論文
2-10-4

안전한 웹 서비스를 위한 SSL/TLS 프로토콜 취약성 분석 (An Analysis of the Vulnerability of SSL/TLS for Secure Web Services)

조 한 진* 이 재 광**
(Han-Jin Cho) (Jae-Kwang Lee)

요 약

SSL은 인터넷의 표준 프로토콜인 TCP/IP 트래픽의 암호화를 위해, 기밀성과 인증, 그리고 데이터 무결성을 제공하는 프로토콜이다. 또한 SSL은 인터넷 클라이언트/서버 통신 보안을 위해, 여러 어플리케이션 계층에 적용할 수 있는 연결 중심형 메커니즘을 제공하기 위한 것을 목적으로 한다. 넷스케이프사에 의해 개발된 SSL은 현재 모든 클라이언트의 브라우저와 보안을 제공하는 서버에 의해 지원된다. 현재 SSL의 버전은 3.0이며, 1999년 1월 IETF의 TLS 작업 그룹이 TLS v1.0 명세서를 발표하였다. SSL은 여러 버전을 거치면서, 많은 취약성을 드러내었다. SSL과 TLS는 핸드쉐이크 프로토콜을 이용하여 암호학적 파라미터들을 교환하여 비밀키를 생성하는데, 이 과정에서 많은 공격이 취해질 수 있으며, 또한 레코드 프로토콜에서도 공격이 일어날 수 있다. 본 논문에서는 SSL 프로토콜을 분석하고, TLS와의 차이점을 비교하였으며, 현재 알려진 공격들을 자세히 분석하여, 개발자들이 구현 시 주의해야 할 것들을 제시하였다.

ABSTRACT

The Secure Sockets Layer is a protocol for encryption TCP/IP traffic that provides confidentiality, authentication and data integrity. Also the SSL is intended to provide the widely applicable connection-oriented mechanism which is applicable for various application-layer, for Internet client/server communication security. SSL, designed by Netscape is supported by all clients' browsers and server supporting security services. Now the version of SSL is 3.0. The first official TLS v1.0 specification was released by IETF Transport Layer Security working group in January 1999. As the version of SSL has had upgraded, a lot of vulnerabilities were revealed. SSL and TLS generate the private key with parameters exchange method in handshake protocol, a lot of attacks may be caused on this exchange mechanism, also the same thing may be come about in record protocol. In this paper, we analyze SSL protocol, compare the difference between TLS and SSL protocol, and suggest what developers should pay attention to implementation.

* 학생회원 : 한남대학교 대학원 컴퓨터공학과 박사과정
** 정회원 : 한남대학교 컴퓨터공학과 부교수

논문접수 : 2001. 9. 29.
심사완료 : 2001. 10. 17.

1. 서론

인터넷과 웹의 폭발적인 성장으로 사용자가 계속적으로 늘어나고 있지만, 개방형 네트워크 상에서 사용자들이 주고받는 메시지들에 대한 안전성은 보장되지 않는다. 그러므로 당사자들은 자신들의 메시지를 암호화하여 전송하고 해독하는 형태를 취하고 있다.

요즘, 인터넷을 이용하여 인터넷뱅킹을 이용하는 사용자가 현저히 증가하고 있다. 인터넷뱅킹에서 주고받는 메시지들에 기밀성을 제공하기 위해 대부분의 서버들이 SSL(Secure Sockets Layer)을 이용하고 있는 실정이다.

SSL은 클라이언트와 서버 사이의 통신에 보안서비스를 제공하기 위해, 어플리케이션 계층에 적용할 수 있는 연결-중심형 서비스를 제공하는 것을 목적으로 한다. 넷스케이프사에 의해 개발된 SSL은 현재 모든 브라우저들에 의해 지원되고 있으며, 여러 버전을 거치면서 많은 취약성을 드러내고 있다[2][3].

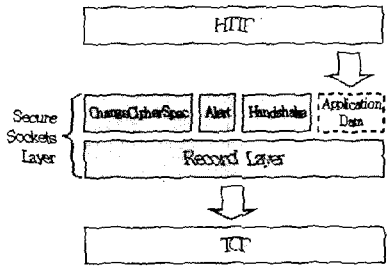
표준화하기 위한 작업으로는, 1996년 IETF의 TLS(Transport Layer Security) WG가 구성되어, 1999년 버전 1.0을 발표하였다. SSL과 TLS는 인터넷상의 클라이언트와 서버 사이의 통신에 기밀성, 무결성, 인증 서비스를 제공한다[2].

본 논문에서는 SSL 프로토콜을 암호학적인 면에서 자세히 분석하고, TLS와 비교하여 차이점을 살펴본다. 또한 SSL/TLS를 개발하려는 개발자들이 구현시 고려해야 하는 취약성들을 분석하여 해결책을 제시하고자 한다.

2. SSL 프로토콜

2.1 SSL 구조

SSL은 [그림 1]과 같이 ChangeCipherSpec 프로토콜(20), Alert 프로토콜(21), Handshake 프로토콜(22), 어플리케이션 데이터 프로토콜(23)과 레코드 계층(Record Layer) 프로토콜로 구성된다[1][4].



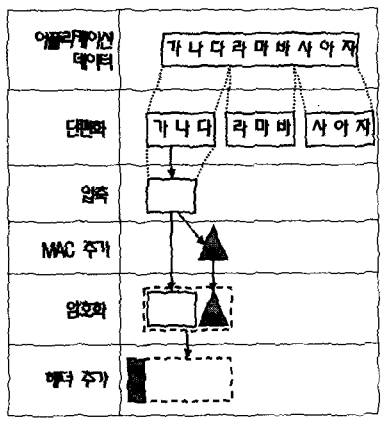
[그림 1] SSL 프로토콜 구조도

[Fig. 1] SSL Protocol Architecture

2.2 레코드 계층 프로토콜

레코드 계층 프로토콜의 역할은 메시지를 단편화하여 압축한 다음, MAC을 추가하여 암호화한다. 암호화된 부분은 헤더를 추가하여 사용한다[그림 2].

레코드 계층의 형식은 헤더, 다른 프로토콜 메시지들, MAC 순으로 구성된다. 만약 서비스가 진행중이라면, 암호화의 의무도 가지고 있다[5].



[그림 2] SSL 레코드 프로토콜

[Fig. 2] SSL Record Protocol

2.3 ChangeCipherSpec 프로토콜

```
struct {
    enum { change_cipher_spec(1), (255) } type;
  } ChangeCipherSpec;
```

이 프로토콜은 ChangeCipherSpec 메시지만을 갖는 간단한 프로토콜이며, 보안 서비스를 호출한다.

2.4 Alert 프로토콜

시스템은 상대 당사자에게 에러나 경고를 알리기 위해 이 프로토콜을 사용한다. Alert 프로토콜은 두 개의 필드를 정의하고 있다.

