Cve-2016-1503 漏洞分析

一、漏洞成因

首先来看一下官方的描述：

“A vulnerability in the Dynamic Host Configuration Protocol service could enable an attacker to cause memory corruption, which could lead to remote code execution.”

攻击者可能会通过动态主机配置协议服务中的漏洞破坏内存，从而执行远程代码。

Diff：

```
-               if (opt->type & (UINT32 | IPV4))
+               if (opt->type & (UINT32 | SINT32 | IPV4))
                        sz = sizeof(uint32_t);
-               if (opt->type & UINT16)
+               else if (opt->type & (UINT16 | SINT16))
                        sz = sizeof(uint16_t);
-               if (opt->type & UINT8)
+               else if (opt->type & UINT8)
                        sz = sizeof(uint8_t);
-               if (opt->type & (IPV4 | ARRAY))
-                       return dl % sz;
-               return (dl == sz ? 0 : -1);
+               if (opt->type & ARRAY) {
+                       /* The result of modulo zero is undefined. There are no
+                        * options defined in this file that do not match one of
+                        * the if-clauses above, so the following is not really
+                        * necessary. However, to avoid confusion and unexpected
+                        * behavior if the defined options are ever extended,
+                        * returning false here seems sensible. */
+                       if (!sz) return -1;
+                       return (dl % sz == 0) ? 0 : -1;
+               }
+               return (sz == dl) ? 0 : -1;
        }
```

图 1.1

从官方打下的 Patch 来看，主要的区别是一个连续的 if 语句改成了互斥的 if else 语句，另外一个关键就是之前可以返回除了 0 或者-1 之外的其他值，而 patch 之后只能返回 0 或者-1.

通过以上的分析，可以确定了具体的漏洞原因就是 dhcpcd 在解析 options 的时候长度 dl 的校验出现了问题，导致了后续的远程执行漏洞。

二、函数追踪

定位一下该 patch 的地方可以得知该 Patch 的位置在文件 dhcp.c 中的 valid_length(uint8_t option, int dl, int *type)函数里，查找一下 valid_length 的引用有一处,是在文件 dhcp.c 中的 get_option(const struct dhcp_message *dhcp, uint8_t opt, int *len, int *type)函数，如下：

```
get_option(const struct dhcp_message *dhcp, uint8_t opt, int *len, int *type)
{
        const uint8_t *p = dhcp->options;
        const uint8_t *e = p + sizeof(dhcp->options);
        uint8_t l, ol = 0;
        uint8_t o = 0;
        uint8_t overl = 0;
        uint8_t *bp = NULL;
        const uint8_t *op = NULL;
        int bl = 0;

        ……


        if (valid_length(opt, bl, type) == -1) {
                errno = EINVAL;
                return NULL;
        }
        if (len)
                *len = bl;
        if (bp) {
                memcpy(bp, op, ol);
                return (const uint8_t *)opt_buffer;
        }
        if (op)
                return op;
        errno = ENOENT;
        return NULL;
}
```

从代码可以看到，只要返回值不为-1，即可通过 options 的长度校验，也就是说，只要 dl % sz 不等于-1 即可。sz 是根据 option 的类型来确定，根据不同的类型来取不同的值，如果是 UINT32，sz 则为 4；如果是 UINT16，sz 则为 2，如果是 UINIT8，sz 则为 1。而在 valid_length 函数中的 dl 其实就是 get_option 函数中的 bl，这个值是服务器发出来的数据包中单个 option 的长度。在校验完 bl 的长度后，将会把这个 bl 的值赋给 get_option 函数中的第三个参数*len。

接下来我们继续追踪这个指针 len，查找 get_option 函数的引用，在 dhcp.c 文件中一共有 7 处，排除之后找到 configure_env(char **env, const char *prefix, const struct dhcp_message *dhcp,

