The Design and Implementation of Protocol-Based Hidden Key Recovery

Eu-Jin Goh[1], Dan Boneh[1], Benny Pinkas[2], and Philippe Golle[1]

[1] Stanford University
{eujin,dabo,pgolle}@cs.stanford.edu
[2] HP Labs
benny@pinkas.net

Abstract. We show how to add key recovery to existing security protocols such as SSL/TLS and SSH without changing the protocol. Our key recovery designs possess the following novel features: (1) The Key recovery channels are "unfilterable" — the key recovery channels cannot be removed without also breaking correct operation of the protocol. (2) Protocol implementations containing our key recovery designs can inter-operate with standard (uncompromised) protocol implementations — the network traffic produced is indistinguishable from that produced by legitimate protocol implementations. (3) Keys are recovered in real time, hence most or all application data is recovered. (4) The key recovery channels exploit protocol features, rather than covert channels in encryption or signature algorithms.
Using these designs, we present practical key recovery attacks on the SSL/TLS and SSH 2 protocols. We implemented the attack on SSL/TLS using the OpenSSL library, a web browser, and a network sniffer. These tools allow us to eavesdrop on SSL/TLS connections from the browser to *any* server.

1 Introduction

Intuitively, protocol key recovery means that a third party, by observing network traffic, can recover the secret key protecting the data transferred between two parties. Unlike previous key recovery or escrow schemes that work at the level of encryption and signature algorithms [23, 24], our designs use features of the protocol. We also consider practical issues such as ease of deployment and usage.

Our key recovery designs possess the following properties.

1. Unfilterability
   Network filters cannot remove key recovery channels from protocol messages without also breaking the correct operation of the protocol.
   Previously suggested key escrow schemes and classic covert channels do not have this unfilterable property. For example, timing-based channels introduced by Lampson [14] leak secret bits by using small variations in packet transmission time. The timing-based covert channel is easily filtered by a proxy that ensures all packets are sent at regular intervals. Our key recovery designs do not use such channels and cannot be filtered by such proxies.

2. Inter-operability
   A protocol implementation with key recovery works seamlessly with other implementations that do not contain key recovery while still supporting key recovery. Keys can be recovered from any protocol session in which *at least one* of the participating parties support key recovery. For example, a SSL web server with key recovery inter-operates with a standard (uncompromised) web browser while still leaking key information.
   Note that maintaining the ability to inter-operate precludes protocol changes. Previously suggested key recovery systems [5, 12, 4] require protocol redesigns.
3. Hidden key recovery
   The network traffic generated by a protocol implementation with key recovery is computationally indistinguishable from traffic generated by implementations without key recovery.
4. Real time key recovery
   Session keys are recovered before user data is exchanged.

These properties are essential for any practical key recovery scheme. The inter-operability and hiding properties allow incremental (and unnoticeable) deployment of servers and clients with key recovery. Unfilterability ensures key recovery channels cannot be easily circumvented. Real time operation allows instant decryption and analysis of application data without large storage requirements.[3]

In this paper, we design key recovery schemes for SSL/TLS and SSH2 that possess the stated properties. The basic method for adding key recovery to a security protocol is to change how the protocol implementation generates fields that are "random looking". For example, when the server needs a random nonce, the server uses the encryption of the session key as the nonce. Assuming the cipher is semantically secure [1], this ciphertext is a valid nonce because it is indistinguishable from a random string.

2 Applications and Threats

In this section, we discuss several consequences of hidden and unfilterable key recovery systems.

2.1 Hackers

A hacker can break into a SSL web server and patch the SSL library with hidden key recovery. This attack allows the hacker to eavesdrop on all SSL connections to the server without ever having to access the server again. From the network point of view, there is no detectable difference between the original and the hacked server. Therefore, it is hard to detect these attacks without host based intrusion detection system such as Tripwire [13].

[3] This property seems well suited for a wiretapping device such as the government's Carnivore program [9].

2.2 Governments

The government can convince major software vendors to distribute SSL/TLS or SSH2 implementations with hidden and unfilterable key recovery. Using a Carnivore device, the government can monitor all encrypted connections established using these protocols in which either the client or server has been compromised. Users will not notice the key recovery mechanism because the scheme is hidden. Even savvy users who use a proxy to counter standard covert channels cannot circumvent key recovery because the channels are unfilterable.

