Analysis of the SSL 3.0 protocol

David Wagner
University of California, Berkeley
`daw@cs.berkeley.edu`

Bruce Schneier
Counterpane Systems
`schneier@counterpane.com`

Revised November 19, 1996

Abstract

The SSL protocol is intended to provide a practical, application-layer, widely applicable connection-oriented mechanism for Internet client/server communications security. This note gives a detailed technical analysis of the cryptographic strength of the SSL 3.0 protocol. A number of minor flaws in the protocol and several new active attacks on SSL are presented; however, these can be easily corrected without overhauling the basic structure of the protocol. We conclude that, while there are still a few technical wrinkles to iron out, on the whole SSL 3.0 is a valuable contribution towards practical communications security.

1 Introduction

The recent explosive growth of the Internet and the World Wide Web has brought with it a need to securely protect sensitive communications sent over this open network. The SSL 2.0 protocol has become a de facto standard for cryptographic protection of Web `http` traffic. But SSL 2.0 has several limitations—both in cryptographic security and in functionality—so the protocol has been upgraded, with significant enhancements, to SSL 3.0. This new version of SSL will soon see widespread deployment. The IETF Transport Layer Security working group is also using SSL 3.0 as a base for their standards efforts. In short, SSL 3.0 aims to provide Internet client/server applications with a practical, widely-applicable connection-oriented communications security mechanism.

This note analyzes the SSL 3.0 specification [FKK96], with a strong focus on its cryptographic security. We assume familiarity with the SSL 3.0 specification. Explanations of some of the cryptographic concepts can be found in [Sch96].

The paper is organized as follows. Section 2 briefly gives some background on SSL 3.0 and its predecessor SSL 2.0. Sections 3 and 4 explore several possible attacks on the SSL protocol and offer some technical discussion on the cryptographic protection afforded by SSL 3.0; this material is divided into two parts, with the SSL record layer analyzed in Section 3 and the SSL key-exchange protocol considered in Section 4. Finally, Section 5 concludes with a high-level view of the SSL protocol's strengths and weaknesses.

2 Background

SSL is divided into two layers, with each layer using services provided by a lower layer and providing functionality to higher layers. The SSL record layer provides confidentiality, authenticity, and replay protection over a connection-oriented reliable transport protocol such as TCP. Layered above the record layer is the SSL handshake protocol, a key-exchange protocol which initializes and synchronizes cryptographic state at the two endpoints. After the key-exchange protocol completes, sensitive application data can be sent via the SSL record layer.

SSL 2.0 had many security weaknesses which SSL 3.0 aims to fix. We briefly describe a short list of the flaws in SSL 2.0 which we have noticed. In export-weakened modes, SSL 2.0 unnecessarily weakens the authentication keys to 40 bits. SSL 2.0 uses a weak MAC construction, although post-encryption seems to stop attacks. SSL 2.0 feeds padding bytes into the MAC in block cipher modes, but leaves the padding-length field unauthenticated, which may potentially allow active attackers to delete bytes from the end of messages. There is a ciphersuite rollback attack, where an active attacker edits the list of ciphersuite preferences in the hello messages to invisibly force both endpoints to use a weaker form of encryption than they otherwise would choose;

this serious flaw limits SSL 2.0's strength to "least common denominator" security when active attacks are a threat. Others have also discovered some of these weaknesses: Dan Simon independently pointed out the ciphersuite rollback attack, Paul Kocher has addressed these concerns [Koc96], and the PCT 1.0 protocol [PCT95] discussed and countered some (though not all) of these flaws.

3 The record layer

This section considers the cryptographic strength of the record layer protocol, and assumes that the key-exchange protocol has securely set up session state, keys, and security parameters. Of course, a secure key-exchange protocol is vital to the security of application data, but an examination of attacks on the SSL key-exchange protocol is postponed until the next section.

The SSL record layer addresses fairly standard problems that have received much attention in the cryptographic and security literature [KV83], so it is reasonable to hope that SSL 3.0 provides fairly solid protection in this respect. As we shall see, this is not far from the truth. We consider confidentiality and integrity protection in turn.

3.1 Confidentiality: eavesdropping

The SSL protocol encrypts all application-layer data with a cipher and short-term session key negotiated by the handshake protocol. A wide variety of strong algorithms used in standard modes is available to suit local preferences; reasonable applications should be able to find an encryption algorithm meeting the required level of security, US export laws permitting. Key-management is handled well: short-term session keys are generated by hashing random per-connection salts and a strong shared secret. Independent keys are used for each direction of a connection as well as for each different instance of a connection. SSL will provide a lot of known plaintext to the eavesdropper, but there seems to be no better alternative; since the encryption algorithm is required to be strong against known-plaintext attacks anyway, this should not be problematic.

3.2 Confidentiality: traffic analysis

When the standard attacks fail, a cryptanalyst will turn to more obscure ones. Though often maligned, traffic analysis is another passive attack worth considering. Traffic analysis aims to recover confidential information about protection sessions by examining unencrypted packet fields and unprotected packet attributes. For example, by examining the unencrypted IP source and destination addresses (and even TCP ports), or examining the volume of network traffic flow, a traffic analyst can determine what parties are interacting, what type of services are in use, and even sometimes recover information about business or personal relationships. In practice, users typically consider the threat of this kind of coarse-grained tracking to be relatively harmless, so SSL does not attempt to stop this kind of traffic analysis. Ignoring coarse-grained traffic analysis seems like a reasonable design decision.

