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

Evan Grant · Follow

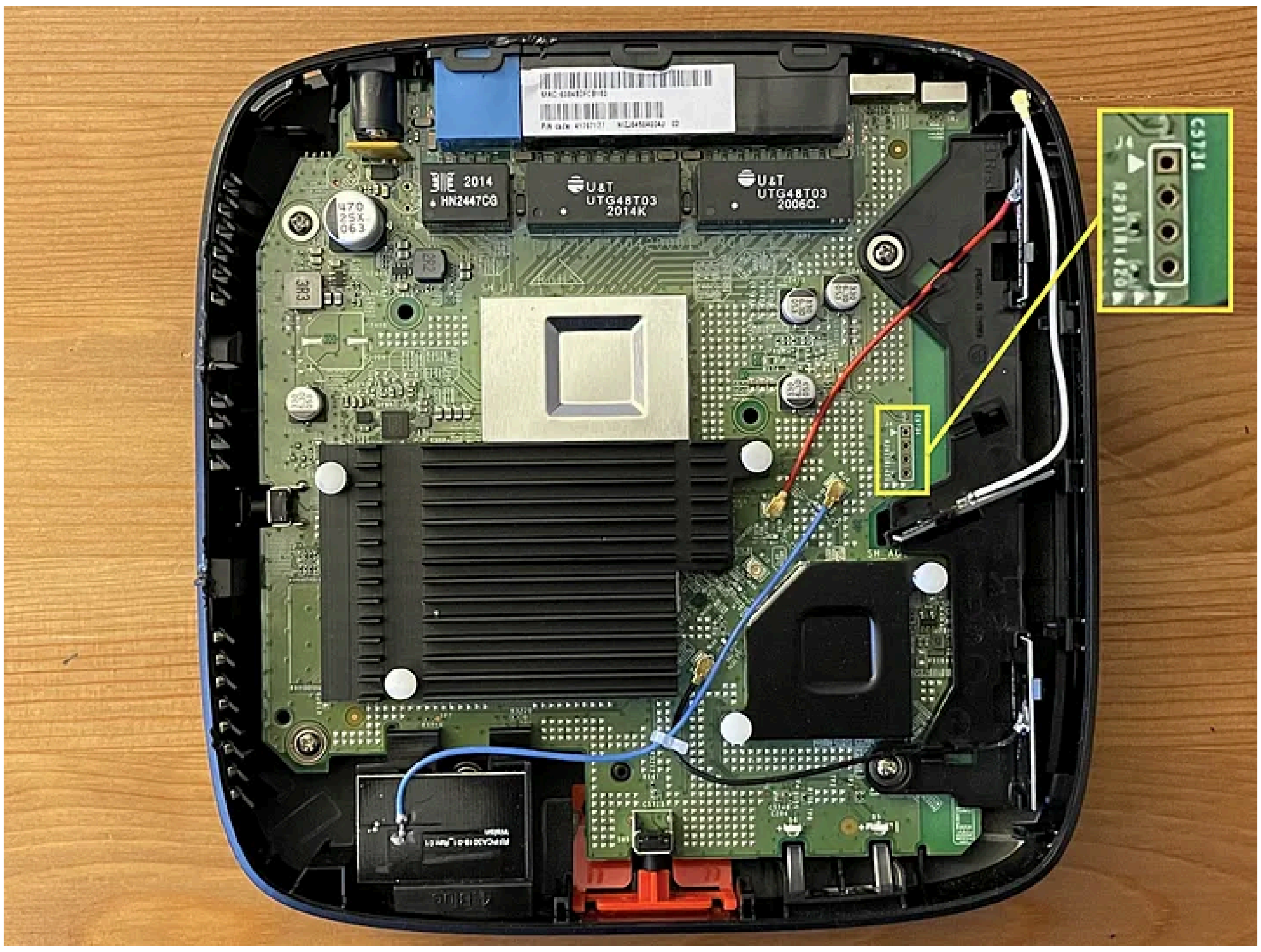
Published in Tenable TechBlog · 13 min read · Aug 3, 2021

A while back I was browsing Amazon Japan for their best selling networking equipment/routers (as one does). I had never taken apart or hunted for vulnerabilities in a router and was interested in taking a crack at it. I came across the **Buffalo WSR-2533DHP3** which was, at the time, the third best selling device on the list. Unfortunately, the sellers didn't ship to Canada, so I instead bought the closely related **Buffalo WSR-2533DHPL2** (though I eventually got my hands on the **WSR-2533DHP3** as well).

