Digital Whisper
Technologic Papers

איך פורצים ומשתלטים על לוויין?

מאת יניב קפיליאנוב

הקדמה

דמיינו לכם קובייה מתכת, בגודל קופסת נעליים, חגה בגובה מאות קילומטרים מעל ראשכם. היא נראית תמימה, אבל בתוכה - מחשב לכל דבר. הוא מקבל פקודות, משדר נתונים, מפעיל חיישנים, מנהל משימות מדעיות, ולעיתים גם מתקשר עם מערכות קרקעיות קריטיות.

מי ירצה לפרוץ אליו? אולי מדינה זרה שתנסה לשבש ניסוי טכנולוגי מתקדם של יריבה, אולי החוקר שיחשוף מחדל בקוד פתוח ששוגר לחלל ואולי מתחרה מסחרי שיבקש לפגוע במשימת צילום או תקשורת. והאמצעים? לא טילים, אלא אנטנה, מחשב וקוד טוב. שליטה בפיקוד של לוויין משמעה לעיתים שליטה במשימה כולה: שינוי מסלול, הפסקת שידורים, גישה לנתונים גולמיים, שליחה של קושחה עוינת - או פשוט הפעלת המפרש שיסיים את המשימה.

איום תיאורטי בלבד? לא בהכרח, במקרים מסוימים, כל מה שנדרש הוא קוד באורך 32 ביט.

במסגרת המחקר חקרתי source code ו-firmware של הלוויין האקדמי PW-Sat2, לווין מחקר שנבנה באוניברסיטת ורשה לטכנולוגיה. מדובר בפרויקט אקדמי שהסתיים תפעולית בשנת 2021, לכן כל המאמר נכתב בפרספקטיבה תאורטית בלבד. PW-Sat2 שוגר והפעולה שלו הסתיימה, אך פרויקט PW-Sat ממשיך להתקיים, וגרסה עתידית, PW-Sat3, נמצאת בפיתוח. ייתכן שחלקים מקוד המקור של הגרסה השנייה (PW-Sat2) משמשים כבסיס גם בגרסה הבאה. לכן, פרסום הממצאים רלוונטי גם לעתיד הקרוב.

במחקר גיליתי כי מנגנון האימות (Authentication) לפקודות uplink מתבסס על קוד אבטחה יחיד בגודל 32 ביט, ללא הצפנה, חתימה דיגיטלית או מנגנון אימות נוסף. חישוב מתקפת Brute Force נאיבית העלה שהמהירות המוגבלת של החיבור (1200 bps) וחלון התקשורת המצומצם, הופכים את ההתקפה לבלתי אפשרית לאורך חיי המשימה של הלוויין. כמו כן, מבדיקה שטחית לא מצאתי חולשה בפרסור של פקודות uplink. עם זאת, ניתוח סטטי של קבצי הקושחה שפורסמו פומבית לפני השיגור הוביל אותי למציאת קוד האבטחה שנצרב על הלוויין, מה שמאפשר לתוקף לשלוח פקודות ללוויין ממש כמו תחנת הקרקע האמיתית. במאמר אדגים כיצד שליטה מלאה בפרוטוקול הפקודות מאפשרת ביצוע פעולות קריטיות, עד כדי העלאת קושחה זדונית למסלול (in-orbit firmware update).

במסגרת המאמר, אעסוק בעיקר בתוכנת ה-OBC (On Board Computer) של הלוויין, במבט על ה-source וה-firmware, במטרה למצוא דרכים לפרוץ ללוויין כתוקף שאין לו גישה לתחנת הקרקע, אלא עם צלחת שידור עצמאית.

גילוי נאות: מטרת המחקר היא העלאת מודעות וקידום סטנדרטים גבוהים של אבטחת מידע בעולם הלוויינות המודרני. לאחר המחקר, נעשו מספר ניסיונות ליצירת קשר דרך ערוצים שונים עם צוות PW-Sat, אך ניסיונות אלו העלו חרס.

מבוא: Curiosity in Orbit

כחוקר סייבר ומפתח Low Level, מעולם לא עסקתי ישירות בתחום הלוויינות, אך סקרנות לגבי "סייבר על לוויינים" הובילה אותי לנסות להבין כיצד נראית ארכיטקטורה של לוויין מודרני מהזווית התוכנתית ובעיקר אילו חולשות אבטחה עלולות להתקיים במערכות כאלה. במיוחד עניין אותי האם ניתן, תיאורטית, להשיג שליטה במערכת לוויינית **מבלי** להשתלט על תחנת הקרקע (מבצע סייבר קלאסי).

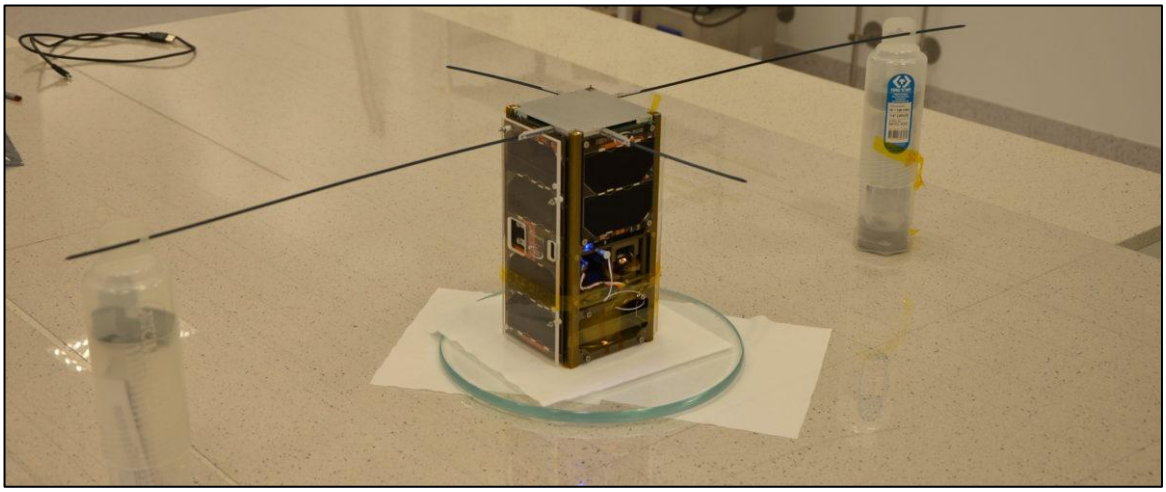
חיפשתי ב-github פרויקטי לוויינים Open source-ים, ומצאתי שישנם מעטים ולרוב הם יהיו לוויני מחקר קטנים הנקראים CubeSat הטסים במסלול LEO (Low Earth Orbit). הלוויינים החדשים, משתמשים בחומרות חדשות יחסית לתחום הלוויינות, כגון מעבדי ARM מודרניים. הדבר מאפשר לייצר מערכת תוכנתית מורכבת יותר ובעקבות זאת סביר שיהיו משטחים רבים יותר לחולשות.

