

IT 資格道場

LPIC-3 303試験 学習用テキスト

予備知識ゼロから合格ラインへ——読んで理解する1冊目

無料サンプル(第1章まで)

LPI公式 LPIC-3 Exam 303 Objectives Version 3.0(303-300・2026-10-13 取得)準拠
2026年10月版

ご利用にあたって

- このテキストは「IT資格道場」(www.shikaku-doujou.com)の購入者限定テキストのうち、学習用テキストの第1章までを収録した**無料サンプル**です。どなたでもご覧いただけます。
- 無料サンプルも著作物です。転載・改変しての再配布・二次販売はできません。
- AI モデルの学習データとしてのご利用(学習目的の入力・データセット化を含む)はできません。
- 製品版(学習用テキスト・頻出度別ノート・暗記用ノートの3点)は、ご購入者を識別できる透かし入りのPDFとして提供されます。
- 本PDFはスマートフォン・タブレットでの閲覧に合わせたA5版面・文字サイズで組版しています。

目次

この教材群の使い方(3点セット)

第1章 Cryptography [331.1/331.2/331.3/331.4]

暗号の部品: 共通鍵・公開鍵・ハッシュ・署名 [331.1]

X.509 証明書の中身 [331.1]

形式: PEM・DER・PKCS [331.1]

鍵の生成と管理 [331.1]

CSR と自己署名証明書 [331.1]

認証局 (CA) の構築と運用 [331.1]

信頼の連鎖・PKI・証明書の透明性 [331.1]

失効: CRL と OCSP [331.1]

Let's Encrypt・ACME・certbot [331.1]

CFSSL [331.1]

SSL/TLS: 版・握手・暗号スイート [331.2]

Apache(mod_ssl)でHTTPSを出す [331.2]

SNI と名前ベースの仮想ホスト [331.2]

HSTS [331.2]

クライアント証明書による利用者認証 [331.2]

OCSP ステープリング [331.2]

openssl s_client と s_server で試す [331.2]

ブロックデバイスの暗号化とファイルシステムの暗号化 [331.3]

dm-crypt と LUKS1: cryptsetup [331.3]

/etc/crypttab と起動時の自動利用 [331.3]

plain dm-crypt [331.3]

LUKS2 の機能 [331.3]

eCryptfs: ホームディレクトリの暗号化 [331.3]

cryptmount [331.3]

Clevis と NBDE (Tang・TPM2) [331.3]

DNS・ゾーン・リソースレコード [331.4]

DNSSEC の仕組み [331.4]

BIND を権威サーバにして署名済みゾーンを出す [331.4]

鍵の管理: ロールオーバー・再署名 [331.4]

DNSSEC を検証するキャッシュサーバ [331.4]

CAA と DANE: 証明書の情報を DNS に載せる [331.4]

TSIG [331.4]

DNS over TLS・DNS over HTTPS・マルチキャスト DNS [331.4]

サンプルはここまでです

この教材群の使い方(3点セット)

種類	役割
① 学習用テキスト	これだけで問題が解けるようになる本編
② 頻出度別ノート	テキストを読んだら次はこれ。頻出順に絞った問題と解説で「解ける」に橋渡し。試験直前の最終チェックにも
③ 暗記用ノート	覚えるべき要点だけを一問一答に凝縮。空き時間の反復と用語の再確認用

使い方: ① 学習用を読む → ② 頻出度別で解き方を掴む → ③ 問題演習で腕試し → ④ 頻出度別に戻って再確認(試験直前もここ)。暗記用は空き時間や用語の再確認に。

このテキストの位置づけ: ①の学習用テキストです。まずはここを通読し、これだけで問題が解けるようになることを目標にしてください。

- 対象試験: LPIC-3 303試験 (303-300、出題範囲 Version 3.0.0)。60問・90分、多肢選択と穴埋め、合格は 200～800 のスケールで 500点
- 構成: 全5章。各章の見出しに細目の番号を書きました(暗号 331・ホスト 332・アクセス制御 333・ネットワーク 334・脅威と脆弱性評価 335)
- 本文の脚注は、巻末の「商標と出典」にまとめています

第1章 Cryptography [331.1/331.2/331.3/331.4]

この章は4つの細目からなります。証明書と PKI (331.1)、TLS と mod_ssl (331.2)、暗号化ファイルシステム (331.3)、DNS と暗号 (331.4) です。道具は OpenSSL・Apache HTTPD・cryptsetup・BIND で、コマンドと設定項目の綴りまで問われます。

暗号の部品:共通鍵・公開鍵・ハッシュ・署名 [331.1]

部品	鍵	できること	代表例
共通鍵暗号	暗号化と復号で同じ鍵	大量のデータを速く暗号化する	AES
公開鍵暗号	公開鍵と秘密鍵の組	鍵の受け渡しと署名	RSA、ECDSA
ハッシュ	鍵なし	任意長のデータから固定長の要約を作る。元には戻せない	SHA-256
デジタル署名	秘密鍵で署名、公開鍵で検証	改ざんの検知と作成者の確認	RSA-PSS、ECDSA

公開鍵暗号は遅いので、通信の本体には使いません。公開鍵暗号で共通鍵を取り決め、データの暗号化は共通鍵に任せます (TLS も暗号化ファイルシステムもこの組み合わせ)。署名は、データのハッシュ値を秘密鍵で変換して作ります。受け取った側は、公開鍵で署名を検証し、自分で計算したハッシュ値と合うかを確認めます (RSA では公開鍵で署名を戻して比べる)。鍵の向きは暗号化と逆です。暗号化は「公開鍵で包み、秘密鍵で開ける」。署名は「秘密鍵で作ри、公開鍵で確かめる」。

```
$ openssl dgst -sha256 -sign web.key -out m.sig m.txt
$ openssl dgst -sha256 -verify web.pub -signature m.sig m.txt
Verified OK
$ echo tamper >> m.txt          # 書き足すと検証は Verification failure
                                になる
```

1バイトでも書き換えるとハッシュ値が変わり、検証は通りません。openssl dgst は、-sha256 などではッシュを計算し、-sign と -verify で署名と検証も行います。

PFS (前方秘匿性) は、サーバの秘密鍵が後から漏れても、過去に記録した通信を解読できない性質です。通信ごとに使い捨ての鍵 (ECDHE など) で共通鍵を作り、使い終わったら捨てるので成り立ちます。TLS 1.2 までの RSA 鍵交換 (スイート名が TLS_RSA_ で始まるもの) は、サーバの秘密鍵で共通鍵を包んで送るため、PFS がありません。TLS 1.3 は RSA 鍵交換を廃止し、完全な握手の鍵交換を ECDHE か DHE にするので、PFS が成り立ちます (再開で PSK だけを使うモードには PFS がない)。

X.509 証明書の中身 [331.1]

X.509 証明書は、公開鍵と持ち主の名前を、CA (認証局) の署名で結びつけたデータです。形は RFC 5280 が決めています。[^c1-1] 署名の対象になる部分 (tbsCertificate)、署名のアルゴリズム、署名値の3つで成り立ちます。

フィールド	中身
version	v3 なら値は 2。拡張を持てるのは v3 だけ
serialNumber	発行した CA の中で一意な番号。失効はこの番号で指す

フィールド	中身
signature	CA が署名に使ったアルゴリズム (例 sha256WithRSAEncryption)
issuer	発行した CA の名前
validity	notBefore (開始) と notAfter (終了)
subject	持ち主の名前 (CN、O、C など)
subjectPublicKeyInfo	持ち主の公開鍵とアルゴリズム
extensions	v3 拡張。用途や別名、失効情報の取得先を載せる

自己署名証明書は、issuer と subject が同じで、自分の鍵で自分に署名した証明書です (ルート CA の証明書がこの形)。v3 拡張には `critical` の印を付けられます。critical の拡張を理解できないクライアントは、その証明書を拒否します。印のない拡張は、読み飛ばしてかまいません。[^c1-2]

拡張	役割と値の例
basicConstraints	<code>CA:TRUE</code> か <code>CA:FALSE</code> 。 <code>pathlen:n</code> は、この CA の下に続けられる CA の段数
keyUsage	鍵の使い道。digitalSignature、keyEncipherment、keyCertSign (証明書への署名)、cRLSign (CRL への署名)
extendedKeyUsage	用途の限定。serverAuth、clientAuth、codeSigning、emailProtection、OCSPSigning
subjectAltName	別名の一覧。 <code>DNS:</code> 、 <code>IP:</code> 、 <code>email:</code> 。1枚で複数のホスト名を守れる

拡張	役割と値の例
subjectKeyIdentifier / authorityKeyIdentifier	自分の鍵の識別子と、署名した CA の鍵の識別子。連鎖をたどる手がかり
crlDistributionPoints	CRL の取得先 URI
authorityInfoAccess	OCSP;URI: で OCSP の問い合わせ先、 caIssuers で発行者の証明書の取得先

最近のブラウザは、ホスト名を subjectAltName で照合します。CN は見ません。SAN にないホスト名は、CN が一致しても警告になります。サーバ証明書には `serverAuth`、クライアント証明書には `clientAuth` を入れます。

証明書のライフサイクルは、次の順に進みます。

- 1. 鍵ペアを作る (秘密鍵は持ち主の手元から出さない)
- 2. CSR を作り、CA に送る
- 3. CA が申請者を確認し、署名して証明書を発行する
- 4. サーバやクライアントに証明書を配備する
- 5. notAfter の前に更新する (再発行)
- 6. 鍵の漏えいなどがあれば、期限前に失効させる

形式: PEM・DER・PKCS [331.1]

同じ証明書でも、ファイルの形式は2つあります。PEM は、DER を Base64 にして `-----BEGIN ...-----` の行で挟んだ文字形式です (Linux の標準)。DER は Base64 にする前のバイナリで、Windows や Java でよく見かけます。

```
$ openssl x509 -in web.crt -outform DER -out web.der
$ openssl x509 -inform DER -in web.der -noout -subject
subject=CN=www.example.com
$ openssl pkcs12 -export -in web.crt -inkey web.key -certfile ca.crt -
out web.p12
```

入出力の形式は `-inform` と `-outform` で指定し、省略すると PEM になります。PKCS は RSA 社が決めた規格群の名前で、番号と用途の対応が問われます。

規格・形式	用途	PEM の見出し行
PKCS#1	RSA 鍵そのものの形式	BEGIN RSA PRIVATE KEY
PKCS#8	秘密鍵の汎用形式(鍵の種類によらない)	BEGIN PRIVATE KEY、暗号化したものは BEGIN ENCRYPTED PRIVATE KEY
PKCS#10	証明書署名要求(CSR)	BEGIN CERTIFICATE REQUEST
PKCS#7	証明書の束(鍵は含まない)。 <code>.p7b</code>	BEGIN PKCS7
PKCS#12	証明書・中間 CA・秘密鍵を1つのパスワード付きファイルにまとめる。 <code>.p12</code> 、 <code>.pfx</code> 。 <code>openssl pkcs12 -export</code> で作る	バイナリ
その他	証明書、公開鍵、CRL	BEGIN CERTIFICATE、 BEGIN PUBLIC KEY、 BEGIN X509 CRL

