Digital Whisper
Technologic Papers

איך פרצנו לרכבים בווגאס

מאת דן פלד וויטלי בריזק

הקדמה

ההיסטוריה של "פריצה" למערכות רכב (או "שדרוגים", תלוי איפה גדלתם), ניצתה אי שם בתחילת המאה ה-20, בימיה הראשונים של תעשיית הרכב. כבר באותה תקופה, מכונאים וחובבי רכב התעסקו במערכות המכניות של רכבים על מנת לשפר את הביצועים שלהם. הסצנה הזאת קיימת עד היום, כמובן, ויש לה ייצוג גם במשחקים וסרטים כמו "Need for Speed" ו"מהיר ועצבני".

אלא שבשנות ה-80 יצרניות רכב החלו לשלב מערכות אלקטרוניות (ECUs) לתוך מכוניות, ו"פריצות" מהסוג הזה קיבלו קונוטציות מרושעות יותר. ECU (או Electronic Control Unit) הוא התקן בתוך הרכב, ששולט על מערכת אלקטרונית אחת או יותר בתוכו. הבלמים, הגה כח, מערכת המולטימדיה וכדומה - כל אחת מהמערכות האלה בנפרד היא ECU.

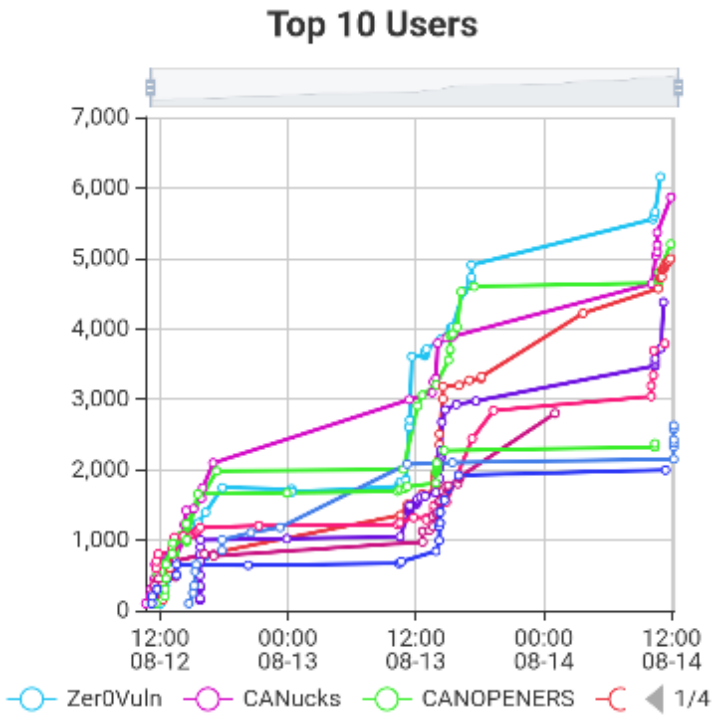
עם השנים כלי הרכב הפכו למורכבים ומקושרים יותר. רכב ממוצע כיום עשוי להכיל יותר מ-100 ECUs שונים, כולל ממשקים כגון Bluetooth, Wi-Fi, USB וסלולר. אחד העקרונות הידועים באבטחת מידע הוא שככל שבמערכת יש יותר פונקציונליות, כך משטח התקיפה גדל. עיקרון זה נכון גם לקלות השימוש - לדוגמה, כאשר מפתחים מערכת, אפשר להגן עליה בסיסמה מורכבת וארוכה שדורשת שילוב של אותיות, תווים ומספרים. הגנה מסוג זה תהפוך את המערכת למוגנת יותר, אך פחות נוחה למשתמש. מצד שני, אפשר שלא להגן על המערכת עם סיסמה כלל - מה שיגרום למערכת להיות פחות מוגנת, אבל הרבה יותר ידידותית למשתמש.

שני חוקרי האבטחה Charlie Miller ו-Chris Valasek פירסמו ב-2015 את אחד המחקרים המתוקשרים ביותר בנושא: הם הצליחו להשתלט מרחוק על Jeep Cherokee, לרבות שליטה על מערכות ההיגוי והבלמים. מחקר זה הביא לריקול של כ-1.4 מיליון רכבי Jeep והגביר משמעותית את המודעות לאבטחת מידע ברכבים.