Our designs provide an easy way of deploying a government key recovery system. For example, key recovery enabled web browsers can be installed at public libraries to ensure that criminals and terrorists do not conduct their business using SSL sessions at these facilities. The social and political implications of a government key recovery system are outside the scope of this paper.

2.3 Trusting Closed Source Products

While the design and implementation of security protocols should be left to experts, end users must verify that the implementation correctly follows the protocol. In particular, users should ensure that implementations do not contain the key recovery mechanisms described in this paper.

Unfortunately, validating implementations is often impractical. Most commercial software libraries are closed source and can only be validated using black box testing. Black-box testing is carried out by running the product and observing external state (such as generated network traffic) to ensure that it works as promised. Hidden key recovery, however, cannot be detected by observing external state. Hence, black-box testing is insufficient for validating security products. Note that checking MD5 fingerprints and standard code signing techniques is also inadequate for detecting compromised implementations.

The problem is much harder for hardware implementations of security protocols. Many vendors already sell hardware implementations of IPsec. Next generation SSL accelerators implement the entire SSL protocol in hardware. Note that even FIPS 140-2 [16] certified hardware must be checked. Indeed, it is harder to verify FIPS level 3 and 4 certified hardware because they are designed to be tamper resistant. Furthermore, the FIPS certification might provide users with a false sense of security. How can end-users ensure that these implementations do not provide hidden key recovery?

It is currently an open problem whether there is a way for a closed implementation to prove to the outside world that it is properly following the protocol and does not implement our hidden key recovery. Therefore, it is fair to say that closed source software and hardware implementations of security protocols cannot be fully trusted since it is hard for end-users to validate them.

3 Assumptions and Definitions

We define some useful terms and also state our assumptions about the operational model.

Escrow Agency Field (EAF). In our key recovery system, a key (such as a master key or a session key) that enables decryption of network traffic is encrypted with another key (the recovery key). *The result of encrypting the session key with the recovery key is called the Escrow Agency Field, or EAF.*[4] The recovery agency can decrypt the EAF using the recovery key. We also assume that the ciphers used for key recovery are semantically secure [1].

Model. We consider three entities, a client-side user, a server-side user and a recovery agency. The client-side and server-side users use a standard protocol P to communicate securely while the recovery agency aims to decrypt their communication. Both users execute P correctly and neither colludes with the recovery agency.

Hidden key recovery. Given a cryptographic protocol P with specified inputs and outputs, let P' be a new protocol created with key recovery. Consider a device implementing P which uses a set of keys K, and a device implementing P' that knows the same set of keys K (and possibly some other keys that P' uses). We say that P' contains *hidden key recovery* if the inputs and outputs of the implementations of P and P' are computationally indistinguishable to everyone except the key recovery agency. Hence, P' conforms with the specifications of P, and P' can inter-operate seamlessly with all implementations of P.

4 Design and Implementation

In this section, we provide an outline of the techniques for adding hidden and unfilterable real time key recovery to existing security protocols.

4.1 Hidden and Unfilterable Key Recovery

Hidden Key Recovery. The basic idea is to embed the Escrow Agency Field (EAF) in protocol fields containing random looking data. The rationale is that the EAF is a ciphertext; A semantically secure cipher produces ciphertexts that look random to everyone except the recovery agency. Therefore, the EAF or parts of the EAF can be sent in place of any field containing random data.

For example, when the protocol requires a random nonce, the nonce is replaced with the EAF. Note that we are excluding standard covert channels because they are easy to filter (see Section 8).

We prefer encrypting the EAF using a asymmetric (public key) encryption scheme as opposed to a symmetric (bulk) encryption scheme. If a symmetric encryption scheme is used, every implementation must contain a unique key so as to prevent a system wide compromise by an attacker extracts the key from a single client. On the other hand, a large number of unique keys results in a key

[4] This field has the same functionality as the Law Enforcement Agency Field (LEAF) in the Escrow Encryption Standard (EES) scheme [5].

management nightmare for the recovery agency. If a asymmetric cipher is used, then only the public key is embedded in every implementation of the protocol. In this case, an adversary that reverse engineers a compromised implementation only recovers the public key and cannot decrypt any messages.

Unfilterable Key Recovery. For a key recovery scheme to be unfilterable, key information must be hidden in fields that are essential for the correct operation of the protocol. For example, the SSL protocol requires implementations to verify a checksum of all the messages exchanged before the negotiated cryptographic parameters are used. If any message is altered by a third party such as a filter, the verification will fail, halting further communication.