In the following sections we will look at how I took the Buffalo devices apart, did a not-so-great solder job, and used a shell offered up on UART to help find a couple of bugs that could let users bypass authentication to the web interface and enable a root BusyBox shell on telnet.

At the end, we will also take a quick look at how I discovered that the

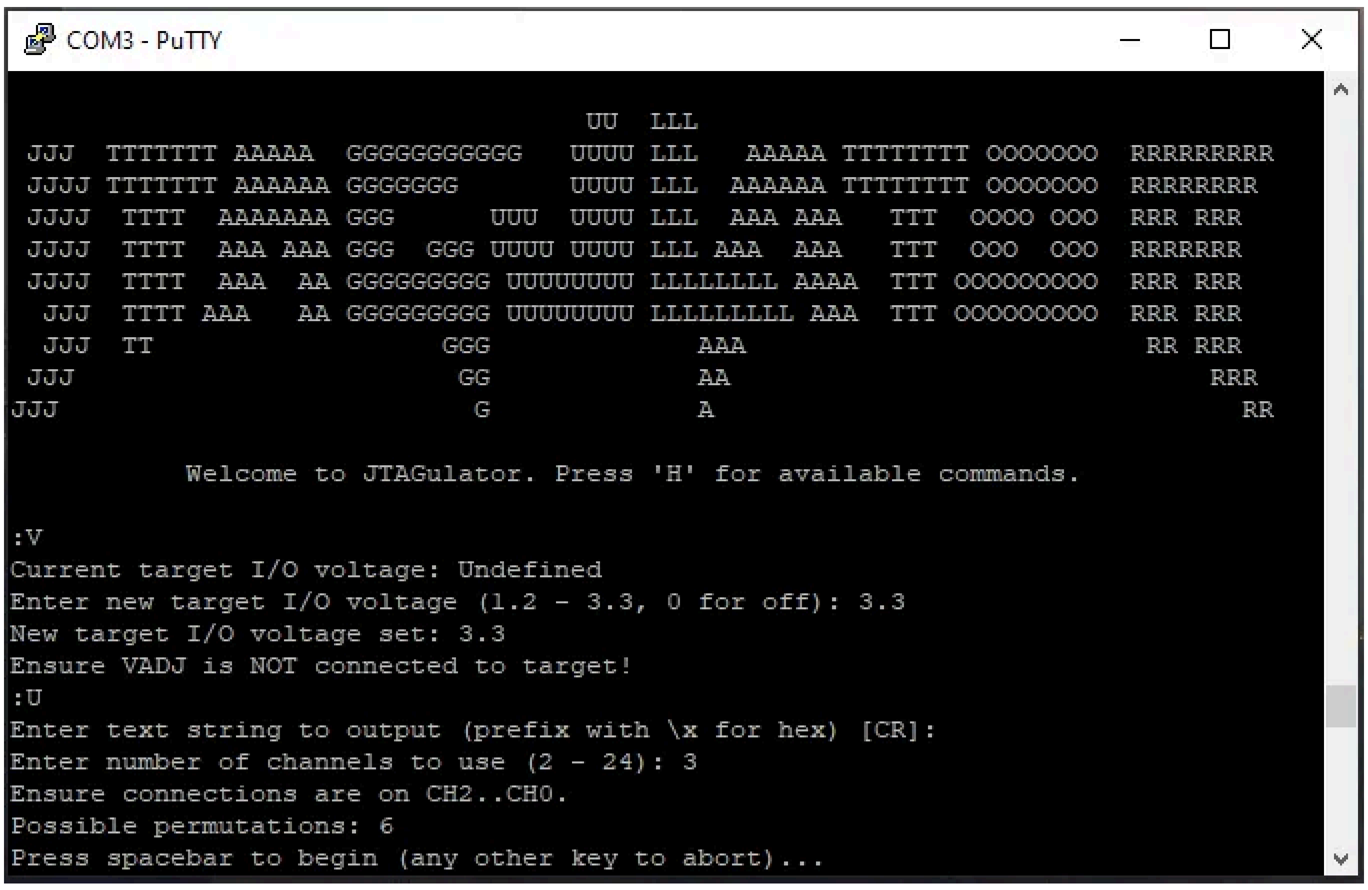
The first step is for us to open up the router's case and try to identify if there is