const struct if_options *ifo)函数如下：

```
configure_env(char **env, const char *prefix, const struct dhcp_message *dhcp,
        const struct if_options *ifo)
{
        unsigned int i;
        const uint8_t *p;
        int pl;
        struct in_addr addr;
        struct in_addr net;
        struct in_addr brd;
        char *val, *v;
        const struct dhcp_opt *opt;
        ssize_t len, e = 0;
        char **ep;
        char cidr[4];
        uint8_t overl = 0;

        ……

        //循环读取 options
        for (opt = dhcp_opts; opt->option; opt++) {
                if (!opt->var)
                        continue;
                if (has_option_mask(ifo->nomask, opt->option))
                        continue;
                val = NULL;
                p = get_option(dhcp, opt->option, &pl, NULL);
                if (!p)
                        continue;
                /* We only want the FQDN name */
                if (opt->option == DHO_FQDN) {
                        p += 3;
                        pl -= 3;
                }
                len = print_option(NULL, 0, opt->type, pl, p);
                if (len < 0)
                        return -1;
                e = strlen(prefix) + strlen(opt->var) + len + 4;
                v = val = *ep++ = xmalloc(e);
                v += snprintf(val, e, "%s_%s=", prefix, opt->var);
                if (len != 0)
                        print_option(v, len, opt->type, pl, p);
        }

        return ep - env;
}
```

从以上代码可以看到，之前的校验出现了问题的 bl，其实赋值给了 pl，而 pl 在 configure_env

函数中为一个局部变量，在调用 get_option 函数给 pl 赋值后，后来又 2 次调用 print_option 函数，并传入了 pl 参数。接下来就是来跟踪一下这个 pl 的值，查看 print_option(char *s, ssize_t len, int type, int dl, const uint8_t *data)函数如下：

```
print_option(char *s, ssize_t len, int type, int dl, const uint8_t *data)
{
        const uint8_t *e, *t;
        uint16_t u16;
        int16_t s16;
        uint32_t u32;
        int32_t s32;
        struct in_addr addr;
        ssize_t bytes = 0;
        ssize_t l;
        char *tmp;

        ……


        if (!s) {
                if (type & UINT8)
                        l = 3;
                else if (type & UINT16) {
                        l = 5;
                        dl /= 2;
                } else if (type & SINT16) {
                        l = 6;
                        dl /= 2;
                } else if (type & UINT32) {
                        l = 10;
                        dl /= 4;
                } else if (type & SINT32) {
                        l = 11;
                        dl /= 4;
                } else if (type & IPV4) {
                        l = 16;
                        dl /= 4;
                } else {
                        errno = EINVAL;
                        return -1;
                }
                return (l + 1) * dl; //第一次调用  print_option 在这里返回
        }
//第二次调用  print_option 函数才可以到这里
        t = data;
        e = data + dl;
        while (data < e) {
                if (data != t) {
                        *s++ = ' ';
```

```
                bytes++;
                len--;
        }
        if (type & UINT8) {
                l = snprintf(s, len, "%d", *data);
                data++;
        } else if (type & UINT16) {
                memcpy(&u16, data, sizeof(u16));
                u16 = ntohs(u16);
                l = snprintf(s, len, "%d", u16);
                data += sizeof(u16);
        } else if (type & SINT16) {
                memcpy(&s16, data, sizeof(s16));
                s16 = ntohs(s16);
                l = snprintf(s, len, "%d", s16);
                data += sizeof(s16);
        } else if (type & UINT32) {
                memcpy(&u32, data, sizeof(u32));
                u32 = ntohl(u32);
                l = snprintf(s, len, "%d", u32);
                data += sizeof(u32);
        } else if (type & SINT32) {
                memcpy(&s32, data, sizeof(s32));
                s32 = ntohl(s32);
                l = snprintf(s, len, "%d", s32);
                data += sizeof(s32);
        } else if (type & IPV4) {
                memcpy(&addr.s_addr, data, sizeof(addr.s_addr));
                l = snprintf(s, len, "%s", inet_ntoa(addr));
                data += sizeof(addr.s_addr);
        } else
                l = 0;
        if (len <= l) {
                bytes += len;
                break;
        }
        len -= l;
        bytes += l;
        s += l;
    }
    return bytes;
}
```

从上述的代码可以看到，第一次的调用将会在”return (l + 1) * dl ”这里返回，并且会返回一个 len 作为第二次调用的第二个参数再次传入 print_option 函数。第二次调用 print_option 函数的时候，dl 的值影响了 while 循环，这个循环在第一次调用 print_option 函数是无法进入的。

分析到这里，可以推断之前在 valid_length 函数中未合理校验的 dl 值传入到了此 while 循环中，正是由于之前的校验不够完整使得在此 while 循环中 dl 的值不合法，最终导致了该漏洞的形成。

