SecHack365

StegRDB

RDBを介してパケットを転送する仮想L2スイッチ

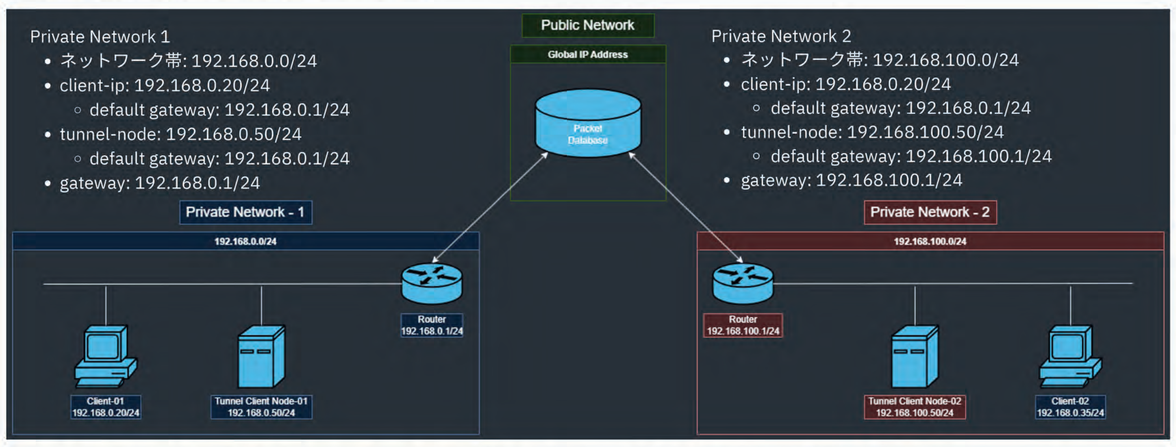
学習駆動コース　坂井ゼミ
相田 優希(Masaki Aida)

ソフトウェアの概要

StegRDBの最大の特徴は、名前の通りRDB(Relational Database)を介してパケットを転送する点です。RDBへの接続が確立できる環境であれば動作し、RDBを経由して全パケットを記録し永続化する為、リアルタイムでのステートフル解析や転送パケットの完全な記録と保証が可能となります。

ソフトウェアの実装説明と実行環境

StegRDBは、Linux環境で動作するネットワークのトンネルです。
実際の実行環境は以下に添付した下の画像の通りです。
(下: 実行環境のネットワーク図)



このシステムは、複数のトンネルクライアントノード(Node-01、Node-02)で構成され、各ノードがプロミスキャスモードで、
各ネットワークパケットを収集します。
システムの中核となるパケットキャプチャエンジンは、
Linuxカーネルの低レベルネットワークインターフェースを直接制御しています。具体的には、AF_PACKET SOCKETを利用しlibc::sockaddr_ll構造体のsll_pkttypeフィールドを使用してパケットの種類を識別し、値がPACKET_OUTGOINGの場合を除外することで、自身が送信したパケットを効率的にフィルタリングしています。(右画像: パケットの取得処理部分)

```rust
let mut buf = vec![0u8; 65536];

info!("インターフェース {} でパケット受信を開始", interface.name);
let writer = PacketWriter::default();

loop {
    match socket::recvfrom::<SockaddrStorage>(sock_fd.as_raw_fd(), &mut buf) {
        Ok((size, Some(addr))) => {
            unsafe {
                let sock_addr_ll = addr.as_ptr() as *const libc::sockaddr_ll;

                trace!(
                    "受信アドレス: {:?}, ソケットアドレス: {:?}, パケットタイプ: {}",
                    addr,
                    sock_addr_ll,
                    (*sock_addr_ll).sll_pkttype
                );

                if !sock_addr_ll.is_null() && (*sock_addr_ll).sll_pkttype == PACKET_OUTGOING {
                    // 自身が送信したパケットはスキップ
                    continue;
                }
            }

            if let Err(e) = writer.process_packet(&buf[..size]).await {
                error!("パケット処理エラー: {}", e);
            }
        },
        Ok((_, None)) => continue,
        Err(nix::errno::Errno::EAGAIN) => continue,
        Err(e) => {
            error!("パケット読み取りエラー: {}", e);
            break;
        },
    }
}
```