However, there are some more subtle threats posed by traffic analysis in the SSL architecture. Bennet Yee has noted that examination of ciphertext lengths can reveal information about URL requests in SSL- or SSL-encrypted Web traffic [Yee96]. When a Web browser connects to a Web server via an encrypted transport such as SSL, the GET request containing the URL is transmitted in encrypted form. Exactly which Web page was downloaded by the browser is clearly considered confidential information—and for good reason, as knowledge of the URL is often enough for an adversary to obtain the entire Web page downloaded—yet traffic analysis can recover the identity of the Web server, the length of the URL requested, and the length of the `html` data returned by the Web server. This leak could often allow an eavesdropper to discover what Web page was accessed. (Note that Web search engine technology is certainly advanced enough to catalogue the data openly available on a Web server and find all URLs of a given length on a given server which return a given amount of `html` data.)

This vulnerability is present because the ciphertext length reveals the plaintext length.[1] SSL includes support for random padding for the block cipher modes, but not for the stream cipher modes. We believe that SSL should at the minimum support the usage of random-length padding for all cipher modes, and should also strongly consider requiring

[1] This is strictly speaking only true of stream ciphers, but they are currently the common case. With block ciphers, plaintexts are padded out to the next 8-byte boundary, so one can only recover a close estimate of the plaintext length.

it for certain applications.

3.3 Confidentiality: active attacks

It is important that SSL securely protect confidential data even against active attacks. Of course, the underlying encryption algorithm should be secure against adaptive chosen-plaintext/chosen-ciphertext attacks, but this is not enough on its own. Recent research motivated by the IETF IPSEC (IP security) working group has revealed that sophisticated active attacks on a record layer can breach a system's confidentiality even when the underlying cipher is strong [Bel96]. It appears that the SSL 3.0 record layer resists these powerful attacks; it is worth discussing in some depth why they are foiled.

One important active attack on IPSEC is Bellovin's cut-and-paste attack [Bel96]. Recall that, to achieve confidentiality, link encryption is not enough—the receiving endpoint must also guard the sensitive data from inadvertent disclosure. The cut-and-paste attack exploits the principle that most endpoint applications will treat inbound encrypted data differently depending on the context, protecting it more assiduously when it appears in some forms than in others.[2] The cut-and-paste attack also takes advantage of a basic property of the cipher-block chaining mode: it recovers from errors within one block, so transplanting a few consecutive ciphertext blocks between locations within a ciphertext stream results in a corresponding transfer of plaintext blocks, except for a one-block error at the beginning of the splice. In more detail, Bellovin's cut-and-paste attack cuts an encrypted ciphertext from some packet containing sensitive data, and splices it into the ciphertext of another packet which is carefully chosen so that the receiving endpoint will be likely to inadvertently leak its plaintext after decryption. For example, if cut-and-paste attacks on the SSL record layer were feasible, they could be used to compromise site security: a cut-and-paste attack on a SSL server-to-client Web page transfer could splice ciphertext from a sensitive part of that `html` transfer into the hostname portion of a URL included elsewhere in the transferred Web page, so that when a user clicks on the booby-trapped URL link his browser would interpret the decryption of the spliced sensitive ciphertext as a hostname and send a DNS domain name lookup for it in the clear, ready for capture by the eavesdropping attacker. Cut-and-paste attacks, in short, enlist the unsuspecting receiver to decrypt and inadvertently leak sensitive data for them.

SSL 3.0 stops cut-and-paste attacks. One partial defense against cut-and-paste attacks is to use independent session keys for each different context. This prevents cutting and pasting between different connections, different directions of a connection, etc. SSL already uses independent keys for each direction of each incarnation of each connection. Still, cutting and pasting within one direction of a transfer is not prevented by this mechanism. The most comprehensive defense against cut-and-paste attacks is to use strong authentication on all encrypted packets to prevent enemy modification of the ciphertext data. The SSL record layer does employ this defense, so cut-and-paste attacks are completely foiled. For a more complete exposition on cut-and-paste attacks, see Bellovin's paper [Bel96].

The short-block attack is another active attack against IPSEC which can be found in Bellovin's paper [Bel96]. The short-block attack was originally applied against DES-CBC IPSEC-protected TCP data when the final message block contains a short one-byte plaintext and the remainder of it is filled by random padding. One guesses at the unknown plaintext byte by replacing the final ciphertext block with another ciphertext block from a known plaintext/ciphertext pair. Correct guesses can be recognized by the validity of the TCP checksum: an incorrect guess will cause the packet to be silently dropped by the receiver's TCP stack, but the correct guess will cause a recognizable ACK to be returned. Knowledge of the corresponding plaintext for a correctly guessed replacement ciphertext block enables the enemy to recover the unknown plaintext byte. Because the receiving IPSEC stack ignores the padding bytes, the short-block attack requires about 2^8 known plaintexts and 2^8 active online trials to recover such an unknown trailing byte. Many distracting technicalities have been significantly simplified; see Bellovin's paper [Bel96] for more details.