בחיפושים נתקלתי בפרויקט קוד פתוח יוצא דופן - PW-Sat2. הלוויין הוא CubeSat (הסבר מפורט על סוג הלווין יצורף בהמשך) שנבנה והופעל על ידי סטודנטים מהפוליטכניקה של ורשה. מאגר הקוד לא כלל רק קוד מקור מפורט אלא גם גרסאות בינאריות מקומפלות של ה-firmware ששוחררו, הזדמנות נדירה לעיין בקוד לוויין אמיתי ומלא, זאת בניגוד לרוב המשימות המסחריות או הממשלתיות שאינן פומביות.

את המחקר כיוונתי לעבר מנגנוני התקשורת של PW-Sat2 והפרוטוקולים שמאפשרים uplink של פקודות (uplink הוא חיבור בכיוון אחד מהקרקע אל הלוויין, תחנת הקרקע משדרת והלוויין קולט) ובפרט במנגנון האימות של פקודות uplink.

רקע: CubeSats, PW-Sat2 ומודל תקשורת

לווייני CubeSat הם לוויינים זעירים בפורמט מודולרי המבוסס על קוביות בגודל 10×10×10 ס"מ הנקראות "יחידות". לוויינים אלו פותחו במקור למטרות חינוכיות ואקדמיות, אך הפכו לפלטפורמה פופולרית גם לפרויקטי מחקר, ניסויים טכנולוגיים ואפילו יוזמות מסחריות. בשל עלותם הנמוכה יחסית (מאות אלפי דולרים לפרויקט), הם זמינים גם לצוותים אקדמיים קטנים ומאפשרים ביצוע ניסויים בחלל במסלול לווייני נמוך (LEO). פרויקט PW-Sat2 תוקצב ב-180,000 יורו, כך שפגיעה בלוויין ופעילותו תהווה נזק כלכלי משמעותי לאוניברסיטה.



[תמונה מתוך האתר הרשמי של הפרויקט]

לווייני CubeSat הם לעיתים קרובות מערכות Embedded עם רכיב פלאש שעליו נצרב ה-firmware, מעבד מרכזי, זיכרונות, רכיבי תקשורת, מקורות מתח וחיישנים. המגבלות המובנות בפרויקטים מסוג זה, הן מגבלות חומרתיות ותפעוליות כגון זיכרון מוגבל, כוח חישוב מוגבל, רוחב פס וחלון תקשורת קצר. מגבלות אלו גורמות למפתחי המערכות לבחור בפתרונות אבטחה מנוונים ביחס לפתרונות מודרניים כגון אימות קריפטוגרפי לפקודות, secure boot והפרדת תחומי הרשאות. כמו כן, מפתחי לוויינים לרוב יהיו עם פחות מודעות אבטחתית מאשר שחקני סייבר שעלולים לאיים על המשימות, מה שתורם גם כן לפריצות בתחום.

בכל לוויין קיימים שני ערוצי תקשורת עיקריים: Uplink - פקודות מהקרקע ללוויין ו-downlink - טלמטריה ונתונים מהלוויין לקרקע. לרוב, ה-uplink איטי יותר לצורך אמינות וצריכת חשמל נמוכה, בעוד ה-downlink מהיר יותר. ערוץ ה-uplink נחשב לנקודת שליטה קריטית, ולכן הוא יעד מועדף לתקיפות סייבר. PW-Sat2 פעל במסלול LEO בין ה-3 בדצמבר 2018 ל-23 בפברואר 2021, סה"כ 813 ימים במסלול. ברמת התקשורת, הלוויין השתמש ב-uplink של 1200 bps (VHF), ו-downlink בקצבים גבוהים יותר.

איך נראית ארכיטקטורה של לוויין?

ארכיטקטורת מערכות לוויין נחלקת באופן מסורתי לשלושה רכיבים עיקריים: מקטע החלל (Space Segment), מקטע הקרקע (Ground Segment) ומקטע המשתמש (User Segment). מקטע החלל כולל את הלוויין עצמו, ובמקרים מסוימים גם קישורים בין לווייניים (ISL - Inter-Satellite Links) המאפשרים תקשורת ישירה בין לווינים. מקטע הקרקע אחראי לניהול ובקרה של הלוויין באמצעות טלפקודות (TC או TeleCommand) וטלמטריה (TM או Telemetry) המועברות בפרוטוקולי תקשורת ייעודיים (Space Protocols), ומבצע גם קליטה ועיבוד של הנתונים המתקבלים. מקטע המשתמש מתייחס למערכות קצה כגון, מחשבים, טלפונים חכמים או מערכות ניווט - הצורכות את השירותים שהלוויין מספק (כגון תקשורת, מיקום או תצפית).

