

Hybrid Public Key Encryption (HPKE)

セキュリティエンジニアリング特論 第10回

栗原 淳

兵庫県立大学大学院

2021-12-09

はじめに

この資料は

2019年にIETFにおいてドラフト提案、2021年現在において標準化が進んでいる、**Hybrid Public Key Encryption (HPKE)**という規格について、栗原自身の理解を深めるとともに、技術者にざっくり解説することを目的とする。

HPKE の IETF 標準化状況 (2021-10-01 時点で draft 12)

- IETF: <https://datatracker.ietf.org/doc/draft-irtf-cfrg-hpke/>
- GitHub: <https://github.com/cfrg/draft-irtf-cfrg-hpke>

本講義資料では、ある程度の前提知識を要求する。栗原の別の講義資料¹を参照しておくと良いだろう。

¹https://github.com/junkurihara/lecture-security_engineering の 2~6 回目あたり。

HPKEをざっくり言うと

Hybrid Public Key Encryption という「規格」

古典的な認証機能付きハイブリッド暗号化のフォーマット規定という位置付け、と思って良い。サポート性、再利用性、将来的な拡張性を高める目的で策定されている。

- 任意長のデータを、受信者の公開鍵²使って暗号化
- 将来的に耐量子公開鍵暗号をサポートすることも想定

²現状では楕円曲線暗号の公開鍵のみ (i.e., ECDH)

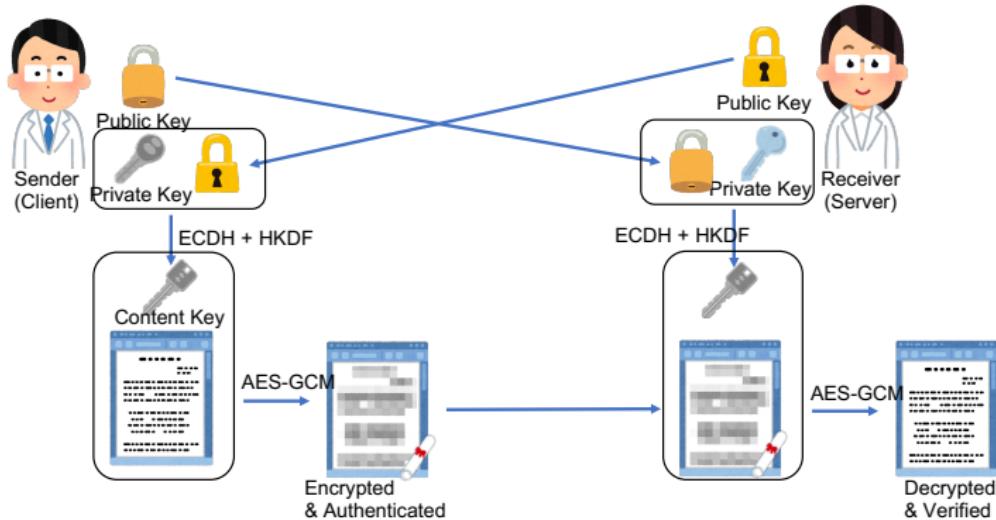


Figure: ECDH+HKDF+AES-GCMによるハイブリッド暗号化イメージ
 上図のようなハイブリッド暗号化のプロトコル、鍵フォーマット、暗号化データのフォーマットなどを一定に定めて、HPKEを策定。
 ⇒今までバラバラだったハイブリッド暗号化プロトコルが統一され、**複数環境での相互接続性が担保**される。³

³例: 言語によって異なるライブラリを使っていても、データは同様に扱えるように。JavaScript (WebCrypto) と C (OpenSSL) で統一されたハイブリッド暗号化ライブラリが用意されたり (すると良いけど JS 向けは今の所自分で作るしかない)。

HPKE の応用事例

HPKE はまだその規格は IETF RFC のドラフトであるが、既に別の IETF RFC ドラフトや商用サービス、Open Source Software (OSS) で利用されている。

例:

- RFC ドラフト: TLS Encrypted Client Hello (ECH)⁴⁵
- RFC ドラフト: Messaging Layer Security (MLS)⁶
- RFC ドラフト: Oblivious DNS over HTTS (ODoH)⁷
- Apple iCloud+ Private Relay (ODoH の商用利用)
- OSS: DNSCrypt⁸

HPKE を説明した後にもう少し詳しく述べる。

⁴<https://datatracker.ietf.org/doc/draft-ietf-tls-esni/>

⁵ESNI (Encrypted Server Name Indication) とも呼ばれていた

⁶<https://github.com/mlswg>

⁷<https://datatracker.ietf.org/doc/draft-pauly-dprivate-oblivious-doh/>

⁸<https://dnscrypt.info>

HPKE 概略 (Draft 12 ベース)

HPKEのビルディングブロック

HPKE は、(KEM, KDF, AEAD) の triplet でその構成を表す。⁹

KEM (Key Encapsulation Mechanism)

公開鍵暗号を使って、固定長の shared secret から固定長カプセル化情報を作成、あるいはカプセル化情報から shared secret を復号するメカニズム。後述する「モード」ごとに動作にバリエーションがある。

KDF (Key Derivation Function)