There are no obvious short-block attacks on SSL. The SSL record layer format is rather similar to the old vulnerable IPSEC layout, so it is admittedly conceivable that a modified version of the attack might work against SSL. In any case, standard SSL-encrypting Web servers probably would not be threatened by a short-block type of attack, since

[2] In the IPSEC world, encrypted data to TCP user ports is not protected by the operating system nearly as strongly as encrypted data to the system TCP login or telnet port. For a SSL-protected Web connection, the client browser will guard the path portion of a URL more carefully than the hostname portion, as the hostname portion may subsequently appear unencrypted in DNS queries and IP source addresses, whereas the path portion of a URL is encrypted via SSL.

they do not typically encrypt short blocks. (Note, however, that a SSL-encrypting `telnet` client should demand particularly robust protection against short-block attacks, as each keystroke is typically sent in its own one-byte-long packet.)

In summary, our analysis did not uncover any active attacks on the confidentiality protection of the SSL 3.0 record layer.

3.4 Message authentication

In addition to protecting the confidentiality of application data, SSL cryptographically authenticates sensitive communications. On the Internet, active attacks are getting easier to launch every day. We are aware of at least two commercially available software packages to implement active attacks such as IP spoofing and TCP session hijacking, and they even sport a user-friendly graphical interface. Moreover, the financial incentive for exploiting communications security vulnerabilities is growing rapidly. This calls for strong message authentication.

SSL protects the integrity of application data by using a cryptographic MAC. The SSL designers have chosen to use HMAC, a simple, fast hash-based construction with some strong theoretical evidence for its security [BCK96]. In an area where several initial ad-hoc proposals for MACs have been cryptanalyzed, these provable security results are very attractive. HMAC is rapidly becoming the gold standard of message authentication, and it is an excellent choice for SSL. Barring major unexpected cryptanalytic advances, it seems unlikely that HMAC will be broken in the near future.

We point out that SSL 3.0 uses an older obsolete version of the HMAC construction. SSL should move to the updated current HMAC format when convenient, for maximal security.

On the whole, SSL 3.0 looks very secure against straightforward exhaustive or cryptanalytic attacks on the MAC. SSL 2.0 had a serious design flaw in that it used an insecure MAC—though post-encryption saved this from being a direct vulnerability—but SSL 3.0 has fixed this mistake. The SSL MAC keys contain at least 128 bits of entropy, even in export-weakened modes, which should provide excellent security for both export-weakened and domestic-grade implementations. Independent keys are used for each direction of each connection and for each new incarnation of an connection. The choice of HMAC should stop cryptanalytic attacks. SSL does not provide non-repudiation services, and it seems reasonable to deliberately leave that to special higher-level application-layer protocols.

3.5 Replay attacks

The naive use of a MAC does not necessarily stop an adversary from replaying stale packets. Replay attacks are a legitimate concern, and as they are so easy to protect against, it would be irresponsible to fail to address these threats. SSL protects against replay attacks by including an implicit sequence number in the MACed data. This mechanism also protects against delayed, re-ordered, or deleted data. Sequence numbers are 64 bits long, so wrapping should not be a problem. Sequence numbers are maintained separately for each direction of each connection, and are refreshed upon each new key-exchange, so there are no obvious vulnerabilities.

3.6 The Horton principle

Let's recall the ultimate goal of message authentication. SSL provides message integrity protection just when the data passed up from the receiver's SSL record layer to the protected application exactly matches the data uttered by the sender's protected application to the sender's SSL record layer. This means, approximately, that it is not enough to apply a secure MAC to just application data as it is transmitted over the wire—one must also authenticate any context that the SSL mechanism depends upon to interpret inbound network data. For lack of a better name, let's call this "the Horton principle" (with apologies to Dr. Seuss) of semantic authentication: roughly speaking we want SSL to

> "authenticate what was meant, not what was said."

To phrase it another way,

> Eschew unauthenticated security-critical context.

This design principle is hardly original; Abadi and Needham [AN96] gave a version of it in the context of building secure protocols. The Horton principle is essentially a restatement of their Principle 1 in terms of requirements for record-layer message authentication.

Figure 1: Analysis of security-critical context

```
encrypted_fragment**
    ↓               [read_key*, read_IV (1)]
padded compressed fragment
    ↓                    [cipher_type* (2)]
SSLCompressed.fragment
    ↓                  [CompressionMethod*]
SSLPlaintext.fragment
    ↓                        [ContentType**,
    ↓                       ProtocolVersion,
    ↓                 SSLPlaintext.length**]
"meaning"
```

Notes:

- ∗ Denotes session state synchronized by the key-exchange protocol.
- ∗∗ Protected by the MAC.
- (1) `read_IV` is initially taken from the session state, then taken from the last ciphertext block of the previous `encrypted_fragment`.
- (2) For block ciphers, padding is removed from the end of the padded fragment.

SSL 2.0 suffered from at least one flaw along these lines: the SSL 2.0 MAC covered padding data but not the length of the padding, so an active attacker was free to manipulate the cleartext padding length field to compromise message integrity. An analysis checking SSL 2.0's compliance with the Horton principle would have uncovered this flaw. With this motivation, we undertake an informal analysis of SSL 3.0 following the guidelines of the Horton principle.

