计算机网络实验 10 报告

学号　2017K8009929059
姓名　於修远

网络地址转换实验

一、实验内容

- 补全实验代码，实现对 SNAT 和 DNAT 的处理函数，并编写配置读取函数。
- 运行给定网络拓扑，用两个私网节点访问公网节点的 http 服务，验证 SNAT 功能。
- 运行给定网络拓扑，用公网节点访问两个私网节点的 http 服务，验证 DNAT 功能。
- 构建包含两个 NAT 的网络拓扑，用私网节点穿透两个 NAT 后访问另一私网节点的 http 服务。

二、实验流程

（一）配置信息读取

1、私网公网端口配置

建立读取缓冲区后打开文件。由于使用 CLion 作为调试工具，可能会出现配置文件路径不匹配的情况，增加提示执行文件路径信息的报错。

```
char line[256];
FILE *fp = fopen(filename,  modes: "rb");
if (fp == NULL) {
    printf( format: "Open fail errno = %d. reason = %s \n", errno, strerror(errno));
    char buf[1024];
    printf( format: "Working path : %s\n", getcwd(buf,  size: 1024));
}
```

按行读取配置文件，使用` sscanf `函数分割端口类型和端口名。匹配端口类型后根据端口名检索端口，绑定为 NAT 的公网端口或私网端口。

```
char type[128], name[128], exter[64], inter[64];
while (!feof(fp) && !ferror(fp)) {
    strcpy(line,  src: "\n");
    fgets(line, sizeof(line), fp);
    if (line[0] == '\n') break;
    sscanf(line,  format: "%s %s", type, name);
    type[14] = '\0';
    if (strcmp(type, "internal-iface") == 0) {
        printf( format: "[Internal] Loading iface item : %s .\n", name);
        nat.internal_iface = if_name_to_iface(name);
    } else if (strcmp(type, "external-iface") == 0) {
        printf( format: "[External] Loading iface item : %s .\n", name);
        nat.external_iface = if_name_to_iface(name);
    } else printf( format: "[Unknown] Loading failed : %s .\n", type);
}
```

2、DNAT 规则配置

在之前的循环中跳过空行后进入配置文件的 DNAT 规则部分。按行读取后使用` sscanf `函数分割出规则类型和公网 IP 和端口及私网 IP 和端口。对于非 DNAT 规则的条目直接忽略。

```
u32 ip4, ip3, ip2, ip1, ip;
u16 port;
while (!feof(fp) && !ferror(fp)) {
    strcpy(line, src: "\n");
    fgets(line, sizeof(line), fp);
    if (line[0] == '\n') break;
    sscanf(line, format: "%s %s %s %s", type, exter, name, inter);
    type[10] = '\0';
    if (strcmp(type, "dnat-rules") == 0) {...}
    else printf( format: "[Unknown] Loading failed : %s .\n", type);
}
return 0;
```

对同时包含 IP 地址和端口号的地址信息，首先通过正则匹配分离出端口号，再将 IP 字段分别读取、移位后重组为 32 位 IP 地址。最后，根据地址信息为公网或私网写入 DNAT 的规则条目中，并加入规则队列。

```
printf( format: "[Dnat] Loading rule item : %s to %s.\n", exter, inter);
struct dnat_rule *rule = (struct dnat_rule*)malloc(sizeof(struct dnat_rule));
list_add_tail(&rule->list, &nat.rules);

sscanf(exter, format: "%[^:]:%hu", name, &port);
sscanf(name, format: "%u.%u.%u.%u", &ip4, &ip3, &ip2, &ip1);
ip = (ip4 << 24) | (ip3 << 16) | (ip2 << 8) | (ip1);
rule->external_ip = ip;
rule->external_port = port;
printf( format: "    |---[External] ip : %08x ; port : %hu\n", ip, port);

sscanf(inter, format: "%[^:]:%hu", name, &port);
sscanf(name, format: "%u.%u.%u.%u", &ip4, &ip3, &ip2, &ip1);
ip = (ip4 << 24) | (ip3 << 16) | (ip2 << 8) | (ip1);
rule->internal_ip = ip;
rule->internal_port = port;
printf( format: "    |---[Internal] ip : %08x ; port : %hu\n", ip, port);
```