OpenSSL 3.0 の `genrsa` が出力するのは PKCS#8 形式で、見出し行は `BEGIN PRIVATE KEY` です。昔の `BEGIN RSA PRIVATE KEY` の形は、`-traditional` を付けないと出ません。

鍵の生成と管理 [331.1]

RSA の秘密鍵は、`openssl genrsa` でも、より新しい `openssl genpkey` でも作れます。[^c1-3] 最後の数 (numbits) が鍵の長さ (ビット) で、OpenSSL 3.0 の既定は 2048 です。

```
$ openssl genrsa -aes256 -out ca.key 4096
$ openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -out
web.key
$ openssl genpkey -algorithm EC -pkeyopt ec_paramgen_curve:P-256 -out
ec.key    # EC 鍵
$ openssl rsa -in web.key -pubout -out web.pub
$ openssl rsa -in web.key -noout -text
```

`-aes256` を付けると (`-aes128`、`-des3` も)、秘密鍵がパスフレーズで暗号化されます (CA の鍵には必ず付ける)。パスフレーズはコマンド行に直書きせず (`pass:パスフレーズ` は `ps` で見える)、`-passin file:` や `env:` で渡すか、対話入力にします。`openssl rsa` の `-pubout` は、秘密鍵から公開鍵を取り出します。`-text -noout` は、modulus・publicExponent・素数 p と q を文字で表示し、符号化した鍵は出力しません。秘密鍵のファイルは、所有者だけが読める権限にします (`chmod 600`、CA の鍵は `400`)。公開鍵と証明書は、配ってかまいません。

CSR と自己署名証明書 [331.1]

CSR (Certificate Signing Request) は、持ち主の名前と公開鍵、欲しい拡張を、持ち主自身の秘密鍵で署名したものです。[^c1-4] CA は署名を検証して、申請者が秘密鍵を持っていることを確かめます。CA に送るのは CSR だけで、秘密鍵は送りません。

```
$ openssl req -new -key web.key -subj "/CN=www.example.com" \
    -addext "subjectAltName=DNS:www.example.com,DNS:example.com" -out
web.csr

$ openssl req -in web.csr -noout -verify -text
Certificate request self-signature verify OK

$ openssl req -x509 -newkey rsa:2048 -noenc -keyout s.key -out s.crt -
days 365 -subj "/CN=selfsigned.test"
```

オプション	意味
-new -key	既存の鍵から CSR を作る。-newkey rsa:2048 なら鍵も同時に作る
-noenc	秘密鍵を暗号化せずに保存する (3.0 より前の名前は -nodes)
-x509	CSR の代わりに自己署名証明書を出力する。-days で有効日数
-addext / -subj	SAN などの拡張を足す / 対話入力を省いて名前を渡す

認証局 (CA) の構築と運用 [331.1]

自前の CA は `openssl ca` で運用します。この命令は、設定ファイル (`openssl.cnf`) の `[ ca ]` 節にある `default_ca` が指す節を読み、台帳を更新しながら署名します。[^c1-5] 先に、次のディレクトリとファイルを作っておきます (`openssl.cnf` と同じ階層)。

名前	役割
<code>private/</code>	CA の秘密鍵。権限を絞る
<code>certs/</code>	CA 自身の証明書
<code>newcerts/</code>	発行した証明書の控え (<code>シリアル.pem</code>)
<code>crl/</code>	発行した CRL
<code>index.txt</code>	発行した証明書の台帳。最初は空のファイル
<code>serial</code>	次のシリアル番号 (16進。例 <code>1000</code>)
<code>crlnumber</code>	次の CRL 番号

```
[ ca ]
default_ca = CA_default
[ CA_default ]
dir                = /root/ca
database           = $dir/index.txt
new_certs_dir      = $dir/newcerts
certificate        = $dir/certs/ca.crt
private_key        = $dir/private/ca.key
serial            = $dir/serial
crlnumber          = $dir/crlnumber
default_md         = sha256
policy             = policy_any
copy_extensions    = copy
[ v3_ca ]
basicConstraints   = critical,CA:TRUE,pathlen:0
keyUsage           = critical,keyCertSign,cRLSign
```

`policy` は、CSR の名前の欄の条件を決める節です (`match` は CA と同じ値、`supplied` は必須、`optional` は任意)。 `copy_extensions = copy` を書くと、CSR の拡張 (SAN など) が証明書へ写ります。この行がないか `none` だと、CSR の拡張は無視されて、SAN のない証明書が出ます。

ルート CA の準備と自己署名、サーバ証明書の発行

```
$ touch index.txt; echo 1000 > serial; echo 1000 > crlnumber
$ openssl genrsa -aes256 -out private/ca.key 2048
$ openssl req -config openssl.cnf -new -x509 -key private/ca.key -days
3650 \
    -subj "/CN=Example Root CA" -extensions v3_ca -out certs/ca.crt
$ openssl ca -config openssl.cnf -days 365 -in web.csr -out web.crt -
notext
$ cat index.txt
V          271013102345Z          1000          unknown
/CN=www.example.com
```

`index.txt` の各行は、タブ区切りで「状態・期限・失効日時・シリアル・unknown・subject」です。状態は `V` (有効)、`R` (失効)、`E` (期限切れ) の3つです。失効した行は、3列目に失効日時が入ります(理由つきなら `,keyCompromise` のように続く)。`-notext` は証明書の前の文字表示を省きます。`-extensions` は拡張の節名、`-days` は有効日数の上書き、`-batch` は確認の質問の省略です。

`openssl x509 -req` でも署名はできますが、台帳を持たないので失効を管理できません(試験用の簡易な手段)。`-CAcreateserial` は、CA 証明書と同じ場所に `ca.srl` というシリアルのファイルを作ります(`openssl ca` が使う `serial` ファイルとは別物)。

```
$ openssl x509 -req -in web.csr -CA certs/ca.crt -CAkey private/ca.key
-CAcreateserial -days 90 -out web2.crt
```

ルート CA の鍵は、パスフレーズ付きにして、通常はネットワークから切り離れた機器に置きます。日常の発行は、ルートが署名した中間 CA に任せます。CA の証明書には `CA:TRUE` と `pathlen` を、サーバ証明書には `CA:FALSE` を入れ、サーバ証明書が別の証明書を発行できないようにします。

`pathlen` の数え方は罫になります。ルートに `pathlen:0` を付けると、ルートの下に中間 CA を置けません。ルートが `pathlen:0` で中間 CA を発行した場合、その下のサーバ証明書は error 25 (`path length constraint exceeded`) で拒否されます。中間 CA を置くルートには `pathlen:1` 以上 (または `pathlen` なし) を、サーバ証明書を直接発行する中間 CA には `pathlen:0` を付けます。

信頼の連鎖・PKI・証明書の透明性 [331.1]

PKI (公開鍵基盤) は、鍵・証明書・CA・失効の仕組みを合わせた全体の呼び名です。クライアントは、サーバ証明書から出発して発行者をたどり、最初から信頼している証明書 (トラストアンカー、通常はルート CA) に届くかを確かめます。この道筋を信頼の連鎖 (証明書パス) と呼びます (ルート CA・中間 CA・サーバ証明書の3段が典型)。途中の中間 CA 証明書をクライアントが持っていないければ、検証は先へ進みません。サーバは、自分の証明書に続けて中間 CA も送ります (Apache の設定は331.2で扱う)。

`openssl verify` では、`-CAfile` で信頼するルートを、`-untrusted` で途中の中間 CA を渡します。

```
$ openssl verify -CAfile certs/ca.crt leaf.crt
error 20 at 0 depth lookup: unable to get local issuer certificate
$ openssl verify -CAfile certs/ca.crt -untrusted int.crt leaf.crt
leaf.crt: OK
```

`unable to get local issuer certificate` (error 20) は、発行者の証明書が見つからない合図です(中間 CA の渡し忘れて出る)。クライアントは、署名・有効期限・用途・名前(SAN)・失効を見ます(連鎖の各段では `basicConstraints` と `pathlen` も)。

証明書の透明性(Certificate Transparency、CT)は、CA が発行した証明書を、誰でも読める追記専用の公開ログに載せる仕組みです。[^c1-6] 持ち主はログを監視して、自分のドメインの証明書が勝手に発行されていないかを見つけられます。ログに載せると SCT (Signed Certificate Timestamp) が返ります。SCT は、証明書の拡張、TLS の拡張、OCSP 応答のどれかでクライアントへ届きます(SCT のない証明書を受け付けないブラウザもある)。

失効:CRL と OCSP [331.1]

鍵が漏れた、持ち主が変わった、といった事情があれば、期限の前に証明書を失効させます(知らせ方は CRL と OCSP の2つ)。

CRL (Certificate Revocation List) は、失効した証明書のシリアル番号と失効日時を並べた、CA 署名付きのリストです。`thisUpdate` (発行日時)と `nextUpdate` (次の更新予定)を持ちます。クライアントは `crlDistributionPoints` の URL から取得します。`nextUpdate` を過ぎた CRL は新しい情報として扱われないため、cronなどで定期的に再発行します。

```
$ openssl ca -config openssl.cnf -revoke web.crt -crl_reason
keyCompromise
Revoking Certificate 1000.
Database updated

$ openssl ca -config openssl.cnf -gencrl -out crl/ca.crl

$ openssl crl -in crl/ca.crl -noout -text

    Issuer: CN=Example Root CA

    Last Update: Oct 13 10:23:45 2026 GMT

    Next Update: Nov 12 10:23:45 2026 GMT

    Serial Number: 1000

    Revocation Date: Oct 13 10:23:45 2026 GMT

$ openssl verify -crl_check -CAfile ca-and-crl.pem web.crt
error 23 at 0 depth lookup: certificate revoked
```

オプション	意味
<code>-revoke</code> 証明書	台帳の行を <code>R</code> に書き換える。CRL にはまだ反映されない
<code>-gencrl</code>	台帳から CRL を作り直す
<code>-crl_reason</code> 理由	<code>keyCompromise</code> 、 <code>CACompromise</code> 、 <code>affiliationChanged</code> 、 <code>superseded</code> 、 <code>cessationOfOperation</code> など
<code>-crl days</code> / <code>-crl hours</code>	次の更新までの長さ

`openssl verify` で失効を見るには、`-crl_check` を付け、CRL 連結の PEM を `-CAfile` に渡します (CRL がなければ error 3 で止まる)。

