サイバーセキュリティ PBL I

高野 祐輝

2019 年 3 月 1 日

目次

1　目的と評価方法

最優　ファイアウォール技術とペネトレーションテストを組み合わせて，検疫ネットワークの設計と構築を行うことができる

優　ファイアウォール技術で DeMilitarized Zone のあるネットワーク設計と構築を行うことができる

良　ファイアウォール技術で適切なネットワークアクセスコントロールができる

可　各種サイバー攻撃手法と防御手法について論じることができる

2　セキュリティ哲学

2.1　サイバーセキュリティとは何か

2.2　サイバーキルチェーン

[1]

2.3　セキュリティポリシとユーザビリティ

[2]

3　TCP/IP の基礎

3.1　OSI 参照モデル

インターネットで利用されるプロトコルは、The Internet Engineering Task Force (IETF) という標準化団体により策定され、その標準は Request for Comments (RFC) という名のオープンな仕様として発行されている。例えば、我々が利用しているインターネットプロトコルであるインターネットプロトコル バージョン 4 は、1981 年に 791 番目の RFC として策定された [3]。

IETF 以外の通信に関する標準化団体としては International Telecommunication Union Telecommunication Standardization Sector (ITU-T) や、International Organization for Standardization (ISO) が存在する。実は、1977 年から 1982 年かけて、ITU-T や ISO がコンピュータネットワークの標準通信プロトコルとして、Open Systems Interconnection (OSI) の策定を行っていた。その当時は標準的な通信プロトコルは存在せず、ベンダーごとに様々なプロトコルが利用されていたため、通信プロトコルの統一化が求められていたのである。しかしながら、最終的に OSI は主流とはならず、IETF によって策定されたインターネットプロトコルが広く利用されるようになっていった。

OSI 自体は残らなかったが、OSI 策定の際に考案された OSI 参照モデルと呼ばれるネットワークの抽象化手法は、今日でも広く受け入れられている。図 1 は、OSI 参照モデルによるネットワークの抽象化モデルを表している。OSI 参照モデルでは、ネットワークの機能を階層構造にもとづいて抽象化しており、この抽象化

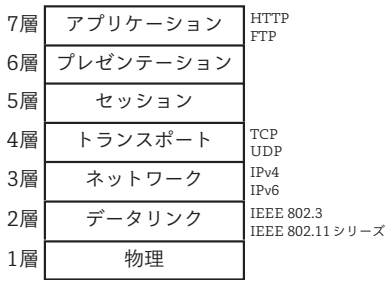

図 1　OSI 参照モデル

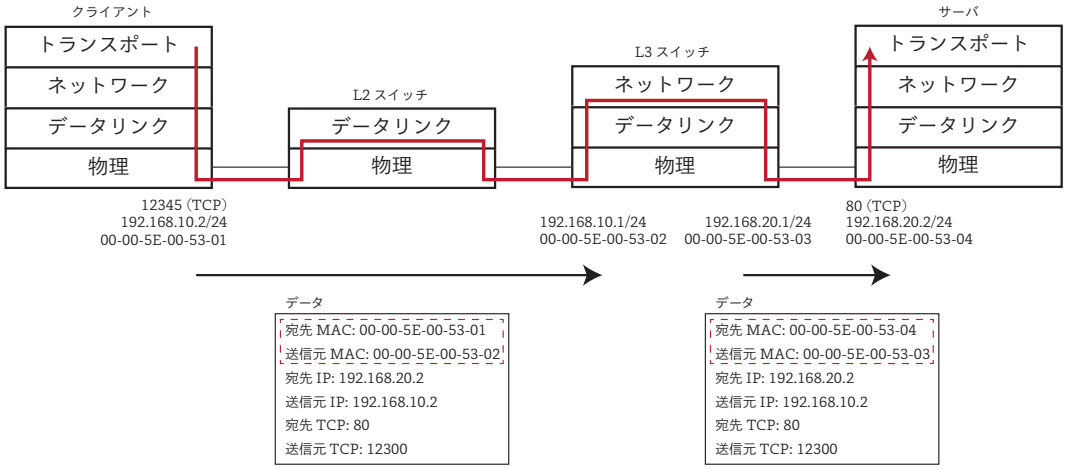

図 2　各層でのデータ転送

をレイヤリングなどと呼ぶ。OSI 参照モデルでは、下から順に 1 層に物理層、2 層にデータリンク層、3 層にネットワーク層、4 層にトランスポート層、5 層にセッション層、6 層にプレゼンテーション層、7 層にアプリケーション層が位置する。ちなみに、各層のことをレイヤ 1、レイヤ 2 といったり、更に略して L1、L2 などということもある。

図 2 は各層でデータ転送が行われている様子を示している。[*1] データリンク、ネットワーク、トランスポート層のプロトコルにはそれぞれアドレスがあり、各層は、そのアドレスに基づいて転送を行う。データリンク層プロトコルの一つである IEEE 802 では、アドレスは 42 ビットで表され、16 進数で表現すると 00-00-5E-00-53-02 といった表記になる。図 2 中で宛先 MAC と示される値は、IEEE 802 の宛先 MAC アドレスを示している。なお、MAC は Media Access Control の略である。データリンク層は、ローカルなネットワークでの通信を行うために用いられる。そのため、MAC アドレスはそのローカルな環境では一意に識別できる必要がある。データリンク層の詳細については 3.4 節で解説する。

ネットワーク層プロトコルの一つである IP のアドレスは、192.168.10.2/24 という 32 ビットの数値で表され、/24 はネットワークのサブネット長を示している。図 2 では、192.168.10.0/24 と 192.168.20.0/24 というサブネットが示されている。IP は、全世界で通信を行うために用いられるプロトコルであり、基本的には IP アドレスは世界で一意に識別できるように割り当てるのが設計理念となっている（現実的にはそうはなっ

[*1] この図の意味することは現時点では理解できないかもしれないが、この図の意味することろを説明するのが本節の目標であるため、現段階で理解できなくても問題ない。