UART interface on the WSR-2533DHP3

We can see a header labeled J4 which may be what we're looking for. The next step is to test the contacts with a multimeter to identify power (VCC), ground (GND), and our potential transmit/receive (TX/RX) pins. Once we've identified those, we can solder on some pins and connect them to a tool like JTAGulator to identify which pins we will communicate on, and at what baud rate.

We could identify this in other ways, but the JTAGulator makes it much ea… er ea… r some other specified bytes) and receiving on each pin, at different bauds, which helps us identify what combination thereof will let us communicate with the device.



```
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
New target I/O voltage set: 3.3
Ensure VADJ is NOT connected to target!
:U
Enter text string to output (prefix with \x for hex) [CR]:
Enter number of channels to use (2 - 24): 3
Ensure connections are on CH2..CH0.
Possible permutations: 6
Press spacebar to begin (any other key to abort)...
```

Running a UART scan on JTAGulator

The UART scan shows that sending a carriage return over pin 0 as TX, with pin 2 as RX, and a baud of 57600, gives an output of **BusyBox v1**, which looks like we may have our shell.

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.

While examining the web interface I noticed that, even after logging in, no session cookies are set, and no tokens are stored in local/session storage, so how was it tracking who was authenticated? Opening httpd up in Ghidra, we find a function named **evaluate_access()** which leads us to the following snippet:

```
iVar1 = get_tid();
```

FU

making the current request matches that of an IP from a previous valid login.

