

---

## Hacking The Hacker's Tool

### Patching Flipper Zero's Levels For Fun

מאת נוי פרל

---

#### הקדמה

לאחרונה - גם אני נחשפתי לצעצוע שנקרא "Flipper Zero". הפליפר הוא משדר-מקלט עבור פנטסטרים וחובבים, שמאפשר התממשקות עם דברים כמו פרוטוקולי רדיו, מערכות IoT, סנסורים ועוד. הוא פתוח לשינויים והתאמה אישית, כך שתוכלו להרחיב אותו בהרבה דרכים יצירתיות.

הפליפר מבוסס על Tamagotchi - מתחילים מרמה 1 וככל שעושים יותר פעולות עולים רמות. הפעולות שונות ומגוונות ויכולות להיות קריאת כרטיס NFC, קריאת סיגנל RF משלט ועוד.

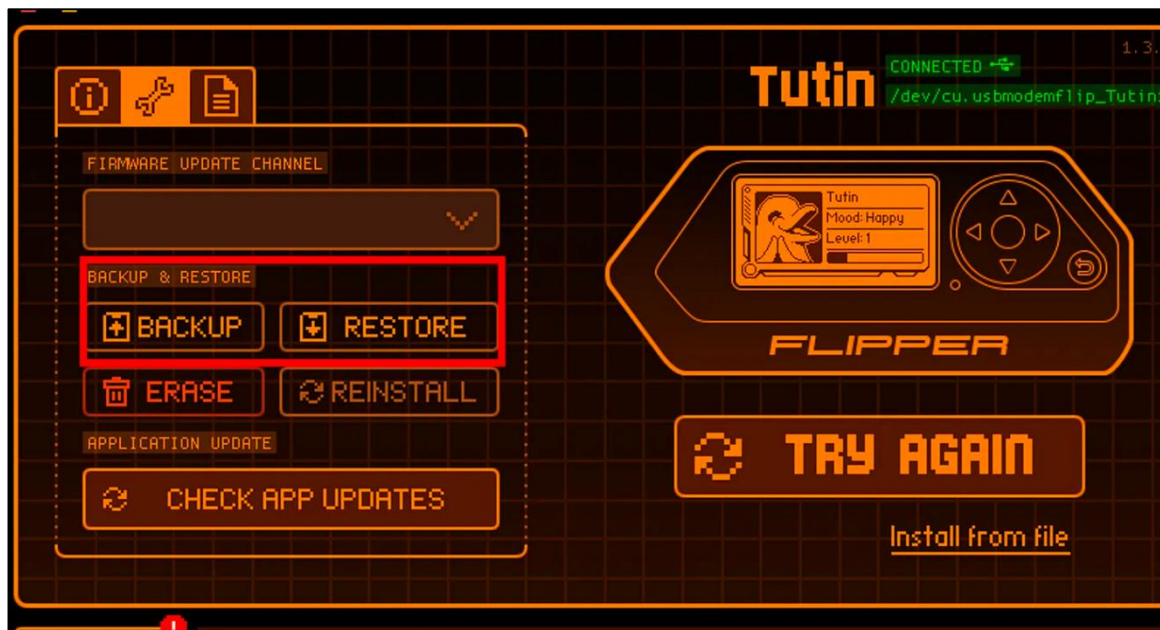
הוא נראה ככה:



הדבר הראשון שהייתי חייבת לנסות זה למצוא דרך לעלות רמות מהר - בצורה לא הוגנת אך מעניינת...

