

# 黑客攻防

内核加固

Unit12

# 内容

上午	09:00 ~ 09:30	作业讲解和回顾
	09:30 ~ 10:20	知识讲解
	10:30 ~ 11:20	
	11:30 ~ 12:00	
下午	14:00 ~ 14:50	实训案例
	15:00 ~ 15:50	
	16:00 ~ 16:50	扩展提高
	17:00 ~ 17:30	总结和答疑

# 知识讲解



# 拒绝服务攻击



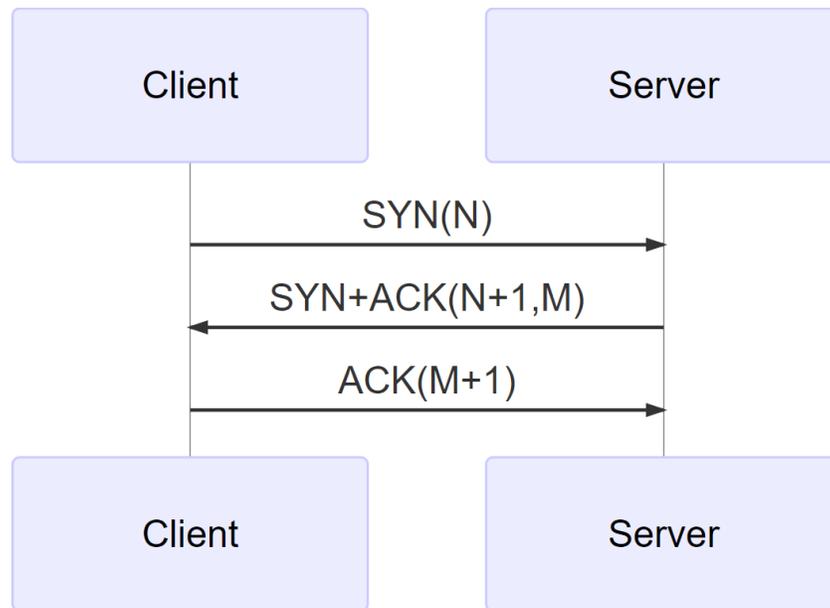
# 拒绝服务攻击

- 拒绝服务(Denial of Service, DoS)攻击是一种简单有效的攻击方式。它通过大量消耗服务器主机的系统资源，阻碍其提供正常网络服务，达到攻击目的
- TCP SYN泛洪攻击是一种典型的拒绝服务攻击，其攻击发起者利用TCP协议漏洞，模拟众多服务请求，使服务器主机疲于奔命，无法响应正常服务请求



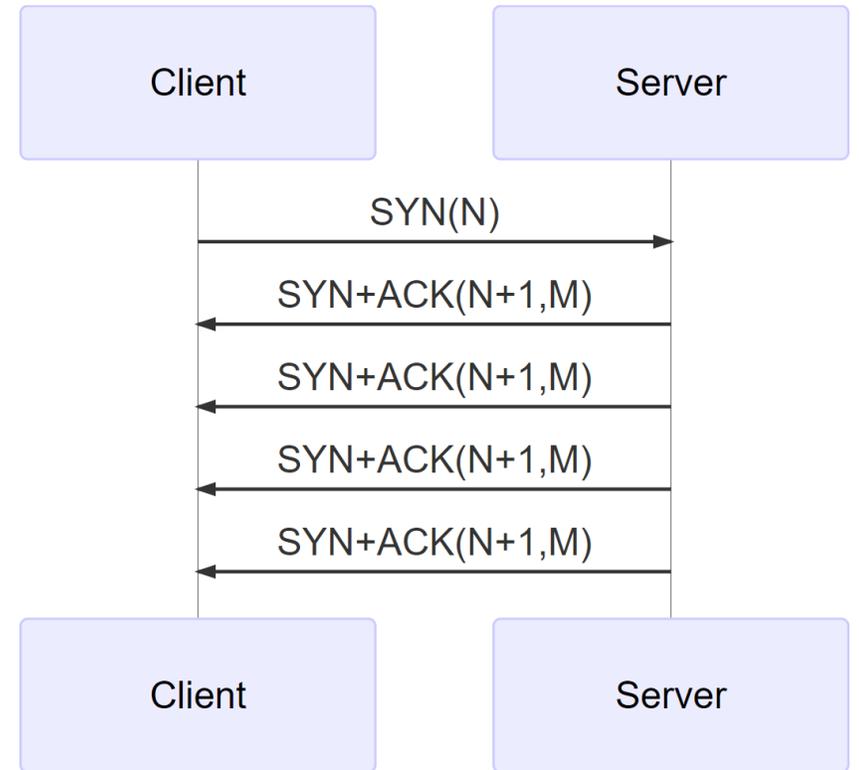
# 拒绝服务攻击

- TCP连接的三路握手过程，如下图所示：
  - 连接建立之前，服务器打开特定的端口并监听
  - 客户端向该端口发送一个TCP数据包，其包头中带有SYN标志位，发送序列号为N
  - 服务器收到该数据包后，向客户端返回一个TCP数据包，其包头中带有SYN和ACK两个标志位，接收序列号为N+1，发送序列号为M
  - 客户端收到该数据包后，再次向服务器发送一个TCP数据包，其包头中带有ACK标志位，接收序列号为M+1
  - 至此三路握手完成，TCP连接建立



# 拒绝服务攻击

- 如果服务器在向客户端发出SYN+ACK(N+1,M)数据包后，未能在给定时间内收到对方的应答，就会认为所发送数据包已丢失，进而重发该数据包。如果重发多次，始终未能收到客户端的应答，服务器才会最终放弃尝试。
  - 在这个过程中，服务器需要记录客户端数据包信息，维护重发定时器，判断是否超时等等，由此造成的资源消耗远大于正常发送数据的消耗



# 拒绝服务攻击

- 如果攻击者不停地向服务器发送大量的孤立SYN数据包，服务器资源将很快消耗殆尽，无法继续提供正常的网络服务
  - 这就是TCP SYN泛洪拒绝服务攻击的基本原理



# 拒绝服务攻击

- 如果攻击者不停地向服务器发送大量的孤立SYN数据包，服务器资源将很快消耗殆尽，无法继续提供正常的网络服务
  - 这就是TCP SYN泛洪拒绝服务攻击的基本原理

