunk transmission, it is zero. The data then follows this.

Within the client code, the data is stored in an array ready for retransmissions if needed, along with the hash chain. After every message is sent, the receive buffer is checked briefly to see if there is any data. The data sent from the server → client is as follows, this is the server to client ACK message:

Length	Seq. No.	MAC Addr	hash of this msg
2 bytes	4 bytes	6 bytes	16 bytes

The length is the size of the entire message (always 28 bytes). The sequence number is the same sequence number of the last verified message received. Notice, since the server will only send this ACK message once it has at least 2 packets, since it is not possible to verify the first packet by itself. Consider the following scenario: The server receives messages with sequence numbers 1, 2, 3, 4 and 5. Message 2 is used to verify message 1, it checks out ok, sequence number is set to "1" then. Then, say, sequence number 2 cannot be verified, possibly because of a modified 2 or 3 packet. Finally, packet 4 checks out ok by using packet 5. Regardless of good or bad packet, an ACK message like this is sent to the server every time. Even though message 4 is ok, message 2 or 3 is not. Therefore, in each ACK, the sequence number is still set to "1". This type of ACK is very similar to what TCP uses. On the client side, each time it receives an ACK, it first computes the hash and checks with the hash sent to verify the ACK message itself has not been tampered with. Provided it is ok, the sequence number is then looked up in a table to see if this ACK was received already. If it was not yet seen, it is recorded, if it was seen, this is the second ACK with this sequence number, and means there was a problem with sequence number + 1 message. It can tell there was a problem just by receiving a duplicate ACK because there is no timeout retransmission on the server side. The client will now retransmit the message that comes after the sequence number this duplicate ACK contained.

Once the client receives a packet from the server, the "double ACK" field reflects the sequence number from the ACK packet. This is piggybacked on outgoing client messages to allow the server to check that its ACK got there. If the server notices that the double-ACK value does not match the last one it sent out, and the last packet of the

chain was received from the client, it now has to retransmit the ACK packets. Instead of waiting for the entire chain to end, a threshold can be set for the number of packets received with the same double-ACK to allow for quicker retransmissions of ACK packets.

2.2 Code Specifics

At the time of this writing, this protocol has been revised several times. The implementation and code provided correctly uses the hash chain to secure communication, but the retransmission on the client and server sides does not work 100% correctly. There is a problem if the very first packet is modified. This seems to be just a simple bug, but there was not enough time to take care of it. All other packets in the chain can be modified and retransmissions will correctly resend those packets. Also, the header above from client → server is only 32 bytes, but it was described earlier as 36 bytes. In the code, it is actually 36 bytes, as a 4-byte nonce field is included in each packet as well, but it is always zero except for the same packet. This was not also changed due to time constraints, but it works correctly as it is now, and may be good to keep in every packet, because a zeroed nonce field in the header means there is more packets to come that are still part of the same hash chain.

There is a makefile included to build the server and client programs. The server takes no arguments when starting up. The client takes either 1 or 2 arguments. The first argument is the file that you wish to send. Note, however, that since this is still in the development phase, there are hard limits on the max file you can send (see the #defines in the top of server.h and client.h) which is currently set to a little less than 100KB. There is a man-in-the-middle attacker program too, but it is easier to simulate an attempt by using the second argument on the client. The second argument determines which message will be modified/injected before sending to the server. This can then be verified on the server side.

The output to stdout on the server is pretty explanatory, it will show you the hash chain, and whether or not they are ok. These can be compared with the hash chain in the client output as well if needed (they will appear in reverse order). Messages detected that have been injected or modified will print out the bad hash, and then the hash that was expected underneath.

Lastly, it is worth mentioning that the size of the fields in the headers can be tuned to give the desired security strength / header size tradeoff. Also, a better pseudo-random number generator should be used, instead of a 4-byte rand()'ed integer with a seed based on current seconds from gettimeofday().

3 Driver code

Initially, I tried to implement the security protocol into a generic prism2 driver. While this worked with the card, the documentation was terrible and *a lot* of time was wasted trying to learn the code and get things to work. Eventually, I restarted with the hostap driver (specifically the hostap_plx driver). This code is relatively organized and self-explanatory. Because I spent most of the time with the user-level program to ensure

the protocol is correct (no point in implementing a broken protocol), there is no working protocol within the driver, again, due to time constraints.

3.1 Hostap Driver Code

To start implementing the protocol in the driver, there are a few interesting areas to point out to help. First of all, the files are logically broken up, and the function names are usually self-explanatory. In hostap.c, there is a basic struct hfa384x_tx_frame, which contains useful information. Also, depending on implementation, the hostap_get_hdrlen(…) function in hostap.c can be useful if these headers are packet into the 802.11 header. This function is currently set up to make a header size of 16 bytes by default, or 30 bytes for data, or 10 bytes for a CTS or ACK packet. An idea may be to implement another header type here for this security protocol. That way, the driver can switch between normal operation, and injection-protected operation via this protocol. Throughout the hostap driver, a in-argument is usually struct net_device. This struct is used to give information about the actual card and interface, so its fields are worth learning. Lastly, in hostap.c, there is a prism2_change_mtu() function to obviously change the MTU. This is very important if piggybacking the security-protocol's header within the data MTU, because the header may push data of that packet to something greater than the max MTU. I am currently unsure, but this may be ok, because fragmentation should be handled at a lower layer within the driver (see hostap_plx.c).

The main area to focus on within the hostap driver code is everything within hostap_80211_tx.c (and its header file). The actual protocol would be implemented within the hostap_data_start_xmit(…) and hostap_mgmt_start_xmit(…) (I believe it would have to be in both places, since these are callbacks along two different code paths within the driver). One of the arguments to these functions is of type struct sk_buff. This is what needs to be modified. The sk_buff struct is part of the Linux kernel source tree, and can be found in (usually) /usr/src/linux/include/linux/skbuff.h. This was taken from a 2.6.x kernel. This struct is a base-like structure for any kind of data packet. (i.e. it contains unions of fields for UDP, TCP, etc.). The last struct that needs to be dealt with is hostap_ieee80211_hdr. The fields in this struct are straightforward, in fact, fields of it might be able to be hijacked to use for this security protocol (the extra address fields).

There is a makefile already included for the driver itself, and will automatically find the Linux source tree and compile correctly. A 'make install' after that will finish the installation of the driver, then to test it, it can simply be enabled via insmod or modprobe.