KEM で生成あるいは復号した shared secret を伸長・攪拌したりして、AEAD で利用する擬似ランダム共通鍵を生成するメカニズム。

AEAD (Authenticated Encryption with Associated Data)¹⁰

KDF で生成した鍵、および nonce、aad (Associated Data) を入力とした、認証タグ付き共通鍵暗号化。AES-GCM 等が使われる。

⁹ex. (DHKEM(X25519, HKDF-SHA256), HKDF-SHA256, AES-128-GCM)、DHKEM は shared secret 生成で KDF を使う。“DH”KEM と言いかながら ECDH による KEM なことに注意。

¹⁰RFC5116 <https://datatracker.ietf.org/doc/html/rfc5116>

HPKE の「モード」

HPKE の利用モードとして、"Base", "PSK (Pre-shared Key)", "Auth", と PSK+Auth な "AuthPSK" の 4 モードが準備されている。

Base Mode

基本モード。AEAD での改竄検知のみで、送信者の認証はない。

PSK (Pre-Shared Key) Mode

送受信者で同じ PSK を共有しているかどうかを認証。

Auth (Authentication) Mode

送信者が KEM の Private Key を有していることを認証。¹¹

AuthPSK Mode

PSK と Auth のメカニズムを両方利用。

¹¹ Signcryption の一種と考えられる。

各モード共通の基本フロー

送信者・受信者共々、各モード共通の基本フローは次の通り。

- 1 “Encryption Context” オブジェクトを生成。Auth, PSK, AuthPSK のときは生成時の引数が増える。KEM と KDF を実行。
- 2 “Encryption Context” に格納される共通鍵で、暗号化もしくは復号。つまり AEAD を実行。

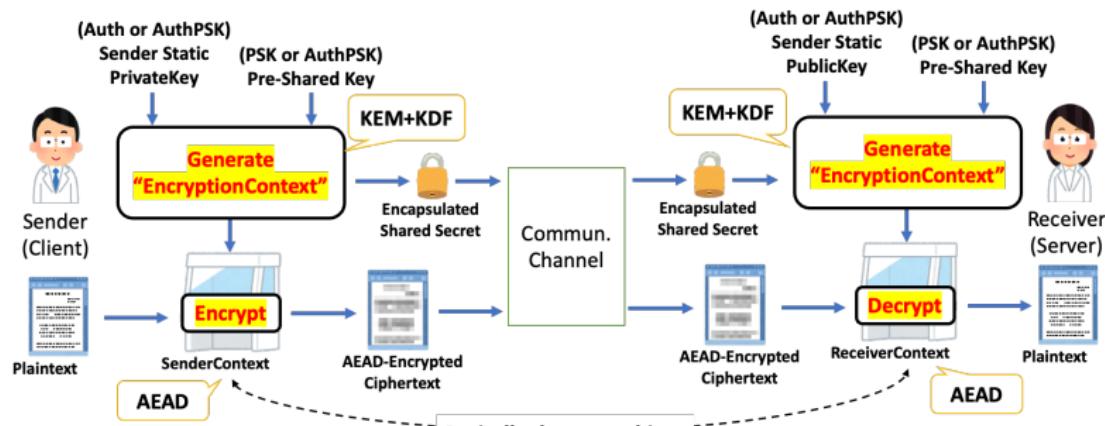


Figure: HPKE のモード共通での基本フロー

モードの動作の違いは“Encryption Context”の生成方法の違いと考えられる。このため、まずは各モードにおける“Encryption Context”的生成について解説する。