4.2 Real Time Key Recovery

We use the following criteria to determine which messages in a protocol are suitable for real time key recovery — information hidden in a session must be related to the current session key. We can hide information in a message about the current session key only if we have that information before the message is sent. Otherwise, fields in that message can be used to store older session keys.

Encryption keys shared between two parties are usually generated from a combination of secret and public randomness. Either the encryption key or the secret part of the randomness can be escrowed. Since the encryption key is usually smaller than the secret randomness, it appears that the encryption key should always be escrowed. If, however, the only suitable key recovery channels are in messages exchanged before key generation, and one party generates the entire secret randomness, then the system should hide the secret randomness instead. We will see an example of this in SSL.

4.3 Using Low Capacity Channels

In this section, we describe techniques for hiding the EAF in very short fields. The motivation for developing these techniques is because in many protocols, fields suitable for key recovery are usually very short and cannot contain the entire EAF.

Recall that the EAF is encrypted with the recovery agency's public key (recovery key). The shortest public key cipher is a variant of the ElGamal cryptosystem over elliptic curves (ECEG) [15]. ECEG produces ciphertext that is 327 bits ($\approx$ 41 bytes) long if the plain text is no greater than 163 bits ($\approx$ 20 bytes). We therefore assume that 41 bytes is the minimum capacity for hiding an EAF encrypted using an asymmetric cipher. We also assume that the ElGamal cryptosystem over standard elliptic curves is semantically secure.

How much information to hide. To minimize the EAF size, the key information should be less than 20 bytes. Recall that we can hide either the session key or the secret randomness used to generate session keys. Without loss of generality, we can assume that the key information a compromised implementation discloses is shorter than 20 bytes. If the session key or secret randomness is longer than 20 bytes, the compromised implementation can generate them from a short seed (less than 20 bytes) using a pseudo-random generator. This seed is leaked instead of the full key or randomness. Therefore, at most 41 bytes of capacity is required for hiding the ECEG encrypted EAF.

Unfortunately, a 41 byte EAF is often larger than the capacity of a single protocol session. We describe three methods for hiding the EAF even if each protocol session has less than 41 bytes of capacity.

Method 1. Since a ECEG encrypted EAF cannot fit into the capacity of one session, the EAF is split and distributed over a few sessions. One drawback is that this method cannot support real time eavesdropping. Another drawback is that the recovery agency, to decrypt a particular session, has to assemble and correlate key information recovered over a number of sessions. The recovery agency also has to store all encrypted traffic until they can be decrypted.

Method 2. Method 2 is a hybrid method that has comparable security to the first while mostly providing real time eavesdropping. The compromised implementation performs the following procedure.

1. Picks a random key K for a symmetric cipher. K is used to encrypt the EAFs. Any semantically secure symmetric cipher can be used.
2. Use ECEG with the public key of the recovery agency to encrypt K. Denote $E[K]$ as the encryption of K under the public key of the recovery agency. Recall that $E[K]$ is 41 bytes long.
3. Divide $E[K]$ up into shares so that each share is no bigger than the capacity of a single session. Let i be the number of shares required to reconstruct $E[K]$. $E[K]$ is then hidden over i sessions. Note that we cannot hide any other information for those i sessions.
   Erasure codes can be used to divide the EAF into shares to provide redundancy against lost or missed sessions.
4. Once K has been transmitted, the EAF in future sessions is encrypted using the symmetric encryption scheme instead of ECEG. For example, we can use IDEA with key K and 0 IV.

One disadvantage is that the recovery agency cannot eavesdrop on the initial i sessions. Another disadvantage is that the recovery agency has to intercept at least i sessions containing $E[K]$. If less than i sessions are intercepted, the recovery agency cannot eavesdrop until $E[K]$ is sent out again. In fact, it is a non-trivial task to differentiate between sessions containing encrypted session keys and those containing $E[K]$. Therefore, this hybrid method is only suitable if the recovery agency can deduce when $E[K]$ is transmitted.

Method 3. This method compromises the security of the cryptosystem protecting the EAF for the ability to carry out real time eavesdropping. For example, the compromised implementation can use ECEG with a shorter key to encrypt the EAF. In this case, the largest ECEG key is used to encrypt the EAF, while ensuring that the EAF still fits in the available capacity. One disadvantage is that an adversary can extract the private key from the public key in reasonable time.

