BLOG: INTERNET OF THINGS

Vulnerable Wi-Fi dildo camera endoscope. Yes really

Beau du Jour
03 Apr 2017

Share

Sometimes, our jaws hit the floor. We see some pretty bad things in IoT security, but this has to take the biscuit. After the WeVibe lawsuit and settlement, we started looking at the security of IoT sex toys again.

A few questions:

- Is there any reason a vibrator should also be a Wi-Fi access point?
- What about a vibrator which has an endoscope camera in the end of it?
- Should that vibrator also contain hidden functionality to connect itself to Skype?
- To save videos automatically to a network file share?
- Or send pictures in emails?
- What about if it has code injection in its web interface?

Well, that's the Svakom Siime Eye, a vibrator endoscope. Yes, this thing exists.

It is a pretty normal, slightly awkwardly-shaped vibrator with a camera in the end. It costs $250. But, more relevant than the novelty/enjoyment value of the product and its price, is what it's running, how we hacked it, and why it's an interesting case of another IoT device produced without much care or attention.

First Impressions

The Siime Eye normal use-case is in conjunction with an iPhone or Android app. You turn it on, connect to its AP (SSID "Siime Eye") with the default password ("88888888"), open the app, then 'insert' it. The app itself is limited: You can view the live video stream, take pictures or videos and save them to your device. Everything you'd expect from a camera/vibrator, I guess.

The Android app looks a bit like this:

Categories

Show all

See the other cool stuff we've been doing...

VULNERABILITY DISCLOSURE
What's My Name Again? Reolink camera command injection
13 DEC 2022

CONSUMER ADVICE
Consumer advice for buying smart IoT devices this Christmas
02 DEC 2022

DFIR
Hive Ransomware is on the rise. How should you deal with it?
18 NOV 2022

INTERNET OF THINGS
Effecting positive change in the Internet of Things
21 OCT 2022

SOCIAL ENGINEERING
Social Engineering dos and don'ts
20 OCT 2022

SERVICES
Automotive & IoT Testing

Find out more »

Our People

Being introduced to, and getting to know your tester is an often overlooked part of the process. Yes, our work is über technical, but faceless relationships do nobody any good.
Meet the team »

With names like “wingedcamlib” and “skyviper”, it looks a lot like libraries in here were written with drones in

mind.

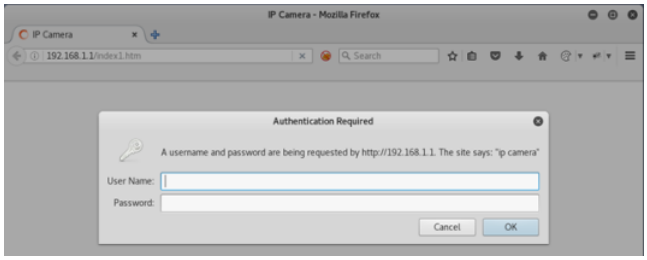
We had a chat with the guys at Sky Viper (who make awesome drone cameras BTW) who were as surprised as us to find their brand name in mobile app code that runs an IoT dildo. They suspect that the early version of the Sky Viper cam control app was partly written by the same guys who wrote the dildo app code.

That’s not everything we can get out of the app though: The “com.SiimeEye” source includes some hard-coded credentials, and a hard-coded IP address and port.

```
public void onClick(View view) {
    switch (view.getId()) {
        case R.id.help:
            new helpDialog(this).show();
            return;
        case R.id.camera_item:
            Log.e("wingedcam", "camera_pwd->" + m_cam.pwd());
            if (!this.camera) {
                this.camera = true;
                m_cam.set_user("admin");
                m_cam.set_pwd("");
                m_cam.model = 1;
                m_cam.update_lan_status(true, "192.168.1.1", 80, false);
                m_cam.add_listener(this);
                if (m_cam != null && m_cam.status() == CONN_STATUS.IDLE && this.camera) {
                    m_cam.start_connect(true, 3);
                    show_progress();
```

There’s an account called “admin”, with a blank password.

On connecting to the Siime Eye AP with a laptop, we can try the server at 192.168.1.1:80. An authentication box pops up, prompting for a username and password.