整个在客户端的过程如图所示：

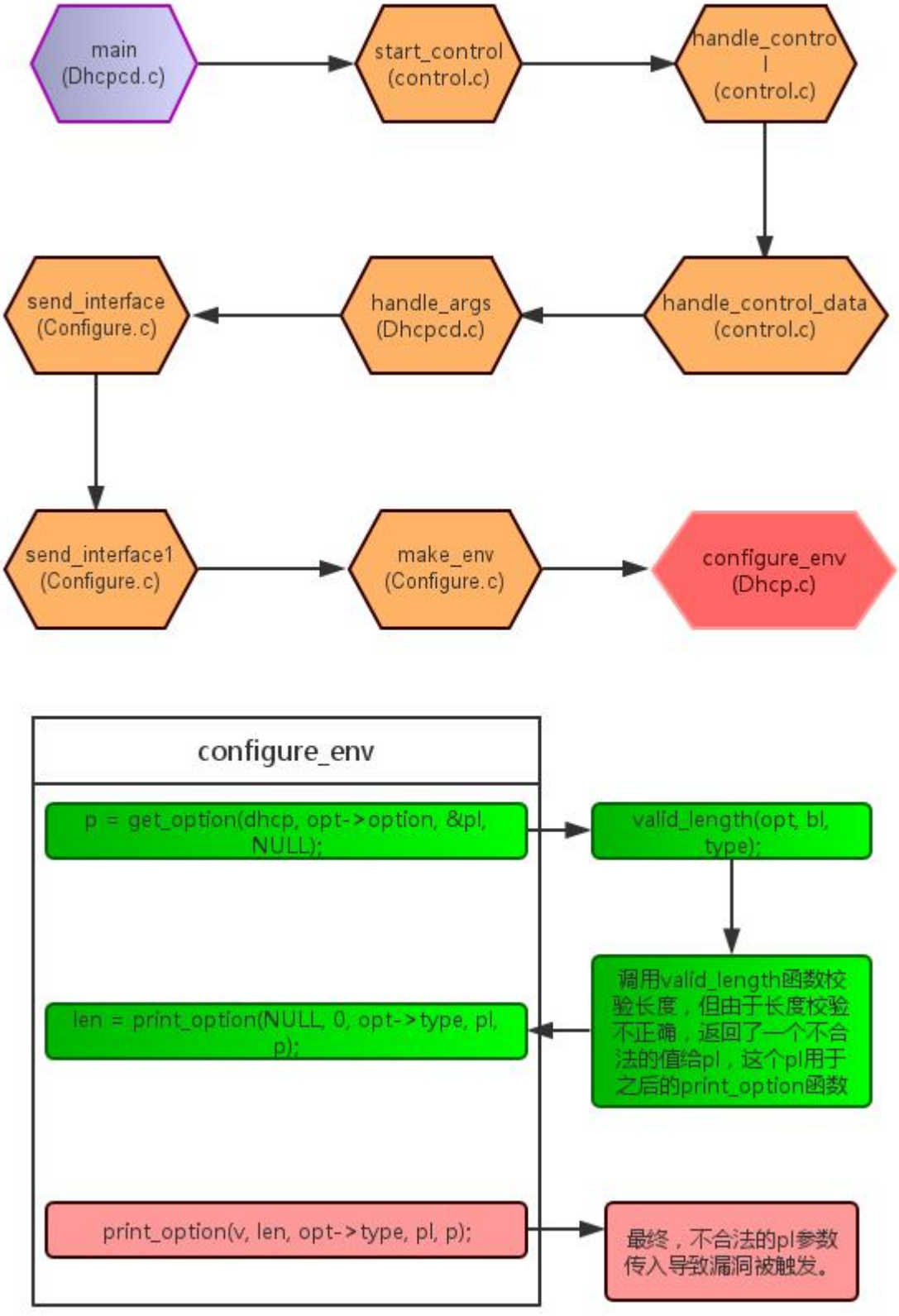


图 2.1

三、测试环境的搭建

由于该漏洞的特殊性，需要开启一个热点，搭建一个 Dhcpd 服务器以及一个带 log 的 dhcpcd 的客户端来验证此漏洞，这里选用的系统为 Ubuntu14.04。