OCSP (Online Certificate Status Protocol) は、証明書1枚の状態をレスポндаに問い合わせる方法です(問い合わせ先は `authorityInfoAccess`)。応答は `good`、`revoked`、`unknown` の3つで、レスポндаが署名します。

```
$ openssl ocsp -port 8888 -index index.txt -CA certs/ca.crt \  
    -rsigner certs/ca.crt -rkey private/ca.key  
  
$ openssl ocsp -issuer certs/ca.crt -cert web.crt -url  
http://127.0.0.1:8888 -CAfile certs/ca.crt  
  
Response verify OK  
  
web.crt: revoked  
  
    This Update: Oct 13 10:23:55 2026 GMT  
    Revocation Time: Oct 13 10:23:54 2026 GMT
```

openssl ocsp の使い方	意味
<code>-index</code> / <code>-port</code> / <code>-rsigner</code> / <code>-rkey</code>	台帳から答える簡易レスポндаになる(1回目)
<code>-issuer</code> / <code>-cert</code> / <code>-url</code>	発行者の証明書、調べる証明書、レスポндаの場所(2回目)

比較	CRL	OCSP
取得するもの	失効した全証明書のリスト	1枚の状態
鮮度	<code>nextUpdate</code> までは古い情報のまま	問い合わせた時点の情報
負荷	リストが大きくなる	接続のたびの問い合わせで遅くなり、レスポнда停止にも弱い

比較	CRL	OCSP
プライバシー	閲覧先が CA に伝わらない	閲覧先のサイトが CA に伝わる

OCSP の負荷とプライバシーの問題は、サーバが応答をあらかじめ取得して握手に添える OCSP ステージングで和らぎます (331.2 で扱う)。Let's Encrypt は2024年12月に、2025年8月6日に OCSP レスポンダを止めると発表しました (失効情報は CRL に一本化する)。^[^c1-7] 試験の範囲には、OCSP と CRL の両方が入っています。

CA の証明書を失効させるときは、その CA に署名した上位の CA が CRL に載せます。中間 CA を失効させると、その配下の証明書はすべて信頼できなくなります。ルート CA には上位がないので、失効の手段がありません。クライアントのトラストストアから外して、信頼を取り消します。

Let's Encrypt・ACME・certbot [331.1]

Let's Encrypt は、無料で証明書を自動発行する公開 CA です (運営は非営利団体 ISRG)。自動化の手順を決めたプロトコルが ACME (RFC 8555) で、クライアントのソフトに certbot があります。サーバは、そのドメインを管理していることを「チャレンジ」で示します。^[^c1-8]

チャレンジ	証明の方法	特徴
http-01	<code>http://ドメイン/.well-known/acme-challenge/トークン</code> にファイルを置く	80番ポートが必要。ワイルドカード証明書は取れない

チャレンジ	証明の方法	特徴
dns-01	<code>_acme-challenge.ドメイン</code> に TXT レコードを置く	ワイルドカード証明書が取れる。DNS を API で更新できると自動化しやすい
tls-alpn-01	443番ポートで ALPN の <code>acme-tls/1</code> に応答する	80番を開けられない環境向け

```
$ certbot certonly --webroot -w /var/www/html -d example.com -d www.example.com
$ certbot certonly --standalone -d example.com
$ certbot --apache -d example.com
$ certbot renew --dry-run
```

指定	動き
<code>certonly --webroot -w ルート</code>	動いている Web サーバのドキュメントルートにチャレンジのファイルを置く
<code>certonly --standalone</code>	certbot が一時的に80番ポートで待ち受ける。Web サーバは止めておく
<code>--apache / --nginx</code>	証明書の取得とサーバ設定の書き換えまで行うプラグイン
<code>renew</code>	期限が近い証明書を更新する。 <code>--dry-run</code> は試験用サーバで手順だけ試す
<code>revoke --cert-name ドメイン</code>	証明書を失効させる

取得した証明書は `/etc/letsencrypt/live/ドメイン/` に置かれます。[^c1-9]

- `fullchain.pem` : サーバ証明書と中間 CA。Apache の `SSLCertificateFile` に使う
- `privkey.pem` : 秘密鍵。Apache の `SSLCertificateKeyFile` に使う
- `cert.pem` : サーバ証明書だけ
- `chain.pem` : 中間 CA だけ

証明書の有効期間は90日で、2027年2月に64日、2028年2月に45日へ短くなる予定です(2025年12月の発表)。短い期間を自動更新で回すので、`certbot renew` を cron や systemd のタイマーで定期的に走らせます。

CFSSL [331.1]

CFSSL は、Cloudflare が作った PKI と TLS の道具一式です。[^c1-10] 設定と入力を JSON で書き、`cfssl` が 鍵・CSR・証明書を JSON で出力し、`cfssljson` がファイルに書き分けます。署名の API サーバとして動かすこともできます(`openssl ca` にはない使い方)。

```
$ cfssl gencert -initca ca-csr.json | cfssljson -bare ca
$ cfssl gencert -ca ca.pem -ca-key ca-key.pem -config ca-config.json \
    -profile server server.json | cfssljson -bare server
$ ls
ca-key.pem  ca.csr  ca.pem  server-key.pem  server.csr  server.pem
```

要素	内容
<code>gencert -initca</code>	自己署名のルート CA を作る
<code>gencert -ca ... -profile</code>	既存の CA で署名してサーバ証明書を出す

要素	内容
<code>ca-config.json</code> の <code>profiles</code>	用途 (<code>usages</code>) と有効期間 (<code>expiry</code> 、例 2160h)。CSR の JSON には <code>CN</code> 、 <code>hosts</code> (SAN)、 <code>key</code> を書く
<code>cfssl serve</code>	API サーバ。既定の待ち受けは <code>127.0.0.1:8888</code>
<code>cfssl ocspsign</code> / <code>ocspserve</code> / <code>certinfo</code>	OCSP 応答の作成 / 提供 / 証明書の中身の表示

SSL/TLS:版・握手・暗号スイート [331.2]

SSL は Netscape が作り、IETF が TLS として引き継ぎました。SSL 2.0 と 3.0 は破られていて、使えません。TLS 1.0 と 1.1 も、RFC 8996 で使用をやめるよう定められました。[^c1-11] 現役は TLS 1.2 と TLS 1.3 です。

版	状態	特徴
SSL 2.0 / 3.0	廃止	Apache の既定は SSLv3 を外す
TLS 1.0 / 1.1	非推奨 (RFC 8996)	<code>SSLProtocol</code> で明示的に外す
TLS 1.2	現役	暗号スイートに鍵交換と認証の方式が入る。握手は2往復
TLS 1.3	現役 (RFC 8446)	RSA 鍵交換を廃止し、鍵交換は ECDHE/DHE が基本で PFS が成り立つ。握手は1往復。弱い暗号を削除

握手は、クライアントの ClientHello (使える版・暗号スイート・SNI) で始まり、サーバが ServerHello と証明書で答えます。クライアントは証明書を検証し、鍵交換で共通鍵を作って、以降の通信を暗号化します。TLS 1.3 は握手の途中から暗号化が始まるので、証明書も盗み見られません。

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

ECDHE	鍵交換（使い捨ての鍵。PFS が成り立つ）
RSA	サーバ認証（証明書の鍵の種類）
AES_256_GCM	通信データの暗号化（共通鍵と動作モード）
SHA384	鍵導出などに使うハッシュ

OpenSSL の表記では、ハイフンでつないで `ECDHE-RSA-AES256-GCM-SHA384` と書きます。TLS 1.3 のスイートは `TLS_AES_256_GCM_SHA384` の形で、鍵交換と認証は名前に現れません（握手の中で別に決まる）。TLS 1.3 のスイートは少なく、次の3つが中心です（一覧は `openssl ciphers -s -tls1_3`）。

- `TLS_AES_128_GCM_SHA256`
- `TLS_AES_256_GCM_SHA384`
- `TLS_CHACHA20_POLY1305_SHA256`

避けるのは NULL (暗号化なし)・EXPORT・RC4・DES と 3DES・MD5・匿名 (aNULL) です。`SSLCipherSuite HIGH:!aNULL:!MD5` のように、`!` で除外します。

Apache (mod_ssl) で HTTPS を出す [331.2]

Apache HTTPD 2.4 では、`mod_ssl` が TLS を担当します。HTTPS の既定ポートは443です（`Listen 443 https` で待ち受ける）。[^{^c1-12}] Rocky Linux は `mod_ssl` を入れると `conf.d/ssl.conf` ができ、`httpd.conf` から読み込まれます。Debian 系は `a2enmod ssl` で有効にします。

```
Listen 443 https
<VirtualHost *:443>
    ServerName www.example.com
    DocumentRoot /var/www/www
    SSLEngine on
    SSLProtocol all -SSLv3 -TLSv1 -TLSv1.1
    SSLCertificateFile /etc/pki/tls/web.crt
    SSLCertificateKeyFile /etc/pki/tls/web.key
</VirtualHost>
```

ディレクティブ	役割
SSLEngine on	その仮想ホストで TLS を有効にする
SSLCertificateFile	サーバ証明書 (PEM)。2.4.8 以降は、続けて中間 CA 証明書も置ける
SSLCertificateKeyFile	秘密鍵 (PEM)。root だけが読める権限にする
SSLCertificateChainFile	中間 CA 証明書。2.4.8 で廃止予定になり、SSLCertificateFile へ連結する
SSLProtocol	許す版。既定は all -SSLv3 。-TLSv1 -TLSv1.1 を足して古い版を外す
SSLCipherSuite	許す暗号スイートの並び (OpenSSL の書式)
SSLHonorCipherOrder	on でサーバ側の並びを優先する。既定は off (クライアントの希望を優先)
SSLPassPhraseDialog	暗号化した秘密鍵のパスフレーズの聞き方。既定は builtin (起動時に端末で入力)

設定を変えたら、`httpd -t` で構文を確かめ、`systemctl reload httpd` で反映します。暗号化した秘密鍵は起動のたびに入力が必要、そのままでは自動起動できません（`SSLPassPhraseDialog exec:` で外部プログラムに答えさせる方法はある）。

証明書チェーンは、サーバが自分の証明書の後ろに中間 CA を続けて送ります。ルート CA はクライアントが持っているので、送れません。2.4.8 以降は、サーバ証明書と中間 CA を連結した1ファイルを `SSLCertificateFile` に指定します。

```
$ cat web.crt int.crt > web-fullchain.crt
SSLCertificateFile /etc/pki/tls/web-fullchain.crt
$ openssl s_client -connect www.example.com:443 -servername
www.example.com -showcerts </dev/null # 送られた証明書を全部表示
```

中間 CA を送り忘れたサーバでも、PC のブラウザでは動くことがあります（ブラウザが中間 CA を手元に持っていたり、取りに行ったりして補う）。curl や Java では、error 20 や error 21 になります（下の表）。