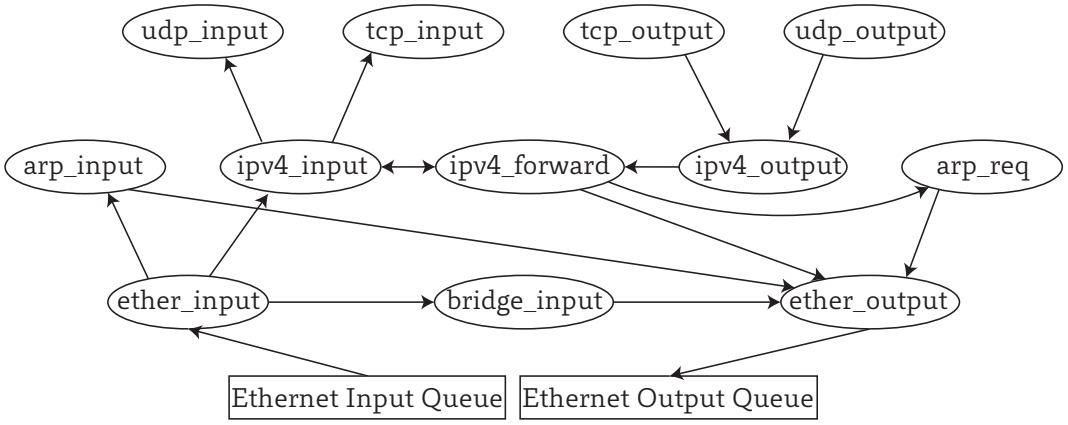

図 3　おもちゃのネットワークスタックのデータフロー図

ていないが)。なお、前述のアドレスは IPv4 アドレスであるが、IPv6 の場合は 128 ビットのアドレス空間を持つ。ネットワーク層の詳細については 3.7 節で解説する。

トランスポート層プロトコルの TCP と UDP のアドレスは 16 ビットで示され、一般的にポート番号と呼ばれ、TCP や UDP はポート番号をもとにアプリケーションプロセスの識別を行う。よく利用されるポート番号は、インターネット上で利用される識別情報の管理割当を行っている Internet Assigned Number Authority (IANA) が定義しており [4]、一般的にこのようなポート番号を Well Known ポート番号と呼ぶ。例えば、TCP の 80 番ポートは HTTP で利用され、普段我々が Web を閲覧する際は、Web ブラウザが Web サーバの TCP80 番ポートへ接続する。

図 2 では、クライントからサーバの TCP80 番ポートへむけて通信を行っている様子を示している。一般的に、インターネット上の通信ではデータ中に含まれる各層のアドレスをもとに、L2 または L3 スイッチが転送を行う。L2 スイッチのことをスイッチングハブといったり、L3 スイッチのことをルータということもあるが、本書では L2 スイッチ、L3 スイッチと呼ぶことにする。この図が示すように、L2 スイッチ、L3 スイッチによってデータが転送されても、データ中の IP アドレスとポート番号は変わらないが、MAC アドレスは L3 スイッチでの転送時に更新される。これは、MAC アドレスはローカルなネットワーク内でのみ通用するアドレスであり、L3 スイッチはローカルなネットワーク同士をつなぎ合わせる役割を持っているためである。以降の節では、データリンク、ネットワーク、トランスポートの動きについて詳しく説明する。

重要ポイント

- インターネット関連のプロトコルは、IETF が発行する RFC によって標準化されている
- コンピュータネットワークはレイヤで考えることができる
- Ethernet のアドレスは 48 ビットの MAC アドレス、IPv4 のアドレスは 32 ビットの IPv4 アドレス、IPv6 のアドレスは 128 ビットの IPv6 アドレス、TCP と UDP のアドレスは 16 ビットのポート番号

3.2　おもちゃのネットワークスタック

これより本章では、おもちゃのネットワークスタックを用いて、ネットワークスタックの設計と実装を解説する。おもちゃと言っても、実際に IP ルータや Ethernet ブリッジとして動作するれっきとしたネットワー

クスタックである。図 3 はおもちゃのネットワークスタックのデータフロー図を示している。この図の下部には、入力と出力用の Ethernet Input/Ouptput Queue というキューがあり、ここで物理的な入出力が行われる。実際に、ネットワークインターフェースカードには入出力用のキューが用意されており、デバイスドライバはこれらキューに対して読み書きすることでデータの送受信を行う。

なお、このおもちゃのネットワークは、Ethernet ブリッジや、IPv4 のルーティングは行うことができるが、TCP のセッション管理などは行えないし、扱えるのは基本的に IP はユニキャストのみで、IP マルチキャスト通信はサポートしていない。また、実際の OS ではネットワークスタックの上にソケットレイヤが配置され、ネットワークに関する操作が抽象化されているが、おもちゃのネットワークスタックではソケットレイヤは省略されている。すなわち、あくまでも、ネットワークスタックの仕組みから理解してファイアウォールなどを運用するために必要最低限と思われる機能のみが実装されている。

3.3 ネットワークインターフェース

スタックの説明を行う前に、ネットワークインターフェース情報を表すための構造体を説明しよう。ソースコード 1 は、おもちゃのネットワークスタックで定義するネットワークインターフェース用の my_ifnet 構造体となる。

ソースコード 1　ネットワークインターフェースを表す構造体 (my_ifnet.h)

```
1  // インターフェース情報を保持する構造体
2  struct my_ifnet {
3      int idx;                         // インデックス
4      uint8_t ifaddr[6];               // MAC アドレス
5      struct in_addr addr;             // IPv4 アドレス
6      struct in6_addr addr6;           // IPv6 アドレス
7      uint8_t plen;                    // IPv4 プレフィックス長
8      uint8_t plen6;                   // IPv6 プレフィックス長
9      char infile[128];                // 入力UNIX ファイル名
10     char outfile[128];               // 出力UNIX ファイル名
11     int sockfd;                      // 入力先UNIX ドメインソケット
12     struct sockaddr_un outun;        // 出力UNIX アドレス
13     LIST_ENTRY(my_ifnet) pointers; // リスト
14 };
```

my_ifnet 構造体のメンバ変数は基本的にはコメントにあるとおりだが、もう少し詳しく説明したのが表 1 となる。表 1 で示すように、ネットワークインターフェースには各種アドレスが紐付けられる。また、おもちゃのネットワークスタックでは、データの送受信に UNIX ドメインソケットのデータグラム通信を行うため、UNIX ドメインソケット用のデータがいくつが用意されている。

ソースコード 1 はおもちゃのネットワークスタックの割り込みハンドラを表している。割り込みハンドラとは、ネットワークカードにデータが到着した際に呼び出される関数のことを指す。実際の OS では物理的な入力が割り込みハンドラを起動するが、おもちゃのネットワークスタックでは UNIX ドメインソケットへの入力があったときに dev_input 関数を呼び出すようにしている。

