Linuxカーネルハッキング
～IPv6プロトコルスタックとxfrm～

横河電機技術開発本ユビキタス研究所

宮澤　和紀 (USAGIプロジェクト)

自己紹介

YOKOGAWA

- 1999年横河電機入社
  - Javaを使ったアプリケーションの研究開発に従事
  - Linuxに関しては、１ユーザ(アプリケーションプログラマ)
- 2001年USAGIプロジェクトに参加
  - 主にIPsecの開発を主に担当
  - Linux-2.4ベースのIPsecを実装
  - Linux-2.5に実装されたIPsecにIPv6サポートを追加

Topics

- 導入
- IPsec
- Linuxのカーネル開発
- Linuxのネットワークスタック
  - xfrmの概要
  - ソースを参照しながらの説明
- まとめ

プロトコルスタックの実装手順

1. RFCなどプロトコルを規定した文書等を参照
2. 現状の設計や実装の分析
3. プロトコル以外のスペックの検討
4. 全体設計
5. プログラミング
6. 既存機能の動作検証
7. 新機能の動作検証
8. 相互接続性の検証

IPsec

IPsecの概要

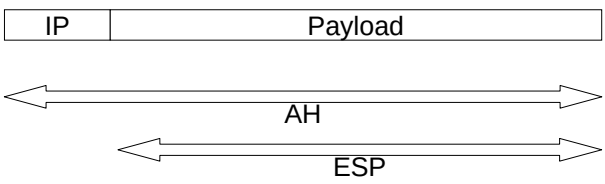
- IP層において、IP層を通過するトラフィックに対してセキュリティサービスを提供する
- セキュリティサービス
  - 完全性 (connection less)
  - 秘匿 (トラフィックの内容)
  - リプレイプロテクション
  - 送信元認証

IPsec関連文書

- RFC4301 ← RFC2401
  - IPsecアーキテクチャを規定
- RFC4302 ← RFC2402
  - AH (Authentication Header)
- RFC4303 ← RFC2403
  - ESP (Encapsulation Payload)
- RFC4304
  - ESN (Extended Sequence Number)
- RFC4305
  - Algorithm Requirement
- RFC4306
  - IKEv2 (鍵交換プロトコル)

AHとESP

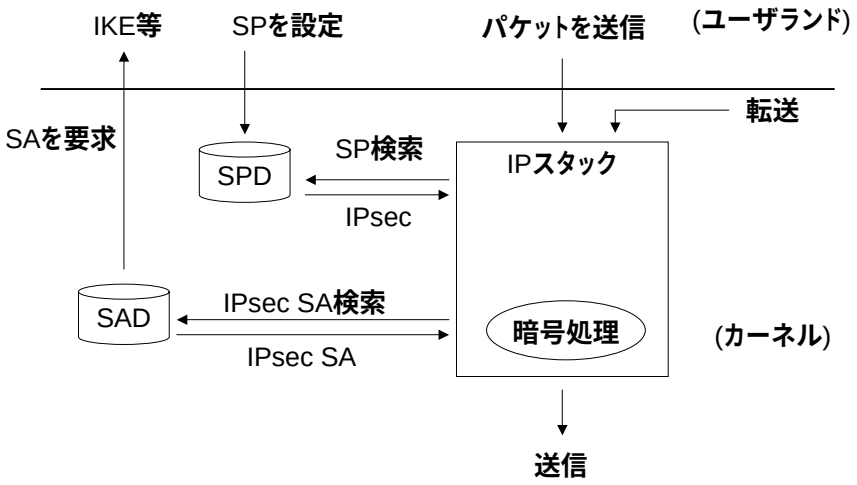
- Authentication Header (AH)
  - 認証、リプレイプロテクション、送信元認証サービスを提供する
  - IPヘッダまで含まれる
- Encapsulation Payload (ESP)
  - IPパケットのペイロードに対して、秘匿、認証、リプレイプロテクションサービスを提供
  - ペイロードのみに適用される