בשנה שלאחר מכן Keen Labs, מעבדות המחקר של חברת Tencent, הדגימו יכולת פריצה לרכבי טסלה, ומאז התפרסמו עוד מחקרים רבים בתחום. בהתאם, בשנים האחרונות יצרניות הרכב וגורמי ממשל נוקטים צעדים משמעותיים על מנת להגן על כלי רכב מפני פריצה. הממשל הפדרלי עובד על חקיקת רגולציות על מנת להבטיח שמכוניות חדשות יצטרכו לעמוד ברף מינימלי של אבטחת מידע, ויצרניות הרכב משקיעות משאבים משמעותיים כדי לאבטח את המכוניות המקושרות.

כל הסיפור בתוך חידה מווגאס

DEF CON הוא אחד מכנסי הסייבר הגדולים בעולם, שמתקיים אחת לשנה בלאס וגאס. ב-2022 קבוצת הסייבר של General Motors השתתפה בתחרות ה-Car Hacking שנערכה בכנס, וזכתה במקום הראשון מתוך 96 קבוצות מרחבי העולם. התחרות כללה אתגרים רבים ממגוון תחומים, בהם פריצת Bluetooth, הצפנה, Reverse Engineering ו-CAN Bus (פרוטוקול המשמש לתקשורת ברכבים). דירוג הקבוצות לאורך התחרות, בהן הקבוצה שלי מג'נרל מוטורס (Zer0Vuln):



במאמר זה בחרתי להציג את הפתרון לאחד האתגרים בתחרות, ושמו "CRC'ly" (נהגה "Seriously"). השאלה שהוצגה למתחרים הייתה:

```
1.  This is a reverse engineering problem, in TriCore! Can you figure out what is
    going on under the hood? You might need a unicorn or a multi-headed serpent to
    help you out. You just have to figure out 4 things!
```

לשאלה צורפו הקבצים הבאים:

```
→ defcon_dump_crcly ls
start_0x10000000_0x10017FFF  start_0x40100000_0x4010FFFF  start_0x70000000_0x7003BFFF  start_0x90030000_0x9003FFFF  start_0xAFFF0000_0xAFFFFFFF
start_0x10018000_0x1001BFFF  start_0x40110000_0x40117FFF  start_0x7003C000_0x7003FFFF  start_0x90040000_0x9007FFFF  start_0xB0000000_0xB000FFFF
start_0x100C0000_0x100C17FF  start_0x401C0000_0x401C2FFF  start_0x700C0000_0x700C17FF  start_0x90080000_0x900BFFFF  start_0xB0010000_0xB001FFFF
start_0x10100000_0x1010FFFF  start_0x50000000_0x50017FFF  start_0x70100000_0x7010FFFF  start_0x900C0000_0x900FFFFF  start_0xB0020000_0xB002FFFF
start_0x10110000_0x10117FFF  start_0x50018000_0x5001BFFF  start_0x70110000_0x70117FFF  start_0x90100000_0x9010FFFF  start_0xB0030000_0xB003FFFF
start_0x101C0000_0x101C2FFF  start_0x500C0000_0x500C17FF  start_0x701C0000_0x701C2FFF  start_0x90110000_0x9011FFFF  start_0xB0040000_0xB007FFFF
start_0x30000000_0x30017FFF  start_0x50100000_0x5010FFFF  start_0x80000000_0x802FFFFF  start_0x90400000_0x9040FFFF  start_0xB0080000_0xB00BFFFF
start_0x30018000_0x3001BFFF  start_0x50110000_0x50117FFF  start_0x80300000_0x805FFFFF  start_0x90410000_0x9041FFFF  start_0xB00C0000_0xB00FFFFF
start_0x300C0000_0x300C17FF  start_0x501C0000_0x501C2FFF  start_0x80600000_0x808FFFFF  start_0x99000000_0x990FFFFF  start_0xB0100000_0xB010FFFF
start_0x30100000_0x3010FFFF  start_0x60000000_0x6003BFFF  start_0x80900000_0x80BFFFFF  start_0x99100000_0x991FFFFF  start_0xB0110000_0xB011FFFF
start_0x30110000_0x30117FFF  start_0x6003C000_0x6003FFFF  start_0x80C00000_0x80EFFFFF  start_0x99200000_0x992FFFFF  start_0xB0400000_0xB040FFFF
start_0x301C0000_0x301C2FFF  start_0x600C0000_0x600C17FF  start_0x8FFF0000_0x8FFFFFFF  start_0x99300000_0x993FFFFF  start_0xB0410000_0xB041FFFF
start_0x40000000_0x40017FFF  start_0x60100000_0x6010FFFF  start_0x90000000_0x9000FFFF  start_0xA0000000_0xA02FFFFF  start_0xC0000000_0xC00C0000
start_0x40018000_0x4001BFFF  start_0x60110000_0x60117FFF  start_0x90010000_0x9001FFFF  start_0xA0300000_0xA05FFFFF  start_0xD0000000_0xD00C0000
start_0x400C0000_0x400C17FF  start_0x601C0000_0x601C2FFF  start_0x90020000_0x9002FFFF  start_0xA0600000_0xA08FFFFF  start_regs.txt
```