The SSL record layer depends on a lot of context to interpret, decrypt, decompress, de-multiplex, and dispatch data from the wire. It is instructive to follow the chain of this processing of inbound network data, catalogue all the security-critical context which this processing depends on, and check to ensure that the critical context has been authenticated. This ensures that we have applied the MAC properly to all security-relevant items and fulfilled the Horton principle. Because the `encrypted_fragment` field is authenticated by the MAC, we will assume that that field is trustworthy, and follow its transformation into application data ("meaning"). The right-justified bracketed items in Figure 1 identify security-critical context used in each step of processing.

Figure 1 indicates that SSL 3.0 follows the Horton principle fairly closely. One minor exception is that the integrity of the `ProtocolVersion` field is not protected. (We refer specifically to the `SSLCiphertext.ProtocolVersion` field in the record layer, not the `ClientHello.client_version` field from the handshake protocol; the latter is protected, but the former is not.) If the `ProtocolVersion` field is ever used by SSL, it should be authenticated; if not, it should not be present in the packet format. Also, it is worth mentioning that the final result of the inbound processing is a stream of bytes from the application data stream, and message boundaries are not preserved. Any application that relies on message boundaries—such as a UDP-based program—will have to impose a higher-layer message length protocol on top of SSL. On the whole, though, our "Horton principle"-inspired analysis revealed no major weaknesses, to SSL 3.0's credit.

3.7 Summary

In summary, the protection of application data by the SSL record layer is, on the whole, quite good. The preceding section indicated a few small areas of concern, but they should be considered minor and the exception to the rule.

4 The key-exchange protocol

This section considers the security of the SSL handshake protocol as well as other SSL meta-data transport. The design of a secure key-exchange protocol is a thorny endeavor. There is a significant amount of complexity involved, so the discovery of a few weaknesses should not prove surprising. The following analysis describes a number of shortcomings of the SSL meta-data protection mechanisms, mostly in areas that have seen recent changes. The SSL 3.0 key-exchange protocol appears to be a significant advance over SSL 2.0, but it still bears a few scars from growing pains.

4.1 Overview of the handshake flow

The SSL 3.0 handshake-protocol message flow involves the client and server negotiating a common ciphersuite acceptable to both parties, exchanging random nonces, and the client sending an encrypted `pre_master_secret`. Then each endpoint derives a `master_secret` from the `pre_master_secret` and verifies that their protocol runs match by authenticating all messages with the `master_secret`. Assuming that the check succeeds, both generate session keys from the `master_secret` and proceed to send cryptographically-protected application data. The SSL protocol also includes a more lightweight session resumption protocol which allows two parties who have already exchanged a `master_secret` to generate updated session keys and start a new connection with those parameters.

4.2 Ciphersuite rollback attacks

The SSL 2.0 key-exchange protocol contained a serious flaw: an active attacker could silently force a domestic user to use export-weakened encryption, even if both endpoints supported and preferred stronger-grade algorithms. This is known as a ciphersuite rollback attack, and it can be performed by editing the cleartext list of supported ciphersuites sent in **hello** messages. SSL 3.0 fixes this vulnerability by authenticating all the handshake protocol messages with the `master_secret`, so such enemy tampering can be determined at the end of the handshake and the session terminated if necessary.

We describe the SSL 3.0 mechanism for preventing modification of handshake protocol messages in more detail. There are several generic vulnerabilities in this part of the SSL handshake protocol, so some introduction is in order. All the initial handshake protocol messages are sent, unprotected, in the clear. Instead of modifying the parameters in use at the moment, the key-exchange protocol modifies a pending session state. After the negotiation is complete, each party sends a short **change cipher spec** message, which simply alerts the other to upgrade the status of the pending session state to current. The new session state is used starting with the next message, though the **change cipher spec** message is unprotected.[3] Immediately following the **change cipher spec** comes the **finished** message, which contains a MAC on all the handshake protocol messages keyed by the `master_secret`. (For peculiar non-security reasons, the **change cipher spec** and alert messages are not authenticated in the **finished** message.) The 48-byte `master_secret` is never disclosed; instead, session keys are generated from it. This ensures that even if the session keys are recovered, the `master_secret` will remain secret, so the handshake protocol messages will be securely authenticated. The **finished** message is itself protected with the newly established ciphersuite. Neither party is supposed to accept application data until it has received and verified a **finished** message from the other party.

[3] More precisely, it is protected with the old session state, which initially is set up to provide no protection. The discussion ignores the complicating case of a handshake protocol execution which changes cryptographic parameters on a connection that already has some protection in effect.

4.3 Dropping the change cipher spec message

One quirk of the SSL key-exchange protocol is that the **change cipher spec** message is not protected by the message authentication in the **finished** message. This can potentially allow the cryptanalyst to get a foot in the door. We recall the normal SSL message flow:

...
1. $C \to S$: [**change cipher spec**]
2. $C \to S$: [**finished:**] $\{a\}_k$
3. $S \to C$: [**change cipher spec**]
4. $S \to C$: [**finished:**] $\{a\}_k$
5. $C \to S$: $\{m\}_k$

...

where $\{\cdot\}_k$ represents the keyed cryptographic transforms used by the record layer, m denotes a plaintext message sent after the key-exchange is finished, and a represents the **finished** message's authentication code, which is obtained by computing a symmetric MAC on the previous handshake messages (excluding the **change cipher spec** message). Note that before the receipt of a **change cipher spec** message, the current ciphersuite offers no encryption or authentication and the pending ciphersuite includes the negotiated ciphersuite; upon receiving a **change cipher spec** message, implementations are supposed to copy the pending ciphersuite to the current ciphersuite and enable cryptographic protection in the record layer.