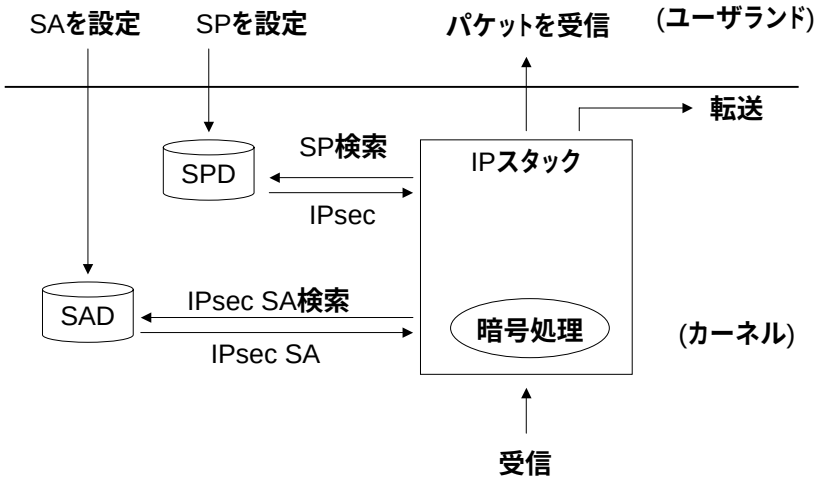
IPsecの構成要素

- Security Policy
  - トラフィックを識別するセレクタと適用する処理(AH,ESP)を規定
- Security Association (IPsec SA)
  - トラフィックを処理するために必要なパラメータのセット
  - 一方向に一つあり、双方向通信では二つ必要
- SPD, SAD
  - Security Policy, Security Associationのデータベース
- 鍵交換アプリケーション
  - 通信相手との間でIPsec SAを管理するアプリケーション

IPsecの動作 (outbound)



IPsecの動作 (inbound)



Linuxのカーネル開発

プロトコルスタックの実装手順(Linux)

- RFCなどプロトコルを規定した文書等を参照
- 現状の設計や実装の分析
  - カーネルを変更するべきか
  - 同じプロトコルを扱う既存のプロジェクトの動向など
- プロトコル以外のスペックの検討
  - API等の後方互換性の確保
- 設計と実装
- 各種検証
- パッチの作成
- パッチの送付

Linuxの開発サイクル

- 2.4,2.5のような安定版と開発版の区別はもはやない
- ３～４ヶ月ごとのリリース
- 新規リリース後一週間程度が新機能(パッチ)の受付期間
- 受付終了、同時に-rc1リリース
- -rcX (release candidate)による検証とバグフィックス
- 新規リリース
- linux-2.6.x.y:-stableでのバグフィックス

Linux Network Stack 開発

- ML
  - linux-kernel@vger.kernel.org : カーネル全般
  - netdev@vger.kernel.org : ネットワーク開発
  - linux-net@vger.kernel.org: ネットワーク全般
  - linux-crypto@vger.kernel.org : カーネルの暗号処理
- Netdev
  - バグの報告 (linux-netの方が良い場合も)
  - 現状の実装や設計に関する質問 (コードを指定して)
  - 新機能や修正の設計に関する議論
  - 実装に関する議論やコメントの募集 [RFC]
  - パッチの送付

カーネルの変更

- モジュラリティを重視する
- 重複コードは避ける
- #ifdefは、できるだけモジュールに閉じる
- ユーザランドでできることはユーザランドで
- ネットワークスタックとユーザランドのインタラクションは netlinkがbetter (場合による)
- smpのスケーラビリティ
- 後方互換性を確保する
  - 古いユーザランドのバイナリが新しいカーネルで動作すること
  - 新規インターフェースは拡張性を大切に
- 特許に敏感

パッチファイルの送付

- パッチを送付する場合パッチ機能や対象ファイルを元に適切に分割する
  - ひとめ見て変更が分かるパッチがよい
- Singned-off-by
  - パッチの出所明示 (Developers Certificate of Origin 1.1)
- RFC2646に注意
  - Thunderbirdでは、先頭にスペースしか無い行のスペースが削除されるのでpatchファイルをcopy and pasteすると壊れる
- hit and awayは嫌われる
  - 新機能を実装するがメンテナンスしない
  - 結果として継続しているプロジェクトが有利になることもある
  - 粘り強い交渉が必要な場合もある