5 Overview of TLS

We provide an overview of the Transport Layer Security (TLS) protocol [8] before describing our key recovery attacks on TLS. Readers who are familiar with TLS can proceed to the next section.

TLS provides communication privacy and data integrity between applications. TLS is used on the Internet to secure data traffic between HTTPS clients and web servers. TLS is the successor to the Secure Socket Layer 3.0 (SSL3) protocol [10]. TLS and SSL3 are very similar and unless otherwise noted, SSL3 shares the same vulnerabilities as TLS. For brevity, we will only describe TLS.

TLS runs on top of a reliable transport protocol such as TCP/IP. It consists of two layers, the record protocol and the handshake protocol. The record protocol provides a private connection using symmetric encryption and keyed MACs. The handshake protocol is layered over the record protocol without any cryptographic parameters, and provides peer authentication and shared secret negotiation.

A TLS session occurs in two phases. First the handshake protocol is run to authenticate the server to the client (and optionally vice versa). The TLS handshake protocol supports RSA, Ephemeral Diffie-Hellman (EDH) and other key exchange algorithms. In this paper, we will only refer to TLS as it is most commonly used with RSA key exchange and server only authentication. The client and server also agree on a symmetric cipher, a MAC function, and a compression algorithm. The cryptographic keys are derived from a combination of a premaster secret chosen by the client and random strings chosen by both parties. Finally, the application's protocol runs over the record protocol with the cryptographic parameters negotiated in the handshake.

Next, we give a simplified description of the messages exchanged during the handshake protocol. Figure 1 provides an overview of the messages exchanged during a TLS handshake.

1. *Client Hello:* The client sends a list of supported ciphers, together with 28 bytes of randomness and a 4 byte time-stamp. These 32 bytes make up the client randomness.
2. *Server Hello:* The server chooses a cipher among those supported by the client. The server also sends 28 bytes of randomness with a 4 byte time-stamp and a session identifier (session ID). The session ID can be up to 32 bytes long. The 28 bytes of randomness and the 4 byte time-stamp make up the server randomness. The session ID is used for resuming sessions with previously established cryptographic parameters.
3. *Server certificate:* The server sends a chain of $X509$ certificates to authenticate itself to the client. If requested by the server, the client may also authenticate itself to the server.

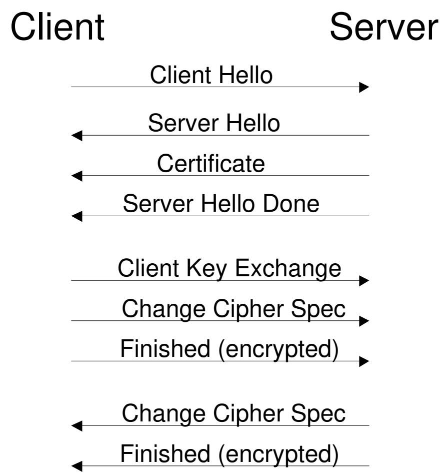

Fig. 1. TLS Handshake with RSA key exchange

$$\begin{aligned} \text{TLS Record } &= \text{headers} \mid\mid \text{encrypted data} \\ \text{encrypted data } &= E\big[\text{data} \mid\mid \text{MAC(headers} \mid\mid \text{data)} \mid\mid \text{pad}\big] \end{aligned}$$

Fig. 2. TLS Record Structure. || denotes concatenation, E[ ] and MAC() denote symmetric encryption and message authentication. Padding is unnecessary for stream ciphers

4. *Client Key Exchange:* The client sends a 48-byte premaster secret encrypted with the public key in the server's $X509$ certificate.
5. *Client and Server Finished:* The client and server separately derive common cryptographic parameters from the shared premaster secret and randomness. They also verify the key exchange and authentication by exchanging the output of a pseudo-random function applied to a hash of all handshake messages and a predefined string. These two messages are encrypted with the negotiated parameters.

After the handshake, the server and client share a common set of cryptographic parameters which are used to secure further communication. The format of TLS messages in the record protocol is shown in Table 2.