3.1 Hostapd 热点的搭建

安装 Hostapd：
sudo apt-get install hostapd

安装了软件以后，在/etc/hostapd 文件夹中建立一个 hostapd.conf 的文件，在里面写入接入点的信息。
配置 Hostapd：
sudo nano /etc/hostapd/hostapd.conf

hostapd.conf 文件改成如下：
****
interface=wlan0//改成对应的网卡
driver=nl80211//这个 driver 一定得是这个
ssid=baobaonihao
hw_mode=g
channel=10
macaddr_acl=0
auth_algs=3
wpa=2
wpa_passphrase=qqqq1111
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP CCMP
rsn_pairwise=TKIP CCMP
****
注意要自己设置其中的无线热点名称 ssid 和认证密码 wpa_passphrase。

上述配置完成以后，在终端执行 sudo hostapd /etc/hostapd/hostapd.conf -B(-B 是需要在后台运行的时候添加)，到这里，就表明 Hostapd 的安装和配置结束了，现在已经可以在手机终端上可以搜索到这个 baobaonihao 的热点了，但是无法连接到这个热点，此时应该出现的情况是：正在获取 IP 地址，但是一直获取不到。这是由于 dhcpd 服务器没搭建好的原因，接下来就是 dhcpd 服务器的搭建。

3.2 Dhcpd 服务器的编译与搭建

这个 dhcpd 服务器不能直接用 apt-get 来安装，可以在官网 https://www.isc.org/ 里面找到源码并且下载，进行编译安装。

下载之后为一个 dhcp-4.3.4.tar.gz 包。
安装官方原始版本如下：
tar zxvf dhcp-4.3.4.tar.gz

cd dhcp-4.3.4

chmod 777 configure

sudo ./configure

sudo make

sudo make install

安装 debug 版本如下，debug 版本能输出 log，在之后的构建 package 中方便查看调试以及 log 信息：

tar zxvf dhcp-4.3.4.tar.gz

cd dhcp-4.3.4

chmod 777 configure

sudo ./configure –enable-debug

sudo make

sudo make install

这样就可以编译安装一个调试版本了。

同样的，安装了软件以后，在/etc/dhcp 文件夹中建立一个 dhcpd.conf 的配置文件，在里面写入 dhcpd 的配置信息。

dhcpd.conf 文件改成如下：

****

```
ddns-update-style none;
log-facility local7;

subnet 172.20.94.0 netmask 255.255.255.0 {
            option routers                     172.20.94.1;
            option subnet-mask                 255.255.255.0;
            option broadcast-address           172.20.94.255;
          option domain-name "internal.baidu.com";
            option domain-name-servers         172.22.1.253,172.22.1.254;
            option ntp-servers                 172.20.94.1;
            option netbios-name-servers        172.20.94.1;
            option netbios-node-type 2;
            default-lease-time 86400;
            max-lease-time 86400;
          range 172.20.94.0 172.20.94.100;
}
```

在上述配置完成以后我们需要手动给 wlan0 配置 IP 地址并且启动它，在终端输入：

sudo ifconfig wlan0 172.20.94.1

sudo dhcpd /etc/dhcp/dhcpd.conf

就可以启动 dhcpd 了，此时可以获取到 ip 地址了，已经可以成功发包了，如果要上网，则还要输入以下命令：

开启内核 IP 转发

bash -c "echo 1 > /proc/sys/net/ipv4/ip_forward"

开启防火墙 NAT 转发(如果本机使用 eth0 上网,则把 ppp0 改为 eth0)

iptables -t nat -A POSTROUTING -o ppp0 -j MASQUERADE

这样 Dhcpd 服务器的编译与搭建到此就完成了，以及可以修改源码取任意构造数据包了。

3.3 Dhcpcd 客户端增加 log 编译。

采用 4.4.4 版本的源码来编译

log 函数如下：

```
void MYLOG(const char* ms, ...);
void MYLOG(const char* ms, ...)
{
	char wzLog[1024] = {0};
	char buffer[1024] = {0};
	va_list args;
	va_start(args, ms);
	vsprintf( wzLog ,ms,args);
	va_end(args);

	time_t now;
	time(&now);
	struct tm *local;
	local = localtime(&now);
	sprintf(buffer,"%04d-%02d-%02d %02d:%02d:%02d %s\n", local->tm_year+1900,
local->tm_mon,
			local->tm_mday, local->tm_hour, local->tm_min, local->tm_sec,
			wzLog);
	FILE* file = fopen("/data/local/tmp/dhcplog","a+");
	fwrite(buffer,1,strlen(buffer),file);
	fclose(file);
```

```
    // syslog(LOG_INFO,wzLog);
        return ;
}
```