We describe an attack that takes advantage of the lack of protection for **change cipher spec** messages. We assume the special case where the negotiated ciphersuite includes only message authentication protection and no encryption. The active attacker intercepts and deletes the **change cipher spec** messages, so that the two endpoints never update their current ciphersuite; in particular, the two endpoints never enable message authentication or encryption in the record layer for incoming packets. Now the attacker allows the rest of the interaction to proceed, stripping off the record layer authentication fields from **finished** messages and session data. At this point there is no authentication protection for session data in effect, and the active attacker can modify the transmitted session data at will. The impact is that, when an authentication-only transform is negotiated, an active attacker can defeat the authentication protection on session data, transparently causing both parties to accept incoming session data without any cryptographic integrity protection. We summarize the attack flow:

...		
1.	$C \to M$:	[**change cipher spec**]
2.	$C \to M$:	[**finished:**] $\{a\}_k$
2′.	$M \to S$:	[**finished:**] a
3.	$S \to M$:	[**change cipher spec**]
4.	$S \to M$:	[**finished:**] $\{a\}_k$
4′.	$M \to C$:	[**finished:**] a
5.	$C \to M$:	$\{m\}_k$
5′.	$M \to S$:	m
...		

Remember, in this flow $\{m\}_k$ denotes the transmission of a message m along with a message authentication field keyed by k; given $\{m\}_k$ it is easy to strip off the MAC field and recover m, since no encryption is in use here. Note moreover that the attacker can easily replace the unprotected session data m in flow $5'$ by forged data of his choice.

It is worth pointing out what happens when the negotiated ciphersuite includes encryption. Then the client's **finished** message is sent encrypted, but the server expects to receive it unencrypted, so it does not suffice to strip off the MAC field—instead, the attacker must recover the encryption key k and decrypt $\{a\}_k$ to obtain a. Therefore the attack will be foiled when the negotiated ciphersuite includes strong encryption. In the intermediate case where weak encryption (such as a 40-bit exportable mode) is used, the attacker may be able to carry out this attack if it possible to perform an online exhaustive keysearch to recover the short encryption key.[4] In all fairness, real-time online exhaustive keysearch of a 40-bit cipher is currently out of reach for many adversaries, although advances in computation power may make it a more serious threat in the future.

The simplest fix is to require that a SSL implementation receive a **change cipher spec** message before accepting a **finished** message. (Indeed, there is some language in the specification which could be interpreted to mandate this restriction, although it is not entirely clear.) Some readers might complain that this requirement ought to be obvious with a moment's reflection, even if it is not explicitly stated in the SSL specification. We cannot fault such clarity of vision. However, we settle for the observation that at least one implementation has fallen for this pitfall. After performing the theoretical analysis, we examined Netscape's SSLRef 3.0b1 reference source code for SSL 3.0. Indeed, the necessary check is not made there; though we have not actually implemented the attack, it appears that SSLRef 3.0b1 will fall to a **change cipher spec** dropping attack when an authentication-only ciphersuite is negotiated.[5]

A more radical fix would include the **change cipher spec** message in the the **finished** message's message authentication calculation. This would require a change to the SSL specification; however, it also would have the advantage of being more robust in face of implementation flaws.

At the least, we recommend that future SSL documents include a warning about this pitfall. Explicitness is a virtue.

4.4 Key-exchange algorithm rollback

The SSL 3.0 handshake protocol also contains another design flaw. A server can send short-lived public key parameters, signed under its long-term certified signing key, in the **server key exchange** message. Several key-exchange algorithms are supported, including ephemeral RSA and Diffie-Hellman public keys. Unfortunately, the signature on the short-lived parameters does not protect the field which specifies which type of key-exchange algorithm is in use. Note that this violates the Horton principle: SSL should sign not just the public parameters but also all data needed to interpret those parameters.

For convenience, we reprint the relevant SSL 3.0

[4] A note about the amount of known plaintext available is in order. When a block cipher mode (such as 40-bit RC2 or 40-bit DES) is in use, there will be 4 bytes of known plaintext in the header of the **finished** message and another 4–8 bytes in the padding fields, so enough known text is available. For unpadded 40-bit stream cipher modes, there is only the 4 bytes of known plaintext in the **finished** message header; if the client immediately sends encrypted session data after sending the **finished** message (as is allowed in Section 7.6.9 of the SSL 3.0 specification) then enough additional known plaintext will probably be available to uniquely recover the stream cipher key; otherwise, about 2^8 possible 40-bit keys will be suggested, and the attacker must settle for a 2^{-8} chance of success.