6 Key Recovery in TLS

In this section, we examine TLS to find subliminal channels for key recovery. We then discuss our implementation of the key recovery scheme in TLS and SSL 3.0. Table 1 summarizes our findings on which message fields in TLS can be used

Table 1. Candidate fields for key recovery in TLS and SSL. These fields are all unfilterable

Field	Whom	Size (bits)	Frequency	Session	Restrictions
CipherSuites	Client	a few	session	current	
Randomness	Client	224	session	current	
Time stamp	Client	a few	session	current	
Randomness	Server	224	session	previous	
Time stamp	Server	a few	session	previous	
Session ID	Server	up to 256	session	previous	
Pad Contents	Both	52	record	current	SSL 3.0

for hidden key recovery. Note that the listed channels are unfilterable. For each field,

- **Whom:** indicates whether the field can be used by either the client, the server or both.
- **Size:** the size (in bits) of data that can be hidden in the field.
- **Frequency:** indicates whether a field can be used once per TLS session or once per TLS record.
- **Session:** describes the most recent session key that can be leaked in the field.
- **Restrictions:** indicates any restrictions that may apply to using the field.

Surprisingly, even the block cipher padding in SSL 3.0 can be used as a channel. We spend the rest of this section describing how to hide information in padding schemes.

If a block cipher is used to encrypt the record, the record has to be padded to the cipher's block size. At first glance, the pad seems useless as a subliminal channel because it is encrypted. However, we can manipulate the pad so that its cipher text serves as a subliminal channel. Padding for block ciphers is different in TLS and SSL3. We first discuss SSL 3.0 padding.

SSL 3.0 Padding. The pad can be up to the block size, which is 8 bytes in most ciphers. The last byte of the pad specifies the pad length excluding the last byte. The following code snippet shows how the pad is checked in the OpenSSL implementation of SSL 3.0:

```
// here bs is the block size
l=rec->length;
i=rec->data[l-1]+1;
if (i > bs) return
(SSL_ERROR_BLOCK_CIPHER_PAD_IS_WRONG);
rec->length-=i;
```

Only the last byte of the pad is checked because the other bytes are unspecified in the protocol.

We assume that the block size is 8 bytes. We can also assume that the data length in the record is a multiple of the block size because the TLS implementation controls the amount of data in each record. If the data length is a multiple of block size, the pad length will be 8 bytes (64 bits).

We create a 52 bit subliminal channel in the following manner — fix the first 52 bits of the ciphertext to contain the EAF. Next, iterate over the 2^{12} possible values of the last 12 bits of the cipher text block until the decryption of the last byte is 7. We will be able to find the desired cipher text with very high probability because we are iterating over 2^{12} possible values.

TLS Padding. The pad can be between 0 and 255 bytes if the record size is a multiple of the block size. All bytes in the pad are set to the length of the pad. Only the pad length can be used as a subliminal channel because the content of the pad is specified. Given an encrypted record, however, we cannot determine the length of either the data or padding segment. As a result, we cannot use the padding in TLS as a subliminal channel.

6.1 Implementation in SSL/TLS

This section describes our implementation of key recovery in SSL/TLS.

Hidden and Unfilterable Key Recovery. The TLS implementation containing key recovery should be indistinguishable from standard TLS. We can achieve this goal by using only the subliminal channels listed in Table 1, which are all unfilterable. We ignore the lower capacity channels and focus on the client randomness, server randomness and server Session ID.

Real time Key Recovery. To carry out key recovery in real time, the EAF for a TLS session must contain key information for the current session. Examining the session column in Table 1, we see that only the client randomness field and the pad contents (for SSL 3.0) are suitable.

Session keys in SSL 3.0 and TLS are created by applying a pseudo-random function (PRF) to the concatenation of the public client randomness, public server randomness, and premaster secret. When RSA is used for key exchange, the client creates the premaster secret. The client can generate the session keys only after it receives the server random in the Server Hello message. Unfortunately, the only channel suitable for real time key recovery is the client randomness field found in the first message exchanged. Therefore, the only way to carry out real time key recovery using a compromised client is to hide the premaster secret instead of the session key.

The client can leak the premaster secret in the Client Hello message because the client generates the entire premaster secret when RSA key exchange is used. Even if the server does not contain key recovery capabilities, the client can compromise the communication between them. If Diffie-Hellman key exchange is used, the client creates only part of the premaster secret. In this case, the client cannot compromise the secure connection without the help of the server.