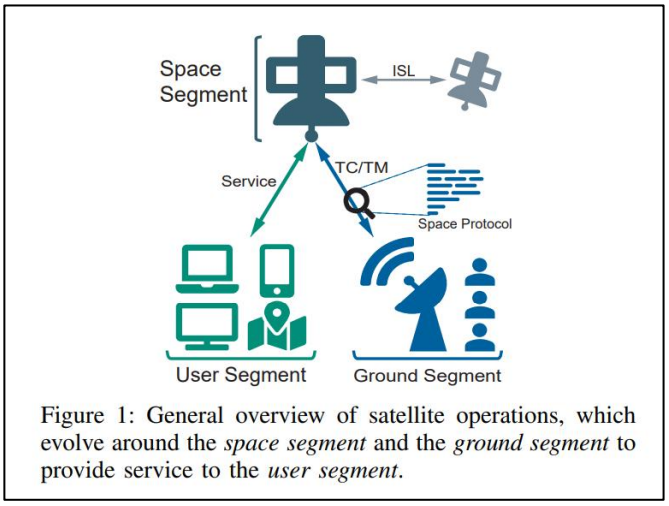


Figure 1: General overview of satellite operations, which evolve around the *space segment* and the *ground segment* to provide service to the *user segment*.

[תרשים מתוך המאמר Space Odyssey: An Experimental Software Security Analysis of Satellites]

לוויין מודרני מורכב משני חלקים עיקריים: ה-Bus (המוצג באפור כהה) וה-Payload (באפור בהיר), כפי שמוצג בתמונה מטה. ה-Bus כולל את תתי המערכות החיוניות לתפעול הלוויין: מערכת בקרת הכיוון והיציבות (ADCS - Attitude Determination and Control System) האחראית לשמירה על זווית וכיוון מדויקים בחלל; מערכת אספקת הכוח (EPS - Electrical Power System) המייצרת ומנהלת אנרגיה חשמלית באמצעות פאנלים סולאריים וסוללות; **מערכת בקרת הנתונים והפיקוד (CDHS - Command and Data Handling System) המהווה את מרכז הבקרה והמחשוב של הלוויין; ומערכת התקשורת (COM - Communications) המאפשרת העברת פקודות ונתונים בין הלוויין למקטע הקרקע**. ה-Payload מהווה את הרכיב הייעודי של הלוויין בהתאם למשימתו. לדוגמה, מצלמות תצפית, חיישנים מדעיים או משדרי תקשורת. השילוב בין ה-Bus ל-payload יוצר מערכת שלמה שמסוגלת לבצע את משימתה תוך שמירה על תפקוד רציף ואמין לאורך זמן בחלל.

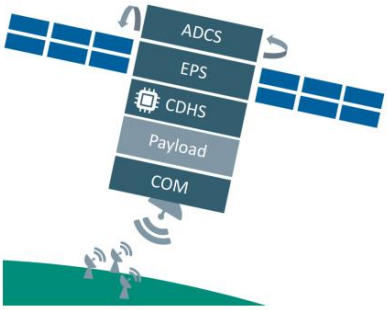


Figure 2: Overview of the different components typically found in a common satellite that features default components for a satellite payload (light grey) and bus (dark grey).

[תרשים מתוך המאמר Space Odyssey: An Experimental Software Security Analysis of Satellites]

בהסתמך על הארכיטקטורה שהוצגה לעיל, מוקד העניין במחקר זה הוא תת המערכת CDHS (Command and Data Handling System) ובפרט הרכיב במערכת שאחראי על עיבוד מסגרות הנתונים (Frames) המתקבלות ממקלט התקשורת (ה-Receiver). המקלט עצמו ממוקם בתת המערכת COM, האחראית על קליטת האותות מהקרקע, המרתם לנתונים דיגיטליים והעברתם ל-CDHS. בשלב זה, ה-CDHS מבצע Parsing לפריימים, בודק את תקינותם, מפענח את הפקודות המוכלות בהם, ומתרגם אותן לפעולות ממשיות המבוצעות על ידי שאר תתי-המערכות בלוויין.

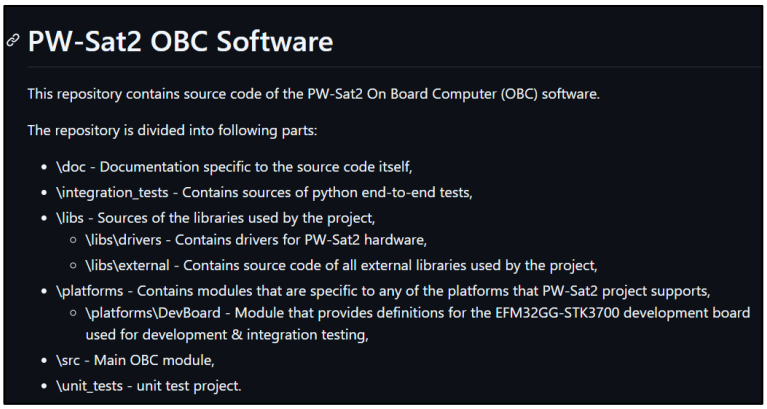
רכיב זה מהווה את "המוח" של הלוויין, ובשל שליטתו הישירה על כלל המערכות, הוא מהווה נקודת מפתח עבור המחקר.

תחילת עבודה

תחילה, הורדתי את ה-Source המלא של PW-Sat2 מ-GitHub, הכתוב ב-C++ והתחלתי לעיין בו.

מתוך כלל הקבצים, בחרתי להתמקד בחלקים שמטפלים בפרוטוקול התקשורת ובמנגנון הטלפקודות (telecommands) במטרה להבין כיצד מתבצע האימות מול הפקודות הנשלחות מהקרקע.

מהי היררכיית ה-repository?



החלקים המעניינים הם:

- ה-src, שם יושב המודול המרכזי של OBC (On Board Computer) - לוגיקת אתחול המערכת.
- Libs\drivers - מכיל את ה"דרייברים" בשביל החומרות השונות, כלומר הקוד שיודע לקבל פקודה מהחומרה הרלוונטית נמצא פה.

