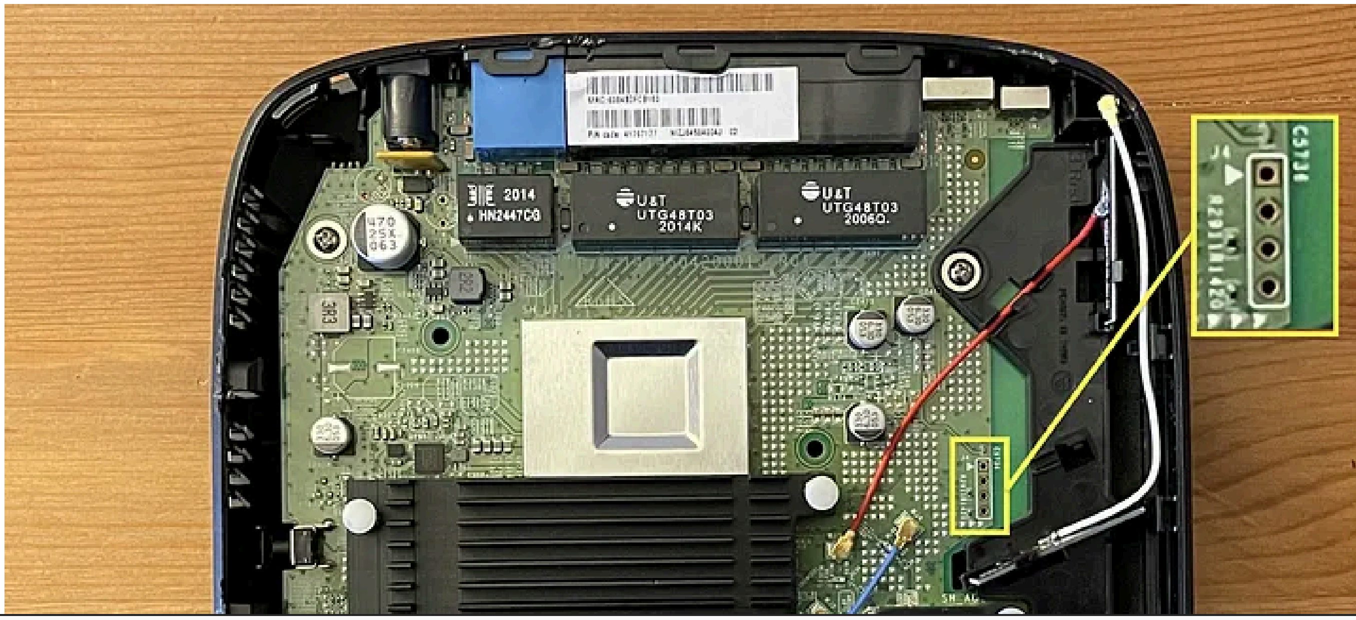
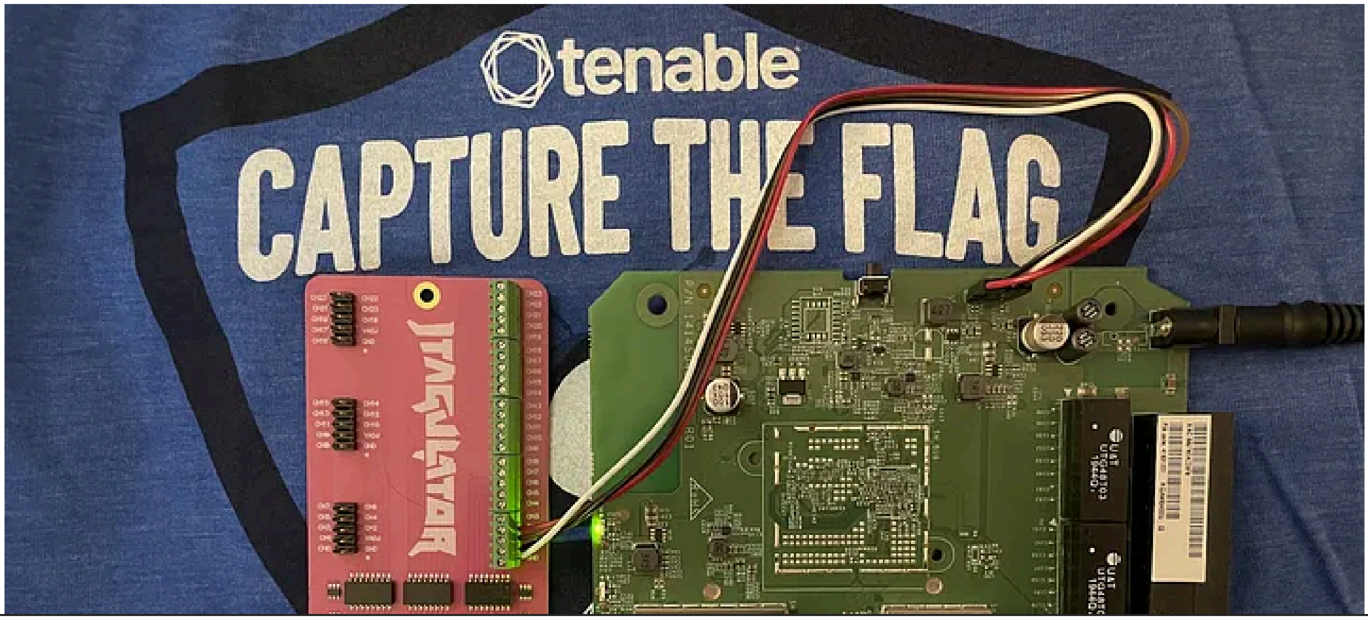
Bypassing Authentication on Arcadyan Routers with CVE-2021–20090 and rooting some Buffalo