## התממשקות

לפליפר יש אפליקציית Desktop שנקראת [qFlipper](#) ומאפשרת לנו לעשות הרבה עם המכשיר - גיבוי, טעינה מגיבוי, צריבת Firmware, ביצוע Factory Reset, העלאת קבצים ועוד. ניתן למשוך את קבצי הגיבוי ע"י כפתור Backup בממשק ולטעון את הגיבוי ע"י כפתור Restore:



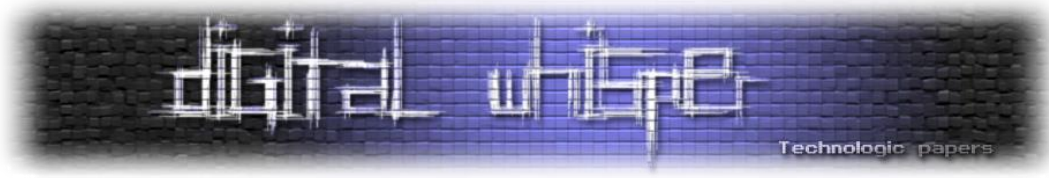
קבצי הגיבוי מועתקים למחשב שמחובר ל-Flipper ב-USB בתוך gzip archive:

```
→ ~/Documents/flipper/backups tar xvf Tutin-backup-20230602-193712.tgz
x int/
x int/.bt.settings
x int/.dolphin.state
x int/.notification.settings
x int/.bt.keys
x int/.desktop.settings
x int/.region_data
```

ספציפית, הקובץ *.dolphin.state* הוא הקובץ שהכי סביר שיכיל את השלב XP של ה-Dolphin (ה-Dolphin). זה האוואטר של Flipper שעולה רמות):

```
→ ~/Documents/flipper/backups/int xxd .dolphin.state
00000000: d001 cf00 0000 0000 000f 1414 0100 022e .....
00000010: 0000 0000 3a00 0000 0000 0000 0000 0000 .....
00000020: 113e 7a64 0000 0000 0a .....>zd.....
```

נתחיל לצלול לתוך הקוד של ה-Firmware כדי להבין מה הולך שם ואיך לעלות רמות בצורה "הגונה".



## צלילה לתוך הקוד

*dolphin\_state\_filename.h* - קובץ זה מכיל את השם של הקובץ השמור שראינו קודם:

```
#define DOLPHIN_STATE_FILE_NAME ".dolphin.state"
```

ובקובץ *dolphin\_state.c* ניתן לראות שהנתיב עצמו מוגדר, לצד פרמטרים כמו: LEVEL2\_THRESHOLD, DOLPHIN\_STATE\_HEADER\_VERSION ו-DOLPHIN\_STATE\_HEADER\_MAGIC:

```
#define DOLPHIN_STATE_PATH INT_PATH(DOLPHIN_STATE_FILE_NAME)
#define DOLPHIN_STATE_HEADER_MAGIC 0xD0
#define DOLPHIN_STATE_HEADER_VERSION 0x01
#define LEVEL2_THRESHOLD 300
#define LEVEL3_THRESHOLD 1800
#define BUTTHURT_MAX 14
#define BUTTHURT_MIN 0
```

אם נעקוב אחרי הרפרנסים ל-DOLPHIN\_STATE\_PATH נוכל לראות פונקציה שאחראית לשמור את ה-struct שראינו מקודם - *dolphin\_state\_save* שקוראת ל-*saved\_struct\_save* ונראה שמעבירה את הגודל של ה-struct, הנתיב (*path*), המידע עצמו (*data*) ופרמטרים נוספים שימושיים:

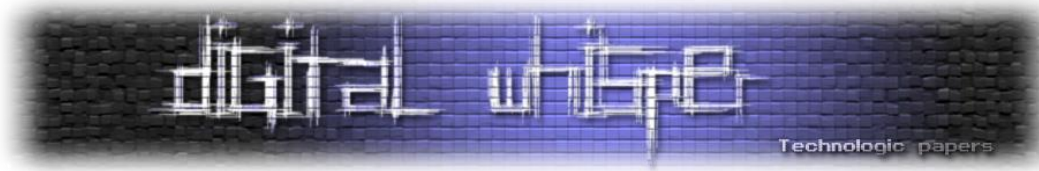
```
bool dolphin_state_save(DolphinState* dolphin_state) {
    if(!dolphin_state->dirty) {
        return true;
    }
    bool result = saved_struct_save(
        DOLPHIN_STATE_PATH,
        &dolphin_state->data,
        sizeof(DolphinStoreData),
        DOLPHIN_STATE_HEADER_MAGIC,
        DOLPHIN_STATE_HEADER_VERSION);
    if(result) {
        FURI_LOG_I(TAG, "State saved");
        dolphin_state->dirty = false;
    } else {
        FURI_LOG_E(TAG, "Failed to save state");
    }
    return result;
}
```