ソースコード 2　割り込みハンドラ (my_ifnet.c)

```
1  /*
2   * インターフェース入力割り込み関数
3   * 引数:
4   *   fd: UNIX ドメインソケットへのファイルデスクリプタ
```

表 1　my_ifnet 構造体のメンバ変数

idx	複数インターフェースの番号を識別するためのメンバ変数
addr	インターフェースに対応付けられた IPv4 アドレス
addr6	インターフェースに対応付けられた IPv6 アドレス
plen	IPv4 プレフィックスアドレス（3.7 節にて解説）
plen6	IPv6 プレフィックスアドレス（3.7 節にて解説）
infile	データ受信を行うための UNIX ドメインソケットへのファイル名
outfile	データ送信を行うための UNIX ドメインソケットへのファイル名
sockfd	データ受信用の UNIX ドメインソケットへのデスクリプタ
outun	データ送信用の UNIX ドメインソケットへのアドレス
pointers	複数インターフェースをリストで管理するためのポインタ。sys/queue.h を利用

```
5  */
6  void dev_input(int fd) {
7      for (struct my_ifnet *np = LIST_FIRST(&ifs); np != NULL;
8           np = LIST_NEXT(np, pointers)) {
9          if (np->sockfd == fd) {
10             char buf[4096];
11             ssize_t size;
12         again:
13             size = recv(fd, buf, sizeof(buf), 0);
14             if (size < 0) {
15                 if ((errno) == EAGAIN)
16                     goto again;
17
18                 perror("recv");
19                 break;
20             }
21
22             ether_input(np, (struct ether_header *)buf, size);
23             break;
24         }
25     }
26 }
```

ソースコード 1 では引数に入力用の UNIX ドメインソケットを受け取り、7〜8 行目で対応する UNIX ドメインソケットを持つ my_ifnet 構造体をリストから検索している。入力インターフェースの my_ifnet 構造体が見つかったら（9 行目）、13 行目でデータ読み込みを行っている。14〜20 行目エラー処理で、読み込みに失敗した場合は errno が EAGAIN であれば、再度読み直しそれ以外であれば読み込み失敗として関数を抜ける。データを読み込んだ後、22 行目で ether_input 関数を呼び出して実際の処理に入る。これは図 3 で示される、Ethernet Input Queue から ether_input 関数へのデータフローに相当する。

重要ポイント

- ネットワークインターフェースカードにデータが到着した際に、OS で設定した割り込みハンドラと呼ばれる関数が呼び出される
- 割り込みハンドラから、実際にネットワーク処理を行うための関数が呼ばれる

3.4 データリンク層

ソースコード 3 は Ethernet (IEEE 802.3) プロトコルのヘッダ構造体を示している。我々が普段利用している無線や有線の Ethernet では、内部的にはこのようなフォーマットのヘッダがデータの先頭に付与され、その後に IP ヘッダ、TCP ヘッダなどのより上位のヘッダが続き、最後にアプリケーションデータが続く。もう少し正確に言うと、ソースコード 3 で示す Ethernet ヘッダの前にプリアンブルなどのハードウェアで利用されるデータが続くが、本書ではその説明は割愛する。

ソースコード 3　Ethernet プロトコルヘッダ定義 (/usr/include/net/ethernet.h)

```
1  #define ETHER_ADDR_LEN  6       /* Ethernet address length             */
2
3  /*
4   * The length of the combined header.
5   */
6  struct  ether_header {
7          u_int8_t  ether_dhost[ETHER_ADDR_LEN];
8          u_int8_t  ether_shost[ETHER_ADDR_LEN];
9          u_int16_t ether_type;
10 };
11
12 #define ETHERTYPE_IP            0x0800  /* IP protocol */
13 #define ETHERTYPE_ARP           0x0806  /* Addr. resolution protocol */
14 #define ETHERTYPE_IPV6          0x86dd  /* IPv6 */
```

ソースコード 3 で示されるように、Ethernet ヘッダの構造体は、OpenBSD では、/usr/include/net/ethernet.h にて定義されている。なお、以降特に断りがない限り対象とする OS は OpenBSD とし、/usr/include のパスから始まるソースコードは、OS が提供するソースコードであるとする。1 行目の ETHER_ADDR_LEN では、Ethernet アドレス（MAC アドレス）のバイト数を 6 バイトと定義している。6 行目以降が Ethernet ヘッダを示す ether_header 構造体となる。ether_header 構造体では、ether_dhost と ether_shost というメンバ変数を持ち、それぞれ宛先 MAC アドレスと送信元 MAC アドレスを示している。ether_type メンバ変数は、Ethernet ヘッダ以降に続くプロトコル種類を示している。

ether_type メンバ変数で利用できる値は IANA によって定義されている [5]。例えば、IPv4 が続く場合は 16 進数表記で 0x0800 という値が ether_type に格納される。他には、IPv6 の場合は 0x08DD、仮想的な LAN を構築するための IEEE 802.1Q VLAN プロトコルの場合は 0x8100 が格納される。この値は ether_header 構造体と同じファイルにて定義されており、ソースコード 3 に 12〜14 行目に一部抜粋してある。ただし、ether_type メンバ変数のバイトオーダはビッグエンディアンであるため、比較や格納する際はバイトオーダを変換してから行わなければならない。

ソースコード 4 はおもちゃのネットワークスタックの Ethernet フレームを受け取り処理を行う ethernet_input 関数である。この関数では、引数に入力インターフェースを指す my_ifnet 構造体のポインタ、入力 Ethernet フレームへのポインタ、フレーム長をとり、Ethernet ヘッダ中のプロトコルタイプに応じて上位レイヤの関数に渡している。