暗号の設定は、TLS 1.2 以前と TLS 1.3 で別です。TLS 1.3 のスイートは、プロトコル指定の `TLSv1.3` を付けた `SSLCipherSuite` で指定します。TLS 1.3 には再ネゴシエーションがないので、ディレクトリ単位ではこの指定ができません。問われやすいのは、`SSLProtocol` で古い版を外すこと、`SSLCipherSuite` の除外の `!`、`SSLHonorCipherOrder` の意味です。

SNI と名前ベースの仮想ホスト [331.2]

HTTP のホスト名による仮想ホストの選択は、TLS の握手が終わってからでないといけません。すると、証明書を選ぶ時点ではホスト名が分からず、1つの IP

アドレスとポートには1枚の証明書しか出せません。SNI (Server Name Indication) は、ClientHello にホスト名を入れて送る TLS の拡張で、この問題を解きます。

```
<VirtualHost *:443>
    ServerName www.example.com

    SSLEngine on

    SSLCertificateFile /etc/pki/tls/web.crt

    SSLCertificateKeyFile /etc/pki/tls/web.key
</VirtualHost>

<VirtualHost *:443>
    ServerName blog.example.com

    SSLEngine on

    SSLCertificateFile /etc/pki/tls/blog.crt

    SSLCertificateKeyFile /etc/pki/tls/blog.key
</VirtualHost>
```

同じ `*:443` に仮想ホストを並べ、`ServerName` で区別します。クライアントが送った SNI と `ServerName` (または `ServerAlias`) が一致する仮想ホストの証明書が使われます。

```
$ openssl s_client -connect 127.0.0.1:443 -servername blog.example.com
# SNI を送る
subject=CN=blog.example.com

$ openssl s_client -connect 127.0.0.1:443
# SNI を送らない
subject=CN=www.example.com
```

SNI を送らない古いクライアントや、IP アドレスで直接つないだ接続には、最初にした仮想ホストの証明書が出ます。 `SSLStrictSNIVHostCheck on` を既定の仮想ホストに書くと、SNI を送らないクライアントは、どの仮想ホストにも入れません(既定値は `off`)。

HSTS [331.2]

HSTS (HTTP Strict Transport Security、RFC 6797) は、「このサイトには今後 HTTPS だけでつなげ」とブラウザに覚えさせるヘッダです。[^c1-13] HTTP 経由の接続を乗っ取る攻撃 (SSL ストリップ) を防ぐため、`mod_headers` で HTTPS の応答に付けます。

```
Header always set Strict-Transport-Security "max-age=63072000;
includeSubDomains"

$ curl -sS -D - https://www.example.com/ -o /dev/null | grep -i strict
Strict-Transport-Security: max-age=63072000; includeSubDomains
```

項目	意味
<code>max-age</code>	必須。ブラウザが覚えておく秒数 (63072000 秒は2年)
<code>includeSubDomains</code>	サブドメインにも適用する

HTTP で受け取ったヘッダは、無視されます。HSTS は HTTPS の仮想ホストに書き、HTTP の側には HTTPS へのリダイレクトを置きます。受け取ったブラウザは、証明書の期限切れなどの警告も、利用者が無視して進めなくなります (更新を怠ると、サイトにつながらない)。

クライアント証明書による利用者認証 [331.2]

mod_ssl は、サーバだけでなくクライアントも証明書で認証できます (相互 TLS)。パスワードの代わりに、CA が発行したクライアント証明書を持つ利用者だけを通します。

```
<VirtualHost *:443>
    ServerName secure.example.com

    SSLEngine on

    SSLCertificateFile    /etc/pki/tls/sec.crt
    SSLCertificateKeyFile /etc/pki/tls/sec.key
    SSLCACertificateFile  /etc/pki/tls/ca.crt

    SSLVerifyClient require

    SSLVerifyDepth 1
</VirtualHost>
```

ディレクティブ	役割
SSLVerifyClient	none (既定、証明書を求めない)、optional (出せば検証)、require (必須)、optional_no_ca (出せばよく、CA の検証には通らなくてもよい)
SSLCACertificateFile / SSLCACertificatePath	クライアント証明書を発行した CA の証明書。ファイル1つか、ハッシュ名のリンクを並べたディレクトリ(openssl rehash で作る)
SSLVerifyDepth	たどる CA の段数。既定は1。0は自己署名のクライアント証明書だけ。1は信頼する CA が直接発行した証明書まで

ディレクティブ	役割
<code>SSLCARevocationFile</code> / <code>SSLCARevocationPath</code>	クライアント証明書の失効を調べる CRL
<code>SSLCARevocationCheck</code>	<code>chain</code> か <code>leaf</code> で CRL の確認を有効にする。既定は <code>none</code>
<code>SSL0CSPEnable</code>	クライアント証明書の OCSP 確認。既定は <code>off</code> (サーバ側のステープリングとは別)
<code>SSLOptions +StdEnvVars</code> <code>+ExportCertData</code>	証明書の情報を環境変数 (<code>SSL_CLIENT_S_DN_CN</code> など)に出し、CGI から読めるようにする

`SSLCACertificateFile` を書き忘れると、検証できる CA がなく、どのクライアントも通りません。クライアント証明書が中間 CA の発行なら `SSLVerifyDepth 2` が必要で、既定の1のままでは拒否されます。証明書の中身で絞るには、`Require expr "%{SSL_CLIENT_S_DN_CN} == 'alice'"` のように書きます。クライアント証明書は、`extendedKeyUsage = clientAuth` で発行します。

クライアントの状態	結果
証明書を出さない	TLS 1.3 では <code>tlsv13 alert certificate required</code> で切られる
信頼されない証明書	<code>unknown ca</code> で切られる。エラーログの番号は、自己署名の証明書なら <code>Error (18): self-signed certificate</code> 、知らない CA が発行した証明書なら <code>Error (20): unable to get local issuer certificate</code>
正しい証明書	通る

ディレクトリ単位で `SSLVerifyClient` を書くと、リクエストを読んだあとで再ネゴシエーションが起きます。

OCSP ステープリング [331.2]

OCSP ステープリングは、サーバが自分の証明書の OCSP 応答を先に取得して、TLS の握手に添える方法です (RFC 6066)。クライアントが CA に問い合わせる必要がなくなり、閲覧先を CA に知られません。応答は CA が署名しているので、サーバが偽造することはできません。

```
SSLStaplingCache "shmcb:/run/httpd/ssl_stapling(32768)"  
<VirtualHost *:443>  
  
...  
    SSLUseStapling on  
</VirtualHost>
```

`SSLStaplingCache` はサーバ全体の設定で、`<VirtualHost>` の外に書きます。キャッシュの設定は必須で、ないと有効にできません (`SSLUseStapling` の既定は `off`)。応答の取得先は、証明書の `authorityInfoAccess` に入っている OCSP の URL で、取得した応答はキャッシュして使い回します。

```
$ openssl s_client -connect 127.0.0.1:443 -servername
secure.example.com -status </dev/null
OCSP response: no response sent          ← ステープリングが働いていない
OCSP Response Status: successful (0x0)   ← 働いている
# 証明書に tlsfeature = status_request (Must-Staple) の拡張があると、この拡張を解釈するクライアントは、ステープルのない握手を拒否する (RFC 7633)
```

openssl s_client と s_server で試す [331.2]

`openssl s_client` は、TLS のクライアントとして接続し、握手の結果を表示します。`openssl s_server` は、試験用の TLS サーバとして待ち受けます。

オプション	意味
<code>-connect host:port</code>	接続先
<code>-servername 名前</code>	SNI を送る
<code>-CAfile</code> / <code>-CApath</code>	信頼する CA。付けないと既定の信頼ストア (OS の CA 群) で検証するので、自前の CA が発行した証明書は通らない (既定のストアも読まないのは <code>-no-CAfile</code>)
<code>-showcerts</code>	サーバが送った証明書をすべて表示
<code>-status</code>	OCSP 応答 (ステープル) を要求して表示
<code>-tls1_2</code> / <code>-tls1_3</code>	使う版を指定。古い版を拒否する設定の確認にも使う
<code>-starttls smtp</code>	平文で始めて TLS へ切り替える (smtp、imap、ftp など)

オプション	意味
<code>-cert</code> / <code>-key</code>	クライアント証明書で接続する
<code>-verify_hostname</code> 名前	証明書の SAN と照合するホスト名
<code>-brief</code>	版、暗号スイート、検証結果だけを短く表示

Verify return code	意味
0 (ok)	検証成功
10	証明書の期限切れ
18	自己署名証明書
20	発行者の証明書が見つからない(中間 CA の送り忘れなど)
21	最初の証明書を検証できない(信頼する CA の指定がないとき)
62	ホスト名の不一致 (SAN にない名前がつないだ)

```
$ openssl s_client -connect 127.0.0.1:443 -servername www.example.com -
CAfile ca.crt -brief </dev/null
CONNECTION ESTABLISHED
Protocol version: TLSv1.3
Ciphersuite: TLS_AES_256_GCM_SHA384
Peer certificate: CN=www.example.com
Verification: OK

$ openssl s_server -accept 4433 -cert web.crt -key web.key -www      # 待
ち受けポート、証明書、鍵。-www は簡単な状態の HTML を返す

$ openssl s_client -connect 127.0.0.1:4433 -CAfile ca.crt -
verify_hostname wrong.example.com
Verify return code: 62 (hostname mismatch)
```

古い版を拒否したことの確認では、サーバが `SSLProtocol` で外した版を `s_client -tls1_1` で試します。握手は成立しません。

ブロックデバイスの暗号化とファイルシステムの暗号化 [331.3]

ディスクの暗号化には、層の違う2つの方式があります。

方式	暗号化する層	代表	隠れるもの
ブロックデバイス暗号化	デバイス全体(ブロック単位)	dm-crypt と LUKS	ファイル名・サイズ・ディレクトリ構造を含む中身のすべて
ファイルシステム暗号化	ファイルシステムの上に重ねる	eCryptfs	ファイルの中身と(設定すれば)ファイル名。下の層のファイルの数と大きさは見える

用語	役割
dm-crypt	カーネルの device-mapper の暗号化ターゲット。 <code>/dev/mapper/名前</code> の仮想デバイスを作り、読み書きのたびに暗号化と復号を行う
cryptsetup	dm-crypt と LUKS を操作するコマンド
LUKS (Linux Unified Key Setup)	デバイスの先頭に置くヘッダの規格。暗号方式・鍵スロット・鍵の導出方法を記録する
plain	LUKS を使わない dm-crypt。ヘッダがない

dm-crypt と LUKS1: cryptsetup [331.3]

LUKS では、データを暗号化する本物の鍵(マスター鍵)を乱数で作ります。パスフレーズからは別の鍵を導出し、その鍵でマスター鍵を包んで、ヘッダの鍵スロットに保存します。LUKS1 のスロットは8個です。[^c1-14] スロットごとに別のパスフレーズを持てるので、追加・削除・変更をしてもデータの再暗号化は要りません。反対に、ヘッダが壊れると、パスフレーズが正しくてもデータは開けません。