编译的话直接 make 即可编译

四、dhcp 发包交互过程

要想触发该漏洞，当然得了解 dhcp 服务器与客户端之间的交互。那它们之间是怎么样交互的呢？ DHCP 协议采用 UDP 作为传输协议，主机发送请求消息到 DHCP 服务器的 67 号端口，DHCP 服务器回应应答消息给主机的 68 号端口。

1. DHCP Client 以广播的方式发出 DHCP Discover 报文。

2. 所有的 DHCP Server 都能够接收到 DHCP Client 发送的 DHCP Discover 报文，所有的 DHCP Server 都会给出响应，向 DHCP Client 发送一个 DHCP Offer 报文。
   DHCP Offer 报文中“Your(Client) IP Address”字段就是 DHCP Server 能够提供给 DHCP Client 使用的 IP 地址，且 DHCP Server 会将自己的 IP 地址放在“option”字段中以便 DHCP Client 区分不同的 DHCP Server。DHCP Server 在发出此报文后会存在一个已分配 IP 地址的纪录。

3. DHCP Client 只能处理其中的一个 DHCP Offer 报文，一般的原则是 DHCP Client 处理最先收到的 DHCP Offer 报文。
   DHCP Client 会发出一个广播的 DHCP Request 报文，在选项字段中会加入选中的 DHCP Server 的 IP 地址和需要的 IP 地址。

4. DHCP Server 收到 DHCP Request 报文后，判断选项字段中的 IP 地址是否与自己的地址相同。如果不相同，DHCP Server 不做任何处理只清除相应 IP 地址分配记录；如果相同，DHCP Server 就会向 DHCP Client 响应一个 DHCP ACK 报文，并在选项字段中增加 IP 地址的使用租期信息。

5. DHCP Client 接收到 DHCP ACK 报文后，检查 DHCP Server 分配的 IP 地址是否能够使用。如果可以使用，则 DHCP Client 成功获得 IP 地址并根据 IP 地址使用租期自动启动续延过程；如果 DHCP Client 发现分配的 IP 地址已经被使用，则 DHCP Client 向 DHCPServer 发出 DHCP Decline 报文，通知 DHCP Server 禁用这个 IP 地址，然后 DHCP Client 开始新的地址申请过程。

6. DHCP Client 在成功获取 IP 地址后，随时可以通过发送 DHCP Release 报文释放自己的 IP 地址，DHCP Server 收到 DHCP Release 报文后，会回收相应的 IP 地址并重新分配。

可以得知 DHCP　Server 会向 DHCP Client 响应一个 DHCP ACK 报文，而这个 ACK 报文能触发到漏洞代码片段，而需要知道的就是如何构建自己的 DHCP ACK 报文。查看 Server 端的代码，找到处理客户端报文的函数为 void dhcp (struct packet *packet) ，代码片段如下：

```
switch (packet -> packet_type) {
      case DHCPDISCOVER:
    dhcpdiscover (packet, ms_nulltp);
    break;

      case DHCPREQUEST:
    dhcprequest (packet, ms_nulltp, lease);
    break;

      case DHCPRELEASE:
    dhcprelease (packet, ms_nulltp);
    break;

      case DHCPDECLINE:
    dhcpdecline (packet, ms_nulltp);
    break;

      case DHCPINFORM:
    dhcpinform (packet, ms_nulltp);
    break;

      case DHCPLEASEQUERY:
    dhcpleasequery(packet, ms_nulltp);
    break;

      case DHCPACK:
      case DHCPOFFER:
      case DHCPNAK:
      case DHCPLEASEUNASSIGNED:
      case DHCPLEASEUNKNOWN:
      case DHCPLEASEACTIVE:
    break;

      default:
    errmsg = "unknown packet type";
    goto ↑bad_packet;
} « end switch packet->packet_type »
```

