协议栈测试神器 packetdrill

packetdrill 历史

packetdrill 在 **2013** 年开源，在 Google 内部久经考验，Google 用它发现了 10 余个 Linux 内核 bug，同时用测试驱动开发的方式开发新的网络特性和进行回归测试，确保新功能的添加不影响网络协议栈的可用性。

安装步骤

1. 首先从 github 上 clone 最新的源码
2. 进入源码目录 `cd gtests/net/packetdrill`
3. 安装 bison和 flex 库：`sudo yum install -y bison flex`
4. 为避免 offload 机制对包大小的影响，修改 netdev.c 注释掉 `set_device_offload_flags` 函数所有内容
5. 执行 ./configure
6. 修改 Makefile，去掉第一行的末尾的 -static
7. 执行 make 命令编译
8. 确认编译无误地生成了 packetdrill 可执行文件
9. 关闭防火墙（停止 firewall 服务）

初体验

packetdrill 脚本采用 c 语言和 tcpdump 混合的语法。脚本文件名一般以 **.pkt** 为后缀，执行脚本的方式为 sudo ./packetdrill test.pkt

```
 1 0    socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
 2 +0   setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
 3 +0   bind(3, ..., ...) = 0
 4 +0   listen(3, 1) = 0
 5
 6 //TCP three-way handshake
 7 +0   < S 0:0(0) win 4000 <mss 1000>
 8 +0   > S. 0:0(0) ack 1 <...>
 9 +.1  < . 1:1(0) ack 1 win 1000
10
11 +0 accept(3, ..., ...) = 4
12 +0 < P. 1:201(200) win 4000
13 +0 > . 1:1(0) ack 201
```

脚本格式-时间

脚本每一行都有一个时间参数用来表明执行的时间或者预期事件发生的时间，packetdrill 支持绝对时间和相对时间。

绝对时间就是一个简单的数字，相对时间会在数字前面添加一个 + 号。

```
// 300ms 时执行 accept 调用
0.300 accept(3, ..., ...) = 4

// 在上一行语句执行结束 10ms 以后执行
+.010 write(4, ..., 1000) = 1000`
```

逐行解析

```
1 0   socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
2 +0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
3 +0  bind(3, ..., ...) = 0
4 +0  listen(3, 1) = 0
```

逐行解析

第 1 行：0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

成功时返回文件描述符，失败时返回 -1

```
int socket_fd = socket(AF_INET, SOCK_STREAM, 0);
```

- domain 表示套接字使用的协议族信息，IPv4、IPv6等。AF_INET 表示 IPv4 协议族，AF_INET6 表示 IPv6 协议族。绝大部分使用场景下都是用 AF_INET，即 IPv4 协议族
- type 表示套接字数据传输类型信息，主要分为两种：面向连接的套接字（SOCK_STREAM）和面向无连接报文的套接字（SOCK_DGRAM）。众所周知，SOCK_STREAM 默认协议是 TCP，SOCK_DGRAM 的默认协议是 UDP。
- protocol 表示为给定的协议族和套接字类型选择协议。

逐行解析

```
0 socket(..., SOCK_STREAM, IPPROTO_TCP) = 3
```

- ... 表示当前参数省略不相关的细节信息，使用 packetdrill 程序的默认值。
- 脚本返回新建的 socket 文件句柄，这里用 = 来断言会返回 3

socket 函数调用语法图解

逐行解析

```
+0  setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
```

函数示意

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);