בנוסף, קיבלנו קובץ שמכיל רשימת מילים (Wordlist). ניתוח ראשוני הביא אותנו למסקנות הבאות:

- כנראה שמדובר בארכיטקטורת TriCore - ארכיטקטורה זו נפוצה בעולם הרכבים ואפשר למצוא אותה ב-ECUs כגון מנוע, גיר, מערכות בטיחות ועוד.
- יתכן שהאזכור ל-Unicorn מתייחס ל-Unicorn Engine שמשמש לסימלוץ מעבדים.
- Multi Headed Serpent - תיאור שיכול להזכיר את Ghidra - כלי שמפותח על ידי ה-NSA ומיועד ל-Reverse Engineering. ראוי לציין שהכלי נחשף בשנת 2017 בהדלפה ב-WikiLeaks, והחל מ-2019 הוא הפך להיות Open Source וזמין לכלל הציבור בחינם.
- "Figure Out 4 Things" - כנראה שהתשובה מורכבת מארבעה חלקים.
- הקובץ start_regs.txt מכיל את המצב של כל אוגר:

```
→ defcon_dump_crcly head -n20 start_regs.txt
{name: 'D0', unit: 'CPU', value: 0x0186A0, core: 0}
{name: 'D1', unit: 'CPU', value: 0x037C8192, core: 0}
{name: 'D2', unit: 'CPU', value: 0x00, core: 0}
{name: 'D3', unit: 'CPU', value: 0x00, core: 0}
{name: 'D4', unit: 'CPU', value: 0x00, core: 0}
{name: 'D5', unit: 'CPU', value: 0x00, core: 0}
{name: 'D6', unit: 'CPU', value: 0x00, core: 0}
{name: 'D7', unit: 'CPU', value: 0x00, core: 0}
{name: 'D8', unit: 'CPU', value: 0x3C, core: 0}
{name: 'D9', unit: 'CPU', value: 0x3C, core: 0}
{name: 'D10', unit: 'CPU', value: 0x00, core: 0}
{name: 'D11', unit: 'CPU', value: 0x00, core: 0}
{name: 'D12', unit: 'CPU', value: 0x00, core: 0}
{name: 'D13', unit: 'CPU', value: 0x00, core: 0}
{name: 'D14', unit: 'CPU', value: 0x00, core: 0}
{name: 'D15', unit: 'CPU', value: 0xFFFC000F, core: 0}
{name: 'A0', unit: 'CPU', value: 0x00, core: 0}
{name: 'A1', unit: 'CPU', value: 0x00, core: 0}
{name: 'A2', unit: 'CPU', value: 0xF0030000, core: 0}
{name: 'A3', unit: 'CPU', value: 0xF0030000, core: 0}
```

ניתן להניח שנצטרך לבצע אמולציה ל-Firmware באמצעות Unicorn Engine והקבצים start_0x00000000_0xFFFFFFFF.