HPKE Base モードの Context 生成

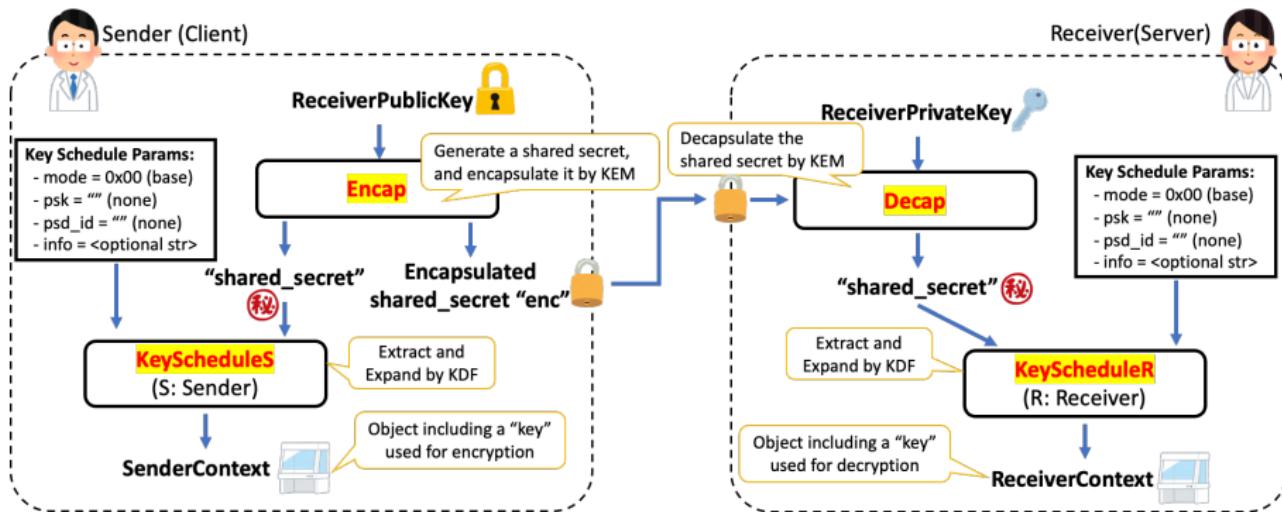


Figure: HPKE Base モードの Context 生成

Context 生成は、Encap/Decap の KEM 部分と、KeyScheduleS/R の KDF による鍵スケジュール部分で構成される。

HPKE PSK モードの Context 生成

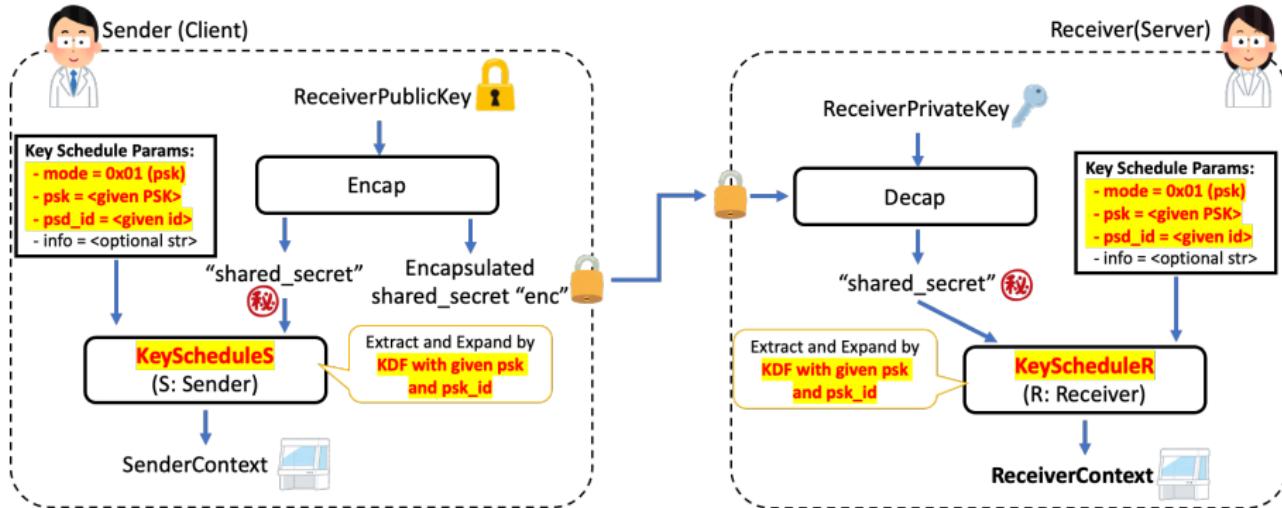


Figure: HPKE PSK モードの Context 生成

Base モードと比較し、KeySchedule において、PSK を使って shared_secret を攪拌し、Context 内の鍵を生成する部分が異なる。
⇒ 送受信で PSK が異なれば、同じ Context が生成できない。

HPKE Auth モードの Context 生成

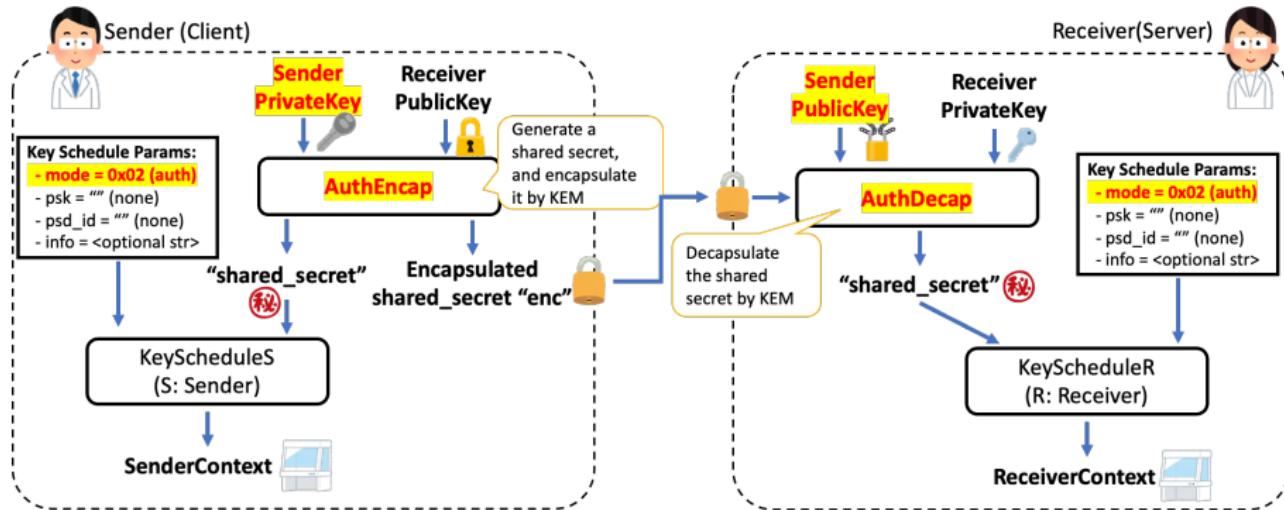


Figure: HPKE Auth モードの Context 生成

Base モードと比較し、Encap/Decap が、送信者の秘密鍵・公開鍵を追加引数とする AuthEncap/AuthDecap に置き換わる部分が異なる。
⇒ 送受信で送信者の鍵ペアが正しい対でなければ、同じ Context が生成できない