ソースコード 4　ether_input 関数 (ether.c)

```
1  // ADDR が IPv4 ブロードキャストアドレスなら真、それ以外なら偽を返すマクロ
2  #define IS_BROADCAST(ADDR)                                              \
3      (((ADDR)[0] == 0xFF) && ((ADDR)[1] == 0xFF) && ((ADDR)[2] == 0xFF) &&      \
```

```
4       ((ADDR)[3] == 0xFF) && ((ADDR)[4] == 0xFF) && ((ADDR)[5] == 0xFF))
5
6   /*
7    * Ethernet フレーム入力関数
8    * 引数:
9    *   ifp: 入力インターフェース
10   *   eh: 入力フレーム
11   *   len: 入力フレーム長
12   */
13  void ether_input(struct my_ifnet *ifp, struct ether_header *eh, int len) {
14      printf("ether_input:\n");
15      printf("␣␣␣␣IF#:␣%d\n", ifp->idx);
16      printf("␣␣␣␣SRC␣MAC:␣%02X-%02X-%02X-%02X-%02X-%02X\n", eh->ether_shost[0],
17              eh->ether_shost[1], eh->ether_shost[2], eh->ether_shost[3],
18              eh->ether_shost[4], eh->ether_shost[5]);
19      printf("␣␣␣␣DST␣MAC:␣%02X-%02X-%02X-%02X-%02X-%02X\n", eh->ether_dhost[0],
20              eh->ether_dhost[1], eh->ether_dhost[2], eh->ether_dhost[3],
21              eh->ether_dhost[4], eh->ether_dhost[5]);
22      printf("\n");
23
24      if (IS_BROADCAST(eh->ether_dhost)) {
25          // ブロードキャストアドレスの場合、ブリッジ処理へ
26          if (IS_L2BRIDGE)
27              bridge_input(ifp, eh, len);
28      } else if (memcmp(ifp->ifaddr, eh->ether_dhost, ETHER_ADDR_LEN) != 0) {
29          // 宛先MACアドレスが自インターフェース宛でないならブリッジ処理を行い終了
30          if (IS_L2BRIDGE)
31              bridge_input(ifp, eh, len);
32          return;
33      }
34
35      switch (ntohs(eh->ether_type)) {
36      case ETHERTYPE_IP: // IPv4 入力
37          ipv4_input((struct ip *)((uint8_t *)eh + ETHER_HDR_LEN));
38          break;
39      case ETHERTYPE_IPV6: // IPv6 入力
40          ipv6_input((struct ip6_hdr *)((uint8_t *)eh + ETHER_HDR_LEN));
41          break;
42      case ETHERTYPE_ARP: // ARP 入力
43          arp_input(ifp, (struct arphdr *)((uint8_t *)eh + ETHER_HDR_LEN));
44          break;
45      default:
46          printf("eh->ether_type␣is␣neither␣IPv6␣nor␣IPv6\n");
47          return;
48      }
49
50      return;
51  }
```

14〜22 行目では入力 Ethernet フレームの送信元と宛先 MAC アドレスを表示している。24 行目では、宛先がブロードキャストアドレスかチェックしている。すなわち、FF-FF-FF-FF-FF-FF という MAC アドレスが Ethernet のブロードキャストアドレスであるため、この値かどうかを、IS_BROADCAST マクロで判定している。宛先がブロードキャストアドレスの場合かつ、L2 ブリッジが有効であるなら、L2 ブリッジ処理を行う bridge_input 処理を行い、自身のネットワークスタック入力処理へと進む。28 行目では、宛先が受信したネットワークインターフェースの MAC アドレスと同じであるか（すなわち自分宛てであるか）を

チェックし、自分宛てで無いならば、L2 ブリッジが有効の場合に L2 ブリッジ処理を行う関数へデータを渡し、ether_input 処理を終了する。L2 ブリッジが有効かどうかは、IS_L2BRIDGE というマクロで判定する。

35 行目から始まる switch 文では、上位のレイヤのプロトコルタイプを判別して、対応するプロトコルの関数に渡している。おもちゃのネットワークスタックでは、IPv4 のみに対応しているが、例のために IPv6 用のダミー関数も用意している。また、IPv4 で通信を行うためには、アドレス解決プロトコル（Address Resolution Protocol、ARP） [6] というプロトコルで MAC アドレスと IPv4 アドレスの対応の解決を行わなければならない。そのため、おもちゃのネットワークスタックでも ARP をサポートしている。

なお、ETHERTYPE_IP、ETHERTYPE_IPV6、ETHERTYPE_ARP といった定義は、ソースコード 3 で示したように、/usr/include/net/ethernet.h で定義されている。35 行目では ntohs という関数を利用するが、これは 2 バイト変数のバイトオーダをホストバイトオーダ（ホスト CPU に依存）からからネットワークバイトオーダ（ビッグエンディアン）に変換する標準 C ライブラリ関数となる。

重要ポイント

- 入力インターフェースの MAC アドレスと、Ethernet ヘッダ中の宛先 MAC アドレスを比較して、自身宛ての Ethernet フレームか判別する
- ブロードキャスト MAC アドレス（FF-FF-FF-FF-FF-FF）の場合は自身宛てと判別する
- Ethernet ヘッダ中のプロトコルタイプフィールドを判別して、IPv4、IPv6、ARP など上位層のプロトコル種別を判別する

3.5 L2 ブリッジ

3.6 アドレス解決プロトコル (ARP)

Ethernet での通信は MAC アドレスベースで行われる、より上位層の IP は MAC アドレスではなく IP アドレスで通信を行う。したがって、IP パケットを Ethernet フレームで転送するためには、どの MAC アドレスがどの IP アドレスに対応しているかを知らなければならない。ARP は、IPv4 アドレスから MAC アドレスの対応を得るために使うプロトコルである。

図 4 は ARP の動作を示した図となる。この図では 3 つのホスト A、B、C があり、A が B へと通信するために ARP で IPv4 アドレスと MAC アドレスの対応の解決を行っている。それぞれのホストは ARP テーブルと呼ばれる、IPv4 アドレスと MAC アドレスの対応を保存しておくテーブルを持っており、初期状態では ARP テーブルは空である。

ホスト A がホスト B に通信を行うために、ホスト A は、まず ARP リクエストと呼ばれるパケットを Ethernet ブロードキャストでリンク内の全てのノードに送信する（ただしここで、ホスト A はホスト B の IPv4 アドレスを知っているとする)。これは図 4 の上部に赤の破線で示されるパケットであり、この ARP リクエストには、192.168.0.2 という IPv4 アドレスを持っているノードは誰かを問い合わせている。ARP リクエストを受け取ったホスト B、C は自身の ARP テーブルにホスト A の IPv4 アドレスと MAC アドレスを記憶しておく。すると、図 4 の下部に示されるように、ホスト B、C の ARP テーブルが更新される。

ARP リクエストを受け取ったホスト B は、リクエスト先の IPv4 アドレスが自身のアドレスと一致するため、ARP リプライをホスト A に送信する。ARP リプライを受け取ったホスト A は、自身の ARP テーブルにホスト B の情報を追加して、その後、実際に IPv4 アドレスでの通信を開始する。