It looks a lot like Basic authentication, and the admin:[blank] credentials work.

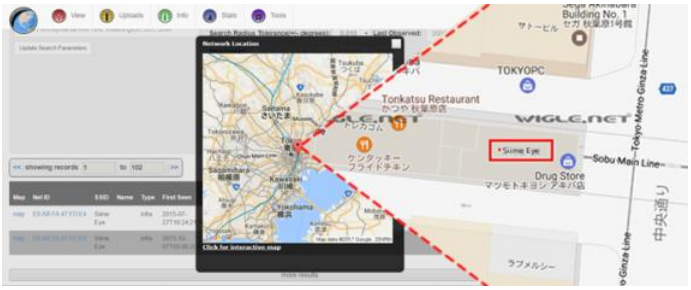
So, it’s trivial to connect to the AP and auth to the web interface.

Remember, the credentials are hard-coded in the official app, so any user wanting to use the Siime Eye the official way will never change these credentials. If you can get onto the wireless AP, you’ll have instant access to everything on this web application. It allows multiple concurrent connections too, without any fuss at all.

THAT WEB APP SERVES THE VIDEO FROM THE CAMERA! OMG!

Oh, and being a Wi-Fi AP means you can find users too…

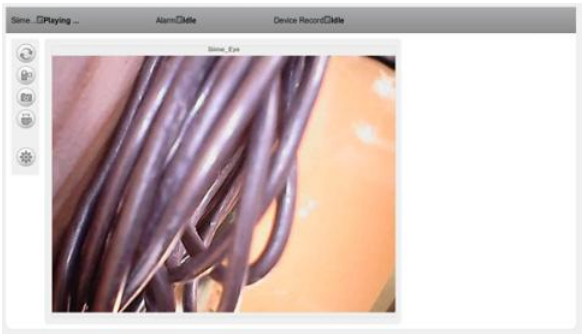
Under normal use, the wireless AP name is also static. That means we can query a wardriving site like wigle.net, and find locations where a "Siime Eye" might be. Here's one seen in Tokyo:

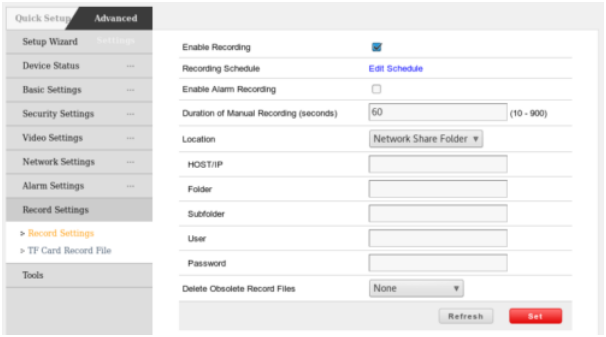


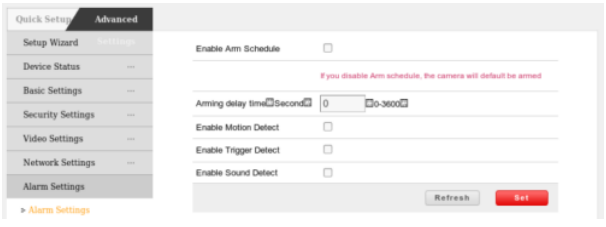
That's bad enough, but what else could we do? Could we get a root shell and persistence?

Software

The web application is designed for administering a much more general-purpose camera, one attached to a drone, for example. We can do a lot more here than the mobile app allows us to do.







There's some NFS settings, motion detection settings, and much more. As we often see in embedded device web interfaces, all settings requests get processed by an array of .cgi files. A lot of requests to change settings are sent to /set_params.cgi. All it does is returns JSON data indicating success or failure. Its sibling file, /get_params.cgi, sends back a lot of configuration data in the response if you send it a GET request. This includes parameters like "skype_pwd", "smtp_pwd", "ddns_pwd". From that, we can assume that there's functionality in the Siime Eye to send emails, change DNS settings, and even add a Skype account.