On the other hand, the server can only generate the session keys after it receives the encrypted premaster secret in the Client Key Exchange message. Therefore, it is impossible to implement real time key recovery in TLS on the server. Although real time key recovery on a SSL 3.0 server is possible, it is infeasible because of the small capacity of the pad contents channel. Furthermore, it is only possible if a block cipher is selected as the symmetric encryption algorithm. We shall concentrate on implementing key recovery in the client.

Low Capacity Channels. Recall an ECEG encrypted EAF is 326 bits (41 bytes) long. The client randomness field, however, is only 28 bytes long. If a block cipher is used with SSL 3.0, we can use the pad as another channel. The pad has a capacity of 6.5 bytes, giving a total of 34.5 bytes (276 bits).

The premaster secret comprises of 46 securely generated random bytes and 2 bytes denoting the client's TLS version. The client version is a public value sent in the Client Hello message. Hence, we only need to leak the 46 random bytes. As suggested in Section 4.3, we leak a small seed of 16 bytes and derive the premaster secret from it. This derivation is done by applying a SHA-1 hash on thirds of the seed. We only use 46 out of the 60 bytes generated by the hashes.

To encrypt the EAF, we can use either a short (224 or 276 bits) RSA key (a variant of method 3 in Section 4.3), or the hybrid scheme (method 2). The hybrid scheme leaks the encrypted symmetric key over two handshake sessions, meaning that the recovery agency cannot decipher the traffic of these two sessions. Furthermore, if the recovery agency misses either session, it cannot decrypt any traffic until the encrypted symmetric key is sent again.

Prototype implementation. We implemented a prototype for the client side key recovery scheme in SSL 3.0 and TLS using OpenSSL [17], a free SSL library. For SSL 3.0, we implemented the hybrid scheme where we use ECEG to protect a 128 bit IDEA key and leak it over two handshakes. We use IDEA in CBC mode to protect the 16 byte seed used to generate the premaster secret. ECEG was implemented using routines provided by Miracl [19]. For TLS, we implemented a variant of method 3 which uses a short (224 bit) RSA key to protect the EAF.

Both TLS and SSL 3.0 required only a few lines of additional code to call on our key recovery routines. The key recovery routines are contained in a source file containing about 400 lines of unoptimized C code. These key recovery routines can be reused for other protocols. The small amount of extra code added to both protocols demonstrates the ease of implementing key recovery.

We compiled w3m [11], a web browser, with the modified SSL library to show compromised HTTPS connections. We also modified ssldump [18], a SSL packet sniffer, to serve as the recovery agency. It decrypts the EAF to obtain the premaster secret which is used to generate the session key. The sniffer then decrypts the session traffic and prints out the decrypted contents. We do not handle session resumes in our implementation but this can be easily added.

The computational overhead imposed by key recovery is trivial compared to the total amount of work done by the client. For SSL 3.0, the IDEA key is generated and encrypted with ECEG off-line. Hence, no extra work is carried out by the client during the initial two SSL handshakes. Subsequent handshakes incur the cost of a encrypting the premaster secret seed with IDEA. For TLS, every handshake incurs the cost of an RSA operation with a very short modulus. Note that the EAF can be generated and encrypted off-line.

7 Key recovery in SSH version 2

In this section, we describe a simple hidden and unfilterable real time key recovery attack on SSH2.

SSH2 Overview. SSH is a secure shell protocol designed to replace insecure login mechanisms such as telnet. Version 2 of the protocol (SSH2) is designed by the `secsh` working group in the IETF [20] to solve the first version's shortcomings. Like TLS, SSH2 is divided into several layers. The lowest layer is the Transport Layer Protocol, which runs over TCP/IP. The Transport Layer is analogous to the Record Layer in TLS. The next layer is the Authentication Protocol, which runs over the Transport Layer. The last layer is the Connection Protocol, which runs over the Authentication Protocol. We will focus on the Transport Layer Protocol.

All messages in SSH2 are carried at the Transport Layer by the Binary Packet Protocol. This protocol defines the record structure shown in Table 2. The packet_length field is the byte length of the packet, not including itself or the MAC. The length of the padding must be between 4 to 255 bytes, and the padding should consist of random bytes. Furthermore, both block and stream ciphers are required to pad the structure so that the total size (in bytes) of the second to fifth fields is either a multiple of the cipher block size or 8, whichever is larger. The payload_length field is calculated by the following equation — payload_length = packet_length - pad_length - 1. The length fields, payload and padding are included in the computation of the MAC.

