ＳＳＬ安全性調査報告書

＜詳細編＞

改訂第２版

２００４年2月6日

株式会社ケー・ディ・ディ・アイ研究所

SSL 安全性調査報告書

＜詳細編＞

改訂第２版

2004 年 2 月 6 日

（株）KDDI 研究所

目次

0. はじめに 7

1. SSL/TLS の比較 9

1.1. SSL 3.0 の概要 9

1.1.1. SSL レイヤ構造 9

1.1.2. CipherSpec／SecurityParameters 12

1.1.3. Record Layer プロトコル 12

1.1.4. Handshake プロトコル 17

1.1.5. Alert プロトコル 25

1.1.6. ChangeCipherSpec プロトコル 26

1.1.7. Application Data プロトコル 26

1.2. SSL 3.0／TLS 1.0 の比較 27

1.2.1. MAC（Message Authentication Code：メッセージ認証符号） 28

1.2.2. master_secret の計算 30

1.2.3. key_block 計算 31

1.2.4. final_client_write_key および final_server_write_key の計算 32

1.2.5. Exportable encryption algorithms 使用時の client_write_IV および server_write_IV の計算 33

1.2.6. ServerKeyExchange メッセージの Signature の構造体 34

1.2.7. CertificateVerify メッセージで使用されるハッシュ計算 35

1.2.8. Finished メッセージ 36

1.2.9. CipherSpec（SSL）と SecurityParameters（TLS） 37

1.2.10. Alert プロトコル 39

1.2.11. その他 40

1.3. TLS 拡張作業の概要 41

1.3.1. 機能拡張 42

1.3.2. 認証方式の拡張 49

1.3.2. 鍵共有方式の拡張 50

1.3.3. 暗号方式の拡張 51

1.3.4. その他 55

2. SSL における既存セキュリティホールとその対策について 56

2.1. 暗号方式上のセキュリティホールとその対策 56

2.1.1. DSA のセキュリティホール 56

2.1.2. PKCS#1 のセキュリティホール 58
2.1.3. PKCS#1 v.2.0 OAEP に関するセキュリティホールについて 60
2.1.4. PKCS#1 (v.1.5) に対する適用的選択暗号文攻撃の拡張 62
2.1.5. CBC モードのパディング方式における脆弱性を利用したサイドチャネル攻撃 65
2.1.6. その他の CBC パディングの脆弱性を利用したサイドチャネル攻撃 71
2.1.7. PKCS#1 V2.1 などの脆弱性を利用したサイドチャネル攻撃 72
2.1.8. SSL ソフトウェアに対するタイミング攻撃 80

2.2. プロトコル上のセキュリティホールとその対策 82
2.2.1. Man-in-the-middle attack 82
2.2.2. Replay attack 82
2.2.3. Version rollback attack 83
2.2.4. Ciphersuite rollback attack 83
2.2.5. Key-exchange algorithm rollback attack 83
2.2.6. Cut-and-paste attack 84
2.2.7. Short-block attack 84
2.2.8. Dropping the change cipher spec message 85
2.2.9. Attack against finished messages 85
2.2.10. Traffic analysis 86
2.2.11. Attack against a resuming session 86

2.3. 実装上のセキュリティホールとその対策(2002 年 3 月まで) 87
2.3.1. Interner Explorer HTTPS certificate attack 87
2.3.2. Bypassing Warnings For Invalid SSL Certificates In Netscape Navigator 88
2.3.3. Multiple Vendor SSL Certificate Validation Vulnerability 89
2.3.4. RSA BSAFE SSL-J Authentication Bypass Vulnerability 90
2.3.5. OpenSSL PRNG Internal State Disclosure Vulnerability 91
2.3.6. Microsoft IE SSL Spoofing Vulnerability 92
2.3.7. Netscape Communicator Inconsistent SSL Certificate Warning Vulnerability 93
2.3.8. OpenSSL Unseeded Random Number Generator Vulnerability 94
2.3.9. IIS / Site Server Multithread SSL Vulnerability 95
2.3.10. NT IIS SSL DoS Vulnerability 96
2.3.11. Netscape Navigater SSL における疑似乱数の seed によるセッション鍵の推定による攻撃 97
2.3.12. Microsoft IE 4.x, 5.x における Cached Web Credentials 問題 99
2.3.13. その他 SSL というキーワードで検索されるが直接 SSL とは関係のないセキュリティホール 100

2.4. 実装上のセキュリティホールとその対策(2002 年 3 月以降) 102
2.4.1. ASN.1 に関するセキュリティホール 102
2.4.2. Buffer Overflow による脆弱性 107
2.4.3. ルート証明書配送の問題 111
2.4.4. 表示の不具合が実質上のセキュリティホールになる問題 114
2.5. 運用上の注意点 116
2.5.1. 秘密鍵や証明書などの重要なファイルの管理 116
2.5.2. RandomSEED の取得 117
2.5.3. セッション鍵のライフタイム 117
2.5.4. 使用するプロトコルバージョン 119
2.5.5. 使用する暗号アルゴリズム 119
2.5.6. 証明書検証 120
2.5.7. ログ 121
2.5.8. 警告通知 121
2.5.9. その他設定 122
2.5.10. パッチマネージメントについて 123
参考文献 125

0. はじめに

SSL は、インターネットにおいて最も利用されているセキュリティプロトコルである。Web 上の暗号化、認証機能として利用されており、SSL を用いた各種電子行政サービスや、民間のオンラインクレジット決済サービスなどで広く利用されている。しかしながら、SSL を使っているから安全といった認識が蔓延しており、SSL がどの程度の安全性を有するのかを客観的に調査・評価した試みはあまり見られない。

上記の背景にともない、2001 年度に CRYPTREC 殿の仕様に従い、SSL/TLS について（1）暗号方式そのものの安全性、（2）プロトコル（メカニズム）としての安全性、（3）実装に関する安全性、（4）運用上の安全性といった観点から、これまで報告されてきたセキュリティホールについて調査を行った。約 2 年経過した 2004 年 1 月、その後、方式や実装に関わる新たな攻撃や改良された攻撃が提案されている。

従って、本調査では、SSLVer3.0/TLS Ver1.0 以上を対象とし、前回の調査から現在（2004 年 1 月）に至るまでに新たに発見・公開されたセキュリティホールおよびその対処法を調査することを目的としている。具体的には、前回の調査の後、報告されている SSL/TLS に対する攻撃として、サイドチャネル攻撃、タイミング攻撃、その他の実装攻撃の観点から調査を行った。以下、本調査結果の概要を述べる。

サイドチャネル攻撃については、SSL/TLS で用いられる暗号化のパディング（CBC パディング）に関して、バイト単位の固定のパディングパターンを使用している場合、パディングの正当性チェックの結果を利用して、平文をバイト単位で逐次求めることが可能であるという攻撃手法が存在する。また、RSA Encryption については、これまで報告された Bleichenbacher の PKCS#1（v.1.5）に対する適用的選択暗号文攻撃を改良した実用的な攻撃手法が提案されている。いずれの攻撃法についても、対策が提案され、最新のソフトウェアでは対処がなされている。そして、PKCS#1（v.2.1）に従う EME-OAEP-DECODE 手法による復号プロセスにおいて、攻撃者が平文の一部のハミング重みを集めることにより、平文の最下位ビットを露呈させ、さらにその情報から所望の平文を得るという手法や、RSA-KEM において、攻撃者が秘密鍵のあるビットにフォールトを故意に起こさせることにより、秘密鍵をビットごとに露呈させる手法も提案されている。しかし、前者はハミング重みに対する電力サイドチャネル攻撃を、後者は秘密鍵に対する電力フォールトサイドチャネル攻撃をそれぞれ前提としているため、SSL/TLS においては実用的な攻撃手法ではない。

タイミング攻撃については、SSL のソフトウェアで用いられている RSA 実装において、高速化を目的として用いられる Montgomery reduction や Karatsuba 乗算法に起因し、RSA 復号処理においてその入力値の大きさにより処理ロジックが異なり、タイミングアタックが成功する報告もなされている。本アタックについても対策が提案されている。　実装上の攻撃としては、ASN.1 に対する脆弱性を含め Buffer Overflow に関するセキュリティ

ホールが種々報告されており、これらをまとめている。また、SSL/TLS プロトコルに付随する証明書管理技術におけるルート証明書の更新手法の課題や、ブラウザの表示に関する不具合が実質上の攻撃につながる問題も記載している。報告されている種々のセキュリティホールについては、2004 年 2 月 19 日現在パッチによる対策がなされているが、脆弱性公開存在しても公開されない場合や、公開からパッチが提供までに時間がかかるなど問題が残っている。

その他、運用上の対策として最新パッチを管理する技術が重要となるため、米国の NIST（National Institute of Standards and Technology）では、セキュリティ上の問題を修正するパッチの運用指針を定めた“Procedures for Handling Security Patches”を発行している。SSL/TLS とは直接関係はないが、一般的なソフトウェアの運用の課題として、本文書の概要を述べる。

これらの調査結果は、2 章以降において、2001 年度の報告書を更新する形で反映している。

SSL/TLS は、ある種、完成されたセキュリティプロトコルと考えられているが、2001 年度の調査後も、Buffer Overflow 等の実装上の不具合に加えて、サイドチャネル攻撃やタイミング攻撃などの新たな攻撃手法が提案されているのが実状である。従って、電子政府などの各種システムで SSL/TLS を安全に利用するためには、今後も継続的に SSL/TLS の安全性について監視を行い、脆弱性に関する正確な知識をもち、最新の対策を施すことが望まれる。本調査報告がその一助となれば幸いである。

1. SSL/TLS の比較

SSL と TLS は、プロトコル構造や提供する機能に関して、ほぼ同等の規格である。従って、本節では、始めに SSL の概要を述べ、その後、SSL と TLS の差分について具体的に言及することとする。

1.1. SSL 3.0 の概要

1.1.1. SSL レイヤ構造

SSL は、OSI 参照モデルでいうところのセッション層（TCP/UDP の直上）に位置するプロトコルで、本プロトコルを用いることにより、盗聴や改竄を防止できる。

尚、1.1 節において、SSL と TLS が同じ機能を保有する場合は、SSL としてすべて記述した。また、微小な差（例えば、プロトコルバージョンなどのパラメータ値等）については、SSL および TLS の差が分かるよう、明記した。明確な機能の差については、1.2 節に示す。

SSL は、図 1.1.1.1 に示すとおり、構造上さらに二つのレイヤに分かれており、下位レイヤには、上位層／下位層からのデータの分割／組立、圧縮／伸張、暗号化／復号化を行う Record Layer プロトコルがある。また、上位レイヤには Handshake、Change Cipher Spec、Alert、Application Data の四つのプロトコルがあり、認証手続きや SSL で通信するためのネゴシエーション、異常状態を知らせる警告メッセージ送受信などの機能を有する。

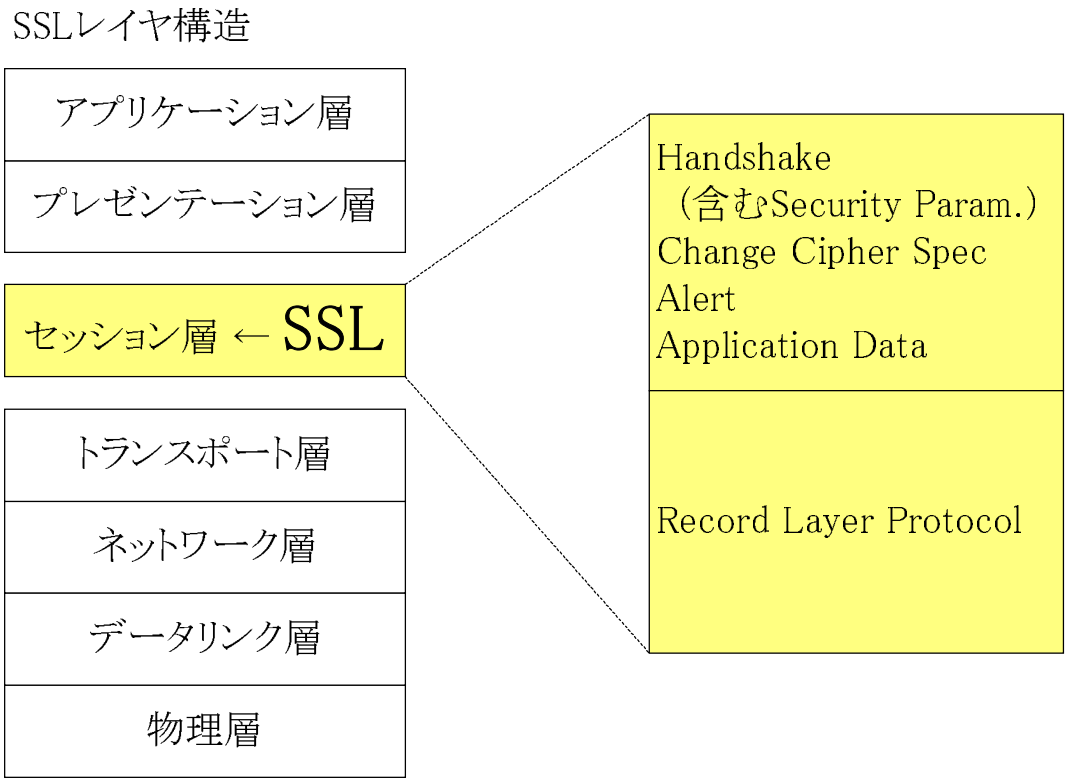


図 1.1.1.1　SSL レイヤ構造

送信時においては、Application Data、Alert、Handshake、Change Cipher Spec のプロトコルから送られるデータ・メッセージはすべて Record Layer Protocol で圧縮・暗号化の処理を行い、通信相手に送信される。受信時では逆に Record Layer Protocol 解凍・復号化の処理を行い、上位の各プロトコルに引き渡される。
Application Data プロトコルは、さらに上位レイヤのアプリケーションで使用されるデータを送受信する。
Handshake プロトコルでは、サーバ・クライアント間の認証や暗号化方式・鍵の値や MAC 方式などのネゴシエーションを行う。またこのプロトコルで設定された値について、通信相手に Change Cipher Spec プロトコルはこの値の使用開始を通知するとともに、Record Layer プロトコルにおいて使用される CipherSpec／SecurityParameters にパラメータをセットする。
各プロトコルにおいて、何らかの異常があったときは Alert プロトコルに通知する。Alert プロトコルでは異常状態に応じたメッセージを通信相手に送信し、コネクション切断などの処置を行う。

各プロトコルは 1.1.2 節以降で説明する。

1.1.2. CipherSpec／SecurityParameters

CipherSpec（TLS では SecurityParameters）は SSL での動作環境を示している。これらは構造体の形を呈しているが、この構造体そのものが通信されることはなく、自身での暗号処理や MAC 処理、認証処理などで参照する。これらの値は、Handshake プロトコルにおいて暗号方式や圧縮方式、MAC 方式などが決定されたときに設定される。
CipherSpec と SecurityParameters の構造体および違いに関して 1.2.9 節で示す。

1.1.3. Record Layer プロトコル

Record Layer プロトコルでは、上位レイヤからデータが渡されると、データサイズが 2 の 14 乗バイトを超える場合はそれを超えないように分割し、SSLPlaintext、SSLCompressed、SSLCiphertext の順に処理を行い、その結果を下位レイヤに引き渡す。なお受信したものを下位レイヤから受け取った場合は、逆の手順で上位レイヤに引き渡す。

上位レイヤから下位レイヤへのデータの引き渡しを図 1.1.3.1.に示す。
SSLPlaintext、SSLCompressed、SSLCiphertext の構造は後述の通り、どれも ProtocolVersion、ContentType、length、fragment で構成される。このうち ProtocolVersion と ContentType は同じ値がそのまま使用される。
上位から送られたデータ・メッセージはまず SSLPlaintext.fragment となる。SSLPlaintext.fragment は Handshake プロトコルで規定された圧縮方式により圧縮処理が行われ SSLCompressed.fragment となる。SSLCompressed.fragment は Handshake プロトコルで規定された暗号化・MAC 方式により暗号化・MAC 処理が行われ SSLCiphertext.fragment となり、下位レイヤに引き渡されていく。length は SSLPlaintext、SSLCompressed、SSLCiphertext それぞれの fragment の大きさを示すことになる。
ただし、初めてコネクションを張る場合は、まだサーバ・クライアント間でのネゴシエーションが取れていないので、この場合は圧縮・暗号化・MAC アルゴリズムがどれも指定されていないタイプで通信することにより、Handshake プロトコルで圧縮・暗号化・MAC アルゴリズムのネゴシエーションを取る。

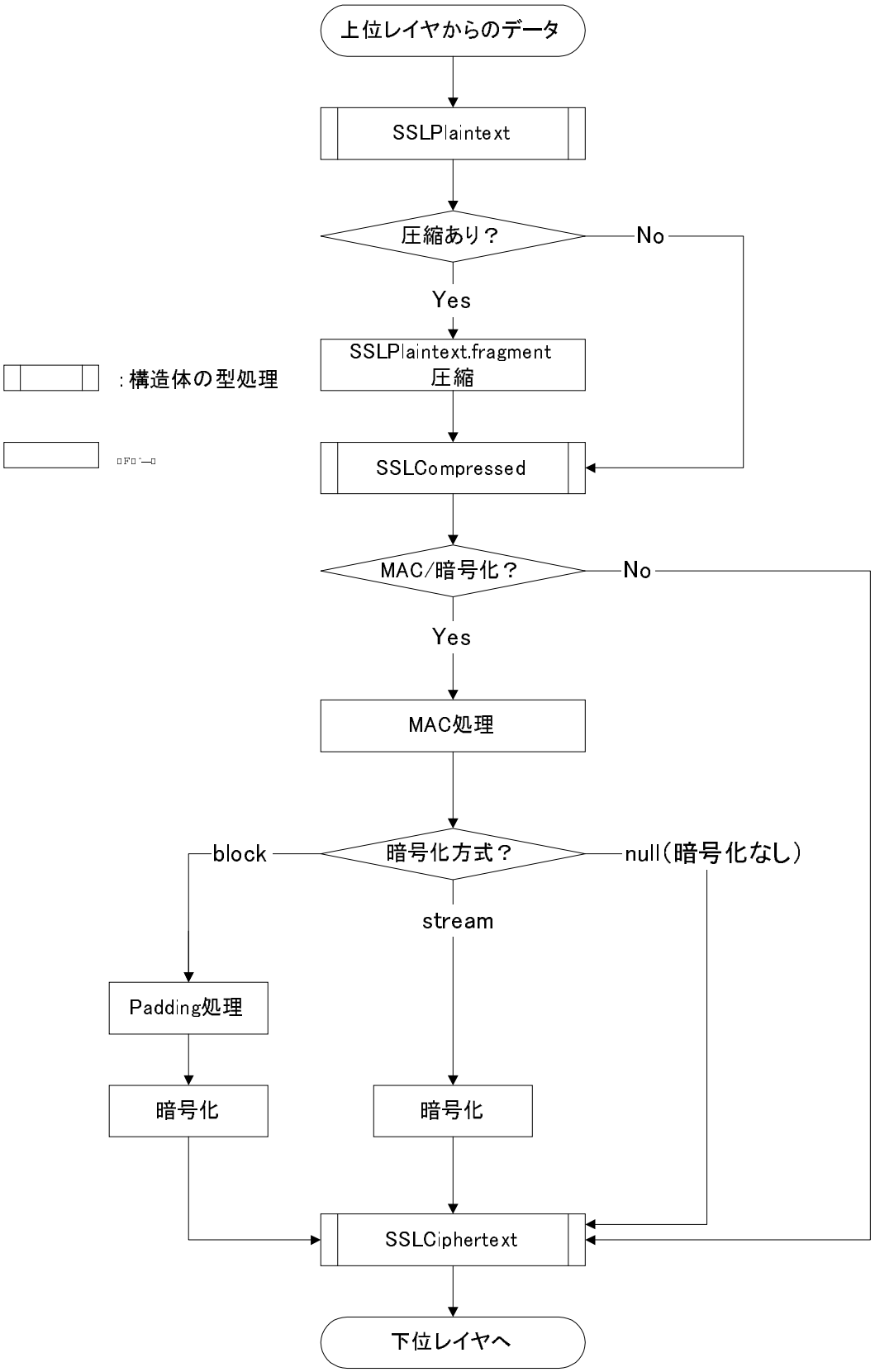


図 1.1.3.1. Record Layer での圧縮／暗号化／MAC 処理フロー

以下、uintαはαビットの数値、opqueはデータ列（長さは構造体の[ ]内の値）を表す。

（1）SSLPlaintext（TLS では TLSPlaintext）

準備：
```
struct {
    uint8  major, minor;
} ProtocolVersion;
enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;
```

構造体：
```
struct{
    ContentType  type;
    ProtocolVersion  version;
    uint16  length;
    opaque  fragment[SSLPlaintext.length]
} SSLPlaintext;
```

上位から受け取ったデータを Record Layer ではまず上記に示す構造体の形にする。type は上位レイヤからどんなメッセージ・データが送られてきたのかを示している。version では使用するプロトコルバージョン（SSL 3.0 なら「3, 0」、TLS 1.0 なら「3, 1」）が明記される。length は次に続く実際のデータ（またはメッセージ）部分の長さを示しており、2 の 14 乗バイトを超えない値になっている。最後の fragment は上位からの実際のデータ（またはメッセージ）となる。

（2）SSLCompressed（TLS では TLSCompressed）

構造体：
```
struct{
    ContentType  type;              /* SSLPlaintext.type と同じ*/
    ProtocolVersion  version;       /* SSLPlaintext.version と同じ*/
    uint16  length;
    opaque  fragment[SSLCompressed.length]
} SSLCompressed;
```

SSLPlaintext から、圧縮アルゴリズムにしたがいデータ・メッセージを圧縮し、SSLCompressed という構造体に変換する。

ここでは type、version では SSLPlaintext での値をそのまま用いる。length は圧縮したあとのデータ・メッセージの長さを示す。この値は 2 の 14 乗＋1024 バイトの値を超えない。fragment は SSLPlaintext.fragment を圧縮したものである。fragment 展開時には 2 の 14 乗バイトを超えてはならず、超えた場合は alert において decompression failure で応答しなければならない。ここの圧縮で使用されるアルゴリズムは Handshake プロトコルで規定される。また初期値としては非圧縮方式（CompressionMethod.null）となっている。

注：SSL の Draft または TLS の RFC においては、非圧縮方式のみしか採用されていない。ただし TLS の RFC においては方式を追加しても構わないことが明記されている。

（3）SSLCiphertext（TLS では TLSCiphertext）

```
構造体：struct{
        ContentType    type;            /* SSLCompressed.type と同じ*/
        ProtocolVersion    version;     /* SSLCompressed.versionと同じ*/
        uint16    length;
      select (CipherSpec.cipher_type){
            case  stream: GenericStreamCipher;
            case  block: G enericBlockCipher;
      } fragment;
    } SSLCiphertext;
```

SSLCompressed は、暗号化・MAC 手続きを経て、SSLCiphertext に変換される。上記構造体のうち type、version は SSLCompressedtext と同じ、length は 2 の 14 乗＋2048 バイトを超えない fragment 長をあらわす。
fragment は SSLCompressed.fragment を MAC 付きで暗号化したものである。暗号化の方法は CipherSpec で示されている cipher_type および hash_size の値で決められる。MAC の計算方法は 1.2.1 節を参照のこと。

cipher_type が null（暗号化しない）または stream を示すとき、以下の形の物を作成し、これを Handshake プロトコルで規定した方式で暗号化したものを fragment とする。

```
stream-ciphered    struct (
    opaque    content[SSLCompressed.length];
    opaque    MAC[CipherSpec.hash_size];
} GenericStreamCipher;
```

cipher_type が block の場合、content と MAC の他に、暗号化するときに長さを調節する

padding およびその長さを示す padding_length を連接し、それを暗号化した物を fragment とする。

```
block-ciphered    struct (
    opaque    content[SSLCompressed.length];
    opaque    MAC[CipherSpec.hash_size];
    uint8    padding[GenericBlockCipher.padding_length];
    uint8    padding_length;
} GenericBlockCipher;
```

両方式の場合も、content と MAC（block の場合はさらに padding、padding 長）を連接した後で暗号化する。

1.1.4. Handshake プロトコル

Handshake プロトコルはサーバ・クライアント間の認証や暗号化方式・鍵の値や MAC 方式などのネゴシエーションを行う。ここで通信されるプロトコルでは、以下のようなメッセージのやりとりが行われる。なお、ChangeCipherSpec は Handshake とは別プロトコルとなるので、1.1.6 節で説明を行う。

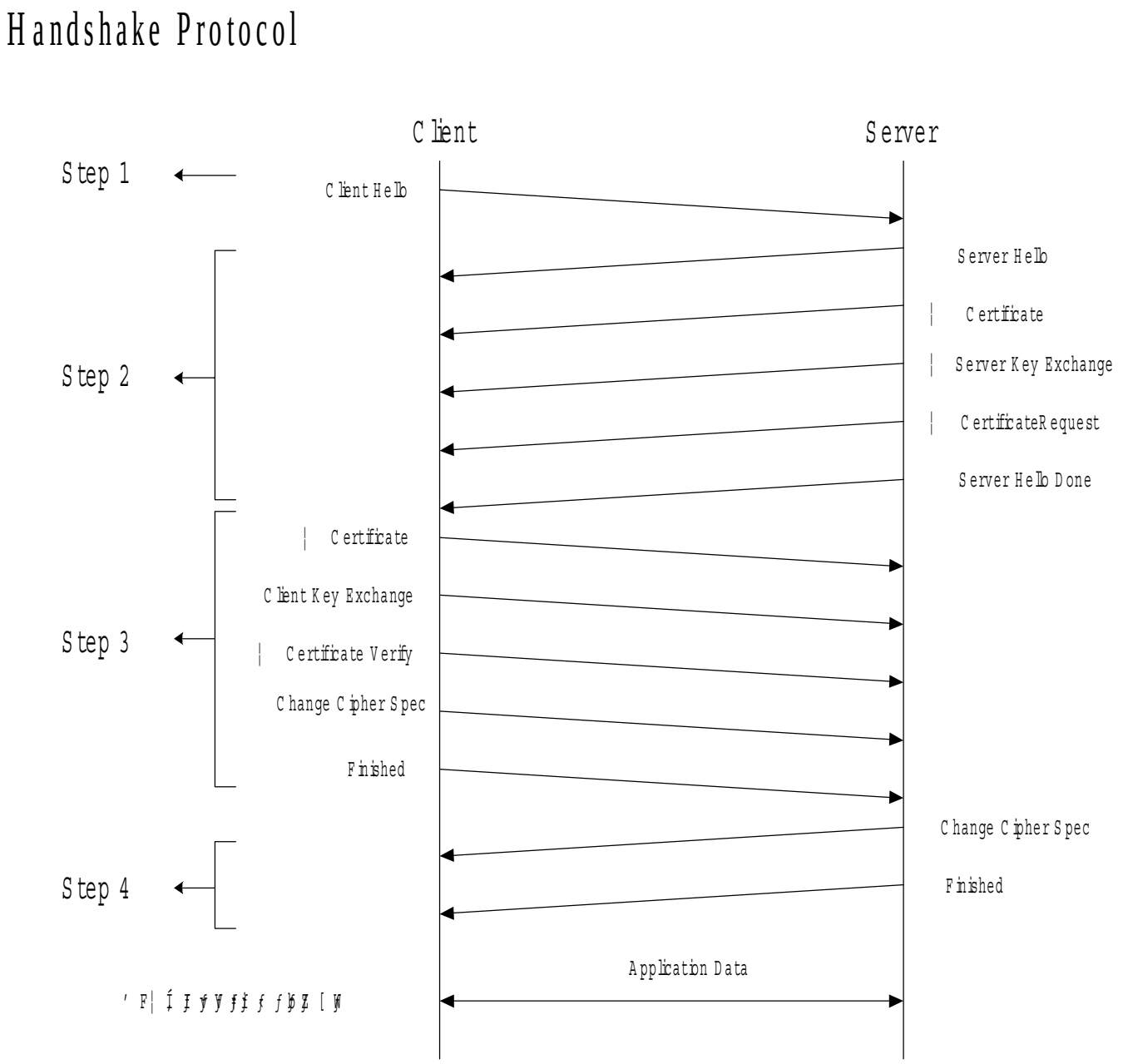


図 1.1.4. SSL の Handshake protocol 処理概要

Handshake プロトコルは次の構造体を持つ。

```
enum {
    hello_request(0), client_hello(1), server_hello(2), certificate(11),
    server_key_exchange(12), certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16), finished(20), (255)
} HandshakeType;
struct {
    HandshakeType    msg_type;
    uint24    length;
  select (HandshakeType) {
        case    hello_request:          Hellorequest;
        case    client_hello:           ClientHello;
        case    server_hello:           ServerHello;
        case    certificate:            Certificate;
        case    server_key_exchange:    ServerKeyExchange;
        case    certificate_request:    CertificateRequest;
        case    server_hello_done:      ServerHelloDone;
        case    certificate_verify:     CertificateVerify;
        case    client_key_exchange:    ClientKeyExchange;
        case    finished:               Finished;
  } body;
} Handshake;
```

msg_type はこのメッセージのタイプ、length はメッセージのバイト長を示す。body は各メッセージ内容を示し、これはメッセージのタイプによって違いがある。以下に、メッセージタイプごとの body の中身を説明する。
以下、メッセージの概要を送出順に示す。

（1）HelloRequest

これは実際の Handshake プロトコルでの手続きに使われるものでなく、再手続きが開始されるべきであることをサーバから通知するメッセージである。クライアントがこのメッセージを受け取っても、手続きを開始する義務はなく、Alert プロトコルで no_renegotiation 応答（TLS のみ）または HelloRequest を無視してもよい。サーバがこのメッセージを送信したのにもかかわらず、クライアントから ClientHello メッセージが送信されてこなかった場合、fatal レベルの Alert プロトコルで応答し、コネクションを終了してもよい。
このメッセージは、次のような空の構造体になっている。

```
struct { } HelloRequest;
```