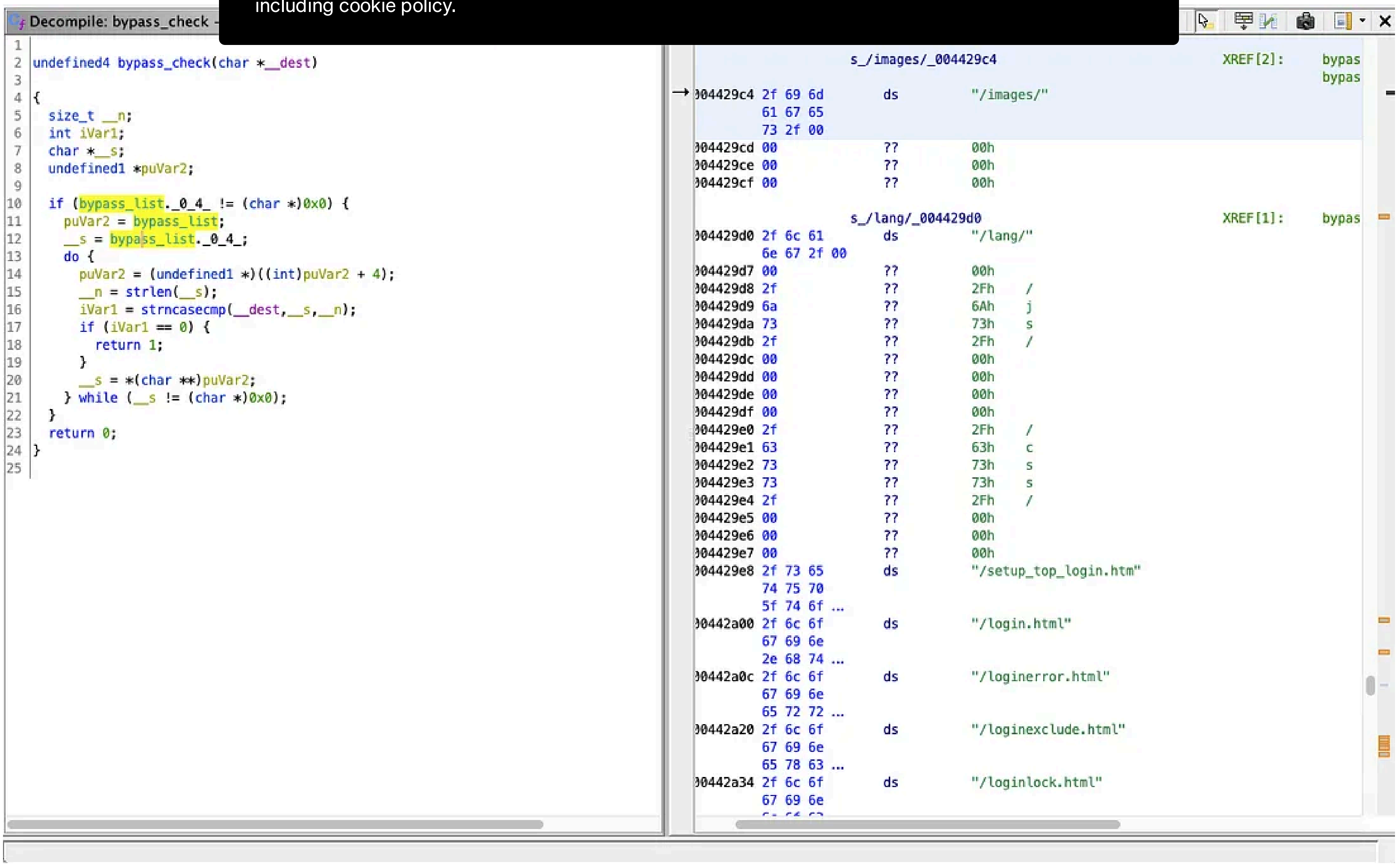
Now that we know what **evaluate_access()** does, lets see if we can get around it. Searching for where it is referenced in Ghidra we can see that it is only called from another function **process_request()** which handles any incoming HTTP requests.

Decompile: process_request – (httpd)

```
43      uVar1 = bypass_check(__dest);
44      *(undefined4 *)(param_1 + 0x1890) = uVar1;
45      if (((*(code **)(v_ops + 0x14) == (code *)0x0) ||
46          (iVar3 = (**(code **)(v_ops + 0x14))(param_1), iVar3 != 2)) &&
47         ((*(int *)(param_1 + 0x1890) != 0 ||
48          (iVar3 = evaluate_access(__dest,0,param_1), iVar3 == 0)))) {
49        *(undefined4 *)(param_1 + 0x1b74) = 0;
50        pcVar2 = *(char **)(param_1 + 0x1d00);
51        iVar3 = strcmp(pcVar2,"HEAD");
52        if (iVar3 == 0) {
53          *(undefined4 *)(param_1 + 0x1b74) = 1;
54          if (*(int *)(param_1 + 0x1b70) == 0) {
55            process_get(param_1);
56          }
57          else {
58            *(undefined4 *)(param_1 + 0x1b74) = 0;
59            answer(param_1,400,"Invalid HTTP/0.9 method.");
60          }
61        }
62        else {
63          iVar3 = strcmp(pcVar2,"GET");
64          if (iVar3 == 0) {
65            process_get(param_1);
66          }
67          else {
68            iVar3 = strcmp(pcVar2,"POST");
69            if (iVar3 == 0) {
70              process_post();
71            }
72            else {
73              answer(param_1,400,"Invalid or unsupported method.");
74            }
75          }
```

process_request() deciding if it should allow the user access to a page

Something which immediately stands out is the logical OR in the larger **if** statement (lines 45–48 in the screenshot above) and the fact that it checks the value of **uVar1** (set on line 43) before checking the output of