2.4.1 보안 등급(Security Level)

```
enum { warning(1), fatal(2), (255) } AlertLevel;
```

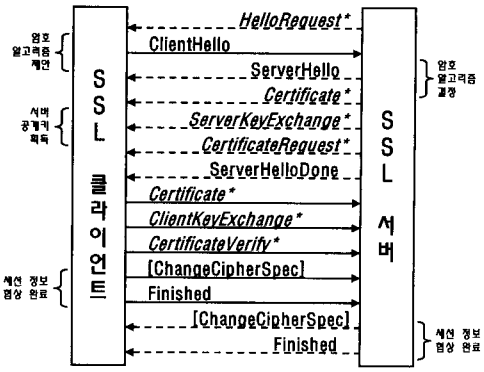
보안 등급 필드는 상태의 정도를 나타낸다. 이 필드는 warning이나 fatal 중 하나이다. fatal 경고는 통신에서 심각한 문제를 뜻하며, 두 당사자가 즉시 세션을 중단하도록 요구한다. warning 경고는 중요하지는 않다. 경고를 받고 있는 시스템은 현재의 세션이 계속되는 것을 허락 할 수 있으나, 차후에 세션을 재시작해서는 안 된다.

2.4.2 경고 설명 (Warning Description)

경고 설명 필드는 좀더 세부적으로 특정한 에러를 기술한다. 이 필드의 크기는 1 바이트이다.

2.5 Handshake 프로토콜

SSL 프로토콜은 클라이언트와 서버가 핸드쉐이크 메시지를 교환함으로써 암호화된 세션을 수립하며, 여러 핸드쉐이크 메시지들은 단일 레코드 계층 메시지로 결합될 수 있다. [그림 3]은 SSL 핸드쉐이크 프로토콜 단계를 나타내고 있으며, "*" 은 옵션이다[4].



[그림 3] SSL 핸드쉐이크 프로토콜

[Fig. 3] SSL Handshake Protocol

2.5.1 HelloRequest 메시지

```
struct { } HelloRequest;
```

HelloRequest는 서버가 클라이언트에게 SSL 핸드쉐이크 협상을 재시작하도록 요청하는 것을 허용한다. 이 메시지는 자주 사용되지는 않는다.

2.5.2 ClientHello 메시지

```
struct {
      uint32 gmt_unix_time;
      opaque random_bytes[28];
} Random;
opaque SessionID<0..32>;
uint8 CipherSuite[2];
enum { null(0), (255) } CompressionMethod;
struct {
      ProtocolVersion client_version;
      Random random;
      SessionID session_id;
      CipherSuite cipher_suites<2..2^16-1>;
      CompressionMethod
      compression_methods<1..2^8-1>;
} ClientHello;
```

ClientHello 메시지는 일반적으로 SSL 핸드쉐이크의 협상을 시작한다. ClientHello 메시지들의 핸드쉐이크 메시지 유형은 1이고, 다양한 메시지 크기를 갖는다.

버전 필드는 2 바이트, 랜덤 필드는 32 바이트가 사용된다. SSL 명세서는 현재 날짜와 시간을 랜덤 값의 처음 4 바이트로 사용한다. 다음 필드는 세션 ID의 길이와 세션 ID가 위치한다. 만약 클라이언트가 이전 세션의 재시작을 원하지 않는다면, 세션 ID를 제거한다. 클라이언트가 제안한 암호 조합의 목록들이 온다. 목록의 크기를 나타내는 1 바이트와 암호 조합의 목록들이 나열된다. 마지막 필드는 클라이언트가 세션에 제안한 압축 방법들을 나열하고 있다.

2.5.3 ServerHello 메시지

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
} ServerHello;
```

ServerHello 메시지는 ClientHello 메시지와 비슷하다. 단지 차이점은 서버가 Handshake 메시지 유형 값(1 대신에 2)과 목록이 아닌 하나의 암호 조합과 압축 방법만을 지정한다는 사실이다. 이처럼 서버는 클라이언트가 제안한 것 중에서 하나를 선택해야만 한다.

2.5.4 Certificate 메시지

```
opaque ASN.1Cert<1..2^24-1>;
struct {
    ASN.1Cert certificate_list<1..2^24-1>;
} Certificate;
certificate_list
        sequence of X.509v3 certificate
        sender  s cert must come first
        the sequence order must track
        validation chain
```

Certificate 메시지 유형은 11이고, 메시지 유형과 표준 Handshake 메시지 길이로 시작된다. 메시지는 공개키 인증서 체인을 포함한다.

인증서 체인은 SSL이 인증서 계층구조를 지원하도록 허용한다. 체인 내의 첫 번째 인증서는 항상 송신자의 것이다. 그 다음의 인증서는 송신자의 인증서를 발행한 기관의 인증서이다. 만약 세 번째 인증서가 존재하면, 그 기관에 대한 CA에 속한다. 체인은 루트 CA의 인증서에 도달할 때까지 계속된다.

2.5.5 ServerKeyExchange 메시지

```
enum { rsa, diffie_hellman, fortezza_kea }
        KeyExchangeAlgorithm;
struct {
    opaque rsa_modulus<1..2^16-1>;
    opaque rsa_exponent<1..2^16-1>;
} ServerRSAParams;
struct {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHParams;
struct {
    opaque r_s [128];
} ServerFortezzaParams;
struct {
    select (KeyExchangeAlgorithm) {
        case diffie_hellman:
            ServerDHParams params;
            Signature signed_params;
        case rsa:
            ServerRSAParams params;
            Signature signed_params;
        case fortezza_kea:
            ServerFortezzaParams params;
    };
} ServerKeyExchange;
```

ServerKeyExchange 메시지는 서버에서 클라이언트로 키 정보를 운반한다. 이 메시지의 형식은 키 정보 교환에 사용되는 암호 알고리즘에 의존한다.

Diffie-Hellman, RSA, Fortezza의 다양한 키 교환 알고리즘이 사용된다.

ServerKeyExchange 메시지를 해석하기 위해서, 클라이언트는 반드시 이전의 ServerHello 메시지 내에 있던 정보를 사용해야 한다.

Diffie-Hellman 키 교환 메시지는 파라미터(p, q, Y^S)에 대한 길이와 실제 값을 포함한다.

RSA 키 교환 메시지에서는 키의 정보가 RSA 계수와 공개 지수로 구성된다.

Fortezza/DMS 키 교환 메시지는 Fortezza의 rs 값을 운반한다. rs는 항상 128 바이트이기 때문에, ServerKeyExchange 메시지 내에 분리된 길이 파라미터가 필요 없다.

만약, 서버가 Certificate 메시지를 보낸다면, 서명된 파라미터의 형식은 서버의 인증서 내에 표시된 서명 알고리즘에 의존한다.

2.5.6 CertificateRequest 메시지

```
enum {
      rsa_sign(1), dss_sign(2), rsa_fixed_dh(3),
      dss_fixed_dh(4), rsa_ephemeral_dh(5),
      dss_ephemeral_dh(6), fortezza_kea(20),
      (255)
} ClientCertificateType;
          /* 클라이언트 인증서 서명 유형 */
opaque DistinguishedName<1..2^16-1>;
struct {
    ClientCertificateType
    certificate_types<1..2^8-1>;
    DistinguishedName
    certificate_authorities<3..2^16-1>;
    // acceptable CA list
} CertificateRequest;
```

서버는 클라이언트의 신원을 인증하기 위해 CertificateRequest 메시지를 보낸다. 이 메시지는 클라이언트의 인증서를 보내달라고 요청할 뿐만 아니라, 인증서가 서버에게 받아들일 만한지를 알려준다.

CertificateRequest 메시지는 서버가 적절하다고 생각하는 인증기관들을 가리킨다. 이 목록은 자신의 2 바이트 길이의 필드로 시작하고, 하나 이상의 DN (Distinguished Name)을 포함한다.

2.5.7 ServerHelloDone 메시지