Linuxのネットワークスタック

ソースツリーの構成(略)

linux

- /arch
- /drivers/net
- /include
  - /asm
  - /asm-generic
  - /linux ←ユーザランドに公開されるヘッダ
  - /net　←カーネルソース内に公開されるヘッダ
- /net
  - /core
  - /ipv4
  - /ipv6
  - /xfrm

よく用いられるコード

```
struct param;

struct protocol {
   int (*receive)(struct param *p);
};

int ipv6_receive(struct param *);
struct protocol ipv6_cb;

int init(){
   ipv6_cb.receive = ipv6_receive;
   register_protocol(IPV6, &ipv6_cb);
}
```

送信処理(UDPの場合)

- udpv6_sendmsg : ソケットオプション、udpヘッダ処理
- ip6_sk_dst_lookup : ルーティング
- ip6_append_data : パケットのペイロード部分を生成
- udp_v6_push_pending_frames: チェックサムの計算
- ip6_push_pending_frames : IPv6ヘッダと拡張ヘッダ
- dst_output : 送信一般関数
- ip6_output : IPv6スタックの送信関数
- dev_queue_xmit : デバイスドライバの送信関数呼び出し

受信処理(UDPの場合)

- sock_queue_rcv : ソケットへキューイング
- udpv6_rcv : チェックサムの計算
- ip6_input : 拡張ヘッダ処理
- ip6_route_input : 受信パケットのルーティング
- ip6_rcv : IPv6パケットの受信ヘッダとhop-by-hop処理
- netif_receive_skb : ドライバが受信したパケットを処理
- NET_RX_SOFTIRQ->net_rx_action
- netif_rx_schedule

3種類のパケット処理フレームワーク

- routing table + netdevice　(静的)
  - 宛先アドレスをベースにパケットの送信先や送信方法を決定する
  - ポリシルーティングによって細かな制御も可能
  - 仮想ネットワークデバイスでパケットを操作することも可能
- netfilter (動的)
  - フィルタ機能を提供する
  - トラフィックを識別するセレクタと処理からなる
  - コネクショントラッキングが特徴
- xfrm (半動的)
  - トラフィックを識別するセレクタとセレクタに従った処理を行う機構の2つから成る
  - outbound処理に対してはキャッシュ機構を持つ

xfrmを構成する主な構造体

- xfrm_policy
  - IPsecのセキュリティポリシに対応する構造体
- xfrm_state
  - IPsec SAに対応する構造体
- xfrm_tmpl
  - xfrm_policyとxfrm_stateを関連付けるための構造体
- xfrm_selector
  - xfrm_policyやxfrm_stateなどでトラフィックを識別する情報を格納する

送信処理とxfrm

- udpv6_sendmsg
- ip6_sk_dst_lookup
- xfrm_lookup : **ポリシの検索**
- ip6_append_data
- udp_v6_push_pending_frames
- ip6_push_pending_frames
- dst_output
- xfrm6_output → esp6_output : **パケット処理**
- ip6_output
- dev_queue_xmit

xfrm_lookup

- **ソケットに関連付けられた**xfrm_policy**があるかチェック**
- xfrm_policy_lookup**を呼び出し**xfrm_policy**を検索**
- **結果は**flow_cache**にキャッシュされる**

- **ポリシが**IPsec**の場合**
- xfrm_find_bundle **で** xfrm_policy **にキャッシュされた** dst_entry**をチェック (後述の**xfrm_bundle_create**にて生成)**