At the end, we will also take a quick look at how I discovered that the authentication bypass vulnerability was not limited to the Buffalo routers, and how it affects at least a dozen other models from multiple vendors spanning a period of over ten years.

Root shells on UART

It is fairly common for devices like these Buffalo routers to offer up a shell via a serial connection known as Universal Asynchronous Receiver/Transmitter (UART) on the circuit board. Manufacturers often leave


2014
HN2447CG
U&T
UTG48T03
2014K
U&T
UTG48T03


tenable
CAPTURE THE FLAG
JTAGULATOR


COM3 - PuTTY
Welcome to JTAGulator. Press 'H' for available commands.
:V
Current target I/O voltage: Undefined
Enter new target I/O voltage (1.2 - 3.3, 0 for off): 3.3

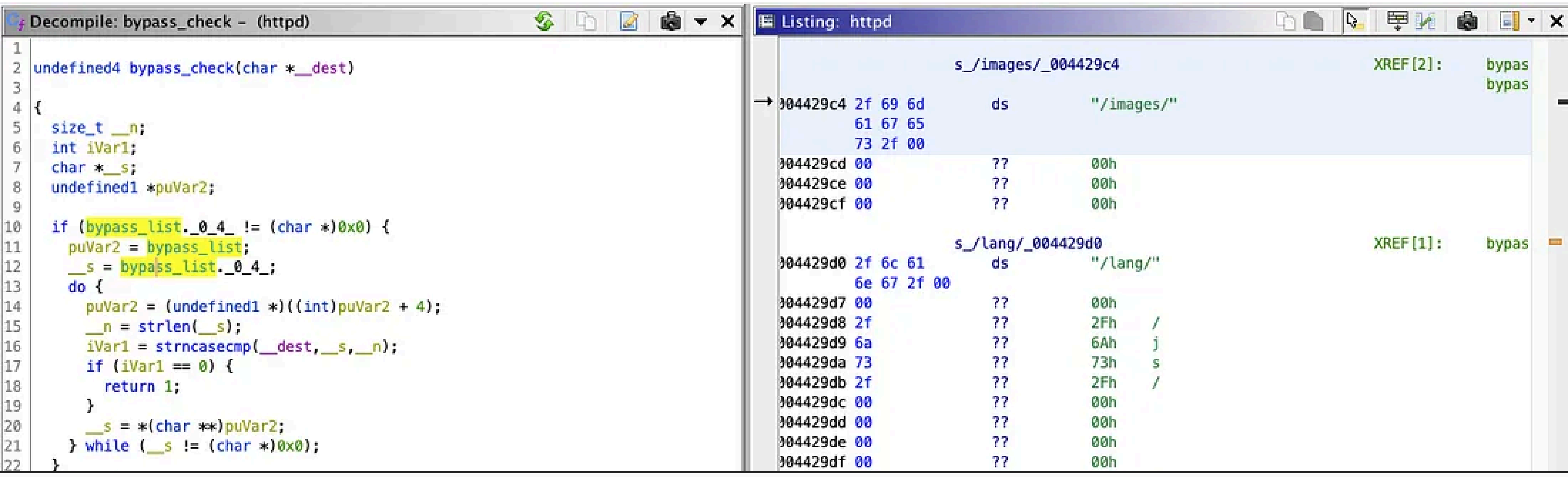
We can now use this shell to explore the device, and transfer any interesting binaries to another machine for analysis. In this case, we grabbed the httpd binary which was serving the device's web interface.