Encap/Decap の中身と AuthEncap/AuthDecap との違い

HPKE の現仕様では、ECDH+HDKF のみが KEM として考慮されているので、それに則ってイメージを解説する。¹² 受信者の (公開鍵, 秘密鍵) の鍵ペアを (PK_R, SK_R) 、送信者が作成する Ephemeral 鍵ペアを (PK_E, SK_E) とする。このとき、Base モードで `shared_secret` は以下のように生成される。

Encap: $\text{shared_secret} = \text{HKDF}(\text{ECDH}(SK_E, PK_R)),$

Decap: $\text{shared_secret} = \text{HKDF}(\text{ECDH}(SK_R, PK_E)).$

また、“enc”は Ephemeral 公開鍵 PK_E となる。

一方 Auth モードでは、追加で送信者の鍵ペア (PK_S, SK_S) を用いて、

AuthEncap: $\text{shared_secret} = \text{HKDF}(\text{ECDH}(SK_E, PK_R) || \text{ECDH}(SK_S, PK_R)),$

AuthDecap: $\text{shared_secret} = \text{HKDF}(\text{ECDH}(SK_R, PK_E) || \text{ECDH}(SK_R, PK_S)).$

すなわち、Ephemeral 鍵ペアと受信者の鍵ペアの ECDH、送信者の鍵ペアと受信者の鍵ペアの ECDH、の結果を結合して HKDF を通したもの `shared_secret` としている。よって、受信者があらかじめ受け取った PK_S に対して、正しく対となる SK_S を持っていないと同一の `shared_secret` が生成できない。

¹² 細かなパラメータは省略しているので注意。

HPKE AuthPSK モードの Context 生成

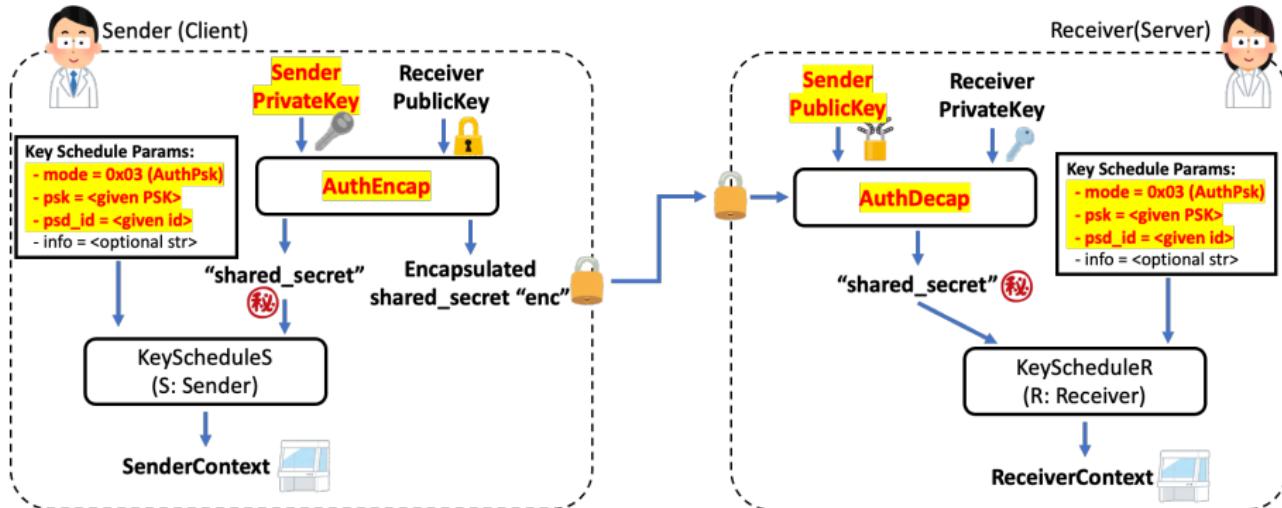


Figure: HPKE AuthPSK モードの Context 生成

Auth モードと PSK モードの単純な組み合わせとなる。

データ自体の認証付暗号化・復号: HPKE AEAD

送受信者で Context を共有した後は、AEAD で暗号化・復号。

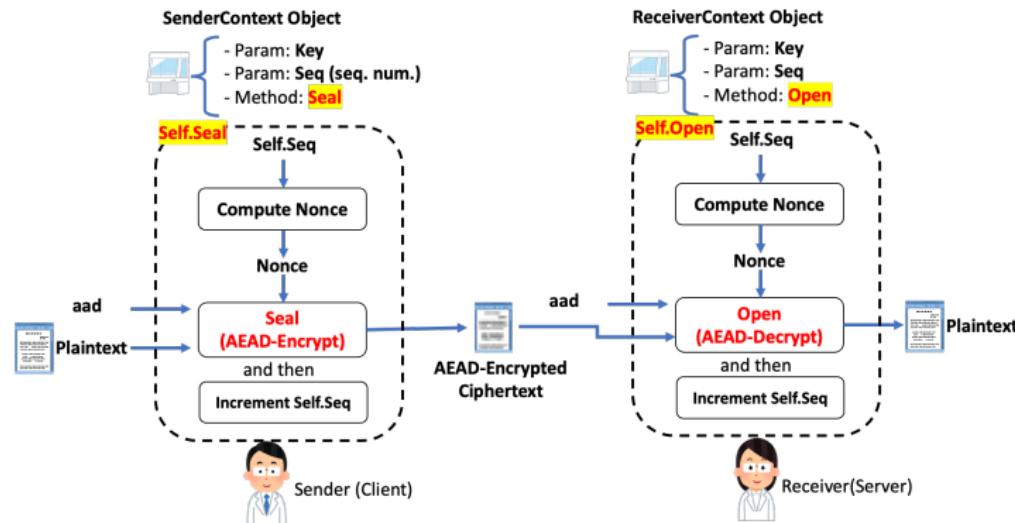


Figure: HPKE AEAD による暗号化・復号

上図の通り、Context オブジェクト内部で、インスタンス変数 Seq をインクリメントしながら、Seq を元にした Nonce を生成する。Nonce と Key を用いて、与えられた aad および plaintext/ciphertext を合わせて暗号化・復号する。