```
struct { } ServerHelloDone;
```

ServerHelloDone 메시지는 서버의 Handshake 협상을 종료한다. 이 메시지는 아무런 정보도 전달하지 않는다.

2.5.8 ClientKeyExchange 메시지

```
struct {
      select (KeyExchangeAlgorithm) {
          case rsa: EncryptedPreMasterSecret;
          case diffie_hellman:
          ClientDiffieHellmanPublic;
          case fortezza_kea: FortezzaKeys;
      } exchange_keys;
} ClientKeyExchange;
struct {
      ProtocolVersion client_version;
      opaque random[46];
} PreMasterSecret;
struct {
     public-key-encrypted PreMasterSecret
     pre_master_secret;
} EncryptedPreMasterSecret;
```

클라이언트는 ClientKeyExchange 메시지를 이용하여 서버에게 기밀성을 제공하기 위한 키 재료들을 제공한다. 이 메시지의 형식은 양 당사자들이 사용하는 키 교환 알고리즘에 의존한다.

2.5.9 CertificateVerify 메시지

```
struct {
    Signature signature;
} CertificateVerify;
enum { anonymous, rsa, dsa }
SignatureAlgorithm;
digitally-signed struct {
    select(SignatureAlgorithm) {
        case anonymous: struct { };
        case rsa:
            opaque md5_hash[16];
            opaque sha_hash[20];
        case dsa:
            opaque sha_hash[20];
    };
} Signature;
```

클라이언트는 CertificateVerify 메시지를 이용하여 자신의 공개키 인증서에 대응하는 개인키가 있는지를 검사한다. 이 메시지는 클라이언트에 의해 서명된 해쉬를 포함한다. 형식은 클라이언트의 인증서가 RSA 서명인지 DSA 서명인지에 따라 다르다.

RSA 인증서에 대해서, 두 해쉬가 서명되어 연결된다(MD5와 SHA). DSA 인증서에 대해서는 SHA 해쉬만 생성되어 서명된다.

2.5.10 Finished 메시지

```
enum { client(0x434C4E54), //    클라이언트
       server(0x53525652) //    서버
} Sender;
struct {
    opaque md5_hash[16];
    opaque sha_hash[20];
} Finished;
md5_hash =
MD5(master_secret + pad2 +
MD5(handshake_messages + Sender +
master_secret + pad1));
sha_hash =
SHA(master_secret + pad2 +
SHA(handshake_messages + Sender +
master_secret + pad1));
```

이 메시지는 SSL 협상이 끝났으며 협상된 암호 조합이 유효하다는 것을 나타낸다. Finished 메시지 자체는 암호 조합 파라미터들을 이용해서 자체적으로 암호화된다.

Finished 메시지는 두 해쉬(MD5와 SHA) 결과 값으로 구성된다. 두 해쉬 계산은 동일한 입력 값을 사용한다.

CertificateVerify 메시지에 대한 해쉬 계산과 이 계산이 유사하지만, 차이점이 있다. 첫째, Finished 메시지의 해쉬는 송신자의 역할을 포함하는 반면에, CertificateVerify 메시지는 포함하지 않는다. 둘째, 두 개의 해쉬가 계산될 때, Handshake 메시지의 설정이 달라진다. ChangeCipherSpec 메시지를 Handshake 프로토콜 메시지로 간주하지 않으므로, 해쉬 값 계산에 포함되지 않는다.

3. SSL 메시지 보안

SSL 프로토콜은 어플리케이션 데이터에 MAC (Message Authentication Code)을 제공하여 무결성을 보장하고, 데이터에 암호를 제공하여 기밀성을 보장한다. 본 장에서는 MAC과 암호 및 암호학적 파라미터 생성에 대해 자세히 살펴본다.

3.1 메시지 인증 코드(MAC)

SSL은 MAC에 대해 MD5와 SHA 알고리즘을 제공한다. 통신에 대해 특정 알고리즘은 협상된 암호 조합에 의해 결정된다. MD5는 16 바이트 해쉬 값을 생성하고, SHA는 20 바이트 해쉬 값을 생성하며, Application 데이터에 추가된다[5].

```
hash(MAC_write_secret + pad_2 +
    hash(MAC_write_secret + pad_1 + seq_num +
        SSLCompressed.type +
        SSLCompressed.length +
        SSLCompressed.fragment));
```

pad_1 0x36 repeated 48 times for MD5 or 40 times for SHA.
pad_2 0x5c repeated 48 times for MD5 or 40 times for SHA.
seq_num sequence number(8 bytes; connection state)
hash Hashing algorithm derived from the cipher suite.

MAC을 계산하기 위해서, 시스템은 Handshake 메시지 내의 해쉬 계산과 매우 유사한 두 단계를 거친다. 1 단계에서, MAC_write_secret이라는 특정 값으로 시작하여, 패딩 1, 64비트 순서번호, 프로토콜 유형, 항목의 길이(16 비트 값), 마지막으로 항목 자체가 온다. 패딩 1은 1 바이트 값인 0x36(00110110)이고, MD5에 대해서는 48회 반복되고, SHA에 대해서는 40회 반복된다. 2 단계에서, 시스템은 MAC_write_secret, 패딩 2, 1 단계의 해쉬의 결과를 사용한다. 패딩 2는 1 바이트 0x5c(01011100)이고, MD5에 대해서는 48회 반복되고, SHA에 대해서는 40회 반복된다. 이 결과가 SSL 메시지에 추가되는 MAC 값이다.

3.2 암호화

SSL 프로토콜은 스트림 암호와 블록 암호를 모두 제공한다. 스트림 암호 알고리즘을 이용하면, 다른 파라미터들이 필요 없다. 그러나, 블록 암호에서는 암호화되는 데이터는 반드시 여러 개의 블록으로 나누어져야 한다. 그리고, 어플리케이션 데이터는 패딩을 사용한다.

길이를 블록 크기의 배수로 맞추기 위해 어플리케이션 데이터에 더미(dummy) 데이터를 추가한다. 수신자는 실제 어플리케이션 데이터를 추출하기 위해, 어플리케이션 데이터의 끝과 패딩의 시작이 어디인지 알아야 한다. 암호화된 정보의 마지막 바이트는 패딩의 길이를 갖는다. 블록을 복호화한 후에, 수신자는 어플리케이션 데이터의 끝을 찾기 위해 패딩 길이로부터 거꾸로 계산한다.

3.3 암호학적 파라미터 생성

SSL의 암호와 MAC 알고리즘은 양 당사자들이 알고 있는 비밀 정보에 의존한다.

첫 비밀 정보는 마스터 비밀(master_secret)이며, 마스터 비밀은 예비마스터 비밀(pre_master_secret)에 기초한다. 일반적으로 클라이언트는 안전한 랜덤 값을 생성함으로써 예비마스터 비밀을 선택한다. 클라이언트는 서버의 공개키를 사용해서 이 값을 암호화하고, 이것을 ClientKeyExchange 메시지 내에 넣어서 서버에게 보낸다.

서버가 ClientKeyExchange 메시지를 수신하면, 양 당사자들은 동일한 예비마스터 비밀을 알게 된다. 양 당사자들은 자신의 Hello 메시지에 대해 선택된 랜덤 값을 해쉬 함수에 입력한다. 여러 해쉬 출력 값을 연결한 후, 두 시스템은 동일한 마스터 비밀을 갖게 된다. 아래는 이 과정을 수식으로 표현한 것이다.