キャプチャされたパケットは、L2(Ethernet)からL4(TCP/UDP)まで、各レイヤーで解析が行われます。
具体的には、MACアドレス、EtherType、IPアドレス、プロトコル、ポート番号、TCPフラグなどの情報を解析し、ホワイトリスト/ブラックリストベースのフィルタリングに適用します。
適用後、データベースへバルクインサートされます。
(左画像: ethernet freamの解析部分)

```rust
pub fn parse_ethernet_header(frame: &[u8]) -> Result<EthernetHeader, AnalyzeResult> {
    if frame.len() < 14 {
        logs_log!("ethernet headerが短すぎます");
        return Err(AnalyzeResult::Reject);
    }

    let (src_mac, dst_mac) = extract_mac_addresses(frame);
    let ether_type = parse_ether_type(frame);

    Ok(EthernetHeader { src_mac, dst_mac, ether_type })
}

fn extract_mac_addresses(frame: &[u8]) -> (MacAddr, MacAddr) {
    let dst_mac = MacAddr([frame[0], frame[1], frame[2], frame[3], frame[4], frame[5]]);
    let src_mac = MacAddr([frame[6], frame[7], frame[8], frame[9], frame[10], frame[11]]);
    (src_mac, dst_mac)
}

fn parse_ether_type(frame: &[u8]) -> EtherType {
    let type_value = u16::from_be_bytes([frame[12], frame[13]]);
    EtherType::from(type_value)
}
```

データベースには、パケットのメタデータ（タイムスタンプ、送信元/宛先のMAC/IPアドレス、ポート番号など）とともに、パケット本体が保存されます。また、processed_packetsテーブルによる処理済みパケットの管理により、効率的なパケットの運用を実現しています。
各ノードは10ミリ秒間隔でデータベースをポーリングし、他ノードが書き込んだパケットを取得します。取得したパケットは、タイムスタンプに基づいて適切な順序で再生されます。この機能により、ネットワークトラフィックの完全な再現が可能です。

timestamp	node_id	src_mac	dst_mac	ether_type	ip_protocol	src_ip	dst_ip	src_port	dst_port	raw_packet
2754744 2025-01-20 10:15:58.770925 +00:00	160	1c:7c:98:54:df:c0	bc:24:11:11:42:f2	2048	6	13.41.103.230	192.168.0.35	443	40358	0xBC24111142F21C7C
2754745 2025-01-20 10:15:58.770178 +00:00	160	1c:7c:98:54:df:c0	bc:24:11:11:42:f2	2048	6	13.41.103.230	192.168.0.35	443	40358	0xBC24111142F21C7C
2754746 2025-01-20 10:15:57.805951 +00:00	160	1c:7c:98:54:df:c0	bc:24:11:11:42:f2	2048	6	13.41.103.230	192.168.0.35	443	40358	0xBC24111142F21C7C
2754747 2025-01-20 10:15:56.629095 +00:00	160	1c:7c:98:54:df:c0	bc:24:11:11:42:f2	2048	17	1.1.1.1	192.168.0.35	53	32824	0xBC24111142F21C7C
2754748 2025-01-20 10:15:56.572345 +00:00	160	1c:7c:98:54:df:c0	bc:24:11:11:42:f2	2048	17	1.1.1.1	192.168.0.35	53	39958	0xBC24111142F21C7C
2754749 2025-01-20 10:15:52.907339 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	17	154.213.187.13	192.168.0.150	38004	123	0x00E04C6802B71C7C
2754750 2025-01-20 10:15:31.455210 +00:00	160	1c:7c:98:54:df:c0	01:00:5e:00:00:01	2048	2	192.168.0.1	224.0.0.1	4452	61083	0x01005E000001C7C
2754751 2025-01-20 10:15:30.943210 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	194.0.234.36	192.168.0.150	65105	22	0x00E04C6802B71C7C
2754752 2025-01-20 10:15:30.912284 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	3.143.152.247	192.168.0.150	50852	7473	0x00E04C6802B71C7C
2754753 2025-01-20 10:15:28.385800 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	17125	0x00E04C6802B71C7C
2754754 2025-01-20 10:15:23.406215 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	3119	0x00E04C6802B71C7C
2754755 2025-01-20 10:15:17.956402 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	890	0x00E04C6802B71C7C
2754756 2025-01-20 10:15:12.001920 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	16547	0x00E04C6802B71C7C
2754757 2025-01-20 10:15:09.963239 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	4732	0x00E04C6802B71C7C
2754758 2025-01-20 10:15:08.914725 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	10027	0x00E04C6802B71C7C
2754759 2025-01-20 10:14:48.336798 +00:00	160	1c:7c:98:54:df:c0	00:e0:4c:68:02:b7	2048	6	36.212.60.55	192.168.0.150	60000	4276	0x00E04C6802B71C7C