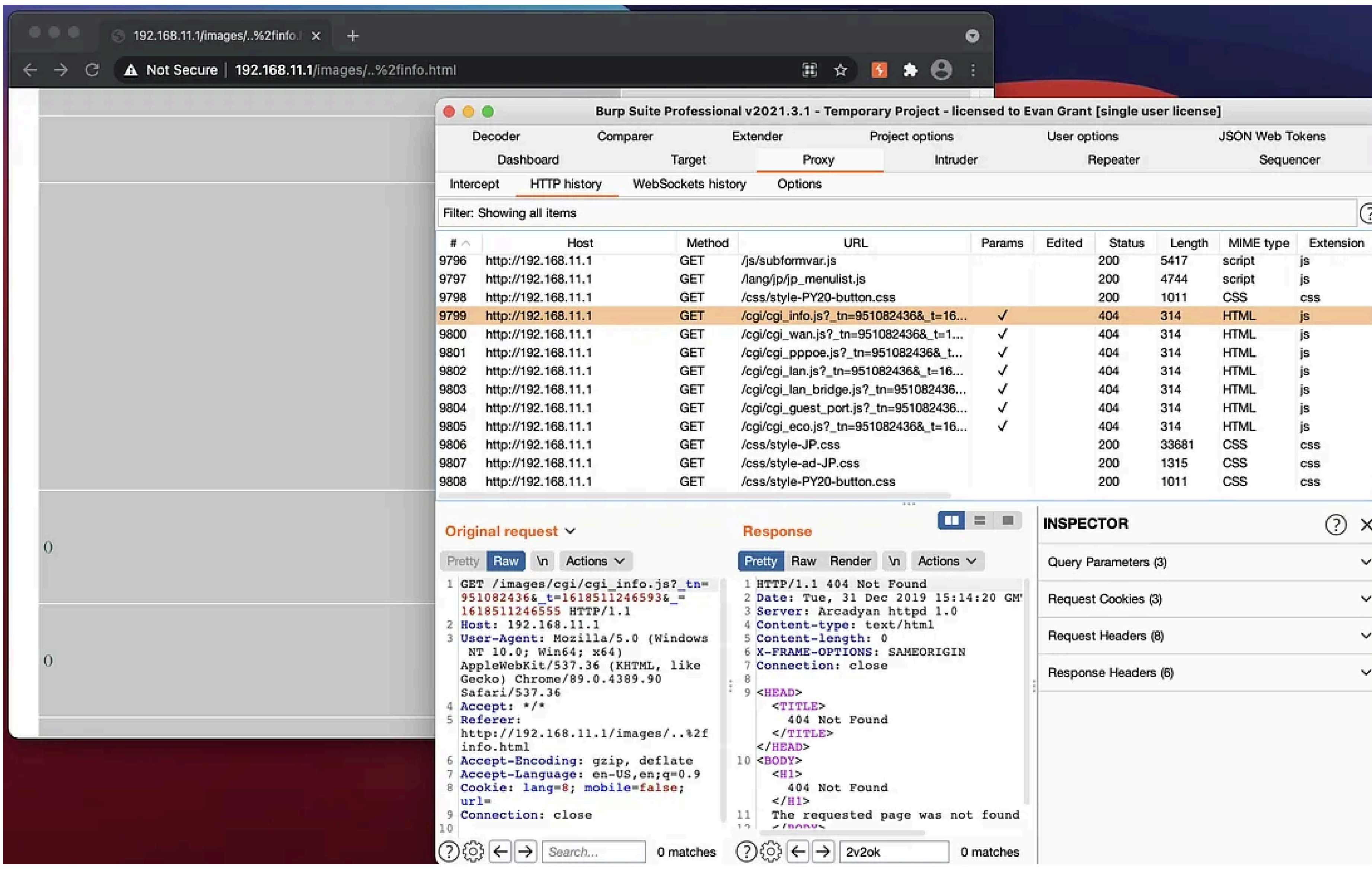


the bypass_list checked in bypass_check()