Table 2. Format of a SSH2 Binary Packet

field type	field description
unsigned int (32 bits)	packet_length
byte	pad_length
byte[payload_length]	payload
byte[pad_length]	random padding
byte[mac_length]	MAC

SSH2 Key Recovery. When SSH2 is used as the secure equivalent of telnet, a packet is sent and returned for almost every keystroke. This mode of usage results in a network traffic pattern consisting of a large number of small packets

exchanged between the server and client. Furthermore, every packet contains a single keystroke, resulting in only 1 byte of payload per packet. The total size of the second, third, and fourth field is 6 bytes. According to the padding rules stated in the previous paragraph, there will be at least 8 bytes of ciphertext derived directly from the pad. In SSL 3.0, the last byte of the padding specifies the pad length. In contrast, the pad length in SSH2 is specified in a separate field. The pad contents are specified as random bytes. In a single packet, 8 bytes of the EAF can be hidden as the ciphertext of the pad.

For our key recovery attack on SSH2, the symmetric key used to encrypt the binary packets is hidden in the EAF. If ECEG is used to encrypt the EAF, the EAF can be completely disclosed in 5 to 10 packets sent by only one party. The probability of either the client or server sending at least 10 packets is very high because of the network traffic pattern.

Comparison to SSL. For the padding attack to be viable in SSL 3.0, the symmetric encryption algorithm has to be a block cipher. In SSH2, however, padding is required for all symmetric encryption algorithms, including stream ciphers. Hence, SSH2 is always vulnerable to our key recovery attack. In addition, either the client or the server can carry out the key recovery attack.

8 Related Work

This paper draws on contributions from two complementary lines of research — key escrow mechanisms and subliminal channels. In contrast to previous work, we examine the threat of key escrow through subliminal channels at the protocol level rather than the cryptosystem level.

Key escrow refers to methods allowing participants to hold encrypted communication while a third party holds the secret key for the communication. A basic key escrow scheme takes the session key, encrypts it using an escrow agency's key, and sends this information together with the encrypted session data. Key escrow received a large amount of attention with the (failed) introduction of the Escrow Encryption Standard (EES) [5] by the US government. Proposed key escrow schemes at the protocol level, such as the EES, rely on known key escrow fields such as the LEAF. Blaze showed that the LEAF is filterable [2]. Denning [4], Kilian and Leighton [12] provides a survey of key escrow schemes.

Simmons [21] gave the first definition of subliminal channels are defined as means to convey information in the output of a cryptosystem such that the information is hidden to everyone except the escrow party. Simmons also gives the first practical example of a subliminal channel for ElGamal signatures [22]. Desmedt et. al. provide another example using the Fiat-Shamir authentication protocol [7]. As a solution, Desmedt advocates the use of active wardens to eliminate the possibility of subliminal channels in a variety of protocols [6].

Young and Yung [23] investigated subliminal channels in public key cryptography. In their paper, the compromised key generation algorithms selects keys for the RSA and ElGamal public key cryptosystems such that the public key reveals the private key to the escrow agency. Young and Yung later expanded and refined these ideas [24].

Lampson mentions a related idea of covert channels [14]. He defines covert channels as channels that are not intended for information transfer. An example of a covert channel is the timing delay between transmitting network packets. We do not consider covert channels in this paper because they tend to be easily filterable. For example, if an adversary uses timing delay between messages to hide information, then adding random delays to messages would degrade the channel. Following Desmedt [6], we only consider channels originally intended to exchange information.

9 Summary and Conclusions

We showed how to add unfilterable and hidden key recovery to any security protocol. We also described methods for recovering key information in real time, as well as in low capacity channels. We used these methods to carry out our key recovery attacks on SSL 3.0, TLS and SSH2. We implemented a prototype of hidden and unfilterable real time key recovery in SSL 3.0 and TLS.

Our results show the ease with which hidden and unfilterable key recovery is added to existing security protocols. In particular, we can undetectably add key recovery without changing the protocol. Our prototype implementation illustrates the danger of trusting closed source and hardware implementations of security protocols.

It is not easy to design security protocols that resist our hidden key recovery attack. Canetti and Ostrovsky [3] define a model of distributed computation with honest-looking parties and prove some initial results on the feasibility of secure function evaluation in this model. It is currently an open problem to build a practical conversion procedure that will translate any security protocol into a protocol that will remain secure in the presence of honest looking parties.