StegRDBの名前の由来

StegRDBという名称は、情報隠蔽技術「Steganography」と「Relational Database (RDB)」を組み合わせて生まれました。
一見、通常のRDBクエリに見える通信の中に、実は別の情報を隠蔽できるという本システムの特徴を表現するため、古くから使われてきた情報隠蔽技術「Steganography」の概念を名前に取り入れました。
これにより、RDBを介した新しい形の安全な通信トンネルを実現しています。

なぜRDBを中核に据えたのか

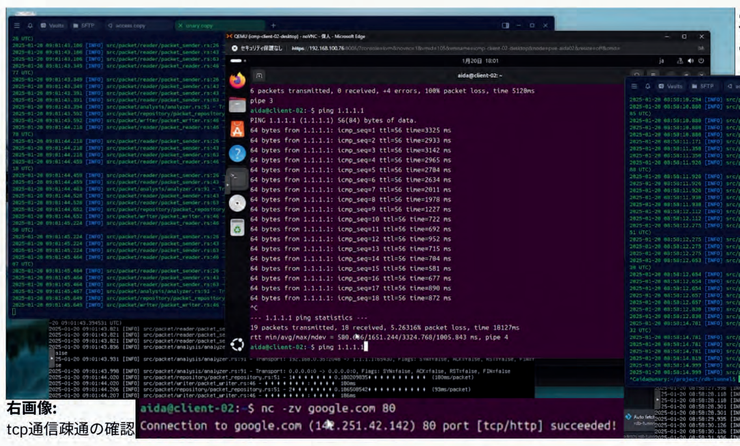
私は当初、SecHack365においてIDS（侵入検知システム）の開発に取り組んでいました。しかし、IDSは既に優れた技術が確立されており、より革新的なアプローチを模索したいと考えるようになりました。
そこで着目したのが、通信をより能動的に制御できるVPNの開発です。さらに、IDS開発時に検出結果や解析内容をRDBに保存していた経験から、RDBでのパケット転送でパケットを転送するという着想を得ました。これらを組み合わせ、世界初となるRDBを通信制御の中核に据えた仮想L2スイッチの開発に着手しました。

StegRDBを作るにあたっての思想のお話

従来のネットワークシステムでは、スループットや遅延などの性能指標が重視される傾向にあります。しかし、StegRDBでは、あえて性能最適化の道から離れ、RDBを中核に据えることで得られる新たな可能性を追求しました。
私は、低レイヤーのネットワーク技術とアプリケーションレイヤーのRDBという、普通なら組み合わせることが考えにくい技術を掛け合わせることで、ネットワークの世界に新しいアイデアを生み出すことができたと感じています。異なるレイヤーの技術を組み合わせるという発想は、様々な技術分野に触れてきた経験があったからこそ生まれたのだと思います。