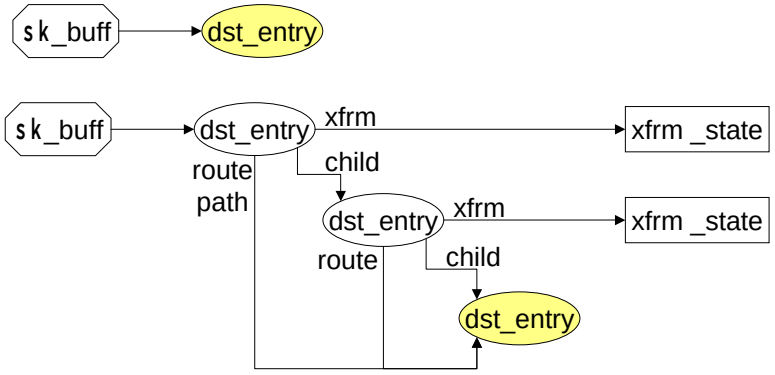
- **キャッシュがない場合**
- xfrm_temple_resolve

xfrm_tmpl_resolve

- xfrm_policyのxfrm_tmplを元にxfrm_stateを探す
- xfrm_stateが見つからない場合でxfrm_stateを要求する必要がある場合は、ユーザーランドに要求メッセージ(SADB_ACQUIREなど)が送信される
- xfrm_tmpl_resolveの戻り値がEAGAINであった場合xfrm_lookup内では、schedule()を呼び出しprocessがブロックされる

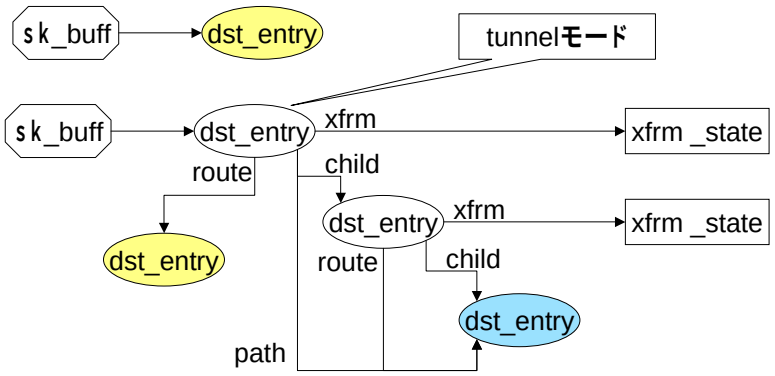
xfrm_bundle_create

- xfrm_bundle_createは、xfrm_tmpl_resolveの結果を元にdst_entryのチェーンを作る関数である。



xfrm_bundle_create (cont)

- xfrm_bundle_createは、xfrm_tmpl_resolveの結果を元にdst_entryのチェーンを作る関数である。

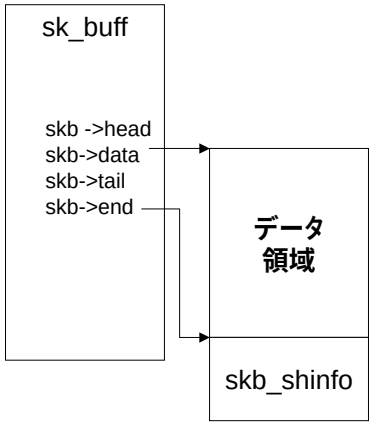


ip6 _append _data

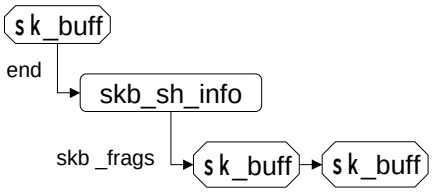
- ユーザランド(ユーザ空間)からデータをコピーしてsk_buffにつめる関数
- あらかじめ後の処理で必要になる領域を確保(IPv6ヘッダやフラグメントヘッダ,ESPのオーバーヘッドなど)
- MSG_MOREを使わずに、MTUに収まる場合は簡単
- フラグメントが必要な場合には、フラグメントヘッダの分を空けながらコピーする
- MSG_MOREで後からMTUを超えた場合はデータをコピーしてフラグメントヘッダの領域を確保する