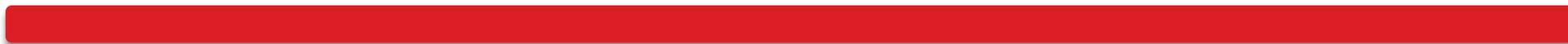


# 拒绝服务攻击

- 如果攻击者不停地向服务器发送大量的孤立SYN数据包，服务器资源将很快消耗殆尽，无法继续提供正常的网络服务
  - 这就是TCP SYN泛洪拒绝服务攻击的基本原理



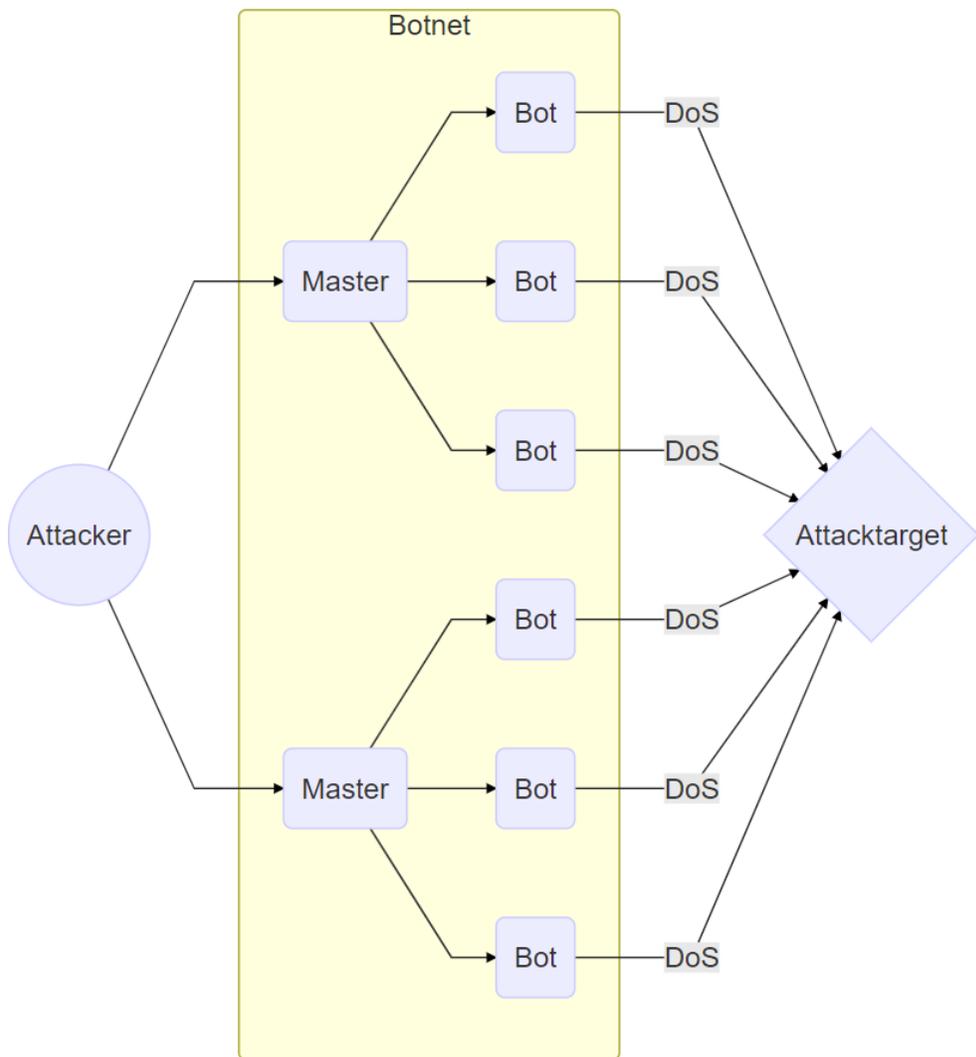
# 僵尸网络



# 僵尸网络

- 与早期由单台主机发起的单兵作战式的拒绝服务攻击不同，近年来渐渐流行起来的分布式拒绝服务(Distributed Denial of Service, DDoS)攻击借助多台被植入攻击木马的傀儡主机，同时向一个目标主机发起集团作战式的拒绝服务攻击，致使被攻击主机遭受巨大的压力，即使是高带宽、高配置的网络服务器也难以幸免
- 目前大部分分布式拒绝服务攻击都是通过僵尸网络(Botnet)实现的。攻击者先将攻击程序部署在僵尸网络的各个被控主机上，在选定攻击目标后，通过僵尸网络向所有被控主机上的攻击程序发送指令，使其同时向攻击目标发起进攻，剧烈消耗目标主机的网络带宽和运算资源，以致其瞬间瘫痪。如下图所示：

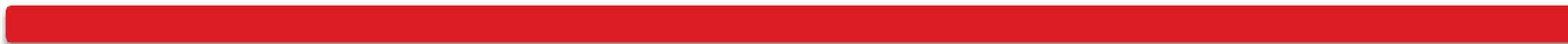
# 僵尸网络



- 事实上，只要僵尸网络的规模足够大，即便每个攻击程序只是发出普通的服务请求，也能大量消耗被攻击主机的系统资源，阻塞其网络带宽，达到拒绝服务的目的



# 内核强化



# 获取内核源码

- 执行如下命令查看内核版本：
  - `uname -r`
    - 4.15.0-47-generic
- 执行如下命令下载特定版本的内核源码包并解压缩：
  - `cd /usr/src`
  - `sudo apt-get install linux-source-4.15.0`
  - `sudo tar -jxvf linux-source-4.15.0.tar.bz2`
- 内核源码放在 `/usr/src/linux-source-4.15.0` 目录下
- 也可以执行如下命令下载特定版本的内核源码：
  - `sudo wget https://www.kernel.org/pub/linux/kernel/v4.x/linux-4.15.1.tar.gz`

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/af\_inet.c:1612

```
static struct net_protocol tcp_protocol = {  
    ...  
    .handler = tcp_v4_rcv,  
    ...  
};
```

- Linux内核通过定义静态全局变量tcp\_protocol注册对TCP协议的处理函数tcp\_v4\_rcv

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:1627

```
int tcp_v4_rcv(...) {  
    ...  
    if (sk->sk_state == TCP_LISTEN) {  
        ret = tcp_v4_do_rcv(...);  
        ...  
    }  
    ...  
}
```

- 对于侦听套接字(`sk->sk_state == TCP_LISTEN`), `tcp_v4_rcv`函数调用 `tcp_v4_do_rcv`函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:1453

```
int tcp_v4_do_rcv(...) {  
    ...  
    if (sk->sk_state == TCP_LISTEN) {  
        struct sock* nsk = tcp_v4_cookie_check(...);  
        ...  
    }  
    ...  
    if (tcp_rcv_state_process(...)) {  
        ...  
    }  
    ...  
}
```