```
master_secret =
  MD5(pre_master_secret + SHA('A'
  + pre_master_secret +
       ClientHello.random
       + ServerHello.random)) +
  MD5(pre_master_secret + SHA('BB'
  + pre_master_secret + ClientHello.random +
  ServerHello.random)) + MD5(pre_master_secret +
  SHA('CCC' + pre_master_secret +
   ClientHello.random + ServerHello.random));
```

각 시스템이 마스터 비밀을 계산하고 나면, 필요한 비밀 정보를 생성할 준비가 된 것이다. 그 과정의 첫 번째 단계는 얼마나 많은 비밀 정보가 필요한지를 결정하는 것이다. 정확한 크기는 양 당사자들이 협상하는 특정 암호 조합과 파라미터들에 의존한다.

공유 비밀 정보를 생성하기 위해, 양 당사자는 마스터 비밀을 만드는 과정과 매우 유사한 과정을 사용한다. 먼저 ASCII 문자 'A', 서버의 랜덤 값, 그리고 클라이언트의 랜덤 값의 SHA 해쉬를 계산한다. 시스템은 마스터 비밀과 해쉬의 중간 결과를 합하여 MD5 해쉬를 계산한다. 만약 그 결과가 부족하다면, ASCII 문자 'A' 대신 'BB'로 그 과정을 반복한다.

당사자들은 이 계산을 이용하여 ('CCC', 'DDDD', 'EEEEE' 등) 비밀 정보가 충분할 때까지 이 과정을 여러 번 반복한다.

```
key_block =
    MD5(master_secret + SHA('A' +
    master_secret +
        ServerHello.random +
        ClientHello.random)) +
    MD5(master_secret + SHA('BB' +
    master_secret +
        ServerHello.random +
        ClientHello.random)) +
    MD5(master_secret + SHA('CCC' +
    master_secret +
        ServerHello.random +
        ClientHello.random)) + [...];
```

3.4 암호 조합

SSL 버전 3.0 명세서는 암호학적 알고리즘과 파라미터의 다양한 선택을 대표하는, 31개의 암호 조합을 정의하고 있다.

3.4.1 키 교환 알고리즘

SSL 명세서는 총 14개의 키 교환 알고리즘을 정의하고, 사용 가능한 변수를 계산한다. <표 1>는 이러한 알고리즘을 기술한다. 수출 가능한 암호 조합의 부분인 이러한 키 교환 알고리즘을 위해 이 표 또한 U.S 수출 정책이 정의하는 알고리즘의 크기 제한을 나타낸다.

3.4.2 암호 알고리즘

SSL 프로토콜은 9개의 암호화 알고리즘을 지원하고, 변수를 계산한다. <표 2>은 키 재료 크기, 효율적인 키 크기, 초기화 벡터 크기를 나타내고 있다.

<표 1> 키 교환 알고리즘

<Table 1> Key-Exchange Algorithms

알고리즘	설 명	키 크기 제한
DHE_DSS	DSS 서명을 이용한 일회용 DH	없음
DHE_DSS_EXPORT	DSS 서명을 이용한 일회용 DH	DH: 512 비트
DHE_RSA	RSA 서명을 이용한 일회용 DH	없음
DHE_RSA_EXPORT	RSA 서명을 이용한 일회용 DH	DH: 512 비트 RSA: 없음
DH_anon	익명의 DH	없음
DH_anon_EXPORT	익명의 DH	DH: 512 비트
DH_DSS	DSS 인증서를 이용한 DH	없음
DH_DSS_EXPORT	DSS 인증서를 이용한 DH	DH: 512 비트
DH_RSA	RSA 인증서를 이용한 DH	없음
DH_RSA_EXPORT	RSA 인증서를 이용한 DH	DH: 512 비트 RSA: 없음
FORTEZZA_DMS	FORTEZZA/DMS	
NULL	키 교환 없음	
RSA	RSA 키 교환	없음
RSA_EXPORT	RSA 키 교환	RSA: 512 비트

<표 2> 암호 알고리즘

<Table 2> Encryption Algorithms

알고리즘	유형	키 재료	키 크기	IV 크기
3DES_EDE_CBC	블록	24 바이트	168 비트	8 바이트
DES_CBC	블록	8 바이트	56 비트	8 바이트
DES40_CBC	블록	5 바이트	40 비트	8 바이트
FORTEZZA_CBC	블록		96 비트	20 바이트
IDEA_CBC	블록	16 바이트	128 비트	8 바이트
NULL	스트림	0 바이트	0 비트	
RC2_CBC_40	블록	5 바이트	40 비트	8 바이트
RC4_128	스트림	16 바이트	128 비트	
RC4_40	스트림	5 바이트	40 비트	

3.4.3 해쉬 함수

SSL 암호 조합의 마지막 요소는 메시지 인증 코드에 이용된 해쉬 알고리즘이다. <표 3>는 SSL이 정의한 세 개의 해쉬 알고리즘을 보여주며, MAC 그 자체를 포함해, 몇몇 SSL 계산에 이용된 패딩 크기를 보여준다.

<표 3> 해쉬 알고리즘

<Table 3> Hash Algorithms

알고리즘	해쉬 크기	패딩 크기
MD5	16 바이트	48 바이트
NULL	0 바이트	0 바이트
SHA	20 바이트	40 바이트

4. TLS 프로토콜

SSL은 넷스케이프에 의해 처음으로 개발되었지만, IETF(Internet Engineering Task Force)가 표준화를 시작하였다. IETF는 IPSEC(IP Security) 프로토콜과 구별하기 위해, SSL을 TLS(Transport Layer Security)라는 이름으로 바꾸었다. <표 4>는 SSL과 TLS의 차이점을 나타내고 있다[2][4].

<표 4> SSL과 TLS의 차이점

<Table 4> Differences between SSL and TLS

항목	SSL v3.0	TLS v1.0
메시지 내의 프로토콜 버전	3.0	3.1
경고 프로토콜 메시지 종류	12개	24개
메시지 인증	임시	표준
키 재료 생성	임시	PRF
CertificateVerify	복잡	간단
Finished	임시	PRF
Fortezza 암호 조합	포함	포함하지 않음

4.1 TLS 프로토콜 버전

현재 TLS 표준은 버전 1.0이다. 실제로, TLS의 첫 번째 버전이다. 그러나, SSL 버전 3.0 시스템과 상호동작을 유지하기 위해, 실제 프로토콜 메시지에 보고된 프로토콜 버전은 3.0 보다 더 높아야 한다. TLS 설계자들은 TLS 메시지에 나타나는 프로토콜 버전을 3.1로 명시하였다.

4.2 경고 프로토콜 메시지 종류

TLS가 SSL을 개선한 부분 중 하나는 실제적인 보안 경고의 통지에 대한 처리이다. TLS에서 새로 만들어진 것은 "•" 표시를 하였고, 경고 설명 41(NoCertificate)은 TLS에서 삭제되었다. TLS 명세서에서는 이 경고가 제거됐는데, 실제로 구현하기 어렵기 때문이다. <표 5>는 TLS 경고 프로토콜에 대한 메시지들을 나열하고 있다.

<표 5> TLS 경고 설명

<Table 5> TLS Alert descriptions

값		이름
	0	CloseNotify
	10	UnexpectedMessage
	20	BadRecordMAC
•	21	DecryptionFailed
•	22	RecordOverflow
	30	DecompressionFailure
	40	HandshakeFailure
삭제됨	41	NoCertificate
	42	BadCertificate
	43	UnsupportedCertificate
	44	CertificateRevoked
	45	CertificateExpired
	46	CertificateUnknown
	47	IllegalParameter
•	48	UnknowCA
•	49	AccessDanied
•	50	DecodeError
•	51	DecryptError
•	60	ExportRestriction
•	70	ProtocolVersion
•	71	InsufficientSecurity
•	80	InternalError
•	90	UserCanceled
•	100	NoRenegotiation

4.3 메시지 인증

SSL을 개선한 TLS의 또 다른 점은, 메시지 인증에 대한 알고리즘이다. 키 정보와 어플리케이션 데이터를 결합하는 SSL 메시지 인증 방법은 SSL 프로토콜에 의해 일시적으로 생성된다.

그러나 TLS 프로토콜은, H-MAC(Hashed Message Authentication Code)으로 알려진 표준 메시지 인증 코드를 사용한다. H-MAC은 특정 해쉬 알고리즘(MD5 또는 SHA)을 지정하지 않는다.