sk _buff

__alloc_skb直後



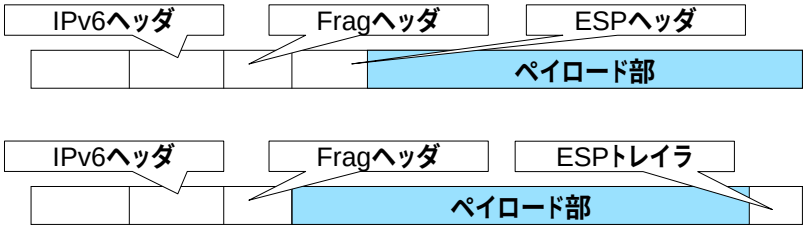
sk_buffとskb_sh_info



ip6 _append _data

ペイロードのコピー

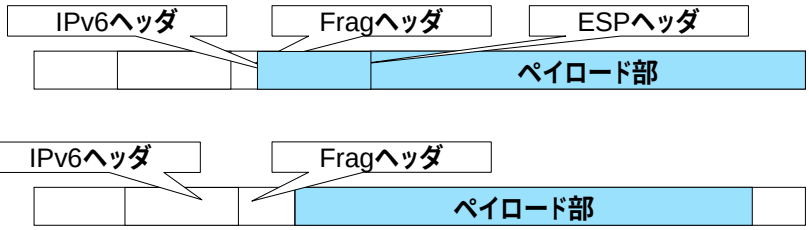
ペイロードのコピー (フラグメント)



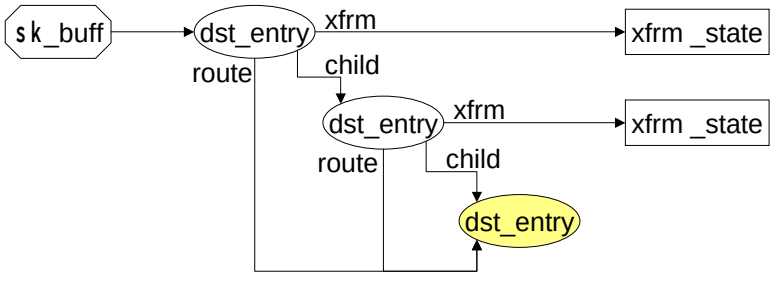
ip6_push_pending_frams

ヘッダの生成

ヘッダの生成 (フラグメント)



dst_output→xfrm6_output



受信処理とxfrm

- sock_queue_rcv
- xfrm_policy_check : **ポリシの検証**
- udpv6_rcv
- xfrm6_input → esp6_input : **パケット処理**
- ip6_input
- ip6_route_input
- ipv6_rcv
- netif_receive_skb
- NET_RX_SOFTIRQ->net_rx_action
- netif_rx_schedule

xfrm6_input

- **AH, ESP, IPCOMPの共通関数**
- **パケットを解き**xfrm_state**を検索し認証や復号などを行う。**
- **トンネルモードの場合は、処理後**netif_rx**を呼び出す。**
- **使用した**xfrm_state**は、**sk_buff**の**sec_path**にそのポインタが格納され、後の**xfrm_policy_check**で利用される**

xfrm_policy_check

- 受信用のxfrm_policyを検索し、そのxfrm_tmplとsk_buffのsec_pathを比較検証することで正しい処理が行われたかを判断する。

まとめ１

- ネットワークスタックを開発するにあたって
  - 仕様がかならずしも全てを網羅していない場合がある
  - 仕様で疑問に思ったことはMLなどで聞いてみる
    - 仕様の改善につながることもある
  - 相互接続性が重要

まとめ２

- Linuxカーネル
  - メモリマネージメントやスケジューラは別だが、ネットワークスタックはカーネルの開発だからといって特別難しいところは少ない
  - ロックやリファレンスカウントの管理に注意
- 種々のユーティリティ関数が用意されている
- 関数ポインタとコールバックを多用しているので一見読みにくく感じる

まとめ３

- 成果を本家に反映するのであれば、Linux開発サイクルを意識した工程を考える
- 特許問題などは明確にしておく
- パッチを送付する際は、分かりやすく分割すること
- まずは質問する
- 完成してからではなく、過程を見せることも重要

まとめ４

- xfrmは、カーネル内でポリシに従い定義された動作を行う汎用なフレームワークになっている
- 多重にキャッシュされるので効率的に処理できるが、変数を変更する際は注意が必要
  - 基本的な部分は、実装されている