图 4.1

对应的 ACK 报文当然是 DHCPREQUEST，继续查看 dhcprequest 函数，在结尾处找到:

```
/* Otherwise, send the lease to the client if we found one. */
if (lease) {
    ack_lease (packet, lease, DHCPACK, 0, msgbuf, ms_nulltp,
           (struct host_decl *)0);
} else
    log_info ("%s: unknown lease %s.", msgbuf, piaddr (cip));

  out:
if (subnet)
    subnet_dereference (&subnet, MDL);
if (lease)
    lease_dereference (&lease, MDL);
return;
```

图 4.2

可以发现，是 ack_lease (packet, lease, DHCPACK, 0, msgbuf, ms_nulltp,(struct host_decl *)0);这个函数，继续追踪，还是在尾部：

```
#if defined(DELAYED_ACK) && !defined(DHCP4o6)
        if (enqueue)
            delayed_ack_enqueue(lease);
        else
#endif
            dhcp_reply(lease);
    }
} « end ack_lease »
```

图 4.3

进入 dhcp_reply(lease),继续追踪：

```
/* Insert such options as will fit into the buffer. */
packet_length = cons_options (state -> packet, &raw, lease,
                  (struct client_state *)0,
                  state -> max_message_size,
                  state -> packet -> options,
                  state -> options, &global_scope,
                  bufs, nulltp, bootpp,
```

图 4.4

进入 cons_options 函数，还是在函数的尾部发现：

```
memcpy(outpacket->options, buffer, index);

length = DHCP_FIXED_NON_UDP + index;

return length;
```

这里的 buffer 就是储存数据包中 options 字段的地方了，在这个 memcpy 之前改写这个 buffer，再对应的把 index 改成数据包中 options 字段实际的长度就可以了。