נעבור להגדרה של הפונקציה *saved\_struct\_save* כדי לראות מה הסוגים של ה-arguments שהיא מקבלת:

```
bool saved_struct_save(const char* path, void* data, size_t size, uint8_t magic, uint8_t version){
```

נראה שההגדרה תואמת את הפרמטרים שראינו קודם ואנחנו מתחילים להבין מה המידע שנשמר - למשל *dolphin\_state->data* שחדש לנו. נמשיך ונקרא את ההגדרה של הפונקציה *saved\_struct\_save* עצמה. נוכל לראות אתחול של משתנה בוליאני:

```
bool result = true
```



ובמידה ויש ניסיון לטעון את ה-state ב-saved\_struct\_load - המשתנה result מוגדר false במידה וה-header אינו בגודל הנכון:

```
if(bytes_count != (sizeof(SavedStructHeader) + size)) {
    FURI_LOG_E(TAG, "Size mismatch of file \"%s\"", path);
    result = false;
}
```

ישנן בדיקות נוספות - למשל וידוא ה-magic וה-version של ה-header של ה-state:

```
if(result && (header.magic != magic || header.version != version)) {
    FURI_LOG_E(
        TAG,
        "Magic(%d != %d) or Version(%d != %d) mismatch of file \"%s\"",
        header.magic,
        magic,
        header.version,
        version,
        path);
    result = false;
}
```

וחישוב של ה-checksum. המשתנה size = size\_t בארגומנט של הפונקציה (ראו הגדרה למעלה), והוא לבסוף sizeof(DolphinStoreData).

ה-checksum מחושב בצורה הזו:

```
if(result) {
    uint8_t checksum = 0;
    const uint8_t* source = (const uint8_t*)data_read;
    for(size_t i = 0; i < size; i++) {
        checksum += source[i];
    }
}
```

ה-checksum הינו סכום הבתים שיש ב-data\_read, כאשר הערך ההתחלתי שלו הוא 0, כך שהערך הסופי של checksum הינו סכום כל הבתים של ה-state.

במידה וה-checksum שגוי - הפסדנו:

```
if(header.checksum != checksum) {
    FURI_LOG_E(TAG, "Checksum(%d != %d) mismatch of file \"%s\"", header.checksum, checksum, path);
    result = false;
}
```

ולא נוכל לשחזר מהגיבוי כי result יהיה false:

```
if(result) {
    memcpy(data, data_read, size);
}
```

זה מה שידוע לנו עד עכשיו:

- איך ה-checksum מחושב
- איך מוודאים את ה-state, איך הוא נשמר ונטען



מה שנשאר לנו לדעת:

- איפה ה-checksum נשמר בבינארי dolphin.state.
- איך ואיפה השלב וה-XP שמורים בבינארי dolphin.state - כדי שנוכל לשנות אותם

בואו נתחיל.

בקובץ saved\_struct.c נמצא ה-header של ה-state - שימו לב למיקום של ה-checksum ב-struct הבא:

```
typedef struct {
    uint8_t magic;
    uint8_t version;
    uint8_t checksum;
    uint8_t flags;
    uint32_t timestamp;
} SavedStructHeader;
```

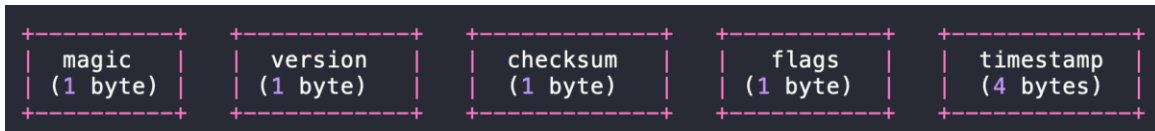
ה-checksum הוא בית אחד והוא נמצא שני בתים (16 ביט) אחרי האיבר הראשון ב-struct, כך:

```
00000000: d001 cf00 0000 0000 000f 1414 0100 022e .....
00000010: 0000 0000 3a00 0000 0000 0000 0000 0000 .....
00000020: 113e 7a64 0000 0000 0a .>zd.....
```