- 对于侦听套接字(`sk->sk_state == TCP_LISTEN`), `tcp_v4_do_rcv`函数先调用 `tcp_v4_cookie_check`函数, 处理客户端为建立连接发来的包头中带有SYN标志位的TCP数据包, 而后调用 `tcp_rcv_state_process`函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:5850

```
int tcp_rcv_state_process(...) {
    ...
    switch (sk->sk_state) {
        ...
        case TCP_LISTEN:
            ...
            if (th->syn) {
                ...
                acceptable = icsk_af_ops->conn_request(...);
                ...
            }
            ...
        }
        ...
    }
    ...
}
```

- 对于侦听套接字(case TCP\_LISTEN)收到的包头中带有SYN标志位(th->syn)的TCP数据包, tcp\_rcv\_state\_process函数通过conn\_request函数指针调用tcp\_v4\_conn\_request函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:1313

```
int tcp_v4_conn_request(...) {  
    ...  
    return tcp_conn_request(...);  
    ...  
}
```

- tcp\_v4\_conn\_request函数调用tcp\_conn\_request函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:6252

```
int tcp_conn_request(...) {
    ...
    if (want_cookie) {
        isn = cookie_init_sequence(...);
        ...
    }
    ...
    if (fastopen_sk) {
        af_ops->send_synack(...);
        ...
    } else {
        ...
        if (!want_cookie)
            inet_csk_reqsk_queue_hash_add(..., tcp_timeout_init(...));
        af_ops->send_synack(...);
        ...
    }
    ...
}
```

- tcp\_conn\_request函数通过send\_synack函数指针调用tcp\_v4\_send\_synack函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:863

```
static int tcp_v4_send_synack(...) {  
    ...  
    skb = tcp_make_synack(...);  
    if (skb) {  
        ...  
        err = ip_build_and_send_pkt(...);  
        ...  
    }  
    ...  
}
```

- tcp\_v4\_send\_synack函数负责构建并返回包头中带有SYN和ACK标志位的TCP包
- 客户端收到该包后，再次向服务器发送一个包头中带有ACK标志位的TCP包
- 服务器的tcp\_v4\_rcv函数再次被调用，该函数调用tcp\_v4\_do\_rcv函数

# 分析内核源码

- 正常建立TCP连接

- /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:1453

```
int tcp_v4_do_rcv(...) {
    ...
    if (sk->sk_state == TCP_LISTEN) {
        struct sock* nsk = tcp_v4_cookie_check(...);
        ...
        if (nsk != sk) {
            if (tcp_child_process(...)) {
                ...
            }
            ...
        }
    }
    ...
}
```

- 对于侦听套接字(`sk->sk_state == TCP_LISTEN`), `tcp_v4_do_rcv`函数先调用 `tcp_v4_cookie_check`函数, 处理客户端发来的包头中带有ACK标志位的TCP数据包, 套接字状态(`sk->sk_state`)被更新为`TCP_SYN_RECV`, 同时返回一个不同于侦听套接字(`sk`)的用于后续通信的新套接字(`nsk != sk`), 然后调用`tcp_child_process`函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_minisocks.c:827

```
int tcp_child_process(...) {  
    ...  
    if (!sock_owned_by_user(child)) {  
        ret = tcp_rcv_state_process(...);  
        ...  
    }  
    ...  
}
```

- tcp\_child\_process函数调用tcp\_rcv\_state\_process函数

# 分析内核源码

- 正常建立TCP连接
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:5850

```
int tcp_rcv_state_process(...) {
    ...
    if (req) {
        ...
        if (!tcp_check_req(...))
            ...
    }
    ...
    switch (sk->sk_state) {
        case TCP_SYN_RECV:
            ...
            tcp_set_state(sk, TCP_ESTABLISHED);
            ...
        ...
    }
    ...
}
```

- tcp\_rcv\_state\_process函数通过tcp\_set\_state函数将套接字状态设置为TCP\_ESTABLISHED, 至此TCP连接建立完成

# 分析内核源码

- 遭受TCP SYN攻击
  - 如前所述，服务器在收到客户端为建立连接发来的包头中带有SYN标志位的TCP数据包后，在tcp\_conn\_request函数中，通过send\_synack函数指针调用tcp\_v4\_send\_synack函数，向客户端返回包头中带有SYN和ACK两个标志位的TCP数据包



# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:6252

```
int tcp_conn_request(...) {
    ...
    if (want_cookie) {
        isn = cookie_init_sequence(...);
        ...
    }
    ...
    if (fastopen_sk) {
        af_ops->send_synack(...);
        ...
    } else {
        ...
        if (!want_cookie)
            inet_csk_reqsk_queue_hash_add(..., tcp_timeout_init(...));
        af_ops->send_synack(...);
        ...
    }
    ...
}
```

- tcp\_conn\_request函数还会调用inet\_csk\_reqsk\_queue\_hash\_add函数，其参数中包含超时

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/inet\_connection\_sock.c:760

```
void inet_csk_reqsk_queue_hash_add(..., unsigned long timeout) {  
    reqsk_queue_hash_req(..., timeout);  
    ...  
}
```

- inet\_csk\_reqsk\_queue\_hash\_add函数调用reqsk\_queue\_hash\_req函数，其参数中包含超时

# 分析内核源码

- 遭受TCP SYN攻击

- /usr/src/linux-source-4.15.0/net/ipv4/inet\_connection\_sock.c:742

```
static void reqsk_queue_hash_req(..., unsigned long timeout) {  
    ...  
    timer_setup(..., reqsk_timer_handler, ...);  
    mod_timer(..., jiffies + timeout);  
    inet_eshash_insert(...);  
    ...  
}
```

- reqsk\_queue\_hash\_req函数先调用timer\_setup函数开启定时器，再调用mod\_timer函数设置超时，最后通过inet\_eshash\_insert函数将半连接套接字加入哈希队列。定时器处理函数reqsk\_timer\_handler将被周期性地调用

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/inet\_connection\_sock.c:676

```
static void reqsk_timer_handler(...) {  
    ...  
    if (!expire && (... !inet_rtx_syn_ack(...) ...)) {  
        ...  
    }  
    ...  
}
```

