7 minutes

Your Amiibo's Haunted

Exploiting Flipper Zero's NFC file loader

Flipper Zero is a self-described *portable multi-tool for pentesters and geeks in a toy-like body*. The device comes with several built-in applications to transmit and recieve sub-1GHz frequencies, such as RFID, NFC, and Bluetooth.

This post demonstrates a buffer overflow in Flipper Zero's NFC file loader that I discovered for BGGP3.

You Wouldn't Download a Wolf Link

Nintendo Amiibos are collectible figurines with an embedded NFC chip that grant special in-game items on the Nintendo Switch. At anywhere from $15 to $70 USD, and considerably more for limited editions, it's not surprising there's an active community around cloning and emulating these NFCs.

Anyway, I just bought a Flipper Zero to keep amiibos from overflowing my desk and if you're reading this you probably did too. The problem, or opportunity, is Flipper Zero firmware up to 0.65.2 is vulnerable to a buffer overflow that can crash your device and maybe execute code, who knows. The latest version, released 06 Sept 2022, includes the patch for this vulnerability.

Before we dive in, let's look at how a Flipper Zero stores Amiibo files.

```
Filetype: Flipper NFC device
Version: 2
# Nfc device type can be UID, Mifare Ultralight, Mifare
Classic, Bank card
Device type: NTAG215
# UID, ATQA and SAK are common for all formats
UID: 04 C1 8A 01 27 40 03
ATQA: 44 00
SAK: 00
# Mifare Ultralight specific data
Data format version: 1
Signature: 35 47 82 B2 E9 D5 8F 70 C3 BC 79 FE 8A ED 77
16 EC 25 8C 9A 95 32 4B 94 19 00 2E 12 5A D0 AC 90
Mifare version: 00 04 04 02 01 00 11 03
Counter 0: 0
Tearing 0: 00
Counter 1: 0
Tearing 1: 00
Counter 2: 95
Tearing 2: 00
Pages total: 4
Pages read: 4
Page 0: 33 33 33 33
Page 1: 33 33 33 33
Page 2: 33 33 33 33
Page 3: 33 33 33 33
```

The NFC file contains some metadata indicating the device type, version, and so on. We're not really concerned with that and anyway the code to parse them looks fine. What we do care about is the page total field, here `4` and the 4 bytes of page data in each page which I've set to `0x33` for readability.

Buffer Overflow

A buffer overflow exists in the `nfc_device_load_mifare_ul_data` function of `nfc_device.c` that allows out of bounds write on the `dev_data.mf_ul_data.data` buffer. If you need a refresher on

memory corruption bugs on ARM chips then best check Azeria's blog on the subject.

The Flipper firmware hardcodes the length of the NFC data buffer `MF_UL_MAX_DUMP_SIZE` at 2040 bytes, enough to store data from the largest Mifare chips. When this function reads an NFC file it does not check that the length of page data in the file is less than the length of this buffer. Since there's no check to prevent more than 510 pages from being read (`510 * 4 = 2040` bytes), nor a check to ensure the data read is less than the length of `MF_UL_MAX_DUMP_SIZE`, we can store up to `uint16_t` pages of data in our file and read enough of them into memory to corrupt the device.

This may result in various crashes including a BusFault crash and a NULL pointer exception. In some cases, the device is unresponsive to a reboot (idk why, sorry) and must be re-flashed to recover it.

BGGP3

This is my first year participating in Binary Golf Grand Prix 3. The rules are simple; you need to crash a program using a file of no more than 4096 bytes. Unfortunately, this NFC file is way too large to be a valid entry (it's a chonky 23k), but I still had fun finding a bug, writing the exploit, and writing a patch for the Flipper Zero firmware.

Special thanks to all the lovely people at BGGP who took time to organize this event, review submissions, offer support and encouragement, and to all the ghosts.

BGGP3 has bonus challenges that each grant additional points if completed. Copy pasta from their site:

```
Bonus points will be awarded for the following
additional accomplishments:

+1024 pts, if you submit a writeup about your process
and details about the crash
+1024 pts, if the program counter is all 3's when the
program crashes
+2048 pts, if you hijack execution and print or return
"3"
+4096 pts, if you author a patch for your bug which is
merged before the end of the competition
```

Count to 3

Reminder that this bug is a heap based buffer overflow. We're writing out of bounds on the `dev.dev_data.mf_ul_data.data` buffer which for this example we'll say is stored in struct `dev` at address `0x2000f9e0`. In order to make the program call into code we control, we'll need to find a value somewhere on the heap (at a higher address than the start of our buffer) that we can reach without crashing the program.

Very lucky that the `dev` struct contains the address of a callback function! Let's look at that in the code.