Magic  
Version  
Checksum

לכן, כאשר נשנה את הערכים של ה-state עצמו - נצטרך לזכור לחשב checksum תקין - אחרת השינוי שלנו לא יעבוד והרמה של ה-Dolphin תתאפס.

ניתן להתייחס ל-header של ה-state בצורה הזו:



[8 בתים סה"כ]

נצטרך לראות איך הרמות XP מחושבים ושמורים ב-state. נחפש שוב את הערך שמוכר לנו כבר - DOLPHIN\_STATE\_FILE\_NAME ונשים לב לדבר הבא:

```
#define LEVEL2_THRESHOLD 300
#define LEVEL3_THRESHOLD 1800
```

אוקיי, אז יש 3 רמות ויש רף מספרי (1800, 300) לכל רמה. אולי הרף זה בעצם XP? בואו נראה. בתחתית העמוד נשים לב לדבר הבא:

```
bool dolphin_state_is_levelup(uint32_t icounter) {
    return (icounter == LEVEL2_THRESHOLD) || (icounter == LEVEL3_THRESHOLD);
}

uint8_t dolphin_get_level(uint32_t icounter) {
    if(icounter <= LEVEL2_THRESHOLD) {
        return 1;
    } else if(icounter <= LEVEL3_THRESHOLD) {
        return 2;
    } else {
        return 3;
    }
}
```

נראה שיש משתנה בשם icounter שמחליט על הרמה הנוכחית של ה-Dolphin.



בפונקציה אחרת באותו קובץ ניתן לראות את הדבר הבא:

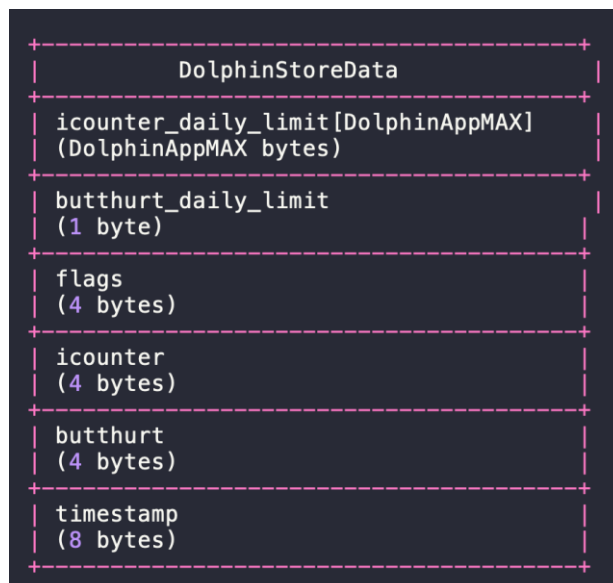
```
uint32_t dolphin_state_xp_to_levelup(uint32_t icounter) {
    uint32_t threshold = 0;
    if(icounter <= LEVEL2_THRESHOLD) {
        threshold = LEVEL2_THRESHOLD;
    } else if(icounter <= LEVEL3_THRESHOLD) {
        threshold = LEVEL3_THRESHOLD;
    } else {
        threshold = (uint32_t)-1;
    }
    return threshold - icounter;
}
```

מעולה. די ברור לנו עכשיו שה-icounter מכיל את ה-XP לרמה של ה-Dolphin וזה משתנה שמאוד מעניין אותנו. כעת, כשברור לנו איך ה-icounter מתנהג, בואו נראה איפה הוא נשמר בתוך ה-struct כדי שנוכל לשנות אותו ואת ה-checksum בהתאם!