- reqsk\_timer\_handler函数在超时到期之前(!expire)通过inet\_rtx\_syn\_ack函数向客户端发送包头中带有SYN和ACK两个标志位的TCP数据包
- 当服务器收到来自客户端包头中带有ACK标志位的TCP数据包后，在tcp\_rcv\_state\_process函数中通过tcp\_set\_state函数将套接字状态设置为TCP\_ESTABLISHED，表示TCP连接建立完成

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:5850

```
int tcp_rcv_state_process(...) {
    ...
    if (req) {
        ...
        if (!tcp_check_req(...))
            ...
    }
    ...
    switch (sk->sk_state) {
        case TCP_SYN_RECV:
            ...
            tcp_set_state(sk, TCP_ESTABLISHED);
            ...
        ...
    }
    ...
}
```

- 在tcp\_rcv\_state\_process函数调用tcp\_set\_state函数之前会先调用tcp\_check\_req函数

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/tcp\_minisocks.c:578

```
struct sock* tcp_check_req(...) {  
    ...  
    return inet_csk_complete_hashdance(...);  
    ...  
}
```

- tcp\_check\_req函数调用inet\_csk\_complete\_hashdance函数

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/inet\_connection\_sock.c:944

```
struct sock* inet_csk_complete_hashdance(...) {  
    if (own_req) {  
        inet_csk_reqsk_queue_drop(...);  
        ...  
    }  
    ...  
}
```

- inet\_csk\_complete\_hashdance函数调用inet\_csk\_reqsk\_queue\_drop函数

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/inet\_connection\_sock.c:660

```
void inet_csk_reqsk_queue_drop(...) {  
    if (reqsk_queue_unlink(...)) {  
        ...  
    }  
}
```

- inet\_csk\_reqsk\_queue\_drop函数调用reqsk\_queue\_unlink函数

# 分析内核源码

- 遭受TCP SYN攻击
  - /usr/src/linux-source-4.15.0/net/ipv4/inet\_connection\_sock.c:642

```
static bool reqsk_queue_unlink(...) {  
    ...  
    if (sk_hashed(req_to_sk(...))) {  
        ...  
        found = __sk_nulls_del_node_init_rcu(req_to_sk(...));  
        ...  
    }  
    if (timer_pending(...) && del_timer_sync(...))  
        ...  
}
```

- reqsk\_queue\_unlink函数先通过\_\_sk\_nulls\_del\_node\_init\_rcu函数将半连接套接字从哈希队列中删除，再通过timer\_pending和del\_timer\_sync函数销毁未到期的定时器

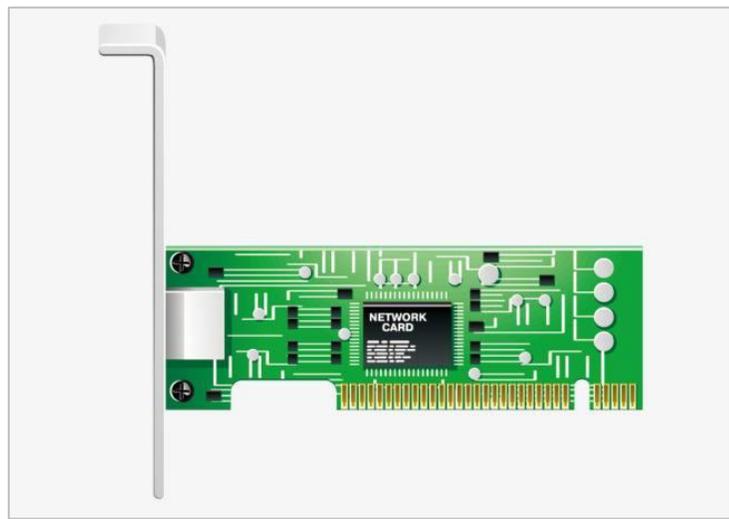
# 分析内核源码

- 遭受TCP SYN攻击
  - 利用TCP协议的这种特点，攻击者向服务器的某个开发端口发送大量孤立的SYN请求包，并伪造不同的源IP地址
  - 系统为每个这样的请求包都要维护一个独立的超时定时器，并将相应的半连接套接字加入重发队列，每隔一段时间就要为每个这样的套接字重发SYN+ACK响应包，系统资源将很快耗尽
  - 可见，TCP SYN攻击对Linux系统的危害十分严重



# 修改内核源码

- 当以太网卡接收到一个数据包时，其DMA控制器会触发一个硬件中断，通知系统内核该设备已然就绪，即可被轮询(Poll)
  - 针对该中断的处理函数在网卡驱动中实现
  - 以Intel 8255x系列网卡的断处理函数为例



# 修改内核源码

- /usr/src/linux-source-4.15.0/drivers/net/ethernet/intel/e100.c:2217

```
static irqreturn_t e100_intr(...) {  
    ...  
    if (likely(napi_schedule_prep(...))) {  
        e100_disable_irq(...);  
        __napi_schedule(...);  
    }  
    ...  
}
```

- e100\_intr函数先通过napi\_schedule\_prep函数检查该设备是否已在就绪设备队列中，如果不在，则通过e100\_disable\_irq函数关闭当前中断，然后调用\_\_napi\_schedule函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:5282

```
void __napi_schedule(...) {  
    ...  
    ____napi_schedule(...);  
    ...  
}
```

- \_\_napi\_schedule函数调用\_\_\_\_napi\_schedule函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:3587

```
static inline void ____napi_schedule(...) {  
    list_add_tail(...);  
    __raise_softirq_irqoff(NET_RX_SOFTIRQ);  
}
```

- \_\_\_\_napi\_schedule函数先通过list\_add\_tail函数将该网卡追加到读就绪设备队列的尾部, 然后通过\_\_raise\_softirq\_irqoff函数触发一个名为NET\_RX\_SOFTIRQ的软中断

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:8817

```
static int __init net_dev_init(void) {  
    ...  
    open_softirq(NET_RX_SOFTIRQ, net_rx_action);  
    ...  
}
```

- 从net\_dev\_init函数可以看出，针对NET\_RX\_SOFTIRQ软中断的中断处理函数是net\_rx\_action

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:5684

```
static __latent_entropy void net_rx_action(...) {  
    ...  
    for (;;) {  
        ...  
        n = list_first_entry(...);  
        budget -= napi_poll(n, ...);  
        ...  
    }  
    ...  
}
```

- net\_rx\_action函数在循环中轮询就绪设备队列，对其中的每一个就绪设备(n)调用napi\_poll函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:5622

```
static int napi_poll(...) {  
    ...  
    if (test_bit(...)) {  
        work = n->poll(...);  
        ...  
    }  
    ...  
}
```