int optval = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
```

逐行解析

```
+0  bind(3, ..., ...) = 0
```

```
int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

逐行解析

```
+0  listen(3, 1) = 0
```

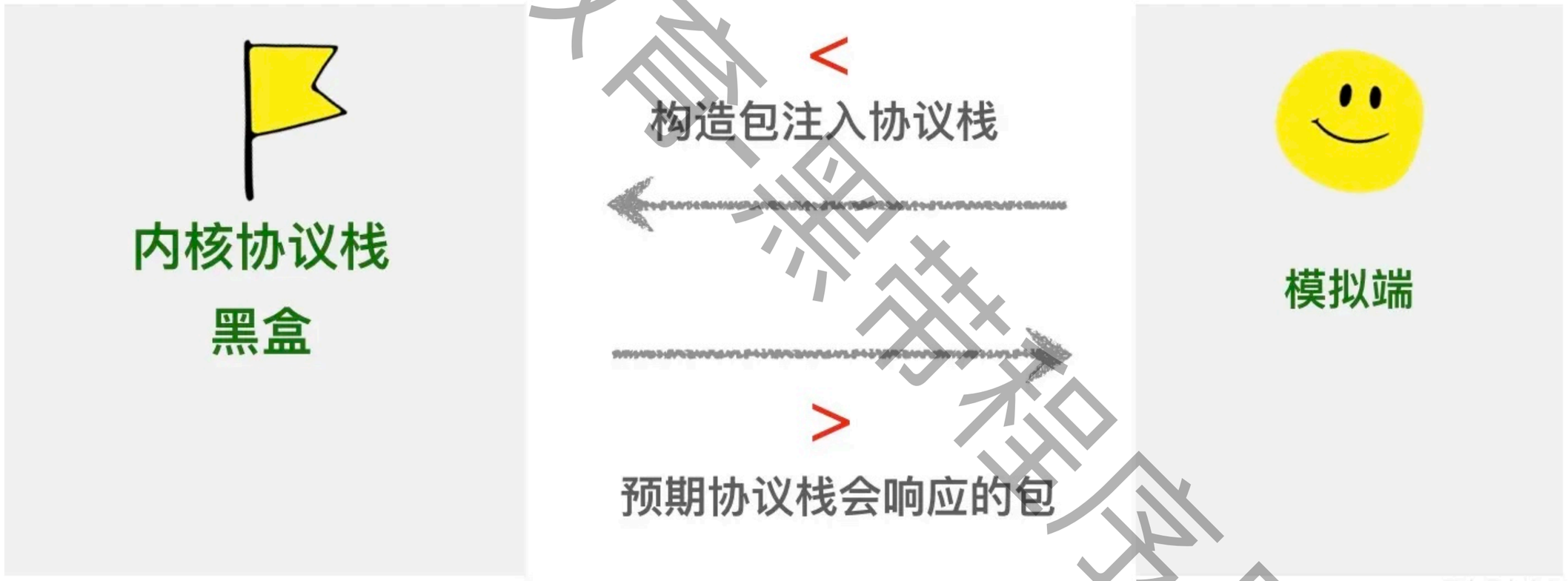
```
int listen(int sockfd, int backlog);
```

逐行解析：三次握手

类 tcpdump 的语法

```
7 +0  < S 0:0(0) win 4000 <mss 1000>
8 +0  > S. 0:0(0) ack 1 <...>
9 +.1 < . 1:1(0) ack 1 win 1000
```

注入与断言的游戏



逐行解析：三次握手第一步（客户端 SYN）

```
+0  < S 0:0(0) win 4000 <mss 1000>
```

< 表示输入的数据包（input packets)， packetdrill 会构造一个真实的数据包，注入到内核协议栈。

```
// 构造 SYN 包注入到协议栈
+0  < S 0:0(0) win 32792 <mss 1000,sackOK,nop,nop,nop,wscale 7>

// 构造 icmp echo_reply 包注入到协议栈
0.400 < icmp echo_reply
```

逐行解析：三次握手第二步（服务端回复 SYN+ACK）

```
+0  > S. 0:0(0) ack 1 <...>
```

断言协议栈会立刻回复 SYN+ACK 包，因为还没有发送数据，所以包的 seq开始值、结束值、长度都为 0，ack 为上次 seq + 1，表示第一个 SYN 包已收到。

> 表示预期协议栈会响应的包（outbound packets），这个包不是 packetdrill 构造的，是由协议栈发出的，packetdrill 会检查协议栈是不是真的发出了这个包，如果没有，则脚本报错停止执行。比如