ARP ヘッダ構造体と Ethernet で用いる ARP 構造体は、ソースコード 5 と 6 に示すように、/usr/in-

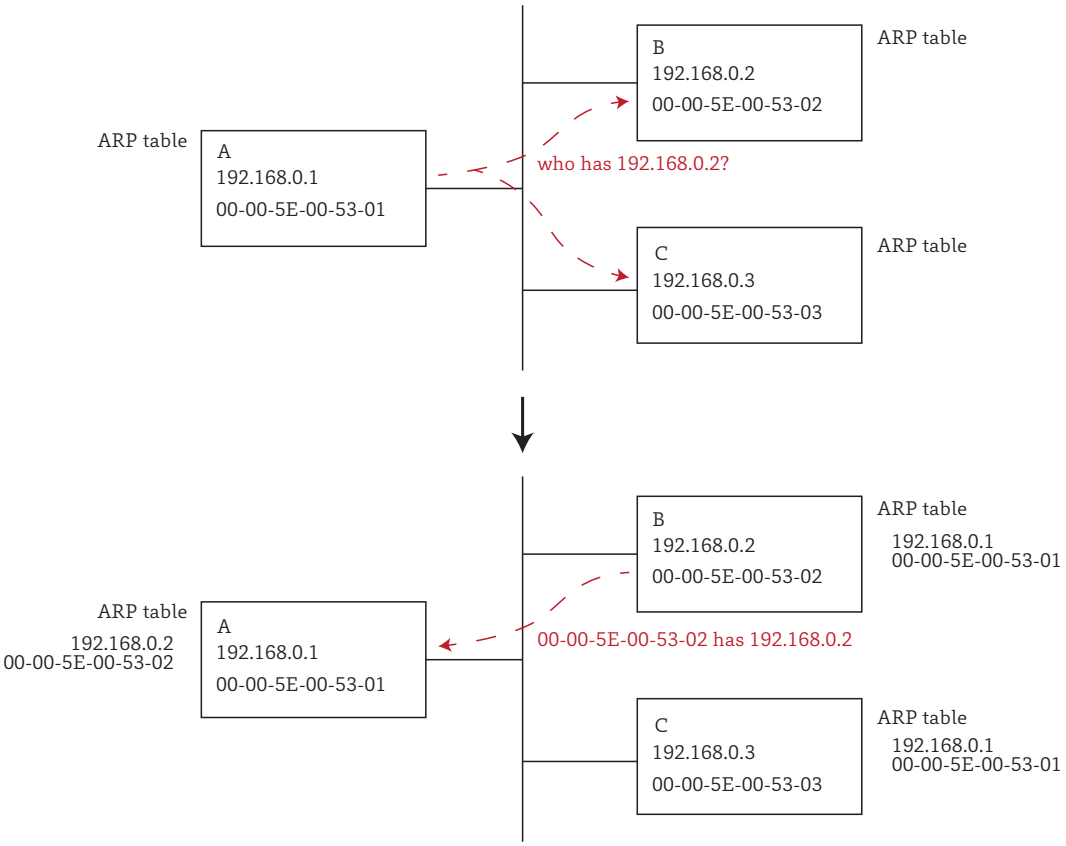

図 4　ARP の動作図

clude/net/if_arp.h と/usr/include/netinet/if_ether.h にて定義されている。

ソースコード 5　ARP 構造体 (/usr/include/net/if_arp.h)

```
1  /*
2   * Address Resolution Protocol.
3   *
4   * See RFC 826 for protocol description.  ARP packets are variable
5   * in size; the arphdr structure defines the fixed-length portion.
6   * Protocol type values are the same as those for 10 Mb/s Ethernet.
7   * It is followed by the variable-sized fields ar_sha, arp_spa,
8   * arp_tha and arp_tpa in that order, according to the lengths
9   * specified.  Field names used correspond to RFC 826.
10  */
11 struct  arphdr {
12         u_int16_t ar_hrd;       /* format of hardware address */
13 #define ARPHRD_ETHER     1      /* ethernet hardware format */
14 #define ARPHRD_IEEE802   6      /* IEEE 802 hardware format */
15 #define ARPHRD_FRELAY    15     /* frame relay hardware format */
16 #define ARPHRD_IEEE1394 24      /* IEEE 1394 (FireWire) hardware format */
17         u_int16_t ar_pro;       /* format of protocol address */
18         u_int8_t  ar_hln;       /* length of hardware address */
19         u_int8_t  ar_pln;       /* length of protocol address */
20         u_int16_t ar_op;        /* one of: */
21 #define ARPOP_REQUEST    1      /* request to resolve address */
22 #define ARPOP_REPLY      2      /* response to previous request */
```

表 2　arphdr 構造体のメンバ変数（全てビッグエンディアン）

ar_hrd	L2 プロトコル識別子。Ethernet の場合は 1
ar_pro	L3 プロトコル識別子。IPv4 の場合は ETHERTYPE_IP
ar_hln	L2 アドレスのバイト数。MAC アドレスの場合は 6 バイト
ar_pln	L3 アドレスのバイト数。IPv4 アドレスの場合は 4 バイト
ar_op	ARP の種類。リクエストの場合は 1 で、リプライの場合は 2

```
23 #define ARPOP_REVREQUEST 3      /* request protocol address given hardware */
24 #define ARPOP_REVREPLY  4       /* response giving protocol address */
25 #define ARPOP_INVREQUEST 8      /* request to identify peer */
26 #define ARPOP_INVREPLY  9       /* response identifying peer */
27 /*
28  * The remaining fields are variable in size,
29  * according to the sizes above.
30  */
31 #ifdef COMMENT_ONLY
32         u_int8_t  ar_sha[];     /* sender hardware address */
33         u_int8_t  ar_spa[];     /* sender protocol address */
34         u_int8_t  ar_tha[];     /* target hardware address */
35         u_int8_t  ar_tpa[];     /* target protocol address */
36 #endif
37 };
```

ソースコード 6　Ethernet 用 ARP 構造体 (/usr/include/netinet/if_ether.h)

```
1  /*
2   * Ethernet Address Resolution Protocol.
3   *
4   * See RFC 826 for protocol description.  Structure below is adapted
5   * to resolving internet addresses.  Field names used correspond to
6   * RFC 826.
7   */
8  struct  ether_arp {
9          struct  arphdr ea_hdr;  /* fixed-size header */
10         u_char  arp_sha[ETHER_ADDR_LEN];        /* sender hardware address */
11         u_char  arp_spa[4];     /* sender protocol address */
12         u_char  arp_tha[ETHER_ADDR_LEN];        /* target hardware address */
13         u_char  arp_tpa[4];     /* target protocol address */
14 }
```