Taking a look at **bypass_check()** in the screenshot above, we can see that it is looping through **bypass_list**, and comparing the first **n** bytes of **_dest** to a string from **bypass_list**, where **n** is the length of the string grabbed from **bypass_list.** If no match is found, we return 0 and will be required to pass the checks in **evaluate_access()**. However, if the strings match, then we don't care about the result of **evaluate_access()**, and the server will process our

After a bit of match/replace to account for relative paths, we still see an un

us



404s for requests to made to js files

Looking at the Burp traffic, we can see a number of requests to **/cgi/<various_nifty_cgi>.js** are returning a 404, which normally return all of the info we're looking for. We also see that there are a couple of parameters requests failing due to improper Referrer header

W *is invalid from http://192.168.11.1/images/..%2finfo.html*. We will dive into **httoken**s next, but the token we have here is correct, which means that the part causing the failure is the “from” url, which corresponds to the **Referer** header in the request. So, if we create a quick match/replace rule in Burp Suite to fix the **Referer** header to remove the **/images/..%2f** then we can see the info table, confirming our ability to bypass authentication.

our content loaded :)

A quick summary of where we are so far:

- We can bypass authentication and access pages which should be

The info.html page we accessed with the path traversal was populating its in… pa… **httoken** we’re trying to figure out.

We can see that the links used to grab the info from /cgi/ are generated using the **URLToken()** function, which sets the **httoken** (the parameter **_tn** in this case) using the function **get_token()**, but **get_token()** doesn’t seem to be defined anywhere in any of the scripts used on the page.

Looking right above where **URLToken()** is defined we see this strange string defined.

Looking into where it is used, we find the following snippet.

Which, when run adds the following script to the page:

We've found our missing **getToken()** function, but it looks to be doing so … an …

page (with differing encoded strings). What is going on here?

getToken() is getting data from this spacer img tag

The **httoken**s are being grabbed from these spacer img src strings and are used to make requests to sensitive resources.

We can find a function where the **httoken** is being inserted into the img tag in Ghidra.

We can use the tokens for any requests we need as long as the token and the Re

pa

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.

Injecting configuration options and enabling telnetd

One of the first places I check for any web interface / application which has utilities like a **ping** function is to see how those utilities are implemented, because even just a quick Google turns up a number of historic examples of router ping utilities being prone to command injection vulnerabilities.

While there wasn't an easily achievable command injection in the ping command, looking at how it is implemented led to another vulnerability. When the ping command is run from the web interface, it takes an input of the host to ping.

With this in mind, we can take a look at any interesting configuration settings we see, and hope that we're able to overwrite them by injecting 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 (which is necessary for certain configuration changes to take effect).

- Use those **httokens** in combination with the path traversal to make a

- **In that valid request to apply_abstract.cgi, inject the ARC_SYS_TelnetdEnable=1** configuration option
- Send another valid request to reboot the device

Running a quick PoC against the WSR-2533DHPL2

Surprise: More affected devices

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

On April 21st, 2021, Tenable reported CVE-2021–20090 to four additional

ve

Ar

vendors were affected and contacting and tracking them all would become very difficult, and so on May 18th, Tenable reported the issues to the **CERT 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.

Takeaways

The Buffalo WSR-2533DHPL2 was the first router I'd ever purchased for the purpose of discovering vulnerabilities, and it was a super fun experience. The strange obfuscations and simplicity of the bugs made it feel like my own personal CTF. While I got a little more than I bargained for upon learning how widespread one of the vulnerabilities (CVE-2021–20090) was, it was an important lesson in how one should approach research on consumer electronics: The vendor selling you the device is not necessarily the one who manufactured it, and if you find bugs in a consumer router's firmware, they could potentially affect many more vendors and devices than just the one you are researching.

I'd also like to encourage security researchers who are able to get their hands

Written by Evan Grant

75 Followers · Writer for Tenable TechBlog

https://twitter.com/stargravy