- napi\_poll函数通过poll函数指针调用由网卡驱动提供的e100\_poll函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/drivers/net/ethernet/intel/e100.c:2245

```
static int e100_poll(...) {  
    ...  
    e100_rx_clean(..., &work_done, budget);  
    ...  
    if (work_done < budget) {  
        ...  
        e100_enable_irq(...);  
    }  
    ...  
}
```

- e100\_poll函数通过e100\_rx\_clean函数从设备中读取数据。如果实际读到的字节数比期望读取的字节数少(work\_done < budget), 则说明设备中的数据已经读完, 这时通过e100\_enable\_irq函数恢复当前中断

# 修改内核源码

- /usr/src/linux-source-4.15.0/drivers/net/ethernet/intel/e100.c:2078

```
static void e100_rx_clean(...) {  
    ...  
    for (...) {  
        err = e100_rx_indicate(...);  
        ...  
    }  
    ...  
}
```

- e100\_rx\_clean函数调用e100\_rx\_indicate函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/drivers/net/ethernet/intel/e100.c:1979

```
static int e100_rx_indicate(...) {  
    ...  
    if (...) {  
        ...  
    } else if (...) {  
        ...  
    } else {  
        ...  
        netif_receive_skb(...);  
        ...  
    }  
    ...  
}
```

- e100\_rx\_indicate函数通过netif\_receive\_skb函数，将从设备中读到的数据交给上层协议栈处理

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:4657

```
int netif_receive_skb(...) {  
    ...  
    return netif_receive_skb_internal(...);  
}
```

- netif\_receive\_skb函数调用netif\_receive\_skb\_internal函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:4602

```
static int netif_receive_skb_internal(...) {  
    ...  
    ret = __netif_receive_skb(...);  
    ...  
}
```

- netif\_receive\_skb\_internal函数调用\_\_netif\_receive\_skb函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:4544

```
static int __netif_receive_skb(...) {  
    ...  
    if (...) {  
        ...  
        ret = __netif_receive_skb_core(...);  
        ...  
    }  
    else  
        ret = __netif_receive_skb_core(...);  
    ...  
}
```

- \_\_netif\_receive\_skb函数调用\_\_netif\_receive\_skb\_core函数

# 修改内核源码

- /usr/src/linux-source-4.15.0/net/core/dev.c:4368

```
static int __netif_receive_skb_core(...) {  
    ...  
    list_for_each_entry_rcu(..., &ptype_all, ...) {  
        ...  
    }  
    list_for_each_entry_rcu(..., &skb->dev->ptype_all,...) {  
        ...  
    }  
    ...  
}
```

- \_\_netif\_receive\_skb\_core函数先遍历ptype\_all链表，将数据包传给系统中注册的相关原始套接字，然后遍历skb->dev->ptype\_all链表，将数据包传给与其协议类型相应的上层协议处理，如将IP包传给IP层处理。数据链路层处理结束

# 修改内核源码

- 为了抵御TCP SYN攻击，可以在 `_netif_receive_skb_core` 函数中添加必要代码，在数据包进入上层协议栈处理之前，对其进行拦截，若确系TCP SYN攻击包，则通知协议栈予以丢弃，以此达到加固系统内核的目的。如下所示：
  - `/usr/src/linux-source-4.15.0/net/core/dev.c:4368`

```
/*  
 * Added by MW for Anti-DoS at 26 Apr 2019  
 */  
int (*pfn_anti_dos_hook)(struct sk_buff* skb) = NULL;  
EXPORT_SYMBOL(pfn_anti_dos_hook);  
/*  
 * End of Addition  
 */
```

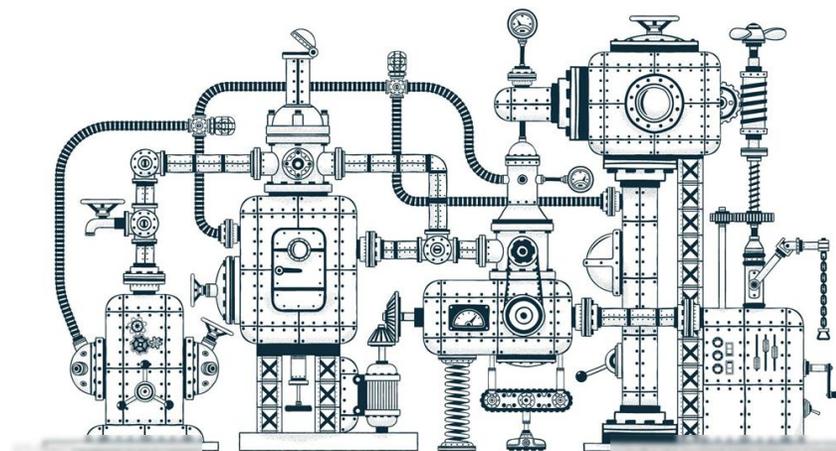
```
static int __netif_receive_skb_core(struct sk_buff* skb, ...) {  
    ...  
    skb_reset_network_header(skb);  
    if (!skb_transport_header_was_set(skb))  
        skb_reset_transport_header(skb);  
    skb_reset_mac_len(skb);  
}
```

```
/*  
 * Added by MW for Anti-DoS at 26 Apr 2019  
 */  
if (pfn_anti_dos_hook && pfn_anti_dos_hook(skb))  
    goto drop;  
/*  
 * End of addition  
 */
```

```
pt_prev = NULL;  
  
another_round:  
    skb->skb_iif = skb->dev->ifindex;  
  
    __this_cpu_inc(softnet_data.processed);  
    ...  
}
```

# 编译内核源码

- 安装工具
  - `sudo apt-get install build-essential`
  - `sudo apt-get install libncurses-dev`
  - `sudo apt-get install kernel-package`
  - `sudo apt-get install initramfs-tools`
  - `sudo apt-get install module-init-tools`
  - `sudo apt-get install libelf-dev`



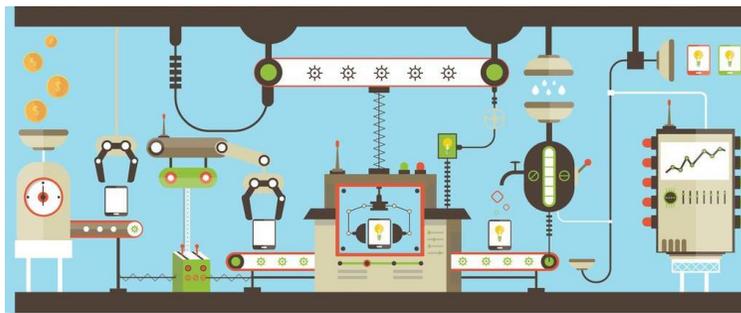
# 编译内核源码

- 准备环境
  - `cd /usr/src/linux-source-4.15.0`
  - `sudo cp /boot/config-`uname -r` .config`
  - `sudo make mrproper`
  - `sudo make clean`
  - `sudo make menuconfig`