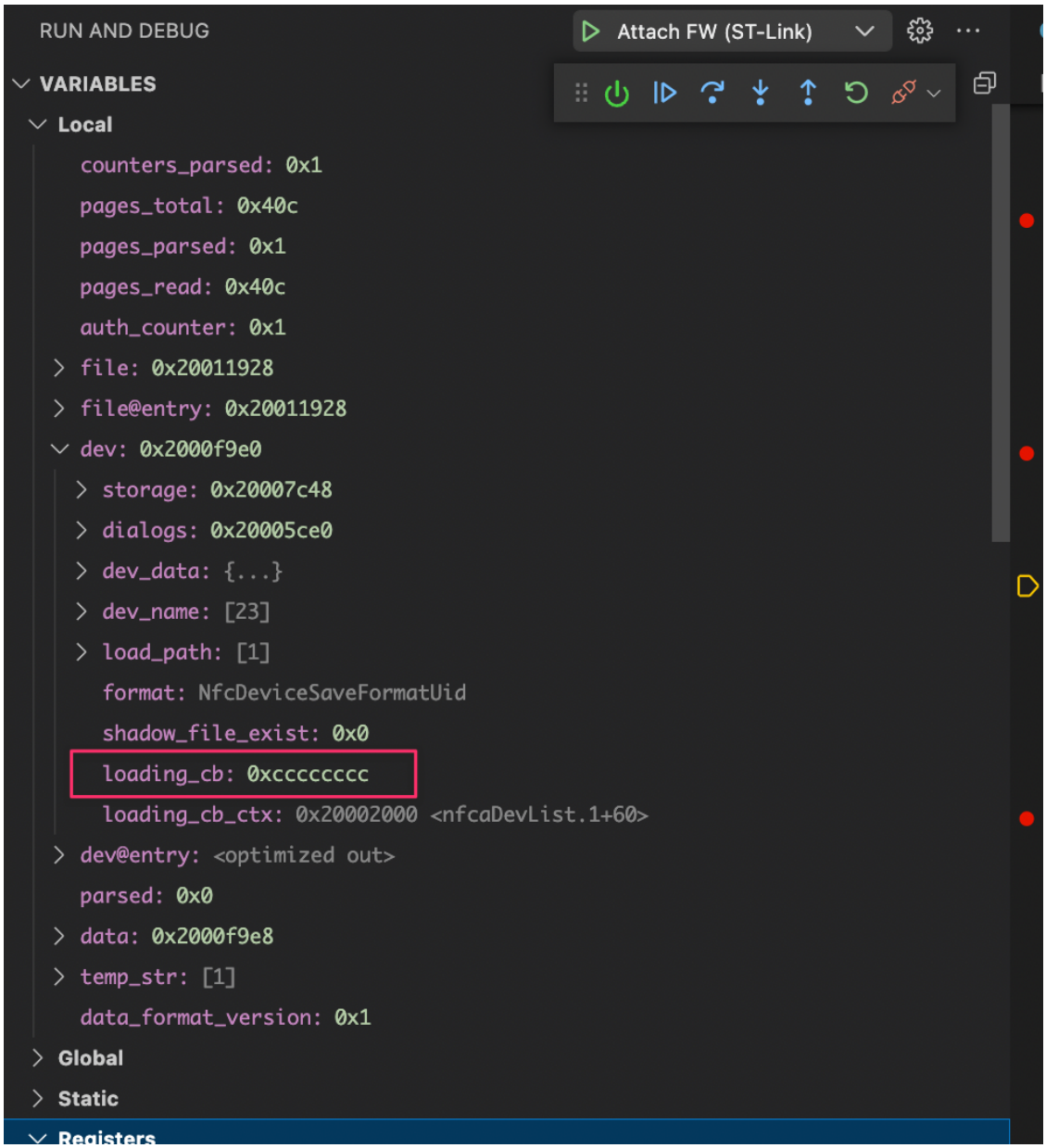
```
1     typedef struct {
2         Storage* storage;
3         DialogsApp* dialogs;
4         NfcDeviceData dev_data;
5         char dev_name[NFC_DEV_NAME_MAX_LEN + 1];
6         string_t load_path;
          NfcDeviceSaveFormat format;
```

```
7           bool shadow_file_exist;
8
9           NfcLoadingCallback loading_cb;  // owo
10          void* loading_cb_ctx;
11      } NfcDevice;
12
```

After loading the NFC file, the program does a bit of cleanup and then calls the `loading_cb` function as it transitions between sub-menus. So if we can control the address without crashing we're guaranteed to load this code right away.