```
HMAC_hash(MAC_write_secret, seq_num +
        TLSCompressed.type
        + TLSCompressed.version +
        TLSCompressed.length
        + TLSCompressed.fragment));
        HMAC(K, M) = H(K xor opad
        + H(K xor ipad + M))
ipad a string consisting of the byte 0x36
opad a string consisting of the byte 0x5c
K   64 bytes (if shorter than 64, pad to the
    right with 0)
```

4.4 키 재료 생성

TLS는 의사랜덤 출력을 생성하기 위해 H-MAC을 이용하는 프로시저를 정의하고 있다. 이 프로시저는 비밀 값과 초기 시드 값을 취해서, 랜덤 출력을 안전하게 생성한다. 이 프로시저는 필요한 만큼의 랜덤 출력을 생성할 수 있다.

H-MAC 표준을 사용함으로써, 프로시저는 특정 해쉬 알고리즘에 의존하지 않아도 된다. MD5와 SHA를 포함한 어떠한 해쉬 알고리즘이라도 의사랜덤 출력에 이용될 수 있다.

```
PRF(secret, label, seed) = P_MD5(S1, label +
seed) XOR
                        P_SHA-1(S2, label
                        + seed);
        S1 is the first half of the secret
        S2 is the second half of the secret

P_hash(secret, seed) = HMAC_hash(secret, A(1) +
                        seed) +
                        HMAC_hash(secret, A(2) +
                        seed) +
                        HMAC_hash(secret, A(3) +
                        seed) + ...
A() is defined as:
        A(0) = seed
        A(i) = HMAC_hash(secret, A(i-1))
```

TLS는 PRF를 생성하기 위해 의사난수 출력 프로시저를 이용한다. PRF는 의사난수 출력 프로시저의 두 인스턴스를 결합한다. 하나는 MD5 해쉬 알고리즘을 이용하고, 다른 하나는 SHA 해쉬 알고리즘을 이용한다.

PRF는 비밀 값, 시드 값, 그리고 레이블로 시작한다. 함수는 비밀을 두 부분으로 분리한다. 하나는 MD5 해쉬를 위한 것이고, 다른 하나는 SHA 해쉬를 위한 것이다. 또한 레이블과 시드를 하나의 값으로 결합한다.

```
key_block =
        PRF(SecurityParameters.master_secret,
                "key expansion",
        SecurityParameters.server_random +
        SecurityParameters.client_random);
```

각 시스템은 예비마스터 비밀로 시작한다. 다음에 마스터 비밀을 생성하고, 그 다음에 마스터 비밀로부터 요청된 키 재료를 생성한다. 키 재료를 생성하기 위해, TLS는 PRF에 의존한다. PRF에 대한 입력 값은 마스터 비밀, 아스키 문자열 "key expansion", 과 서버 랜덤 값과 클라이언트 랜덤 값의 연결한 값을 이용한다. 또한, 48 바이트 마스터 비밀 자체는 PRF를 이용해 계산된다. 이 경우, 입력 값은 예비마스터 비밀, 아스키 문자열 "master secret", 그리고 클라이언트 랜덤 값과 서버 랜덤 값을 연결한 것이 된다.

4.5 CertificateVerify 메시지

```
CertificateVerify.signature.md5_hash =
        MD5(handshake_messages);
Certificate.signature.sha_hash =
        SHA(handshake_messages);
handshake_messages
        All of the data from all handshake
        messages up to
        does not include record layer headers
```

TLS의 CertificateVerify 메시지 구성은 SSL과 다르다. CertificateVerify 함수에서 서명된 정보는 복잡한 두 단계의 핸드쉐이크 메시지와 마스터 비밀, 그리고 패딩으로 구성된다. TLS의 경우 서명된 정보는 단순히 세션 동안 미리 교환된 핸드쉐이크 메시지이다.

4.6 Finished 메시지