使い始めの手順は、次の4つです。

- `cryptsetup luksFormat --type luks1 /dev/sdb1` : LUKS として初期化する。
中身は消える。確認で大文字の `YES` を入力する
- `cryptsetup open /dev/sdb1 vault` : パスフレーズで開き、`/dev/mapper/vault` ができる (`luksOpen` は古い名前)
- `mkfs.ext4 /dev/mapper/vault` で初回だけファイルシステムを作り、`mount /dev/mapper/vault /mnt/vault` でマウントする
- 終わったら `umount` して、`cryptsetup close vault` (`luksClose`) で閉じる

```
$ cryptsetup luksDump disk.img
Version:          1
Cipher name:      aes
Cipher mode:      xts-plain64
Hash spec:        sha256
Payload offset:   4096
MK bits:          512
Key Slot 0: ENABLED
Key Slot 1: DISABLED
...
Key Slot 7: DISABLED
$ cryptsetup status vault
/dev/mapper/vault is active.
  type:    LUKS1
  cipher:  aes-xts-plain64
  keysize: 512 [bits]
```

`Payload offset` は、データが始まる位置です (512バイト単位)。`aes-xts-plain64` はディスク暗号の定番の指定です。XTS は鍵を2本使うため、`keysize` は512ビットになります (AES-256 相当)。`luksFormat` の `--cipher`、`--key-size`、`--hash`、`--iter-time` で既定を変えられます。

サブコマンド	働き
<code>luksFormat</code>	LUKS として初期化 (データは失われる)。 <code>-type luks1</code> または <code>luks2</code>
<code>open</code> / <code>luksOpen</code>	開いて <code>/dev/mapper/名前</code> を作る

サブコマンド	働き
<code>close</code> / <code>luksClose</code>	閉じてマッピングを消す
<code>luksAddKey</code>	新しいパスフレーズを空きスロットに追加。 既存のパスフレーズで認証する
<code>luksRemoveKey</code>	渡したパスフレーズに一致するスロットを消す
<code>luksKillSlot</code>	スロット番号を指定して消す。残る別のスロットのパスフレーズで認証する
<code>luksChangeKey</code>	パスフレーズを変更する
<code>luksDump</code>	ヘッダの内容を表示 (スロットの使用状況を含む)
<code>isLuks</code>	LUKS かどうかを終了コードで返す (0ならLUKS)
<code>luksUUID</code>	デバイスの UUID を表示
<code>luksHeaderBackup</code> / <code>luksHeaderRestore</code>	<code>--header-backup-file</code> でヘッダを保存・書き戻し
<code>status</code>	開いているマッピングの暗号・鍵長・デバイスを表示

鍵スロットの扱いは、場面で問われます。

```
$ cryptsetup luksAddKey disk.img /tmp/k2 --key-file /tmp/k1      # k1
(スロット0) で認証し、k2 をスロット1に足す

$ cryptsetup luksKillSlot disk.img 1 --key-file /tmp/k1          # スロ
ット1を番号で消す。認証は残るスロットの鍵

$ cryptsetup luksRemoveKey disk.img /tmp/k2                      # 消え
た k2 を指す

No key available with this passphrase.

$ cryptsetup luksHeaderBackup /dev/sdb1 --header-backup-file hdr.bak
$ cryptsetup luksHeaderRestore /dev/sdb1 --header-backup-file hdr.bak
```

全部のスロットを消すとデータは開けなくなるため、`cryptsetup` は最後の1つを消すときに警告を出します(対話で消す場合。 `--key-file` や `-q` を付けると確認なしで消えます)。パスフレーズの代わりにファイルを鍵に使うときは、 `--key-file` を指定します(標準入力なら `-d -`)。ヘッダのバックアップは、取った時点の鍵スロットの状態を持ちます。あとで消したパスフレーズも、古いヘッダを書き戻せば開けるので、バックアップ自体を厳重に保管します。

/etc/crypttab と起動時の自動利用 [331.3]

起動時に暗号化デバイスを開く設定が `/etc/crypttab` で、1行が4つの欄からなります。[^c1-15]

欄	内容	例
1	マッピングの名前 (<code>/dev/mapper/</code> の下にで きる)	<code>vault</code>

欄	内容	例
2	元のデバイス。UUID= で指定できる	UUID=2bcb3623-f876-4796-a815-0bccecfb56bc
3	鍵ファイルのパス。none か - なら起動時にパスフレーズを聞く	none、/root/vault.key
4	オプション(コンマ区切り)	luks,discard

```
vault UUID=2bcb3623-f876-4796-a815-0bccecfb56bc none luks,discard
data /dev/sdb1 /root/data.key luks,nofail
swap /dev/sdc2 /dev/urandom swap # 起動ごとに鍵が変わり、前
回の中身は残らない
```

4欄目でよく使うのは、次の4つです。

- luks : LUKS として開く
- discard : TRIM を通す
- noauto : 起動時には自動で開かない
- nofail : 開けなくても起動を止めない

開いたあとの /dev/mapper/vault は、/etc/fstab に書いてマウントします。デバイス名は、変わらない UUID で書くのが安全です (UUID は cryptsetup luksUUID や blkid で調べる)。ルートファイルシステムを暗号化している場合は、initramfs の中で開くため、crypttab の変更後に initramfs を作り直します (Red Hat 系は dracut -f、Debian 系は update-initramfs -u)。

plain dm-crypt [331.3]

plain モードはヘッダを持たず、暗号・鍵長・ハッシュを開くたびに自分で指定します。

```
$ cryptsetup open --type plain p.img plainvol
$ cryptsetup status plainvol          # type: PLAIN, cipher: aes-cbc-
essiv:sha256, keysize: 256 [bits]
$ cryptsetup close plainvol
```

項目	LUKS	plain
ヘッダ	あり(暗号・鍵スロット・鍵導出の情報)	なし
パスフレーズの検証	違うと開けない	違ってても開く。中身は意味のないデータになる
鍵スロット	複数	なし。パスフレーズの追加や変更はできない
暗号・鍵長・ハッシュ	ヘッダに記録	開くたびに指定
判別	<code>luksDump</code> や <code>isLuks</code> で分かる	外から暗号化デバイスだと分らない
マッピングの操作	<code>close</code> 、 <code>status</code> 、 <code>resize</code>	<code>close</code> (古い名は <code>remove</code>)、 <code>status</code> 、 <code>resize</code>

古い書き方は `cryptsetup create 名前 デバイス` で、今は `open --type plain` に置き換わりました。指定を省くと既定値(上の例では `aes-cbc-essiv:sha256`、256ビット)が使われ、警告が出ます。別のマシンで開くときは同じ指定が要るので、`--cipher`、`--key-size`、`--hash` を明示します。

LUKS2 の機能 [331.3]

現在の cryptsetup が luksFormat で作る形式は、既定で LUKS2 です (LUKS1 のヘッダを持つデバイスも、引き続き開ける)。^[^c1-16]

項目	LUKS1	LUKS2
ヘッダの形式	固定のバイナリ	JSON のメタデータ。2つの写しを持ち、片方が壊れても復旧しやすい
鍵導出 (KDF)	PBKDF2	既定は argon2id (メモリを大量に使い、総当たりには強い)
鍵スロット	8個	最大32個 (鍵スロットの領域の大きさと鍵長しだい。番号は0〜31)
トークン	なし	外部の情報をヘッダに添えられる (Clevis が使う)
認証付き暗号	なし	--integrity で dm-integrity を使い、改ざんも検出する
再暗号化	停止して行う	cryptsetup reencrypt で、使いながら鍵や暗号を変更できる

```
$ cryptsetup luksFormat --type luks2 /dev/sdb1
$ cryptsetup luksDump /dev/sdb1
Version:                2
Keyslots:
  0: luks2
      PBKDF:             argon2id
$ cryptsetup convert --type luks2 /dev/sdb1      # LUKS1 のデバイスを
LUKS2 に変換する
```

Clevis など LUKS2 の機能に頼る道具を使うときは、LUKS2 で作ります。

eCryptfs:ホームディレクトリの暗号化 [331.3]

eCryptfs は、ファイルシステムの上に重ねるスタック型の暗号化です。下のディレクトリ(暗号化された実体)を上のマウントポイントに重ねて開くと、読み書きのたびにファイル単位で暗号化と復号が行われます。暗号の情報(メタデータ)は、ファイルの先頭に持ちます(`ecryptfs_xattr` を付けると拡張属性に置く)。パーティションを切らずに、ユーザーのホームの一部だけを守る点、dm-crypt との違いです。

```
$ ecryptfs-setup-private
$ ecryptfs-mount-private
$ ecryptfs-umount-private
```

`ecryptfs-setup-private` は、暗号の実体 `~/Private` (権限700) と開いた口 `~/Private` (権限500) を作ります。`[^c1-17]` マウント用パスフレーズ(乱数16バイト)は、ログインパスワードで包まれて `~/ecryptfs/wrapped-passphrase` に保

存されます。鍵の署名は `~/.ecryptfs/Private.sig` に入ります。マウント用パスフレーズは必ず控えます(失うと復元の手段がない)。

オプション	意味
<code>-w / --wrapping</code>	ログインパスワードとは別の、独立した包み用パスフレーズを使う
<code>-n / --no-fnek</code>	ファイル名を暗号化しない(既定は、対応環境でファイル名も暗号化)
<code>-b / --bootstrap</code>	新しいユーザーのホーム全体を暗号化する
<code>-u</code>	対象のユーザー
<code>--noautomount / --noautoumount</code>	ログイン時の自動マウント、ログアウト時の自動アンマウントをしない

root が直接マウントするときは、`mount -t ecryptfs 下 上` の形で、補助の `mount.ecryptfs` が動きます。

```
# mount -t ecryptfs /secret /secret \  
-o  
key=passphrase,ecryptfs_cipher=aes,ecryptfs_key_bytes=16,ecryptfs_passt  
hrough=no,ecryptfs_enable_filename_crypto=no
```

オプション	意味
<code>key=passphrase</code>	鍵の種類。パスフレーズ(<code>openssl</code> なら RSA 鍵ファイル)
<code>ecryptfs_sig</code>	内容の暗号に使う鍵の署名。事前にキーリングに入っている必要がある

オプション	意味
<code>ecryptfs_fnek_sig</code>	ファイル名の暗号に使う鍵の署名
<code>ecryptfs_cipher</code> / <code>ecryptfs_key_bytes</code>	ファイルごとの暗号と鍵の長さ(バイト)
<code>ecryptfs_passthrough</code>	暗号化されていないファイルも読み書きできるようにする。既定は無効
<code>ecryptfs_enable_filename_crypto</code>	ファイル名の暗号化。 <code>y</code> か <code>n</code>
<code>passphrase_passwd_file</code>	パスフレーズを書いたファイル。 <code>ps</code> に見える <code>passphrase_passwd=</code> より安全