最终实现的读取反馈如下图所示。

```
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj10# ./nat exp2.conf
DEBUG: find the following interfaces:  n1-eth1 n1-eth0.
Routing table of 2 entries has been loaded.
[Internal] Loading iface item : n1-eth0 .
[External] Loading iface item : n1-eth1 .
[Dnat] Loading rule item : 159.226.39.43:8000 to 10.21.0.1:8000.
   |---[External] ip : 9fe2272b ; port : 8000
   |---[Internal] ip : 0a150001 ; port : 8000
[Dnat] Loading rule item : 159.226.39.43:8001 to 10.21.0.2:8000.
   |---[External] ip : 9fe2272b ; port : 8001
   |---[Internal] ip : 0a150002 ; port : 8000
```

（二）IP 包翻译中转

1、数据包方向判断

数据包方向的判定基于如下事实：来自私网的数据包，从 NAT 的内部端口转入，并从 NAT 的外部端口转出发送向目的地址；来自公网的数据包，从 NAT 的外部端口转入，并以 NAT 外部端口的 IP 地址作为目的地址。根据上述逻辑，提取数据包的源、目的地址并查找路由表，即可进行方向匹配。

```c
static int get_packet_direction(char *packet) {
    // TODO: determine the direction of this packet.
    struct iphdr *iphdr = packet_to_ip_hdr(packet);
    u32 daddr = ntohl(iphdr->daddr);
    u32 saddr = ntohl(iphdr->saddr);
    u32 d_out = longest_prefix_match(daddr)->iface->ip;
    u32 s_out = longest_prefix_match(saddr)->iface->ip;
    u32 int_ip = nat.internal_iface->ip;
    u32 ext_ip = nat.external_iface->ip;
    if (d_out == ext_ip && s_out == int_ip) return DIR_OUT;
    else if (s_out == ext_ip && daddr == ext_ip) return DIR_IN;
    else return DIR_INVALID;
```

2、数据包翻译更新

只有合法数据包会进入翻译阶段。翻译阶段大致可以分为查找已有连接和创建新连接两种策略，后者针对无法查找到已有连接的 TCP。不论采用何种处理方式，都首先需要根据数据包方向确定远端地址和端口号，依此计算已有或新建连接在连接映射表中的哈希索引值。

```c
struct iphdr *iphdr = packet_to_ip_hdr(packet);
u32 daddr = ntohl(iphdr->daddr);
u32 saddr = ntohl(iphdr->saddr);
u32 raddr = (dir == DIR_IN) ? saddr : daddr;
struct tcphdr *tcphdr = packet_to_tcp_hdr(packet);
u16 sport = ntohs(tcphdr->sport);
u16 dport = ntohs(tcphdr->dport);
u16 rport = (dir == DIR_IN) ? sport : dport;
u8 idx = remote2hash(raddr, rport);
struct list_head *head = &nat.nat_mapping_list[idx];
struct nat_mapping *entry;
```

如前所述，之后需要先遍历已建立连接表。若查找失败，首先检查该数据包是否为 SYN 包，即是否为建立连接的 TCP 包。若不然，与无法查询到 DNAT 规则的 DNAT 包及无法分配新端口的 SNAT 包一并作为无效数据包处理。由于涉及连接列表查询及可能的增删，所以这些操作都需要在互斥锁内完成。

```c
pthread_mutex_lock(&nat.lock);
list_for_each_entry(entry, head, list) {...}
if ((tcphdr->flags & TCP_SYN) == 0) {
    printf( format: "Invalid packet!\n");
    icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
    free(packet);
    pthread_mutex_unlock(&nat.lock);
    return;
}
if (dir == DIR_OUT) {...} else {...}
printf( format: "No available port!\n");
icmp_send_packet(packet, len, ICMP_DEST_UNREACH, ICMP_HOST_UNREACH);
free(packet);
```