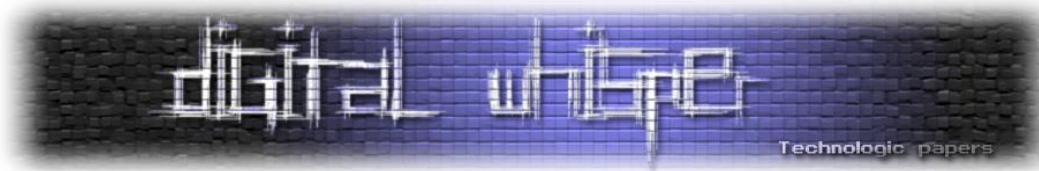
נחפש את ה-icounter בקובץ ה-header של אותו קובץ, הנקרא dolphin\_state.h:

```
typedef struct DolphinState DolphinState;
typedef struct {
    uint8_t icounter_daily_limit[DolphinAppMAX];
    uint8_t butthurt_daily_limit;
    uint32_t flags;
    uint32_t icounter;
    int32_t butthurt;
    uint64_t timestamp;
} DolphinStoreData;
```

אוקיי, זה נראה כמו ה-struct שאנחנו רוצים. יש לו מערך של 8 ביט בגודל DolphinAppMAX. כרגע אנחנו לא יודעים מה זה, אבל מצאנו כאן את ה-icounter ונוכל להתחיל לחשב offsets ולהבין איפה הוא בתוך הבינארי של ה-state. הנה התיאור הבינארי של ה-struct בזיכרון:



בואו ננסה להבין מה הגודל של המערך icounter\_daily\_limit כדי שנוכל למצוא את הבתים של icounter בזיכרון.



נראה ש-DolphinAppMAX הוא enum בקובץ dolphin\_deed.c:

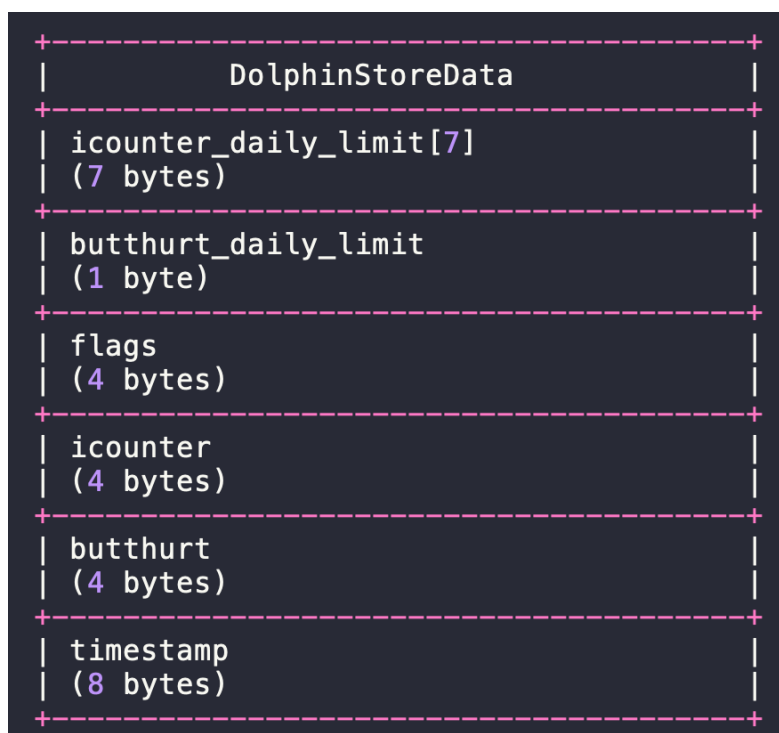
```
typedef enum {
    DolphinAppSubGhz,
    DolphinAppRfid,
    DolphinAppNfc,
    DolphinAppIr,
    DolphinAppIbutton,
    DolphinAppBadusb,
    DolphinAppPlugin,
    DolphinAppMAX,
} DolphinApp;
```