After end of the data buffer, `0x4242`, I've copied the heap bytes up to `0xcccccccc` as they appear when the buffer is 2040 bytes long and not overflowed, so the payload is rebuilding the `dev` struct on the heap (you'll need to fix the alignment too) to reach the address of the `loading_cb` function in the `dev` struct. It looks like this:

```
0x20010a60:    0x41414141 0x41414141 0x41414141 0x41414141
0x20010a70:    0x41414141 0x41414141 0x41414141 0x41414141
0x20010a80:    0x41414141 0x41414141 0x41414141 0x42424141
0x20010a90:    0x67627067 0x00330000 0x00000000 0x00000000
0x20010aa0:    0x00000000 0x00000000 0x00120000 0x001c0000
0x20010ab0:    0x0a402001 0x00020000 0xcccccccc 0x20002000
```



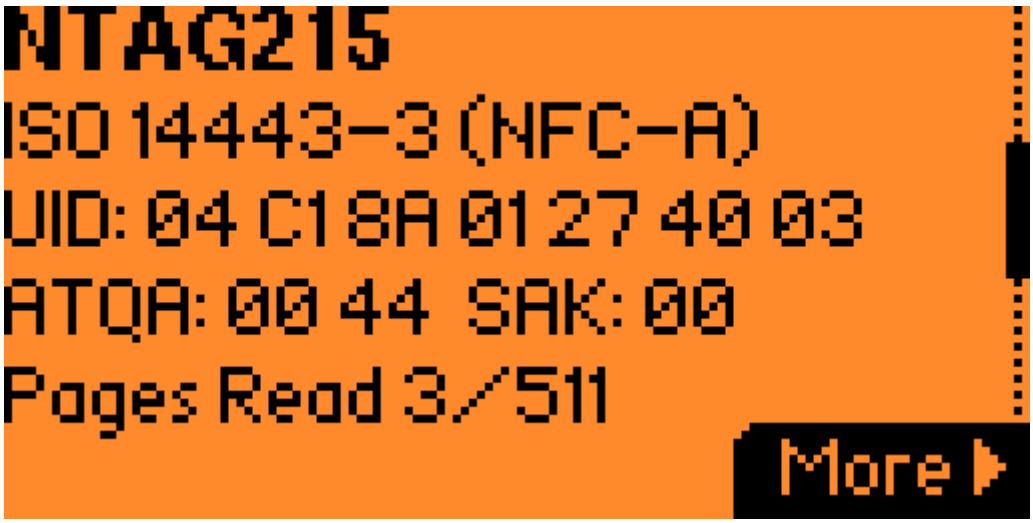
Oopsie!

```
halted: PC: 0x08032e22
halted: PC: 0x08032e26
halted: PC: 0x08032e28
halted: PC: 0xcccccccc
```

Print 3

We can control some data on the heap, and we can control a callback pointer. So how can we get the program to print 3?

This image shows the number of pages read from the NFC file as `3` of `511`. It should say `510` of `511`, so what's going on here?



Look at the NFC data on the heap and you'll notice that immediately following the page data is the number `0x07f8`.

```
...
0x2000e4a8: 0x41414141 0x41414141 0x41414141 0x41414141
0x2000e4b8: 0x41414141 0x41414141 0x41414141 0x41414141
0x2000e4c8: 0x42424141 0x07f84242 0x00000000 0x00000000
```

`0x07f8` is the length of the `MF_UL_MAX_DUMP_SIZE` buffer (2040 bytes). Divide by 4 and you have the number of pages read. To print `3` instead of the actual page count, just replace this value with `0x0c` which results in `3` when divided by 4.

But the challenge is *code execution!* Just overflowing a value on the stack isn't very cash money of me.

So let's talk about ROP. In another post haha. I was hoping to modify one of the menu animations to include `BGGP3` but I've run out of time. >.<

Patching the bug

The final challenge is to submit a patch and have it merged before the end of the competition. My patch was submitted on the last day of BGGP3 and merged into the Flipper Zero firmware repository on Sept 5, after the competition closed.

The patch is a one-liner that checks if the size of either `data_size` or `data_read` is greater than the length of `MF_UL_MAX_DUMP_SIZE` buffer. If it is, the function breaks triggering a file parser failure which is the same behaviour as the other checks on the NFC metadata fields.

Here's the patch. Ezpz.

```
if(data->data_size > MF_UL_MAX_DUMP_SIZE || data-
>data_read > MF_UL_MAX_DUMP_SIZE) break;
```

#Flipper Zero #BGGP3 #Amiibo #NFC #Nintendo

1311 Words

2022-09-05 20:00 -0400

← CVE-2022-3525… Nim - Parsing PE … →