```
struct {
        opaque verify_data[12];
} Finished;
verify_data
        = PRF(master_secret, finished_label,
            MD5(handshake_messages) +
            SHA-1(handshake_messages)) [0..11];
finished_label
        client: "client finished".
        server: "server finished".
```

TLS에서 Finished 메시지의 내용은 마스터 비밀, 레이블 "Client finished"(클라이언트에 대해) 또는 "Server finished"(서버에 대해), 그리고 모든 핸드쉐이크 메시지의 MD5와 모든 핸드쉐이크 메시지의 SHA 해쉬의 연결을, PRF에 적용함으로써 생성된 12 바이트이다.

4.7 표준 암호 조합

표준으로, TLS는 SSL과 같은 거의 동일한 암호 조합의 집합을 지원한다. 그러나 Fortezza/DMS 암호 조합에 대한 지원은 없어졌다. TLS 암호 조합에 정의된 집합은 새로운 암호 조합이 개발되면 확장될 것이면, 현재 NTRU, AES, ECC가 트래프트 문서로 진행되고 있다. <표 6>은 TLS 표준 암호 조합을 나열하고 있다.

<표 6> TLS 버전 1.0 표준 CipherSuite 값

<Table 6> TLS v1.0 Standard CipherSuite values

값	암호 조합
0,1	TLS_RSA_WITH_NULL_MD5
0,2	TLS_RSA_WITH_NULL_SHA
0,3	TLS_RSA_EXPORT_WITH_RC4_40_MD5
0,4	TLS_RSA_WITH_RC4_128_MD5
0,5	TLS_RSA_WITH_RC4_128_SHA
0,6	TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
0,7	TLS_RSA_WITH_IDEA_CBC_SHA
0,8	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
0,9	TLS_RSA_WITH_DES_CBC_SHA
0,10	TLS_RSA_WITH_3DES_EDE_CBC_SHA
0,11	TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
0,12	TLS_DH_DSS_WITH_DES_CBC_SHA
0,13	TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
0,14	TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
0,15	TLS_DH_RSA_WITH_DES_CBC_SHA
0,16	TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
0,17	TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
0,18	TLS_DHE_DSS_WITH_DES_CBC_SHA
0,19	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
0,20	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
0,21	TLS_DHE_RSA_WITH_DES_CBC_SHA
0,22	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
0,23	TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
0,24	TLS_DH_anon_WITH_RC4_128_MD5
0,25	TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA
0,26	TLS_DH_anon_WITH_DES_CBC_SHA
0,27	TLS_DH_anon_WITH_3DES_EDE_CBC_SHA

5. SSL 프로토콜 보안 취약성

SSL 프로토콜은 웹 전자상거래에 이용되어 왔으며, TLS라는 새로운 이름으로 개정되었다. 현재 개발자들은 SSL과 TLS 구현에 대해 많은 경험을 가지고 있으며, 프로토콜의 보안을 개선하는데 많은 도움을 주었다. 실제로 SSL이나 TLS 프로토콜 자체에서 알려지지 않은 보안 결점이 있지만, SSL을 사용하는 시스템 내의 취약성은 대학이나 연구기관에서 이용되고 있다[3][5].

이 장은 이러한 취약성에 대하여 분석한다. 이것은 SSL 구현을 설계하거나 평가하는 사람들에게 매우 중요하다. 물론 아래의 항목들을 자세히 규명한 것은 아니며, 새로운 위협과 공격은 차후에도 일어날 수 있다.

5.1 인증 관련 이슈

이 절은 X.509 인증서를 중심으로 SSL 인증의 이슈를 설명한다.

5.1.1 인증기관

CA는 모든 X.509 인증서에 서명하고, 모든 SSL 구현은 통신중인 실체의 CA를 신뢰할 수 있는지 결정해야만 한다. 일반적으로 구현은, 실체의 CA와 구현이 믿을 수 있는 기관의 내부 목록과 비교한다. 인증서를 검증하기 위해 실체에 의해 제공된 CA 인증서로부터의 공개키가 아닌, 자신의 내부 저장소로부터의 공개키를 이용해야 한다.

공격자는 공개키를 제외한 모든 부분에 실제 인증서와 동일한 허위 CA 인증서를 생성할 수 있는데, 대체된 공개키는 공격자의 개인키와 부합한다. 그러므로 내부 저장소로부터 CA 공개키를 복구하는 것만이 이러한 공격을 막는 구현이 될 것이다.

만약 구현이 신뢰된 CA의 내부 목록을 유지하도록 한다면, 그 구현은 사용자가 그 목록을 어떻게 갱신하도록 허용할지 신중히 고려해야 한다. 일시적인 구현에 대해, 이러한 갱신이 필요하지 않을 수도 있지만, 일반적으로 사용자는 신뢰된 CA의 목록을 변경하기 위한 방법이 필요할 것이다.

5.1.2 인증서 서명

SSL 구현은 CA의 서명을 검사함으로써 수신된 모든 인증서를 증명해야 한다.

5.1.3 인증서 유효기간

모든 SSL 구현은 모든 인증서에 대해 유효기간을 검사해야 한다. 그 유효기간은 "not before"와 "not after"를 포함한다.

5.1.4 인증서 취소 상태

인증서 취소를 지원하는 환경에서 동작하는 구현은 인증서를 허용하기 전에, 모든 인증서의 취소상태를 검사한다. 불행히도 모든 환경이 효과적으로 인증서 취소를 지원하는 것은 아니다. 이러한 경우에 구현은, 인증서 효력정지 및 폐지 목록을 수동으로 가져오도록 허용함으로써, 사용자에게 대안을 제공해야 한다.

5.1.5 인증서 소유자

구현은 정당한 당사자를 인증하였다는 것을 보장해야 한다. 공격자도 정당한 CA로부터 정당한 인증서를 얻을 수 있으며, 그 인증서는 공격자에 대한 인증서가 될 것이다.

구현이 인증서가 의도된 소유자에 대한 것인지를 검사하는 방법은 CA의 정책에 의존한다. 예를 들어, VeriSign Class 3 인증서는 인증서의 소유자의 "commonName" 필드에 인증된 웹사이트의 호스트 이름을 기입한다. 넷스케이프와 익스플로러는 URL 내에 사용자가 입력하는 호스트 이름과 이 필드를 비교한다.

5.1.6 Diffie-Hellman 트랩도어

SSL 구현이 일시적인 Diffie-Hellman 키 교환을 사용할 때, 서버는 Diffie-Hellman 파라미터의 전체 집합을 명시한다. 그러나 여기에는 충분히 안전한 Diffie-Hellman 파라미터를 구성하는 것에 관해 합법적인 불일치가 있다. 일시적인 Diffie-Hellman 키 교

환을 지원하는 클라이언트는 서버로부터 수신한 파라미터를 검사한다. 클라이언트가 신뢰하는 값을 선택한 서버가 적절한 보안을 제공할 것이라는 것을 보장한다[6][7].

5.1.7 알고리즘 롤백