Despite this, protocol designers should still consider this family of attacks when designing security protocols. The authors of SSH state that SSH2 was "not designed to eliminate covert channels". As we have shown, it is particularly easy to carry out our key recovery attacks on SSH2 because of its cipher padding scheme. Our attack on SSH2 will be seriously hindered if the protocol used a padding scheme similar to the one used in TLS. We have only examined SSL 3.0, TLS and SSH2. Our ideas will apply equally well to other security protocols such as IPsec and Kerberos.

References

1. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *Proceedings of Crypto 1998*, volume 1462 of *LNCS*, pages 26–45. Springer-Verlag, Aug 1998.
2. M. Blaze. Protocol failure in the escrowed encryption standard. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 59–67. ACM, ACM Press, Nov 1994.

3. R. Canetti and R. Ostrovsky. Secure computation with honest-looking parties: What if nobody is truly honest? In *Proceedings of the 31st Symposium on Theory of Computing*, pages 255–264. ACM, ACM Press, May 1999.
4. D. Denning. Descriptions of key escrow systems. Technical report, Georgetown University, Feb 1997. http://www.cs.georgetown.edu/∼denning/.
5. D. Denning and M. Smid. Key escrowing today. *IEEE Communications*, 32(9):58–68, 1994.
6. Y. Desmedt. Abuses in cryptography and how to fight them. In S. Goldwasser, editor, *Proceedings of Crypto 1988*, volume 403 of *LNCS*, pages 375–389. Springer-Verlag, Aug 1988.
7. Y. Desmedt, C. Goutier, and S. Bengio. Special uses and abuses of the Fiat-Shamir passport protocol. In C. Pomerance, editor, *Proceedings of Crypto 1987*, volume 293 of *LNCS*, pages 21–39. Springer-Verlag, Aug 1987.
8. T. Dierks and C. Allen. The TLS protocol. RFC 2246, Jan 1999.
9. FBI. Carnivore diagnostic tool. http://www.fbi.gov/hq/lab/carnivore/carnivore.htm.
10. A. Freier, P. Karlton, and P. Kocher. The SSL protocol version 3.0. http://www.netscape.com/eng/ssl3/draft302.txt, Nov 1996.
11. A. Ito. w3m: text based browser. http://w3m.sourceforge.net/.
12. J. Kilian and T. Leighton. Fair cryptosystems, revisited: A rigorous approach to key-escrow. In D. Coppersmith, editor, *Proceedings of Crypto 1995*, volume 963 of *LNCS*, pages 208–221. Springer-Verlag, Aug 1995.
13. G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, ACM Press, 1994.
14. B. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, Oct 1973.
15. A. J. Menezes and S. A. Vanstone. Elliptic curve cryptosystems and their implementations. *Journal of Cryptology*, 6(4):209–224, 1993.
16. National Institute of Standards and Technology. Security requirements for cryptographic modules. FIPS 140-2, NIST, Jun 2001. http://csrc.nist.gov/publications/fips/.
17. OpenSSL Project. http://www.openssl.org/.
18. E. Rescorla. ssldump version 0.9b2. http://www.rtfm.com/ssldump/.
19. M. Scott. MIRACL - Multiprecision Integer and Rational Arithmetic C/C++ Library v. 4.6. http://indigo.ie/∼mscott/.
20. secsh IETF Working Group. http://www.ietf.org/html.charters/secsh-charter.html.
21. G. J. Simmons. The prisoners' problem and the subliminal channel. In D. Chaum, editor, *Proceedings of Crypto 1983*, pages 51–67. Plenum Press, Aug 1983.
22. G. J. Simmons. The subliminal channel and digital signatures. In T. Beth, N. Cot, and I. Ingemarsson, editors, *Proceedings of Eurocrypt 1984*, volume 0209 of *LNCS*, pages 364–378. Springer-Verlag, Apr 1984.
23. A. Young and M. Yung. The dark side of "black-box" cryptography, or: Should we trust Capstone. In N. Koblitz, editor, *Proceedings of Crypto 1996*, volume 1109 of *LNCS*, pages 89–103. Springer-Verlag, Aug 1996.
24. A. Young and M. Yung. Kleptography: Using cryptography against cryptography. In W. Fumy, editor, *Proceedings of Eurocrypt 1997*, volume 1233 of *LNCS*, pages 62–74. Springer-Verlag, May 1997.