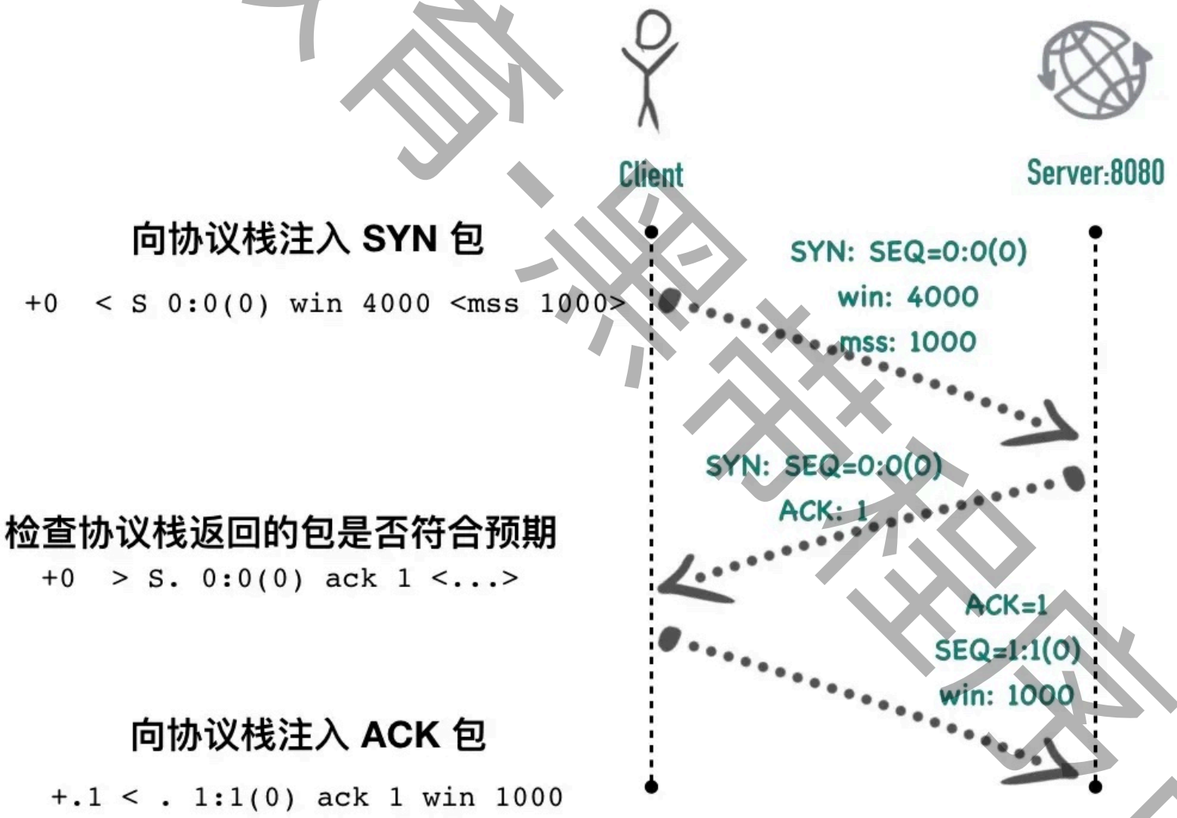
```
// 调用 write 函数调用以后，检查协议栈是否真正发出了 PSH+ACK 包
+0  write(4, ..., 1000) = 1000
+0  > P. 1:1001(1000) ack 1

// 三次握手中过程向协议栈注入 SYN 包以后，检查协议栈是否发出了 SYN+ACK 包以及 ack 是否等于 1
0.100 < S 0:0(0) win 32792 <mss 1000,nop,wscale 7>
0.100 > S. 0:0(0) ack 1 <mss 1460,nop,wscale 6>
```

逐行解析：三次握手第三步（客户端回复 ACK）

```
+.1 < . 1:1(0) ack 1 win 1000
```

0.1s 以后注入一个 ACK 包到协议栈，没有携带数据，包的长度为 0，至此三次握手完成。

逐行解析



逐行解析，接收客户端连接

```
+0 accept(3, ..., ...) = 4
```

accept 系统调用返回了一个值为 4 的新的文件 fd，这时 packetdrill 可以往这个 fd 里面写数据了

函数定义

```
int accept(int socket, struct sockaddr *restrict address,
socklen_t *restrict address_len);
```

```
int client_fd = accept(fd, (struct sockaddr *) &client_addr, &len);
```