מאחר והערך של DolphinAppMax אינו מוגדר - הוא מקבל את הערך ברירת-המחדל לפי ה-index שלו -  
7. ה-index של הערכים ב-enum מתחיל מ-0 בצורה הזו:

```
typedef enum {
    DolphinAppSubGhz, (val is 0)
    DolphinAppRfid, (val is 1)
    DolphinAppNfc, (val is 2)
    DolphinAppIr, (val is 3)
    DolphinAppIbutton, (val is 4)
    DolphinAppBadusb, (val is 5)
    DolphinAppPlugin, (val is 6)
    DolphinAppMAX, (val is 7) <----- this!
} DolphinApp;
```

אם ככה - icounter\_daily\_limit זה בסה"כ מערך בגודל 7 בתים.

כעת, כשיש לנו את כל המספרים - נוכל לתאר את ה-body של ה-state בזיכרון כך:



[28 בתים סה"כ]



ה-struct שנקרא DolphinStoreData הוא למעשה איבר של struct אחר שנקרא DolphinState בקובץ dolphin\_state.h:

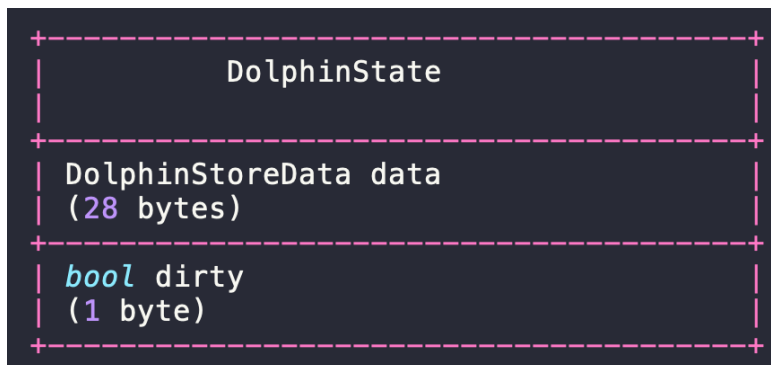
```
struct DolphinState {
    DolphinStoreData data;
    bool dirty;
};
```

שהוא למעשה ה-struct שמאלקצים ומשחררים מהזיכרון בפעולות שמירה וטעינה של backup של הקובץ הבינארי dolphin.state:

```
void dolphin_state_free(DolphinState* dolphin_state);
bool dolphin_state_save(DolphinState* dolphin_state);
bool dolphin_state_load(DolphinState* dolphin_state);
DolphinState* dolphin_state_alloc();
```

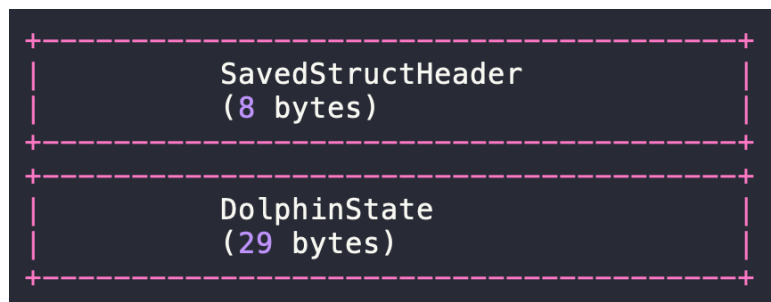
### חישוב הבתים וה-Offsets

אז עכשיו ידוע לנו איך ה-struct של DolphinState נראה בזיכרון:



והגודל של ה-struct הוא למעשה 29 בתים.