表 2 は arphdr 構造体のメンバ変数を説明した表となる。基本的に、ar_op 変数以外の値は Ethernet と IPv4 を扱うときは表で示した値で固定となる。ただし、構造体定義からもわかるように、ARP は Ethernet や IPv4 に限らず様々なプロトコルで利用可能な設計となっている。ether_arp 構造体のメンバ変数は、MAC アドレスと IPv4 アドレスを保存する変数であることは自明のため詳細は割愛する。

次に、実際におもちゃのネットワークスタックで ARP 処理を行う関数を見ていく。ソースコード 7 は、ARP パケットを受け取り、リクエストやリプライなどそれぞれに対応した関数を呼び出す arp_input 関数である。

ソースコード 7　arp_input 関数

```
1  /*
```

```
 2  * ARP リクエスト及び応答を受け取る関数
 3  * 引数:
 4  *   ifp: 入力インターフェース
 5  *   arph: 入力ARP
 6  */
 7 void arp_input(struct my_ifnet *ifp, struct arphdr *arph) {
 8     // Ethernet 以外は未対応
 9     if (ntohs(arph->ar_hrd) != ARPHRD_ETHER || arph->ar_hln != ETHER_ADDR_LEN)
10         return;
11
12     // IP 以外は未対応
13     if (ntohs(arph->ar_pro) != ETHERTYPE_IP ||
14         arph->ar_pln != sizeof(struct in_addr))
15         return;
16
17     switch (ntohs(arph->ar_op)) {
18     case ARPOP_REQUEST:
19         // ARP リクエストを受け取り応答
20         arp_req_input(ifp, arph);
21         return;
22     case ARPOP_REPLY:
23         // リプライを受け取って送信バッファ中のフレームを送信
24         arp_reply_input(ifp, arph);
25         return;
26     default:
27         // ARP リクエストとリプライ以外は未対応
28         return;
29     }
30 }
```

arp_input 関数は引数に入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、入力 ARP パケットをさす arphdr 構造体へのポインタ変数である arph をとる。9、10 行目で L2 プロトコルと L2 プロトコルアドレスの長さをチェックし、Ethernet かつ 6 バイト意外であるなら、未対応として処理を終了する。13〜15 行目では L3 プロトコルが IPv4 であるかをチェックし、そうでないなら処理を終了する。17 行目で ARP プロトコルの種類を判別し、ARP リクエストであれば 20 行目で arp_req_input 関数を呼び出し、ARP リプライであれば arp_reply_input 関数を呼び出す。ただし、おもちゃのネットワークスタックでは ARP リクエストとリプライ以外は未対応であるため、これら以外の ARP パケットが来た場合は何もせずに処理を終了する。

ソースコード 8 は ARP リクエストを送信するための arp_req 関数である。図 4 上部のホスト A は、この arp_req 関数を用いて ARP リクエストを送信する。

ソースコード 8　arp_req 関数

```
 1 /*
 2  * ARP リクエストを送信
 3  * 引数:
 4  *   ifp: 送信を行うインターフェース
 5  *   addr: 問い合わせを行う IP アドレスへのポインタ
 6  */
 7 void arp_req(struct my_ifnet *ifp, struct in_addr *addr) {
 8     uint8_t buf[ETHER_HDR_LEN + sizeof(struct ether_arp)];
 9     struct ether_header *eh = (struct ether_header *)buf;
10     struct ether_arp *req = (struct ether_arp *)(buf + ETHER_HDR_LEN);
11
```

```
12      // Ethener ヘッダ設定
13      memcpy(eh->ether_shost, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元MAC は自分
14      memset(eh->ether_dhost, 0xff, ETHER_ADDR_LEN); // 宛先MAC はブロードキャスト
15      eh->ether_type = htons(ETHERTYPE_ARP);
16
17      // ARP ヘッダ設定
18      req->ea_hdr.ar_hrd = ntohs(ARPHRD_ETHER); // ハードウェアタイプ（Ethernet）
19      req->ea_hdr.ar_pro = ntohs(ETHERTYPE_IP); // プロトコルタイプ（IP）
20      req->ea_hdr.ar_hln = ETHER_ADDR_LEN; // 6バイト。MAC アドレスサイズ
21      req->ea_hdr.ar_pln = sizeof(struct in_addr); // 4バイト。IPv4 アドレスサイズ
22      req->ea_hdr.ar_op = ntohs(ARPOP_REQUEST); // ARP リクエスト
23
24      // ARP リクエスト設定
25      memcpy(req->arp_sha, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元MAC は自分
26      memcpy(req->arp_spa, &ifp->addr, sizeof(req->arp_spa)); // 送信元IP は自分
27      memset(req->arp_tha, 0xff, ETHER_ADDR_LEN); // 宛先MAC はブロードキャスト
28      memcpy(req->arp_tpa, addr, sizeof(*addr)); // 問い合わせIP
29
30      ether_output(ifp, eh, ETHER_HDR_LEN + sizeof(struct ether_arp));
31  }
```

arp_req 関数は出力先インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、問い合わせ IPv4 アドレスをさす in_addr 構造体へのポインタ変数である addr を引数にとる。8 行目のでは送信 ARP パケット用のバッファを作成しており、9、10 行目で、Ethernet ヘッダと ARP のためのデータを格納する先のアドレスを計算している。13、14 行目では Ethernet ヘッダ情報を設定しており、送信元アドレスは出力先インターフェースの MAC アドレスとしている。その一方、宛先アドレスは、ネットワーク全体に届くようブロードキャストアドレス（FF-FF-FF-FF-FF-FF）を設定している。また、プロトコルタイプは今回は ARP であるため、ETHERTYPE_ARP を設定している。18〜22 行目では ARP ヘッダの設定をしており、これは前述したとおりである。25〜28 行目では ARP リクエストパケットに IPv4 アドレスと MAC アドレスを設定している。送信元の MAC と IPv4 アドレスは出力先インターフェースのアドレスであり、ターゲットの IPv4 アドレスは問い合わせ IPv4 アドレスである。ただし、問い合わせ MAC アドレスは不明なので、ここではブロードキャストに設定している。ARP パケットに値を設定した後、最後の 30 行目で ethernet_output 関数を呼び出し、ARP パケットを送信する。