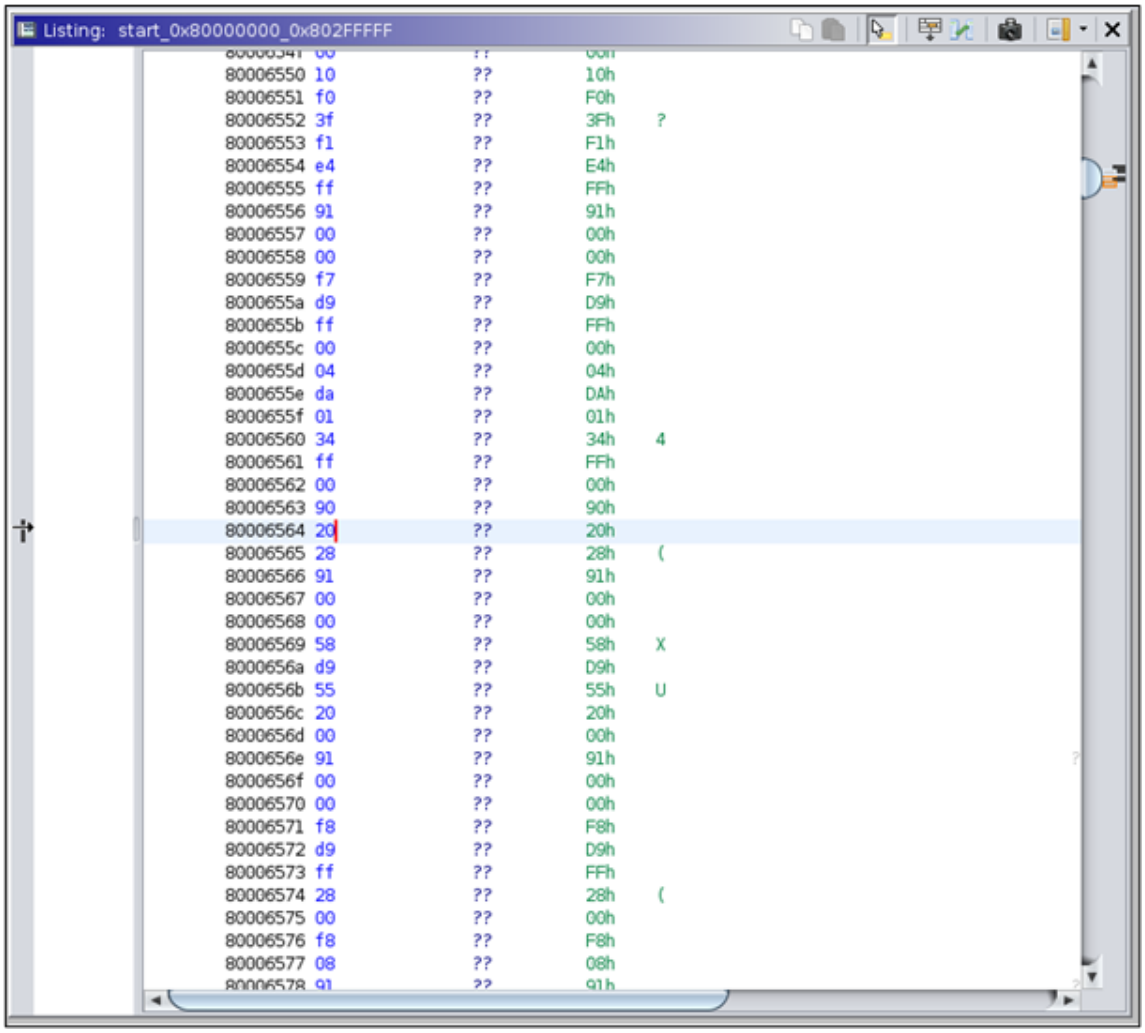
צעד לאחור, שלושה קדימה

הנדסה לאחור היא תהליך שבו משהו מורכב, כמו מכונה או טכנולוגיה, נבחן באופן קפדני ונלמד לעומק במטרה להבין כיצד הוא עובד ולשכפל את הפונקציונליות שלו. לרוב, תהליך זה כולל פירוק של המערכת, ניתוח של הרכיבים והבנה כיצד הם פועלים יחדיו. כך לדוגמה, ב-2011 איראן הצליחה לשים את ידיה על מל"ט RQ-170 של ארצות הברית ופיתחה על בסיסו מל"טים משלה, כגון השאהד 171.