Httpd and web interface authentication

Having access to the httpd binary makes hunting for vulnerabilities in the web interface much easier, as we can throw it into Ghidra and identify any interesting pieces of code. One of the first things I tend to look at when analyzing any web application or interface is how it handles authentication.

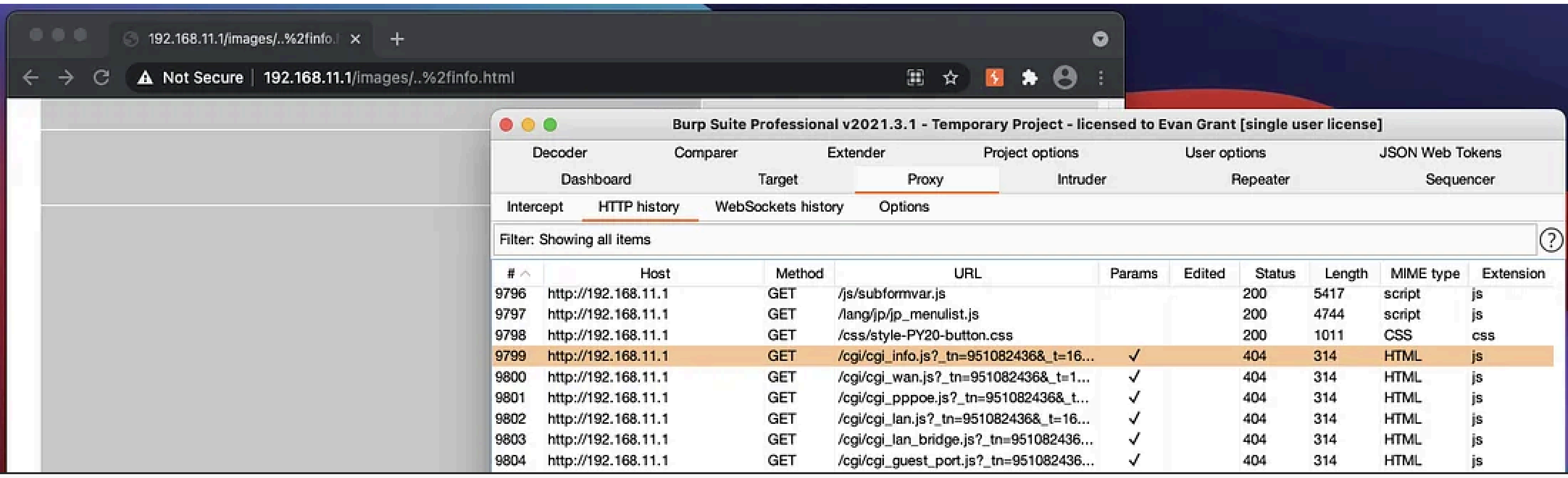
```
iVar1 = get_tid();
if (iVar1 != -1) {
  iVar1 = mapi_ccfg_match_str(iVar1,"ARC_SYS_LogEnable","1");
  if (iVar1 != 0) {
    get_mac_by_ip(param_1 + 0x18d8,&local_28);
  }
  iVar2 = FUN_0041f9d0(*(undefined4 *)(param_1 + 0x1898),*(undefined4 *)(param_1 + 0x189c),
                       *(undefined4 *)(param_1 + 0x18a0),*(undefined4 *)(param_1 + 0x18a4),
                       *(undefined4 *)(param_1 + 0x18a8),*(undefined4 *)(param_1 + 0x18ac),
                       *(undefined4 *)(param_1 + 0x18b0));
  if (iVar2 == 1) {
    iVar2 = strcmp((char *)(param_1 + 0x1b7c),"/apply.cgi");
    if (iVar2 == 0) {
      printf("[%s] %s, Buffalo DDNS request should not timeout.\n","check_auth",param_1 + 0x18d8);
      return 0;
    }
    printf("[%s] %s login time out, reauth\n","check_auth",param_1 + 0x18d8);
    httoken_entry_remove(1);
    snprintf(acStack1064,0x400,"Location: /login.html\n\n");
  }
  else {
    if (iVar2 == 2) {
      printf("[%s] new user %s(%s) comes to login, check user and auth\n","check_auth",
             param_1 + 0x18d8,param_1 + 0x4c);
```

```
Decompile: process_request -  (httpd)
      uVar1 = bypass_check(__dest);
      *(undefined4 *)(param_1 + 0x1890) = uVar1;
      if (((*(code **)(v_ops + 0x14) == (code *)0x0) ||
          (iVar3 = (**(code **)(v_ops + 0x14))(param_1), iVar3 != 2)) &&
         ((*(int *)(param_1 + 0x1890) != 0 ||
          (iVar3 = evaluate_access(__dest,0,param_1), iVar3 == 0)))) {
        *(undefined4 *)(param_1 + 0x1b74) = 0;
        pcVar2 = *(char **)(param_1 + 0x1d00);
        iVar3 = strcmp(pcVar2,"HEAD");
        if (iVar3 == 0) {
          *(undefined4 *)(param_1 + 0x1b74) = 1;
          if (*(int *)(param_1 + 0x1b70) == 0) {
            process_get(param_1);
          }
          else {
            *(undefined4 *)(param_1 + 0x1b74) = 0;
            answer(param_1,400,"Invalid HTTP/0.9 method.");
          }
        }
        else {
          iVar3 = strcmp(pcVar2,"GET");
          if (iVar3 == 0) {
            process_get(param_1);
          }
```