查找连接时的匹配条件包括 4 项记录的匹配：远端 IP 和远端端口、内部 IP 和端口号（对于来自内网的数据包）或外部 IP 和端口号（对于来自外网的数据包）。一旦匹配，就将 IP 头部和 TCP 头部的源地址或目的地址修改为映射记录中的剩余两项条目。同时设置更新时点，解析 TCP 数据包并记录连接情况，供老化线程检查。处理完毕后重新计算 TCP 和 IP 部分的校验和，转发数据包即可

```
if (raddr != entry->remote_ip || rport != entry->remote_port) continue;
if (dir == DIR_IN) {
    if (daddr != entry->external_ip || dport != entry->external_port) continue;
    iphdr->daddr = htonl(entry->internal_ip);
    tcphdr->dport = htons(entry->internal_port);
    entry->conn.external_fin = ((tcphdr->flags & TCP_FIN) != 0);
    entry->conn.external_seq_end = tcp_seq_end(iphdr, tcphdr);
    if (tcphdr->flags & TCP_ACK) entry->conn.external_ack = tcphdr->ack;
} else {
    if (saddr != entry->internal_ip || sport != entry->internal_port) continue;
    iphdr->saddr = htonl(entry->external_ip);
    tcphdr->sport = htons(entry->external_port);
    entry->conn.internal_fin = ((tcphdr->flags & TCP_FIN) != 0);
    entry->conn.internal_seq_end = tcp_seq_end(iphdr, tcphdr);
    if (tcphdr->flags & TCP_ACK) entry->conn.internal_ack = tcphdr->ack;
}
entry->update_time = time( timer: NULL);
pthread_mutex_unlock(&nat.lock);
tcphdr->checksum = tcp_checksum(iphdr, tcphdr);
iphdr->checksum = ip_checksum(iphdr);
ip_send_packet(packet, len);
return;
```

建立并保存新的映射关系时，首先考虑私网发起连接的情况。遍历所有可分配给 SNAT 连接的端口（本实验中设置为 12345 到 23456 号端口），找到一个未使用的端口，填写映射信息并记录连接状态后加入哈希索引得到的表项中，生成连接映射条目，并以前述方法修改 IP 头及 TCP 头的源地址及端口信息，更新校验和后转发数据包。具体的条目填充方式可见代码，限于篇幅不附于此。

```
u16 pid;
for (pid = NAT_PORT_MIN; pid <= NAT_PORT_MAX; ++pid) {
    if (!nat.assigned_ports[pid]) {
        struct nat_mapping *new_entry = (struct nat_mapping *) malloc(sizeof(struct nat_mapping));
        list_add_tail(&new_entry->list, head);
```

对于公网发起连接的情况，处理思路大致相同，不同点在于不需要搜索空闲端口并分配，而是直接遍历检索初始化时导入的映射规则。若找到了匹配的条目，则按该条目的映射关系填写连接记录并加入索引，修改数据包并转发。

```
struct dnat_rule *rule;
list_for_each_entry(rule,  head: &nat.rules, list) {
    if (daddr == rule->external_ip && dport == rule->external_port) {
        struct nat_mapping *new_entry = (struct nat_mapping *) malloc(sizeof(struct nat_mapping));
        list_add_tail(&new_entry->list, head);
```

（三）端口映射记录老化及清除

1、老化检查线程

老化线程每隔 1 秒唤醒一次，遍历检查 NAT 的端口映射记录。连接条目老化的判定依据包括两方面：连接超时和连接结束。对于前者，只需计算连接状态最后一次被更新的时点距检查时刻的时间即可，删除超过预设 TIMEOUT（本实验中设置为 60 秒）的条目。对于后者，调用实验提供的` is_flow_finished `函数检查最近一次更新的连接状态是否显示连接已经结束。若是，则清除该条目，并释放端口占用，使之可以被重新使用。

```
while (1) {
    // TODO: sweep finished flows periodically.
    pthread_mutex_lock(&nat.lock);
    struct nat_mapping *entry, *q;
    time_t now = time( timer: NULL);
    for (int i = 0; i < HASH_8BITS; ++i) {
        list_for_each_entry_safe(entry, q,  head: &nat.nat_mapping_list[i], list) {
            if ((now - entry->update_time >= TCP_ESTABLISHED_TIMEOUT) || is_flow_finished(&entry->conn)) {
                nat.assigned_ports[entry->external_port] = 0;
                list_delete_entry(&entry->list);
                free(entry);
            }
        }
    }
    pthread_mutex_unlock(&nat.lock);
    sleep( seconds: 1);
}
```