HPKE の Context 生成における KEM: DHKEM の場合

前述したが、現行のドラフトでは、KEM として DHKEM のみを規定している。¹³ ここでは、もう少し詳細に図解する。

¹³ 将来的には異なるアルゴリズムも採用できるようになっている。

Encap/Decap の構造を以下に示す。shared_secret を生成するにあたり、Ephemeral & Receiver 公開鍵をある意味「ラベル」として用いて、ECDH で導出した bits と合わせて KDF を実行している。

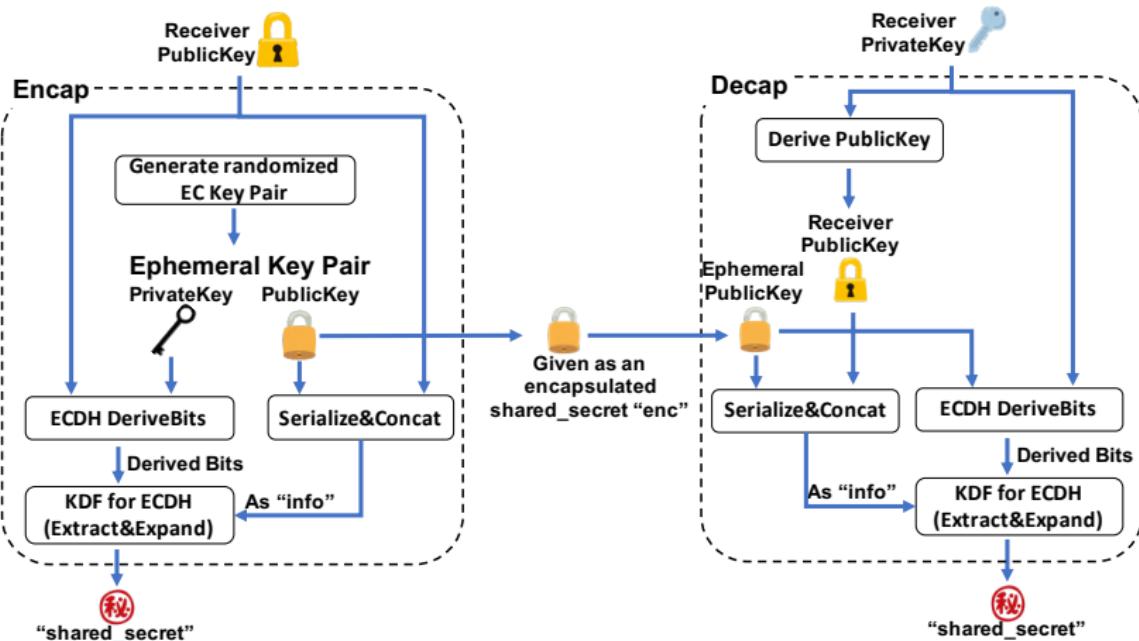


Figure: DHKEM Encap/Decap の構成

AuthEncap/AuthDecap の構造を以下に示す。Encap/Decap に対して、「ラベル」生成のために Sender 公開鍵を利用しつつ、ECDH 導出 bits を追加して KDF への入力を増やしている点が異なる。