（2）ClientHello

クライアントが最初にサーバに接続するときの最初のメッセージが ClientHello である。また HelloRequest の応答としても使用される。クライアントは ClientHello を送信したら ServerHello を待ち、もし HelloRequest 以外のメッセージがサーバから返答されたら、fatal エラーとする。このメッセージは次の構造体をとる。

```
準備：  struct {
            uint32  gmt_unix_time;
            opaque  random_bytes[28];
        } Random;
        opaque  SessionID<0...32>;
        uint8  CipherSuit[2]
構造体：struct {
            ProtocolVersion  client_version;
            Random  random;
            SessionID  session_id;
            CipherSuite  cipher_suites<0...2^16-1>;
            CompressionMethod  compression_methods<0...2^8-1>;
        } ClientHello;
```

client_version は、このセッションで通信しようとするプロトコルのバージョンを示すもので、サポートする最新のバージョン（SSL 3.0 なら「3,0」、TLS 1.0 なら「3,1」）が望ましい。
random は鍵計算などで使用されるもので、UNIX 標準（32 ビット表示）の現在時間と、28 バイトの乱数から構成される。
session_id は再ネゴシエーション時のみ、そのセッション ID が示される。新規セッションの場合は 0x00 の値のみ挿入する。

cipher_suites はクライアントでサポートしている暗号方式のリストを示す。session_id になんらかの値が設定されているときは、その ID でのセッションで使用している暗号方式もリストに加えなければならない。
compression_methods はクライアントによってサポートされている圧縮方式のリストを示す。これにはかならず CompressionMethod.null（非圧縮）を必ずリストに加えなければならない。

注： compression_methods は、Draft または RFC では非圧縮のみサポート

（3）ServerHello
ServerHello は ClientHello の応答としてサーバから送信されるメッセージである。基本的な構造体は ClientHello と似ているが、ClientHello においてサイズが可変な変数であった CipherSuite と CompressionMethod はサーバが値を決定し、ServerHello には session_id 以外のサイズが可変な変数は存在しない。もし ClientHello で示されているものがサーバで採用できない場合は、Alert プロトコルの handshake_failure で応答する。

```
準備：  struct {
            uint32  gmt_unix_time;
            opaque  random_bytes[28];
        } Random;
        opaque  SessionID<0...32>;
        uint8  CipherSuit[2]
構造体：struct {
            ProtocolVersion  server_version;
            Random  random;
            SessionID  session_id;
            CipherSuite  cipher_suites;
            CompressionMethod  compression_methods;
        } ClientHello;
```

server_version ではクライアントが示したもののうち、自分が採用できる最上位のバージョンを示す。
random はサーバ側で生成した乱数で、ClientHello.random とは異なる値である
session_id は、ClientHello で値が示されている場合はその値を、新規セッションを示している場合は新たなセッション ID を発行する。また空で返すこともでき、セッションの再利用ができないことを示す。
cipher_suites、compression_methods は ClientHello で示されたリストのうち、採用する

ものを返答する。

（４）Certificate（サーバ側から）

このメッセージは、cipher_suites で匿名方式でない鍵交換方式を採用するときに、サーバ証明書として送信される。ServerHello の直後に送信され、次のような構造体を持つ。

```
準備：  opaque  ASN.1Cert<1...2^24-1>;
構造体：struct {
            ASN.1Cert  certificate_list<0...2^24-1>;
        } Certificate;
```

certificate_list はそのサーバ証明書からルート証明機関の証明書にいたるまでの X.509v3 証明書のリストを示す。

（５）ServerKeyExchange

このメッセージはサーバからの Certificate（このメッセージが送られない匿名ネゴシエーションのときは ServerHello）の直後に送信され、premaster_secret を送信するのに必要となる暗号情報として使用される。このメッセージは以下の構造体を持つ。ただし「＊」マークは SSL でのみ採用されている fortezza に関するパラメータである。

```
準備：  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;
```

```
構造体：struct {
            select (KeyExchangeAlgorithm) {
                    case   diffie_hellman:
                        ServerDHParams   params;
                        Signature   signed_params;
                    case   rsa:
                        ServerRSAParams   params;
                        Signature   signed_params;
                ＊ case   fortezza_kea:
                        ServerFortezzaParams   params;
                };
            } ServerKeyExchange;
```

params は鍵交換に必要となるパラメータで、rsa、diffie_hellman は公開値を示している（fortezza は省略）。
signed_params は（ランダム値を連接させた）params にハッシュをかけ、それに署名をしたものである。この構造体に関しては 1.2.6 節で示す。

（６）CertificateRequest
このメッセージは、クライアントに証明書を要求するときに ServerKeyExchange（このメッセージを送信していないときはサーバからの Certificate）の直後に送信され、次の構造体を持つ。ただし、「＊」マークは SSL でのみ採用されているパラメータである。

```
準備：   enum {
            rsa_sign(1), dss_sign(2), rsa_fixed_dh(3), dss_fixed_dh(4),
            ＊rsa_ephemeral_dh(5),  ＊dss_ephemeral_dh(6),
            ＊fortezza_missi(20), (255)
        } CertificateType;
        opaque   DistinguishedName<1...2^16-1>;
構造体：struct {
            CertificateType   certificate_types<1...2^8-1>;
            DistinguishedName   certificate_authorities<3...2^16-1>;
        } CertificateRequest;
```

certificate_types は要求される証明書の種類のリストを示す。
certificate_authorities は受理できる認証局の DistinguishedName のリストを示す。

（7）ServerHelloDone

このメッセージは ServerHello から CertificateRequest までのうち、送信すべきものを送信し終わった後に送信するメッセージである。サーバはこのメッセージを送信後、クライアントからの応答を待つ。クライアントは、このメッセージを受信するまでに受け取ったすべてのメッセージから、応答すべきパラメータを計算し返信する。このメッセージは、次のような空の構造体になっている。

```
struct { } ServerHelloDone;
```

（8）Certificate（クライアント側から）

ServerHelloDone を受信した後にクライアントが最初に送信することのできるメッセージで、サーバから CertificateRequest を受信したときのみ、クライアントから Certificate を送信できる。

サーバからの要求に適合する証明書を所持していない場合は、このメッセージの代わりに Alert プロトコルの no_certificate メッセージで応答する。

構造体は（4）で示したものと同じである。

（９）ClientKeyExchange

クライアントからの Certificate の直後（これを送信しないときは、ServerHelloDone 受信直後）に送信され、premaster_secret を設定する。このメッセージは次の構造体になっている。ただし「＊」マークは SSL でのみ使用されるパラメータである。

```
struct {
  select (KeyExchangeAlgorithm) {
        case   rsa:   EncryptedPreMasterSecret;
        case   diffie_hellman:   ClientDiffieHellmanPublic;
    * case   fortezza_kea:   FortezzaKeys;
  } exchage keys;
} ClientKeyExchange;

RSA:    struct {
            ProtocolVersion client_version;
            opaque   random[46];
        } PreMasterSecret;
        struct {
            public-key-encrypted   PreMasterSecret   pre_master_secret;
        } EncryptedPreMasterSecret;
Diffie-Hellman:
        enum {implicit, explicit} PublicValueEncoding;
        struct {
          select (PublicValueEncoding){
                case   implicit:   struct { };
                case   explicit:  opaque dh_Yc<1...2^16 -1>;
          } dh_public;
        } ClientDiffieHellmanPublic;
```

注：クライアント側からの Certificate に、既に適切な鍵が記載されている場合は implicit、鍵を送信する必要があるときは explicit

（10）CertificateVerify

ClientHello からこのメッセージの直前で送受信されたメッセージ全部に署名をしたものを、このメッセージとして送信する。このメッセージの構造体は次のようになる。

```
struct {
    Signature  signature;
} CertificateVerify;
```

signature の内容は 1.2.6,1.2.7.節参照のこと。

（11）Finished

このメッセージはサーバ、クライアントとも ChangeCipherSpec の直後に送信され、Handshake プロトコルの手続きが完了したことを示す。

構造体は 1.2.8 節参照。

1.1.5. Alert プロトコル

Alert プロトコルは、SSL（または TLS）で、何らかの異常状態が起きたときにその内容と重大さを相手に通知するためのものである。このプロトコルが送受信されたときは、その重大さに応じた異常手続きを行うことになる。このプロトコルの構造体を以下に示す。

```
準備：  enum {warning(1), fatal(2), (255)} AlertLevel;
構造体：struct {
            Alertlevel  level;
            AlertDescription  description;
        } Alert;
```

level は、この Alert の重大さを示し、fatal の場合は直ちにセッションを閉じることになる。warning の場合はクライアント・サーバの裁量で手続きを決定する。

description は Alert の内容を示し、これは 1.2.10 節に書かれているメッセージが用意されている。このうち、close_notify メッセージは、このメッセージを投げたコネクションでこれ以上の送信を行わないことを通知するものであり、それ以外のメッセージがエラー警告を示すメッセージとなる。

1.1.6. ChangeCipherSpec プロトコル

ChangeCipherSpec プロトコルは Handshake プロトコルでの手続き中に送信される。これは Handshake プロトコルでネゴシエーションがとれたパラメータをもとに CipherSpec（または SecurityParameters）などが設定されるので、このメッセージを送信することにより以後の通信はこのパラメータを使った SSL（または TLS）通信を行うことになる。この構造体は以下に示す。

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

基本的に change_cipher_spec という値のみを持つ構造体で、これを送信したあとはネゴシエーションの取れたパラメータを使用して通信を行う。

1.1.7. Application Data プロトコル

Application Data プロトコルは、これまで示したプロトコル以外のものすべてで、上位レイヤから送られるアプリケーションなどのデータを送るものである。これは特別な構造体を持つものでなく、上位からのデータをそのまま Record Layer プロトコルに送ることになっている。

1.2. SSL 3.0／TLS 1.0 の比較

SSL3.0 SSL3.0 と TLS1.0 の主な相違点としては、共有する鍵や初期値の生成関数が異なる。図 1.2.1 に SSL3.0 における鍵生成の生成シーケンスの概要を示す。鍵共有メカニズムを用いて作成された鍵（pre_master_secret）は、一方向性関数により Master_secret が作成され、同様の一方向性関数を用いて、key_block を作成する。Key_block のビット列は、

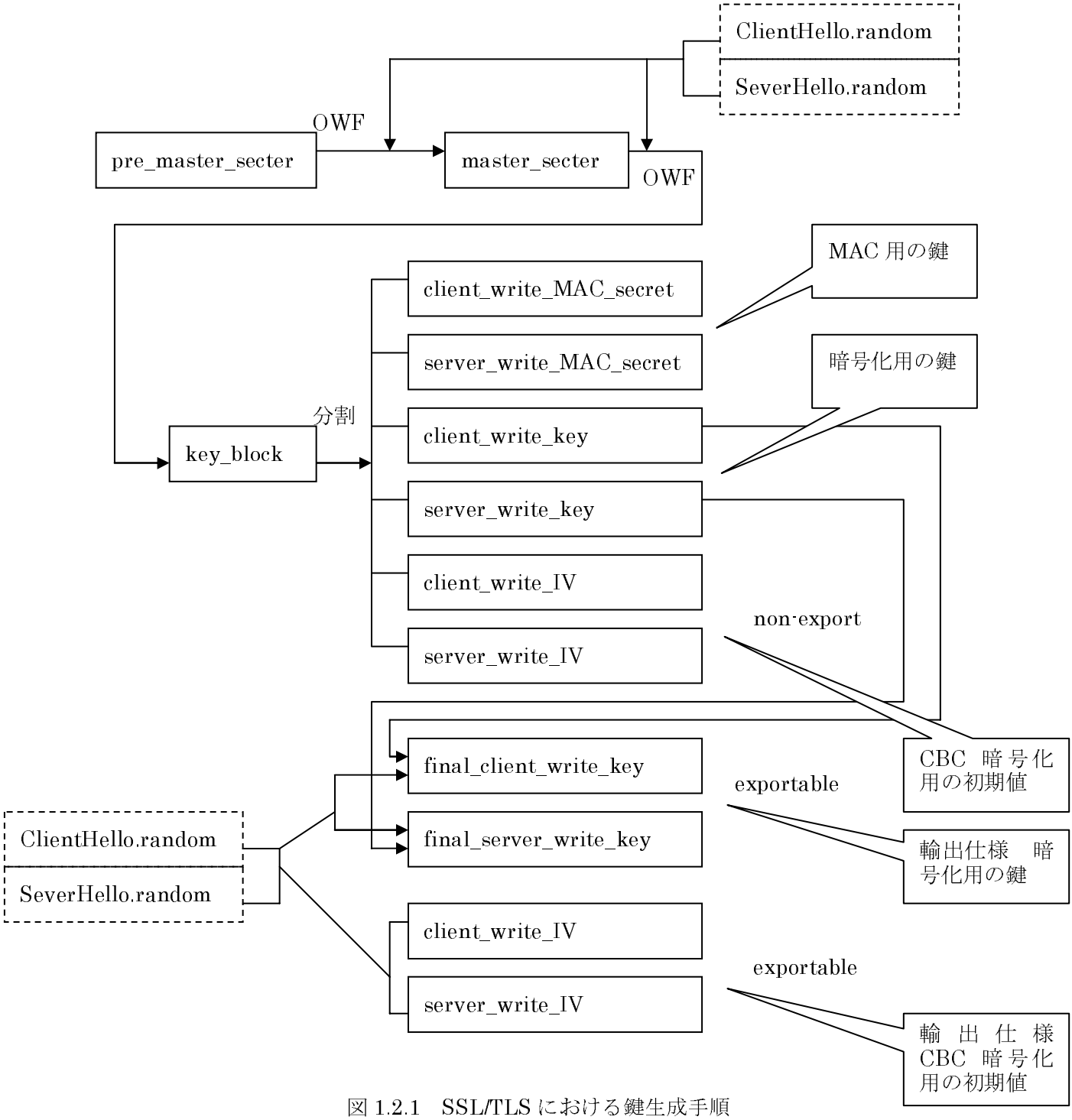


図 1.2.1 SSL/TLS における鍵生成手順

６つの MAC 鍵、暗号鍵、CBC 初期値を生成する。ただし、輸出仕様においては、暗号化鍵および初期値の生成が北米仕様（non-export）と異なる点に留意する必要がある。以下、各相違点の詳細について述べる。

注：文中の“+”は文字の連接、“XOR”は排他論理和を表す。

1.2.1. MAC（Message Authentication Code：メッセージ認証符号）

SSL MAC = **hash**(MAC_write_secret + pad_2 +
hash(MAC_write_secret + pad_1 + seq_num +
SSLCompressed.type + SSLCompressed.length +
SSLCompressed.fragment))

TLS MAC = HMAC_**hash**(MAC_write_secret ,
seq_num + TLSCompressed.type +
TLSCompressed.version + TLSCompressed.length +
TLSCompressed.fragment))

[解説]

SSL/TLS での MAC の機能は、Record Layer で実際に送信するデータ・メッセージ（M）とそれにハッシュ関数をかけたもの（C）を送信し、受信側で受信したデータ・メッセージ（M）にハッシュ関数をかけ、これと受信したメッセージ（C）と比較することで、改ざんがあったことを発見するものである。

SSL/TLS とも秘密鍵に相当する MAC_write_secret を用いてデータ・メッセージをハッシュ化している。SSL の MAC の計算は NMAC のスキームと同一の方式となっており、NMAC と同等の安全性があるものと考えられる[27]。

注：SSL における MAC_write_secret は、後述の 1.2.3 節で示されている key_block の write_MAC_secret に相当する。

TLS では RFC2104 に規定された HMAC アルゴリズムを使用している。HMAC は NMAC アルゴリズムから考え出されたもので[27]、HMAC の安全性は使用しているハッシュ関数の安全性に依存しており、そのハッシュ関数が安全であれば HMAC も安全であること、birthday attack への耐性があることが RFC で述べられている。

注：HMAC

HMAC_**hash**(secret , message) = **hash**(((secret + pad) XOR opad) +
hash(((secret + pad) XOR ipad) + message))

pad: 指定バイト列を作るように secret の終わりまでゼロを追加する

ipad: 0x36 を指定バイト列分繰り返した文字列
opad: 0x5C を指定バイト列分繰り返した文字列

1.2.2. master_secret の計算

SSL 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))

TLS master_secret = PRF(pre_master_secret , "master secret" ,
ClientHello.random + ServerHello.random)

[解説]

master_secret は、後述する鍵の計算や Handshake プロトコルなどで使用される。この値は SSL/TLS 共に、pre_master_secret、ClientHello.random、ServerHello.random の値から、48 バイト列の数値が導出される。導出関数において相違点がある。

SSL は上記 3 つの値および「A」「BB」「CCC」の文字列を用い、ハッシュ関数 MD5 または SHA で計算したものを連接することで、master_secret を導出している。

TLS は新たに定義した疑似乱数関数 PRF を定義し、上記 3 つの値および「master secret」の文字列から master_secret を導出している。PRF を説明するのに、まず次の関数を定義する。

P_hash(a, b) = HMAC_hash(a, A(1) + b) + HMAC_hash(a, A(2) + b) +・・・・

ただし A(0) = b

A(i) = HMAC_hash(a, A(i-1))

P_hash は必要な長さのデータを生成するまで計算する。必要な長さの出力が得られないときは、その長さを超えるまで計算し、初めて超えた出力値の下位バイト列を切り捨てる。

この P_hash を用いて PRF は次のように定義される

PRF(secret , label , seed) =P_MD5(S1, label + seed) XOR
P_SHA(S2, label + seed)

ただし secret = S1 + S2 かつ S1 と S2 は同バイト長

TLS では 48 バイト列の数値を作成するので、PRF に用いられる P_MD5、P_SHA の計算は、それぞれ HMAC 処理を 3 回繰り返している。

1.2.3. key_block 計算

SSL 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))

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

[解説]

key_block は次の 6 種類の値を得るために使われる値である。

client_write_MAC_secret[CipherSpec.hash_size / SecurityParameters.hash_size]
server_write_MAC_secret[CipherSpec.hash_size / SecurityParameters.hash_size]
client_write_key[CipherSpec.key_material / SecurityParameters. key_material_length]
server_write_key[CipherSpec.key_material / SecurityParameters. key_material_length]
client_write_IV[CipherSpec.IV_size] ‥ non-export
server_write_IV[CipherSpec.IV_size] ‥non-export

注：上記の[ ]内の表記は SSL の場合の、それぞれの値のバイト長を表す。TLS の場合は CipherSpec → SecurityParameters、key_material → key_material_length と読み替えること。
※SecurityParameters.IV_size というメンバは存在しない（1.2.9 節参照）
また最後二つは non-export cipher のためにのみ生成される。Exportable の場合は 1.2.5 節を参照のこと。

key_block を上記 6 種類の値になるように仕切り、余分な部分を切り捨てる。この key_block の導出関数に相違点がある。

SSL は master_secret、ServerHello.random、ClientHello.random および「A」「BB」「CCC」‥‥（以下必要分だけ続く）の文字列を用い、ハッシュ関数 MD5 または SHA で計算したものを連接することで、key_block を導出している。

TLS は疑似乱数関数 PRF を用いて、master_secret、SecurityParameters.server_random、SecurityParameters.client_random および「key expansion」の文字列から key_block を導出している。

1.2.4. final_client_write_key および final_server_write_key の計算

SSL final_client_write_key = MD5(client_write_key + ClientHello.random + ServerHello.random)
final_server_write_key = MD5(server_write_key + ServerHello.random + ClientHello.random)

TLS final_client_write_key = PRF(client_write_key, “client write key”, SecurityParameters.client_random + SecurityParameters.server_random)
final_server_write_key = PRF(server_write_key , “server write key” , SecurityParameters.client_random + SecurityParameters.server_random)

[解説]

本計算は、exportable が指定されたときにのみ付加的に処理される。final_client_write_key、final_server_write_key は実際に暗号鍵として使用される。これらの値を計算する導出関数に相違点がある。

SSL は client_write_key（または server_write_key）、ServerHello.random 、ClientHello.random を用い、ハッシュ関数 MD5 で計算し、それぞれの値を導出している。

TLS は疑似乱数関数 PRF を用いて、client_write_key（または server_write_key)、SecurityParameters.server_random、SecurityParameters.client_random および「client write key（または server write key)」の文字列からそれぞれの値を導出している。

1.2.5. Exportable encryption algorithms 使用時の client_write_IV および server_write_IV の計算

SSL client_write_IV = MD5(ClientHello.random + ServerHello.random)
server_write_IV = MD5(ServerHello.random + ClientHello.random)

TLS iv_block = PRF("" , "IV block" , SecurityParameters.client_random + SecurityParameters.server_random)
iv_block = client_write_IV + server_write_IV
ただし client_write_IV , server_write_IV 共に
SecurityParameters.IV_size バイト長

[解説]

Exportable encryption algorithms 使用時は、client_write_IV および server_write_IV を key_block とは別に計算する。本 IV の計算には、master_secret は用いられていない。このため、セッションをタッピングする第 3 者が、IV を計算することが可能である点で、non-export より安全性が劣ると考えられる。これらの値を計算する導出関数に相違点がある。

SSL は ServerHello.random、ClientHello.random を用い、ハッシュ関数 MD5 で計算し、それぞれの値を導出している。

TLS は疑似乱数関数 PRF を用いて、SecurityParameters.server_random 、SecurityParameters.client_random および「IV block」の文字列からからそれぞれの値を導出している。このとき関数 PRF の第一入力値はゼロバイト長の文字を表す。

1.2.6. ServerKeyExchange メッセージの Signature の構造体

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

[解説]

ServerKeyExchange メッセージにおいて、署名アルゴリズムを選択する個所がある。このアルゴリズムの違いにより

```
md5_hash = MD5(ClientHello.random + ServerHello.random +
                ServerParams);
sha_hash = SHA(ClientHello.random + ServerHello.random +
                ServerParams);
enum {anonymous, rsa, dsa} SignatureAlgorithm;
```

を使用して署名を行うが、この構造体に相違点がある。

SSL では digitally-signed が最上位の構造であるため、署名アルゴリズムが anonymous の際には、空の値に署名を行う形になり。署名アルゴリズム上問題になる危険がある。

この問題を回避するため、TLS ではまず署名アルゴリズムに依存して digitally-signed の要否を指定できるようになっており、anonymous の場合に空の値に署名を行わない。

1.2.7. CertificateVerify メッセージで使用されるハッシュ計算

SSL CertificateVerify.signature.md5_hash
= MD5(master_secret + pad_2 +
MD5(handshake_messages + master_secret + pad_1))
CertificateVerify.signature.sha_hash
= SHA(master_secret + pad_2 +
SHA(handshake_messages + master_secret + pad_1))

TLS CertificateVerify.signature.md5_hash = MD5(handshake_messages)
CertificateVerify.signature.sha_hash = SHA(handshake_messages)

[解説]

CertificateVerify メッセージは、1.2.6 節の signature 構造に基づいて署名され、クライアント証明書の検証を行うのに使用される。ClientHello からこのメッセージを送信するまでの（CertificateVerify を除く）送受信されたすべての Handshake プロトコルメッセージを連鎖したものを handshake_messages とし、このメッセージをハッシュ関数にかけて署名を行う。このときハッシュ関数での計算方法に相違点がある。

SSL は handshake_messages、mastersecret（および pad_1、pad_2）を用いて上記の計算を行う。

TLS では、得られた値に署名を施した物が CertificateVerify メッセージとなることから、master_secret を連接する必要がないことから、単純に handshake_messages のみをハッシュ関数にかける演算に変更されている。

1.2.8. Finished メッセージ

SSL 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))
ただし Sender = Sender.client（クライアント送信時） または
Sender.server（サーバ送信時）

TLS struct {
opaque verify_data[12]
} Finished;
verify_data = PRF(master_secret , finished_label ,
MD5(handshake_messages) + SHA(handshake_messages))
ただし finished_label = “client finished”（クライアント送信時） または
“server finished”（サーバ送信時）

[解説]

Finished メッセージは、鍵交換と認証処理が成功したことを確認する。ClientHello からこのメッセージを送信するまでの（Finished を除く）送受信されたすべての Handshake プロトコルメッセージを連結したものを handshake_messages とし、このメッセージから Finished メッセージを導出する。このときメッセージに使用される構造体メンバの個数および計算に使用する導出関数に相違点がある。

SSL は 2 つのメンバを持ち、handshake_messages、mastersecret、Sender（および pad1、pad2）を用いて上記の計算を行う。TLS は 1 つのメンバを持ち、handshake_messages、mastersecret、finished_label を用いて上記の計算を行う。

1.2.9. CipherSpec（SSL）と SecurityParameters（TLS）

```
SSL     enum {stream, block} CipherType;
        enum {true, false} IsExportable;          /* Exportable な暗号方式か否か */
      * enum {null, rc4, rc2, des, 3des, des40, fortezza } BulkCipherAlgorithm;
        enum {null, md5, sha} MAC  Algorithm;
          struct {
                BulkCipherAlgorithm   bulk_cipher_algorithm;
                MACAlgorithm   mac_algorithm;
                CipherType   cipher_type;
                IsExportable   is_exportable;
                uint8   hash_size;
      *         uint8   key_material;
      *         uint8   IV_size;
          } CipherSpec;

TLS     * enum {null(0), (255)} CompressionMethod;
        * enum {server, client} ConnectionEnd;
        * enum {null, rc4, rc2, des, 3des, des40, idea} BulkCipherAlgorithm;
          enum {stream, block} CipherType;
          enum {true, false} IsExportable;          /* Exportable な暗号方式か否か */
          enum {null, md5, sha} MACAlgorithm;
            struct {
                ConnectionEnd   entity;
                BulkCipherAlgorithm   bulk_cipher_algorithm;
                CipherType   cipher_type;
                uint8   key_size;
      *         uint8   key_material_length;
                IsExportable   is_exportable;
                MACAlgorithm   mac_algorithm;
                uint8   hash_size;
                CompressionMethod   compression_algorithm;
      *         opque   master_secret[48];
      *         opaque   client_random[32];
      *         opaque   server_random
            } SecurityParameters;
```

[解説]

CipherSpec と SecurityParameters はそれぞれ SSL/TLS による通信において使用される暗号の方式や鍵計算用数値を規定するものである。それぞれの構造体において、相違点がある（またはどちらか一方のみ規定されている）メンバを上記に示す（相違点のあるところに※マークが施されている）。

BulkCipherAlgorithm で規定される暗号方式のうち、SSL は値「fortezza」が用意されているが、TLS では fortezza をサポートしていないのでこの値は存在しない。TLS では暗号方式 IDEA に予め対応できるように、値「idea」が用意されている。

SSL での key_material と TLS の key_material_length は同じ用途で用いられている。（1.2.3 節参照）

TLS のみ設定されている値のうち、ConnectionEnd はクライアントかサーバかを示しており、SSL にはこれに相当する物はない。master_secret は SSL でも計算され、用いられている（1.2.2 節参照）。client_random、server_random 、compression_algorithm に関しても、SSL では ClientHello、ServerHello で random の数値や CompressionAlgorithm が規定されている。

SSL のみ IV_size の項目があり、TLS には規定されていない。ただし TLS でも IV_size に依存する項目がある（1.2.3 節参照）。

1.2.10. Alert プロトコル

両方式　unexpected_message(10), bad_record_mac(20), decompression_failure(30),
handshake_failure(40), illegal_parameter(47)（以上、常に fatal）
close_notify(0), bad_certificate(42), unsupported_certificate(43),
certificate_revoked(44), certificate_expired(45),
certificate_unknown(46)（warning または fatal）

SSL　no_certificate(41)（warning または fatal）

TLS　decryption_failed(21), record_overflow(22), unknown_ca(48),
access_denied(49), decode_error(50), export_restriction(60),
protocol_version(70), insufficient_security(71),
internal_error(80)　（以上、常に fatal）
no_renegotiation(100)（常に warning）
user_canceled(90)　（一般的に warning）
decrypt_error(51)（warning または fatal）

[解説]

SSL/TLS においてなんらかのトラブルが発生したときに、それを通知するものとして Alert プロトコルがある。これは 1.1.5 節で示す構造体を持ち、どういった障害が発生したかを warning レベル・fatal レベルと合わせて示してある。

上記で示した相違点は 1.1.5 節における AlertDescription に関する物である。SSL で no_certificate とされていたものを、TLS ではさらに細分化してトラブルの原因を特定しやすくなっている。このとき AlertLevel が fatal となっているものは、常にコネクションが閉鎖される。warning レベルのものは、コネクションを閉鎖するかどうかはクライアントまたはサーバの裁量で決めてよい。また、常に AlertLevel が決まっているものを除いて、AlertLevel が示されていない Alert に関しては、クライアントまたはサーバの裁量で fatal または warning を決定できる。

1.2.11. その他

SSL ・Fortezza サポート
・IDEA の記述なし
・Protocol Version = 3.0

TLS ・Fortezza 非サポート
・IDEA の記述あり
・Protocol Version = 3.1

[解説]

その他で SSL と TLS とでは上記の相違点がある。

暗号方式 Fortezza は SSL まではサポートされていたが、TLS ではサポートされていない。

暗号方式 IDEA は、SSL の Draft では記述がないが、TLS の RFC では記述されている個所がある（1.2.9 節参照）。ただし、これは Draft／RFC に記述されているかされていないかだけの問題であり、SSL で IDEA がサポートされないわけではない。例えば Apache-SSL では、規定されている方式以外の暗号方式も後から組み込むことができるという SSL プロトコルの性質を利用して、IDEA をサポートしている。

Protocol Version は SSL では 3.0、TLS では 3.1 となる。Handshake プロトコルにおいても、ClientHello、ServerHello で示されるバージョンの上限は上記の値となる。