# 编译内核源码

- 编译安装
  - `sudo make -j8`
  - `sudo make install`
  - `sudo make modules`
  - `sudo make modules_install`



# 实训案例

实训案例

实训案例

提升系统内核抵御TCP SYN攻击的能力

程序清单

# 实训案例

---

# 提升系统内核抵御TCP SYN攻击的能力

- 通过编程增强Linux系统内核对TCP SYN攻击的抵抗能力
- 程序在不影响已存在的TCP连接的前提下，过滤TCP SYN数据包
- SYN攻击是以大量消耗目标系统资源的方式实现攻击的，因此要求程序实现上述功能占用尽量少的系统资源
- 不需要实现交互界面等，只需实现基本的数据包过滤功能即可

# 程序清单

- 拒绝服务攻击防御模块
  - antidos.c
- 拒绝服务攻击防御模块构建脚本
  - Makefile

# 扩展提高



# 其它拒绝服务攻击

---

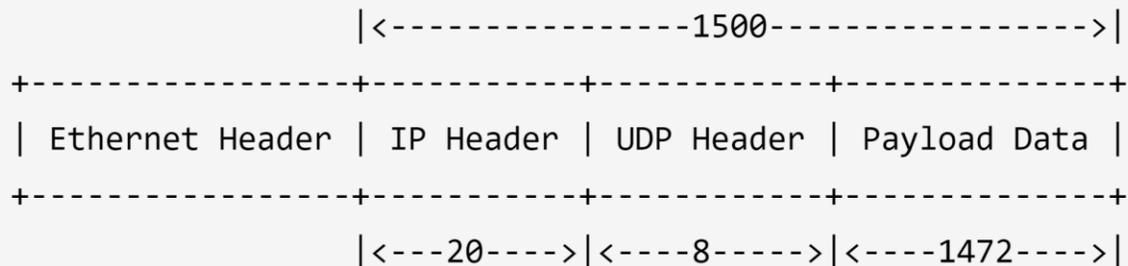
# IP碎片攻击

- 网络协议栈的数据链路层对所能传输的数据包大小设定了上限，即最大传输单元(Maximum Transfer Unit, MTU)。以太网的最大传输单元是1500字节。该值可通过如下命令查看：

– netstat -i

```
Kernel Interface table
Iface MTU  Met RX-OK  RX-ERR RX-DRP RX-OVR TX-OK TX-ERR TX-DRP TX-OVR Flg
ens33 1500  0   161818 0       0       0   77637 0       0       0   BMRU
```

- 以UDP传输为例，一个最大以太网包的结构如下图所示：
  - 如果所要发送的有效载荷多于1472个字节，IP层就会对其分片，以使每个以太网包都满足最大传输单元的限制





# IP碎片攻击

- Tear Drop攻击

- 正常IP分片的偏移位置，即IP包头frag\_off字段中的D，应该保证所有分片无间隔、无重叠、无包含地连续分布
- 某些操作系统存在漏洞，一旦某个IP分片的偏移位置发生错误，使其刚好被包含于另一个IP分片的内部，就会发生崩溃或重启现象
- 另一些操作系统遇到这种情况虽然不会崩溃或重启，但也会在试图根据错误的偏移位置重组分片的过程中浪费大量时间
- 攻击者利用了这一点，故意向服务器发送大量伪造的，包含错误偏移位置的IP分片，导致服务器系统异常甚至崩溃，无法继续提供服务，这就是Tear Drop攻击

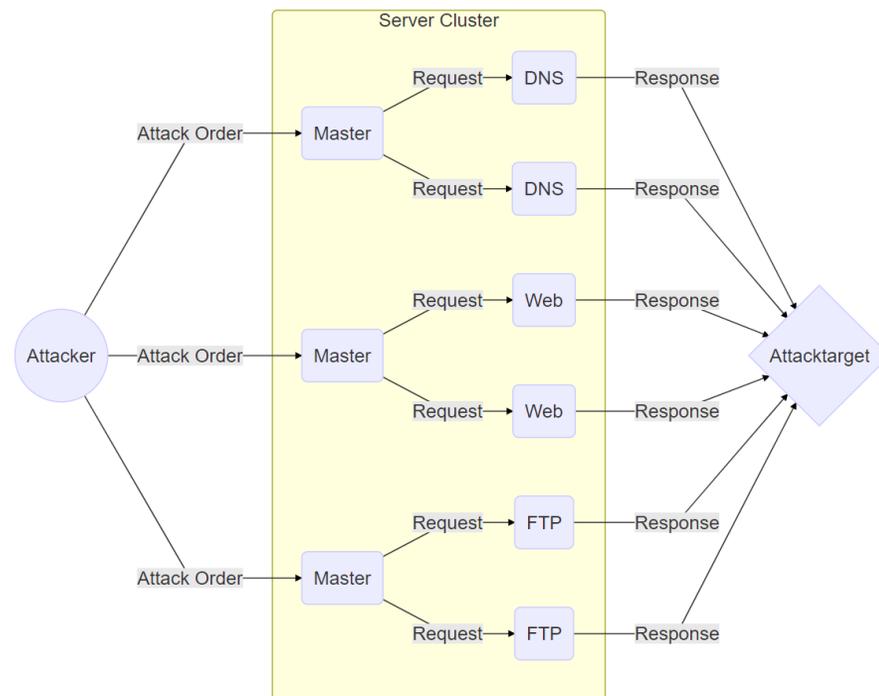
# IP碎片攻击

- Pingo Death攻击
  - Pingo Death攻击是利用ICMP协议的一种碎片攻击
  - 某些操作系统在进行ICMP分片重组时预分配的缓冲区大小为65535字节
    - 攻击者通过分片重组发送一个长度超过65535字节的ICMP应答请求数据包
    - 目的主机在重组分片时会因为缓冲区溢出而崩溃或挂起