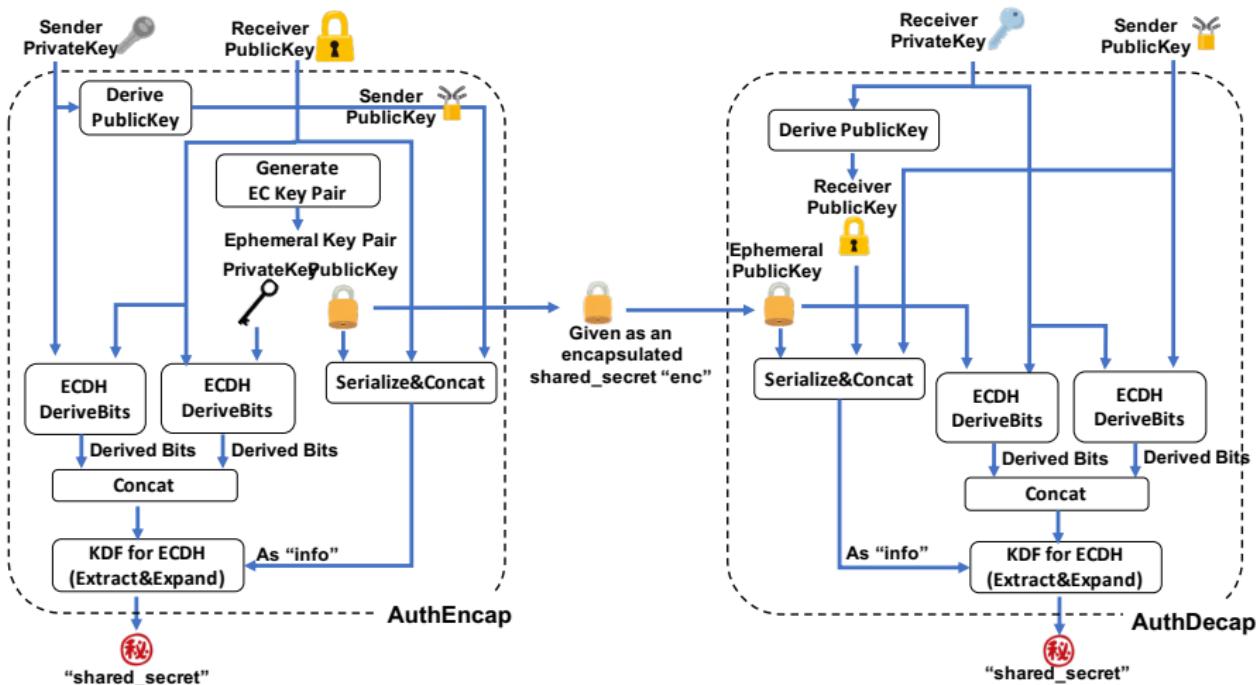


Figure: DHKEM AuthEncap/AuthDecap の構成

HPKE のセキュリティ

HPKE で、ざっと担保されるセキュリティ事項は以下の通り。¹⁴

- IND-CCA2: AEAD を使うことで選択暗号文攻撃 (Chosen Ciphertext Attack) に対して安全
- PSK, Auth, AuthPSK モードによる送信者認証

¹⁴ Secret export に対する indistinguishability もあるが今回はその解説は省略。

HPKE は、その構成上、以下の制限事項が存在する。

- Receiver Key Pair は Ephemeral ではないため、受信者が Compromise されることについて、Perfect Forward Secrecy を担保しない。
 - Base/Auth モードでは受信者の秘密鍵が Compromise されていないこと、PSK/AuthPSK モードでは受信者の秘密鍵と PSK が Compromise されていないことが必須条件である。
 - 万一の場合に被害を拡大させないためには、受信者が頻繁に鍵ペアを更新・公開するなどで暫定対処するしかない¹⁵
- PSK, Auth, AuthPSK における送信者認証は、常に Sender Private Key および PSK が Compromise されていないことが条件。(当然と言えば当然)
- 耐量子: 現行の DHKEM では量子コンピュータに対して脆弱なため、将来的に Context 生成は耐量子公開鍵暗号を規格化する必要性がある。また、Auth モード等での量子コンピュータに対するセキュリティ性能は未証明。

¹⁵Cloudflare が寄与した ODoH の実装では 1 日 1 回ローテートさせている。

<https://github.com/DNSCrypt/doh-server>

Draft 12 で利用可能な KEM, KDF, AEAD のパラメタ一覧

概略解説の最後に、Draft 12 で規定される KEM,KDF,AEAD を一覧する

KEM (DHKEM(_curve_, _kdf_))

- DHKEM(X25519¹⁶, HKDF-SHA256)
- DHKEM(X448¹⁷, HKDF-SHA512)
- DHKEM(P-256, HDKF-SHA256)
- DHKEM(P-384, HKDF-SHA384)
- DHKEM(P-521, HDKF-SHA512)

KDF

- HKDF-SHA256
- HKDF-SHA384
- HDKF-SHA512

AEAD

- AES-128-GCM
- AES-128-GCM
- ChaCha20Poly1305¹⁸

¹⁶RFC7748

¹⁷RFC7748

¹⁸RFC8439

HPKE応用: TLS Encrypted Client Hello

TLSにおけるEncrypted Client Hello (ECH)