ソースコード 9 は ARP リクエストを受け取り ARP リプライを返信する arp_req_input 関数となる。

ソースコード 9　arp_req_input 関数

```
1   /*
2    * ARP リクエストを受け取り、ARP リプライを返す関数
3    * 引数:
4    *   ifp: 受信したインターフェース
5    *   arph: ARP リクエストへのポインタ
6    */
7   static void arp_req_input(struct my_ifnet *ifp, struct arphdr *arph) {
8       struct ether_arp *req = (struct ether_arp *)arph;
9       uint8_t buf[ETHER_HDR_LEN + sizeof(struct ether_arp)];
10      struct ether_header *eh = (struct ether_header *)buf;
11      struct ether_arp *reply = (struct ether_arp *)(buf + ETHER_HDR_LEN);
12
13      char addr[16];
14      inet_ntop(PF_INET, req->arp_tpa, addr, sizeof(addr));
15      printf("ARP: who has %s?\n", addr);
16
```

```
17      // ARP テーブルに追加
18      add2arptable((struct in_addr *)req->arp_spa, req->arp_sha);
19
20      // 問い合わせ IPv4 アドレスが自身の IPv4 アドレスかチェック
21      struct in_addr *tpa = (struct in_addr *)req->arp_tpa;
22      if (tpa->s_addr != ifp->addr.s_addr)
23          return;
24
25      // Ethener ヘッダ設定
26      memcpy(eh->ether_shost, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元 MAC は自分
27      memcpy(eh->ether_dhost, req->arp_sha, ETHER_ADDR_LEN); // 宛先 MAC
28      eh->ether_type = htons(ETHERTYPE_ARP);
29
30      // ARP ヘッダ設定
31      reply->ea_hdr.ar_hrd = ntohs(ARPHRD_ETHER); // ハードウェアタイプ (Ethernet)
32      reply->ea_hdr.ar_pro = ntohs(ETHERTYPE_IP); // プロトコルタイプ (IP)
33      reply->ea_hdr.ar_hln = ETHER_ADDR_LEN; // 6バイト。MAC アドレスサイズ
34      reply->ea_hdr.ar_pln = sizeof(struct in_addr); // 4バイト。IPv4 アドレスサイズ
35      reply->ea_hdr.ar_op = ntohs(ARPOP_REPLY); // ARP リプライ
36
37      // ARP リプライ設定
38      memcpy(reply->arp_sha, ifp->ifaddr, ETHER_ADDR_LEN); // 送信元 MAC は自分
39      memcpy(reply->arp_spa, &ifp->addr, sizeof(reply->arp_spa)); // 送信元 IP は自分
40      memcpy(reply->arp_tha, req->arp_sha, ETHER_ADDR_LEN);          // 宛先 MAC
41      memcpy(reply->arp_tpa, req->arp_spa, sizeof(reply->arp_tpa)); // 質問先 IP
42
43      ether_output(ifp, eh, ETHER_HDR_LEN + sizeof(struct ether_arp));
44  }
```

arp_req_input 関数は引数に、入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、ARP リクエストへのポインタ arph をとる。8 行目では、arph ポインタを ether_arp 構造体へのポインタにキャストしている。9〜11 行目では、ARP リプライ用のバッファを確保し、そこから Ethernet ヘッダと ARP リプライ構造体へのアドレスを計算している。13〜15 行目は ARP リクエストの内容を表示しているのみである。18 行目は ARP リクエストの情報を元に ARP テーブルにデータを追加している。これは図 4 上部でホスト B と C が ARP リクエストを受け取り、下部でホスト B と C が自身の ARP テーブルに情報を追加している動作に相当する。21〜23 行目では、ARP リクエストの問い合わせ IPv4 アドレスが受信したインターフェースの IPv4 アドレスかをチェックし、自身宛でないなら処理を終了する。26〜28 行目では Ethernet ヘッダの設定、31〜35 行目では ARP ヘッダの設定、38〜41 行目では ARP リプライの設定を行っており、最後に 43 行目で ether_output 関数を呼び出て ARP リプライを送信する。

ソースコード 10 は ARP リプライを受け取り、送信バッファで待機中のフレームを送信する関数となる。この送信バッファは、はじめに IPv4 パケットを送信する場合、ARP テーブルに対応する IPv4 パケットがないため必要となる。対応する ARP エントリがない場合、一旦送信バッファに退避してから ARP リクエストを送信して、アドレス解決を行った後に退避しておいた IPv4 パケットを実際に送信する。

ソースコード 10　arp_reply_input 関数

```
1   /*
2    * ARP リプライを受け取り、送信バッファ中のフレームを送信する関数
3    * 引数:
4    *   ifp: 入力インターフェース
5    *   arph:
6    */
```

```
7  static void arp_reply_input(struct my_ifnet *ifp, struct arphdr *arph) {
8      struct ether_arp *rep = (struct ether_arp *)arph;
9
10     char addr[16];
11     inet_ntop(PF_INET, rep->arp_spa, addr, sizeof(addr));
12     printf("ARP: %02X-%02X-%02X-%02X-%02X-%02X has %s\n", rep->arp_sha[0],
13            rep->arp_sha[1], rep->arp_sha[2], rep->arp_sha[3], rep->arp_sha[4],
14            rep->arp_sha[5], addr);
15
16     // ARP テーブルに追加
17     add2arptable((struct in_addr *)rep->arp_spa, rep->arp_sha);
18
19     // 送信バッファ中のフレームを送信
20     struct sendbuf *np;
21     for (np = sbuf.lh_first; np != NULL;) {
22         if (np->eh->ether_type == htons(ETHERTYPE_IP)) {
23             // ARP テーブルに宛先 IPv4 アドレスがあるか検索
24             struct ip2mac *mac = find_mac(&np->nextip);
25             if (mac == NULL) {
26                 np = np->pointers.le_next;
27                 continue;
28             }
29
30             // 宛先MAC アドレスを設定
31             memcpy(np->eh->ether_dhost, mac->macaddr, ETHER_ADDR_LEN);
32
33             // インターフェースへ出力
34             ether_output(ifp, np->eh, np->ethlen);
35
36             // バッファを解放
37             free(np->eh);
38             struct sendbuf *tmp = np->pointers.le_next;
39             LIST_REMOVE(np, pointers);
40             free(np);
41             np = tmp;
42         }
43     }
44 }
```