A typical response from one of the more banal .cgi files looks like this:

```
{"error":0,"id":"E8ABFA567DF2","alias":"Siime_Eye"}
```

It seems like it's got a permissive Cross-Origin Resource Sharing (CORS) policy, as it sends back the "Access-Control-Allow-Origin" header with a wildcard value. In theory, this would let us, or any browser which has access to this web application, read the contents of any response from the server. The facility to read the response from another website is usually restricted by browsers, and is an integral part of the "Same Origin Policy" (SOP). It's a restriction which means that malicious websites can't just arbitrarily read your bank balance.

Ideally, the SOP also means that an arbitrary site can't see and subsequently siphon off your IP camera video stream. But if a server specifies the "Access-Control-Allow-Origin" header with a wildcard value, anyone from any website can read the response. It makes the SOP useless.

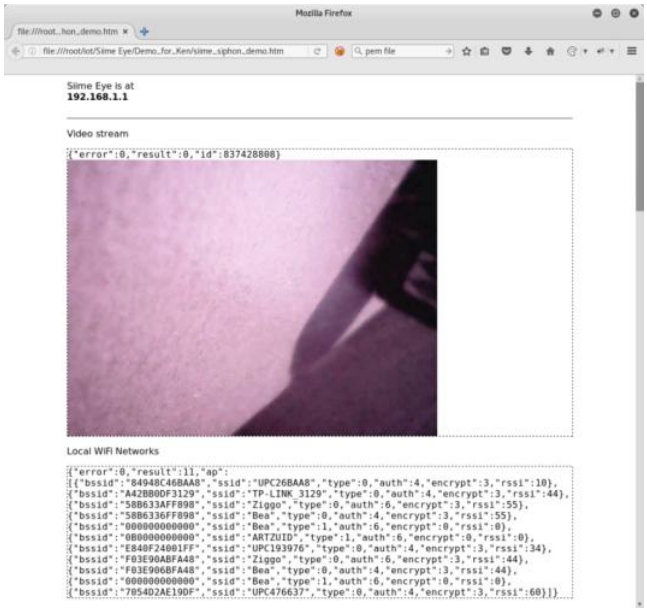
So, I wrote a quick piece of JavaScript using XMLHttpRequest calls to try to siphon data off the device. However, weirdly, the browser kept complaining about us breaking the SOP. That wasn't what I was expecting; it required some investigation.

After some digging, I found out the problem. It turns out one of the idiosyncrasies of CORS with XHR is that a pre-flight request with the OPTIONS header is sent first, expecting the Access-Control-Allow-Origin header in return. But, the Siime Eye interprets OPTIONS as a RTSP request and we get a standard RTSP response.

So you can't siphon data off it that way because the browser thinks you're breaking the SOP; it's not getting the response it expects. It also seems that the RTSP protocol is running out of port 80 as well.

All the .cgi files, however, do allow you to specify a JSONP callback in every request, which means SOP is still useless. JSONP lets you specify a variable name, import the resulting JSON data as a script, and use it like any other JSON data (https://en.wikipedia.org/wiki/JSONP).

So I wrote a small PoC to dump a load of information from the device using JSONP, including a list of local Wi-Fi networks it can see, and the video stream. The video stream is "protected", in that you have to negotiate a token before you can view it. But using JSONP still means we can siphon it off, with very little hassle. You can see my code here.



But, that can't be everything that's wrong with this. Time to start Googling.

Hidden services

There wasn't anything too damning turning up just from some cursory poking at the web interface. So, after some light googling of some .cgi filenames, I came across the Reecam developer documentation (http://wiki.reecam.cn/CGI/Params).

only had port 80 open, with a slightly ropey but relatively robust-looking web interface. I trawled through lots of developer info on their site but, despite a lot of digging, I couldn't actually find the firmware anywhere.

However, one parameter documented on the site, which I didn't come across while initially testing the web interface was "telnetd". Browsing to the following link restarted the Siime Eye:

http://192.168.40.17/set_params.cgi?telnetd=1&save=1&reboot=1

When it came back up, telnet was available. This seems like an easy win. Default Mirai credentials and we're in, right? Unfortunately it wasn't so simple.

After hitting it with a wordlist for way too long, I gave up. Nothing seemed to be working, and there were no clues online about what the password might be.