HPKEの応用箇所として、IETF TLSWGで検討が進んでいる¹⁹Encrypted Client Hello (ECH)²⁰について簡単に紹介する。

Client Hello

TLSサーバに繋ぎに行く際、接続したいFQDNや接続可能なL7プロトコル情報(ALPN²¹)を入れてクライアントが最初に投げるメッセージ。

¹⁹<https://datatracker.ietf.org/doc/draft-ietf-tls-esni/>

²⁰ECHは過去にはEncrypted Server Name Indication (ESNI)と呼ばれていた。

²¹Application Layer Protocol Negotiation

Client Hello の脆弱性

TLS1.3 では、ClientHello より後にやり取りされるメッセージは全て暗号化されたが、Client Hello 自身は平文のまま。

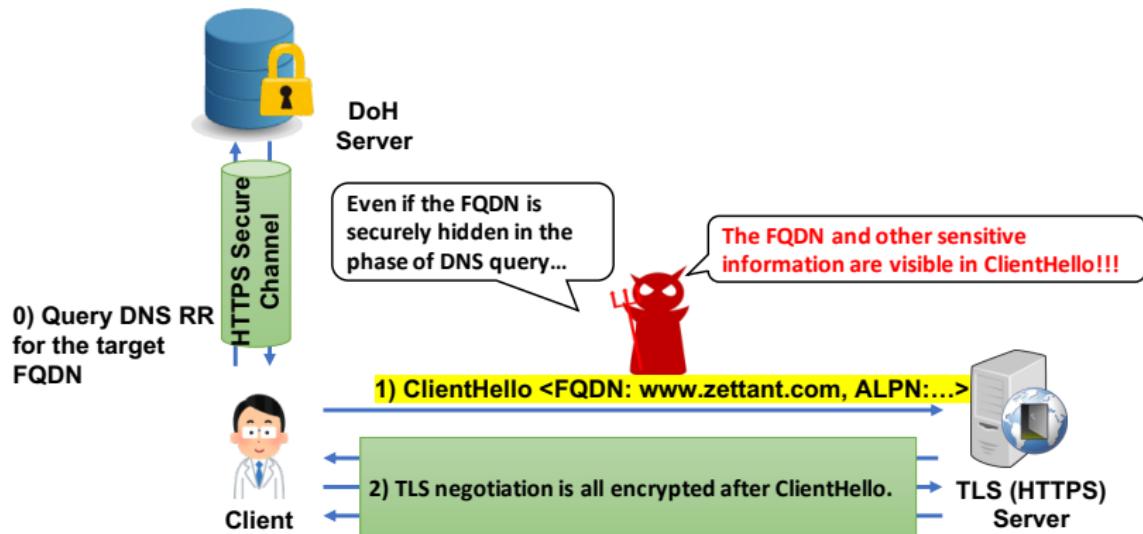


Figure: ClientHello からアクセス先の情報が漏れる

たとえ DNS のやりとりを暗号化しても、ClientHello を盗聴されればアクセス先が漏洩してしまう。

ECH: TLS ネゴシエーションを全部暗号化

Client Hello から全てネゴシエーションを暗号化してしまおう！
⇒ ECH。そしてその実現のために HPKE を応用。

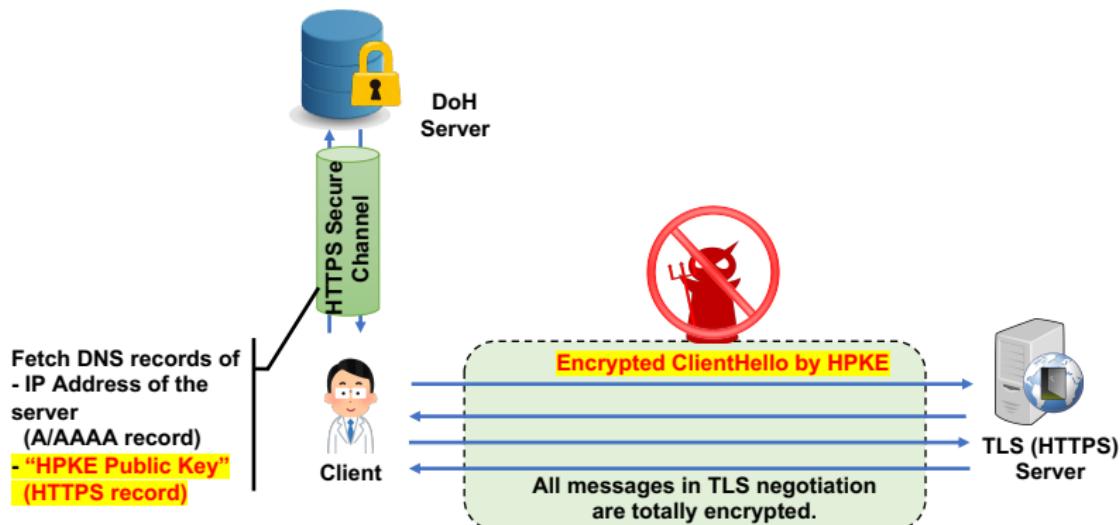


Figure: DNS HTTPS レコードを利用した Encrypted Client Hello 概略

HPKE の Receiver (サーバ) Public Key 他を含む Config は、DNS の新設レコード “HTTPS” (Type65) に記述することが検討されている。²²

²²Optional であり、事前に端末に埋め込むなど、他の手法で取得しても良い。

HPKE実装を触ってみる

現行使える HPKE のライブラリ (抜粋)

2021-10-1 時点で、Go/Rust/C (openSSL)/C++ (OpenSSL) ら低級言語中心で、HPKE のライブラリが実装されている。²³

栗原が HPKE の利用シーンでよく見かけるのは以下のあたり:

- Cloudflare ODoH 実装で利用:

Rust: <https://github.com/rozbb/rust-hpke>

Go: <https://github.com/cisco/go-hpke>

- dnscrypt 実装で利用:

Go: <https://github.com/jedisct1/go-hpke-compact>

今回は、Go のライブラリのテストコードを動作させて暗号化・復号を追いつつ、Base モードでの動作フローを解説する。

²³CFRG のリポジトリを参照: <https://github.com/cfrg/draft-irtf-cfrg-hpke>

環境準備: Golang

Golang をインストール済みのこと²⁴

栗原の環境

```
$ go version  
go version go1.17.1 darwin/arm64
```

今回は `hpke-compact`²⁵で動作を追ってみる。学習のためには他の言語・ライブラリも試すのを推奨する。

ライブラリの準備・依存パッケージのダウンロード²⁶

```
$ git clone https://github.com/jedisct1/go-hpke-compact  
$ cd go-hpke-compact  
$ go install *.go
```