ServerKeyExchange 메시지를 이용하여, SSL 서버는 클라이언트에게 클라이언트가 서버에 대한 예비 마스터 비밀을 암호화하는데 사용하는 공개키 정보를 전송한다. 이 키 정보는 서버 Certificate 메시지 내의 공개키에 대응하는 개인키를 이용해 서버에 의해 서명된다. 그러나 클라이언트가 사용해야 할 공개키 알고리즘은 ServerKeyExchange 메시지 내에 분명히 명시되지 않는다. 그래서, 정보는 서버에 의해 서명되지 않는다. 이것은 SSL 프로토콜이 "알고리즘 롤백 공격"에 취약하도록 만들 수 있다[2][3].

알고리즘 롤백 공격에서, 공격자는 두 당사자들이 예비마스터 비밀을 서명하는데 사용된 공개키 알고리즘에 대해 다른 알고리즘을 갖도록 강요한다. 예를 들어, 서버는 Diffie-Hellman을 기대한 반면에, 클라이언트는 RSA를 기대할 수 있다[5].

알고리즘 롤백 공격으로부터 보호하기 위해, SSL 클라이언트 구현은 모든 ServerKeyExchange 메시지 내의 파라미터의 길이와 개수를 검사해야 할 것이다. Diffie-Hellman이 세 개를 사용하는 반면, RSA는 두 개의 파라미터가 필요하다. 모든 수신된 메시지에 대해, 만약 파라미터들의 길이와 서명된 해쉬 값의 길이를 더한 것이 전체 메시지의 길이가 되지 않는다면, 클라이언트는 세션을 거절하고, 경고를 생성해야 한다.

5.1.8 누락된 ChangeCipherSpec 메시지

SSL 프로토콜은, Finished 메시지의 MAC 계산에 ChangeCipherSpec 메시지를 포함하지 않는다. ChangeCipherSpec 메시지는, SSL이 핸드쉐이크 프로토콜 메시지라고 간주하지 않기 때문에 생략된다. 이러한 생략은, 당사자가 암호는 사용하지 않고 인증 기능만 사용할 때, SSL 구현을 공격에 취약하게 만든다[2][3].

이 취약성을 이용하기 위해, 공격자는 통신 스트림의 ChangeCipherSpec 메시지를 삭제한다. 양 당사자는 표면적으로 유효한 Finished 메시지를 받을 것이고, 그들은 협상한 암호 조합을 활성화하지 않고, 어플리케이션 데이터의 전송을 시작할 것이다.

이 공격에 대한 해결책은 ChangeCipherSpec 메시지를 받지 않고서는, Finished 메시지를 허용할 수 없도록 하는 것이다.

5.2 암호 관련 이슈

SSL은 정보의 기밀성을 보호하는데 매우 효과적이다. 그러나 몇 가지 고려해야할 것이 있다. 이 절은 암호 키 크기의 중요성에 대해 살펴보고, SSL 암호화의 관련된 트래픽 분석과 Daniel Bleichenbacher에 의해 처음 확인된 공격에 대해 분석한다.

5.2.1 암호 키 크기

SSL이 제공하는 암호화의 강도는 매우 중요하다. 그 강도는 RC4와 DES와 같은 대칭 암호 알고리즘에 의해 사용되는 키의 크기에 직접적으로 의존한다. 이론적으로 개발자는 충분히 큰 키 크기의 SSL 구현을 생성할 수 있고, 이러한 구현은 실제로는 깨지지 않는다[10].

그러나 미국 정부는 암호의 사용이나 수출에 대해 제약을 하고 있다. U.S 법은 "수출 강도"의 SSL 암호 조합의 생성을 강요하고 있다. 이러한 제한된 키 크기 때문에 프로토콜이 취약해진다. 실제로 암호 조합을 이용해 암호화된 세션은 1995년부터 공격당했고, 현재 대부분의 보안 전문가들은 SSL의 수출 강도 암호 조합을 단지 최소한의 보안을 제공하기 위한 것으로 생각하고 있는 실정이다.

이러한 이유로 넷스케이프사와 마이크로소프트사는 U.S 법 내에서 International Step-Up과 Server Gated Cryptography를 이용하여 강력한 보안 서비스를 제공한다.

5.2.2 트래픽 분석

공격자들이 정보를 해독할 수 없을 지라도, 공격자는 트래픽을 관찰하여, 목적지에 대한 정보를 얻

을 수 있다. 트래픽 분석을 인터넷과 같이 개방된 환경에서 방어하기란 어렵다. SSL 프로토콜은 트래픽 분석 취약성을 가지고 있다. SSL이 스트림 암호를 사용할 때, 암호화된 메시지들의 크기는 암호화되지 않은 데이터의 크기를 누설할 수도 있다(공격자는 MAC 크기를 빼기만 하면 된다). Bennet Yee는 이 취약성이 공격자로 하여금 암호화된 세션에 관련된 정보를 어떻게 알아내는지를 주목해왔다. 만약 블록 암호가 사용된다면, 패딩이 평문 데이터의 크기를 숨기기 때문에, 이러한 취약성은 나타나지 않는다.

SSL 구현은 트래픽 분석 공격으로부터 데이터를 보호하기 위해 블록 암호 조합만을 제공하도록 선택할 수도 있다.

5.2.3 Bleichenbacher 공격

1998년, 루슨트 Bell 연구소의 연구원인 Daniel Bleichenbacher는, RSA 암호를 사용하는 보안 프로토콜들에 대한 능동적인 공격을 보고하였다. 이 공격은 RSA 암호 알고리즘이 데이터를 암호화하기 전에, 데이터들을 인코딩 하는 방식을 이용하였다. 인코딩 된 데이터는 항상 동일한 2 바이트(00과 02)로 시작한다[9].

실제로, Bleichenbacher의 공격은 제약을 가지고 있다. 그것이 요구하는 인위적인 암호문의 개수가 클 수 있다는 것이다. 1024 비트 RSA 계수에 대해, 공격자들은 2^{20}개 정도의 서로 다른 인위적인 암호문 블록들을 만들어야만 한다.

SSL 구현은 이런 공격에 대한 노출을 줄일 수 있는 여러 단계들이 있다. 첫 단계로, 그것을 정당한 복호화로써 받아들이기 전에 복호화된 평문을 엄격하게 검사하는 것이다. 수신된 ClientKeyExchange 메시지의 경우에, 구현은 예비마스터 비밀이 정확한 크기이며(48바이트), 첫 2 바이트가 SSL 버전 번호이고, 00과 02가 존재함을 증명해야 한다. 이러한 단계들은 공격이 요구하는 인위적인 암호문 블록들의 수를 2^{20}부터 2^{40}로 증가시킬 것이다.

이러한 공격을 막을 수 있는 다른 방법은 에러 응답 전송을 최소한으로 줄이는 것이다. 최선의 방법은, ClientKeyExchange 메시지를 복호화할 수 없거나 성공적이더라도 결과적인 평문이 정당하지 않더라도, SSL 구현은 일관성 있게 행동한다.

가능한 방법은 복호화된 ClientKeyExchange 메시지가 RSA 인코딩 형식과 일치하지 않은 것을 무시하는 것이다. 이것을 위한 방법은 유효하지 않은 데이터를 동일한 랜덤 데이터로 대체하는 것이다. 그러면 서버는 유효하지 않은 데이터가 적절히 만들어진 것처럼 에러를 감지하고 응답한다.

RSA 알고리즘에 의해 암호화 된 대칭키만 이 공격에 위험하며, RSA 개인키를 위협하지는 않는다.

5.3 일반적인 이슈

이번 절에서는 RSA 키 크기, 버전 롤백 공격, 때이른 종료, SessionID 값, 난수 발생, 그리고 난수 시드를 포함한 이슈들을 분석해 본다.

5.3.1 RSA 키 크기

대부분의 SSL 구현은 전자서명과 공개키 암호에 대해 RSA의 암호 알고리즘을 사용한다. RSA 알고리즘의 강도는 RSA 공개키의 크기에 직접 의존한다. 계산 능력이 증가하면, 비용은 감소하는데, 적당히 안전하다고 생각되는 키 크기는 Brute-force 공격에 취약하다. RSA 연구소는 RSA 알고리즘에 대해 허용할 수 있는 키의 최소한의 크기를 768비트로 발표했다[10].

5.3.2 버전 롤백 공격

SSL 명세서는 버전을 강요하는 공격으로부터 보호하기 위해 방법을 기술하였지만, 이전 세션의 재시작에 대해서는 발표하지 않았다. SSL 구현은 v3.0을 이용하여 수립된 세션이 v2.0을 이용하여 재시작되도록 허용할 수 있다. 비록 SSL 구현이 이러한 동작을 허용하지 않을 지라도, 구현은 SSL 표준을 따라야 한다. 만약 세션이 SSL v3.0을 이용하여 수립되어진다면, 그 구현은 세션을 재시작하려고 시도하는 모든 것들이 SSL v3.0을 사용한다는 것을 보장해야 한다[2].

5.3.3 절삭 공격