So, I had no firmware, no shell, and only a minor web interface issue. The next step had to be to dismantle it.

Hardware

The Siime runs off a Ralink RT5350F WiSoC which has a little MIPS processor in it. It's often used in things like Wi-Fi extenders, and it's relatively robust. It's also got Winbond W9825G6JH-6 SDRAM and a Winbond 25Q64FVSIG flash chip, which holds the filesystem.

It also has some handy exposed UART pads, which are easy to clip onto. I connected with a BusPirate, trial and error'd the baud rate to 57600, and got a nice stream of useless debug info. I also got the same impenetrable telnet login prompt, and a pretty restrictive uBoot shell.

No easy-wins here either. I decided it was worth trying to dump the firmware. With some cheap eBay clips, a BusPirate, flashrom, and a Stanley knife (to whittle down the cheap clips so they wouldn't nudge each other off the chip), I managed to get a read off the Winbond 25Q64FVSIG chip.

herrings and leftover scripts from previous incarnations of the Reecam firmware.

What I needed at this point was some sense of what was happening in the system while it was running. Rather than just indiscriminately grep everything "just in case", I wanted a more distilled sense of the system; I still needed a shell.

A Hybrid Approach

We left the laptop clipped onto the UART, and started probing the web interface again. This time, I noticed small debug messages from the web application being directed to the UART stream. It looked a lot like stderr and stdout might just be echoed out for debugging purposes.

I returned to the NFS settings page. "How else can you set up a Network File Share other than by sending unsanitised parameters directly to the UNIX 'mount' command?", I thought.

A few minutes later, I found a command injection point, with all stdout and stderr output getting sent to the UART stream on my other laptop.

Sending "192.168.1.1; ls -al; echo" and "192.168.1.1; cat /etc/passwd; echo" as the "HOST/IP" parameter in the NFS settings resulted in mount errors, a listing of the root filesystem and the contents of /etc/passwd.

Cracking the descrypt hash didn't go very well with a wordlist and light brute-forcing. But, as the web application was running as root, I wrote myself into /etc/passwd as another root user and logged in over telnet.

In the end, cracking the hash wasn't even needed. I checked out the running processes and honed in on some custom system binaries: /bin/reecam and /bin/camera.

It's the slightly – but not entirely – secure: reecam4debug.

From here's it's plain sailing. We've got complete control over every inbuilt function in the Siime Eye, easy access to the video stream, **a root shell and persistence on a dildo**.

Video overview

Conclusion

If there's no reason for a user to access relatively complex functionality of a device, then there's no reason to expose it. It's just too easy for an attacker to leverage official, inbuilt functionality and the web application weaknesses.

In this case, overexposure of system services means we could write a rogue application, compel a user to connect our app to the device using the default credentials, and then use the already-inbuilt functionality to perform unsolicited actions on the device. If we could get a user to connect their device to their home Wi-Fi, we (or any website loaded within the user's home network, in a JavaScript drive-by) could siphon all video data, Wi-Fi passwords, and a list of local networks off it and send it somewhere unsolicited.

Even without all that effort, if we can get anywhere near a Siime Eye and crack into the Wi-Fi AP with a (most likely) weak or default password, we can almost immediately get a root shell and a video stream.

If you're a user, change the Wi-Fi password to something complex and long. And/or, try to get a response from Svakom – I didn't have much luck.

Disclosure timeline

24/12/2016 – Svakom Informed about web interface issues. No response.

09/01/2017 – Follow-up email sent to ask for some kind of response. No response.

09/02/2017 – Svakon informed of further code injection issues & intention to write a blog post. No response.

Since there was no response at all after three attempts at contact, the decision was made to publish.

Get in touch

UK Office:
Pen Test Partners LLP
Unit 2, Verney Junction Business Park
Buckingham
MK18 2LB
United Kingdom
+44 20 3095 0500

Contact Us »

US Office:
Pen Test Partners Inc.
222 Broadway 22nd Floor, Suite 2525
New York
NY 10038
United States
+1 646 693 2501

info@pentestpartners.com

Connect

Twitter
LinkedIn
YouTube