Decompile: bypass_check - (httpd)
undefined4 bypass_check(char *__dest)
{
size_t __n;
int iVar1;
char *__s;
undefined1 *puVar2;
if (bypass_list._0_4_ != (char *)0x0) {
puVar2 = bypass_list;
__s = bypass_list._0_4_;
do {
puVar2 = (undefined1 *)((int)puVar2 + 4);
__n = strlen(__s);
iVar1 = strncasecmp(__dest,__s,__n);
if (iVar1 == 0) {
return 1;
}
__s = *(char **)puVar2;
} while (__s != (char *)0x0);
Listing: httpd
s_/images/_004429c4
XREF[2]:
"/images/"
s_/lang/_004429d0
XREF[1]:
"/lang/"

Glancing at the bypass list we see login.html, loginerror.html and some other paths/pages, which makes sense as even unauthenticated users will need to be able to access those urls.

You may have already noticed the bug here. **bypass_check()** is only checking as many bytes as are in the **bypass_list** strings. This means that if a user is trying to reach http://router/images/someimage.png, the comparison will match since **/images/** is in the bypass list, and the url we are trying to reach begins with **/images/**. The **bypass_check()** function doesn't care about strings which come after, such as "someimage.png". So what if we try to reach


192.168.11.1/images/..%2finfo.
Not Secure | 192.168.11.1/images/..%2finfo.html
Burp Suite Professional v2021.3.1 - Temporary Project - licensed to Evan Grant [single user license]
Decoder Comparer Extender Project options User options JSON Web Tokens
Dashboard Target Proxy Intruder Repeater Sequencer
Intercept HTTP history WebSockets history Options
Filter: Showing all items
Host Method URL Params Edited Status Length MIME type Extension
9796 http://192.168.11.1 GET /js/subformvar.js 200 5417 script js
9797 http://192.168.11.1 GET /lang/jp/jp_menulist.js 200 4744 script js
9798 http://192.168.11.1 GET /css/style-PY20-button.css 200 1011 CSS css
9799 http://192.168.11.1 GET /cgi/cgi_info.js?_tn=951082436&_t=16... ✓ 404 314 HTML js
9800 http://192.168.11.1 GET /cgi/cgi_wan.js?_tn=951082436&_t=1... ✓ 404 314 HTML js
9801 http://192.168.11.1 GET /cgi/cgi_pppoe.js?_tn=951082436&_t... ✓ 404 314 HTML js
9802 http://192.168.11.1 GET /cgi/cgi_lan.js?_tn=951082436&_t=16... ✓ 404 314 HTML js
9803 http://192.168.11.1 GET /cgi/cgi_lan_bridge.js?_tn=951082436... ✓ 404 314 HTML js
9804 http://192.168.11.1 GET /cgi/cgi_guest_port.js?_tn=951082436... ✓ 404 314 HTML js

are generated will be discussed in the next section. For now, let's focus on why these particular requests are failing.

Looking at httpd in Ghidra shows that there is a fair amount of debugging output printed when errors occur. Stopping the default httpd process, and running it from our shell shows that we can easily see this output which may help us identify the issue with the current request.

```
[httpd_main] Assigned thread 0 to process
[httpd_child] thread 0 wake up
```

While we know that the **httoken**s are grabbed at some point on the pages we access, we don't know where they're coming from or how they're generated. This will be important to understand if we want to carry this exploitation further, since they are required to do or access anything sensitive on the device. Tracking down how the web interface produces these tokens felt like something out of a Capture-the-Flag event.