2、映射表清除

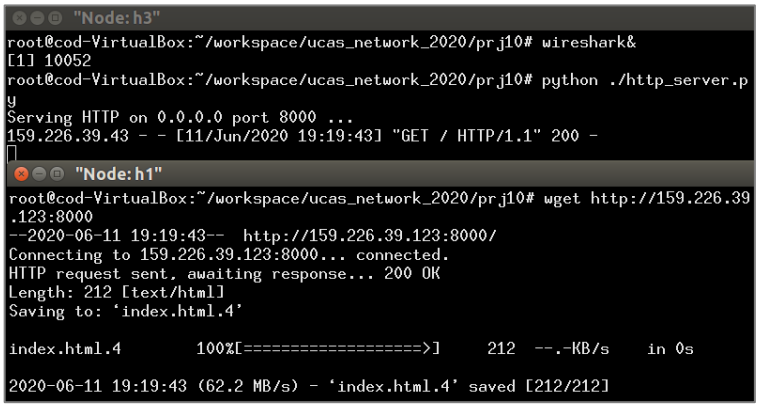
NAT 运行结束时需要调用映射关系清除函数，释放资源。清除函数可视为一个无条件删除条目的老化函数，考虑到重新启动并运行时会重置端口可用性，所以在此不作清零操作。

```
pthread_mutex_lock(&nat.lock);
struct nat_mapping *entry, *q;
for (int i = 0; i < HASH_8BITS; ++i) {
    list_for_each_entry_safe(entry, q,  head: &nat.nat_mapping_list[i], list) {
        list_delete_entry(&entry->list);
        free(entry);
    }
}
pthread_kill(nat.thread, SIGTERM);
pthread_mutex_unlock(&nat.lock);
```

三、结果分析

（一）SNAT 测试

执行` nat_topo.py `脚本后，在 n1 节点上启动 nat 程序，并读入配置文件` exp1.conf `。在 h3 节点运行` http_server.py `脚本建立 http 服务器后，分别从 h1 和 h2 节点执行` wget http://159.226.39.123:8000 `命令访问该服务。两节点的结果相似，以 h1 节点的访问结果为例，如下图所示。可以看到连接正常建立和处理。



通过 wireshark 查看节点 h3 的收发包情况，可以看到 TCP 连接正常建立到关闭的过程。

	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	36:83:d4:03:fa:5c	Broadcast	ARP	42	Who has 159.226.39.123?
2	0.000008613	02:8d:d1:35:f7:f8	36:83:d4:03:fa:5c	ARP	42	159.226.39.123 is at 02:
3	0.000015325	159.226.39.43	159.226.39.123	TCP	74	12345 → 8000 [SYN] Seq=0
4	0.000032566	159.226.39.123	159.226.39.43	TCP	74	8000 → 12345 [SYN, ACK]
5	0.000214283	159.226.39.43	159.226.39.123	TCP	66	12345 → 8000 [ACK] Seq=1
6	0.000411936	159.226.39.43	159.226.39.123	HTTP	212	GET / HTTP/1.1
7	0.000423672	159.226.39.123	159.226.39.43	TCP	66	8000 → 12345 [ACK] Seq=1
8	0.000739447	159.226.39.123	159.226.39.43	TCP	83	8000 → 12345 [PSH, ACK]
9	0.000884749	159.226.39.123	159.226.39.43	HTTP	414	HTTP/1.0 200 OK (text/h
10	0.000979898	159.226.39.43	159.226.39.123	TCP	66	12345 → 8000 [ACK] Seq=1
11	0.001691388	159.226.39.43	159.226.39.123	TCP	66	12345 → 8000 [FIN, ACK]
12	0.001698677	159.226.39.123	159.226.39.43	TCP	66	8000 → 12345 [ACK] Seq=3