1.3. TLS 拡張作業の概要

TLS 拡張作業の推移を図 1.3.1 に示す。

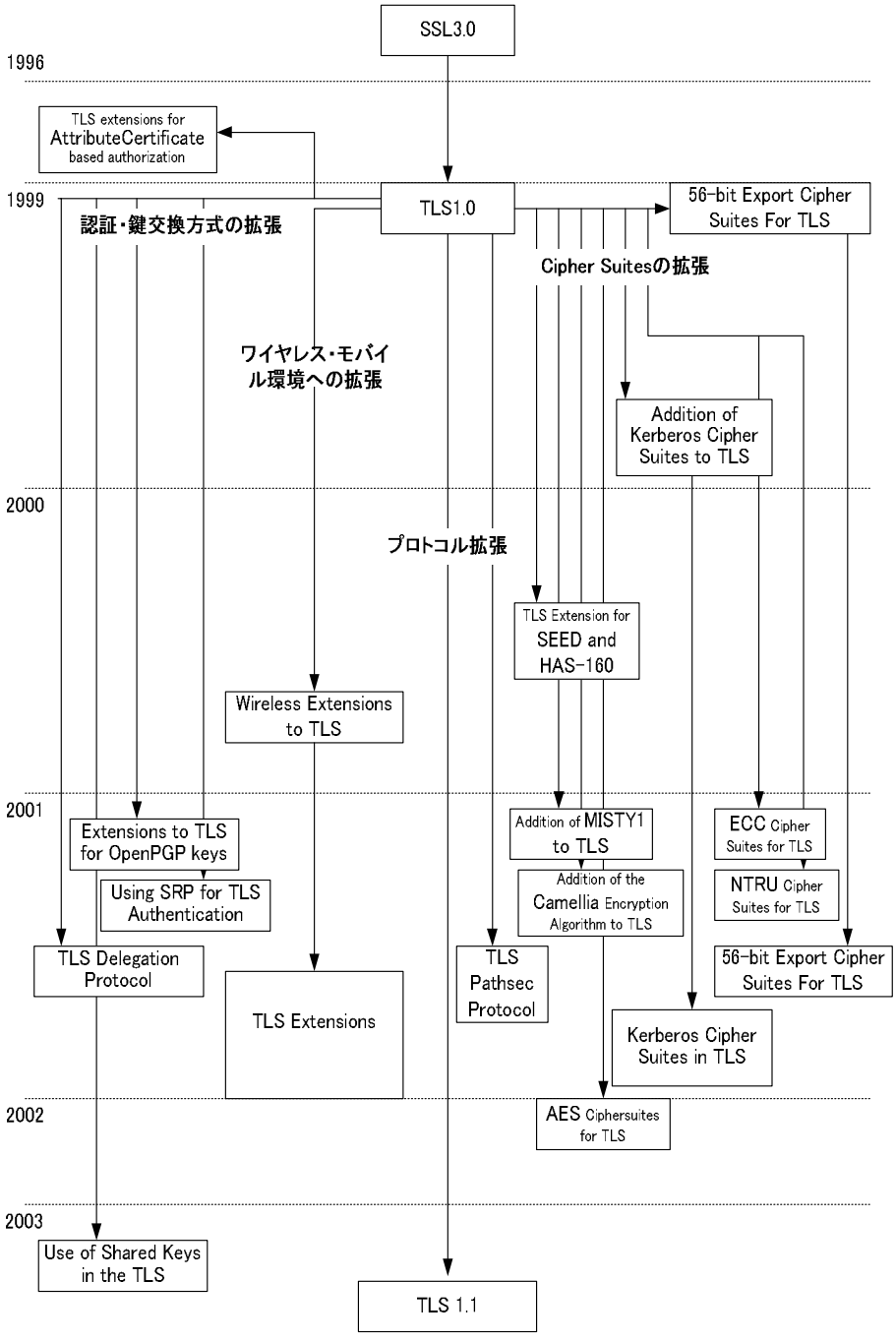


図 1.3.1　TLS 拡張作業の推移

TLS の拡張作業は、A）ワイヤレス・モバイル環境への対応、B）新規暗号アルゴリズムの追加、C）認証・鍵交換手法のバリエーション追加、D）プロトコルの拡張に大別される。TLS 自身については、現在も改訂作業を継続しており、2002 年 2 月にドラフトが発表される予定である。以下、各拡張の内容について説明する。

1.3.1. 機能拡張

[draft-ietf-tls-extensions-02]
TLS Extensions
発行日: 2001 年 12 月

さまざまな TLS 拡張方法のまとめたものである。本ドラフトは、draft-ietf-tls-wireless-00 をベースにして内容を拡張したものであり、携帯電話などのモバイル環境、ワイヤレス通信での使用や、仮想ホストへの SSL アクセス、OCSP(Online Certificate Status Protocol)を使用したサーバ証明書失効情報要求などを規定している。以下にその詳細を記述する。

（1） Ｈｅｌｌｏメッセージの拡張
以下の様々な拡張によって必要となるパラメータを交換するために、Client Hello 及び Server Hello をそれぞれ以下のように拡張する。

・ Client Hello

```
struct {
    ProtocolVersion client_version;
Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
    Extension client_hello_extension_list<0..2^16-1>;
} ClientHello;
```

・ Server Hello

```
struct {
    ProtocolVersion server_version;
Random random;
SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    Extension server_hello_extension_list<0..2^16 -1>;
} ServerHello;
```

すなわち、各 Hello メッセージ中に、拡張パラメータの記述領域 Extension client_hello_extension_list 及び Extension server_hello_extension_list を設ける。拡張パラメータは以下のように記述する。

```
struct {
    ExtensionType extensionType;
opaque extension_data<0..2^16 -1>;
} Extension;
```

上記パラメータ内の Extension Type は以下のように記述する。

```
enum {
    server_name(0), max_fragment_size(1),
    client_certificate_url(2), trusted_ca_keys(3),
    truncated_hmac(4), status_request(5), (65535)
} ExtensionType;
```

（2） ハンドシェイクプロトコル

ハンドシェイクプロトコルに、後述の（5）に対応するための CertificateURL と、（8）に対応するための CertificateStatus という 2 つのプロトコルを追加する。従って、Handshake の構造体は以下のようになる。

```
    enum {
        hello_request(0), client_hello(1), server_hello(2),
        certificate(11), server_key_exchange (12),
        certificate_request(13), server_hello_done(14),
        certificate_verify(15), client_key_exchange( 16),
        finished(20), certificate_url(21), certificate_status(22),
        (255)
} HandshakeType;

    struct {
        HandshakeType msg_type;    /* handshake type */
    uint24 length;          /* bytes in message */
         select (HandshakeType) {
      case hello_request:          HelloRequest;
      case client_hello:           ClientHello;
      case server_hello:           ServerHello;
      case certificate:            Certificate;
          case server_key_exchange: ServerKeyExchange;
      case certificate_request: CertificateRequest;
      case server_hello_done:     ServerHelloDone;
      case certificate_verify:    CertificateVerify;
      case cli          ent_key_exchange: ClientKeyExchange;
              case  finished:                 Finished;
      case certificate_url:       CertificateURL;
      case certificate_status:    CertificateStatus;
    } body;
} Handshake;
```

（３） Server name Indication

仮想ホストへのアクセスを可能にするために、client から ServerName を送信できるように拡張する。Servername は、client hello メッセージ中の Extension Type を server_name とし、extension_data フィールドに以下のような ServerNameList を含める。これにより、アドレスとサーバ名とでアクセス対象を特定できるため、１つのアドレスに対して複数の Server を稼働させることが可能となる。

```
        struct {
            NameType name_type;
             select (name_type) {
          case host_name: HostName;
            }
    } ServerName;

        enum {
       host_name(0), (255)
    } NameType;

    opaque HostName<1..2^16      -1>;

        struct {
            ServerName server_name_list<1..2^16 -1>
    } ServerNameList;
```

（４） Maximum Fragment Size Negotiation

ナローバンドなワイヤレス環境に対応するため、送受信データのフラグメントサイズの最大値を可変とする。レコードサイズは、Client Hello メッセージ中の Extension Type を max_fragment_size とし、extension_data に以下のように記述する。

```
        enum{
            2^9(1), 2^10(2), 2^11(3), 2^12(4), (255)
    } MaxFragmentSize;
```

（５） Client Certificate URLs

モバイル環境（WPKI:Wireless PKI）等への適用を考慮して、Client の証明書をインターネット上の Server から取得できるように拡張が行う。Client Hello メッセージ中の Extension Type を client_certificate_url とし、extension_data は使用しない。また、Client は、Certificate に代わりに CertificateURL メッセージを送信する。Server は、この url 先にアクセスし Client の証明書を取得する。CertificateURL のメッセージフォーマットは、以下の通り

```
    struct {
        URLAndHash url_and_hash_list<1..2^16 -1>;
} CertificateURL;

    struct {
   opaque URL<1..2^16       -1>;
        CertHash certificate_hash;
} URLAndHash;

opaque CertHash<     0..20>;
```

（６） Trusted CA Indication

携帯端末などのモバイル環境では所有する CA のルート鍵が限定されるため、Client が検証可能なルート鍵を Server に通知する必要がある。従って、Client Hello メッセージ中の Extension Type を trusted_ca_keys とし、extension_data フィールドに以下のメッセージを含め送信する。Server は、送られてきたリストに対応する証明書を返信するか、エラーを通知する。

```
    struct {
         TrustedAuthority trusted_authorities_list<0..2^16-1>;
} TrustedAuthorities;

    struct {
        IdentifierType identifier_type;
         select (identifier_type) {
      case pre_agreed: struct {};
      case key_ha          sh_sha: KeyHash;
      case x509_name: DistinguishedName;
```

```
        case cert_hash: CertHash;
    } Identifier;
} TrustedAuthority;

    enum { pre_agreed(0), key_hash_sha(1), x509_name(2), cert_hash(3),
        (255)}
        IdentifierType;

opaque DistinguishedName<1..2^16    -1>;

opaque KeyHash[20];
```

（７） Truncated HMAC

モバイル環境等での利用を考慮して、HMAC 値を 80bit に縮小できるよう拡張する。Truncated HMAC を使用する場合には、Client Hello メッセージ中の Extension Type を truncated_hmac とし、extension_data は空とする。

（８） Certificate Status Request

Client が OCSP を用いて、Server の証明書の失効状態を確認できるように拡張を行う。Server は、Client のリクエストに対して、OCSP を取得し、証明書と共に返信する。OCSP のパラメータは、Client Hello メッセージ中の Extension Type を status_request とし、extension_data に以下のような CertificateStatusRequest の内容を記述する。

```
    struct {
        CertificateStatusType status_type;
        select (status_type) {
      case ocsp: OCSPStatusRequest;
        }
} CertificateStatusRequest;

    enum { ocsp(1), 255) } CertificateStatusType;

    struct {
        Respond erID responder_id_list<0..2^16-1>;
        Extensions   request_extensions;
} OCSPStatusRequest;
```

```
opaque ResponderID<1..2^16    -1>;
opaque Extensions<0..2^16     -1>;
```

これに対し Server は、Server Hello として、Extension Type を status_request とし、extension_data を空にしたメッセージを返す。また、Server は、Certificate メッセージの後に CertificateStatus メッセージを送信する。CertificateStatus は、以下のように記述する。

```
        struct {
            CertificateStatusType status_type;
             select (status_type) {
          case ocsp: OCSPResponse ocsp_response;
             }
    } CertificateStatus;

    opaque OCSPResponse<1..2^24     -1>;
```

[draft-ietf-tls-rfc2246-bis-05]
The TLS Protocol Version 1.1
発行日: 2003 年 6 月

現在策定中。CBC モードのサイドチャネル攻撃に対する対策、PKCS#1（v.1.5）に対する適用的選択暗号文攻撃に対する対策、D.Brumley らのタイミング攻撃に対する対策、非推奨暗号の追加（40bits の共通鍵、512bits の署名鍵）などのセキュリティ向上を意図した改定作業が中心となっている。

1.3.2. 認証方式の拡張

[draft-ietf-tls-attr-cert-00]
TLS extensions for AttributeCertificate based authorization
発行日: 1998 年 2 月

ハンドシェイクにおいて、属性証明書(Attribute Certificate)をベースにした認証を行うための、TLS の拡張方法を記述している。CertificateRequest を ACRequest に、CertificateResponse を ACInfo に置き換えることで、クライアントの Attribute Certificate を送信する。

[draft-ietf-tls-delegation-01]
TLS Delegation Protocol
発行日: 2001 年 7 月

TLS を使用して、代理証明書(Proxy Certificate)または Kerberos 5 forwardable ticket の委譲(delegation)を行うためのプロトコル規定と、そのための TLS の拡張方法を記述している。 TLS Record layer の上位に TLS Delegation Protocol を定義する。

[draft-ietf-tls-kerb-01]
Kerberos Cipher Suites in Transport Layer Security (TLS)
発行日: 2001 年 11 月
クライアントからサーバへの Kerberos チケット委譲(delegation)をサポートするよう RFC2712 をアップデートしたものである。CertificateRequest メッセージにサーバは、realm name または、forwarded ticket などの attribute を含める。それに対して、クライアントの Certificate メッセージには、delegated credential を配送するための Kerberos KRB-CRED メッセージを含める。

1.3.2. 鍵共有方式の拡張

[draft-ietf-tls-openpgp-01]
Extensions to TLS for OpenPGP keys
発行日: 2001 年 3 月

OpenPGP で使用されている証明書、公開鍵アルゴリズム、共通暗号アルゴリズム、ハッシュアルゴリズム、信頼モデルをサポートするよう TLS を拡張する。Certificate において、公開鍵証明書内に OpenPGP 鍵を含めるか、鍵 ID を送信することにより鍵共有を実現する。cipher suites の記述方法は、以下のとおり

```
CipherSuite TLS_PGP_DHE_DSS_WITH_CAST_CBC_SHA       = { 0x01, 0x01 };
CipherSuite TLS_PGP_DHE_DSS_WITH_IDEA_CBC_SHA       = { 0x01, 0x02 };
CipherSuite TLS_PGP_DHE_DSS_WITH_3DES_EDE_CBC_SHA   = { 0x01, 0x03 };
CipherSuite TLS_PGP_DHE_DSS_WITH_CAST_CBC_RMD       = { 0x01 , 0x04 };
CipherSuite TLS_PGP_DHE_DSS_WITH_IDEA_CBC_RMD       = { 0x01, 0x05 };
CipherSuite TLS_PGP_DHE_DSS_WITH_3DES_EDE_CBC_RMD = { 0x01, 0x06 };
CipherSuite TLS_PGP_DHE_RSA_WITH_CAST_CBC_SHA       = { 0x01, 0x10 };
CipherSuite TLS_PGP_RSA_WITH_CAST_CBC_SHA           = { 0x01, 0x20 };
CipherSuite TLS_PGP_RSA_WITH_IDEA_CBC_SHA           = { 0x01, 0x21 };
CipherSuite TLS_PGP_RSA_WITH_3DES_EDE_CBC_SHA       = { 0x01, 0x22 };
CipherSuite TLS_PGP_RSA_WITH_CAST_CBC_RMD       = { 0       x01, 0x23 };
CipherSuite TLS_PGP_RSA_WITH_IDEA_CBC_RMD       = { 0x01, 0x24 };
CipherSuite TLS_PGP_RSA_WITH_3DES_EDE_CBC_RMD       = { 0x01, 0x25 };
CipherSuite TLS_PGP_DSA_WITH_NULL_SHA               = { 0x01, 0xF0 };
```

[draft-ietf-tls-srp-01]
Using SRP for TLS Authentication
発行日: 2001 年 6 月

SRP(Secure Remote Password)に基づいた認証を TLS において行うための拡張方法を記述する。従来のユーザ ID/パスワードによる認証を利用しているアプリケーションにおいて、安全にこれらを交換し、さらに TLS を利用して通信データの保護を行う。Client Hello 中にユーザ名と MD 値のリストを送信し、Server Hello で、サーバが選択した MD 値を返す。この値と、SRP パスワードファイルを元に、pre-master 鍵は計算される。cipher suites の記述方法は、以下のとおり

CipherSuite TLS_SRP_WITH_3DES_EDE_CBC_SHA = { 0x00,0x5B };
CipherSuite TLS_SRP_WITH_RC4_128_SHA = { 0x00,0x5C };
CipherSuite TLS_SRP_WITH_IDEA_CBC_SHA = { 0x 00,0x5D };
CipherSuite TLS_SRP_WITH_3DES_EDE_CBC_MD5 = { 0x00,0x5E };
CipherSuite TLS_SRP_WITH_RC4_128_MD5 = { 0x00,0x5F };
CipherSuite TLS_SRP_WITH_IDEA_CBC_MD5 = { 0x00,0x60 };

[draft-ietf-tls-sharedkeys-02]
Using SRP for TLS Authentication
発行日: 2003 年 10 月

事前共有鍵を使用することで公開鍵暗号処理を不要とする方式について記述している。具体的には、事前共有鍵などの秘密共有情報から PRF(pseudorandom function)を使用して、master secret を生成する方法を以下のように定めている。

master_secret = PRF(shared_secret, "shared secret", "")[0..47]

1.3.3. 暗号方式の拡張

[draft-ietf-tls-56-bit-ciphersuites-01]
56-bit Export Cipher Suites For TLS
発行日: 2001 年 7 月

56 ビット暗号を使用した cipher suites を、TLS に追加する。先のドラフトである draft-ietf-tls-56-bit-ciphersuites-00 を 2 年半ぶりにアップデートしたもので、cipher suites の追加、削除を行っている。 追加された cipher suites は、以下のとおり

CipherSuite TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA = { 0x00,0x62 };
CipherSuite TLS_RSA_EXPORT1024_WITH_RC4_56_SHA = { 0x00,0x64 };
CipherSuite TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA = { 0x00,0x63 };
CipherSuite TLS_DHE_DSS_EXPORT1024_WITH_RC4_56_SHA = { 0x00,0x65 };
CipherSuite TLS_DHE_DSS_WITH_RC4_128_SHA = { 0x00,0x66 };

[draft-ietf-tls-seedhas-00]
TLS Extension for SEED and HAS-160
発行日: 2000 年 7 月

韓国の TTA が制定したブロック暗号 SEED とハッシュアルゴリズム HAS-160 を、TLS の cipher suites に追加する。cipher suites の記述方法は、以下のとおり

CipherSuite TLS_RSA_WITH_SEED_CBC_MD5 = { 0x00, 0x2C };
CipherSuite TLS_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x2D };
CipherSuite TLS_RSA_WITH_SEED_CBC_HAS160 = { 0x00, 0x2E };

[draft-ietf-tls-misty1-01]
Addition of MISTY1 to TLS
発行日: 2001 年 3 月

三菱電機が開発したブロック暗号 MISTY1 を、TLS の cipher suites に追加する。cipher suites の記述方式は、以下のとおり

CipherSuite TLS_RSA_WITH_MISTY1_CBC_SHA = { 0x00,0x3B };
CipherSuite TLS_DH_DSS_WITH_MISTY1_CBC_SHA = { 0x00,0x3C };
CipherSuite TLS_DH_RSA_WITH_MISTY1_CBC_SHA = { 0x00,0x3D };
CipherSuite TLS_DHE_DSS_WITH_MISTY1_CBC_SHA = { 0x00,0x3E };
CipherSuite TLS_DHE_RSA_WITH_MISTY1_CBC_SHA = { 0x00,0x3F };
CipherSuite TLS_DH_anon_WITH_MISTY1_CBC_SHA = { 0x00,0x40 };

[draft-ietf-tls-camellia-01]
Addition of the Camellia Encryption Algorithm to TLS
発行日: 2001 年 5 月

NTT と三菱電機で共同開発されたブロック暗号 Camellia を TLS の cipher suites に追加する。 cipher suites の記述方式は以下のとおり

CipherSuite TLS_RSA_WITH_CAMELLIA_128_CBC_SHA = { 0x00,0x41 };
CipherSuite TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA = { 0x00,0x42 };
CipherSuite TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA = { 0x00,0x43 };
CipherSuite TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA = { 0x00,0x44 };

CipherSuite TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA ={ 0x00,0x45 };
CipherSuite TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA ={ 0x00,0x46 };
CipherSuite TLS_RSA_WITH_CAMELLIA_256_CBC_SHA ={ 0x00,0x47 };
CipherSuite TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA ={ 0x00,0x48 };
CipherSuite TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA ={ 0x00,0x49 };
CipherSuite TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA ={ 0x00,0x4A };
CipherSuite TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA ={ 0x00,0x4B };
CipherSuite TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA ={ 0x00,0x4C };

[draft-ietf-tls-ciphersuite-06]
AES Ciphersuites for TLS
発行日: 2002 年 1 月

DES の後継である AES(Advanced Encryption Standard)を TLS の cipher suites に追加する。 cipher suites の記述方式は以下のとおり

CipherSuite TLS_RSA_WITH_AES_128_CBC_SHA ={ 0x00, 0x2F };
CipherSuite TLS_DH_DSS_WITH_AES_128_CBC_SHA ={ 0x00, 0x30 };
CipherSuite TLS_DH_RSA_WITH_AES_128_CBC_SHA ={ 0x00, 0x31 };
CipherSuite TLS_DHE_DSS_WITH_AES_128_CBC_SHA ={ 0x00, 0x32 };
CipherSuite TLS_DHE_RSA_WITH_AES_128_CBC_SHA ={ 0x00, 0x33 };
CipherSuite TLS_DH_anon_WITH_AES_128_CBC_SHA ={ 0x00, 0 x34 };
CipherSuite TLS_RSA_WITH_AES_256_CBC_SHA ={ 0x00, 0x35 };
CipherSuite TLS_DH_DSS_WITH_AES_256_CBC_SHA ={ 0x00, 0x36 };
CipherSuite TLS_DH_RSA_WITH_AES_256_CBC_SHA ={ 0x00, 0x37 };
CipherSuite TLS_DHE_DSS_WITH_AES_256_CBC_SHA ={ 0x00, 0x38 };
CipherSuite TLS_DHE_RSA_WITH_AES_256_CBC_SHA ={ 0x00, 0x39 };
CipherSuite TLS_DH_anon_WITH_AES_256_CBC_SHA ={ 0x00, 0x3A };

[draft-ietf-tls-ecc-01]
ECC Cipher Suites for TLS
発行日: 2001 年 3 月

楕円曲線暗号(Elliptic Curve Cryptography:ECC)を署名（ECDSA）及び鍵交換(ECDH)として使用するための、TLS のプロトコル拡張と cipher suites の追加を行う。cipher suites の記述方式は以下のとおり

CipherSuite TLS_ECDH_ECDSA_WITH_NULL_SHA = { 0x00, 0x47 }
CipherSuite TLS_ECDH_ECDSA_WITH_RC4_128_SHA = { 0x00, 0x48 }
CipherSuite TLS_ECDH_ECDSA_WITH_DES_CBC_SHA = { 0x00, 0x49 }
CipherSuite TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA = { 0x00, 0x4A }
CipherSuite TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA = { 0x00, 0x4B }
CipherSuite TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA = { 0x00, 0x4C }
CipherSuite TLS_ECDH_ECDSA_EXPORT_WITH_RC4_40_SHA = { 0x00, 0x4B }
CipherSuite TLS_ECDH_ECDSA_EXPORT_WITH_RC4_56_SHA = { 0x00, 0x4C }
CipherSuite TLS_ECDH_RSA_WITH_NULL_SHA = { 0x00, 0x4D }
CipherSuite TLS_ECDH_RSA_WITH_RC4_128_SHA = { 0x00, 0x4E }
CipherSuite TLS_ECDH_RSA_WITH_DES_CBC_SHA = { 0x00, 0x4F }
CipherSuite TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA = { 0x00, 0x50 }
CipherSuite TLS_ECDH_RSA_WITH_AES_128_CBC_SHA = { 0x00, 0x51 }
CipherSuite TLS_ECDH_RSA_WITH_AES_256_CBC_SHA = { 0x00, 0x52 }
CipherSuite TLS_ECDH_RSA_EXPORT_WITH_RC4_40_SHA = { 0x00, 0x53 }
CipherSuite TLS_ECDH_RSA_EXPORT_WITH_RC4_56_SHA = { 0x00, 0x54 }
CipherSuite TLS_ECDH_anon_NULL_WITH_SHA = { 0x00, 0x55 }
CipherSuite TLS_ECDH_anon_WITH_RC4_128_SHA = { 0x00, 0x56 }
CipherSuite TLS_ECDH_anon_WITH_DES_CBC_SHA = { 0x00, 0x57 }
CipherSuite TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA = { 0x00, 0x58 }
CipherSuite TLS_ECDH_anon_EXPORT_WITH_DES40_CBC_SHA = { 0x00, 0x59 }
CipherSuite TLS_ECDH_anon_EXPORT_WITH_RC4_40_SHA = { 0x00, 0x5A }

[draft-ietf-tls-ntru-00]
NTRU Cipher Suites for TLS
発行日: 2001 年 7 月

公開鍵暗号方式である NTRU 暗号アルゴリズムと、これを使用した NSS 署名アルゴリズムを鍵交換方式として使用するための TLS のプロトコル拡張と cipher suites の追加を行う。Cipher suites の記述方式は以下のとおり

```
CipherSuite TLS_NTRU_NSS_WITH_RC4_128_SHA          = { 0x00, 0x61 }
CipherSuite TLS_NTRU_NSS_WITH_3DES_EDE_CBC_SHA     = { 0x00, 0x62 }
CipherSuite TLS_NTRU_NSS_WITH_AES_128_CBC_SHA      = { 0x00, 0x63 }
CipherSuite TLS_NTRU_NSS_WITH_AES_256_CBC_SHA      = { 0x00, 0x64 }
CipherSuite TLS_NTRU_RSA_WITH_RC4_128_SHA          = { 0x00, 0x65 }
CipherSuite TLS_NTRU_RSA_WITH_3DES_EDE_CBC_SHA     = { 0x00, 0x66 }
CipherSuite TLS_NTRU_RSA_WITH_AES_128_CBC_SHA      = { 0x00, 0x67 }
CipherSuite TLS_NTRU_RSA_WITH_AES_256_CBC_SHA      = { 0x00, 0x68 }
```

1.3.4. その他

[draft-ietf-tls-pathsec-00]
TLS Pathsec Protocol
発行日: 2001 年 9 月

TLS セションを複数のサブセションに分割し、3 つのチャネルを使用することで、クライアントとサーバの間に複数の中継者（intermediary）の介在を可能とする。Client Hello 及び Server Hello の Pathsec パラメータを追加し、Alert メッセージについても追加を行う。

2. SSL における既存セキュリティホールとその対策について

本章では、SSL における既存のセキュリティホール（あるいは注意点）について、①暗号方式上、②プロトコル上、③実装上、④運用上の４つの角度から調査・分析を行った。ここで、①暗号方式、②プロトコル上のセキュリティホールとは、方式の仕様上の問題でセキュリティホールとなりえるもので、その中でも純粋に暗号方式に関連するものを①に、man-in-the-middle のような通信に関わる両エンティティが関与するプロトコル手順上の問題について②に分類した。また③は、機能実装の際、生じる問題で、主に不具合と考えられる。④については、SSL を動作させる際に指定可能な構成情報やオペレーションなどに関する問題に類する。これらの①～④については、その被害・影響の重大さを考えると重要度の優劣をつけるのは困難と考える。

2.1. 暗号方式上のセキュリティホールとその対策

2.1.1. DSA のセキュリティホール

2000 年 11 月　IEEE P1363 の working group の会合において、Bell Lab の Bleichenbacher は、DSA 署名方式において、各々のメッセージに対して乱数である秘密鍵を生成する際に偏りがあることを指摘した。本偏りは、ある特定の領域から選択する可能性は他の領域にくらべて高い出現確立となっていることが判明し、DSA の強度を著しく弱めることを指摘した。本来、生成されるいかなる乱数も、指定された領域内で均一(uniform)であるべきであるが、DSA の乱数生成なそうではなかった。Bleichenbacher は、DSS の Annex を解析している際に、本 flaw を発見するとともに、修正案を提示している (詳細は、以下を参考)。本 flaw は、直ちに問題となることはないが、将来的にメッセージの完全性に問題が生じる可能性があると指摘している。具体的に、アタックが成功するためには、アタックに要する時間（operation）は 2^64、メモリは 2^40、署名されたメッセージの数 2^22 が必要となる。

DSA は、DH の鍵交換において公開鍵を保証するための署名アルゴリズムとして利用されている。本 Flaw に対する修正（パッチ）が各 SSL 提供者から配布されている。

詳細なセキュリティホールおよびその対策は以下のとおり、

DSS(FIPS186-2)の Annex.3 において、k を求めるために、一方向性関数 G(t,c)を定義している。G は、Sha-1 あるいは DES を用いて生成され、160bit の乱数を出力する。

k を求めるために、具体的に、
k = G(t,c) mod q
を計算するが、q が 160bit の素数であり、G が 160bit の(すなわち、[0.. q-1]より若干大きい)ため、k に偏りが生じることがわかる。

したがって、FIPS186-2 では、Change Notice1 において以下の修正を行っている。
なお、秘密鍵 x を用いて m の署名を作成する際にも、同様の修正がなされている。

[修正前]

Algorithm:
Step 1. Choose a secret initial value for the seed-key, KKEY.
Step 2. In hexadecimal notation let
t = EFCDAB89 98BADCFE 10325476 C3D2E1F0 67452301.
This is a cyclic shift of the initial value for $H_0 \| H_1 \| H_2 \| H_3 \| H_4$ in the SHS.
Step 3. For j = 0 to m - 1 do
a. k = G(t,KKEY) mod q.
b. Compute $k_{j-1} = k^{-1} \bmod q$.
c. Compute $r_j = (g^k \bmod p) \bmod q$.
d. KKEY = (1 + KKEY + k) mod 2^b.
Step 4. Suppose $M_0, \ldots, M_{m-1}$ are the next m messages. For j = 0 to m - 1 do
a. Let h = SHA-1(M_j).
b. Let $s_j = (k_{j-1}(h + xr_j)) \bmod q$.
c. The signature for M_j is (r_j, s_j).
Step 5. Let t = h.
Step 6. Go to step 3.

[修正後]
Step 1. Choose a secret initial value for the seed-key, *KKEY* .
Step 2. In hexadecimal notation let
t = EFCDAB89 98BADCFE 10325476 C3D2E1F0 67452301.
This is a cyclic shift of the initial value for $H_0 \| H_1 \| H_2 \| H_3 \| H_4$ in the SHS.
Step 3. For *j* = 0 to *m* - 1 do
3.1 For *i* = 0 to 1 do
a. w_i = G(*t*, *KKEY*)
b. *KKEY* = (1 + *KKEY* + w_i) mod 2^b

3.2 $k = (w_0 \| w_1) \bmod q$

3.3 Compute $k_{j-1} = k^{-1} \bmod q$

3.4 Compute $r_j = (g^k \bmod p) \bmod q$

Step 4. Suppose $M_0, \ldots, M_{m-1}$ are the next m messages. For $j = 0$ to $m - 1$ do

a. Let $h = \text{SHA-1}(M_j)$.

b. Let $s_j = (k_{j-1}(h + xr_j)) \bmod q$

c. The signature for M_j is (r_j, s_j).