The info.html page we accessed with the path traversal was populating its information table with data from .js files under the /cgi/ directory, and was

getToken() is getting data from this spacer img tag

authentication bypass to access some actions (like making configuration changes).

Notably, on the WSR-2533DHPL2 just using this knowledge of the tokens means we can access the administrator password for the device, a vulnerability which appears to already be fixed on the WSR-2533DHP3 (despite both having firmware releases around the same time).

Now that we know we can effectively perform any action on the device without being authenticated, let's see what we can do with that.

After the request is made successfully, **ARC_ping_ipaddress** is stored in the

ARC_ping_ipaddress parameter. There are a number of options seen in the configuration file, but one which caught my attention was **ARC_SYS_TelnetdEnable=0**. Enabling telnetd seemed like a good candidate for gaining a remote shell on the device.

It was unclear whether simply injecting the configuration file with **ARC_SYS_TelnetdEnable=1** would work, as it would then be followed by a conflicting setting later in the file (as **ARC_SYS_TelnetdEnable=0** appears lower in the configuration file than **ARC_ping_ipdaddress**). However, after sending the following request in Burp Suite, and sending a reboot request

Here are the pieces we need to put together in a python script if we want to make exploiting this super easy:

- Get proper **httoken**s from the img tags on a page.
- Use those **httoken**s in combination with the path traversal to make a valid request to **apply_abstract.cgi**
- **In that valid request to apply_abstract.cgi, inject the ARC_SYS_TelnetdEnable=1** configuration option

Shortly before the 90 day disclosure date for the vulnerabilities discussed in this blog, I was trying to determine the number of potentially affected devices visible online via Shodan and BinaryEdge. In my searches, I noticed that a number of devices which presented similar web interfaces to those seen on the Buffalo devices. Too similar, in fact, as they appeared to use almost all the same strange methods for hiding the **httoken**s in img tags, and javascript functions obfuscated in "enkripsi" strings.

The common denominator is that all of the devices were manufactured by

Coordination Center for help with that process. A list of the affected devices can be found in either **Tenable's own advisory**, and more information can be found on **CERT's page** tracking the issue.

There is a much larger conversation to be had about how this vulnerability in Arcadyan's firmware has existed for at least 10 years and has therefore found its way through the supply chain into at least 20 models across 17 different vendors, and that is touched on in a whitepaper Tenable has released.

researchers not living in the country where they are sold/provided by a local ISP.

Thanks for reading, and happy hacking!

Tenable Research · Internet Of Things · Hacking · Bug Bounty · Vulnerability