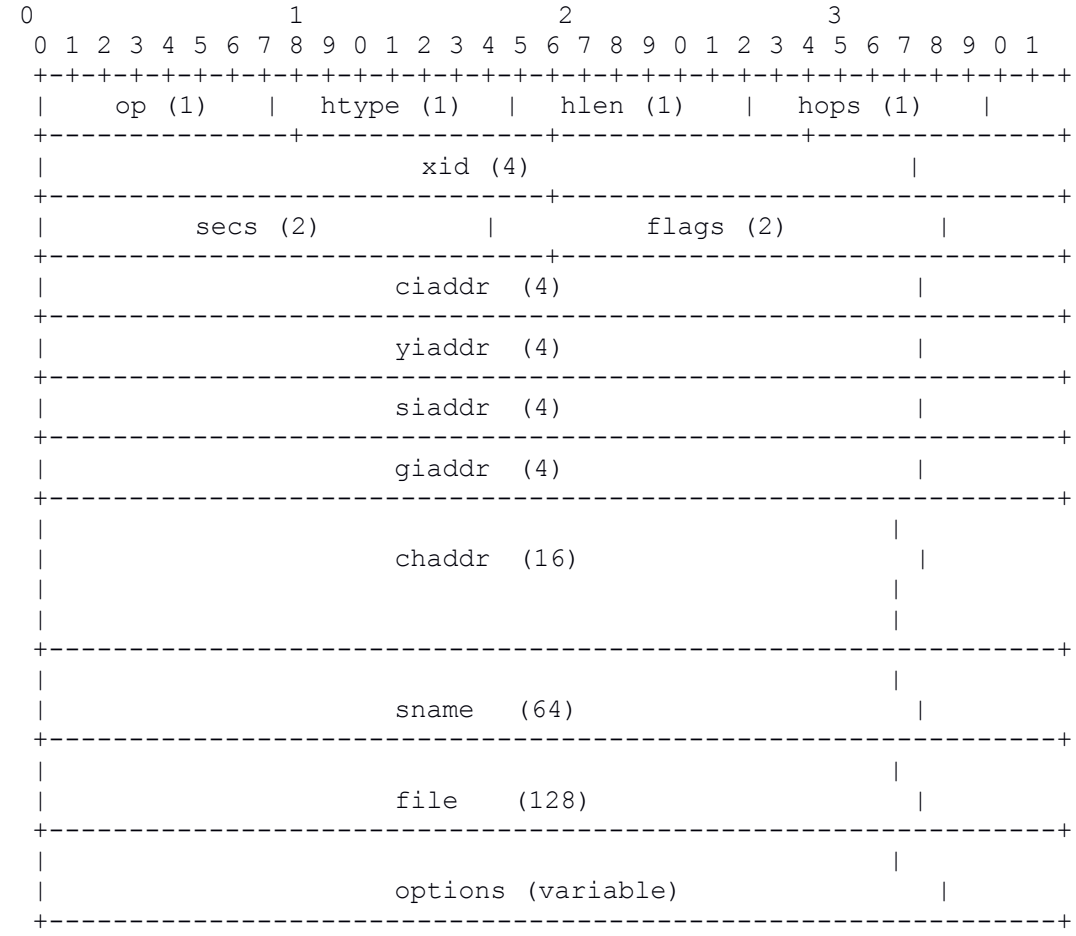
五、发包验证

可以先发个包熟悉一下格式是什么样的（Hex 格式）：

```
000: 02 01 06 00 6a 3e 05 14  00 01 00 00 00 00 00 00   ....j>.....
010: ac 14 5e 52 00 00 00 00  00 00 00 00 cc fa 00 b1   ..^R.....
020: a5 70 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .p.....
030: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
040: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
050: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
060: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
070: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
080: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
090: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
0a0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
0b0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
0c0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
0d0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   .....
0e0: 00 00 00 00 00 00 00 00  00 00 00 00 63 82 53 63   .....c.Sc
0f0: 35 01 05 36 04 ac 14 5e  01 33 04 00 01 3b e3 01   5..6...^.3...;..
100: 04 ff ff ff 00 03 04 ac  14 5e 01 06 04 08 08 08   .....^.....
110: 08 19 13 69 6e 74 65 72  6e 61 6c 2e 62 61 69 64   ...iaternxl.bai.
120: 75 63 2e 63 6f 6d 1c 04  ac 14 5e ff ff
```

图 5.1

图 5.1 表示的是一个完整的 DHCP ACK 数据包，这个数据包的格式如下：



具体参数含义请参考 rfc2131[1]文档，这里关注的是最后一个字段 options，这个字段就是导致漏洞触发的关键点。

为了顺利的利用长度校验不合理的这个缺陷，可以把 opt->type 设置成 UINT16 和 ARRAY，查找一下这种 type 的 option 在客户端的源码对应的 option 号码，如下：

```
{ 24,   UINT32,         "path_mtu_aging_timeout" },
{ 25,   UINT16 | ARRAY, "path_mtu_plateau_table" },
{ 26,   UINT16,         "interface_mtu" },
```

图 5.2

option 的号码为 25，为了清楚 25 这个 option 的格式，查看 rfc2132[2]文档找到描述如下：

4.7. Path MTU Plateau Table Option

```
This option specifies a table of MTU sizes to use when performing
Path MTU Discovery as defined in RFC 1191.  The table is formatted as
a list of 16-bit unsigned integers, ordered from smallest to largest.
The minimum MTU value cannot be smaller than 68.

The code for this option is 25.  Its minimum length is 2, and the
length MUST be a multiple of 2.

 Code   Len     Size 1     Size 2
+-----+-----+-----+-----+-----+-----+---
|  25 |  n  |  s1 |  s2 |  s1 |  s2 | ...
+-----+-----+-----+-----+-----+-----+---
```

从上述的描述可以得知最小长度 n 规定为 2，且长度为 2 的整数倍。但是可以构建一个 option 为 25 长度为 3 的数据包，既可以在长度校验函数中返回 1 从而通过校验，又能进入 print_option 函数中的 while 循环，当进入 while 循环后如下:

```
t = data;
e = data + dl;
while (data < e) {

……

else if (type & UINT16) {
        memcpy(&u16, data, sizeof(u16));
        u16 = ntohs(u16);
        l = snprintf(s, len, "%d", u16);
        data += sizeof(u16);
    }

……


}
```

dl 为 3，而 data 每次循环后只加了 2，导致了此 while 循环多循环了一次，memcpy 多 copy 了一次 2 字节，造成了越界。至此整个漏洞就分析完毕。

By Ericky

2016.06.06

参考链接：

[1] https://tools.ietf.org/html/rfc2131

[2] https://tools.ietf.org/html/rfc2132

[3] https://android.googlesource.com/platform/external/dhcpcd/+/1390ace71179f04a09c300ee8d0300aa69d9db09

[4] http://source.android.com/security/bulletin/2016-04-02.html

[5] http://www.isc.org/downloads/

[6] https://help.ubuntu.com/community/isc-dhcp-server