Step 5. Let $t = h$

Step 6. Go to step 3.

2.1.2. PKCS#1 のセキュリティホール

1998 年の CRYPTO '98 において、Bleichenbacher は、適用型選択暗号文攻撃に対し PKCS#1（v.1.5）が脆弱性を有することを指摘した[24]。それは、サーバの復号した平文が PKCS#1（v.1.5）のフォーマットになっているかどうかにより、クライアントに返すエラーメッセージが異なるという脆弱性である。本攻撃は、RSA による数百万の暗号文を PKCS#1（v.1.5）のフォーマットをチェックするオラクルに送ることにより、平文の一部を露呈させ、さらにその情報から所望の平文を得るという手法である。したがって、本攻撃は million message attack とも呼ばれている。
対策としては、証明可能安全性を有し、Plaintext aware padding である OAEP の使用を前提とした PKCS#1（v.2.0）を利用することが推奨される。

PKCS#1（v.1.5）における、RSA を用いた暗号化フォーマット（PKCS-conforming）は次のとおりである。

EB = 00 ‖ BlockType ‖ PaddingString ‖ 00 ‖ DataBlock

ここで、EB は k bytes の長さを有し、BlockType は 02、PaddingString は少なくとも 8 bytes 長の非零擬似乱数、DataBlock は(k-11) bytes 以下の暗号化対象（例えば premaster-secret）である。ただし、k は RSA の公開鍵 n の byte 単位の長さを表している。

本フォーマットに対し、RSA 暗号処理により得られる暗号文 c は、

$$c = m^e \bmod n$$

となる。ただし、m は EB の整数値、e は RSA の公開鍵を表している。SSL では、RSA による鍵交換プロトコルにおいて、暗号文 c をサーバに送信する。サーバは、受け取った暗号文 c が正しく復号できたかどうかの結果をクライアントに返信する。

オラクルは、攻撃者から送られてきた暗号文の復号結果が PKCS-conforming であるかどう

かをチェックし、その結果を攻撃者に返す役割をする。

オラクルを用いた攻撃手法の詳細は以下のとおりである。
攻撃者は、上記の暗号文 c に対し、以下の計算により得られた c' をオラクルに送る。

$$c' = cs^e \bmod n$$

オラクルにおける復号結果は次のようになり、オラクルは EB'が PKCS-conforming であるかどうかをチェックして、攻撃者にその結果を返す。

$$m' = c'^d \bmod n = c^d s \bmod n = ms \bmod n$$

ただし、EB'は m' の暗号化フォーマット、d は RSA の秘密鍵を表している。約 $1/2^{16}$ の確率で、EB'は以下のような復号結果となる。

$$\text{EB'} = 00 \| 02 \| \text{stuff}$$

ここで、stuff は、一般的に不正な復号結果になるが、もともとの EB の情報が一部露呈していることになる。
さらに、攻撃者は次のアルゴリズムを用いることにより、EB を得ることができる。

Step 1: ブラインディング

整数 c に対し、異なるランダムな整数 s_0 を選択する。オラクルにアクセスすることにより、

$c(s_0)^e \bmod n$ が PKCS-conforming であるかどうかを確認する。

最初に成功した s_0 に対し、

$$c_0 \leftarrow c(s_0)^e \bmod n$$
$$M_0 \leftarrow \{[E, F]\}$$
$$i \leftarrow 1$$

と設定する。ただし、

$$E = 2B$$
$$F = 3B - 1$$
$$B = 2^{8(k-2)}$$

である。

Step 2: PKCS conforming 平文に対する探索

Step 2.a: 探索開始

$i = 1$ ならば、暗号文 $c(s_1)^e \bmod n$ が PKCS-conforming であるような

$$s_1 \geq \frac{n}{F+1}$$

なる最小の正の整数を探索する。

Step 2.b: 1 つ以上区間がある場合の探索

$i > 1$ で M_{i-1} における区間数が少なくとも 2 であるならば、暗号文 $c(s_i)^e \bmod n$ が

PKCS-conforming であるような

$$s_i > s_{i-1}$$

なる最小の整数を探索する。

Step 2.c: 1 つ区間がある場合の探索

M_{i-1} が 1 つの区間（$M_{i-1} = \{[a, b]\}$）のみ含むならば、暗号文 $c(s_i)^e \bmod n$ が PKCS-conforming になるまで、

$$r_i \geq \left\lceil 2\frac{bs_{i-1} - E}{n} \right\rceil$$

かつ

$$\frac{E + r_i n}{b} \leq s_i < \frac{F + r_i n + 1}{a}$$

であるような小さい整数 r_i, s_i を選択する。

Step 3: 解の集合の制限化

s_i が見つかった後、集合 M_i は

$$\text{all } [a, b] \in M_{i-1} \text{ and } \frac{as_i - F}{n} \leq r \leq \frac{bs_i - E}{n}$$

に対して、

$$M_i \leftarrow \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{E + rn}{s_i} \right\rceil\right), \min\left(b, \left\lfloor \frac{F + rn}{s_i} \right\rfloor\right) \right] \right\}$$

として計算される。

Step 4: 解の計算

M_i が長さ 1 の 1 つの区間（$M_i = \{[a, a]\}$）だけを含むならば、

$$m \leftarrow a(s_0)^{-1} \bmod n$$

を設定し、

$$m \equiv c^d \pmod n$$

の解として m を返す。そうでなければ、

$$i \leftarrow i + 1$$

と設定し、step 2 へ戻る。

2.1.3. PKCS#1 v.2.0 OAEP に関するセキュリティホールについて

CRYPTO2001 において、Manger は、PKCS＃1 v2.0 に方式上のセキュリティホールがあることを指摘した[21]。PKCS#1 v2.0 における、RSA をベースとする暗号方式

RSAES-OAEP は、選択暗号文攻撃に対して耐性を有することが知られているが、RSA の復号処理と OAEP のインテグリティチェックにおけるエラー時のメッセージが異なる場合に、選択暗号文攻撃が可能となる。必要な選択暗号文の個数は、Bleichenbacher の PKCS#1 v1.5 に対する同様の攻撃に比べて格段に少なく、RSA 1024bit key の場合、約１１００個、RSA 2048bit key の場合、約２２００個あれば、目的とする暗号文の解読が可能になることを指摘している。

本セキュリティホールの対策としては、RSA 復号時のエラーとインテグリティチェック時のエラーの区別がつかないようエラーメッセージを通知することで回避できる。PKCS#1 v2.1 において本修正がなされているが、実行時間によって、そのエラーを判断できないようにする必要があることも指摘している。

攻撃の概要は以下のとおり、

図に示すように RSAES-OAEP の復号処理は、RSA の復号処理を行った後、integer-octet 変換を行い、OAEP の integrity を検証する手順となっている。
ここで、一般的に法 n で暗号処理を行う対象としては、法 n より小さい数あるいは 1 octet 分だけ小さい数となるので最上 octet が"00"に設定される。攻撃はこの性質を利用して行う。処理的には、RSA の復号処理を行った結果が上記の範囲になっているかどうかを検証し、その後、OAEP のインテグリティチェックを行う。従って、RSA 復号処理結果エラーと OAEP のインテグリティチェックエラーで異なるエラーメッセージを出力する場合、これを RSA 復号処理 oracle として利用する。

攻撃者が暗号文 c に対する、m= c^d (mod n)を求めたい場合、f を選択し、f^e・c(mod n)を計算し、上記 oracle に送信する。Oracle は、RSA の復号処理を行い、上記の復号結果 f・m を得る。上記 f・m が上で述べた範囲にあるかどうかを検証し、その結果を要求者に返送する。攻撃者はその oracle の返送結果に依存して、f の値を変更し、oracle に対して問い合わせることで、最終的に m の値が求まるという適応型選択暗号文攻撃を行うものである。

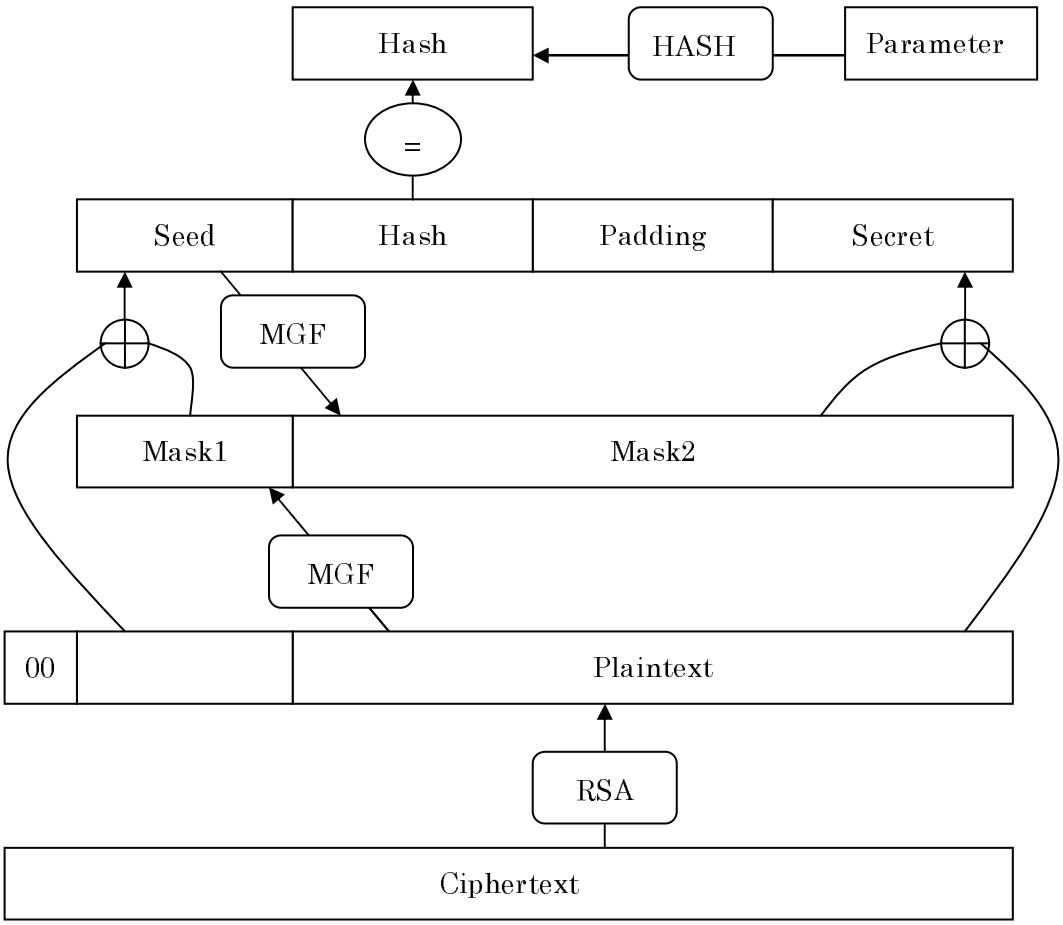


図 2.1.3.1 RSAES-OAEP Decording

2.1.4. PKCS#1（v.1.5）に対する適用的選択暗号文攻撃の拡張

2.1.2 における Bleichenbacher の攻撃は暗号学的に安全ではないが、実用的な攻撃ではなかった。Klima、Pokorny、Rosa は 2003 年、Bleichenbacher の攻撃を拡張した、現実的に攻撃可能なサイドチャネル攻撃を提案した[34]。そこでは、SSL/TLS プロトコルに利用されている PKCS#1（v.1.5）の拡張フォーマットに対し、サーバの復号した平文が正しいフォーマットであるとき、その平文に含まれるバージョン値をチェックした結果から、クライアントに返すエラーメッセージが異なるという脆弱性が指摘されている。本攻撃は、平文のフォーマットのチェックに加えて、SSL/TLS のバージョン値もチェックするオラクルを用いて平文の一部を露呈させることにより、Bleichenbacher の攻撃と同様の手法を利用できる攻撃である。さらに、本攻撃は、Bleichenbacher の攻撃における平文を得るアルゴリズムを改良することにより、実用的な攻撃となっている。ランダムに選択したインターネット上のサーバに対し、2/3 のサーバに現実的な処理時間にて攻撃可能であるというテ

スト結果も紹介されている。この攻撃に関し、2003 年 3 月 19 日、OpenSSL 0.9.6b～0.9.6i、0.9.7 及び 0.9.7a に対するセキュリティパッチが公開されている。

SSL/TLS プロトコルにおける暗号化フォーマット（S-PKCS-conforming）は次のとおりである。

$$EB = 00 \| BlockType \| PaddingString \| 00 \| DataBlock$$

ここで、EB は k bytes の長さを有し、BlockType は 02、PaddingString は(k-51) bytes 長の非零擬似乱数、DataBlock は 48 bytes の暗号化対象（例えば premaster-secret）であり、最初の 2 bytes が SSL/TLS のバージョン値になっている。ただし、k は RSA の公開鍵 n の byte 単位の長さを表している。

本フォーマットに対し、RSA 暗号処理により得られる暗号文 c は、

$$c = m^e \bmod n$$

となる。ただし、m は EB の整数値、e は RSA の公開鍵を表している。SSL では、RSA による鍵交換プロトコルにおいて、暗号文 c をサーバに送信する。サーバは、受け取った暗号文 c に対し、次のように動作する。

ｉ）サーバは復号された平文が S-PKCS-conforming であるかどうかをチェックする。その結果、平文が S-PKCS-conforming でなければ、サーバは新しい premaster-secret をランダムに生成する。そして、クライアントの Finished メッセージを受信後、すぐに通信は切断される。

ⅱ）サーバは各 S-PKCS-conforming 平文をチェックして、SSL/TLS のバージョン値が正しいかどうかを判断する（現在のバージョンは 3.0 と 3.1 である）。このバージョン値テストに失敗したら、サーバは区別可能なエラーメッセージを出す。ただし、テストは S-PKCS-conforming ではない平文には行われない。

オラクル（bad-version-oracle: BVO）は、攻撃者から送られてきた暗号文の復号結果が S-PKCS-conforming であり、かつ SSL/TLS のバージョン値が正しいかどうかをチェックし、その結果を攻撃者に返す役割をする。

BVO を用いた攻撃手法の詳細は以下のとおりである。

攻撃者は、上記の暗号文 c に対し、以下の計算により得られた c'を BVO に送る。

$$c' = cs^e \bmod n$$

BVO における復号結果は次のようになり、BVO は EB'が S-PKCS-conforming であるとき、バージョン値が正しいかどうかをチェックして、攻撃者にその結果を返す。

$$m' = c'^d \bmod n = c^d s \bmod n = ms \bmod n$$

ただし、EB'は m'の暗号化フォーマット、d は RSA の秘密鍵を表している。約 $1/2^{16}$ の確率

で、EB'は以下のような復号結果となる。

EB' = 00 ‖ 02 ‖ stuff1 ‖ 00 ‖ major ‖ minor ‖ stuff2

ただし、major、minor はそれぞれ SSL/TLS の major バージョン値、minor バージョン値を表している。ここで、stuff1 と stuff2 は、一般的に不正な復号結果になるが、もともとの EB の情報が一部露呈していることになる。

さらに、攻撃者は Bleichenbacher の攻撃と同様のアルゴリズムを用いることにより、EB を得ることができる。

ただし、暗号化フォーマットを S-PKCS-conforming、オラクルを BVO、E と F をそれぞれ

$$E = 2B + 1*256^{k-3} + 1*256^{k-4} + \ldots + 1*256^{49} = 2B + 256^{49}(256^{k-51} - 1)/255$$

$$F = 2B + 255*(256^{k-3} + 256^{k-4} + \ldots + 256^{49}) + 0 + 255*(256^{47} + 256^{46} + \ldots + 256^{0})$$
$$= 3B - 255*256^{48} - 1$$

と置き換えることとする。

また、アルゴリズムの各ステップにおいて、次の手法や性質により攻撃の効率化が行われている。

ⅰ）step 2: Parallel-Threads Method

Parallel-Threads Method は M_{i-1} における複数の区間に対して、多重処理を行う手法である。この手法は、すべての区間に対し暗号文 $cs^e \bmod n$ が S-PKCS-conforming であるような s の探索を行い、その s を見つけた場合には、その都度全区間の更新及び不要な区間の消去を行うというものである。

ⅱ）step 2.b: Beta Method

Beta Method は暗号文 $cs^e \bmod n$ が S-PKCS-conforming であるような s を効率良く探索する手法である。この手法は、$c(s_i)^e \bmod n$ と $c(s_j)^e \bmod n$ がともに S-PKCS-conforming であるような s_i、s_j を線形結合して得られた s に対し、$cs^e \bmod n$ も S-PKCS-conforming であるという性質を利用している。

ⅲ）step 3: 範囲の対称性

範囲の対称性は暗号文 $cs^e \bmod n$ が S-PKCS-conforming であるような大きい s をより小さい値に変換するために利用される性質である。この変換は、対称性を保存しながら平文の取りうる値の範囲を変換可能であるという性質を利用している。

攻撃の複雑性は BVO コール数により決定される。

1200 個の暗号文に対し、50%の攻撃が成功するのに必要な BVO コール数として次のような測定結果が記されており、サーバの公開鍵の法は長さ $8r$ （r：整数）bits を使用すべきであると述べられている。

・N =1024 bits：13.34 million 以下

・N =1025 bits : 1.20 million 以下
・N =2048 bits : 19.9 million 以下
・N =2049 bits : 3.47 million 以下
鍵長 1024 bits の RSA を用いた SSL サーバ(2x Pentium Ⅲ/1.4 GHz, 1GB RAM, 100 Mb/s, OS RedHat 7.2, Apache 1.3.27) における攻撃テストとして、以下のような結果が記されている。
・67.7 BVO calls/s
・全攻撃に対する中間時間 : 54 時間 42 分
さらに、常に成功するというわけではないが、インターネット上の 611 個の SSL/TLS サーバの 2/3 が BVO による攻撃が可能であることも示されている。

2.1.5. CBC モードのパディング方式における脆弱性を利用したサイドチャネル攻撃

＜概要＞
CBC による共通鍵暗号化処理には、バイト単位の固定のパディングパターンを使用している場合、パディングの正当性チェックの結果を利用して、平文をバイト単位で逐次求めることが可能であるというセキュリティホールが存在する。本攻撃は、パディングの正当性チェックの結果を処理時間の違いを利用して判別し、平文を求めるものである。ある条件下において、正しい実装がなされていないすべての SSL 及び TLS 実装に対して、理論上は適用可能である。OpenSSL の実装では、0.9.6i 及び 0.9.7a 以降の実装で対策が行われている。

＜CBC-PAD の脆弱性について＞

(a) 攻撃手法

CBC-PAD[41]は、RFC2040 で規定されている共通鍵ブロック暗号利用モードであり（RFC2040 では RC5 について規定)、以下のようなデータ構造を取る。

Message | Padding | Length

Length field は 1 バイト固定で与えられており、また、Padding Field は、Length 値が繰り返される構成となっている。この特徴を利用して以下のような攻撃を行うことが可能であることが報告されている[42]。なお、バイト単位の固定パターンでパディングする方式については、本攻撃が同様に適用できる。また、固定のパディング方式のいくつかには、脆弱性があることが指摘されている[44][45]。

攻撃には、以下の 5 種類の Oracle を使用することで、段階的に行う。このとき、b を使用するブロック暗号の block length (Byte)、N を暗号文 ***Y***のブロック数とする。

Oracle O:
パディングの正当性を判定する Oracle。任意の暗号文入力に対して、パディング処理が正しいかどうか判定を行い、正しければ 1 を、正しくなければ 0 を返す。

Check1:
CBC-PAD で暗号化された任意の暗号文 ***Y***のあるブロック Y_n の任意の Byte y_i について予測した平文 x_i が正しいかどうかの判定を行う。Check1 は、任意の i に対して、長さ b - i のランダムな暗号文 L と、長さ i の暗号文 R=r_i|...|r_b xor i|...|i (各 r_i はバイト単位) から生成される r=L|R を使用して r|y_i なる暗号文を生成してパディングの正当性を判定する Oracle O に入力する。

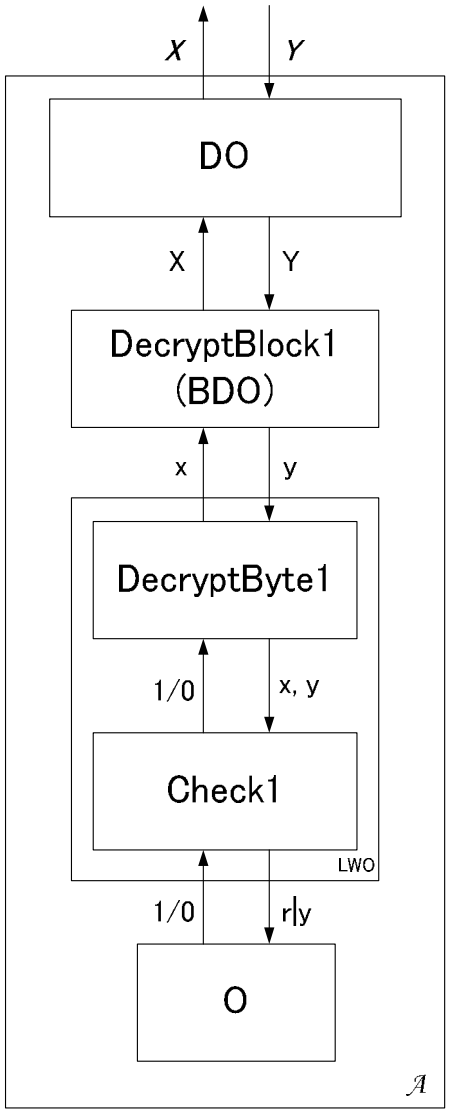
DecryptByte1:
Check1 を利用して、y_i に対する平文 x_i を求める。r_i+1,..., r_b は既知であるとき（i は 1～b)、r_i のすべての値に対して、r_i|...|r_b を作成して、y_i と共に Check1 に問い合わせを行う。

なお、S.Vaudenay の論文では、Check1 と DecryptByte1 をまとめて Last Word Oracle(LWO)と称している。

DecryptBlock1:
DecryptByte1 を利用して、暗号文Yのあるブロック Y_n を求める。Y_n の各 Byte である y_i ごとに、DecryptByte1 を呼び出す。S.Vaudenay の論文では、Block Decryption Oracle(BDO)と称する。

Decryption Oracle (DO):
DecryptBlock1 を利用して、暗号文 ***Y*** を求める。***Y*** の各ブロックである Y_n ごとに DecryptBlock1 を呼び出す。

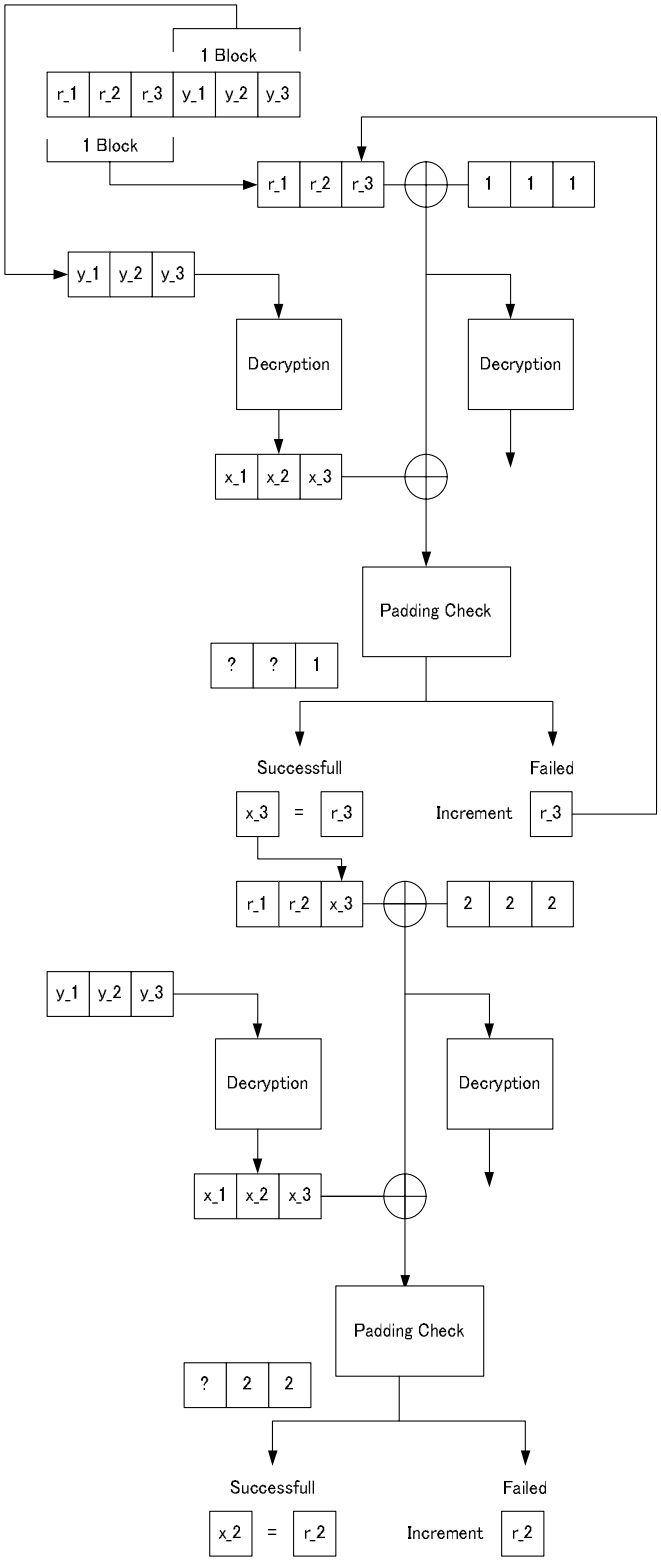
以下に各 Oracle の関係を図示する。



DO は、BDO を N 回呼び出し、BDO は、DO の各呼び出しに対して LWO を b 回呼び出す。DecryptByte1 が、Check1 を 2^8 回呼び出し、Check1 は各呼び出しに対して O を 1 回呼び出す。従って、LWO は、全体として、BDO の各呼び出しに対して O を 2^8 回呼び出す。以上から、攻撃に必要な計算量は、O(2^8bN)であることが判る。本攻撃は、暗号文に対する平文を 1Byte 単位で分割統治的に逐次求めることができるため、全数探索よりも計算量は小さくなる。

図 1 に攻撃アルゴリズムを示す。平文 x_1, x_2, x_3 を推測して r_1, r_2, r_3 を作成し、Rr_1,r_2, r_3 とパディング（1,1,1 や 2,2,2）を xor したものを第一ブロック、暗号文 y_1, y_2, y_3 を第二ブロックとして、復号処理を依頼し、エラーメッセージから推測が正しかったかどうか判定する。攻撃アルゴリズムからわかるように、本脆弱性は、パディング処

理が、ある固定の規則に基づいていることに起因している。



(b)実環境への適用

(a)で述べた攻撃を実環境に適用するためには、以下の 2 つの条件が必要である。

（1） 通信を行うあるエンティティに対して、任意の回数の問い合わせが可能である。
（2） パディングチェックの結果を、攻撃者は与えられる。もしくは、ある優位性を持って推測できる。

条件（1）については、SSL/TLS 等では、パディングチェックが失敗した場合、直ちにセションが切断され、共有鍵の更新が行われるため、連続して、複数回の問い合わせを行うことが難しい。従って、暗号文の 1Byte を求める場合でも、2^-8 の確率でしか成功しない。ところが、送信される情報がパスワード等の常に同一の情報であれば、複数のセションに渡って攻撃を行うことで、最終的に暗号文に対する平文を得ることができる。S.Vaudenay は、Oracle O を Bomb Oracle O (Check が Fail すると使用不可能になる)でモデル化し、議論を行っている。従って、毎回固定の平文に対する暗号文が送られるという条件下においては、条件（1）を満たし攻撃は可能である。
条件（2）については、WTLS 等の一部の仕様では、パディングエラーの場合のエラーメッセージと、メッセージ認証子がエラーになった場合にエラーメッセージが異なるため、ある優位性を持って識別可能となっている。ところが、TLS 1.1(SSL 3.0 及び TLS 1.0 は脆弱性あり)では、2 つのエラーで同一のエラーメッセージを返すため、通常の方法では、攻撃者は、ある優位性を持って識別することはできない。

＜タイミング攻撃を利用した攻撃＞

(a)攻撃手法
B.Canvel らは、S.Vaudenay の方式を拡張することで、TLS1.1 に対しても攻撃可能であることを示した[43]。具体的には、Oracle O を、処理時間を返す Oracle O'へと変更することで、Adversary を構成する。この攻撃は、パディングエラーでは、ハッシュ関数の演算処理を行わないのに対し、メッセージ認証子のエラーでは、ハッシュ関数の演算処理を行うため、処理時間に差が生じるという特性を利用したタイミング攻撃である。このとき、処理時間を最大とするため、仕様上許す範囲において、Oracle への入力暗号文長が最大 (TLS では、2^14+2048Byte) となるように、ランダムなデータ列 f を結合させる。従って、Oracle O'は、入力暗号文 f|r|y に対して、処理時間 T_m を返す。B.Canvel らの解析結果では、パディングエラーとなる場合の処理時間の期待値 μ_W は 21.57msec、分散 σ_W は 1.86 であるのに対し、メッセージ認証子エラーの場合の期待値 μ_R は、23.63msec、分散 σ_R は 1.48 であった。従って、複数回の試行を行い精度を高めなければならない。False Positive