逐行解析，写入数据

```
+0 write(4, ..., 10) = 10
```

packetdrill 调用 write 函数往 socket 里写了 10 字节的数据

函数定义

```
ssize_t write(int fd, const void *buf, size_t count);
```

逐行解析，断言服务端会回复 PUSH + ACK

```
+0 > P. 1:11(10) ack 1
```

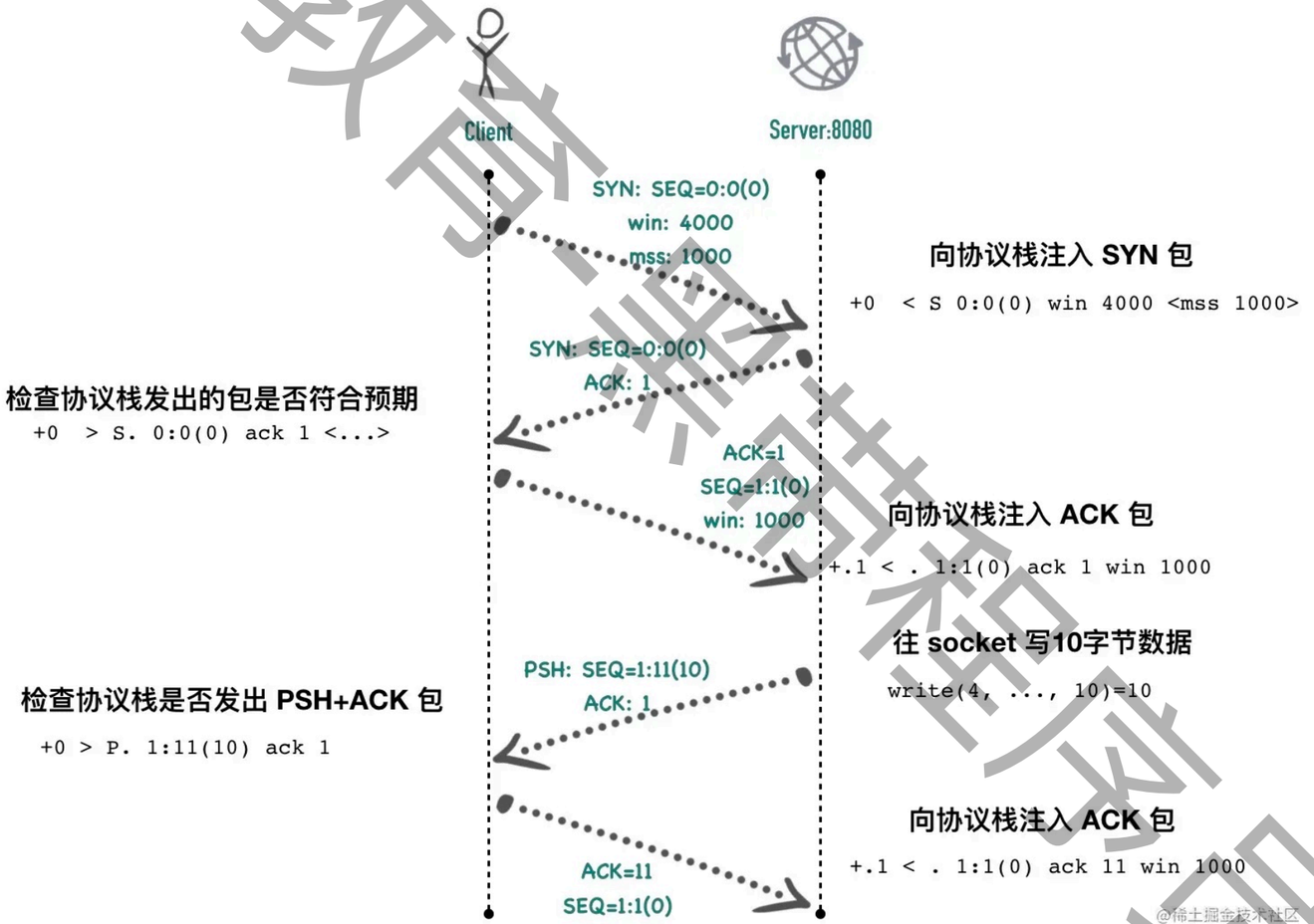
协议栈立刻发出这 10 个字节数据包，同时把 PSH 标记置为 1。这个包的起始 seq 为 1，结束 seq 为 10，长度为 10