アンマウントは `umount` で、補助の `umount.ecryptfs` が鍵をキーリングから外します。ログイン時に自動でマウントするには PAM を使います。`pam_ecryptfs.so` に `unwrap` を付けると、ログインパスワードで包みを解き、`~/Private` をマウントします(サービスは `auth` と `session`)。

```
/etc/pam.d/common-auth    auth    required    pam_ecryptfs.so unwrap
/etc/pam.d/common-session session optional    pam_ecryptfs.so unwrap
```

コマンド	役割
<code>ecryptfs-mount-private</code> / <code>ecryptfs-umount-private</code>	<code>~/Private</code> を開く・閉じる補助。必要ならログインパスワードを聞く
<code>ecryptfsd</code>	公開鍵方式のとき、カーネルからの鍵の要求を処理するユーザー空間のデーモン
<code>ecryptfs-add-passphrase</code>	パスフレーズをキーリングに入れ、署名を表示する

コマンド	役割
<code>ecryptfs-rewrap-passphrase /</code> <code>ecryptfs-unwrap-passphrase</code>	包みのパスワードを付け替える / 包まれたマウント用パスフレーズを表示する
<code>ecryptfs-recover-private</code>	暗号化の実体から、手動で中身を取り出す
<code>ecryptfs-migrate-home -u ユーザー</code>	既存のホームを暗号化ホームへ移行する
<code>ecryptfs-setup-swap</code>	スワップを暗号化する

cryptmount [331.3]

cryptmount は、管理者が設定した暗号化ファイルシステムを、一般ユーザーがパスワードだけで開閉できるようにする道具です (cryptsetup は root が使う)。^[^c1-18] device-mapper とループデバイスの用意を自動で行ってからマウントします。設定は `/etc/cryptmount/cmtab` に、ターゲット名と波括弧で書きます。

```
vault {  
    dev=/dev/sdb1  
    dir=/mnt/vault  
    fstype=ext4  
    flags=user  
    keyfile=/etc/cryptmount/vault.key  
}
```

欄	意味
<code>dev</code> (必須)	元のデバイスか、ファイル
<code>dir</code> (必須)	マウント先

欄	意味
<code>fstype</code> (必須)	ファイルシステムの種類
<code>keyfile</code> (必須)	鍵ファイル。中の鍵はパスワードで保護される
<code>flags</code>	<code>user</code> (一般ユーザーがマウントできる)、 <code>nouser</code> (root だけ)、 <code>fsck</code> (マウント前に検査)

コマンド	意味
<code>cryptmount vault (-m)</code>	マウント。パスワードを聞く
<code>cryptmount -u vault</code>	アンマウント。パスワードは要らない
<code>cryptmount -l</code>	設定済みのターゲットの一覧
<code>cryptmount -S vault</code>	マウント中かどうかの状態
<code>cryptmount -c vault</code>	パスワードの変更
<code>cryptmount -g サイズ vault</code>	新しいターゲットの鍵を作る(サイズはバイト数)
<code>cryptmount -p vault / -r vault</code>	マウントはせず、device-mapper とループデバイスだけを用意(<code>-p</code>)、解放(<code>-r</code>)。root 用
<code>cryptmount -s vault / -x vault</code>	暗号化スワップを有効(<code>-s</code>)、無効(<code>-x</code>)にする。root 用
<code>cryptmount -B / -Q</code>	<code>bootaction</code> を宣言したターゲットを、起動時に開く(<code>-B</code>)、終了時に閉じる(<code>-Q</code>)

Clevis と NBDE (Tang・TPM2) [331.3]

暗号化したディスクは、起動のたびにパスフレーズが要ります。サーバ群では、人が入力できません。かといって、鍵を同じディスクに置けば暗号化の意味がなくなります。Clevis は、条件を満たしたときだけ自動で復号する枠組みで、条件は「ピン」と呼ぶ部品で表します。[^c1-19]

ピン	条件
tang	Tang サーバにネットワークで到達できる
tpm2	同じ機械の TPM 2.0 チップ (必要なら特定の PCR の値) が使える
sss	複数のピンを組み合わせ、しきい値 t 個の成功で復号する (Shamir の秘密分散)

```
$ clevis luks bind -d /dev/sda2 tang
'{"url":"http://tang.example.com"}'

$ clevis luks bind -d /dev/sda2 tpm2 '{"pcr_ids":"7"}'

$ clevis luks bind -d /dev/sda2 sss '{"t":2,"pins":{"tpm2":
{"pcr_ids":"0"},"tang":{"url":"http://tang.example.com"}}}'

$ clevis luks list -d /dev/sda2

$ clevis luks unlock -d /dev/sda2

$ clevis luks unbind -d /dev/sda2 -s 1
```

コマンド	働き
<code>clevis luks bind</code>	新しい強い鍵を作り、LUKS の鍵スロットにパスフレーズとして足す。その鍵をピンで暗号化した結果 (JWE) を、LUKS のヘッダに保管する (LUKS2 ではトークン)。既存のパスフレーズは残るので、手入力でも開ける
<code>clevis luks list</code>	束縛されたピンの一覧
<code>clevis luks unlock</code>	束縛したピンでデバイスを開く
<code>clevis luks unbind -s スロット</code>	スロット番号を指定して束縛を外す
<code>clevis luks regen</code>	同じ設定で束縛を作り直す

起動時に自動で開けるには、`initramfs` へ `Clevis` を組み込みます (Red Hat 系は `dracut -f`、Debian 系は `update-initramfs -u`)。 `sss` の例の `t` が2なら `tpm2` と `tang` の両方が成功したときだけ復号され、1ならどちらか一方で足ります。

`Tang` は、ネットワークに結びつけて鍵を守るサーバです。鍵を預かるのではなく、復号に必要な計算を手伝う点が、鍵預託 (エスクロー) との違いです。この仕組みを NBDE (Network-Bound Disk Encryption) と呼びます。

比較	鍵預託 (エスクロー)	Tang
状態	鍵を保管する (状態を持つ)	状態を持たない
クライアントの鍵	サーバが全部知っている	サーバは一度も見ない
TLS・認証	必須	任意

最初の暗号化のとき、クライアントは `Tang` の公開鍵の一覧 (advertisement) を取得し、署名鍵のサムプリントで信頼を確かめます (SSH

の初回接続と同じ)。復号のときは、保存しておいたメタデータをもとに、HTTP の POST で復元用の値を受け取ります。Tang のサーバは `systemctl enable tangd.socket --now` で起動し、鍵は `/var/db/tang` に置かれます。Tang に到達できない場所(盗まれたサーバなど)では、自動では開きません。鍵を入れ替えるときは `tangd-rotate-keys` を使い、クライアント側は `clevis luks regen` で束縛を更新します。

DNS・ゾーン・リソースレコード [331.4]

DNS は名前の木で、管理の単位がゾーンです。ゾーンは木の一部を切り出したもので、権威サーバがそのゾーンの正式な答えを持ちます。親ゾーンは `NS` レコードで子ゾーンへ権限を渡し(委任)、子の名前サーバの名前が子ゾーン自身の中にあるときは、親に `A` レコードも置きます(グルー)。

型・役割	内容
権威サーバ	ゾーンのデータを持ち、そのゾーンの問い合わせに正式に答える
キャッシュサーバ(再帰リゾルバ)	クライアントの代わりに木をたどって答えを探し、結果を保存する
---	---
SOA	ゾーンの管理情報。レコードにはプライマリ名・管理者・serial・refresh・retry・expire・負のキャッシュの TTL の順に並ぶ(例 <code>example.test. 300 IN SOA</code> <code>ns1.example.test.</code> <code>admin.example.test. 2026101301 3600</code> <code>900 604800 300</code>)
NS	そのゾーンの権威サーバ
A / AAAA	名前に対する IPv4 / IPv6 アドレス

型・役割	内容
CNAME / MX	別名 / メールの宛先サーバ
TXT / PTR	文字列 (SPF や <code>_acme-challenge</code> の検証に使う) / アドレスから名前の逆引き

プライマリからセカンダリへゾーンを複製することをゾーン転送と呼び、全体は AXFR、差分は IXFR で送ります。SOA の serial が大きくなると、セカンダリが更新を取りに来ます。

```
$ dig @192.0.2.1 www.example.com A +short      # 答えだけを表示
$ dig example.com NS +trace                    # ルートから順にたどった経過を表示
```

DNSSEC の仕組み [331.4]

DNSSEC (RFC 4033～4035) は、DNS の応答が正しい持ち主から来て、途中で書き換えられていないことを、デジタル署名で確かめる仕組みです。[[^]c1-20] 偽の応答を信じ込ませるキャッシュポイズニングを防ぎます。守るのはデータの真正性と完全性で、内容の暗号化はしません。ゾーンの持ち主が秘密鍵で署名し、公開鍵を DNS で配ります。

RR	置く場所	中身
DNSKEY	そのゾーン	ゾーンの公開鍵。flags が 256なら ZSK、257なら KSK

RR	置く場所	中身
RRSIG	そのゾーン	RRset(同じ名前と型の RR の組) への署名。署名の期限 (inception と expiration)、鍵番号 (key tag) を持つ
DS (Delegation Signer)	親ゾーン	子ゾーンの KSK のハッシュ。親から子への信頼の受け渡しに使う
NSEC	そのゾーン	名前の並びで次の名前と、その名前にある型の一覧。存在しないことの証明に使う
NSEC3	そのゾーン	NSEC の名前をハッシュにしたもの
NSEC3PARAM	そのゾーン	NSEC3 の作り方 (ハッシュの方式・フラグ・反復回数・ソルト)

```
example.test. 300 IN DNSKEY 257 3 13 ( utmQJ0t3IHjb5kus... ) ; KSK;
key id = 58334
www.example.test. 300 IN RRSIG A 13 3 300 20261112092521 20261013092521
6682 example.test. CGu2...
example.test. IN DS 58334 13 2
E074CF0C781AC3BF8726A903465E13051073F433139F4C8C7E3A7CFEC586706B
example.test. 0 IN NSEC3PARAM 1 0 10 ABCD1234
```

欄	例	読み方
DNSKEY	257 3 13	flags (256 は ZSK、257 は KSK)、プロトコル (常に3)、アルゴリズム
アルゴリズム番号	8、13、14、15	RSASHA256、 ECDSAP256SHA256、 ECDSAP384SHA384、 Ed25519
DS	58334 13 2 E074...	KSK の key tag、アルゴリズム、ダイジェストの種類 (1は SHA-1、2は SHA-256、4は SHA-384)、ダイジェスト
RRSIG	A 13 3 300 ... 6682 example.test.	署名した型、アルゴリズム、ラベル数、元の TTL、期限、開始、署名した鍵の key tag、署名者
NSEC3PARAM	1 0 10 ABCD1234	ハッシュの方式 (1は SHA-1)、フラグ、反復回数、ソルト