ארכיטקטורת קליטת המסגרת (Frame Reception)

מצאתי את המחלקה שאחראית על התקשורת שיושבת בנתיב libs\drivers\comm.

```
class CommObject final : public ITransmitter,       //
                         public IBeaconController, //
                         public ICommTelemetryProvider,
                         public ICommHardwareObserver
```

הלוגיקה לקליטת פריימים נמצאת בפונקציה CommObject::ReceiveFrameInternal, שמבצעת תהליך דו-שלבי: תחילה קריאת אורך (2 בתים) שמייצגים את גודל ה-frame, ולאחר מכן משיכת הפריים המלא. לאחר מכן האובייקט Frame מעובד וכולל מטא-דאטה כמו RSSI ודופלר (מושגים מתחום תקשורת RF), המטא-דאטה נוצר ב-communication hardware על גבי הלוויין ומהווה header לפני הפריים עצמו.

```
bool CommObject::ReceiveFrameInternal(gsl::span<std::uint8_t> buffer, Frame& frame, AggregatedErrorCounter& resultAggregator)
{
    if (buffer.size() < 2)
    {
        return false;
    }

    bool status = this->SendCommandWithResponse(Address::Receiver, num(ReceiverCommand::GetFrame), buffer.subspan(0, 2), resultAggrega
    if (!status)
    {
        return status;
    }

    Reader reader(buffer.subspan(0, 2));
    auto size = reader.ReadWordLE();
    buffer = ReceiveSpan(size, buffer);
    status = this->SendCommandWithResponse(Address::Receiver, num(ReceiverCommand::GetFrame), buffer, resultAggregator);
    if (!status)
    {
        return status;
    }
```

מנקודת מבט של ארכיטקטורת הלוויין, החיבור בין 2 המערכות, ה-CDHS וה-COM טמונה בקטע הקוד הזה, הקורא לפונקציה SendCommandWithResponse עם הפרמטר Address::Receiver שמאחורי הקלעים שולחת את הפקודה ReceiverCommand::GetFrame לכתובת חומרה של ה-Receiver דרך ממשק I^2C, ממתינה, ואז קוראת את תגובת החומרה ל-Buffer.

פענוח ואימות (UplinkProtocol::Decode)

לאחר המטא-דאטה שציינתי, המהווה את החלק הראשון של ה-Frame, יושב הפריים הפנימי, זה blob דאטה כפי שנשלח מהקרקע:

```
DecodeTelecommandResult UplinkProtocol::Decode(span<const uint8_t> frame)
{
    Reader r(frame);

    auto code = r.ReadDoubleWordBE();
    auto command = r.ReadByte();
    auto parameters = r.ReadToEnd();
```

המבנה פשוט, ונראה כך:

- [4 בתים] - קוד אבטחה (32 ביטים)
- [1 בית] - מזהה פקודה (command id)
- [N בתים] - פרמטרים

האימות על הפקודה שהתקבלה מתבצע בהמשך הפונקציה UplinkProtocol::Decode, בהשוואה פשוטה מול שדה באובייקט Hardcoded שנקרא securityCode_:

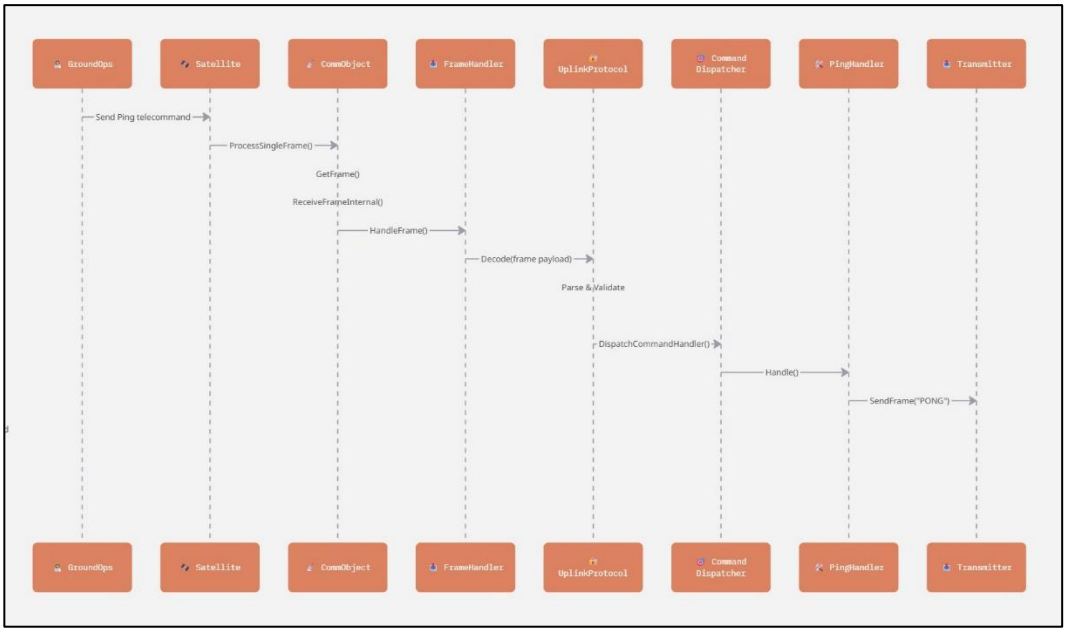
```
if (code != this->_securityCode)
{
    return DecodeTelecommandResult::Failure(DecodeTelecommandFailureReason::InvalidSecurityCode);
}
```

זה מעניין! האימות היחיד שיש על פקודה שנשלחת מהקרקע הוא השוואה פשוטה מול קוד בן 4 בתים.

Flow מלא מקבלת ה-frame בלוויין ועד הרצת הפקודה

1. משיכת ה-frame מה-communication h/w.
2. אימות הקוד שיש ב-frame מול ה-security code הקבוע מראש.
3. הרצת פונקציית ה-Handle המתאימה לפקודה לפי ה-command id ב-frame.

דיאגרמת flow מלאה של פרסור Frame וביצוע פקודה:



מודל האיומים: מי התוקף ומה הן יכולותיו?