[5] A SSLRef implementor has notified us that, after learning of this attack, he fixed the flaw; SSLRef 3.0b3 should contain the fix [Die96].

data structures from the the **server key exchange** message here.

```
enum { rsa, diffie_hellman, ... }
              KeyExchangeAlgorithm;
struct {
   opaque rsa_modulus;
   opaque rsa_exponent;
} ServerRSAParams;
struct {
   opaque dh_p;
   opaque dh_g;
   opaque dh_Ys;
} ServerDHParams;
struct {
   select (KeyExchangeAlgorithm) {
       case diffie_hellman:
             ServerDHParams params;
             Signature signed_params;
       case rsa:
             ServerRSAParams params;
             Signature signed_params;
   }
} ServerKeyExchange;
```

The `KeyExchangeAlgorithm` value is implicitly derived by each endpoint from the negotiated ciphersuite. The `signed_params` field contains the server's signature on a hash of the relevant `ServerParams` field (namely, either `ServerDHParams` or `ServerRSAParams` according to the value of the `KeyExchangeAlgorithm` variable), but the signature does *not* cover the `KeyExchangeAlgorithm` value. Therefore, by modifying each endpoint's view of the negotiated ciphersuite and thus affecting the (unsigned) `KeyExchangeAlgorithm` field, we can abuse the server's legitimate signature on a set of Diffie-Hellman parameters and fool the client into thinking the server signed a set of ephemeral RSA parameters.

We should point out that particularly cautious implementation might not be fooled by such tricks, if they check the length of the `ServerParams` field carefully. For example, SSLRef 3.0b1 is paranoid enough that it would detect such an attack. However, in general, the specification is silent on the matter, and some compliant implementations could easily be vulnerable.

If the implementation can be fooled, an active attack can be constructed. Perform a ciphersuite rollback attack to coerce the server into using the ephemeral Diffie-Hellman key exchange algorithm, while the client uses ephemeral RSA keying. With this change, the two endpoints will each think they have successfully negotiated a ciphersuite—but their ideas of the negotiated ciphersuite will differ. Unless implementors are exceptionally foresighted or paranoid, the server's Diffie-Hellman prime modulus p (`dh_p`) and generator g (`dh_g`) will probably be interpreted by the client as a correctly signed short-lived RSA modulus p (`rsa_modulus`) with exponent g (`rsa_exponent`). Watch as the client encrypts the `pre_master_secret` with the bogus RSA values. Intercept the RSA encrypted value $k^g \bmod p$; recover k, the PKCS encoding of the `pre_master_secret`, by taking g-th roots, which can be done efficiently since p is prime. Now that the `pre_master_secret` is compromised, it is easy to spoof the rest of the key exchange, including forging **finished** messages, to both endpoints. Thereafter one can decrypt all the sensitive application data transmitted or forge fake data on that SSL connection. All cryptographic protection has been wholly defeated.

We summarize this attack in the following attack flow (omitting many irrelevant fields and messages):

[client hello:]
1. $C \to M$: SSL_RSA_...
1′. $M \to S$: SSL_DHE_RSA_...

[server hello:]
2. $S \to M$: SSL_DHE_RSA_...
2′. $M \to C$: SSL_RSA_...

[server key exchange:]
3. $S \to M$: $\{p, g, y\}_{K_S}$
3′. $M \to C$: $\{p, g, y\}_{K_S}$

[client key exchange:]
4. $C \to M$: $k^g \bmod p$
4′. $M \to S$: $g^x \bmod p$

...

At the end of the key-exchange, the client's value of the `pre_master_secret` is k, while the server's value is $g^{xy} \bmod p$ where x was chosen by the attacker M; of course, both of these are known to the attacker M, and all secrets are derived from these values, so all subsequent cryptographic transforms offer no protection against M.

The key-exchange algorithm rollback attack serves to illustrate the dangers of a flexible ciphersuite negotiation algorithm. In the worst case it is possible to end up with "least common denominator security", where SSL is only as secure as the weakest key exchange algorithm (or weakest ciphersuite) supported.

4.5 Anonymous key-exchange

Our examination of SSL 3.0 revealed a minor typo in the specification for anonymous Diffie-Hellman key-exchange. As written, the document indicates that the server should sign an empty structure when anonymous key-exchange is in use. An earlier version of this paper included a critical analysis based on that erroneous interpretation. However, we have since been informed [Aba96, Die96] that our understanding of the specification was faulty: the SSL 3.0 designers intended that the signature be omitted entirely when the server was anonymous, and implementors have followed this route. To prevent confusion, we recommend that this small typo be fixed.

For clarity, we reprint the erroneous definition from the SSL 3.0 specification:

```
digitally-signed struct {
   select (SignatureAlgorithm) {
        case anonymous: struct { };
        case rsa:
             opaque md5_hash[16];
             ...
   }
} Signature;
```

4.6 Version rollback attacks

SSL 3.0 implementations will likely be flexible enough to accept SSL 2.0 connections, at least in the short-term. This threatens to create the potential for version rollback attacks, where an opponent modifies a SSL 3.0 **client hello** to look like a SSL 2.0 hello message and proceeds to exploit any of the numerous SSL 2.0 vulnerabilities.

Paul Kocher designed an intriguing strategy to detect version rollback attacks on SSL 3.0. Client implementations which support SSL 3.0 embed some fixed redundancy in the (normally random) RSA PKCS padding bytes to indicate that they support SSL 3.0. Servers which support SSL 3.0 will refuse to accept RSA-encrypted key-exchanges over SSL 2.0-compatibility connections if the RSA encryption includes those distinctive non-random padding bytes. This ensures that a client and server which both support SSL 3.0 will be able to detect version rollback attacks which try to coerce them into using SSL 2.0. Moreover, old SSL 2.0 clients will be using random PKCS padding, so they will still work with servers that support SSL 2.0.