あるいは、False Negative である可能性は、試行回数 n に対して以下のように指数関数的に減少する。

$$\in_{-} = \varphi\left(\frac{\tau^{t} - \mu_R}{\sigma}\sqrt{n}\right)$$

$$\in_{+} = \varphi\left(-\frac{\tau^{t} - \mu_W}{\sigma}\sqrt{n}\right)$$

$$\varphi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt.$$

判定は、以下のように閾値に基づいて行う。

STOP: $T_1 + \ldots + T_j - j\frac{\mu_R + \mu_W}{2} \notin \left[\tau_{-}^{t}, \mu_{+}^{t}\right]$

ACCEPT: $T_1 + \ldots + T_j - j\frac{\mu_R + \mu_W}{2} > \tau_{+}^{t}$

$$\tau_{+}^{t} = \frac{\sigma^2}{\mu_R - \mu_W}\log\tau_{+}$$

$$\tau_{-}^{t} = \frac{\sigma^2}{\mu_R - \mu_W}\log\tau_{-}.$$

$$\tau_{+} \approx \frac{1-\in_{-}}{\in_{+}}$$

$$\tau_{-} \approx \frac{\in_{-}}{1-\in_{+}}$$

以上のことからエラーを判別するのに十分な試行回数 J は、それぞれ以下のようになる。

$$J_W \approx \frac{2\sigma^2}{(\mu_R - \mu_W)^2}\log\tau_{-}$$

$$J_R \approx \frac{2\sigma^2}{(\mu_R - \mu_W)^2}\log\tau_{+}$$

(b)実環境への適用

CBC-PAD を使用して毎回固定の平文を暗号化して送信しているエンティティに対しては、適用できる可能性がある。しかしながら、ネットワーク上の遅延などが存在する実環境に対しては、どの程度現実的であるか、十分実証されてはいない。しかしながら、本攻撃に対する対策は非常に容易であり、実装においては対策を講じておくことが望ましい。例えば、パディング処理を行った後にメッセージ認証子を付加するように方式を変更する方法などが考えられる。また、OpenSSL 0.9.6i 及び 0.9.7a 以降では、パディング処理が失敗した場合、ハッシュ処理の時間に相当する時間待機し、その後エラーメッセージを返信する方式を実装している。このような手法も本攻撃に対する十分な対策となる。

2.1.6. その他の CBC パディングの脆弱性を利用したサイドチャネル攻撃

2.1.5 で指摘した攻撃は、固定パターンのパディングを行っている他のパディング方式を使用した CBC モードにも適用できるものである。攻撃が成功するための要件は、パディング処理の結果をある優位性をもって取得可能であるという前提のもとで、パディング処理の結果から少なくとも 1bit 以上の情報を確定できる、ということである。特に、ソフトウェア実装においては、効率性の観点から、バイト単位かつ固定のパディング処理を行うことが多く、注意して実装しなければならない。具体的には、攻撃者が Distinguisher を構成できないような実装を心掛けるべきである。
J.Black は、いくつかのパディング方式について、M.Bellare らが示した共通鍵暗号操作モードにおける“Semantic Security”[46]を満たすかどうかの評価を行った[44]。(表 2.1.6.1)

表 2.1.6.1

	Bit-oriented or Byte-oriented	Semantic Security	Queries to Recover Plaintext
CBC-PAD	Byte	No	64b+lg(b)
ESP-PAD	Byte	No	64b+lg(b)
XY-PAD	Byte	No	64b+lg(b)
OZ-PAD	Bit	No	2^{8b-1}
BOZ-PAD	Byte	No	64b+lg(b)
PAIR-PAD	Byte	No	2^{8b-1}
ABYT-PAD	Byte	Yes	N/A
ABIT-PAD	Bit	Yes	N/A

※ b は、1 ブロックあたりのバイト数
※ ESP-PAD は、IPSec-ESP で使用されているパディング
※ XY-PAD は、メッセージ M に対して、M||XYYY...とバイト単位でパディングする方式
X, Y は固定値
※ OZ-PAD は、メッセージ M に対して、M||1000...とビット単位でパディングする方式

※ BOZ-PAD は、OZ-PAD をバイト単位に拡張したもの。M|80, 00, 00,...とパディングする。
※ PAIR-PAD は、XY-PAD を拡張したもので、M|XYYY..., M|YXXX..., を選択可能。
※ ABYT-PAD は、メッセージ M の最下位バイトとは異なるバイト Y で M|YYY,... とパディングする方式。Y は、最下位バイト以外からランダムに選択可能。
※ ABIT-PAD は、ABYT-PAD のビット版。メッセージ M の最下位ビットと、異なるビットでパディングする方式。

上記の表において、少なくとも Semantic Security を満たしていない方式については、脆弱性があると判断できる。2.1.5 節で述べたように、脆弱性があるパディングの使用が直ちに SSL の脆弱性につながる訳では無い。なぜなら、2.1.5 節で述べた対策が SSL の実装において行われているからである。しかしながら、安全なパディング方式を使用するほうがより望ましいと考えられる。
また、K.G.Paterson らは、RSA カンファレンスの Ciphers Track にて ISO 標準に採用されている CBC モードのパディング方式について脆弱性があり、サイドチャネル攻撃が可能との発表を行う予定である[45]。(2004 年 2 月 6 日現在、未確認)

2.1.7. PKCS#1 V2.1 などの脆弱性を利用したサイドチャネル攻撃

Klima と Rosa は 2002 年、次の 3 つの攻撃を提案した[35]。
(1) RSAES-OAEP（PKCS#1 v.2.1）に対するサイドチャネル攻撃
(2) 2.1.2 における Bleichenbacher の攻撃と 2.1.3 における Manger の攻撃をそれぞれ利用してサーバの署名を偽造する攻撃
(3) RSA-KEM に対するフォールトサイドチャネル攻撃
(1) の攻撃は、PKCS#1（v.2.1）に従う EME-OAEP-DECODE 手法による復号プロセスにおいて、攻撃者が平文の一部のハミング重みを集めることにより、平文の最下位ビットを露呈させ、さらにその情報から所望の平文を得るという手法である。(2) の攻撃は、RSA の同じ秘密鍵に関し、暗号方式における復号鍵と署名方式における署名鍵を同時に使用する場合に起こる脆弱性を利用した手法である。(3) の攻撃は、RSA-KEM において、秘密鍵のあるビットにフォールトを故意に起こさせることにより、秘密鍵をビットごとに露呈させる手法である。
(2)の攻撃に関しては、2.1.2 や 2.1.3 において既に各攻撃に対する対策が提案されている。そして、(1) の攻撃はハミング重みに対する電力サイドチャネル攻撃を、(3) の攻撃は秘密鍵に対する電力フォールトサイドチャネル攻撃をそれぞれ前提としているため、SSL/TLS においては実用的な攻撃手法ではない。
また、フォールト攻撃の対応策として、以下のことが推奨されることも記述されている。

a) 秘密鍵やプロセス中に使用されるパラメータのインテグリティをチェックすること。
b) エラーメッセージの範囲を最小にすること。
c) フォールト検出を装備したプラットフォームや補正用の設備（フォールトトレラントシステム）をできる限り使用すること。
また、(3) の攻撃例に対する防御法として、

$$x = y^d \bmod n \text{ と } y = x^e \bmod n$$

の両方をチェックすることも推奨されることも記されている。

以下に、各攻撃に関する詳細を述べる。
（1）RSAES-OAEP（PKCS#1 v.2.1）に対するサイドチャネル攻撃
図 2.1.7.1 に RSAES-OAEP（PKCS#1 v.2.1）の復号プロセスを示す。この復号プロセスにおいて、MGF として使用する SHA-1 に maskedDB の値を入力する際の処理に脆弱性が存在する。

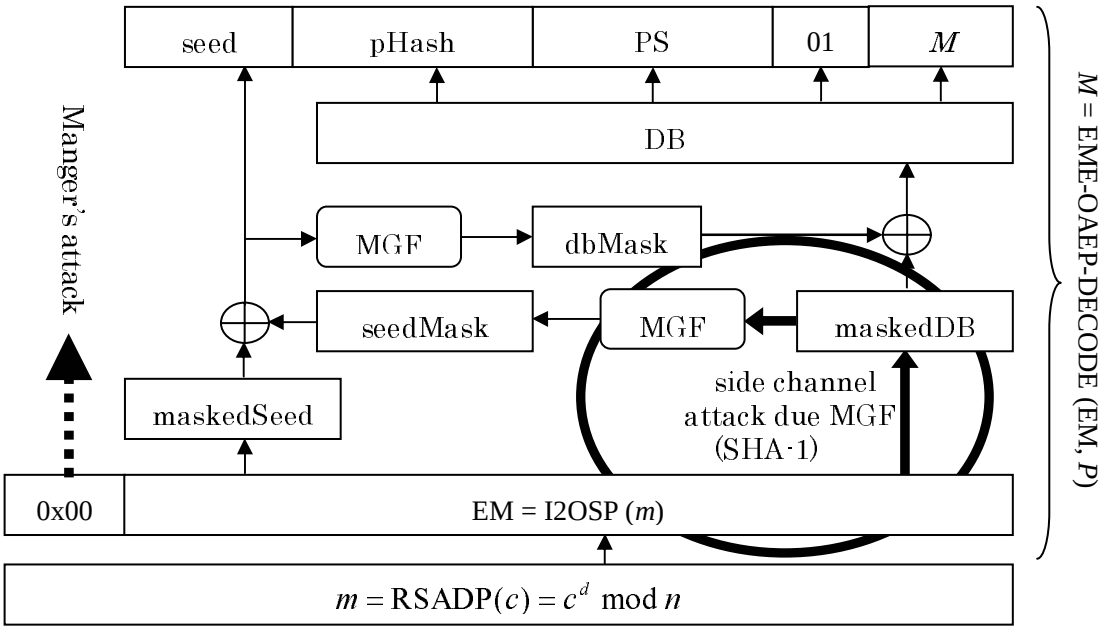


図 2.1.7.1 RSAES-OAEP Decoding（PKCS#1 v.2.1）

SHA-1 に入力する最後のブロックを分割した 32 bits の一部の変数 $w(W_j)(i = 7,8,9,10)$ は次のとおりである。

$$W_7 = m[14]\, m[13]\, m[12]\, m[11]$$
$$W_8 = m[10]\, m[9]\, m[8]\, m[7]$$
$$W_9 = m[6]\, m[5]\, m[4]\, m[3]$$
$$W_{10} = m[2]\, m[1]\, m[0]\, 00$$

ただし、$m[j](j = 0,\cdots,14)$ は平文の j 番目のオクテットであり、W_i の各ビットを $W_{i,31}\ W_{i,30}\ W_{i,29}\cdots W_{i,0}$ と表す（m [0]が最下位オクテットである）。

このとき、$w(W_j)(i = 8,9,10)$ を W_i のハミング重み（1 の個数）とし、攻撃者はこれらのハミング重みを入手可能であるとする。

RSA 暗号処理

$$c = m^e \bmod n$$

により暗号文 c が得られる。ただし、m は平文の整数値、e は RSA の公開鍵を表している。
暗号文 c は RSA 復号処理

$$m = c^d \bmod n = m^{ed} \bmod n$$

により平文 m に復号される。ただし、d は RSA の秘密鍵を表している。
攻撃者は上記の暗号文 c に対し、

$$c' = c * 2^{-e} \bmod n$$

を RSA 復号処理をさせ、その結果を m'とする。
この場合、上記の 32 bits の変数 W_i、及びハミング重み $w(W_i)$に対応する変数とハミング重みをそれぞれ、W_i'、$w(W_i')$とする。

攻撃者はハミング重み $w(W_i)$と $w(W_i')$を集める。
最下位ビット $W_{10,8}$ が 0 のとき、

$$m' = m >> 1$$

最下位ビット $W_{10,8}$ が 1 のとき、

$$m' = (m + n) >> 1$$

との値を返される。ただし、">>1" は右に 1 bit シフトを表している。

$W_{10,8} = 0$ のとき、$w(W_i)$と $w(W_i')$には

$$w(W_8') = w(W_8) - W_{8,0} + W_{7,0}$$
$$w(W_9') = w(W_9) - W_{9,0} + W_{8,0}$$
$$w(W_{10}') = w(W_{10}) - W_{10,8} + W_{9,0} = w(W_{10}) + W_{9,0}$$

の関係が成立し、それらの可能な関係は表 2.1.7.1 のとおりになる。

攻撃者は集めたハミング重み $w(W_i)$と $w(W_i')$のそれぞれの関係を調べ、表 2.1.7.1 のいずれかの関係を満足している場合には、

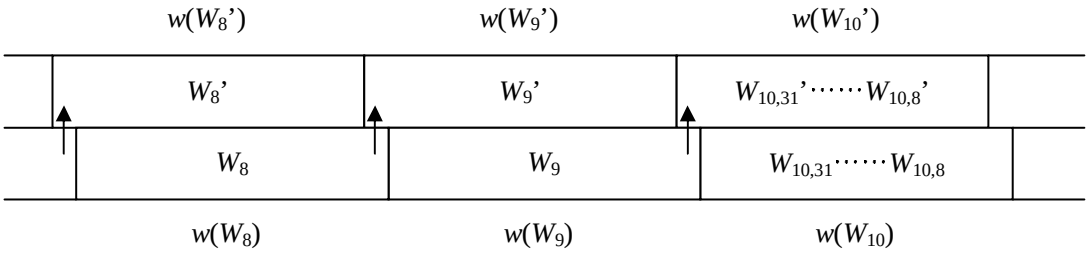
$$W_{10,8} = 0$$

であると判断し、いずれの関係も満足していなければ、

$$W_{10,8} = 1$$

であると判断する。この誤り確率はおよそ 0.008 である。
最下位ビット $W_{10,8}$ の値がわかれば、攻撃者は[83]の定理により多項式時間にて平文を得ることができる。

表 2.1.7.1 $w(W_i)$と $w(W_i')$の可能な関係



$W_{9,0}$	$W_{8,0}$	$W_{7,0}$	可能な関係		
0	0	0	$w(W_{10}') = w(W_{10})$	$w(W_9') = w(W_9)$	$w(W_8') = w(W_8)$
0	0	1	$w(W_{10}') = w(W_{10})$	$w(W_9') = w(W_9)$	$w(W_8') = w(W_8) + 1$
0	1	0	$w(W_{10}') = w(W_{10})$	$w(W_9') = w(W_9) + 1$	$w(W_8') = w(W_8) - 1$
0	1	1	$w(W_{10}') = w(W_{10})$	$w(W_9') = w(W_9) + 1$	$w(W_8') = w(W_8)$
1	0	0	$w(W_{10}') = w(W_{10}) + 1$	$w(W_9') = w(W_9) - 1$	$w(W_8') = w(W_8)$
1	0	1	$w(W_{10}') = w(W_{10}) + 1$	$w(W_9') = w(W_9) - 1$	$w(W_8') = w(W_8) + 1$
1	1	0	$w(W_{10}') = w(W_{10}) + 1$	$w(W_9') = w(W_9)$	$w(W_8') = w(W_8) - 1$
1	1	1	$w(W_{10}') = w(W_{10}) + 1$	$w(W_9') = w(W_9)$	$w(W_8') = w(W_8)$

(2) 2.1.2 における Bleichenbacher の攻撃と 2.1.3 における Manger の攻撃をそれぞれ利用してサーバの署名を偽造する攻撃

Bleichenbacher の攻撃や Manger の攻撃がそれぞれ PKCS#1（v.1.5）や PKCS#1（v.2.0）における暗号化に対して成功する場合には、署名に同じ RSA の秘密鍵を用いると署名も偽造することができるという脆弱性が存在する。
暗号化と署名に同じ鍵を使用する主なアプリケーションの例として、SSL プロトコルがある。

PKCS#1（v.1.5）における平文のフォーマットは、最上位の 2 octets が 00 ‖ 02 であり、11 番目以降のオクテットに 00 が現れるフォーマットになっている。
PKCS#1（v.2.0）における平文のフォーマットは、最上位オクテットが 00 であるフォーマットになっている。

各フォーマットに対し、RSA 暗号処理により得られる暗号文 c は、

$$c = m^e \bmod n$$

となる。
Bleichenbacher の攻撃や Manger の攻撃に使用されるオラクルは、攻撃者から送られてきた暗号文の復号結果が、上記の各フォーマットであるかどうかをチェックし、その結果を攻撃者に返す役割をする。

攻撃者は、上記の暗号文 c に対し、以下の計算により得られた c' を各オラクルに送る。

$$c' = cr^e \bmod n$$

オラクルにおける復号結果は次のようになり、オラクルは正しいフォーマットであるかどうかをチェックして、攻撃者にその結果を返す。

$$m' = c'^d \bmod n = c^d r \bmod n = mr \bmod n$$

各攻撃において、攻撃者はある確率にて正しいフォーマットである復号結果を得られる。平文の情報が一部露呈していることになるため、攻撃者はもとの平文 m も得ることができる。
このとき、c をメッセージとし、そのメッセージ c に対する同様の攻撃を行うことにより、署名 m を得ることができる。

（3）RSA-KEM に対するフォールトサイドチャネル攻撃
RSA-KEM において、受信者が復号して得られたメッセージのインテグリティをチェックした結果により、送信者に返すメッセージが異なるという脆弱性が存在する。本攻撃として、2 つの例を挙げる。

Receiver Oracle（RO）は、メッセージ M のインテグリティチェックを行い、正しければ“yes”、間違っていれば“no”を出力する。

RSA confirmation oracle: RSA-$CO_{d,n}(r, y)$は次のように定義される。
RSA-$CO_{d,n}(r, y)$ → (ANSWER = “yes/no”)の手順
1. $K = \mathrm{KDF}(r)$：KDF - Key Derivation Function
2. $C0 = y$

3. $C1 = \mathrm{DEM.Encrypt}(K, M)$

4. $C = C0 \| C1$

5. 暗号文 C を RO に送る。RO にて続きを行う。

a. 次の手順にて、$K = \mathrm{KEM.Decrypt}(d, C0)$ を計算する。

i．$y = C0 < n$ かどうかチェックする。そうでなければエラーが起こる。

ii．$r' = y^d \bmod n$ を計算する。

iii．$K' = \mathrm{KDF}(r')$

b. $M' = \mathrm{DEM.Decrypt}(K', C1)$

c. *M'*のインテグリティチェックを行う。

d. チェックが正しければ RO は“yes”と、そうでなければ RO は“no”と応答する。

6. RO の応答が“yes”なら、RSA-$CO_{d,n}(r, y)$の応答も“yes”、そうでなければ、RSA-$CO_{d,n}(r, y)$の応答も“no”である。

つまり、上記のオラクルは

$r = y^d \bmod n$ のときいつでも、“yes”

$r \neq y^d \bmod n$ のとき 1 に近い確率で、“no”

を返すオラクルである。

攻撃例 1 は次のとおりである。

秘密鍵 *d* の *i* 番目のビット *d*(*i*)において誤りが起きたとき、

$$d' = d + 2^i \text{ if } d(i) = 0$$

$$d' = d - 2^i \text{ if } d(i) = 1$$

なる *d'* を step 5.a. ii において、*d* の代わりに用いることを考える。

そのとき、$r = y^{d'} \bmod n$ に対して、

$$r = (y^d * \alpha \bmod n) \text{ if } d(i) = 0$$

$$r = (y^d * \alpha^{-1} \bmod n) \text{ if } d(i) = 1$$

となる。ただし、$I = 2^i$、$\alpha \equiv y^I \bmod n$、$\alpha * \alpha^{-1} \equiv 1 \bmod n$ である。

攻撃者は RSA-$CO_{d',n}$ を利用して次の手順を繰り返すことにより、*d*(*i*)の値を得ることができる。

1. $0 < x < n$ なる x をランダムに選ぶ。
2. $y = x^e \bmod n$
3. $r = x * \alpha \bmod n$
4. RSA-$CO_{d',n}(r, y)$が“yes”を返すならば $d(i) = 0$、そうでなければ $d(i) = 1$ と設定する。

攻撃例 2 は次のとおりである。

記号を次のように定義する。

p: $p = t * 2^s + 1$なる素数

t: 小さな素数

g: $\boldsymbol{Z_p}^*$の生成元

$\boldsymbol{Z_p}^*$の位数: *d* の最大値より大きい

Step 1: 値$D_s = d \bmod 2^s$の計算

$$d = \sum_{k=1}^{b} d(b-k) * 2^{b-k}$$

とする。ただし、*b* は *d* の 2 進形のビット数、$d(i) \in \{0,1\}(0 \le i \le b-1)$ である。

(*p*-1)が 2^i で割り切れると仮定し、$I = 2^i$、$J = 2^j$ とおく。

それから、

$$r = g^d \bmod p$$
$$D(i) = d \bmod 2^i$$

と定義する。

そのとき、

$$r^{(p-1)/I} \equiv [g^d]^{(p-1)/I} \equiv [g^{(p-1)/I}]^d \equiv [g^{(p-1)/I}]^{d \bmod I} \equiv [g^{(p-1)/I}]^{D(i)} \bmod p$$

より、

$$r^{(p-1)/I} \equiv [g^{(p-1)/I}]^{D(i)} \bmod p \qquad (1)$$

となる。*D*(*i*)の値は

$$D(i) = \sum_{k=1}^{i} d(i-k) * 2^{i-k}$$

と表される。

以下、帰納法にて示す。

i = 1 に対して、(1)から

$$r^{(p-1)/2} \equiv [g^{(p-1)/2}]^{d(0)} \bmod p$$

が成立する。*r* の定義から、

$$r^{(p-1)/2} \equiv [g^{(p-1)/2}]^d \bmod p$$

となるので、

$$[g^{(p-1)/2}]^d \equiv [g^{(p-1)/2}]^{d(0)} \bmod p \qquad (2)$$

が成立する。

$g^{(p-1)/2} \equiv p-1 \bmod p$ であり、$[g^{(p-1)/2}]^{d(0)} \bmod p$ はビット *d*(0)に依存してたった 2 つの可能な値をとりうる。

そこで、$d(0) = 1$ と仮定して、RSA-$CO_{d',p}$(*p*-1, *p*-1)を利用する。

オラクルが
“yes”を返すなら、$d(0) = 1$
そうでなければ、$d(0) = 0$
と設定する。
ある $0 < j < s$ に対して、値 $D(j)$を知っていると仮定する。
(1)から

$$r^{(p-1)/(2J)} \equiv [g^{(p-1)/(2J)}]^{D(j+1)} \bmod p \tag{3}$$

が成立する。$\alpha = d(j) * 2^j = d(j) * J$ とおくと、

$$D(j+1) = d \bmod 2^{j+1} = \alpha + D(j)$$

が得られる。
(3)の右辺の値に対し、

$$[g^{(p-1)/(2J)}]^{D(j+1)} \equiv [g^{(p-1)/(2J)}]^{\alpha} * [g^{(p-1)/(2J)}]^{D(j)} \equiv [g^{(p-1)/2}]^{d(j)} * [g^{(p-1)/(2J)}]^{D(j)}$$
$$\equiv (p-1)^{d(j)} * [g^{(p-1)/(2J)}]^{D(j)} \bmod p$$

となるので、

$$r^{(p-1)/(2J)} \equiv (p-1)^{d(j)} * [g^{(p-1)/(2J)}]^{D(j)} \bmod p$$

が得られる。r の定義を用いると、

$$[g^{(p-1)/(2J)}]^{d} \equiv (p-1)^{d(j)} * [g^{(p-1)/(2J)}]^{D(j)} \bmod p \tag{4}$$

となる。

ここで、$d(j) = 0$ と推測し、$\mathrm{RSA\text{-}CO}_{d,p}([g^{(p-1)/(2J)}]^{D(j)} \bmod p,\ g^{(p-1)/(2J)} \bmod p)$ を利用する。
オラクルが
“yes”を返すなら、$d(j) = 0$
そうでなければ、$d(j) = 1$
と設定する。
以上の帰納法から、$D_s = D(s)$ が得られる。
Step 2: 値 $D_t = d \bmod t$ の計算

$$r^{(p-1)/t} \equiv [g^{(p-1)/t}]^{j} \bmod p$$

を満足する jを見つけるまであらゆる値 $j = 0, \cdots, t-1$ をテストし、得られた値を D_tとする。
r の定義を用いて

$$[g^{(p-1)/t}]^{d} \equiv [g^{(p-1)/t}]^{j} \bmod p$$

と書き直し、$\mathrm{RSA\text{-}CO}_{d,p}([g^{(p-1)/t}]^{j} \bmod p,\ g^{(p-1)/t} \bmod p)$ を $j = 0, \cdots, t-1$ に対して逐次行う。
オラクルが“yes”を返すとき、$D_t = j$ と設定する。

Step 3: 値 d の計算
Step 1 と Step 2 により、合同式

$$d \equiv D_s \bmod 2^s$$
$$d \equiv D_t \bmod t$$

が得られている。
$\gcd(t, 2^s) = 1$ が成立し、中国人の剰余定理によって両合同式を満足する $0 \le d < t * 2^s$ なるただひとつの値が存在する。
d の値は以下のようにして計算できる。
1. $\gcd(t, 2^s) = 1$ であるから、$\gamma * 2^s \equiv 1 \bmod t$ なる存在するただひとつの値 γ を計算する。
2. $v = (D_t - D_s) * \gamma \bmod t$ を計算する。
3. $d = D_s + v * 2^s$

2.1.8. SSL ソフトウェアに対するタイミング攻撃

D.Brumley らは、SSL で使用されている RSA の復号処理において、タイミング攻撃を実行し、秘密鍵を復元する攻撃を提案している[37]。さらに、OpenSSL に攻撃を試み、評価を行っている。本攻撃は、以下の 2 つの処理時間の違いが攻撃に利用可能である。

（1） Montgomery reduction における extra reduction の発生回数
（2） 多倍長演算において、Karatsuba が選択されるか否か

通常 RSA の復号処理においては、法 N=pq の演算を、p, q について個別に行い中国人剰余定理を用いて結合させて復号する。ある g に対する g mod q の演算においては、g が q のべき乗である場合、Montgomery reduction の extra reduction は著しく減少する。従って、暗号文 g の復号処理においては、g < q であれば遅く、g > q であれば速い。一方、Karatsuba は、g < q において速く、g > qにおいて遅い。
ここで（1）を使用する方式を例にとり説明する。今 q の上位 i-1 番目までが求められている場合の i 番目の bit の求め方は、以下のようになる。i-1 番目までは求めた値、i 番目の 0、i+1 番目以降はすべて 0 とした暗号文 g と i 番目を 1 にした g'を用意する。そして、2 つの暗号文の復号を依頼しその処理時間の差を計測する。もし、処理時間の差が大きければ、g < q < g' が成立しているので、q の i 番目のビットは 1 である。もし、処理時間の差が小さければ、g < g' < q が成立しているので、q の i 番目のビットは 1 である。以上のような判定を利用し、q の半分以上の上位ビットを求め、その後 Coppersmith のアルゴリズム[47]により完全な因数分解を実行して、秘密鍵を得る。

彼らは、Optimized, Unoptimaized 双方のコンパイルによって生成した OpenSSL について攻撃可能であり、Unoptimized な OpenSSL においては、359000 以下のクエリ（復号要求）で秘密鍵が求められると指摘している。また、LAN 環境においても評価を実施し、ネットワーク環境下でも攻撃可能であると結論付けている。TLS 1.1 の RFC ドラフトにおいては、本攻撃に対する対策が明記されていることから、最新の実装には対策は行われていると考えられる。

2.2. プロトコル上のセキュリティホールとその対策

以下に SSL3.0 プロトコルに対する攻撃法とその対策を列挙する。

2.2.1. Man-in-the-middle attack

攻撃者がサーバ・クライアントの間に入り、サーバに対してはクライアント、クライアントに対してはサーバになりすまし、情報を盗聴あるいは改竄を試みる攻撃方法である。SSL には、(1)相互認証、(2)サーバ認証のみ、(3)匿名、の 3 種類の認証モードがあり、それらを選択可能となっている。(1)、(2)の場合、公開鍵証明書を使用し公開鍵が正当であることの検証が行われるため、この攻撃に対して安全である。しかしながら、(3)の場合には、一般的な Man-in-the-middle 攻撃が可能であり、利用することは推奨されない。

2.2.2. Replay attack

攻撃者はサーバやクライアントの通信を盗聴し、それを再送信することによりなりすます攻撃である。本攻撃は、通信路が暗号化されていたとしても、暗号化後のデータを用いるため実行可能である。しかし、SSL においては鍵交換方式ごとにそれぞれ以下のような対策を講じており、攻撃は成功しない。

(i) RSA 認証
RSA を用いる場合、鍵交換とサーバ認証は同時に行われる。公開鍵は server key exchage におけるサーバの証明書もしくは temporary RSA key に含まれる。Temporary RSA key が用いられる場合、サーバによる RSA もしくは DSA 署名がされる。署名には ClientHello.random が含まれるため replay 攻撃はできない。
(ii) Diffie-Hellman
Diffie-Hellman 鍵交換を用いる場合、サーバは fixed Diffie- Hellman parameter を含む証明書を用いるか、DSS もしくは RSA 署名されている temporary Diffie-Hellman parameter（ephemeral Diffie-Hellman）を用いる。Temporary Diffie –Hellman parameter は、hello.random の値とともに署名されるため replay 攻撃はできない。また、クライアントは証明書や署名を検証することにより paramater がサーバのものであることを知ることができる。

また、シーケンス番号については HMAC を使用してその正当性を保証しているため、この点からも replay 攻撃は困難であると考えられる。

2.2.3. Version rollback attack

攻撃者が SSL3.0 に対応しているサーバ、クライアントに対して SSL2.0 やそれ以下の Version で通信を行うように強制する攻撃法である。3.0 以上のサーバは non-random PKCS#1 block type 2 message padding を使用しているため、攻撃を検出することが可能である。しかし、攻撃者が暗号化鍵を総当たり攻撃により解読し、アプリケーションにより設定されている待ち時間以内に、新しい ENCRYPTED- KEY- DATA を代わりに送ることができれば攻撃は成功する。しかしながら、本攻撃についても鍵長を長く取ることで防ぐことが可能である。