逐行解析，模拟客户端回复 ACK

```
+.1 < . 1:1(0) ack 11 win 1000
```

100ms 以后注入 ACK 包，模拟协议栈收到 ACK 包。

完整过程



抓包演示

```
$ sudo tcpdump -i any port 8080 -nn
10:02:36.591911 IP 192.0.2.1.37786 > 192.168.31.139.8080: Flags [S], seq 0, win 4000, options [mss 1000], length 0
10:02:36.591961 IP 192.168.31.139.8080 > 192.0.2.1.37786: Flags [S.], seq 2327356581, ack 1, win 29200, options [mss 1460], length 0
10:02:36.693785 IP 192.0.2.1.37786 > 192.168.31.139.8080: Flags [.], ack 1, win 1000, length 0
10:02:36.693926 IP 192.168.31.139.8080 > 192.0.2.1.37786: Flags [P.], seq 1:11, ack 1, win 29200, length 10
10:02:36.801092 IP 192.0.2.1.37786 > 192.168.31.139.8080: Flags [.], ack 11, win 1000, length 0
```

packetdrill 原理

- packetdrill 语法解析
- TUN/TAP 设备

Flex + Bison

Flex 和 Bison 是 Linux 下用来生成词法分析器和语法分析器两个程序的工具，可以处理结构化输入。

flex 文件是定义 pattern，通过 flex 处理（词法分析）将输出切分成一段一段的 token，从而执行不同的 action。flex 生成的 tokens 可以交给 Bison 处理，当然直接自己处理也可以，但是使用 Bison 可以更方便的处理复杂的逻辑，编写简单，调试方便。

Flex 使用示例

```
%{
  int chars = 0;
  int words = 0;
  int lines =0;

%}

%%
[a-zA-Z]+ {
        words++;
        chars += strlen(yytext);
          }

\n        {     chars++; lines++;}

.         {     chars++;     }

%%


int main(int args, char **argv)
{
    yylex();
    printf("lines=%d words=%d chars=%d\n", lines, words, chars);
    return 0;
}
```

实战 Flex + Bison 实现计算器

TUN/TAP 设备

在脚本的最后一行，加上+0sleep 1000000``，让脚本执行完不要退出，执行 ifconfig 可以看到，比没有执行脚本之前多了一个虚拟的 TUN/TAP 设备 tun0。

```
ya@c2 ~$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.211.55.10  netmask 255.255.255.0  broadcast 10.211.55.255
        inet6 fdb2:2c26:f4e4:0:21c:42ff:febe:259c  prefixlen 64  scopeid 0x0<global>
        inet6 fe80::21c:42ff:febe:259c  prefixlen 64  scopeid 0x20<link>
        ether 00:1c:42:be:25:9c  txqueuelen 1000  (Ethernet)
        RX packets 415643  bytes 222391415 (212.0 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 225301  bytes 67081990 (63.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1  (Local Loopback)
        RX packets 25324  bytes 2320086 (2.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 25324  bytes 2320086 (2.2 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 192.168.141.72  netmask 255.255.0.0  destination 192.168.141.72
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 3  bytes 124 (124.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 94 (94.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

TUN/TAP 设备

tun/tap设备的用处是将协议栈中的部分数据包转发给用户空间的应用程序，给用户空间的程序一个处理数据包的机会。

比较常用的数据压缩，加密等功能就可以在应用程序B里面做进去，tun/tap设备最常用的场景是VPN。

tun/tap 设备与物理网卡的区别，如上图所示:

1. 对于硬件网络设备而言，一端连接的是物理网络，一端连接的是网络协议栈。

2. 对于 tun/tap 设备而言，一端连接的是应用程序（通过字符设备文件 /net/dev/tun），一端连接的是网络协议栈。