# 分布式反弹拒绝服务攻击

- 互联网上有很多服务器，比如DNS服务器、Web服务器、FTP服务器等，它们在收到一个来自客户端的请求报文后，通常会产生一个特定的响应报文返回给客户端，这样的服务器被称为反弹服务器
- 攻击者向由多台反弹服务器组成的反弹服务集群发送大量请求报文，但这些请求报文的源IP地址均被伪造为被攻击主机的IP地址。反弹服务器上开放的服务通常为匿名服务，或者存在身份认证漏洞，因此不会对请求报文的源IP地址做太多验证，就直接向其源IP地址，即被攻击主机，发送响应报文。只要反弹服务集群的规模足够大，数量庞大而时间集中的响应报文足以令被攻击主机陷入瘫痪状态。这就是基于反弹技术的拒绝服务攻击。如下图所示：





# 基于应用层协议漏洞的拒绝服务攻击

- 这种攻击的优势在于：
  - 不留痕迹：IIS服务器只在为客户端提供的服务完成之后才会记录日志，而这种伪造大量数据上传的服务请求根本不可能完成，因此不会留下任何日志记录
  - 难辨真伪：伪造的POST指令很难被防火墙发现，其Content-Length字段的值究竟是真是假无从得知
  - 成本低廉：对攻击主机的性能要求不高，只需快速构造POST请求即可



# TCP SYN Cookie



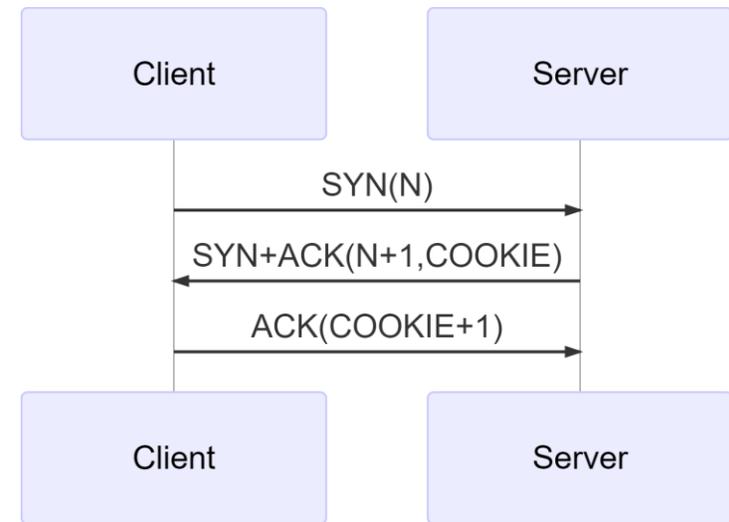
# TCP SYN Cookie原理

- TCP SYN Cookie是截至目前能够有效防御TCP SYN泛洪拒绝服务攻击的各种方法中最著名的一种。该方法最早由D. J. Bernstein和Eric Schenk发明。其在很多操作系统上都有实现，也包括Linux系统
- TCP SYN Cookie对建立TCP连接的三路握手过程进行了优化
  - 传统的TCP服务器，在收到客户端为建立连接发来的包头中带有SYN标志位的TCP数据包后，会立即为其创建连接资源。一旦遭遇TCP SYN泛洪攻击，这些为连接而创建的资源会急剧增加，以至拖垮整个系统



# TCP SYN Cookie原理

- TCP SYN Cookie对建立TCP连接的三路握手过程进行了优化
  - 应用了TCP SYN Cookie技术的TCP服务器，在收到客户端为建立连接发来的包头中带有SYN标志位的TCP数据包后，并不立即为其创建连接资源，而是直接返回包头中带有SYN和ACK两个标志位的TCP数据包。该数据包的发送序列号是由SYN包的源IP地址、源端口号、目的IP地址和目的端口号组合在一起的哈希摘要，即Cookie。在收到客户端返回的包头中带有ACK标志位的TCP数据包后，服务器再将该数据包的接收序列号减一得到Cookie，结合其源IP地址、源端口号、目的IP地址和目的端口号，验证其合法性。只有通过合法性验证的客户端才能得到服务器为其创建的连接资源。如下图所示：



# TCP SYN Cookie原理

- /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:6252
  - 如前所述，服务器在收到客户端为建立连接发来的包头中带有SYN标志位的TCP数据包后，在tcp\_conn\_request函数中，通过send\_synack函数指针调用tcp\_v4\_send\_synack函数，向客户端返回包头中带有SYN和ACK两个标志位的TCP数据包

```
int tcp_conn_request(...) {
    ...
    if (want_cookie) {
        isn = cookie_init_sequence(...);
        ...
    }
    ...
    if (fastopen_sk) {
        af_ops->send_synack(...);
        ...
    } else {
        ...
        if (!want_cookie)
            inet_csk_reqsk_queue_hash_add(..., tcp_timeout_init(...));
        af_ops->send_synack(...);
        ...
    }
    ...
}
```

# TCP SYN Cookie原理

- /usr/src/linux-source-4.15.0/net/ipv4/tcp\_input.c:6252
  - 对于应用了TCP SYN Cookie技术的TCP服务器，其want\_cookie的值为1，在收到客户端为建立连接发来的包头中带有SYN标志位的TCP数据包后：
    - 调用cookie\_init\_sequence函数，根据该数据包的源IP地址、源端口号、目的IP地址和目的端口号计算哈希摘要，作为响应数据包的发送序列号，即Cookie
    - 因为want\_cookie的值为1，所以不会调用inet\_csk\_reqsk\_queue\_hash\_add函数，也就不会将半连接套接字加入哈希队列并开启定时器
- 当服务器收到客户端发来的包头中带有ACK标志位的TCP数据包时，服务器的tcp\_v4\_rcv函数再次被调用，该函数调用tcp\_v4\_do\_rcv函数

# TCP SYN Cookie原理

- /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:1453

```
int tcp_v4_do_rcv(...) {
    ...
    if (sk->sk_state == TCP_LISTEN) {
        struct sock* nsk = tcp_v4_cookie_check(...);
        ...
        if (nsk != sk) {
            if (tcp_child_process(...)) {
                ...
            }
            ...
        }
    }
    ...
}
```

- tcp\_v4\_do\_rcv函数调用tcp\_v4\_cookie\_check函数

# TCP SYN Cookie原理

- /usr/src/linux-source-4.15.0/net/ipv4/tcp\_ipv4.c:1434

```
static struct sock* tcp_v4_cookie_check(...) {  
    ...  
    if (!th->syn)  
        sk = cookie_v4_check(...);  
    ...  
}
```