2.2.4. Ciphersuite rollback attack

攻撃者が、サーバ、クライアントの hello メッセージに含まれる ciphersuite を改ざんし、鍵長が短く弱い暗号方式や、メッセージ認証なしなど攻撃者が望む ciphersuite をサーバ、クライアントに強制する攻撃法である。本攻撃は、SSL2.0 においては有効な攻撃方法であったが、SSL3.0 においては、change cipher spec, alert m e s s a g eを除くすべてのメッセージについて、master_secret を使用して HMAC を作成し、それを finished メッセージにて交換し、メッセージ認証を行うため成功しない。

2.2.5. Key-exchange algorithm rollback attack

攻撃者は、server key exchange メッセージ内の KeyExchangeAlgorithm が署名対象データとなっていないことを利用し、このフィールドを改ざんし man-in-the-middle attack を成功させる攻撃方法である。攻撃においては、サーバには ephemeral Diffie-Hellman 鍵交換アルゴリズムを、クライアントには RSA を使用させるよう改ざんする。すなわち、攻撃者は以下のような手順によって、サーバ、クライアント双方の pre_master_secret を取得する。(C をクライアント、S をサーバ、M を攻撃者とし、{}x は x で署名することを表す。)

[client hello:]
1. C→M : SSL_RSA_･･･
1'. M→S : SSL_DHE_RSA_･･･
[server hello:]
2. S→M : SSL_DHE_RSA_･･･
2'. M→C : SSL_RSA･･･
[server key exchange:]

3. S→M : {p,g,y}Ks
3'. M→C : {p,g,y}Ks
[Client key exchange:]
4. C→M : k^g mod p
4'. M→S : g^x mod p

クライアントCは、k を、サーバSは、g^{xy} mod p を pre_master_secret とする。攻撃者Mは、その両方を所持することができる。従って、偽の finished メッセージを容易に作成することが可能であり man-in-the-middle attack は成功する。

2.2.6. Cut-and-paste attack

別セッションなどで盗聴した暗号文をセッション中でやり取りされる ciphertext に継接ぎする攻撃方法である。例えば、暗号文中にある hostname を、別のセッションで入手した暗号化された hostname で置き換える。その結果復号化したクライアントは、別の hostname を得るため、その改ざんされたアドレスにアクセスを行ったり、DNS lookup やエラーメッセージなどを平文で送信したりする可能性がある。攻撃者は、アクセス先にトラップを仕掛けたり、送信された平文を盗聴したりすることで重要な情報を得ることが可能となる。本攻撃法は、IPSec などに対しては有効な攻撃方法であると提案されたが[26]、SSL3.0 は暗号化鍵をセッションごとに作成するため、攻撃が成功することはない。

2.2.7. Short-block attack

本攻撃法は Bellovin が IPSec に対して考案した攻撃法[26]であり、最後のメッセージブロックが 1 byte の平文とランダムパディングから構成されている場合の実行可能な攻撃法である。攻撃者は、上記のメッセージブロックを用意した既知の暗号文／平文対の暗号文で置き換える。データを受け取る側は、復号化しパディングを除いた状態で TCP チェックサムによって検証を行うため、もし使用した暗号文のメッセージブロックに相当する部分が正しくなければ、エラーパケットとして破棄される。推測した暗号文が正しかった場合には、ACK が返信されるため、これを利用して正しい平文を求めることができる。従って利用する既知暗号文／平文対の数は、高々2^8程度である。SSL3.0 においては、パディングはアプリケーションレイヤーで行われ、また ACK を返すような処理は行われないため、本攻撃はほとんど不可能であると考えられる。

2.2.8. Dropping the change cipher spec message

本攻撃方法は、change cipher spec、finished メッセージの認証対象データでないことを利用して、攻撃者がこのメッセージを削除することによって、送信先の cipher spec の更新を阻害し、finished メッセージやそれ以降のデータをメッセージ認証なしに設定する攻撃方法である。これは、Change cipher spec を受信しない場合、cipher spec はデフォルトの状態（暗号化、メッセージ認証ともに null）のまま更新されないためである。なお、本攻撃は master_secret を取得する攻撃ではないため、ciphersuite を“暗号化なし”のパラメータに設定した場合にのみ実行可能である。攻撃の手順は以下のようになる。（C をクライアント、S をサーバ、M を攻撃者とし、{}k は k を使用して MAC 値を付加することを表す。）

1. C→M：[change cipher spec]
2. C→M：[finished:]{a}k
2'. M→S：[finished:]a
3. S→M：[change cipher spec]
4. S→M：[finished:]{a}k
4'. M→C：[finished:]a
5. C→M：{m}k
5'. M→S：m

以上の手順によって、Client、Server 共に“メッセージ認証なし”の状態でメッセージを受信するため、攻撃者はクライアント、サーバ間でやりとりされるメッセージを自由に改ざんできるようになる。本攻撃は、常に“暗号化あり”の ciphersuite を選択することで防ぐことが可能である。なお、TLS においても同様の脅威が存在する。

2.2.9. Attack against finished messages

finished メッセージに中には master_secret が含まれているため、同時に多くのセッションを張り、finished メッセージを多数収集することで master_secret を推測できる可能性がある。finished メッセージは adhoc-MAC と呼ばれる MAC 計算によって求められているが、“暗号化なし”の ciphersuite を選択していた場合には、finished メッセージに含まれる master_secret 以外のパラメータは容易に推測可能であり、finished メッセージを収集することで master_secret を求めることができる可能性がある。TLS においては、adhoc-MAC は、HMAC と呼ばれる方式に変更されており、master_secret が攻撃者によって求められる可能性は、ほとんど無いと考えられる。

2.2.10. Traffic analysis

攻撃者が流れているデータを解析することによって、IP アドレス、URL のデータ長、html のデータ長を知ることが可能である。本攻撃は、暗号文を盗聴し、そのデータ長から平文のデータ長を推測することによって、重要なデータを推測する攻撃法である。SSL では、ブロック暗号においてはパディング長を設定するフィールドが存在するため、このフィールドを利用しパディング長をランダム化することによって攻撃を防ぐことが可能である。しかしながら、ストリーム暗号方式ではパディング長を設定する項目がないため、Traffic analysis によって暗号文のデータ長から平文のデータ長を推測することは容易に可能である。このことを利用して、攻撃者はＵＲＬ長、html ファイル長などの情報を知ることができる。これらの情報が漏洩することが直ちに重大な脅威とはなり得ないが、攻撃者にとっては情報取得の足がかりとなるため注意を要する。

2.2.11. Attack against a resuming session

１度確立されたセッションが休止状態になり、その後再び接続されるときには、新しい ClientHello.randam, ServerHello.randam が前セッションの master_secret を使用した MAC がクライアント、サーバ間で交換される。従って、master_secret を知らない攻撃者は、なりすまし攻撃を行うことはできないと考えられる。

2.3. 実装上のセキュリティホールとその対策(2002 年 3 月まで)

以下に現在報告されている実装上のセキュリティホールとその対策を示す。また、SSL やTLS というキーワードで検索されるが直接SSLの脆弱性ではないセキュリティホールも参考のため示す。

2.3.1. Interner Explorer HTTPS certificate attack

（Internet Explore rの HTTPS （SSL）に対する攻撃）

Bypassing Warning For Invalid SSL Certificates In Internet Explorer

攻撃をうけるソフトウェアおよびそのバージョン：

Microsoft Internet Explorer 4.0

Microsoft Internet Explorer5.0

Microsoft Internet Explorer6.0

内容：

攻撃者（不正なサーバ）はクライアントに対して不正な証明書の警告をなくすことができる。ここで不正な証明書とは、クライアントがアクセスするホスト名とは異なるホスト名が記載された証明書である。ただし、この証明書は正当な CA により発行されたものでなければならない。

詳細：

Internet Explorer における以下の２点の欠陥により攻撃が可能となる。

（１） 初めてホストに SSL を用いてアクセスするとき証明書の検証は行われるが、それ以降同一のホストに対して検証を行わない。

（２） HTML ファイルに含まれるイメージタグやフレームに HTTPS を用いるファイルがリンクされている場合、証明書におけるホスト名の検証を行わない。（証明書そのものの検証は行う。）例えば HTTP や HTTPS の通信において< img src = “ https://hogehoge.co.jp/whatever.gif “ width=1 height=1 >という記述によって対象のファイルが GET される場合、HTTPS 通信を行いサーバ証明書の正当性を検証するが、証明書に記載されているホスト名とアクセスするホスト名の比較を行わない。

攻撃者はクライアントに（２）を用いて不正な証明書を受理させる。次に（１）により不正な証明書をもつホストと SSL 通信を行わせることができる。

対策：

パッチを当てることにより、証明書記載のホスト名とアクセスするホスト名が異なる場合は警告を出すようにする。

2.3.2. Bypassing Warnings For Invalid SSL Certificates In Netscape Navigator

（Netscape Navigato rにおいて不正な SSL 証明書に対する警告を迂回する攻撃）

攻撃をうけるソフトウェアおよびそのバージョン：

Netscape Communicator 4.0

Netscape Communicator4.05

Netscape Communicator4.06

Netscape Communicator4.07

Netscape Communicator4.5

Netscape Communicator4.51

Netscape Communicator4.6

Netscape Communicator4.61

Netscape Communicator 4.7

Netscape Communicator, 4.72

内容：

攻撃者（不正なサーバ）はクライアントに対して、不正な証明書の警告をなくすことができる。不正な証明書とは、クライアントがアクセスするホスト名とは異なるホスト名が記載された証明書である。ただし、この証明書は正当な CA により発行されたものでなければならない。

詳細：

Netscape Navigator における以下の欠陥により攻撃が可能となる。

（1） 初めてホストに SSL を用いてアクセスするとき証明書の検証が行われるが、それ以降同一のホストに対して検証を行われない。

（2） 証明書に記載されたホスト名と、クライアントがアクセスするホスト名の IP アドレスを比較するが、ホスト名の比較が行わない。

攻撃者はあらかじめ１つの IP アドレスに複数のホスト名を割り当てる。そして、（2）により特定の IP アドレスにアクセスさせて証明書を受理させ、（1）によりクライアントに IP は上記のホストと等しいが、異なるホスト名が記載された証明書をもつホストと HTTPS 通信を行わせることができる。ただし、攻撃者は DNS スプーフもしくは不正な DNS サーバを用いなければならない攻撃できない。

対策：

パッチを当てることにより、認証を行う際の確認を IP アドレスではなくホスト名で行うようにする。

2.3.3. Multiple Vendor SSL Certificate Validation Vulnerability

（複数のブラウザにおける SSL 認証の脆弱性）

攻撃を受けるソフトウェアおよびそのバージョン：

Links Links0.96

University of Kansas Lynx2.7

University of Kansas Lynx 2.8

University of Kansas Lynx 2.84

W3M W3M 0.1.3

W3M W3M 0.1.4

W3M W3M 0.1.6

W3M W3M 0.1.7

W3M W3M 0.1.8

W3M W3M 0.1.9

W3M W3M 0.1.9

W3M W3M 0.1.10

W3M W3M 0.2

W3M W3M 0.2.1

W3M W3M 0.2.2

W3M W3M 0.2.3

内容：

W3M,Lynx,Links などブラウザの初期のバージョンは SSL が実装されているが証明書検証機能が実装されていない。このことから攻撃者は正規のサーバになりすますことができる。また、man-in-the-middle attack も可能である。

対策：

認証機能を実装する。

2.3.4. RSA BSAFE SSL-J Authentication Bypass Vulnerability

（RSA BSAFE SSL-J における認証迂回の脆弱性）

攻撃をうけるソフトウェアおよびそのバージョン：
Cisco iCDN 2.0
RSA Security BSAFE SSL-J SDK 3.0
　　+ Cisco iCDN 2.0
RSA Security BSAFE SSL-J SDK 3.0.1
RSA Security BSAFE SSL-J SDK 3.1
Cisco iCDN 2.0.1 には脆弱性はない

内容：
SSL においてクライアント認証が行われる場合、攻撃者はクライアント認証を迂回しサーバにアクセスすることができる。

詳細：
RSA BSAFE は RSA 社によって開発された暗号化ソフトウェア開発環境である。この問題は BSAFE-J をもちいて SSL 通信を行うプログラムを作成し、プログラムの中でクライアント認証を行う場合に発生する。SSL において同一のクライアントによる連続したセッションは一度クライアント認証が行われるが、それ以降の通信においてキャッシュされ留ため、クライアント認証が行われない。RSA BSAFE-J3.x はこの SSL セッションキャッシュに不具合があり、認証されていないクライアントが認証されているクライアントになりすまし、認証されたクライアントだけにアクセスを許すデータにアクセスできる可能性がある。セッションキャッシュの不具合は、クライアントが handshake しているときエラーが起こった場合に発生する。エラーが起こったときセッション ID は破棄されるべきであるがキャッシュに保存されてしまう。そして、次にクライアントが SSL によってアクセスする際にクライアント認証が行われずに通信が開始される。

対策：
不具合の修正

2.3.5. OpenSSL PRNG Internal State Disclosure Vulnerability

（OpenSSL の疑似乱数生成器（PRNG）の内部状態を暴露する攻撃）

攻撃をうけるソフトウェアおよびそのバージョン：

OpenSSL Project OpenSSL 0.9.1c

OpenSSL Project OpenSSL 0.9.2b

OpenSSL Project OpenSSL 0.9.3

OpenSSL Project OpenSSL 0.9.4

OpenSSL Project OpenSSL 0.9.5

OpenSSL Project OpenSSL 0.9.6 - Frederik Vermeulen QMail tls.patch

OpenSSL Project OpenSSL 0.9.6

SSLeay SSLeay 0.8.1

SSLeay SSLeay 0.9

SSLeay SSLeay 0.9.1

内容：

攻撃者が SSL に用いられる乱数生成器の内部状態を知ることができる。内部状態から疑似乱数を推測しセッション鍵を推測できる可能性がある。ただし、OpenSSL を用いた一般的なアプリケーションにおいてこの問題は発生しない。

詳細：

SSL の実装において乱数生成のために疑似乱数生成器を用いる。上記の OpenSSL における疑似乱数生成器には設計上の欠陥が存在しており、攻撃者は 1byte の PRNG 要求を数百回を行うことにより PRNG の内部状態を再構成することができる。内部状態は２つの変数‘md’，‘state’によって決められる。‘md’はハッシュ関数の出力によって決められる連鎖的な値である（160bit）。そして‘state’は‘md’より大きな値であり暗号化されている。疑似乱数を生成する際に‘md’は以前の値の半分と‘state’の一部からなる値をハッシュすることにより得られる。しかし‘md’生成の入力における半分の値は PRNG の出力の半分の値と同じ値である。また、‘state‘から得られる値は、PRNG 出力として要求されるバイト数に依存するものである。このことから攻撃者は簡単な brute-force 解析により内部状態を再構成できる。

対策：

‘md’を更新する際に以前の‘md’の値をすべてハッシュする。また、‘state’からの値は PRGN に要求されるバイト数と独立な値とする。

2.3.6. Microsoft IE SSL Spoofing Vulnerability

(Internet Explorer の SSL なりすまし攻撃)

攻撃をうけるソフトウェアおよびそのバージョン：
Microsoft Internet Explorer 5.01
Microsoft Internet Explorer 5.0.1SP1
Microsoft Internet Explorer 5.0.1SP2
Microsoft Internet Explorer 5.5
Microsoft Internet Explorer 5.5SP1
Microsoft Internet Explorer 5．5ＳＰ以降問題はない。

内容：
攻撃者は他のサイトになりすますことができる。

詳細：
Internet Explorer を用いてブラウジングする場合、ユーザはアクセスしているサイトの情報をアドレスバーから知ることになる。しかし、上記の Intermet Explorer はフレームや JavaScript などを用いることによりアドレスバーに任意の文字列を表示できる欠陥が存在する。これにより攻撃者は、見かけ上別のサイトになりすますことができる。よって、攻撃者はアドレスバーのアドレスと SSL の証明書に記載されているアドレスが異なる SSL 通信を行える。

対策：
不具合の修正

2.3.7. Netscape Communicator Inconsistent SSL Certificate Warning Vulnerability

(Netscape Communicator の不正な SSL 証明書に対する警告を迂回する攻撃)

攻撃をうけるソフトウェアおよびそのバージョン：

Netscape Communicator 4.0

Netscape Communicator 4.5

Netscape Communicator 4.5.1

Netscape Communicator 4.6

Netscape Communicator 4.6.1

Netscape Communicator 4.7

Netscape Communicator 4.7.2

Netscape Communicator 4.7.3

内容：

攻撃者は Netscape Navigato r を使用するクライアントに対して不正な SSL の証明書の警告をなくすことができる。

詳細：

上記のバージョンの Netscape Navigator は不正なサーバ証明書を受け取ったとき

“hostname does not match name in certificate″

という警告を表示する。しかし、このときユーザが”continue”ボタンをクリックすると、それ以降その Netscape のセッションにおいて、ホスト名や IP アドレスが異なる証明書を受理してしまう。

対策：

不具合の修正

2.3.8. OpenSSL Unseeded Random Number Generator Vulnerability

(OpenSSL の seed をなしの疑似乱数の脆弱性)

攻撃をうけるソフトウェアおよびそのバージョン：

OpenSSL Project OpenSSL 0.9.1c

OpenSSL Project OpenSSL 0.9.2b

OpenSSL Project OpenSSL 0.9.3

OpenSSL Project OpenSSL 0.9.4

OpenSSL Project OpenSSL 0.9.5 において修正されている。

内容：

攻撃者が SSL に用いられる乱数生成器において seed によって初期化されない可能性がある。これにより、疑似乱数を推測しセッション鍵を推測できる可能性がある。

詳細：

OpenSSL はクライアントがサーバと SSL/TLS の handshake を初期化する時に SSL_connect() という関数を用いる。この関数は疑似乱数生成器を seed によって確実に初期化しないため完全な疑似乱数生成器ではなくなってしまう。この問題は qmail の非公式パッチ tls.patch において知られている。このパッチにおいて乱数生成器は seed を使っていない。

対策：

SSL_connect()において疑似乱数生成器を seed によって初期化する。

2.3.9. IIS / Site Server Multithread SSL Vulnerability

(ISS サーバのマルチスレッド SSL における脆弱性)

攻撃をうけるソフトウェアおよびそのバージョン：

Microsoft IIS 4.0
+ Cisco Building Broadband Service Manager 5.0
+ Cisco Call Manger 1.0,2.0,3.0
+ Cisco ICS 7750
+ Cisco IP/VC 3540
+ Cisco Unity Server 2.0,2.2,2.3,2.4
+ Cisco uOne 1.0,2.0,3.0,4.0
+ Microsoft BackOffice 4.0,4.5
+ Microsoft Windows NT 4.0 Option Pack

Microsoft Site Server Commerce Edition 3.0
- Microsoft IIS 4.0
- Microsoft Windows NT 4.0

Microsoft Site Server Commerce Edition 3.0
+ Microsoft BackOffice 4.0,4.5
+ Microsoft Commercial Internet System 2.0
+ Microsoft Site Server Commerce Edition 3.0

内容：
特定の条件下において、サーバがクライアントに暗号文を平文のまま送ってしまう。

詳細：
IIS4 の SSL ISAPI フィルタにはクライアントに暗号文を平文のまま送ってしまう脆弱性がある。この欠陥はフィルタが同じスレッドで動作していることが原因である。高負荷状態において、複数スレッドのクライアントアプリケーションが接続した場合、サーバは暗号化せずにデータを送信してしまい接続を終了する。この際に、もし攻撃者が通信を盗聴していたならば、通信を平文のまま受け取ることができる。

対策：
不具合の修正

2.3.10. NT IIS SSL DoS Vulnerability

(NT 上の IIS において SSL によるの DoS 攻撃が可能になる脆弱性)

攻撃をうけるソフトウェアおよびそのバージョン：

Microsoft IIS 3.0

- Microsoft Windows NT 4.0
- Microsoft Windows NT 4.0SP1,SP2,SP3,SP4,SP5,SP6,SP6a

Microsoft IIS 4.0

+ Cisco Building Broadband Service Manager 5.0
+ Cisco Call Manger 1.0,2.0,3.0
+ Cisco ICS 7750
+ Cisco IP/VC 3540
+ Cisco Unity Server 2.0,2.2,2.3,2.4
+ Cisco uOne 1.0,2.0,3.0,4.0
+ Microsoft BackOffice 4.0,4.5
+ Microsoft Windows NT 4.0 Option Pack

内容：

NT サーバ上の SSL を実装した IIS は DoS 攻撃を受ける可能性がある。

詳細：

上記の IIS は SSL 通信を要求されたページとそうでないページの違いを区別できない．これにより URL の‘http’という文字列を‘https’に変えることによりサーバはすべてのコンテンツを暗号化しようとする。このように攻撃者が https 要求をすることによりサーバの CPU 使用率を 100%とし、サーバを極端な速度低下や機能停止にさせることができる。

対策：不具合の修正

2.3.11. Netscape Navigater SSL における疑似乱数の seed によるセッション鍵の推定による攻撃

攻撃をうけるソフトウェアおよびそのバージョン：
Netscape Navigator 1.1
Netscape Navigator2.0 以降には問題はない。

内容：
攻撃者はクライアント・サーバ間の通信を盗聴することによりセッション鍵を推定できる可能性がある。これにより通信の内容を知ることができる可能性がある。

詳細：
この攻撃は 1996 年に Ian Goldberg と David Wagner によって発見された。Netscape Navigator の SSL において、challenge-data と secret を生成するための使われる乱数生成器の seed は時間の秒とマイクロ秒、pid と ppid の組み合わせにより作られる。以下に疑似乱数生成器の seed を生成する関数と seed から challenge message や秘密情報を生成する関数を示す。

```
NG_CreateContext(){
    (seconds, microseconds) = time of day;
                                    /* Time elapsed since 1970
*/
    pid = process ID;  ppid = parent process ID;
    a = mklcpr(microseconds);
    b = mklcpr(pid + seconds + (ppid << 12));
    seed = MD5(a, b);
 }
mklcpr(x)
 /* not cryptographically significant; shown for compl eteness
*/{
    return ((0xDEECE66D * x + 0x2BBB62DC) >> 1);
}
```

図 2.3.11.1.　Seed を生成する関数

```
RNG_GenerateRandomBytes()
     x = MD5(seed);
     seed = seed + 1;
     return x;

global variable challenge, secret_key;

create_key()
     RNG_CreateContext();
     tmp = RNG_GenerateRandomBytes();
     tmp = RNG_GenerateRandomBytes();
     challenge = RNG_GenerateRandomBytes();
     secret_key = RNG_GenerateRandomBytes();
```

図 2.3.11.2.　Challenge message と秘密情報を生成する関数

攻撃者が Netscape Navigato r が起動されているマシンにアカウントをもつ場合、時間は time デーモン、daytime デーモンや tcpdump の値から推測できる。また、pid, ppid は ps コマンドにより容易に得ることができる。よって、攻撃者はマイクロ秒の推測(10^6 回の計算)程度で seed を推測で着る。

また、攻撃者が Netscape Navigator が起動されているマシンにアカウントを持たない場合、以下の情報により攻撃者は brute force より大幅に少ない計算量で seed を推測できる。

- 時間は 20bit であり、pid, ppid は 15bit であり、(pid+(ppid<<12)) は 27bit である。これらをあわせると 47bit となり、128bit に比べ大幅に少ない。
- 攻撃者は時間をある程度知ることができる。
- ppid は X-windows のメニューから起動された場合 1 である。
- ppid が 1 ではない場合 pid より少し小さい値である。
- 個人用のワークステーションの場合 pid や ppid は小さい値である。
- 攻撃されるホストに sendmail が起動されている場合その pid は message-ID に含まれることが多い。この pid をもちいることにより Netcape Navigator の pid 推測が少し容易になる。
- MS-DOS の場合 pid、ppid はない。よって seed に使われていない可能性がある。

疑似乱数生成器は接続されている間 seed を変えない。

対策：疑似乱数生成アルゴリズムの変更

2.3.12. Microsoft IE 4.x, 5.x における Cached Web Credentials 問題

攻撃を受けるソフトウェアおよびそのバージョン：
Microsoft Internet Explorer 4.x
Microsoft Internet Explorer 5.x prior to version 5.5

内容：
SSL 通信の場合のみに用いられるユーザ ID とパスワードが盗まれる可能性がある。

詳細：
基本認証を使用して、ユーザがセキュリティで保護された Web ページを認証する場合、IE は使用されたユーザ ID とパスワードをキャッシュする。この機能により、ユーザが同じサイトを認証するときに、ユーザ ID とパスワードを入力する手間を省くことができる。この ID とパスワードは暗号化されていない場合は平文で通信路を送信されることになる。一般的に安全なサイトは SSL をもちいることによりパスワードが平文で送信されることがないように設計されている。しかし、上記のバージョンの IE は HTTP と HTTPS の区別をせずに ID とパスワードを送信するため、HTTPS において暗号化されて送信されるべき ID とパスワードが HTTP の場合でも送信されてしまう。攻撃者は HTTPS 通信を HTTP 通信に置き換えることができた場合、ネットワークを盗聴することによりパスワードを盗むことができる。

対策：
不具合の修正

2.3.13. その他 SSL というキーワードで検索されるが直接 SSL とは関係のないセキュリティホール

以下に Cert/CC などセキュリティサイトにおいて SSL というキーワードで検索した場合検索されるが、SSL や TLS のセキュリティホールとは関係のないセキュリティホールを示す。

（1）CERT/CC SSH デーモン，RSAREF2 ライブラリ，Buffer Overflow 問題

【概要】

RSAREF2 中の脆弱性により，ネットワーク経由で任意のコードを実行することが可能となる

【出典】

CA-1999-15 (December 13, 1999), CIAC K-011 (December 21, 1999)

http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0834

【解説】

RSAREF2 の脆弱性は暗号ライブラリの問題であり SSL 固有の問題ではない。また、上記の脆弱性は Core SDI (http://www.core-sdi.com) によって報告されたものであるが、このときのサンプルとしては sshd が用いられており、実行されるための条件として（1）sshd が実行されていること（2）sshd のライブラリとして RSAREF2 が用いられていることの 2 点が挙げられている。つまり、sshd の脆弱性と RSAREF2 の脆弱性を組み合わせなければこの攻撃を実行することはできない。もし仮に SSL の実装において不具合があり、かつ RSAFER2 を用いた場合に脆弱性が発生する可能性があるが、そのような例は確認されていない。

（2） Microsoft IIS 4.0/5.0 Session ID Cookie生成における脆弱性

【概要】

SSL で保護されたサイトに対する Session ID cookie を，悪意のあるユーザに不正に取得されてしまう.

【出典】

MS00-080 (October 23, 2000), CIAC L-010 (October 24, 2000)

【解説】

この脆弱性は IIS のセッション ID(SSL のセッション ID とは異なるもの)の不具合である。IIS がセッション ID を Cookie を用いて管理する場合、HTTP と HTTPS を区別しないため HTTPS で暗号化して共有したセッション ID がその後もし HTTP で通信した場合、平文で送信される脆弱性である。ただし、このセッションＩＤは SSL のセッション ID とはことなるものである。

（3） “Microsoft Corporation”の名を騙った証明書

【概要】

VeriSign 社から, 2001 年 1 月 29 日および 30 日, VeriSign Class 3 (VeriSign Commercial Software Publishers CA)により署名された"Microsoft Corporation" の名前の記載された偽のデジタル証明書が 2 枚発行された.

【出典】

CA-2001-04 (March 22, 2001)

【解説】

この偽のデジタル証明書は SSL においてもちいられる証明書ではなく、マイクロソフトのコード署名にもちいられる証明書である。つまり、IE を利用しているときにマイクロソフトによって署名されたコード（実行可能なプログラム）とみせかけた任意のコードを作成することができる脆弱性である。ただし、前述のように SSL には一切関係がない。

（4）Webシステムにおける不正なHTMLタグ問題

【概要】

SSL セッションにおいて, 正規の HTML タグ(HTML を記述するための符号)の中に, 悪意を持った HTML タグ(例えば, Java スクリプトなど)を紛れ込ませる.

【出典】

CA-2000-02 (February 2, 2000), CIAC K-021 (February 3, 2000)

【解説】

これは一般的にクロスサイトスクリプティングと呼ばれる問題である。SSL の有無に関係なくこの脆弱性は存在する。クロスサイトスクリプティングの詳細は本書の範囲外であるので記載しない。

（5）Cisco VPN3000 Concentrator における TELNET の脆弱性

【概要】

Cisco VPN 3000 series concentrators 上で動作する SSL と telnet サービスの脆弱性により, DoS 攻撃をかけられるとシステムがリブートする.