引数はこれまでと同じく、入力インターフェースをさす my_ifnet 構造体へのポインタ変数である ifp と、ARP リプライへのポインタ arph である。10〜14 行目は受け取った ARP リプライを標準出力へ出力している。17 行目は受け取った ARP リプライの情報を ARP テーブルに追加している。21 行目以降が送信バッファ中にある IPv4 パケットを送信する処理となる。21 行目で送信バッファを走査して IPv4 パケットを取り出し、24 行目で取り出した IPv4 パケットの宛先 IPv4 アドレスが解決されているかを検索し、検索が失敗した場合は次のパケットへ処理を移す（25〜27 行目）。31〜34 行目で、宛先 MAC アドレスを設定し Ethernet フレームを送信している。37 行目以降は、送信済みのバッファを解放する処理となる。

重要ポイント

- ARP を利用して IPv4 アドレスと MAC アドレスの解決が行われる
- OS 内部には IPv4 アドレスと MAC アドレスの対応を記録した ARP テーブルが存在する
- ARP テーブルは ARP パケットは受信したタイミングで ARP テーブルを更新される

3.7 ネットワーク層

ソースコード 11 IPv4 ヘッダ定義 (/usr/include/netinet/ip.h)

```
/*
 * Structure of an internet header, naked of options.
 */
struct ip {
#if _BYTE_ORDER == _LITTLE_ENDIAN
        u_int     ip_hl:4,              /* header length */
                  ip_v:4;               /* version */
#endif
#if _BYTE_ORDER == _BIG_ENDIAN
        u_int     ip_v:4,               /* version */
                  ip_hl:4;              /* header length */
#endif
        u_int8_t  ip_tos;               /* type of service */
        u_int16_t ip_len;               /* total length */
        u_int16_t ip_id;                /* identification */
        u_int16_t ip_off;               /* fragment offset field */
#define IP_RF 0x8000                    /* reserved fragment flag */
#define IP_DF 0x4000                    /* dont fragment flag */
#define IP_MF 0x2000                    /* more fragments flag */
#define IP_OFFMASK 0x1fff               /* mask for fragmenting bits */
        u_int8_t  ip_ttl;               /* time to live */
        u_int8_t  ip_p;                 /* protocol */
        u_int16_t ip_sum;               /* checksum */
        struct    in_addr ip_src, ip_dst; /* source and dest address */
};
```

ソースコード 12 IPv6 アドレス構造体 (/usr/include/netinet6/in6.h)

```
/*
 * IPv6 address
 */
struct in6_addr {
        union {
                u_int8_t   __u6_addr8[16];
                u_int16_t  __u6_addr16[8];
                u_int32_t  __u6_addr32[4];
        } __u6_addr;                    /* 128-bit IP6 address */
};
```

ソースコード 13 IPv6 ヘッダ定義 (/usr/include/netinet/ip6.h)

```
/*
 * Definition for internet protocol version 6.
 * RFC 2460
 */

struct ip6_hdr {
        union {
                struct ip6_hdrctl {
                        u_int32_t ip6_un1_flow; /* 20 bits of flow-ID */
                        u_int16_t ip6_un1_plen; /* payload length */
                        u_int8_t  ip6_un1_nxt;  /* next header */
```

```
12                      u_int8_t  ip6_un1_hlim; /* hop limit */
13              } ip6_un1;
14              u_int8_t ip6_un2_vfc;   /* 4 bits version, top 4 bits class */
15      } ip6_ctlun;
16      struct in6_addr ip6_src;        /* source address */
17      struct in6_addr ip6_dst;        /* destination address */
18 } __packed;
```

3.8 トランスポート層

ソースコード 14　TCP ヘッダ定義 (/usr/include/netinet/tcp.h)

```
1  typedef u_int32_t tcp_seq;
2
3  /*
4   * TCP header.
5   * Per RFC 793, September, 1981.
6   */
7  struct tcphdr {
8          u_int16_t th_sport;             /* source port */
9          u_int16_t th_dport;             /* destination port */
10         tcp_seq   th_seq;               /* sequence number */
11         tcp_seq   th_ack;               /* acknowledgement number */
12 #if _BYTE_ORDER == _LITTLE_ENDIAN
13         u_int32_t th_x2:4,              /* (unused) */
14                   th_off:4;             /* data offset */
15 #endif
16 #if _BYTE_ORDER == _BIG_ENDIAN
17         u_int32_t th_off:4,             /* data offset */
18                   th_x2:4;              /* (unused) */
19 #endif
20         u_int8_t  th_flags;
21 #define TH_FIN    0x01
22 #define TH_SYN    0x02
23 #define TH_RST    0x04
24 #define TH_PUSH   0x08
25 #define TH_ACK    0x10
26 #define TH_URG    0x20
27 #define TH_ECE    0x40
28 #define TH_CWR    0x80
29         u_int16_t th_win;                       /* window */
30         u_int16_t th_sum;                       /* checksum */
31         u_int16_t th_urp;                       /* urgent pointer */
32 };
```

ソースコード 15　UDP ヘッダ定義 (/usr/include/netinet/udp.h)

```
1  /*
2   * Udp protocol header.
3   * Per RFC 768, September, 1981.
4   */
5  struct udphdr {
6          u_int16_t uh_sport;             /* source port */
7          u_int16_t uh_dport;             /* destination port */
8          u_int16_t uh_ulen;              /* udp length */
```

```
9           u_int16_t uh_sum;               /* udp checksum */
10  };
```

3.9　トランスポートより上の層

4　PF (Packet Filter) の基礎

5　パケットフィルタリング

6　攻撃手法と対策

7　DMZ（DeMilitarized Zone）の構築

8　検疫ネットワークの構築

9　ロードバランス

10　ロギング

付録 A　Vagrant による実験環境の構築

付録 B　PF の構文

参考文献

[1] Eric M Hutchins, Michael J Cloppert, and Rohan M Amin. Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. *Leading Issues in Information Warfare & Security Research*, Vol. 1, p. 80, 2011.

[2] B. Fraser. Site security handbook, September 1997. RFC2196.

[3] J. Postel. Internet protocol, September 1981. RFC0791.

[4] Internet Assigned Number Authority (IANA). Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[5] Internet Assigned Number Authority (IANA). IEEE 802 Numbers. https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[6] J. Postel. Echo protocol, May 1983. RFC0862.