לפני שנצלול למתקפות אפשריות, חשוב להגדיר מודל איומים שמתאים ללוויין:

- **יריב "חובבן" עם ציוד רדיו**:
  - תחנת קרקע חובבנית, אנטנות מתאימות, מודולציות ידועות, גישה לזמני מעבר (TLEs).
  - היריב אינו יודע מה קורה ברמת פרוטוקול uplink, הוא יכול להאזין ל-downlink ולקלוט שידורים לא מוצפנים.
- **חוקר אבטחה/אקדמיה**:
  - גישה מלאה לקוד המקור/בינארים שפורסמו, ידע ברוורסינג (IDA/Ghidra/radare2).
  - היריב עם הידע המתאים יכול לשדר frame-ים מתאימים ב-uplink ולהשמיש חולשות.
- **מדינה/ארגון עם משאבי RF**:
  - יכול לייצר רצף תקשורת ארוך יותר מהתחנה הרשמית בעזרת הצבת תחנות רבות גלובלית.
  - היריב בעל כלל יכולות הRF, ידע פנים על פרוטוקולים והשמשת חולשות ברמה גבוהה.

אבטחה באמצעות קוד "סודי" באורך 32 ביטים

מי קובע את הקוד?

חיפשתי בקוד איפה מוגדר הקבוע של ה-securityCode ומצאתי את הדבר הבא:

```
namespace settings
{
    constexpr std::uint32_t CommSecurityCode = ${COMM_SECURITY_CODE};
}
```

בקובץ ה-cmake של הפרויקט יש ברירת מחדל לקוד שנכנס כפרמטר (Variable Substitution), אך ניתן להעביר כפרמטר קוד אחר, ככל הנראה מה שקורה כאשר מורידים גרסה:

```
platforms > mcu > FlightModel > settings.cmake
1   set(COMM_SECURITY_CODE "0xBBCCDDEE" CACHE STRING "32-bit COMM security code written in hex with leading 0x")
```

גילינו בעיה קריטית, ההשוואה הפשוטה הזו של הקוד שנשלח ב-frame אל מול הקוד שנקבע מראש פעם אחת לפני שיגור, היא שומר הסף היחיד שמונע מתוקף לכוון צלחת לוויין על הקרקע לכיוון הלוויין ולשלוח אליו פקודות. המסקנה המתבקשת היא לנסות לעשות Brute Force על הקוד עד שנגיע לאחד הנכון ונקבל חזרה טלמטריה שמאשרת את הפקודה.

קיימת בעיה אחת - אין לי צלחת כזאת. מה שאני יכול לעשות, זה לחשב ולהבין האם המתקפה ברת יישום. החישוב נדרש מפני שבמקרה הנ"ל, בשל הקצבים הנמוכים, לא מובן מאליו שהמתקפה אפשרית, זאת בשונה משני מחשבים מחוברים בכבל, או ברשת האינטרנט על הקרקע.

Brute Force: חישוב ישימות תחת מגבלות קצב שידור

פרמטרים:

- קצב Uplink: 1200 bps
- גודל פריים (פינג מינימלי): 5 בתים (32 bit code + 1 byte cmd)
- סה"כ קשר יומי (לוויין מעל תחנת קרקע אחת): 7 דקות = 420 שניות

קצב ניסיונות משוער:

$$\text{frames/sec} = \frac{1200 \text{ bps}}{5 \text{ bytes} \times 8 \text{ bits}} = 30 \text{ שנייה/ניסיונות}$$

- **ליום**: 30 × 420 = **12,600** נסיונות

זמן לפגיעה בערך יעד רנדומלי (לדוגמה 1.18×10^9 ≈ 0x46707057)

$$\frac{1,180,000,000}{12,600} \approx 93,650 \text{ ימים} \approx 256 \text{ שנים}$$

פרספקטיבה של חיי המשימה

משך המשימה הכולל: **813 ימים** מספר ניסיונות כולל פוטנציאלי: **10,243,800** = 813 × 12,600
וזה כ-**0.23%** בלבד מהמרחב (2^32)

Brute-Force "נאיבי" בקצב השידור הנתון (1200 bps) לא היה מגיע לערך היעד אפילו אם היינו מנסים לאורך כל חיי הלוויין.

אבל - אם אני תוקף מעצמתי

כל החישוב עד כה היה בהנחה שיש תחנת קרקע אחת ממנה תוקפים, לצורך העניין, "התחנה" יכולה להיות צלחת אחת בגג הבית בו אתם גרים. בואו נניח תרחיש אחר, התוקף הוא גוף עם משאבים כמו של מעצמה ויש לו 30 תחנות פרוסות במקומות שונים על כדור הארץ, במקום חלון שידור של 7 דקות, היה לו חלון שידור של כשעה וחצי (90 דקות),

במקום **12,600** ניסיונות ביום, היינו מקבלים **162,000** ניסיונות ביום, שמתרגמות ל:

$$\frac{1,180,000,000}{162,000} \approx 7,283 \text{ ימים} \approx 20 \text{ שנה}$$

התוצאות עדיין אינן מרשימות.

ברור לכל הדעות שלפי החישוב המתקפה איננה ברת יישום, ברצוני לבדוק האם ישנו תרחיש תאורטי בו המתקפה רלוונטית. נניח ומדובר בלוויין מעט יותר חזק מבחינת קצבי שידור, כזה שבאמת מעניין מעצמות, קצב השידור בו היה לצורך העניין 20 kbps. היינו מקבלים פי 17 יותר ניסיונות בשנייה ובתרחיש כזה נקבל תוצאה של 426 ימים ≈ 1 שנה ו- 2 חודשים.

בתרחיש זה אנו היינו מספיקים להגיע לקוד הנכון בקבועי הזמנים של המשימה, לקבל Pong חזרה מהלוויין ולהתחיל להשתלט עליו. זהו תרחיש תאורטי לחלוטין, כמו כן, שמתאים רק למעצמות ולוויינים שעובדים בקצבים גבוהים. לצערי, אין לי יכולות מעצמתיות ולכן אמשיך לפתרונות אחרים.

ניסיון למצוא חולשה ב-frame parsing

לאור ממצאי האימות החלש והעובדה שמתקפת Brute Force לא ברת יישום, ביקשתי לבדוק האם קיימת אפשרות עוקפת - למשל **חולשת buffer overflow** שתאפשר כתיבה מעבר לגבולות הזיכרון **לפני** שמתרחשת בדיקת הקוד, ובכך לעקוף את מנגנון ההשוואה.