鍵	署名する対象	性質
ZSK (Zone Signing Key)	ゾーンの RRset すべて	署名の本数が多いので短く、頻繁に交換する。親への連絡は要らない
KSK (Key Signing Key)	DNSKEY の RRset	信頼の起点。DS として親に登録するので、交換には親との調整が必要

分ける理由は、頻繁に変える鍵 (ZSK) の交換で、親への登録変更を起こさないためです。1本の鍵で両方を兼ねる方式 (CSK) もあります (BIND の `dnssec-policy default` はこの CSK)。

信頼の連鎖は、ルートから順に確かめます。

- 1. トラストアンカー (ルートの KSK) で、ルートの DNSKEY を確かめる
- 2. ルートが署名した jp の DS で、jp の KSK を確かめる
- 3. jp が署名した example.jp の DS で、example.jp の KSK を確かめる
- 4. example.jp の KSK が DNSKEY の組に、ZSK が各 RRset の RRSIG に署名していることを確かめる

DS と DNSKEY の不一致、署名の不一致、署名の期限切れのどれかがあると、リゾルバは応答を bogus (不正) とみなして SERVFAIL を返します。

存在しない名前 (NXDOMAIN) の答えにも、署名が要ります。署名できる RR がないので、「この名前とこの名前の間には何も無い」ことを NSEC か NSEC3 で示します。

比較	NSEC	NSEC3
名前	平文	ハッシュ (ソルトと反復回数つき)
列挙	次の名前を順にたどれて、ゾーン内の全名前を列挙できる (ゾーンウォーク)	しにくい (辞書攻撃は可能)
付随する RR	なし	NSEC3PARAM が必要
opt-out	なし	あり。署名のない委任を NSEC3 の範囲から外し、巨大ゾーンの負荷を下げる

NSEC3 のパラメータには、RFC 9276 の指針があります。反復回数は0、ソルトは使わないのを勧めています (NSEC3PARAM の表示では 1 0 0 -)。[[^]c1-21]

フラグ	意味
DO	クライアントが DNSSEC の RR (RRSIG など) を求める。EDNS0 の印で、 <code>dig +dnssec</code> が立てる。署名付きの応答は大きく、収まらなければ TCP に切り替わる
AD	リゾルバが検証に成功した (Authenticated Data)
CD	クライアントが検証をしないよう求める (Checking Disabled)。 <code>dig +cd</code>

BIND を権威サーバにして署名済みゾーンを出す [331.4]

```
# example.test を ECDSAP256SHA256 で署名する手作業の流れ
$ dnssec-keygen -a ECDSAP256SHA256 -n ZONE -f KSK -K keys example.test
Kexample.test.+013+58334

$ dnssec-keygen -a ECDSAP256SHA256 -n ZONE -K keys example.test      #
ZSK
Kexample.test.+013+06682

$ dnssec-signzone -K keys -o example.test -S -3 abcd1234 -f
example.test.zone.signed example.test.zone

Zone fully signed:

Algorithm: ECDSAP256SHA256: KSKs: 1 active, 0 stand-by, 0 revoked
                                ZSKs: 1 active, 0 stand-by, 0 revoked

$ dnssec-dsfromkey -2 keys/Kexample.test.+013+58334.key
example.test. IN DS 58334 13 2

E074CF0C781AC3BF8726A903465E13051073F433139F4C8C7E3A7CFEC586706B
```

コマンドとオプション	意味
<code>dnssec-keygen</code>	鍵の組を作り、K<ゾーン>+<アルゴリズム △番号3桁>+<key tag>.key (公開鍵)と .private (秘密鍵)を出す
<code>-a / -b / -n ZONE / -K</code>	方式 / RSA の鍵長 / 用途 / 出力先
<code>-f KSK</code>	KSK を作る (付けなければ ZSK)
<code>dnssec-signzone</code>	ゾーンに署名して <code>.signed</code> を作る。有効 期間は既定で30日。親へ渡す DS の情報が dsset-ゾーン名. に出る
<code>-o / -S / -f</code>	ゾーンの起点 / スマート署名 (鍵ファイルを 自動で探す) / 出力ファイル
<code>-3 ソルト / -H / -A</code>	NSEC3 を使う (省くと NSEC) / NSEC3 の反 復回数 / opt-out
<code>-N / -x / -i</code>	SOA の serial の扱い (<code>increment</code> や <code>unixtime</code>) / DNSKEY の署名を KSK だけ にする / 更新間隔 (既定は有効期間の4分 の1)
<code>dnssec-dsfromkey -2</code>	KSK から SHA-256 の DS を作る

```
# named.conf : 署名済みのファイルを読み込む
# (options に version "none"; を書くと version.bind の応答を隠せる)
zone "example.test" {
    type master;
    file "example.test.zone.signed";
};
```

`rndc reload example.test` で反映し、`dnssec-dsfromkey` が出した DS を親ゾーン(レジストラ)へ登録します。DS が親にないと、検証するリゾルバは「署名はあるが親の裏付けがない」ゾーンを、署名のないゾーンと同じ扱いにします。手作業の署名は、RRSIG の期限が切れる前に再署名しないと、応答が検証を通りません。

BIND 9.16 以降は、鍵の生成・署名・更新・ロールオーバーを BIND が自動で行えるので、ゾーンに `dnssec-policy` を書きます。[^{^c1-22}]

```
zone "auto.test" {
    type master;
    file "dyn/auto.test.zone";
    dnssec-policy default;
    inline-signing yes;
};
```

項目	内容
<code>inline-signing yes</code>	元のゾーンファイルは書き換えず、署名したコピーを別に持つ(9.16 では明示する)
<code>default</code> の方針	ECDSAP256SHA256 の CSK 1本
鍵の置き場	<code>key-directory</code> (既定は <code>named</code> の作業ディレクトリ = <code>options</code> の <code>directory</code>) に自動で作られる

項目	内容
独自の方針	<code>dnssec-policy "名前" { keys { ksk lifetime unlimited algorithm ...; zsk lifetime 30d algorithm ...; }; signatures-validity 14d; };</code> で定義する。 <code>lifetime</code> は鍵の寿命で、過ぎると自動で交換される(<code>unlimited</code> は交換しない)

以前の署名の自動化 `auto-dnssec maintain;` は、`dnssec-policy` に置き換わりました(9.18 で非推奨、9.20 で削除)。`update-policy local;` は動的更新を許す別の設定で、置き換えの対象ではありません。

BIND を遠隔操作するのが `rndc` で、接続の認証には共有鍵(`/etc/rndc.key` か `rndc.conf`)を使います。`rndc-confgen -a` で鍵を作り、`named.conf` の `controls` で待ち受け先と鍵を指定します。

rndc のサブコマンド	働き
<code>rndc reload [ゾーン]</code>	設定とゾーンの再読み込み
<code>rndc dnssec -status ゾーン</code>	<code>dnssec-policy</code> を使うゾーンの鍵と、ロールオーバーの状態
<code>rndc loadkeys ゾーン</code>	鍵ディレクトリの鍵を読み直す
<code>rndc sign ゾーン</code>	鍵を読み直して署名し直す
<code>rndc signing -list ゾーン</code>	署名の進み具合。 <code>dnssec-policy</code> を使うゾーンでは使えず、 <code>rndc dnssec</code> を勧められる
<code>rndc flush</code>	キャッシュを消す(検証の確認の前に使う)

確認には、`dnssec-verify -o example.test example.test.zone.signed` (署名)、`named-checkzone` (構文)、`dig +dnssec` (RRSIGの有無)を使います。

```
$ dig @127.0.0.1 www.example.test A +dnssec +noall +answer
www.example.test. 300      IN      A       192.0.2.10
www.example.test. 300      IN      RRSIG   A 13 3 300 20261112092521
20261013092521 6682 example.test. ...
$ dig @127.0.0.1 nope.example.test A +dnssec +noall +authority #
SOA、その RRSIG、NSEC3 と RRSIG が最大3組
```

存在しない名前への答えは、SOA と RRSIG のほかに NSEC3 が最大3組返ります(その名前を含む範囲、最も近い親にあたる名前、ワイルドカードの不在を示す。1つの NSEC3 が2つの役を兼ねると2組に減る)。

鍵の管理：ロールオーバー・再署名 [331.4]

鍵は定期的に交換します(ロールオーバー)。鍵のメタデータ(いつ公開し、使い始め、使い終え、消すか)は、`dnssec-settime` で見て変えます。

オプション	日付の意味
-P	Publish:DNSKEY としてゾーンに載せ始める日
-A	Activate:署名に使い始める日
-R	Revoke:失効の印(REVOKE ビット)を付ける日
-I	Inactive:署名に使うのをやめる日
-D	Delete:DNSKEY をゾーンから消す日

```
$ dnssec-settime -I +90d -D +120d Kexample.test.+013+06682.key # 古い  
ZSK : 90日後に非アクティブ、120日後に削除  
  
$ dnssec-keygen -S Kexample.test.+013+06682.key -i 30d # 後  
継の鍵。古い鍵の無効化の30日前に公開  
  
$ dnssec-settime -p all Kexample.test.+013+06682.key # 日  
付の確認
```

日付は絶対指定 (20261101) か、 +30d のような相対指定です (後継の鍵を作ると、古い鍵の .private に Successor、新しい鍵に Predecessor が残る)。

ZSK の交換は、事前公開 (pre-publish) 方式が基本です。

1. 新しい ZSK を作り、DNSKEY として公開する (-P)。まだ署名には使わない
2. DNSKEY の TTL が過ぎて、各地のキャッシュに新しい鍵が行き渡るまで待つ
3. 新しい ZSK を有効にして (-A) 署名し直し、古い ZSK を非アクティブにする (-I)
4. 古い鍵で署名した RRSIG が、キャッシュから消える時間 (最大の TTL) を待つ
5. 古い ZSK を DNSKEY から消す (-D)

もう一つの二重署名 (double-signature) 方式は、交換の間だけ新旧の鍵の両方で署名します。RRSIG が2倍になり、応答が大きくなります。

KSK の交換には、親ゾーンの DS が絡みます。

1. 新しい KSK を作り、DNSKEY に載せる

- 2. 新しい KSK の DS を、親ゾーンに登録する
- 3. 親の DS の TTL が過ぎて、古い DS がキャッシュから消えるのを待つ
- 4. 古い DS を親から消す
- 5. 古い KSK を DNSKEY から消す

順序を誤ると、検証が通らなくなります。たとえば、古い KSK を先に消してから親の DS を入れ替えたとします。リゾルバの持つ DS は古い KSK を指したままなので、DNSKEY の署名と合わず、ゾーン全体が bogus になります (待ち時間の求め方は RFC 7583)。