【出典】

CIAC L-068 (April 6, 2001)

【解説】

CiscoVPN が SSL と Telnet を正しく処理できない脆弱性であり、SSL のプロトコルや実装とは直接関係ない。

2.4. 実装上のセキュリティホールとその対策(2002 年 3 月以降)

2.4.1. ASN.1 に関するセキュリティホール

[概要]

ASN.1(Abstract Syntax Notation 1)は通信プロトコルを記述するために用いられるものである。SSL/TLS において、X.509 証明書のエンコードやプロトコル記述に用いられている。ASN.1 をバイナリデータにエンコードする方法として BER(Basic Encoding Rules)、CER(Canonical Encoding Rules)、DER(Distinguished Encoding Rules)などが存在する。一般に用いられる BER、DER は、将来の拡張にも対応するため、汎用的であるが実装には適していない。そのため、実装上においてセキュリティホールを誘発する原因となる。

[詳細]

- 背景

ASN.1 の脆弱性はフィンランドの Oulu 大学セキュアプログラミンググループ(University Secure Programming Group：OUSPG)によって発見された[57]。OUSPG ではプロトコルの様々なフィールドに例外データを挿入することにより、プロトコル実装の頑強性を検査する研究を行っている。対象プロトコルとして、WAP、HTTP、LDAPv3、SNMPv1、SIP などの調査を行い、脆弱性を発見している。特に、SNMP(Simple Network Management Protocol)に関して、MIB 情報に対しても調査を行った。MIB 情報は ASN.1 により記述される OID(Object Identifier)である。MIB 情報に対する調査の結果、複数の脆弱性が発見された。前述のように ASN.1 は SNMP の MIB だけでなく幅広く利用されており、X.509 の証明書にも利用されている。そこで、他の ASN.1 により記述されるプロトコルににおいて同様の脆弱性の存在が危惧されている。

- ASN.1 の BER エンコード問題点

ASN.1 は ISO と ITU-T によって規定されており。本来の目的は、通信のプロトコルを記述するための PDU(Protocol Data Unit)の定義であるが、階層的なデータの表現のために使用できる。BER は汎用的なエンコードとするため、各フィールドのサイズが可変である。可変サイズのフィールドは汎用的にエンコードする方法としては有効であるが、実装においては有効ではない。フィールドが可変長の汎用的な ASN.1/BER エンコーダー・デコーダーを実装する場合、注意深く実装しなければ、特定の入力に対してリソースが枯渇し、DoS(Denial of Service)状態や BoF(Buffer Overflow)が発生する可能性がある。

- ASN.1 における BER によるエンコードの詳細

ASN.1 の BER によるエンコードは図 2.4.1 のように規定される。データの種類(Identifier)、データの長さ(Length)、データの内容(Contents)が順に格納される。Identifier octets にはエンコードされた ASN.1 のタグ、Length octets にはエンコードされた Contents

octets のバイト数など,Contents octets にはエンコードされたデータ、End-of-contents octets は Length octets が Infinite 形式の場合存在し、規定のエンコードをされたデータが入る。Figure2 形式を使用する場合、End-of-content octets まで任意のサイズの content を格納することができるためサイズが可変である。また、Identifier octets は図に示すように Figure 3 の low tag number (31 より小さい場合)ではサイズが固定であるが、Figure 4 の high tag number ではエンコードすることにより任意のサイズを指定できる。さらに、Length octets は short form (127 より小さい)ではサイズが固定であるが、long form ではエンコードすることにより任意のサイズを指定できる。

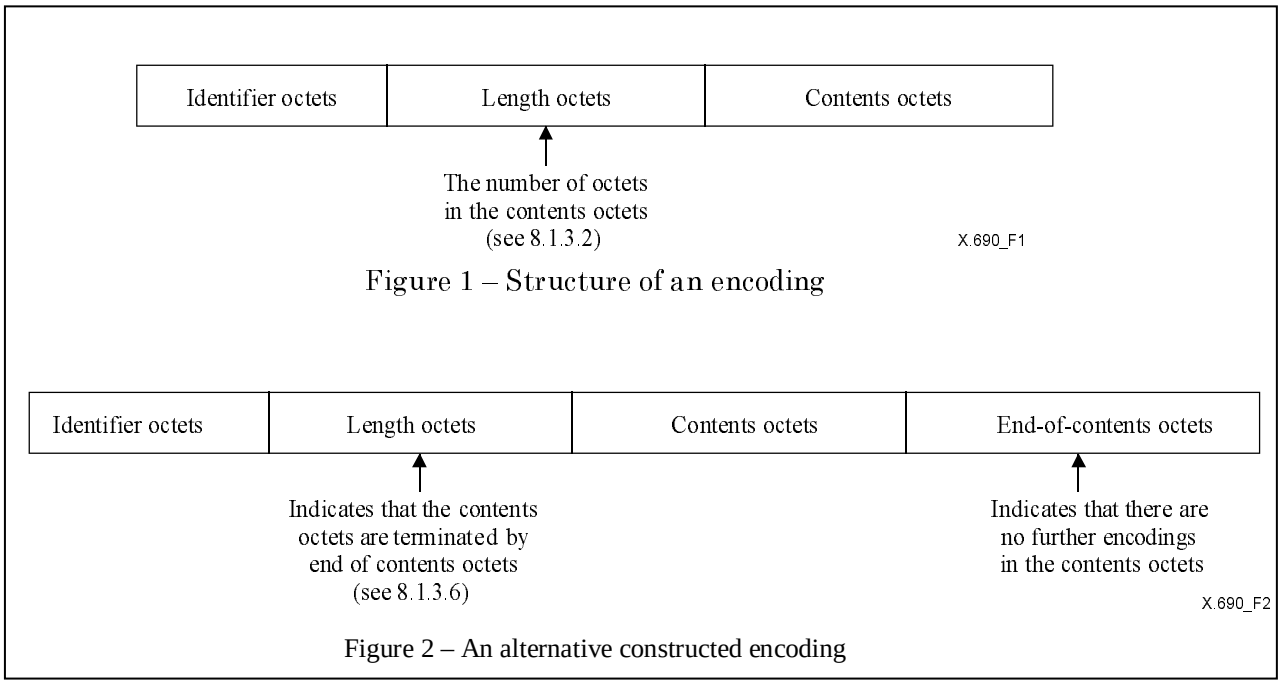


図2.4.1 ASN.1 BER エンコードの構造[58]

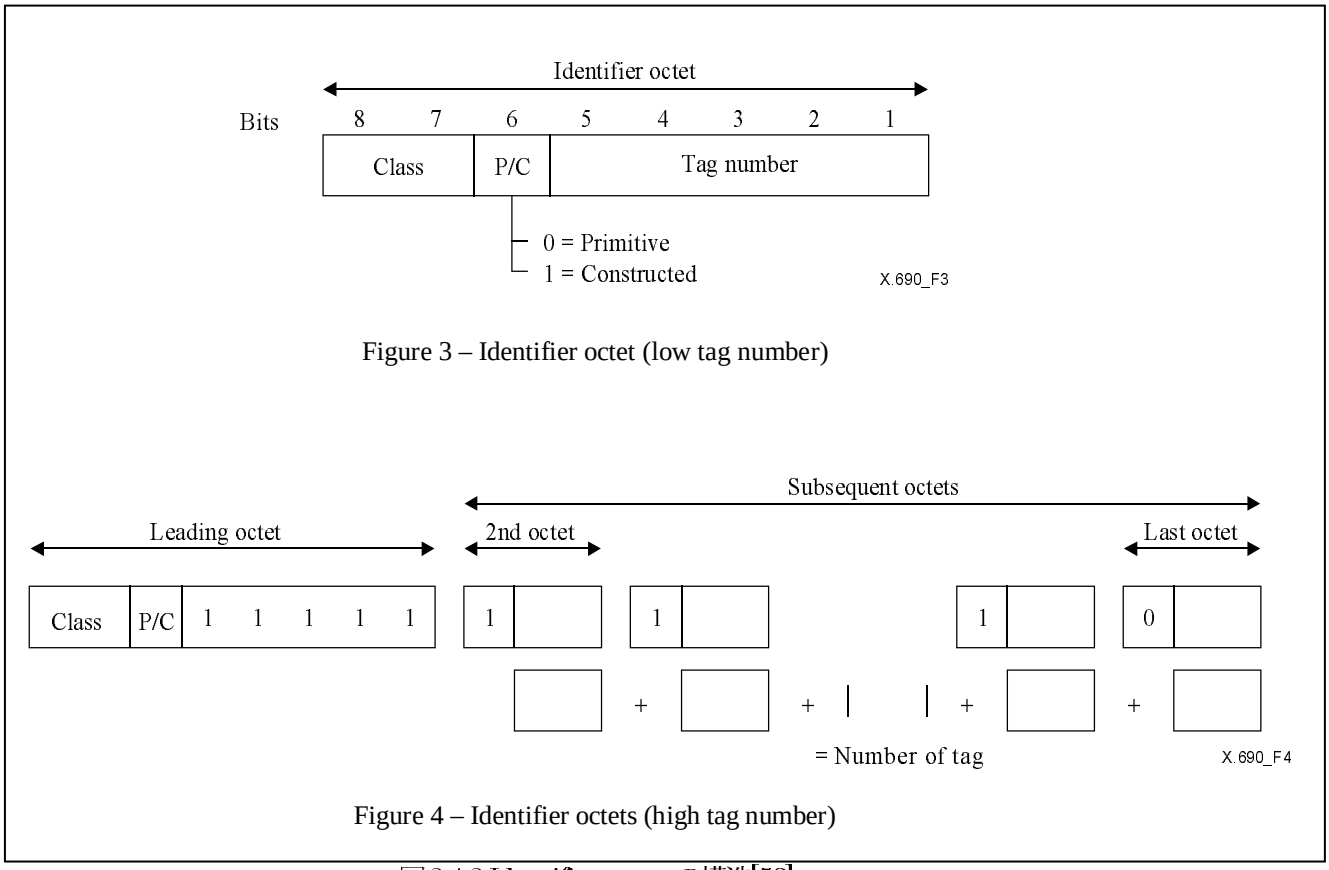


図 2.4.2 Identifier octets の構造[58]

- OpenSSL の ASN.1 脆弱性に関して

OpenSSL における ASN.1 の BER エンコードにおける脆弱性は複数回報告されている。はじめに、2002 年 7 月 30 日に Adi Stav <stav@mercury.co.il>, James Yonan <jim@ntlp.com>それぞれによって個別に発見された。OpenSSL の Adversary[59]によると、脆弱性は以下のとおりである。

Length octetsがC言語におけるデータのサイズとしてsizeof(long)に示されるサイズを越すとき、asn1_get_length()関数が与えられたバッファーの大きさを越えてしまうことにより、DoS を発生する。

次に、2003 年 9 月 30 日に NISCC によって発見された[60]。OpenSSL の Adversary[61]によると、脆弱性の詳細は以下の 4 点である。1,2,3 はクライアント証明書に関するものであるため、リモートからの実行は不可能であるが、4 によりリモートから SSL 接続できるサーバに対して攻撃が可能となる。

1. ASN.1 の構造としては正しいが ASN.1 パーザーが無効と判断する入力が行われたとき、該当する構造体の一部が正しく開放されない。そのため、DoS 状態に陥らせることができる。また、任意のコードの実行ができる可能性があると報告されている。
2. long form のタグを正しく解釈できないため、境界を超えて読み取りが実行される。そのため、リモートからサーバを DoS 状態に陥らせることができる。
3. デバッグモードにおいて公開鍵のデコードエラーを無視するように設定しているとき、不正な公開鍵証明書の公開鍵に対するエラー確認が行われない。そのため、不正な公開鍵(NULL)によりサービス不能に陥らせることができる。ただし、通常はデバックモードに設定されていない。
4. プロトコルハンドリングにおけるエラーによって、サーバは特別要求しないときにクライアント証明書をパーズする。これ自身は特に言及される脆弱性ではないが上記の脆弱性を用いることにより、クライアント認証を enable にしない SSL/TLS サーバにも攻撃ができる。

さらに、2003 年 11 月 4 日 NISCC によって以下の脆弱性が報告された[62]。OpenSSL の Adversary[63]によると、脆弱性は以下のとおりである。OpenSSL では任意のコードを実行できることはないと主張している[63]が、NISCC では任意のコードを実行できる可能性があると主張している[62]。

ASN.1 としては構造としては正しいが、大きな繰り返し構造を正しく処理できない。そのため、OpenSSL におけるクライアント証明書などの ASN.1 構造においてそのような値を設定することにより、リモートから SSL/TLS サーバをサービス不能に陥らせることができる。

このように OpenSSL において ASN.1 に関する脆弱性は度重なり報告されている。前述のように、ASN.1 のエンコードはセキュリティホールを発生させやすいため、今後も発生する可能性がある。

- 攻撃の実行可能性

Buffe Overflow による脆弱性は理論上攻撃が成功する可能性があれば報告されるが、すべての攻撃が実行可能であるとは限らない。上記脆弱性に関しては、任意のコードを実行できる exploit code(セキュリティホールを実証するためのプログラム)は今回の調査の範囲では公開されていないが、DoS を発生させる exploit code は 2004 年 1 月 15 日 bugtraq メーリングリストに投稿された[64]。また、任意のコードを実行できる exploit c o d e は存在しないことが証明されたわけではないため予断を許さない。

- SSL/TLS 以外のプロトコルにおける問題

ASN.1 は SNMP、SSL/TLS だけでなく広く利用されている。SNMP の MIB および SSL/TLS 以外の脆弱性として SIP(Session Initiation Protocol)[65] および

SMIME(PKCS#7)[66]において報告されている。また、今後も ASN.1 を利用する他のプロトコルにおいて脆弱性が発見される可能性が高い。さらに、Microsoft 社の OS である Windows においても脆弱性が発見された。[67]

[対策]

脆弱性が発見された場合、脆弱性に対処するための修正プログラムが公開される。したがって、以下の 3 点の対策をすべきである。

- 脆弱性に関する情報を継続的に収集すること。
- 脆弱性修正プログラムが存在する場合は早期に適用すること。最新バージョンのソフトウェアを利用すること。
- 脆弱性を利用した攻撃が行われたかどうか監視すること。

また、SSL/TLS を用いて認証するとき、認証相手に脆弱性が存在するとき最悪の場合には秘密鍵が漏洩している可能性があるため、認証結果は信用できない。

2.4.2. Buffer Overflow による脆弱性

[概要]

Buffer Overflow による脆弱性は、主に C 言語におけるプログラム不具合が原因となり、DoS を発生させることや任意のコードを実行させることができる。前回報告以降にも多くの実装において同種の脆弱性が発見されている。

[詳細]

- 脆弱性リスト

前回以降 2002 年 1 月から 2004 年 1 月まで報告された脆弱性として、に bugtraq[69] に登録されている SSL に関する脆弱性を表に示す。本稿ではそれらの攻撃を以下のように分類した。この中で Buffer Overflow など実装におけるメモリ関連の不具合が 10 件報告されている。このような不具合により DoS を発生させることや任意のコードを実行することができる可能性がある。

種類	内容
DoS	Buffer Overflow などにより DoS を発生する攻撃
Execute	Buffer Overflow などにより任意のコードを実行する攻撃
ASN.1	2.4.1.節にて説明した攻撃
Side channel	2.1.5 節から 2.1.8.節にて説明した攻撃
Other	その他の攻撃

- Buffer Overflow の問題点

Buffer Overflow が存在し、任意のコードが実行できるとき、プロトコル実装における仮定が崩れるため、安全性は保証されない。そして、最悪の場合サーバの秘密鍵が漏洩する可能性がある。

- 攻撃の実行可能性

前述のように Buffer Overflow の脆弱性存在し、任意のコードが実行できる可能性が報告されたとしても、攻撃が実行可能とは限らない。しかしながら、上記リストの脆弱性のなかで任意のコードを実行する Exploite code などが公開されている[70] [71]。実際にこのコードをコンパイルすることにより特定のバージョンの SSL/TLS サーバに対して任意のコードを実行できることが確認されている。さらに、このコードを利用したワーム Slapper[72] も確認されている。

[対策]

脆弱性が発見された場合、脆弱性に対処するための修正プログラムが公開される。したがって、以下の 3 点の対策をすべきである。

- 脆弱性に関する情報を継続的に収集すること。
- 脆弱性修正プログラムが存在する場合は早期に適用すること。最新バージョンのソフトウェアを利用すること。
- 脆弱性を利用した攻撃が行われたかどうか監視すること。

また、SSL/TLS を用いて認証するとき、認証相手に脆弱性が存在するとき最悪の場合には秘密鍵が漏洩している可能性があるため、認証結果は信用できない。

表 2.4.1 bugtraq に投稿された脆弱性一覧

bid	name	url	cve	published	category
9376	Jabber Server SSL Handling Denial of Service Vulnerability	http://www.securityfocus.com/bid/9376	CAN-2004-0013	7-Jan 2004	remote DoS
8970	OpenSSL ASN.1 Large Recursion Remote Denial Of Service Vulnerability	http://www.securityfocus.com/bid/8970	CAN-2003-0851	4-Nov 2003	ASN.1
8746	OpenSSL SSLv2 Client_Master_Key Remote Denial Of Service Vulnerability	http://www.securityfocus.com/bid/8746	CVE-MAP-NOMATCH	2-Oct 2003	remote DoS
8732	OpenSSL ASN.1 Parsing Vulnerabilities	http://www.securityfocus.com/bid/8732	CVE-MAP-NOMATCH	30-Sep 2003	ASN.1
8134	Apache Web Server SSLCipherSuite Weak CipherSuite Renegotiation Weakness	http://www.securityfocus.com/bid/8134	CAN-2003-0192	8-Jul 2003	Other
7148	OpenSSL Bad Version Oracle Side Channel Attack Vulnerability	http://www.securityfocus.com/bid/7148	CAN-2003-0131	19-Mar 2003	Side channel
7101	OpenSSL Timing Attack RSA Private Key Information Disclosure Vulnerability	http://www.securityfocus.com/bid/7101	CAN-2003-0147	14-Mar 2003	Side channel
6884	OpenSSL CBC Error Information Leakage Weakness	http://www.securityfocus.com/bid/6884	CAN-2003-0078	19-Feb 2003	Other
5875	Ximian Evolution SSL Man-In-The-Middle	http://www.securityfocus.com/bid/5875	CAN-2002-1471	3-Oct 2002	Other
5791	HP VirtualVault Apache mod_ssl Denial Of Service Vulnerability	http://www.securityfocus.com/bid/5791	CVE-MAP-NOMATCH	25-Sep 2002	remote DoS
5778	Microsoft Internet Explorer SSL Certificate Expiration Vulnerability	http://www.securityfocus.com/bid/5778	CVE-MAP-NOMATCH	23-Sep 2002	Other
5690	ssldump PreMasterSecret Buffer Overflow Vulnerability	http://www.securityfocus.com/bid/5690	CVE-MAP-NOMATCH	11-Sep 2002	Execute
5693	ssldump SSLv2 Challenge Buffer Underflow Vulnerability	http://www.securityfocus.com/bid/5693	CVE-MAP-NOMATCH	11-Sep 2002	Execute
5361	OpenSSL Kerberos Enabled SSLv3 Master Key Exchange Buffer Overflow Vulnerability	http://www.securityfocus.com/bid/5361	CAN-2002-0657	30-Jul 2002	Execute
5362	OpenSSL SSLv3 Session ID Buffer Overflow Vulnerability	http://www.securityfocus.com/bid/5362	CAN-2002-0656	30-Jul 2002	Execute
5363	OpenSSL SSLv2 Malformed Client Key Remote Buffer Overflow Vulnerability	http://www.securityfocus.com/bid/5363	CAN-2002-0656	30-Jul 2002	Execute
5364	OpenSSL ASCII Representation Of Integers Buffer Overflow Vulnerability	http://www.securityfocus.com/bid/5364	CAN-2002-0655	30-Jul 2002	Execute
5366	OpenSSL ASN.1 Parsing Error Denial Of Service Vulnerability	http://www.securityfocus.com/bid/5366	CAN-2002-0659	30-Jul 2002	ASN.1
4189	Apache mod_ssl/Apache-SSL Buffer Overflow Vulnerability	http://www.securityfocus.com/bid/4189	CVE-2002-0082	27-Feb 2002	Execute
3803	Multiple Vendor SSL Certificate Validation	http://www.securityfocus.com/bid/3803	CVE-MAP-NOMATCH	3-Jan 2002	Other

2.4.3. ルート証明書配送の問題

[概要]

SSL/TLS に付随する問題として、SSL/TLS に用いる証明書の管理の問題がある。Microsoft 社の Windows ＯＳはソフトウェアの不具合を修正するために Windows Update という機能を備えている。この機能はコンピュータおよびソフトウェアの状態をスキャンし、ソフトウェアの不具合修正や機能追加を自動的に行う機能である。Windows の最新バージョンである Windows XP 以降では、SSL/TLS にも利用されるルート証明書を Windows Update を利用して配信している。

[詳細]

- Windows update によるルート証明書配送

Microsoft 社の技術解説サイト TechNet に以下のニュースが掲載された[75][76] 。記事の内容を以下に転載する。つまり、SSL/TLS に利用される公開鍵証明書のルート証明書が Windows Update によって配信される。しかも、この配信がセキュリティダイアログや警告なしに行われる。

> ルート配信プロセスにおける変更
>
> 新しいルート証明書の従来の登録方法は、Microsoft Internet Explorer ではもう使用できません。マイクロソフトによって承認された新しいルートはすべて、Windows Update を通じて Windows XP クライアントで利用することができます。ユーザがセキュリティ保護された Web サイト (つまり HTTPS を使用) にアクセスしたり、セキュリティ保護された電子メール (つまり S/MIME) を読んだり、 あるいは新しいルート証明書を使用する Active X コントロールをダウンロードしたりする場合、Windows XP 証明書チェーン照合ソフトウェアが適切な Windows Update ロケーションをチェックして、必要なルート証明書をダウンロードします。これはユーザ側ではシームレスに実行され、セキュリティ ダイアログ ボックスや警告などは一切表示されません。ダウンロードもバックグラウンドで自動的に行われます。
>
> 新しいルートは、Windows Update ダウンロード ファイルを通じて、Windows 2000、Windows NT、Windows 95、Windows 98、および Windows Millennium Edition (ME) クライアントで利用できるようになります。

この件に関して、Bruce Schneier 氏は Web サイトにて以下のようにコメントしている[77]。

New Microsoft Root Certificate Program
"New root certificates are no longer available with Microsoft Internet Explorer. Any new roots accepted by Microsoft are available to Windows XP clients through Windows Update. When a user visits a secure Web site (that is, by using HTTPS), reads a secure e-mail (that is, S/MIME), or downloads an ActiveX control that uses a new root certificate, the Windows XP certificate chain verification software checks the appropriate Windows Update location and downloads the necessary root certificate. To the user, the experience is seamless. The user does not see any security dialog boxes or warnings. The download happens automatically, behind the scenes."

This is the kind of thing that worries me. What exactly is this process. What happens when it fails. Why does everyone have to trust a root certificate, just because Microsoft does. And if the user doesn't see any security dialog boxes or warnings, the effects of any failure are likely to be catastrophic.

For a while now I have talked about the differences between vulnerability and risk. Something can be very vulnerable, but not risky because there is so little at stake. As Microsoft continues to centralize security, authentication, permissions, etc., risk rises dramatically because the effect of a single vulnerability is magnified.

- 実際の Windows Update

実際に Windows Update において「ルート証明書のアップデート」という項目が表示される。セキュリティ修正プログラムと同様にルート証明書が存在するため、一般のユーザは重要な情報が更新されていることを認識できない。

Windows update の詳細情報には以下のように記載されている。

ルート証明書のアップデート

このアップデートによって、コンピュータ上のルート証明書の一覧が、Microsoft Root Certificate Program の一部として弊社が認可した最新の一覧に更新されます。コンピュータに新たにルート証明書を追加することによって、セキュリティで保護された Web のブラウズ、電子メール、およびコード配信アプリケーションをより広範囲に、シームレスに利用できます。このアップデートには、Verisign 社、Thawte 社、およびアイルランドの Post.Trust 社からのルート証明書が含まれています。

- 問題点

Microsoft 社が Windows Update のセキュリティに関する詳細情報を正確に公開していない点が問題である。Windows XP 証明書チェーン照合ソフトウェアが適切な Windows Update ロケーションをチェックして、必要なルート証明書をダウンロードする処理の詳

細が公開されておらず、第三者によって検証できない。また、[78]の URL のファイルを実行すると Windows Update と同様の効果があり、ルート証明書が更新される。つまり、[78]に示すファイルの実行が Windows Update によるルート証明書更新そのものである。さらに、Microsoft 社は Windows Update の一部を Akamai Technologies, Inc.[80]に委託している。大量のユーザにコンテンツを配信するとき、Akamai Technologies, Inc.などのコンテンツ配信サービスを利用することは有効であるが、安全性を考慮する必要がある。

[対策]

Microsoft 社を信頼せず、自らルート証明書を更新する場合は以下の操作をすればよい[79]。

(1) コントロール パネルの [プログラムの追加と削除]を開きます。
(2) [Windows コンポーネントの追加と削除] をクリックします。
(3) [ルート証明書の更新] チェック ボックスをオフにします。
(4) [次へ]、[完了] の順にクリックします。

2.4.4. 表示の不具合が実質上のセキュリティホールになる問題

[概要]

ユーザが SSL/TLS による通信を行うとき、サーバが正当であることや通信路が暗号化されていることを確認する方法は、ユーザーインターフェースの表示を見ることが一般的である。SSL クライアントの表示に不具合が存在する場合、実質上の攻撃が成立する場合がある。Microsoft 社の Web ブラウザである Internet Explorer には報告時点(2004 年 1 月 26 日)において、不具合が存在した。これにより、実質上サーバになりすます攻撃が可能となる。

[詳細]

- Basic 認証における不具合

Microsoft 社の Web ブラウザである、Internet Explorer を含む多くのブラウザにはアドレスバー表示の不具合が確認されている。通常 Basic 認証を行う場合、以下のような URL にアクセスすることにより「domain」にユーザ名「user」にて Basic 認証によるアクセスができる。

```
http://user@domain
```

しかしながら、[81]によると、以下のような URL にアクセスすると、「domain1」にアクセスしながら、アドレスバーには「domain2」が表示される場合がある。

```
http://domain2%01@domain1
```

さらに、[82]によると、以下のような URL にアクセスすると、アドレスバー、ステータスバーを含めて「domain2」が表示される。

```
http://domain2%01<nul>@domain1
```

※<nul>は^A などの nul 文字

さらに、以下のような javascript を用いることにより、アドレスバーに「domain2」を表示させることができる。

```
<a href="domain1"
onClick="location.href=unescape('https://domain1%01@domain2');
return false;">domain1 </a>
```

- 問題点

通常ユーザが SSL/TLS により Web サイトを認証するとき、ステーテスバー、アドレスバー、プロパティなどのユーザーインターフェースを見ることによりサイトが正しいことを確かめる。したがって、ステーテスバー、アドレスバー、プロパティを偽装することができるとき、実質上のなりすまし攻撃につながる可能性がある。

- 攻撃の実行可能性

上記 exploit を Internetexplorer 6.0.2800.1106 に対して実行したところ、「アドレスバー」「ステータスバー」「プロパティ」には「https://example.com」と表示された。

[対策]

マイクロソフト社から対応する修正プログラムが公開されている。また、サーバ認証の際に「アドレスバー」「ステータスバー」「プロパティ」の他に「証明書」を確認する方法が有効である。ただし、証明書表示インターフェースに不具合がないことが前提である。

2.5. 運用上の注意点

本節では、SSL 機能を持つサーバ及びクライアントを使用する上での各種設定方法など運用方法と、それに伴うセキュリティについて考察を行う。サーバプログラムとしては、Web サーバである Apache に、SSL 機能を追加する代表的なモジュールである mod_ssl を対象とし調査を行った。ここでmod_ssl は OpenSSL を実装形式にしたものである。また、クライアントプログラムとしてブラウザを用い、代表的なブラウザである Internet Explorer、Netscape Navigator を調査対象とした。バージョンは現在最も使用されている Internet Explorer5.5、Netscape Navigator 4.7.x を中心に調査を行った。

2.5.1. 秘密鍵や証明書などの重要なファイルの管理

SSL を使用時には、秘密鍵や証明書といったファイルを管理する場所を指定する。例えば mod_ssl ではサーバの証明書や秘密鍵を管理するファイルを指定する命令（SSLCertificateFile *filename*，SSLCertificateKeyFile *filename*）、Server Certificate Chain を管理するファイルを指定する命令（SSLCertificateChainFile *filename*）、クライアントに関する CA の証明書を管理するディレクトリやファイルを指定する命令（SSLCACertificatePath *directory*，SSLCACertificateFile *filename*）、CRL を管理するディレクトリやファイルを指定する命令（SSLCARevocationPath *directory*，SSLCARevocationFile *filename*）が該当する。
これらの命令にデフォルト値はなく、ディレクトリやファイル名を指定する。ファイルは PEM エンコーディングされているので、盗聴は難しいと思われる。
サーバの秘密鍵を管理と、その他、証明書関連ファイルの管理とはセキュリティレベルが異なる。サーバの秘密鍵は、暗号化され第３者に露呈しない、値を改竄されない、ファイルを容易にアクセスされ消去できないといった安全性が保障されている必要があるとともに、運用面でも、秘密鍵の作成や更新は、セキュリティ管理者に限定したオペレーションとなるようパスワードやハードウェアトークンによって保護されているべきである。一方、上記の証明書関連ファイルは、秘密鍵ほど厳重な管理をする必要が必ずしも要求されない。証明書や CRL は一般的に公開された情報であるため、暗号化の必要がない。また、証明書や CRL 自身には CA の署名が付与されているためそれ自身、改竄される恐れも無い。しかしながら、不正に消去され、本来失効している証明書が有効であると誤った判断がなされないよう、不正消去に対する対策を講じる必要がある。

2.5.2. RandomSEED の取得

乱数生成時に疑似乱数生成器を用いるとき、その SEED となるソースを与える必要がある。例えば mod_ssl ではソースとなるファイルや、そのソースを生成するプログラムなどを指定する命令（SSLRandomSeed *context source* [*bytes*]）が該当する。
SEED の生成法にデフォルト値はなく、命令によって指定する。*context* にはいつその SEED を求めるかを指定するもので、OpenSSL 起動時（startup）か SSL 接続直前（connect）を指定する。
source で SEED の生成方法を指示する。これには以下の指示がある。
builtin は最小 CPU サイクルを用いるもので、十分な擬似乱数性を有していると考えられるため実用上問題とならない。
file:/path/to/source はデバイスなどのファイルを用いるもので、通常/dev/random や /dev/urandom などが使われる。/dev/urandom はすぐに任意サイズの値を返すが、乱数生成用のエントロピー保持量が足りない場合は、現在保持している値のハッシュ値で返すことになる。/dev/random の方は必ず乱数値で返すが、エントロピー保持量が足りないときは蓄積されるまでブロックされる。安全性の高さを求めるならエントロピーの高い /dev/random が望ましいが、サーバの接続数や通信速度が制限されてしまう問題点がある。
exec:/path/to/program は SEED 生成用用プログラムを指定するもので、AT&T の truerand などを使うときに指示する。サーバ独自の暗号方式を用いる場合、そのプログラムが安全であれば最も安全な方式といえるが、一般的なユーザが独自にプログラムを作るべきではない。現在のところ builtin または file:/dev/(u)random による乱数における攻撃法が発見されていないのでこれらの方式の利用が推奨される。