כחלק מהאתגר, נדרשנו לבצע הנדסה לאחור של הקבצים שקיבלנו. הקובץ start_regs.txt מראה את הערך של אוגר ה-Program Counter (PC), כך שניתן היה להתחיל את התהליך מהכתובת שלו:

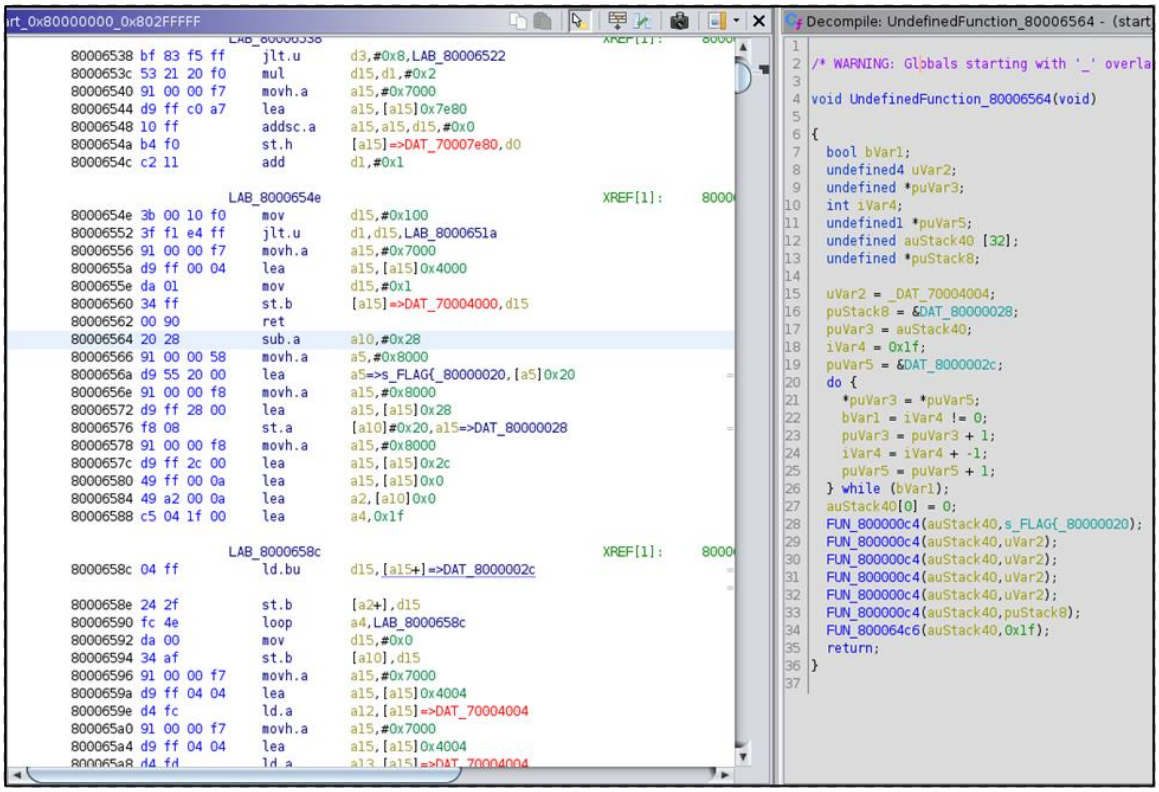
```
→ defcon_dump_crcly cat start_regs.txt | grep PC
{name: 'PC', unit: 'CPU', value: 0x80006564, core: 0}
```

כדי להתחיל בעבודה, טענו את הקובץ start_0x80000000_0x802FFFFF ל-Ghidra ובחרנו בארכיטקטורת TriCore עם Offset של 0x80000000. בהסתכלות מהירה על Offset 0x80006564, נראה ש-Ghidra לא זיהתה קוד או פונקציות:



לחיצה על D או Disassemble מראה תמונה חיובית יותר.

במסך השמאלי ניתן לראות את קוד האסמבלי, לצד המסך הימני שבו ניתן לראות את הקוד לאחר שעבר תהליך דיקומפילציה על ידי Ghidra:



בשורה 28 במסך הימני ניתן לראות אזכור למחרוזת "FLAG{", וזו אינדיקציה חיובית שאנחנו במקום הנכון. בנוסף, בשורה 15 ישנה הפנייה מעניינת למקום לא ידוע בזיכרון (בכתובת 0x700004004).