החשד נבדק בנקודה בה מתקבלת מסגרת uplink מן החומרה, דרך הפונקציה ProcessSingleFrame. תחילה בחנתי כיצד מוגדר ה-buffer שאליו נטענת המסגרת. ההגדרה הרלוונטית:

```
std::uint8_t buffer[PrefferedBufferSize];
```

כאשר PrefferedBufferSize מוגדר כ-**255 בתים**. מטרתי הייתה לבדוק מה מתרחש במקרה שבו נשלחת מסגרת ארוכה יותר - כלומר, חריגה פוטנציאלית מגבולות המערך.

הנתיב שעובר ה-frame מתחיל בקריאה לפונקציה ReceiveFrameInternal, אשר בתוכה מתבצעת קריאה לגודל המסגרת באמצעות ReadWordLE, ולאחר מכן, טעינת המסגרת אל תוך ה-buffer. הקטע הרלוונטי בפונקציה ReceiveFrameInternal:

```
Reader reader(buffer.subspan(0, 2));
auto size = reader.ReadWordLE();
buffer = ReceiveSpan(size, buffer);
status = this->SendCommandWithResponse(Address::Receiver, num(ReceiverCommand::GetFrame), buffer, resultAggregator);
```

במילים פשוטות, הקריאה לפונקציה ReceiveSpan מוודאת כי אין ניסיון לקרוא מעבר לגודל ה-span שהוגדר:

```
static gsl::span<std::uint8_t> ReceiveSpan(std::uint16_t frameSize, gsl::span<std::uint8_t> buffer)
{
    if (buffer.size() <= frameSize + 6)
    {
        return buffer;
    }
    else
    {
        return buffer.subspan(0, frameSize + 6);
    }
}
```

כאן ניתן לראות שהתנאי שומר על כך שלא ייקרא מעבר לגודל המוגדר של buffer. למעשה, אם נשלח גודל מסגרת חריג, הפונקציה תחזיר פשוט את ה-buffer המקורי (עד 255 בתים), ולא תנסה לגשת מעבר לגבול. כלומר, אין Overflow ישיר ברמת קריאת המסגרת.

המסקנה - מנגנון קריאת המסגרת כולל הגנה בסיסית אך אפקטיבית מפני overflow - ואין ניסיון לקרוא מעבר לגבולות ה-span שהוגדר מראש .עם זאת, ראוי לציין כי שכבות נמוכות יותר המטפלות בחומרה, למשל תעבורת I2C, עשויות לכלול משטחי תקיפה נוספים. במחלקות כמו I2CLowLevelBus קיימת גישה ישירה יותר לרכיבי חומרה, שייתכן וכוללת נתיבים פחות מוגנים. בשלב זה בחרתי שלא להעמיק בניתוח שכבות אלו, אך זהו כיוון מעניין למחקר אחר.

הרגע שבו Brute Force וחולשות מפסיקים להיות רלוונטיים

כיווני המחקר עד כה לא הניבו פרי לכן חזרתי למקורות, לפרויקט ה-github באתר, וכך נתקלתי בעמוד ה-Releases. העמוד מכיל את הגרסאות של ה-firmware אשר נצרבו על הלוויין ורצו בחלל. לגבי אחת הגרסאות נכתב בפירוש:

The binary was uploaded as an in-orbit update to the PW-Sat2 On-board Computer on - March 17th, 2019, allowing for switching between Deep Sleep and full flight software.

הנ"ל לא משאיר מקום לספק, אם אני אמצא ב-Released Firmware את קוד האבטחה שעוצר אותי מלהריץ פקודות, זה יהיה האחד שנמצא בחלל ונוכל להשתמש בו.

הורדתי את ה-zip ומצאתי בתיקיית build\FlightModel את ה-firmware בשם pwsat. החלטתי לזרוק את הקובץ לתוך IDA ולהתחיל לרוורס. הדבר הראשון שרציתי למצוא זה את ה-securityCode_ שצרוב על הלוויין. למזלנו הfirmware קומפל עם סימבולים (DWARF debug information) מה שמאוד מקל על העבודה. כיוונתי את עצמי לאזורים שבהם מתבצע אתחול של אובייקטים וציפיתי למצוא ערך סטטי של securityCode_ שטוענים לתוך רגיסטר. לאחר חיפוש קצר מצאתי:

```
MOVW    R6, #(Main.Communication.UplinkProtocolDecoder._securityCode - 0x880182D8)
MOVW    R5, #(Main.Communication.SupportedTelecommands - 0x880182D8)
MOVW    R4, #(Main.Communication.SupportedTelecommands._telecommands.gap0+4 - 0x880182D8)
MOVW    R0, #(Main.Communication.SupportedTelecommands._telecommands.gap0+8 - 0x880182D8)
LDR     R1, =Main.adcs.coordinator
MOVW    R3, #(Main.Communication.SupportedTelecommands._telecommands.gap0+0x60 - 0x880182D8)
STR.W   R1, [R10,LR]
LDR.W   LR, =off_B3054
MOVW    R8, #(Main.Communication.SupportedTelecommands._telecommands.gap0+0xC - 0x880182D8)
STR.W   LR, [R10,R7]
LDR     R7, =0x46707057
MOVW    R2, #(Main.Communication.SupportedTelecommands._telecommands.gap0+0x44 - 0x880182D8)
STR.W   R7, [R10,R6]
```

אפשר לראות כאן בברור ש- securityCode_ מאותחל עם הערך 0x46707057 – מצאנו Secret Leak.

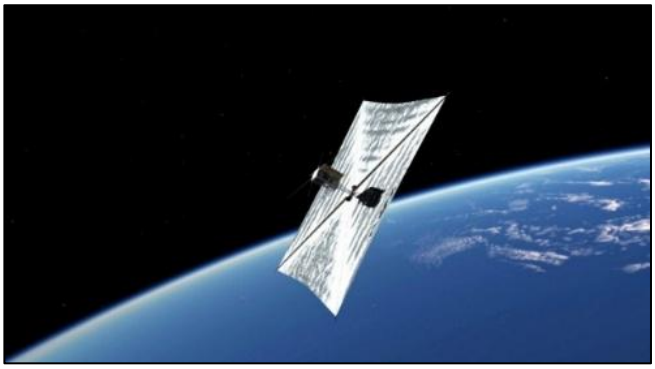
עכשיו תאורטית, אם אכוון צלחת מהקרקע לכיוון הלוויין ואבנה telecommand מתאים יחד עם קוד האבטחה הזה, הלוויין יפרסר את ה-frame ויריץ את הפקודה ללא שום מגבלות!