（二）DNAT 测试

执行` nat_topo.py `脚本后，在 n1 节点上启动 nat 程序，并读入配置文件` exp2.conf `。在 h1 和 h2 节点运行` http_server.py `脚本建立 http 服务器后，从 h3 节点执行` wget `命令访问 h1 的服务，将端口号改为 8001 以访问 h2 的服务。两节点的结果相似，以访问 h2 节点的结果为例，如下图所示。可见连接正常建立和处理。

```
root@cod-VirtualBox:~/workspace/ucas_network_2020/prj10# wget http://159.226.39
.43:8001
--2020-06-11 19:27:01--  http://159.226.39.43:8001/
Connecting to 159.226.39.43:8001... connected.
HTTP request sent, awaiting response... 200 OK
Length: 208 [text/html]
Saving to: 'index.html.5'

index.html.5        100%[===================>]     208  --.-KB/s    in 0s

2020-06-11 19:27:01 (35.5 MB/s) - 'index.html.5' saved [208/208]
```

通过 wireshark 查看节点 h2 的收发包情况，可以看到 TCP 连接正常建立到关闭的过程。

	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	7e:0b:6a:43:1a:1d	Broadcast	ARP	42	Who has 10.21.0.2? Tell
2	0.000014846	66:03:3b:92:ac:e3	7e:0b:6a:43:1a:1d	ARP	42	10.21.0.2 is at 66:03:3b
3	0.000229496	159.226.39.123	10.21.0.2	TCP	74	52664 → 8000 [SYN] Seq=0
4	0.000245868	10.21.0.2	159.226.39.123	TCP	74	8000 → 52664 [SYN, ACK]
5	0.000411142	159.226.39.123	10.21.0.2	TCP	66	52664 → 8000 [ACK] Seq=1
6	0.000473413	159.226.39.123	10.21.0.2	HTTP	211	GET / HTTP/1.1
7	0.000478524	10.21.0.2	159.226.39.123	TCP	66	8000 → 52664 [ACK] Seq=1
8	0.000813209	10.21.0.2	159.226.39.123	TCP	83	8000 → 52664 [PSH, ACK]
9	0.000956278	159.226.39.123	10.21.0.2	TCP	66	52664 → 8000 [ACK] Seq=1
10	0.000962972	10.21.0.2	159.226.39.123	HTTP	410	HTTP/1.0 200 OK (text/h
11	0.000969947	10.21.0.2	159.226.39.123	TCP	66	8000 → 52664 [FIN, ACK]
12	0.000982393	159.226.39.123	10.21.0.2	TCP	66	52664 → 8000 [ACK] Seq=1
13	0.002317738	159.226.39.123	10.21.0.2	TCP	66	52664 → 8000 [FIN, ACK]
14	0.002331449	10.21.0.2	159.226.39.123	TCP	66	8000 → 52664 [ACK] Seq=3