אפשר לייצג את ה-state כך - כאשר SavedStructHeader הוא ה-header עם magic ו-checksum ו-DolphinState הוא ה-body עם ה-icounter וה-butthurt:







כעת נוכל לחשב את ה-*offsets* של הבתים בקובץ האהוב עלינו - *dolphin.state*:

```
checksum offset: magic (1) + version (1) = 1+1 = 2
icounter offset: magic(1) + version(1) + checksum(1) + flags(1) + timestamp(4) +
icounter_daily_limit (7) + butthurt_daily_limit (1) + flags(4) = 1+1+1+1+4+7+1+4= 20
```

כעת נחזור לקובץ האהוב עלינו:

```
00000000: d001 cf00 0000 0000 000f 1414 0100 022e .....
00000010: 0000 0000 3a00 0000 0000 0000 0000 0000 .....:
00000020: 113e 7a64 0000 0000 0a .....>zd.....
```

ועכשיו נוכל לסמן את הבתים לפי ה-*offsets* שחישבנו בכל אחד מה-*structs*:

```
00000000: d001 cf00 0000 0000 000f 1414 0100 022e .....!
00000010: 0000 0000 3a00 0000 0000 0000 0000 0000 .....:
00000020: 113e 7a64 0000 0000 0a .....>zd.....

magic version checksum flags timestamp icounter_limit[7] butthurt_limit
flags icounter
```

אלו הערכים שיש לנו כרגע:

- **checksum** : 0xcf (בדצימלי 207)
- **icounter**: 0x3a (58 בדצימלי)

נרצה להגדיל את ה-*icounter* (וכמובן גם את ה-*checksum*), כרגע נהיה צנועים ונחליט שאנחנו נגדיל את הערכים ב-200:

- **Temp checksum**: 0x197 (בדצימלי 407)
- **new icounter**: 0x102 (258 בדצימלי)

שימו לב שהערך של *checksum* הוא *temp* וזו הסיבה: המשתנה *checksum* הוא באורך בית אחד בלבד (8 ביט):

```
uint8_t checksum
```

והערך המקסימלי שהוא יכול להחזיק הוא 0xff - שזה 255 בדצימלי. אבל - יש לנו כאן את הערך החדש של *checksum* שאמור להיות 0x197 שהוא 407 בדצימלי ואמור איכשהו להיות מיוצג ע"י בית אחד של *checksum*, והערך הזה הוא גדול יותר מ-255.

יש כאן בעיה.

לכן אנחנו נשתמש במודולו של 256 הערך החדש ונוסיף את כמות הפעמים ש-256 נכנס בערך הגדול, המקורי.



במילים אחרות, הערך החדש יהיה  $256\%407 +$  שארית (שארית מהחלוקה של 407 ב-256)

נחשב את הערך החדש של checksum:

```
temp checksum = 407
remainder = temp checksum / 256 = 407 / 256 = 1
result = temp checksum % 256 + remainder = 407 % 256 + 1 = 151
```

- **new checksum:** 0x98 (בדצימלי 151)
- **new icounter:** 0x102 (בדצימלי 258)

אין לנו את אותו עניין עם `icounter` כי הוא `uint32_t`, אז הערך המירבי שהוא יכול להכיל הוא `0xffffffff` שזה  $2^{32}$  (4294967295 בדצימלי, די הרבה).

לאחר שחישבנו את הערכים החדשים של `icounter` ו-`checksum` נוכל לערוך את הקובץ `.dolphin.state` ידנית. התעצלתי כאן אז השתמשתי בטריק מגניב לערוך קובץ בינארי:

- פותחים את הקובץ ב-vim
- מקלידים: `!:%:xxd` בשביל להפוך את הדאטה ל-`hexdump`
- עורכים את הבתים שרצינו
- מריצים `!:%:xxd -r` כדי להפוך את ה-`hexdump` בחזרה לבינארי
- שומרים את הקובץ