מה אפשר לעשות עם שליטה מלאה?

רשימה חלקית של Telecommands קריטיים שקיימים לפי קוד המקור:

- OpenSail - פריסת המפרש הפיזי (אירוע בלתי הפיך שמתחיל את סיום המשימה)
- EraseBootTableEntry / WriteProgramPart / FinalizeProgramEntry - כתיבה/מחיקה של קושחה
- SetTime / SetBitrate / SetBootSlots - שינוי פרמטרים מערכתיים
- PowerCycle / ResetTransmitter - שיבוש תקשורת
- RawI2CTelecommand- גישה ישירה להתקני I2C
- ReadMemoryTelecommand - קריאת זיכרון (כולל מפתחות/סודות נוספים)

ככה נראית פריסת מפרש, שמתחיל תהליך יציאה מ-Orbit - כן כן, סיום המשימה ואין דרך חזרה:



[תמונה מתוך האתר הרשמי של הפרויקט]

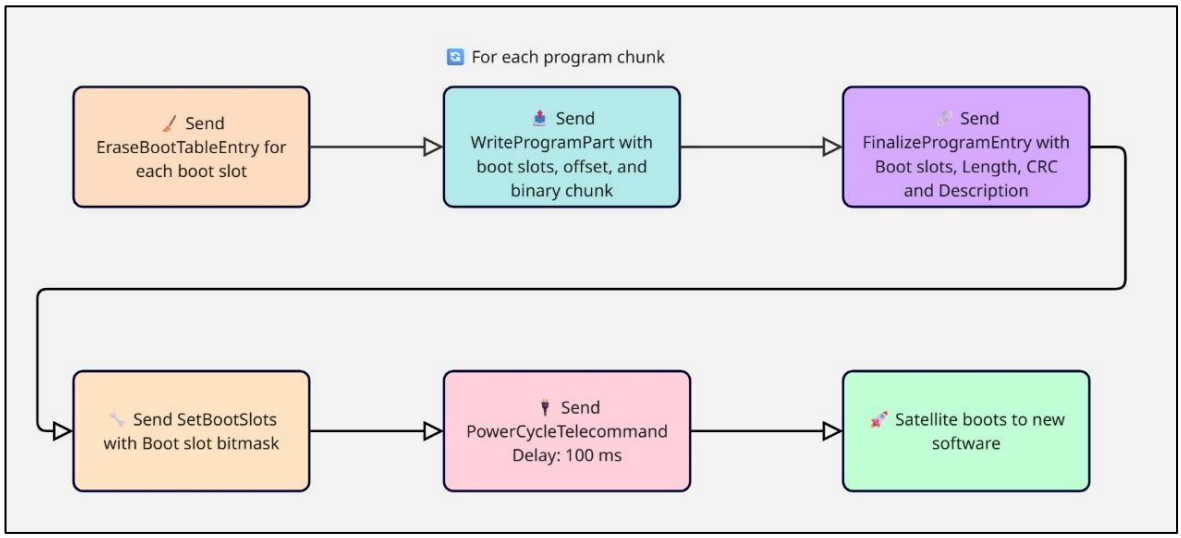
עדכון firmware בחלל: כשפיצ'ר לגיטימי הוא וקטור תקיפה

עכשיו כשיש לנו את הקוד הסודי, ניתן להריץ קוד משלנו, תאורטית כמובן.

ב **17-במרץ 2019** הועלה firmware חדש לחיסכון באנרגיה (OBC_DS_V1.0). בעדכון נוסף Telecommand שנקרא Deep Sleep שאפשרה חיסכון באנרגיה.

העדכון התבצע באמצעות רצף ה-TC הבאים:

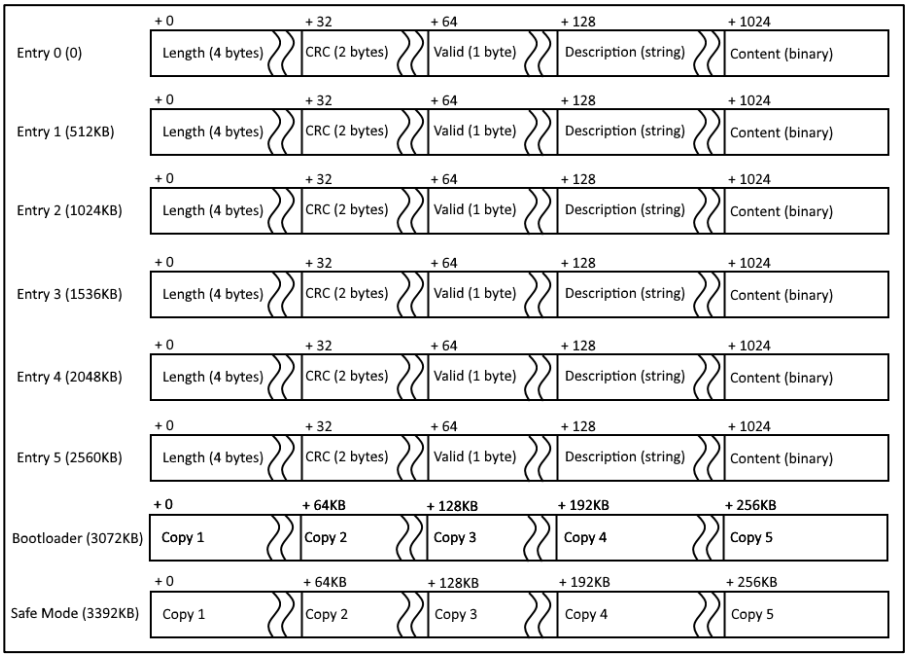
- WriteProgramPart - כתיבה של חלקי firmware
- FinalizeProgramEntry - איחוד לוגי של החלקים
- SetBootSlotsTelecommand - עדכון של ה-BootSlots בהתאם על מנת שלאחר restart יעלה ה-firmware החדש