²⁴Mac/Linux を想定。Zsh/Bash 等でパスが通っている前提。

²⁵<https://github.com/jedisct1/go-hpke-compact>

²⁶test コードの依存パッケージは `indirect` なので直接 *.go ファイルでパッケージダウンロード

Go で HPKE を試す

まずはテストコードを動かしてみる

```
$ go test -v
==== RUN    TestExchange
--- PASS: TestExchange (0.00s)
==== RUN    TestAuthenticatedExchange
--- PASS: TestAuthenticatedExchange (0.00s)
==== RUN    TestVectors
--- PASS: TestVectors (0.00s)
==== RUN    TestExportOnly
--- PASS: TestExportOnly (0.00s)
PASS
    checks: 0 passed 0 todo 0 failed (total)
ok      github.com/jedisct1/go-hpke-compact      0.113s
```

- TestExchange: Base モード (PSK モード) での一連の動作テスト
- TestAuthenticatedExchange: 同様に Auth モード (AuthdPSK モード)
- TestVectors: test vector による出力チェック
- TestExportOnly: secret export のテスト

今回は、基本となる Base モード(引数次第で PSK モードを兼ねる)の動作テスト(TestExchange)の中身を追って、HPKE のやり取りの理解を深める。

1. Triplet 指定による HPKE 設定初期化

hpke_test.go: TestExchange

```
suite, err := NewSuite(KemX25519HkdfSha256, KdfHkdfSha256, AeadAes128Gcm)
if err != nil {
    t.Fatal(err)
}
```

前述したように、(KEM, KDF, AEAD) の triplet で HPKE の設定を表す。上記テストコードでは、triplet の各要素として

- KEM: DHKEM(X25519 + HKDF-SHA256)
- KDF: HKDF-SHA256
- AEAD: AES128-GCM

を利用して HPKE Suite のオブジェクトを生成している。Sender/Receiver の両者で実行。

2@Receiver(Server). Receiver Key Pair を生成

hpke_test.go: TestExchange

```
serverKp, err := suite.GenerateKeyPair()  
if err != nil {  
    t.Fatal(err)  
}
```

suite オブジェクトを生成する際に X25519 を KEM で利用すると指定している。このため、serverKp として、X25519 の鍵ペアを生成。

ここで生成した鍵ペアの公開鍵 serverKp.PublicKey オブジェクトを、Sender(Client) になんらかの方法で渡しておく。

3@Sender(Client). Sender の EncryptionContext と “enc” を生成

hpke_test.go: TestExchange

```
clientCtx, encryptedSharedSecret, err := suite.NewClientContext(  
    serverKp.PublicKey, // 受け取った公開鍵  
    []byte("test"),     // info (App名など)  
    nil                // PSK モードの時は PSK オブジェクトを入れる  
)  
if err != nil {  
    t.Fatal(err)  
}
```

生成した Context: `clientCtx` はクライアントで保持。のちに AEAD 暗号化で利用。

一方、同時に生成する “enc”: `encryptedSharedSecret` オブジェクトは、`Receiver(Server)` になんらかの方法で渡しておく。AEAD 暗号化データと一緒に渡しても良い。

4@Sender(Client). AEAD 暗号化データの生成

hpke_test.go: TestExchange

```
ciphertext, err := clientCtx.EncryptToServer(  
    []byte("message"), // 暗号化対象データ。string を byte にしている。  
    nil               // aad (optional)  
)  
if err != nil {  
    t.Fatal(err)  
}
```

ciphertext を Receiver(Server) へ送付。AEAD 暗号化の Nonce は、前述のように、Context 内部の変数を元に生成されているので意識しなくて良い。

5@Receiver(Server). Receiver の EncryptionContext を生成

hpke_test.go: TestExchange

```
serverCtx, err := suite.NewServerContext(  
    encryptedSharedSecret, // 受け取った “enc”、実態は Ephemeral 公開鍵  
    serverKp,             // 1. で生成した Receiver Key Pair  
    []byte("test"),       // info (App 名など)、sender 側の記述と一致が必須  
    nil                  // PSK モードの時は PSK オブジェクトを入れる  
)  
if err != nil {  
    t.Fatal(err)  
}
```

Sender(Client) から受け取った “enc” を用いて、Sender と同じ EncryptionContext を生成する。

6@Receiver(Server). AEAD 復号

hpke_test.go: TestExchange

```
decrypted, err := serverCtx.DecryptFromClient(
    ciphertext, // AEAD 暗号化データ
    nil         // aad (Optional)、暗号化時と同じものである必要
)
if err != nil {
    t.Fatal(err)
}
```

Receiver(Server) の context を用いて復号。

Sender → Receiver への暗号化データ送付について説明したが、共有した context を使いまわしてその後の「Response」としての Receiver(Server) から Sender(Client) への暗号化データ送付も可能。

7@Receiver(Server)→Sender(Client). 応答の暗号化・復号

hpke_test.go: TestExchange

```
// サーバでの AEAD 暗号化
ciphertext, err = serverCtx.EncryptToClient([]byte("response"), nil)
if err != nil {
    t.Fatal(err)
}

// クライアントでの AEAD 復号
decrypted, err = clientCtx.DecryptFromServer(ciphertext, nil)
if err != nil {
    t.Fatal(err)
}
```

まとめ

まとめ

- HPKE の概略を紹介した
- HPKE の応用: ECH を紹介した
- HPKE の実装、テストコードを解説した

HPKE という規格は、今後さまざまな箇所の E2E 暗号化に応用が期待できる。多分押さえておいて損はない。