Paul Kocher's clever countermeasure stops version rollback attacks, even in the face of active attacks. The central fact which makes it work is that RSA is the only key-exchange algorithm supported by SSL 2.0; if SSL 2.0 servers supported Diffie-Hellman key-exchange, the padding-redundancy trick would not be sufficient.

While Kocher's defense seems to stop version rollback attacks in normal circumstances, we remain somewhat concerned that it might interact adversely with session resumption. The specification does not forbid or discourage SSL 2.0-compatible SSL servers from accepting a SSL 2.0 **client hello** request to resume a session which was originally initiated with SSL 3.0 (or vice versa). This could potentially have subtle and obscure implications. Analysis appears non-trivial, and though we are not aware of any attacks, we are left with a distinct lack of confidence in our attempt at analysis. The issue here is protocol and implementation robustness, and we are concerned that this may represent a portion of SSL where robustness is below-average.

There is room for further examination of the potential interactions between session resumption and version rollback attacks. Lacking a comprehensive analysis, though, there is a natural defensive measure: servers supporting both SSL 2.0 and SSL 3.0 should not let clients mix SSL versions across session resumption. Implementations can easily achieve this by strictly segregating the SSL 2.0 and SSL 3.0 session caches. In any case, this will be irrelevant in the long term when servers stop accepting SSL 2.0 connections.

4.7 Safeguarding the `master_secret`

Ensuring that the `master_secret` remains truly secret is tremendously important to the security of SSL. All session keys are generated from the `master_secret`, and the protection against tampering with the SSL handshake protocol relies heavily on the secrecy of the `master_secret`. Therefore, it is important that the `master_secret` be especially heavily guarded. In protocol design, this means that usage of the `master_secret` should be greatly limited.

Figure 2 lists all of the places where the `master_secret` is used. Each item in the list can be used to recover a relation involving the `master_secret` and some known plaintext.

An enemy can collect unlimited amounts of known

Figure 2: master_secret usage

message	usage
certificate verify	
	hash = adhoc-MAC(master_secret, handshake_messages)
finished	
	hash = adhoc-MAC(master_secret, handshake_messages + sender)
change cipher spec	
	key_block = expand-keys(master_secret, ServerHello.random + ClientHello.random, ...)

plaintext for the master_secret-keyed MAC transformation found in the **finished** message. The informed adversary opens many simultaneous connections via **client hello** messages requesting the resumption of the targeted session. For each such connection, the server will pick a random nonce, calculate a MAC with the master_secret, and send it back encrypted in a **finished** message. The clever adversary should leave all those connections open without responding to the server's **finished** message: sending incorrect data on any of the connections will cause a fatal alert which makes the session unresumable. In this way, the opponent can collect great amounts of known plaintext hashed with the master_secret. If some cryptanalyst discovers an attack on adhoc-MAC() which uses much known plaintext to recover the secret key, the current SSL protocol could become unsafe. A strongly robust handshake protocol should probably limit the amount of known text that is available to a cryptanalyst.

The pre_master_secret is at least as important to protect, for compromise of it would also reveal the master_secret. One way that an attacker may acquire more known text hashed with a master_secret is to replay the original RSA-encrypted ciphertext which contained the pre_master_secret. The attacker will not be able to complete the SSL handshake protocol with this replayed RSA ciphertext, but it may be possible to get the server to send a **finished** message containing some known plaintext hashed with the **master_secret**. This will only be possible if the server is pipelined enough to send a **finished** message after receiving the **client key exchange** message but before receiving a client **finished** message.[6] This trick would be impossible if the client's and server's random nonces were bound more tightly to the pre_master_secret in the RSA key-exchange—perhaps a hash of the nonces should be included in the RSA encryption input.

4.8 Diffie-Hellman key-exchange

SSL 3.0 includes support for ephemerally-keyed Diffie-Hellman key-exchange. Since Diffie-Hellman is the only public key algorithm known which can efficiently provide perfect forward secrecy, this is an excellent addition to SSL. In a SSL 3.0 Diffie-Hellman key-exchange, the server specifies its Diffie-Hellman exponent as well as the prime modulus and generator. To avoid server-generated trapdoors, the client should be careful to check that the modulus and generator are from a fixed public list of safe values. The well-known man-in-the-middle attack is prevented in SSL 3.0 by requiring the server's Diffie-Hellman exponential to be authenticated. (Anonymous clients are not required to possess a certificate.) There is no support for higher-performance variants of Diffie-Hellman, such as smaller (160-bit) exponents or elliptic curve variants.

4.9 The alert protocol

SSL includes a small provision for sending event-driven **alert** messages. Many of these indicate fatal error conditions and instruct the recipient to immediately tear down the session. For instance, the close-notify **alert** message indicates that the sender is finished sending application data on the connection; since **alert** messages are normally authenticated, this prevents a truncation attack. As another example, reception of any packet with an incorrect MAC will result in a fatal **alert**.