署名の再実行 (再署名) も運用の一部です。RRSIG は有効期限を持ち、期限が切れる前に作り直さないと、リゾルバは全応答を bogus と判断します。`dnssec-signzone` は、期限が近い署名だけを作り直します。`cron` で定期的に行い、ゾーンの内容を変えたときも再署名して、`serial` を上げてから `rndc reload` します (`dnssec-policy` なら BIND が行う)。

DNSSEC を検証するキャッシュサーバ [331.4]

検証をするのは、再帰リゾルバとして動く `named` で、設定は `dnssec-validation` で決めます。[^c1-23]

値	動き
<code>auto</code>	検証を有効にし、BIND に組み込みのルートのトラストアンカーを使う。アンカーは自動で更新される。BIND 9 の既定
<code>yes</code>	検証を有効にする。 <code>trust-anchors</code> でアンカーを手で書く必要がある
<code>no</code>	検証しない

```
options {  
    recursion yes;  
    allow-recursion { 127.0.0.1; };  
    dnssec-validation auto;  
};  
trust-anchors {  
    example.test. static-ds 58334 13 2 "E074CF0C781AC3BF...";  
};
```

trust-anchors の型	意味
static-key / static-ds	固定の鍵 / 固定の DS
initial-key / initial-ds	最初の鍵 / DS。以後は自動で更新される

古い `managed-keys` や `trusted-keys` は、これに置き換わりました。組織内だけの署名ゾーンを検証させたいときは、そのゾーンの KSK か DS を書き足します。

検証できた応答には、AD ビットが付きます。署名を壊したときの反応を見ます。

```
$ dig @127.0.0.1 -p 5300 www.example.test A
;; flags: qr rd ra ad; ...                                ← 検証できた
(権威側の A レコードだけを書き換え、署名はそのままにして、キャッシュを
消す)

$ dig @127.0.0.1 -p 5300 www.example.test A
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL

$ dig @127.0.0.1 -p 5300 www.example.test A +cd
www.example.test. 300      IN      A      192.0.2.99      ← 検証を
止めると答えが返る
```

状況	結果
署名が合わない、RRSIG の期限切れ	SERVFAIL。+cd なら答えが返る
DS と子の KSK が一致しない	SERVFAIL (bogus)
サーバの時計が大ききずれている	署名の期間に入らず SERVFAIL
親に DS がない(署名なしのゾーン)	検証されずに答えが通る。AD は付かない
DS があるのに子が署名していない	SERVFAIL (bogus)

SERVFAIL だけでは、検証の失敗か、サーバの障害か、見分けが付きません。
+cd を付けて検証を止めた状態で答えが返れば、DNSSEC の検証だけが落ちています。

delv は、named と同じ検証の仕組みで問い合わせと検証を行う道具です。
dig +dnssec は応答を見せるだけですが、delv は検証の結果まで出し、成功すると答えと RRSIG の前に ; fully validated と表示します。書き方は delv @サーバ 名前 型 で、dig と同じ並びです。自前の練習用ゾーンでは、アンカー

のファイルを `-a` で渡し、`+root=` でアンカーを置く名前を指定します (BIND 9.16 の実機)。

```
$ delv @127.0.0.1 -a anchor.conf +root=example.test www.example.test A
; fully validated
www.example.test. 300      IN      A      192.0.2.80
www.example.test. 300      IN      RRSIG   A 13 3 300 20261113011334
20261014011334 61955 example.test. PT175i...
```

CAA と DANE: 証明書の情報を DNS に載せる [331.4]

CAA (Certification Authority Authorization、RFC 8659) は、証明書を発行してよい CA を DNS に書いておく仕組みです。[^c1-24] CA は証明書を発行する前に CAA を調べ、自分が許されていないと発行しません。

```
example.test. IN CAA 0 issue "ca.example.net"
example.test. IN CAA 0 issuewild ";"
example.test. IN CAA 0 iodef "mailto:sec@example.test"
```

要素	意味
flags	0。128にすると、理解できないタグを持つ CA は発行してはならない (critical)
タグ <code>issue</code>	通常の証明書を発行してよい CA のドメイン。値が <code>;</code> だけなら、どの CA にも発行させない
タグ <code>issuewild</code>	ワイルドカード証明書の発行を許す CA。あれば、ワイルドカードにはこちらが優先される。なければ <code>issue</code> が適用される

要素	意味
タグ <code>iodef</code>	違反の疑いがあるときの通知先 (<code>mailto:</code> か <code>https:</code>)

CAA を見るのは発行する側の CA で、ブラウザなどのクライアントは見ません。CAA がなければ、どの CA も発行できます。CA は、名前を左へ削って木を上へたどり、最初に見つかった CAA を使います (`www.example.test` になれば `example.test`)。

DANE (DNS-based Authentication of Named Entities、RFC 6698) は、サーバが使う証明書や鍵を DNS の `TLSA` レコードに載せる仕組みです。クライアントは、TLS 接続で受け取った証明書を `TLSA` の値と照合します。DNSSEC で検証された `TLSA` だけを信用します。ブラウザではほとんど使われず、メールサーバ間の SMTP で使われています。

```
_443._tcp.www.example.test. IN TLSA 3 1 1
2da03282e7ffc6637912e27b9bbf60136538f360e2ba477a6b27ed109bd98d5d
```

名前の形は `_ポート._プロトコル.ホスト名` です (メールなら `_25._tcp.mail.example.test`)。 `[^c1-25]` 値は3つの数と、照合する値の4つの欄からなります。

欄	値	意味
Certificate Usage	0	PKIX-TA:公開 CA の証明書で、連鎖の中の CA を指定する (CA constraint)

欄	値	意味
	1	PKIX-EE:公開 CA が発行した、そのサーバの証明書を指定する (service certificate constraint)
	2	DANE-TA:信頼の起点にする CA を指定する (trust anchor assertion)。公開 CA でなくてもよい
	3	DANE-EE:サーバの証明書そのものを指定する (domain-issued certificate)。公開 CA による検証は要らない
Selector	0	証明書全体
	1	公開鍵の部分 (SubjectPublicKeyInfo)
Matching Type	0	そのまま比較
	1	SHA-256 のハッシュ
	2	SHA-512 のハッシュ

実際の運用では 3 1 1 が最も多く使われます (証明書を更新しても公開鍵が同じなら、TLSA を書き換えずに済む)。

値は OpenSSL で作る

```
$ openssl x509 -in web.crt -noout -pubkey | openssl pkey -pubin -  
outform DER | openssl dgst -sha256
```

```
2da03282e7ffc6637912e27b9bbf60136538f360e2ba477a6b27ed109bd98d5d ←
```

3 1 1 の値

```
$ openssl x509 -in web.crt -outform DER | openssl dgst -sha256
```

```
0c29ad43e091d451a8e679fa4fcd7a1e72b1ebaaaea3a46bc2b3383dcc04f4a7 ←
```

3 0 1 の値

CAA は発行時に CA が確かめる制約、DANE は接続時にクライアントが確かめる照合です(どちらも DNSSEC があると改ざんに強い)。

TSIG [331.4]

TSIG (Transaction SIGnature、RFC 8945) は、共有秘密鍵と HMAC で DNS メッセージに署名し、通信の相手をつまかめる仕組みです。[^c1-26] ゾーン転送、動的更新、通知の認証に使います。DNSSEC がデータの出所を公開鍵で保証するのに対し、TSIG は通信の相手をつま共有鍵で認証します。

```

$ tsig-keygen xfer-key
key "xfer-key" {
    algorithm hmac-sha256;
    secret "EK+aYBIqZDN6YnhTQRlpNaY4spN+UECWFMQ3rfr9QQs=";
};

zone "example.test" { type master; file "example.test.zone"; allow-
transfer { key xfer-key; }; }; # プライマリ
server 192.0.2.1 { keys xfer-key; };
# セカンダリ（プライマリを指す）
$ dig @127.0.0.1 example.test AXFR
; Transfer failed. ← status: REFUSED
$ dig @127.0.0.1 example.test AXFR -y hmac-sha256:xfer-
key:EK+aYBIqZDN6YnhTQRlpNaY4spN+UECWFMQ3rfr9QQs= # -y アルゴリズム:
鍵名:秘密
(ゾーン全体が返る)

```

出力の `key` 文は、プライマリとセカンダリの両方の `named.conf` に入れ、鍵なしの転送は `REFUSED` で拒否されます。TSIG の署名には時刻が入り、サーバ間の時計が許容範囲（既定は300秒）を超えてずれると、検証が通りません。IP アドレスでの制限（`allow-transfer { 192.0.2.2; };`）より、鍵による認証のほうが、アドレスの偽装に強くなります。

DNS over TLS・DNS over HTTPS・マルチキャスト DNS [331.4]

DNS は昔、平文の UDP でやり取りされました。通信の盗み見を防ぐために、クライアントとリゾルバの間を暗号化する方式が2つあります。

方式	規格	仕組み
DNS over TLS (DoT)	RFC 7858	DNS を TLS で包む。TCP 853番
DNS over HTTPS (DoH)	RFC 8484	DNS のメッセージを HTTPS (443番) で運ぶ。 application/dns-message

どちらも、クライアントとリゾルバの間の経路を暗号化するだけです。DNSSECのように、データそのものの真正性は保証しません。DoH は一般の HTTPS と同じ443番を通るため、DoT のようにポートでの識別や制御はできません。

マルチキャスト DNS (mDNS) は、DNS サーバのない同じリンク内 (LAN) で、名前を解決する仕組みです。問い合わせはマルチキャストで送られ、該当する機器が自分で答えます。

項目	内容
規格	RFC 6762
ポートとアドレス	UDP 5353番。IPv4 は 224.0.0.251、IPv6 は ff02::fb
名前	.local で終わる (例 printer.local)
範囲	同じリンク内。ルーターを越えない
実装	Avahi, systemd-resolved

サンプルはここまでです

続き(第2章～巻末)は、LPIC-3 303 問題集のご購入で、頻出度別ノート・暗記用ノートと合わせて3点すべてダウンロードできます。

SAMPLE

LPIC-3 303試験 学習用テキスト(無料サンプル)

版

2026年10月版(LPI公式 LPIC-3 Exam 303 Objectives Version 3.0(303-300・2026-10-13 取得)準拠)

発行

IT資格道場(<https://www.shikaku-doujou.com>)

お問い合わせ

info@shikaku-doujou.com

LPI および LPIC は Linux Professional Institute Inc. の商標または登録商標です。Linux は Linus Torvalds 氏の米国およびその他の国における商標または登録商標です。記載されているその他の会社名・製品名・サービス名は、各社の商標または登録商標です。

本教材は公式シラバスの内容を基に作成したもので、Linux Professional Institute との提携・承認関係はありません。

本文・図表・設問はすべてオリジナルの著作物です。無断での転載・複製・再配布・二次販売はできません。お気づきの点はinfo@shikaku-doujou.com までご連絡ください。