今後の展望

最後に、今後の展望としては、自動フェイルオーバー対応による可用性の向上や
異なるデータベース製品へのスムーズな移行機能、IPv6のサポートです。
現在はPostgreSQLを利用していますが、MySQLやMariaDBなど、ユーザーの要件に応じて柔軟にデータベース製品を選択できるようにしていきたいと考えています。



右画像: 実行時画像
下画像: 実行時のコンソール画面

右画像:
tcp通信疎通の確認

```
aida@client-02:~$ nc -zv google.com 80
Connection to google.com (142.251.42.142) 80 port [tcp/http] succeeded!
```

```sql
CREATE TABLE IF NOT EXISTS packets
(
    id          BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1),
    timestamp   TIMESTAMPTZ NOT NULL,
    node_id     SMALLINT    NOT NULL,
    src_mac     MACADDR     NOT NULL,
    dst_mac     MACADDR     NOT NULL,
    ether_type  INTEGER     NOT NULL,
    ip_protocol INTEGER     NOT NULL,
    src_ip      INET        NOT NULL,
    dst_ip      INET        NOT NULL,
    src_port    INTEGER     NOT NULL,
    dst_port    INTEGER     NOT NULL,
    raw_packet  BYTEA       NOT NULL,
    CONSTRAINT packets_pkey PRIMARY KEY (id, timestamp)
);

CREATE TABLE processed_packets (
    packet_id BIGINT,
    node_id SMALLINT,
    processed_at TIMESTAMPTZ DEFAULT NOW(),
    PRIMARY KEY (packet_id, node_id)
);
```

左画像: パケット保存用データベースのテーブル構造
パケットの確実な配送を保証するため、processed_packetsテーブルを別途設計しました。当初はタイムスタンプベースでパケットの取得管理を行っていましたが、処理時間の変動やシステム負荷による影響で、パケットの取りこぼしや重複取得が発生する課題がありました。
そこで、各パケットの処理状態を(packet_id, node_id)の組み合わせで管理する方式に移行しました。これにより、処理時間に依存せず、各ノードが未処理のパケットのみを確実に取得できるようになりました。また、パケットの順序はタイムスタンプを用いて制御しつつ、processed_packetsテーブルで重複処理を防止することで、信頼性の高いパケット配送を実現しています。

6月 7月 8月 9月 10月 11月 12月 1月

IDS(侵入検知システム)を作りたい rustを初めて触ってみる

Libpcapを用いたパケットの解析ができた

ビジュアライズの為にTUIアプリを初めて作ってみた

ethernet fragmentからtcpパケットの復元、非暗号通信の解析

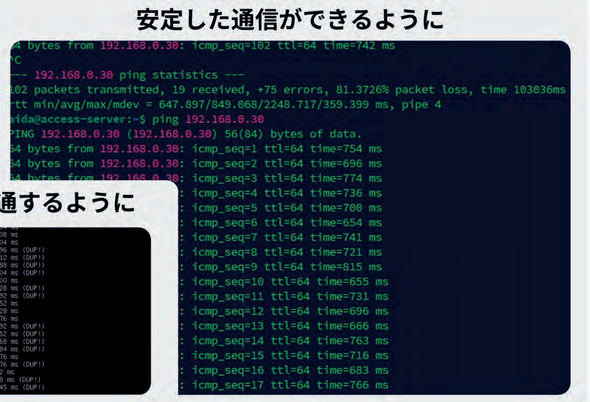
解析した内容をRDBに保存し、webサイトから表示できるように

webサイトに表示する為にRDBへ一時保管していたことをヒントにRDBを利用してパケットを転送することを思いつく

StegRDB開発に着手

ICMPパケットが疎通するように

AF_Packet Socketを利用するなど工夫することでパケットのループを0に安定した通信ができるように



National Institute of Information and Communications Technology

https://github.com/aida0710/stegrdb
https://www.aida0710.work