또 다른 보안 이슈는 절삭 공격의 위협이다. 만약 공격자들이 전송 중인 프로토콜 메시지들을 삭제할 수 있다면, 공격자들은 하나 또는 양 통신 당사자들이 부분적인 정보만을 수신하는 시나리오를 만들 수 있다. 만약 누락된 정보가 통신에 치명적이라면, 공격자는 교환의 전체 보안을 손상시킬 것이다.

SSL 프로토콜은 이 공격으로부터 보호할 수 있는 ClosureAlert 메시지를 정의하고 있다. 그러나 모든 환경이 ClosureAlert에 의존할 수 있는 것은 아니다. 예를 들어, 웹브라우저 사용자들은 컴퓨터가 ClosureAlert 메시지를 보낼 수 있는 기회를 갖기 전에, 작업을 끝낸 후에 그들의 컴퓨터를 끌 수도 있다.

더욱더 철저한 방법은 SSL 보안을 사용하는 어플리케이션들이 절삭 공격에 민감하도록 요구하는 것이다. 예를 들면, HTTP를 지원하는 웹 서버들은 그들이 클라이언트에게 전송하는 각 페이지 내에 "ContentLength" 필드를 포함하고 있다. 클라이언트들은 그들이 수신한 데이터의 양이 이 필드의 값과 일치하는지 검사해야 한다.

5.3.4 SessionID 값

SSL 명세서는 서버에게 특별한 SessionID 값을 선택하기 위한 유연성을 제공한다. SessionID를 선택할 때, 서버 구현은 중요한 정보를 포함하지 않도록 주의해야 한다. SessionID 값은 암호가 활성화되기 전에 ClientHello와 ServerHello 내에 전달되므로, 그 값은 공격자들에게 완전히 노출되어진다.

5.3.5 난수 생성

난수는 ClientHello 내에 교환되어지고, ServerHello 메시지에서 세션에 대한 암호 키를 결정한다. 그러나 난수는 컴퓨터 소프트웨어는 랜덤한 것을 아무 것도 할 수 없다. 대신에 소프트웨어 구현은 일반적으로 의사난수 생성자로 알려진 알고리즘들에 의존한다. 그러나 의사난수 생성자는 두 가지 문제점을 가지고 있다. 첫 번째 문제는 알고리즘 자체의 효율성에 있다. 대부분의 소프트웨어 라이브러리는 선형 합동 생성자 알고리즘을 이용하여 의사난수를 생성한다. 비록 그 알고리즘이 효과적인 의사난수 생성기라고 할지라도, 효과적이지 않을 수도 있다. 게다가, 많은 개발자들은 기본 알고리즘의 개선을 찾고 있다.

Teukolsky, Vetterling, 그리고 Flannery는 광범위하게 사용되는 의사난수 생성자 보고하였다. 극단적인 경우에는 단지 11개의 구별되는 인수만 생성한다.

선형 합동 생성자가 가지고 있는 또 하나의 심각한 문제는 그들이 순차적이어서 예상할 수 있다는 것이다. 만약 알고리즘의 파라미터와 하나의 특정 값을 안다면, 그 알고리즘이 생성하는 차후의 모든 값들을 예상할 수 있다. 예상 가능한 난수는 모든 보안 프로토콜에 심각한 문제이다. 그러므로 SSL의 구현은 일반 의사난수 생성자 라이브러리를 사용하지 않도록 조심해야 한다. 다행히, 암호와 해쉬 알고리즘을 포함한 표준 암호 알고리즘은 효과적인 난수를 제공하기 위해 수정될 수 있다.

5.3.6 난수 시드

알고리즘과 관계없이, 구현은 난수를 생성해야 하며, 일반적으로 구현은 시드(seed)를 가진 알고리즘을 제공해야 한다. 시드에 대한 기본적인 요구사항은 생성되는 시간이 각각 다르다는 것이다. 많은 개발자들 시드로써 시간을 사용한다. 그러나, 보안 어플리케이션 난수 시드는 달라야 하며, 예상할 수 없어야 한다. 시간은 일반적으로 예상이 가능하다. 그러므로 랜덤 시드에 대해 시간을 사용하는 것은 SSL 구현에 대해 수용될 수 없다.

6. 결론

SSL이나 TLS를 개발하는 개발자들은 암호학적 지식이나 프로토콜에 대한 철저한 분석 없이는 보안 취약성에 노출될 수밖에 없는 실정이다.

본 논문은 개발자들이 S/W 구현 시, 알아야할 프로토콜 구조와 점검해야 할 취약성들을 분석하였다. 개발자들은 S/W 개발 시, 이 점검사항들을 검사하여야 한다. 물론, 여기에 나열한 것만이 모든 것은 아니다. 차후에 다른 형태의 공격들이 발생할 수 있다는 것을 밝혀둔다.

독점 기술로 시작된 SSL에서 표준으로서의 TLS로 발전되었다. SSL과 TLS의 발전은 웹브라우저, 웹개발자, 보안이 필요한 인터넷 시스템, 그리고 IETF의 손에 달려있다고 해도 과언이 아니다. SSL v3.0은 보안이 필요한 시스템과 인터넷 트랜잭션 분야에 잘 적용되어, 크게 성장될 것이다.

또한, 본인은 현재 플랫폼 독립적인 자바(JDK 1.3)를 이용하여 SSL과 TLS에서 사용하는 암호 알고리즘들과 국내 암호 알고리즘인 SEED, KCDSA, HAS-160을 구현 중에 있으며, 각 취약성들을 해결할 수 있도록 SSL을 설계하고 구현할 계획이다.

※ 참고문헌

[1] http://home.netscape.com/eng/ssl3/draft, 302.txt, The SSL Protocol Version 3.0, 1996.

[2] http://www.ietf.org/rfc/rfc2246.txt, The TLS Protocol Version 1.0, 1999.

[3] David Wagner, Bruce Schneier, "Analysis of the SSL 3.0 protocol", The Second USENIX Workshop on Electronic Commerce Proceedings, USENIX Press, November 1996.

[4] 이임영, 이재광, 소우영, 최용락, "컴퓨터 통신 보안", 도서출판 그린, 2001.

[5] Eric Rescorla, "SSL and TLS, Addison Wesley, 2001.

[6] Kaufman, Charles, "Network Security", Prentice Hall, 1995.

[7] Bruce Schneier, "Applied Cryptography", John Wiley & Sons, 1996

[8] Apostolopoulos G, Peris V, Pradhan P, Saha D. Securing electronic commerce: reducing the SSL overhead. [Journal Paper] IEEE Network, vol.14, no.4, July-Aug. 2000, pp.8-16. Publisher: IEEE, USA.

[9] Daniel Bleichenbacher, "Chosen Ciphertext Attacks against Ptotocols Based in RSA Encryption Standard PKCS #1", Advances in Cryptology Crypto'98, LNCS vol. 1462, 1998.

[10] http://www/rsasecurity.com/rsalabs/factoring/rsa155.html

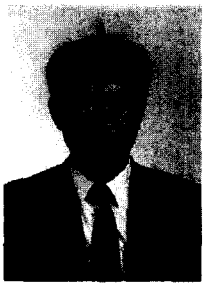
[11] 유성진, 김성렬, 정일용, "안전한 통신 서비스를 제공하는 향상된 SSL(Secure Socket Layer) 기반 정보보호 시스템의 설계", 한국통신학회논문지, 제 25권 9호, 2000.

[12] 정상곤, 정전대, 신재호, 송유진, "SSL Handshake 프로토콜의 성능 개선", 대한전자공학회, 제 21권 제 1호, 1998. 6.

조 한 진

1997년 : 한남대학교 컴퓨터공학과 공학사
1999년 : 한남대학교 대학원 컴퓨터공학과 공학석사
1999년 : 현재 한남대학교 대학원 컴퓨터공학과 박사과정
관심분야 : 컴퓨터네트워크, 정보보호, 보안 프로토콜

이 재 광



1984년 : 광운대학교 전자계산학과 이학사
1986년 : 광운대학교 대학원 전자계산학과 이학석사
1993년 : 광운대학교 대학원 전자계산학과 이학박사
1986년 : 1993년 군산전문대학 전자계산학과 부교수
1997년 - 1998년: University of Alabama 객원교수
1993년 - 현재: 한남대학교 컴퓨터공학과 부교수
관심분야 : 컴퓨터 네트워크, 정보통신 정보보호