היכולת להחליף בינארי מרחוק היא קריטית לתפעול כמו במקרה הזה, אך מייצרת פגיעות בהעדר מנגנוני אימות. תוקף יכול:

- להעלות קושחה הרסנית (Brick)
- לשתול backdoor שידליף מידע
- להנדס false telemetry ובכך לשבש מידע שחוזר למרכז השו"ב (שליטה ובקרה)

תיעוד מתוך הפרויקט לגבי מבנה ה-flash ואיך ה-firmware-ים השונים מאוחסנים, ניתן לראות כאן את ה-slot-ים שציינתי קודם:



[מתוך ה-github של הפרויקט]

לגבי בנייה של תוכנה מתאימה שנוכל לצרוב - אין שום בעיה, יש לנו את כל הקוד ואת ה-toolchain, יכולנו לעשות כמה שינויים קלים בפרויקט, לקמפל ולשלוח!

גם במקרה הזה, אין מנגנון אימות על ה-firmware, בטח שלא secure boot chain. הווידוא היחיד הוא CRC על ה-payload שאנחנו מחשבים מראש ושולחים, והוא בתורו מאומת מול חישוב CRC נוסף על גבי הלוויין עצמו. המנגנון לא נועד להגן מפני הרצת firmware לא חתום, אלא לוודא שלא היה bit flip בדרך.

כל שנשאר הוא לכוון צלחת לשמיים, לקמפל קוד פרי ידינו ולשלוח באמצעות רצף הפקודות הנכון - וכך נוכל להשתלט על הלוויין!

סיכום

סקרנות לגבי לוויינים מודרניים, הובילה אותי לחקור את קוד המקור של PW-Sat2 ולרוורס את הfirmware שפורסם. המחקר הוביל לתובנה שהלוויין אינו מאובטח באמצעים מודרניים, ומנגנון האימות היחיד שקיים על מנת לאמת פקודות שנשלחות מהקרקע הוא קוד בן 32 ביטים שנקבע מראש לפני השיגור. הבנתי שמתקפת Brute Force אינה פיזבילית בגלל קצבי השידור הנתונים, וגם הפרסור של ה-frame מוגן מ-Buffer overflow, אך בהמשך באמצעות קצת מחקר OSINT ורוורסינג גיליתי Secret Leak, צוות העבודה פרסם את ה-firmware שנשלח לחלל, **כולל הקוד הסודי**. הדבר הוביל, תאורטית, לפגיעה בלוויין או הרצת קוד - ניצחון!

כפי שציינתי בתחילת המאמר, פרויקט PW-Sat ממשיך להתקיים והגרסה הבאה כבר בפיתוח. סביר להניח, כי חלקים מקוד המקור של הגרסה שחקרתי משמשים את הגרסה העתידית. כתוצאה מכך, הלווין הבא עלול להיות פגיע באותם מקומות.

עולם ה-CubeSats או ננו-לוויינים מציע כיום "שטח ניסוי" כמעט חופשי - עם קוד פתוח, תיעוד עשיר, ומשימות שאינן תמיד מוגנות מספיק בפני איומי סייבר. תוקף בעל משאבים ויכולת טכנית יכול לנצל את השקיפות הזאת כדי ללמוד, לנצל או לפגוע במשימות חלל. במקרה הזה, השמיים הם לא הגבול ואפשר לדמיין איזה שחקנים שונים ומגוונים עלולים לנסות ולפגוע במשימות חלל.

התחום נמצא כיום בנקודת מפנה: הלוויינים נהיים חכמים, מחוברים, ועשירים ביכולות. בהתאמה, כל איומי הסייבר שאנחנו מכירים בעולם על פני כדור הארץ הופכים להיות רלוונטיים לחלל. זה הזמן שגם אבטחת הסייבר בלוויינים תעבור תפנית. מערכות קריטיות, נתונים רגישים, משימות מדעיות ועוד מונחים על הכף.

המלצות למשימות עתידיות

1. **אימות קריפטוגרפי לכל טלפקודה** - שימוש ב-MAC מבוסס מפתח סימטרי (כגון HMAC-SHA256) או חתימה דיגיטלית א-סימטרית כגון (Ed25519).
2. **רנדומיזציה ורוטציה של מפתחות פר משימה** - כל משימה תקבל מפתח ייחודי שיתחלף לאורך המשימה. אין להשתמש בערכי ברירת מחדל או hard-coded.
3. **הפרדת תחומי הרשאות (Privilege Levels)** - פקודות מסוכנות ידרשו אימות נוסף, חתימה מרובת משתתפים (multi-party auth), או מסלול פיקוד ייעודי.
4. **Rate Limiting וניהול ניסיונות כושלים** - האטה מדורגת של קצב העיבוד, מנגנוני backoff ונעילה (lockout) למניעת מתקפות brute-force או מילון.

5. **אימות קושחה בחתימה (Secure Boot in Space)** - שרשרת אמון מלאה (Root of Trust) מה-bootloader ועד ה-application.

6. **טלמטריה לזיהוי תקיפות** - ניטור ודיווח על התנהגויות חשודות כגון ניסיונות אימות כושלים, שימוש בקודים שגויים, או דפוסי RF חריגים.

7. **ניהול מפתחות ופקודת חירום (Key Rotation & Kill Switch)** - אפשרות לעדכון קוד/מפתח במהלך המשימה, כולל מנגנון לחסימת רכיבים במקרי חירום.

על המחבר

יניב קפיליאנוב, חוקר סייבר, מומחה במחקר ופיתוח Low Level ומערכות הפעלה.

מוזמנים לפנות אליי בכל נושא דרך הלינקדאין שלי.

מקורות מידע

- https://github.com/PW-Sat2/PWSat2OBC - PwSat2 Github Repository
- https://pw-sat.pl/ - האתר הרשמי של הפרויקט
- https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10351029 - מאמר על אבטחת סייבר בלווייינים