2.5.3. セッション鍵のライフタイム

mod_ssl では、頻繁に生じる同一トランザクションを高速に処理するため、セッション情報をメモリ上に展開し、保持するセッションキャッシュ機構が実装されている。ここで、セッションキャッシュとは、一人のユーザの時間的に連続する接続や、複数の TCP コネクションによる接続を一つの SSL セッションとみなし、セッション ID を付与することにより、同じセッション鍵（master secret）を使う技術であり、セキュリティの観点らかは SSL における実効上のセッション鍵のライフタイムに相当する。（通常のブラウザは 4 つの TCP コネクションにより接続する。）セッションキャッシュが用いられる場合ハンドシェイクプロトコルは省略される以下にその図を示す。

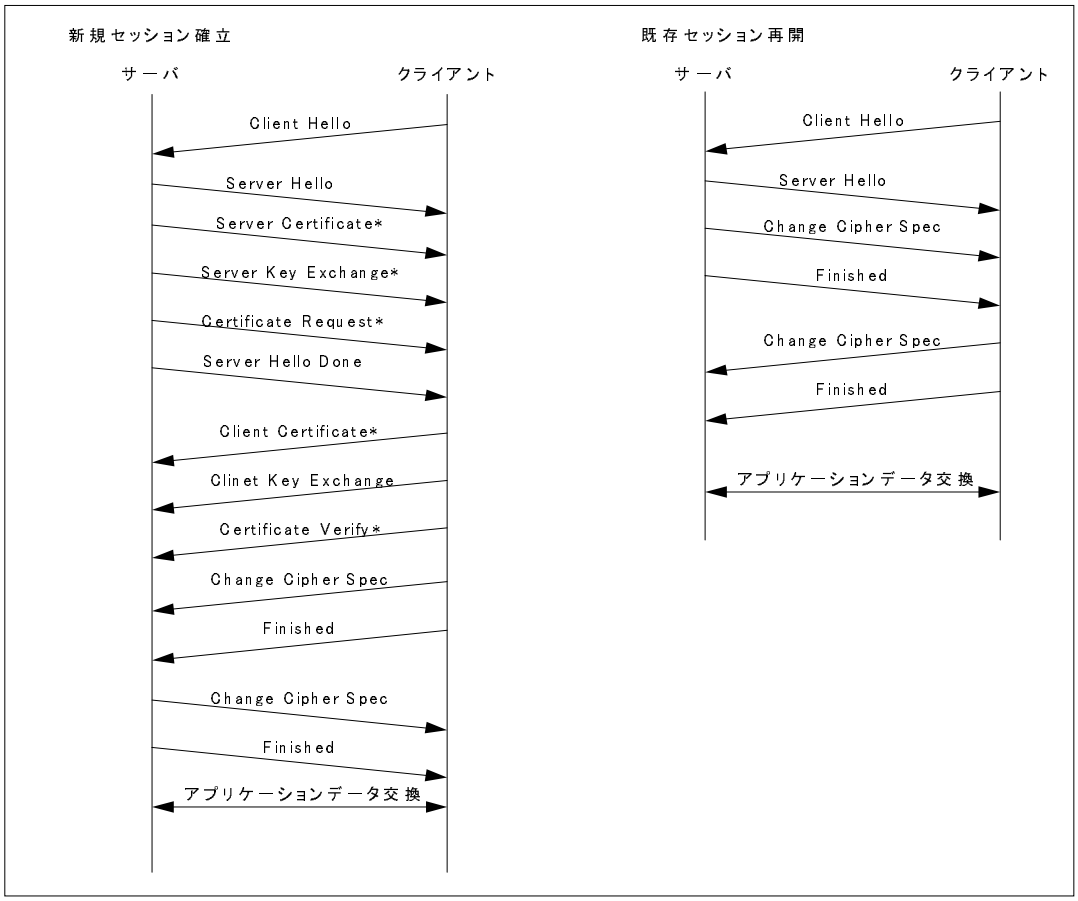


図 2.5.3-1 セッションキャッシュが用いられる場合の handshakeprotocol

このセッションキャッシュは安全性を考慮して一定時間毎に更新される。mod_ssl ではセッションをキャッシュしておく有効期限を決める命令があり（SSLSessionCacheTimeout *sec.*）、指定秒で新たなセッションに更新されることになる。この値が小さい方が頻繁に鍵交換されるので安全になるが、小さすぎるとサーバに負荷がかかり、通信が遅くなる。この命令ではデフォルト値が 300 秒になっている。共通鍵鍵暗号の鍵を 300 秒で解読もしくは盗みだすことは極めて困難であるため、安全性の面では妥当な値が設定されているといえる。一般的な場合、セキュリティと性能を考慮してもう少し大きな値に設定してもよい。ただし、最大値は 24 時間（60×60×24 秒）である。

参考のため、以下にサーバとして mod_ssl、クライアントとして Netscape4.7 を用いた場合の、サーバ側のトレースを示す。ここでは説明のためセッション鍵のライフタイムは 20 秒に設定した。20 秒ごとにセッション鍵（セッション ID）が更新されていることが分かる。

2.5.4. 使用するプロトコルバージョン

現在、多くのサーバプログラム及びブラウザプログラムにおいて、SSL Vetsion2.0, SSL Version 3.0, TLS Version1.0 のどれもがサポートされており、どのプロトコルを使用するかを利用者が選択可能となっている。サーバ用プログラムである mod_ssl においては、SSLProtocol という設定項目で設定を行う。デフォルト値は、SSLProtocol all となっており、すべてのプロトコルをサポートする状態となっている。従って、アクセスしてくるクライアントがサポートしているプロトコルの内、最新のものが使用されることになる。（TLS>SSL3.0>SSL2.0）

一方クライアント側においても、同様の設定項目が存在する。Internet Explorer5.5 においては、PCT、SSL2.0、SSL3.0、TLS1.0 が選択可能となっており、デフォルトの状態では、SSL2.0 及び、SSL3.0 が使用可能となっている。

また、Netscape Navigator 4.7.x においては、SSLv2.0、SSLv3.0 が選択可能となっており、デフォルトの状態では、SSLv2.0 及び SSLv3.0 が使用可能となっている。

いくつかのセキュリティホールが指摘されている SSL 2.0 をサーバ、クライアントソフト共に、デフォルトで使用可能となっているが、ソフトウェアを実際に運用する際には、使用不可に設定するのが望ましい。特に、現在インターネット上の SSL を使用するほとんどのサイトは、SSL3.0 に対応していることから、クライアント側のブラウザは、SSL2.0 を使用不可に設定して使用すべきであると考えられる。

2.5.5. 使用する暗号アルゴリズム

サーバ側においては､多くの実装において使用する暗号アルゴリズムについて､鍵交換、認証、暗号化、ハッシュ関数を個々に使用可能、使用不可能を設定可能である。mod-ssl においては、SSLCipherSuite によって設定を行うが、鍵交換は、RSA、DHwithRSA、DHwithDSS、Ephemeral DH、認証は、NULL（認証なし）、RSA、DSS、DH、暗号化は、NULL、DES、3DES、RC4、RC2、IDEA、ハッシュ関数は、MD5、SHA1、SHA からそれぞれ選択し CipherSuite を作成する。また Aliases として､ADH（Anonymous DH、鍵交換はＤＨを選択し、認証は NULL にする設定と同様）や、LOW（128bit 未満の鍵長を持つ暗号を使用）、MEDIUM（128bit の鍵長を持つ暗号を使用）、HIGH（鍵長 168bit の 3DES を使用）などを使用した設定も可能である。デフォルト値は、ADH を除いた認証、鍵交換方式と、NULL を除くすべての暗号方式、すべてのハッシュ関数が使用可能となっている。

一方クライアント側であるブラウザであるが、Internet Explorer5.5 においては、Fortezza の使用の有無が選択できるのみである。(デフォルトは使用可能)

また、Netscape Navigator 4.7.x においては、SSLv2.0、SSLv3.0 でそれぞれ異なる設定が可能であり、SSLv2.0 においてはすべての暗号方式が、SSLv3.0 においては、暗号化なし以外のすべての暗号方式が使用可能となっている。

サーバの利用用途にもよるが、認証なし、暗号化なし、などを選択することは、推奨されない。暗号についても、可能であれば 128bit 以上の鍵長を持つ暗号を使用することが望ましいと考えられる。また、traffic analysis attack を防ぐためには、少なくともストリーム暗号（RC4）を用いないことが望ましい。

2.5.6. 証明書検証

mod_ssl の証明書検証に関する設定は、クライアント認証を行うか否かを指定する命令（SSLVerifyClient *level* ）と中間証明機関（intermediate certificate issuer）の最大数をを指定する命令（SSLVerifyDepth *number*）により行われる。SSLVerifyClient は因数として

none：クライアント証明書を要求しない。
optional：クライアントが証明書を提示してもよい。提示された場合にその検証を行う。
require：クライアント証明書を要求する。
optional_no_ca：クライアントは証明書を提示してもよい。提示された場合その検証を行わない。

を選択できる。デフォルトでは none に設定されており、必要に応じて require を設定すればよい。optional および optional_no_ca は試験を行うときなどに用いる。SSLVerifyClient は因数として level（整数）を選択できる。この level は中間証明機関を許可する最大数である。デフォルトでは 1 と設定されている。つまり、0（：自己証明書）もしくは 1（：クライアントの公開鍵の証明書そのもの）がサーバに事前登録されている必要がある。自己証明書については、サーバに登録時に通常 CA が行うようなクライアントに対する身元確認を行った上で、サーバの自己責任において、そのクライアント証明書を登録するという運用の前提が必要である。

また、値はルート CA として登録する CA の証明書に依存して決定すればよい。また、mod_ssl は CRL を自動的に更新する OCSP のような機能がない。CRL は 2.4.1 で説明したディレクトリに保存しなければならない。よって、定期的に CRL を更新する必要がある。ただし、CRL 更新プログラムをユーザが作成する場合、不具合により CRL を改ざん

されないよう管理する必要がある。

Ineternet Explorer の証明書検証に関する設定は個人、ほかの人、中間証明機関、信頼できるルート証明機関を設定ができ、サーバ証明書はあらかじめ Microsft 社によってベリサインなどの有名な CA のルート証明書が複数インストールされている。クライアント認証を受けるために必要なクライアント証明書は個人のタブからインポートする必要がある。この際にセキュリティレベルを設定する必要がある。ここで、ルート証明書はあらかじめ登録されているが、知識のあるユーザはどのルート証明書が登録されているのか確認しておくべきである。ユーザのセキュリティポリシー（どの証明書を信用するか）をブラウザベンダーが必ずしも反映していないからである。また、CRL の管理は行われているが、OCSP のようなリアルタイムな確認は行われていない。ユーザは頻繁に CRL をダウンロードしてインストールする必要があり非常に面倒であり、実際に活用されていないことが問題である。よって、OCSP や CRL の自動的なインストールする機能を実装すべきである。

Netscape Navigator の証明書検証機能に関する設定も Ineternet Explore r と同様の設定ができる。ただし、ルート証明書の種類は異なる、ここでも、どのルート証明書が登録されているか確認するべきである。個人の証明書を登録する場合パスワードを設定した方が望ましいが、そのパスワードの管理に気をつける必要がある。また、CRL の管理はここでも行われているが OCSP は実装されていない。よって、Internet Explorer と同様のことがいえる。また、調査対象外ではあるが Netscape6.x には CRL や OCSP が実装されているが仕様しづらく、さらに改善する必要がある。

2.5.7. ログ

SSL でのログに関する設定を行う。例えば mod_ssl では、ログファイルの格納場所を指定する命令（SSLLog *filename*）や、ログレベルを指定する命令（SSLLogLevel *level*）が該当する。

これらの命令では、格納場所のデフォルト値はなく、レベルのデフォルト値は none（規定無し）となっている。単にログの出力に関する設定であり、ログファイルを第三者に閲覧されたくない場合は、アクセスされない場所に管理するなどすればよく、特にセキュリティの脅威になることはない。

2.5.8. 警告通知

クライアント側で表示される警告の有無について、利用者は設定可能である。Internet Explorer5.5 においては、リダイレクト、保護付き/保護なしサイト間移動、無効な証明書

についての警告の有無を選択できる。デフォルト値はすべて“あり”である。

また、Netscape Navigator 4.7.3 においては、暗号化サイトに入るとき、出るとき、暗号化/非暗号化の混ざったページを表示するとき、非暗号化情報を送信するとき、について警告通知を行うように設定可能である。デフォルトは、すべて“あり”となっている。

設定可能な警告はいずれも重要な警告であり、すべて通知するように設定すべきである。特に、保護（暗号化）されたサイトと、保護（暗号化）されないサイト間を移動する際に、警告が出ることは、利用者が保護されていないサイトを保護されていると誤解しないためにも重要な警告メッセージであるといえる。

2.5.9. その他設定

SSL ではその他いろいろな設定が行えるようになっている。例えば mod_ssl ではオプションな設定を行う命令（SSLOptions [+-]*option*）があり、引数に様々なオプションを指定できる。これには CGI/SSI 環境変数に関係する設定を行うもの（StdEnvVars, CompatEnvVars, ExportCertData）、クライアントの X509 証明書を HTTP Basic Authorization のユーザネームに変更するもの（FakeBasicAuth）、不許可すべきアクセスは強制的にアクセス不許可するもの（StrictRequire）、再ネゴシエーションの最適化に関するもの（OptRenegotiate）がある。
これらはデフォルトではどのオプションも指定されておらず、オプション使用時またはオプション解除時に指定することになる。

2.5.10. パッチマネージメントについて

パッチと呼ばれる修正プログラムを迅速かつ正確に、現行ソフトウェアに対して適用することは、ソフトウェアの運用においてもっとも重要な作業のひとつである。このような、パッチのマネージメントは、組織内に専門の部署を設けるほうが望ましい。米国の NIST(National Institute of Standards and Technology)は、セキュリティ上の問題を修正するパッチの運用指針を定めた“Procedures for Handling Security Patches”という文書を発行している。この文書においては、組織内の Patch and Vulnerability Group (PVG) を設け、PVG が主体となってパッチの運用をする方針を打ち出している。(図 2.5.10.1)

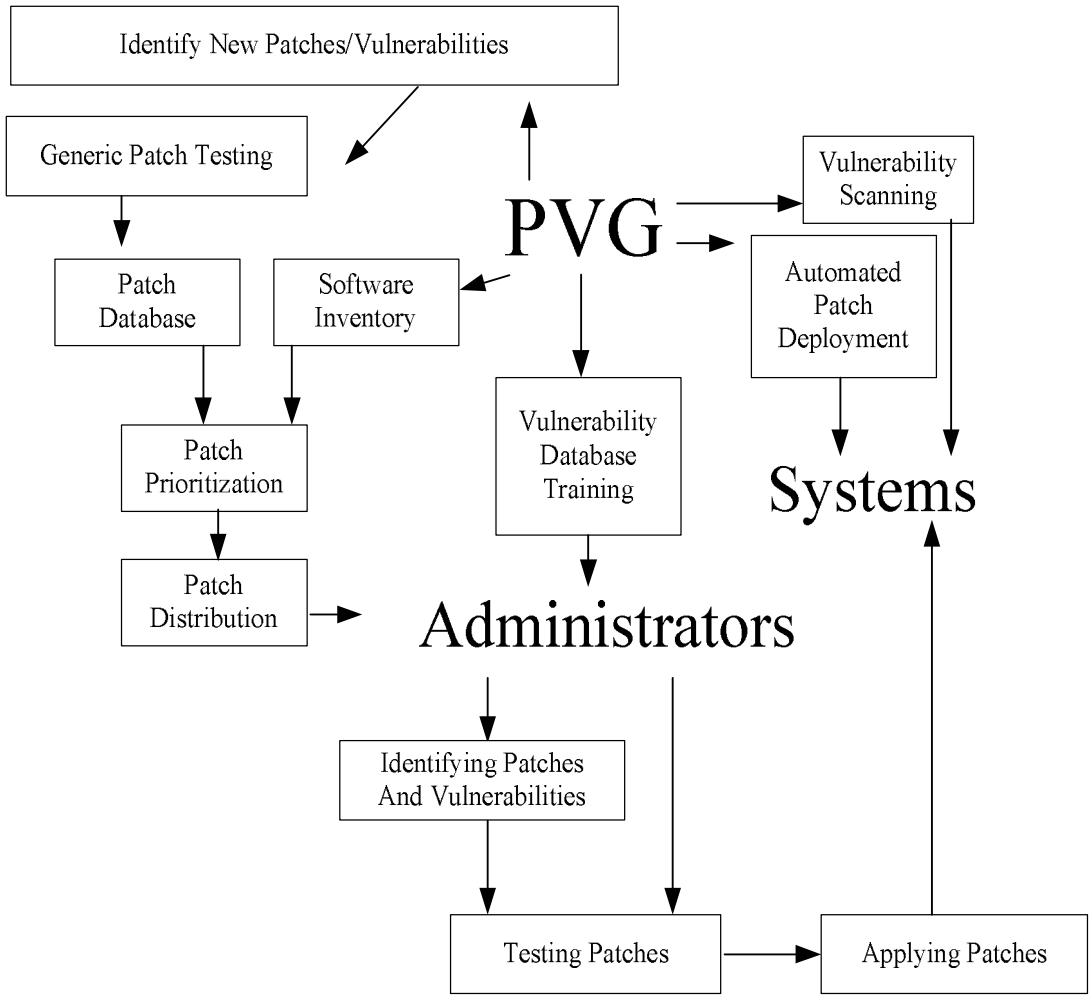


図 2.5.10.1 パッチマネージメントの流れ

PVG は、以下のような業務を行う。

- 組織が使用しているソフトウェア、ハードウェアに関するデータベースを管理する
- 脆弱性やパッチの情報を収集し、パッチの存在を確認する。
- パッチの適用順序に優先順位をつける
- 組織固有のパッチ・データベースを作成する。
- パッチのテストを行う。
- システム管理者に脆弱性情報とパッチを提供する。
- スキャニングによって、パッチの適用状態を検証する。
- システム管理者を教育する。

文書においては、脅威の深刻さの度合いや、適用するシステムの重要性、パフォーマンスや機能の低下の有無、費用等の観点から総合的に判断して、パッチの適用の有無、作業の優先順位を決める必要があるとしている。パッチや脆弱性情報の入手元としては、ベンダーのサイト及びメーリングリスト、第三者の Web サイト、メーリングリスト及びニュースグループ、CVE[48]、CERT/CC[51]、NIST ICAT Vulnerability Index[49]、NIPC(National Infrastructure Protection Center)[50]、FedCIRC (Federal Computer Incident Response Center)[52]が挙げられている。対応する日本のサイトとしては、JPCERT/CC[53]や@POLICE[54]、日本ネットワークセキュリティ協会(JNSA)[55]、情報処理推進機構（IPA）セキュリティセンター[56]などがある。また、SSL の関する情報の収集においては、OpenSSL のサイトや、各種 OS の提供サイトも参照することが望ましい。

参考文献

[1] A. Freier, P. Karlton, and P. Kocher, “The SSL Protocol Version 3.0”, http://home.netscape.com/eng/ssl3/draft302.txt

[2] T. Dierks, A. Freier, ”The TLS Protocol Version 1.0”, http://www.ietf.org/rfc/rfc2246.txt

[3] Josh Benaloh, Butler Lampson, Daniel Simon, Terence Spies, and Bennet Yee, ” The Private Communication Technology Protocol”, http://activex.adsp.or.jp/english/specs/pct.htm

[4] Kipp E.B. Hickman, ”SSL 2.0 PROTOCOL SPECIFICATION”, http://www.netscape.com/eng/security/SSL_2.html

[5] Simon Blake-Wilson, Magnus Nystrom, David Hopwood, Jan Mikkelsen, and Tim Wright, “TLS Extensions”, http://www.ietf.org/internet-drafts/draft-ietf-tls-extensions-02.txt

[6] Simon Blake-Wilson, and Magnus Nystrom, “Wireless Extensions to TLS”, http://www.ietf.org/proceedings/00dec/I-D/draft-ietf-tls-wireless-00.txt

[7] Stephen Farrell, “TLS extensions for AttributeCertificate based authorization”, http://www.ietf.org/proceedings/99jul/I-D/draft-ietf-tls-attr-cert-01.txt

[8] K. Jackson, S. Tuecke, and D. Engert, “TLS Delegation Protocol”, http://www.ietf.org/internet-drafts/draft-ietf-tls-delegation-01.txt

[9] A. Medvinsky, and M. Hur, “Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)”, http://www.ietf.org/rfc/rfc2712.txt

[10] Matthew Hur, Joseph Salowey, Cisco Systems, and Ari Medvinsky, “Kerberos Cipher Suites in Transport Layer Security (TLS)”, http://www.ietf.org/internet-drafts/draft-ietf-tls-kerb-01.txt

[11] W. Price, and M. Elkins, “Extensions to TLS for OpenPGP keys”, http://www.ietf.org/internet-drafts/draft-ietf-tls-openpgp-01.txt

[12] D. Taylor, “Using SRP for TLS Authentication”, http://www.ietf.org/internet-drafts/draft-ietf-tls-srp-01.txt

[13] [13]John Banes, and Richard Harrington, “56-bit Export Cipher Suites For TLS”, http://www.ietf.org/internet-drafts/draft-ietf-tls-56-bit-ciphersuites-01.txt

[14] Joo-won Jung, and ChangHee Lee, “TLS Extension for SEED and HAS-160”, http://www.ietf.org/proceedings/00jul/I-D/tls-seedhas-00.txt

[15] H. Ohta, and H. Tsuji, “Addition of MISTY1 to TLS”, http://www.ietf.org/internet-drafts/draft-ietf-tls-misty1-01.txt

[16] S. Moriai, "Addition of the Camellia Encryption Algorithm to TLS", http://www.ietf.org/internet-drafts/draft-ietf-tls-camellia-01.txt

[17] Pete Chown, "AES Ciphersuites for TLS", http://www.ietf.org/internet-drafts/draft-ietf-tls-ciphersuite-06.txt

[18] Simon Blake-Wilson, Tim Dierks, and Chris Hawk, "ECC Cipher Suites for TLS", http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-01.txt

[19] Ari Singer, "NTRU Cipher Suites for TLS", http://www.ietf.org/internet-drafts/draft-ietf-tls-ntru-00.txt

[20] Joseph Hui, "TLS Pathsec Protocol", http://www.ietf.org/internet-drafts/draft-ietf-tls-pathsec-00.txt

[21] J. Manger "A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS#1 v2.0 CRYPTO'01 LNCS 2139 pp.230-238, Aug 2001

[22] Bleichenbacher Discovery Q&A, http://www.rsasecurity.com/rsalabs/pkcs1/qa.html

[23] On OAEP, PSS, and S/MIME, http://www.ietf.org/proceedings/00dec/slides/smime-5/sld005.htm

[24] D. Bleichenbacher, "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1", *Advances in Cryptology-CRYPTO '98*, LNCS vol. 1462. Springer-Verlag, 1998.

[25] David Wagner and Bruce Schneier, "Analysis of the SSL 3.0 protocol", *The Second USENIX Workshop on Electronic Commerce Proceedings*, USENIX Press, November 1996, pp. 29-40. Revised April 15, 1997.

[26] Steven M. Bellovin, "Problem Areas for the IP Security Protocols", *Proceedings of the Sixth USENIX Security Symposium*, Usenix Association, 1996, pp. 205-214.

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

[28] Ralf S. Engelschall, "mod_ssl", http://www.modssl.org/

[29] Microsoft Corporation, "Internet Explorer 5.5", http://www.microsoft.com/

[30] Netscape, "Netscape Navigator 4.7.3", http://www.netscape.com/

[31] The Apache Software Foundation, "Apache", http://www.apache.org/

[32] Ben Laurie, and Adam Laurie, "Apache-SSL", http://www.apache-ssl.org/

[33] SecurityFocus, "BUGTRAQ", http://www.securityfocus.com/

[34] V. Klima, O. Pokorny and T. Rosa, "Attacking RSA-based Sessions in SSL/TLS", IACR Cryptology ePrint Archive: Report 2003/052.

[35] V. Klima and T. Rosa, "Further Results and Considerations on Side Channel

Attacks on RSA", in Proc. of CHES'02, 2002.

[36] J,Jonsson, B.Kaliski, "On the Security of RSA Encryption in TLS", in Proc. of CRYPTO'02, pp.127-142, 2002.

[37] D.Brumley, and D.Boneh, "Romote Timing Attacks are Practical", in Proc. of the 12th Usenix UNIX Security Symposium, USENIX, 2003.

[38] P.Junod, "On the Optimality of Linear, Differential, and Sequential Distinguishers", in Proc. of EUROCRYPT03, pp.17-32, 2003.

[39] P.Kocher, "Timing Attacks on Implementations of Diffie -Hellman, RSA, DSS, and other Systems", in Proc. of CRYPTO'96, pp.104-113, 1996.

[40] B.Moller, "Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures", http://www.openssl.org/bodo/tls-cbc.txt, 2002.

[41] R.Baldwin, R.Rivest, "The RC5, RC5-CBC, RC5-CBC-Pad, and RC5-CTS Algorithms", RFC 2040, 1996.

[42] S.Vaudenay, "Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS...", in Proc. of EUROCRYPT'02, LNCS, No.2332, pp.534-545, 2002.

[43] B.Canvel, A.Hiltgen, S.Vaudenay, and M.Vuagnoux, "Password Interception in a SSL/TLS Channel", CRYPTO 2003, LNCS, No.2729, pp.583-599, 2003.

[44] J.Black, and H.Urtubia, "Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption", in Proc. of 11th USENIX Security Symposium 2002, pp. 327338, 2002.

[45] K.G.Paterson, and A.Yau, "Padding Oracle Attacks on the ISO CBC Mode Encryption Standard", In, Proc., CT-RSA04, 2004.

[46] M.Bellare, A.Desai, E.Jokipii, and P.Rogaway, "A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation", in Proc. of the 38th Symposium on Foundations of Computer Science, IEEE, 1997.

[47] D.Coppersmith, "Small solutions to polynomial equations and low exponent RSA vulnerabilities, Journal of Cryptology, 10:233-266, 1997.

[48] Common Vulnerability and Exposure (CVE), http://cve.mitre.org/.

[49] NIST ICAT Vulnerability Index, http://icat.nist.gov/.

[50] National Infrastructure Protection Center, http://www.nipc.gov/.

[51] CERT/CC, http://www.cert.org/.

[52] Federal Computer Incident Response Center (FedCIRC), http://fedcirc.org/.

[53] JPCERT/CC, http://www.jpcert.or.jp/.

[54] @POLICE, http://www.cyberpolice.go.jp/.

[55] 日本ネットワークセキュリティ協会(JNSA), http://www.jnsa.org/.

[56] 情報処理推進機構セキュリティセンター, http://www.ipa.go.jp/security/.

[57] PROTOS Test-Suite: c06-snmpv1, http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1/

[58] ASN.1 encoding rules Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER) ITU-T Rec. X.690 (2002) | ISO/IEC 8825-1:2002

[59] OpenSSL Security Adversary, http://www.openssl.org/news/secadv_20020730.txt, 30 July 2002.

[60] NISCC Vulnerability Advisory 006489/OpenSSL, http://www.uniras.gov.uk/vuls/2003/006489/openssl.htm, 30 September 2003.

[61] OpenSSL Security Adversary, http://www.openssl.org/news/secadv_20030930.txt, 30 September 2003.

[62] NISCC Vulnerability Advisory 006489/OpenSSL2, http://www.uniras.gov.uk/vuls/2003/006489/openssl2.htm, 4 November 2003.

[63] OpenSSL Security Adversary, http://www.openssl.org/news/secadv_20031104.txt, 04 November 2003.

[64] OpenSSL ASN.1 parsing bugs PoC / brute forcer, http://www.securityfocus.com/archive/1/349837, Jan 15 2004.

[65] PROTOS Test-Suite: c07-sip, http://www.ee.oulu.fi/research/ouspg/protos/testing/c07/sip/

[66] Securityfocus Multiple Vendor S/MIME ASN.1 Parsing Denial of Service Vulnerabilities, http://www.securityfocus.com/bid/8981

[67] Microsoft Security Bulletin MS04-007, ASN.1 Vulnerability Could Allow Code Execution (828028),

[68] http://www.microsoft.com/technet/security/bulletin/MS04-007.mspx

[69] Bugtraq http://www.securityfocus.com/bid

[70] openssl-too-open, http://www.phreedom.org/solar/exploits/apache-openssl/

[71] openssl sslv3 session_id overrun exploit, http://hsj.shadowpenguin.org/misc/sslv3_exp.txt

[72] slapper worm http://www.symantec.com/region/jp/sarcj/data/l/linux.slapper.worm.html

[73] Microsoft Windows Update, http://windowsupdate.microsoft.com/

[74] Microsoft Windows Update, http://v4.windowsupdate.microsoft.com/ja/about.asp

[75] Microsoft ルート証明書プログラム, http://www.microsoft.com/japan/technet/security/news/rootcert.asp

[76] Microsoft Root Certificate Program,

http://www.microsoft.com/technet/security/news/rootcert.asp
[77] New Microsoft Root Certificate Program,
http://www.schneier.com/crypto-gram-0109.html#8
[78] http://www.download.windowsupdate.com/msdownload/update/v3-19990518/CabPool/rootsupd_882A0A0D36FE385B042EDEC58E1F0E7715BDA1BB.exe
[79] Microsoft TechNet 信頼されたルート証明機関証明書の自動更新を無効にするには,
http://www.microsoft.com/japan/technet/prodtechnol/windowsserver2003/proddocs/standard/sag_CMprocsautorootoff.asp
[80] Akamai Technologies, Inc. http://www.akamai.com
[81] Internet Explorer URL parsing vulnerability
http://www.securityfocus.com/archive/1/346948
[82] IE ホゥル：NUL 文字攻撃で URL 表示捏造可能,
http://altba.com/bakera/hatomaru.aspx/ebi/topic/1003
[83] Internet Explorer 用の累積的なセキュリティ修正プログラム (832894) (MS04-004)
http://www.microsoft.com/japan/technet/security/bulletin/ms04-004.asp
[84] A security update is available that modifies the default behavior of Internet Explorer for handling user information in HTTP and in HTTPS URLs,
http://support.microsoft.com/default.aspx?scid=kb;EN-US;834489

[85] J. Håstad and M. Näslund, “The Security of Individual RSA Bits”, in Proc. of FOCS ’98, pp. 510-521, 1998.