（三）双 NAT 拓扑测试

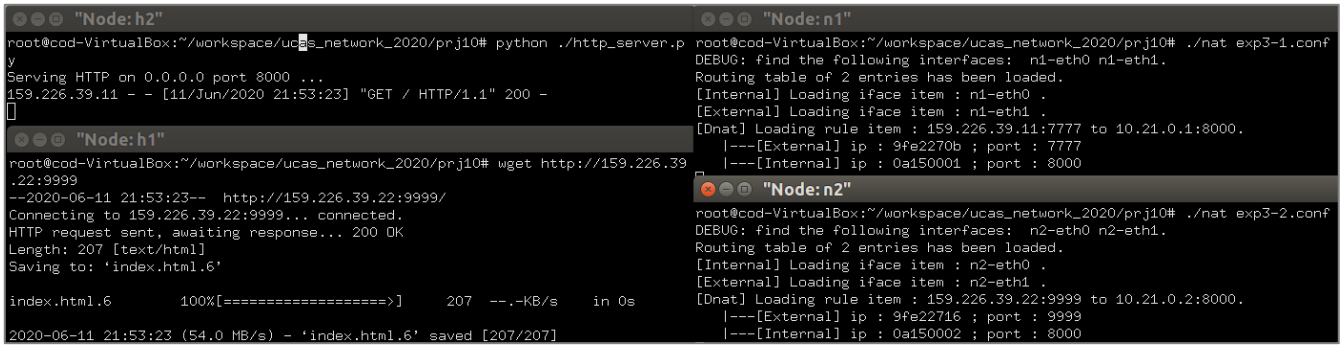
基于` nat_topo.py `构造包含两个主机节点和两个 NAT 节点的网络` h1 – n1 – n2 – h2 `，并设置各端口的 IP 地址如下所示。完整代码实现可见脚本` nat_topo_2.py `。

```
h1.cmd('ifconfig h1-eth0 10.21.0.1/16')
h1.cmd('route add default gw 10.21.0.254')

h2.cmd('ifconfig h2-eth0 10.21.0.2/16')
h2.cmd('route add default gw 10.21.0.254')

n1.cmd('ifconfig n1-eth0 10.21.0.254/16')
n1.cmd('ifconfig n1-eth1 159.226.39.11/24')

n2.cmd('ifconfig n2-eth0 10.21.0.254/16')
n2.cmd('ifconfig n2-eth1 159.226.39.22/24')
```

编写两个 NAT 节点的配置文件，以 n2 为例，设置 eth0 为内部端口，eth1 为外部端口，并设置从 9999 号外部端口到 IP 地址:端口为 10.21.0.2:8000 的内部节点（即 h2 的 eth0 及其默认端口）的映射，如下图所示。节点 n1 的配置文件类似，但用于映射的外部端口号为 7777 号。

```
internal-iface: n2-eth0
external-iface: n2-eth1

dnat-rules: 159.226.39.22:9999 -> 10.21.0.2:8000
```

执行` nat_topo_2.py `脚本后，在 n1 和 n2 节点上启动 nat 程序，并分别读入配置文件` exp3-1.conf `和` exp3-2.conf ` 。在 h2 节点运行` http_server.py `脚本建立 http 服务器后，从 h1 节点执行` wget http://159.226.39.22:9999 `命令访问该服务。更改 h1 和 h2 中的请求方得到的结果相似，以 h1 节点的访问结果为例，如下图所示。可以看到连接正常建立和处理。



（四）思考题

1、NAT 系统对 ICMP 数据包的支持

对 ICMP 数据包进行翻译时，IP 头部部分的地址修改和校验和更新保持不变。

对于 ICMP 的查询或回复类消息，例如 ping，从内网主机发出后，NAT 将数据包中的 identifier 值作为端口来处理，转发时分配的用于连接的端口号作为新的 identifier 值填入。在 NAT 表中，把内部端口号填写为旧 identifier，外部端口号填写为新 identifier。由于这类 ICMP 包的回应包中 identifier 值不变，所以在接收到 ICMP 包时，可以将其 identifier 字段与 NAT 表中的外部端口号进行匹配。匹配成功后根据映射关系，修改其为旧 identifier 并转发，就实现了 ICMP 包的 SNAT 功能。

而对于 ICMP 错误信息，例如 unreachable，无法在上述过程中查找得到映射表项。根据 ICMP 规范，在 ICMP 包内会拷贝原数据包的副本。所以可以通过分析原数据包的内容（包括端口号等信息），将其归类到已建立的连接中，并根据 NAT 表中的信息转发到正确的主机上。

2、NAT 并发连接的扩展

为了扩展 NAT，使其支持超过 65535 个并发连接，可以考虑构建“NAT 网络”。例如，当 k 不超过 65534 时，对于 k+1 个 NAT 元件 N0 和 N1 和 N2……将 N0 的外部端口连接公网，其余每个 NAT 的“外部接口”连接 N0 的内部端口。这样一来，支持的总并发连接数就可以达到 65535*(k+1) – k 的数量。若 k 的数量不小于 65535，可以继续增加 NAT 串联的层数。尽管多层的 NAT 映射可能会造成传输效率的下降，但这样形成的树形结构可以实现 NAT 连接的扩展，且每个 NAT 的内部都是一个相对的内网。