4.10 MAC usage

The SSL 3.0 handshake protocol uses several ad-hoc MAC constructions to provide message integrity. The security of these MACs has not been thoroughly evaluated. We believe that SSL 3.0 should consistently use HMAC whenever a MAC is called for; ad-hoc MACs should be avoided.

[6] This behavior is apparently not prohibited by the SSL 3.0 specification, but David Brownell has indicated [Bro96] that any server exhibiting this behavior would probably not be interoperable with today's clients.

4.11 Summary

The SSL handshake protocol has several vulnerabilities and worrisome features, especially in areas which have seen recent revision. These are only troublesome when active attacks are a concern. Furthermore, these are not universal weaknesses: different implementations may or may not be vulnerable. A flaw in a protocol does not necessarily yield a vulnerable implementation. Nonetheless, if the specification does not explicitly warn of an attack (or prevent it directly), it seems reasonable to offer constructive criticism.

5 Conclusion

This security analysis has dedicated the greatest amount of time to shortcomings of the SSL 3.0 protocol, but that was purely for reasons of exposition. One would be hard-pressed to find any correlation between the amount of space required to explain a technical point and its importance or severity. Therefore, it is worth putting the previous sections in perspective, reviewing the big picture, and summarizing the security of SSL 3.0.

In general SSL 3.0 provides excellent security against eavesdropping and other passive attacks. Although export-weakened modes offer only minimal confidentiality protection, there is nothing SSL can do to improve that fact. The only change to SSL's protection against passive attacks worth recommending is support for padding to stop traffic analysis of GET request lengths.

This analysis has revealed a number of active attacks on the SSL 3.0 protocol (though some implementations may not be vulnerable). The most important new attacks are **change cipher spec**-dropping and `KeyExchangeAlgorithm`-spoofing. The SSL specification should be changed to warn of these new attacks. Fortunately, it is not hard to patch up the small flaws which allowed these attacks, and several possible fixes were listed.

The analysis has also revealed several ways in which the robustness of the SSL protocol can be improved. Many remarks were not inspired by direct vulnerabilities, but still are worth considering for future versions of SSL. Many of the pitfalls in SSL 3.0 were found in areas that have seen recent revision.

It is important not to overstate the practical significance of any of these flaws. Most of the weaknesses described in this note arise from a small oversight and can be corrected without overhauling the basic structure of the protocol. Of course, they are still worth fixing.

SSL 2.0 was subject to quite a number of active attacks on its record layer and key-exchange protocol. SSL 3.0 plugs those gaping holes and thus is considerably more secure against active attacks. SSL 3.0 also provides much better message integrity protection in export-weakened modes—the common case—than SSL 2.0 did: SSL 2.0 provided only 40-bit MACs in those modes, while SSL 3.0 always uses 128-bit MACs. Finally, SSL 3.0 improves a number of non-security aspects of SSL, such as flexible support for a wide variety of cryptographic algorithms. It seems fair to conclude that SSL 3.0 qualifies as a significant improvement over SSL 2.0.

In short, while there are still a few technical wrinkles to iron out, on the whole SSL 3.0 is a valuable step toward practical communications security for Internet applications.

6 Acknowledgements

We are deeply grateful to Martín Abadi, who pointed out a number of mistakes in an earlier version of this paper, for his corrections as well as several enlightening discussions. Paul Kocher kindly provided constructive criticism and many valuable comments; we are greatly indebted to him. Thanks also to Tim Dierks for additional corrections, and to David Brownell for his helpful implementation experience. Of course, any mistakes in this paper are solely our responsibility.

References

[Aba96] M. Abadi, personal communication, October 1996.

[AN96] M. Abadi and R. Needham, "Prudent Engineering Practice for Cryptographic Protocols", *IEEE Transactions on Software Engineering*, 22(1):2–15, January 1996.

[BCK96] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication," *Advances in Cryptology—CRYPTO '96 Proceedings*, Springer-Verlag, 1996, pp. 1–15.

[Bel96] S. Bellovin, "Problem Areas for the IP Security Protocols", *Proceedings of the Sixth USENIX Security Symposium*, Usenix Association, 1996, pp. 205–214. ftp://ftp.research.att.com/dist/smb/badesp.ps.

[Bro96] D. Brownell, personal communication, October 1996.

[Die96] T. Dierks, personal communication, November 1996.

[FKK96] A. Freier, P. Karlton, and P. Kocher, "The SSL Protocol Version 3.0", ftp://ftp.netscape.com/pub/review/ssl-spec.tar.Z, March 4 1996, Internet Draft, work in progress.

[Koc96] P. Kocher, personal communication, 1996.

[KV83] V. Voydock and S. Kent, "Security Mechanisms in High-Level Network Protocols", *ACM Computing Surveys*, v. 5, n. 2, June 1983, pp. 135–171.

[PCT95] J. Benaloh, B. Lampson, D. Simon, T. Spies, and B. Yee, "Microsoft Corporation's PCT Protocol", October 1995, Internet Draft, work in progress.

[RSA93] RSA Data Security, Inc., "Public-Key Cryptography Standards (PKCS)," Nov 93.

[Sch96] B. Schneier, *Applied Cryptography, 2nd Edition*, John Wiley & Sons, 1996.

[Yee96] B. Yee, personal communication, June 1996.