- tcp\_v4\_cookie\_check函数调用cookie\_v4\_check函数

# TCP SYN Cookie原理

- /usr/src/linux-source-4.15.0/net/ipv4/syncookies.c:283

```
struct sock* cookie_v4_check(...) {  
    ...  
    const struct tcphdr* th = tcp_hdr(skb);  
    __u32 cookie = ntohl(th->ack_seq) - 1;  
    ...  
    mss = __cookie_v4_check(ip_hdr(skb), th, cookie);  
    if (mss == 0) {  
        ...  
        goto out;  
    }  
    ...  
    ret = tcp_get_cookie_sock(...);  
    ...  
}
```

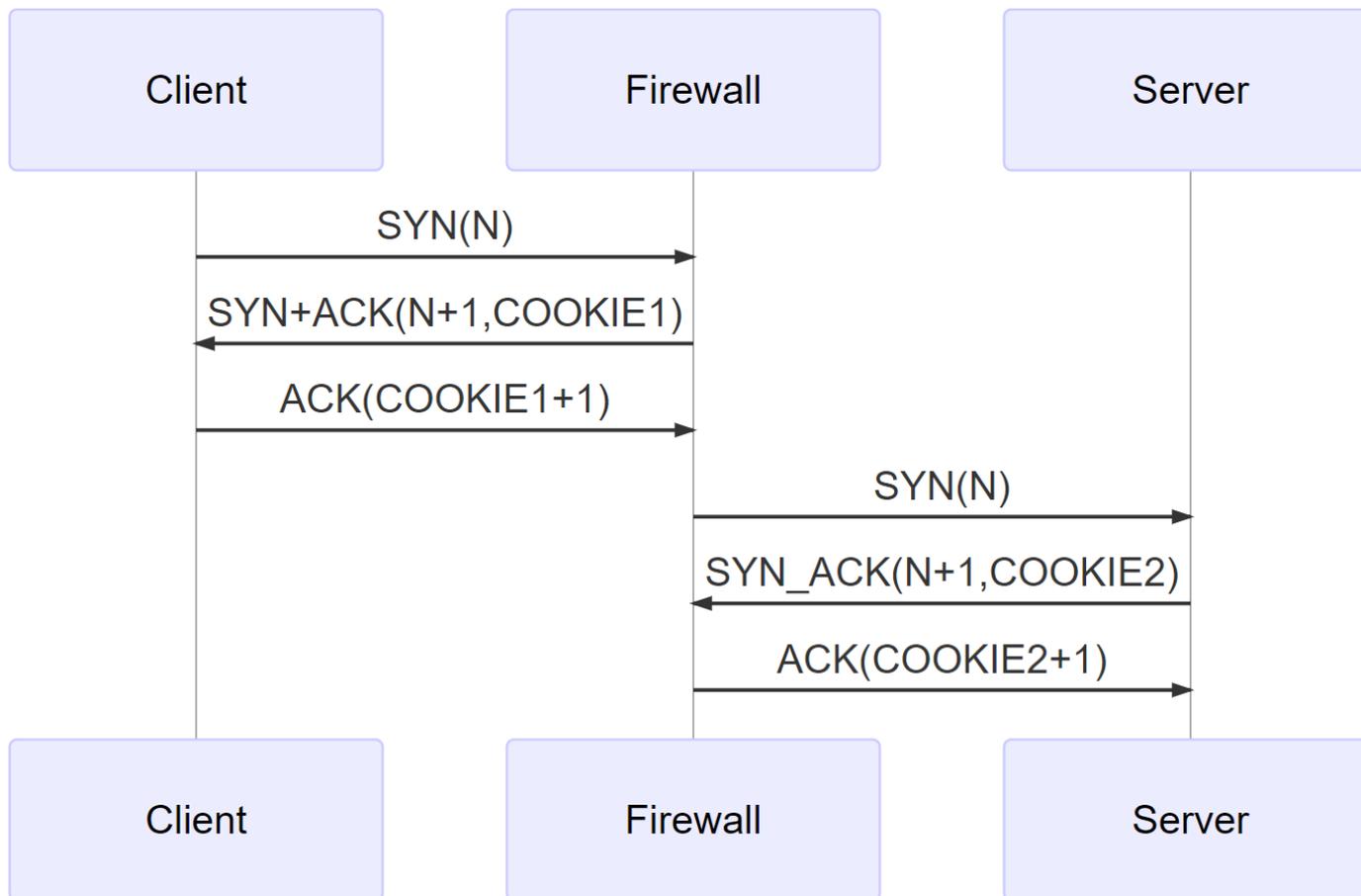
# TCP SYN Cookie原理

- /usr/src/linux-source-4.15.0/net/ipv4/syncookies.c:283
  - cookie\_v4\_check函数先通过\_\_cookie\_v4\_check函数，根据IP包头(ip\_hdr(skb))中的源IP地址、目的IP地址和TCP包头(th)中的源端口号、目的端口号，计算其哈希摘要，与接收序列号减一( $\text{ntohl}(\text{th} \rightarrow \text{ack\_seq}) - 1$ )，即Cookie(cookie)，做相等性比较，二者一致则通过验证，然后通过tcp\_get\_cookie\_sock函数为通过验证的客户端创建连接资源，该函数返回用于后续通信的套接字



# TCP SYN Cookie防火墙

- 基于TCP SYN Cookie原理的防火墙如下图所示：



# TCP SYN Cookie防火墙

- 外网客户端和内网服务器建立TCP连接的过程如下：
  - 当防火墙收到来自外网客户端的SYN数据包时，并不直接向内网服务器转发，而是缓存在本地，同时按照前述TCP SYN Cookie原理向外网客户端返回带有Cookie的SYN+ACK数据包，该包的源IP地址被改写为内网服务器的IP地址
  - 外网客户端收到该SYN+ACK数据包后，发送ACK数据包，并认为与服务器的TCP连接已建立起来
  - 防火墙依据Cookie验证该ACK数据包的合法性，若通过验证则将之前缓存在本地的，来自外网客户端的SYN数据包发送给内网服务器
  - 内网服务器对防火墙响应以SYN+ACK数据包，该数据包中同样带有Cookie
  - 防火墙伪造一个来自外网客户端的ACK数据包发送给内网服务器
  - 至此，外网客户端和内网服务器的TCP连接建立完毕，双方开始数据传输

# TCP SYN Cookie防火墙

- 需要注意的是，防火墙为外网客户端生成的发送序列号(COOKIE1)与内网服务器为防火墙生成的发送序列号(COOKIE2)未必是一样的，因此防火墙在每次转发外网客户端和内网服务器之间的数据包时需要对其中的序列号进行修改，以确保TCP通信正常



# 总结和答疑