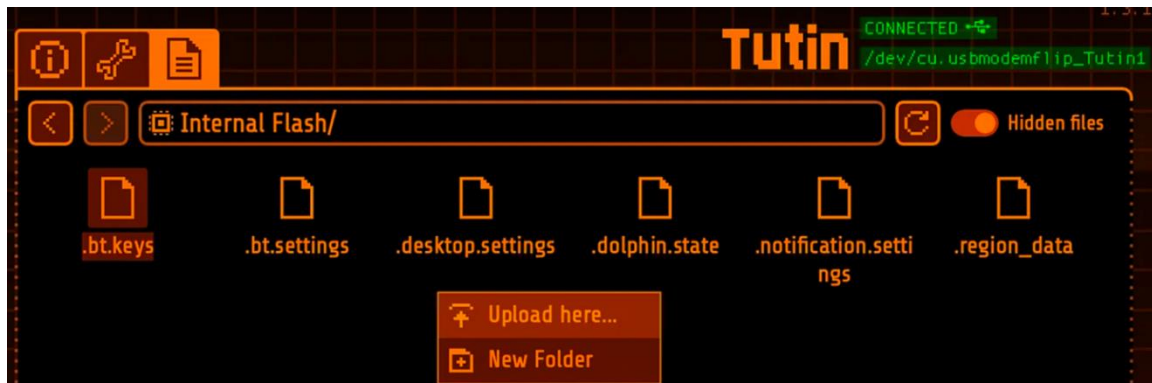
אז כעת הקובץ `.dolphin.state` שלנו נראה כך:

```
00000000: d001 9800 0000 0000 000f 1414 0100 022e .....
00000010: 0000 0000 0201 0000 0000 0000 0000 0000 .....:.....
00000020: 113e 7a64 0000 0000 0a  |>z.....
```

שימו לב לבתים שערכנו: `icounter` ו-`checksum` ולערכים החדשים שלהם.

## שינוי (xp) icounter

כעת נוכל להחליף את הקובץ `.dolphin.state`. לקובץ החדש שלנו ב-Flipper עצמו דרך התוכנה qflipper:



והנה ה-258 XP שלנו התעדכנו!



## שינוי לרמה גבוהה יותר - flex

בואו נשווץ קצת ונשנה את ה-*icounter* ל-858, להרוויח מכל העבודה הקשה שעשינו עד עכשיו. הנה הערכים החדשים:

- *icounter* - 0x35a (בדצימלי 858)
- *checksum* -0xf2 (בדצימלי 242)

ה-hexdump של *dolphin.state*. נראה עכשיו כך:

```
→ ~/Documents/flipper/backups xxd int/.dolphin.state
00000000: d001 f200 0000 0000 000f 1414 0100 022e
00000010: 0000 0000 5a03 0000 0000 0000 0000 0000
00000020: 113e 7a64 0000 0000 0a
```

ועלינו רמה!



הנה סקריפט בפייתון שמשנה את ה-*dolphin.state*. בצורה אוטומטית עם הערך החדש של *icounter* כולל חישוב ועדכון ה-*checksum*:

<https://github.com/noyppearl/FlipFlop/blob/main/levelup.py>

הסקריפט מקבל את הקובץ *dolphin.state*. המקורי וה-*icount* החדש ומשנה בו את הבתים לפי מה שתיארתי במאמר.

וקישור לסרטון של זה בפעולה:

<https://www.youtube.com/watch?v=MAxNkKcwrw>



## סיכום

ל-Flipper יש המון יכולות וניתן לבצע אינסוף מחקרים שונים ומגוונים על כל אחד מהפיצ'רים שלו. במאמר זה הבנו איך עובדת הלוגיקה מאחורי הרמות של ה-Tamagotchi וגם הצלחנו לפצפץ את הגיבוי שטוענים למכשיר ולשנות לנו את ה-XP ללא נגיעה ב-firmware של המכשיר. כמו תמיד - הבנה של איך דברים עובדים מאחורי הקלעים זהו כיוון מרכזי בתחילת מחקר חדש בעולמות החדשים לנו, קריאה של הסורסים תמיד עוזרת וכמובן לנסות, להיכשל ושוב לנסות עד שנצליח.

## מקורות

- <https://github.com/flipperdevices/flipperzero-firmware>
- <https://github.com/flipperdevices/qFlipper>
- <https://github.com/DroomOne/FlipperScripts/blob/main/dolphin-state.py>

## על הכותבת

**נוי פרל** - חוקרת אבטחה, אוהבת לחקור ולגלות דברים שלא נחקרו לפניה ולכתוב/להרצות עליהם בכנסים מדי פעם.