הערה: Ghidra הינה פלטפורמת Revere Engineering אשר פותחה על ידי חטיבת המחקר של ה-NSA. הפלטפורמה מבוססת קוד פתוח, תומכת בשלל פיצ'רים וניתן להרחיבה באמצעות פלאגינים, או סקריפטים ב-Java או Python.

היא תומכת במעבדים עם שלל ארכיטקטורות ונתמך ע"י כל מערכות ההפעלה המובילות היום.

כסיף דקל ו**רונן שוסטין** פרסמו במסגרת המגזין מאמר על אופן השימוש בכלי:
https://www.digitalwhisper.co.il/files/Zines/0x69/DW105-1-Ghidra_Part1.pdf

כדי לראות במה מדובר, טענו קובץ נוסף בשם start_0x70000000_0x7003BFFF. אחרי מעט שינויי שמות, קיבלנו את הקוד הבא:

Decompile: FUN_80006564 - (start_0x80000000_0x802FFFFF)

```
1
2  void FUN_80006564(void)
3
4  {
5    bool bVar1;
6    char *pcVar2;
7    int loop_len;
8    char *pcVar3;
9    char flag [32];
10   char *close_bracket;
11   undefined *first_word_in_wordlist;
12
13   first_word_in_wordlist = PTR_s_rainy_70004004;
14   close_bracket = s_}_80000028;
15   pcVar2 = flag;
16   loop_len = 0x1f;
17   pcVar3 = s_AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA_8000002c;
18   do {
19     *pcVar2 = *pcVar3;
20     bVar1 = loop_len != 0;
21     pcVar2 = pcVar2 + 1;
22     loop_len = loop_len + -1;
23     pcVar3 = pcVar3 + 1;
24   } while (bVar1);
25   flag[0] = '\0';
26   strcat(flag,s_FLAG{_80000020);
27   strcat(flag,first_word_in_wordlist);
28   strcat(flag,first_word_in_wordlist);
29   strcat(flag,first_word_in_wordlist);
30   strcat(flag,first_word_in_wordlist);
31   strcat(flag,close_bracket);
32   FUN_800064c6(flag,0x1f);
33   return;
34 }
35
```

בשורה מספר 9 ניתן לראות מערך באורך 32 בתים בשם flag. בשורה 13 קיים אזכור למילה הראשונה ב-Wordlist - והיא rainy. בשורות 18-24 המערך של flag ממולא ב-31 אותיות A. לאחר שורה 31, flag אמור להראות כך:

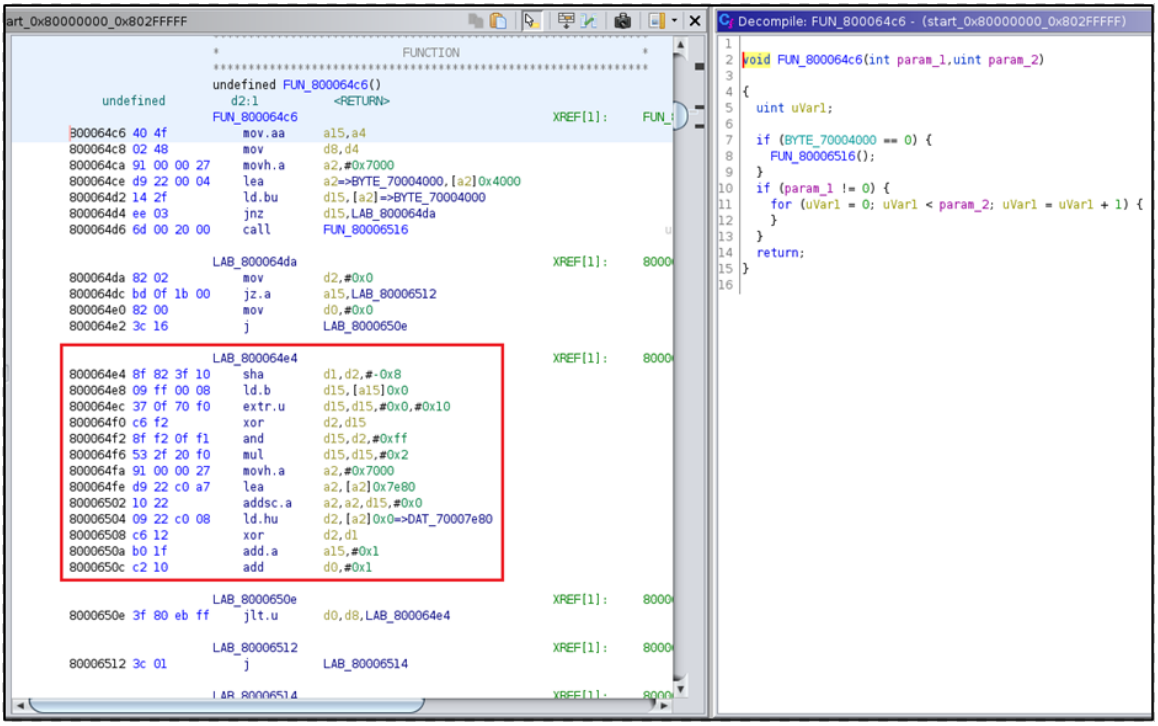
FLAG{rainyrainyrainyrainy}\x00AAAA

בשורה 32, פונקציה בשם FUN_8000646c מקבלת את המערך flag ו-0x1f בתור ארגומנטים.

כפי שניתן לראות בתמונה הבאה, Ghidra לא הצליחה לבצע Decompile לקוד שמסומן באדום. כפי שניתן לראות בתמונה הבאה, Ghidra לא הצליחה לבצע Decompile לקוד שמסומן באדום. Decompilation הוא תהליך שמנסה להפוך קוד מכונה לשפה עילית ותפקידו להקל את עבודת ה-Reverse Engineering.

התהליך יכול להיכשל או להיות לא מדויק בגלל כמה סיבות, כגון אופטימיזציות של הקומפיילר בזמן תהליך הקימפול המקורי, טכניקות של Anti-Reverse Engineering ומגבלות אחרות של ה-Decompiler.

ניתוח של קוד האסמבלי מזכיר אלגוריתם CRC. מכיוון ששם האתגר הוא "CRC'ly", ניתן להניח שאכן מדובר ב-CRC - טכניקה שנועדה לגלות שגיאות בעת העברת מידע ואחסונו. הטכניקה פועלת באמצעות צירוף קוד שנקרא Checksum אל המידע. ה-Checksum מחושב על בסיס תוכן המידע ומצורף אליו בעת השליחה או האחסון שלו. בעת קבלת המידע, ה-Checksum מחושב מחדש על ידי הצד המקבל, ואם תוצאת החישוב החדש אינה תואמת את תוצאת החישוב שצורפה למידע, המשמעות היא שקרתה תקלה כלשהי במהלך השליחה:



בחזרה מפונקציה FUN_800064c6 ניתן לראות השוואה בין הערך המוחזר מהפונקציה לבין 0xd6be, וייתכן שזה ה-CRC של הדגל שאנחנו צריכים למצוא.

```
80006602 6d ff 62 ff     call      FUN_800064c6
80006606 bb e0 6b fd     mov.u     d15,#0xd6be
8000660a 7e 22           jne       d15,d2,LAB_8000660e
8000660c 3c 01           j         LAB_8000660e

                     LAB_8000660e
8000660e 00 90           ret
```

אמולציה של הקוד תאפשר לנו להריץ אותו ולבצע עליו ניתוח דינמי. כדי לבצע את האמולציה השתמשנו ב-Unicorn Engine והסתמכנו על הדוגמה הזאת. מיפינו את start_0x80000000_0x802FFFFF לכתובת 0x80000000 ואת start_0x70000000_0x7003BFFF לכתובת 0x70000000 בהתאם ל-Datasheet (המסמך הטכני) של מעבדי TriCore:

$601C3000_H$	$6FFFFFFF_H$	-	Reserved
70000000_H	$7003BFFF_H$	240 Kbyte	Data ScratchPad RAM (CPU0)
$7003C000_H$	$7003FFFF_H$	16 Kbyte	Data Cache RAM (CPU0)
70040000_H	$700BFFFF_H$	-	Reserved
$700C0000_H$	$700C17FF_H$	6 Kbyte	Data Cache Tag RAM (CPU0)
$700C1800_H$	$700FFFFF_H$	-	Reserved
70100000_H	$7010FFFF_H$	64 Kbyte	Program ScratchPad RAM (CPU0)
70110000_H	$70117FFF_H$	32 Kbyte	Program Cache RAM (CPU0)
70118000_H	$701BFFFF_H$	-	Reserved
$701C0000_H$	$701C2FFF_H$	12 Kbyte	Program Cache TAG RAM (CPU0)
$701C3000_H$	$7FFFFFFF_H$	-	Reserved
80000000_H	$802FFFFF_H$	3 Mbyte	Program Flash (PFI0)
80300000_H	$805FFFFF_H$	3 Mbyte	Program Flash (PFI1)

לאחר מכן אתחלנו את האוגרים בהתאם לערכים שמצאנו בקובץ start_regs.txt, ולבסוף התחלנו את האמולציה בכתובת 0x800006564 (הערך של ה-Program Counter):

```
77  with open('start_0x80000000_0x802FFFFF', 'rb') as f:
78      TRICORE_CODE_0x80000000 = f.read()
79
80  with open('start_0x70000000_0x7003BFFF', 'rb') as f:
81      TRICORE_CODE_0x70000000 = f.read()
82
83  ADDRESS_0x80000000 = 0x80000000
84
85  ADDRESS_0x70000000 = 0x70000000
86
87  mu = Uc(UC_ARCH_TRICORE, UC_MODE_LITTLE_ENDIAN)
88
89  init_regs(mu)
90
91  mu.mem_map(ADDRESS_0x80000000, 0x300000)
92
93  mu.mem_map(ADDRESS_0x70000000, 0x3C000)
94
95  mu.mem_write(ADDRESS_0x80000000, TRICORE_CODE_0x80000000)
96
97  mu.mem_write(ADDRESS_0x70000000, TRICORE_CODE_0x70000000)
98
99  mu.hook_add(UC_HOOK_CODE, hook_code)
100
101 mu.emu_start(0x80006564, ADDRESS_0x80000000 + len(TRICORE_CODE_0x80000000))
102
103 print('>>> Emulation done.')
```

בזמן האמולציה הוספנו Hooks לשורות 26-31 כדי לאמת כיצד ה-Buffer של flag נראה בזמן ריצה.

Hooking היא טכניקה שבה הגורם שמבצע הנדסה לאחור משנה את הקוד של המערכת על מנת לבחון את ההתנהגות שלה. לרוב ניתן לעשות זאת על ידי שתילת קוד נוסף (Hooks) בתוך התוכנה המקורית, כדי ליירט קריאות לפונקציות, לשנות קלט/פלט וכדומה.

```
5    def hook_code(uc, address, size, user_data):
6        if address == 0x800065c2:
7            a4 = uc.reg_read(UC_TRICORE_REG_A4)
8            print('>>> Before line 26: ' + str(uc.mem_read(a4,31)))
9        if address == 0x800065c6:
10           a4 = uc.reg_read(UC_TRICORE_REG_A4)
11           print('>>> After line 26: ' + str(uc.mem_read(a4,31)))
12       if address == 0x800065d0:
13           a4 = uc.reg_read(UC_TRICORE_REG_A4)
14           print('>>> After line 27: ' + str(uc.mem_read(a4,31)))
15       if address == 0x800065da:
16           a4 = uc.reg_read(UC_TRICORE_REG_A4)
17           print('>>> After line 28: ' + str(uc.mem_read(a4,31)))
18       if address == 0x800065e4:
19           a4 = uc.reg_read(UC_TRICORE_REG_A4)
20           print('>>> After line 29: ' + str(uc.mem_read(a4,31)))
21       if address == 0x800065ee:
22           a4 = uc.reg_read(UC_TRICORE_REG_A4)
23           print('>>> After line 30: ' + str(uc.mem_read(a4,31)))
24       if address == 0x800065fa:
25           a4 = uc.reg_read(UC_TRICORE_REG_A4)
26           print('>>> After line 31: ' + str(uc.mem_read(a4,31)))
27       if address == 0x8000660e:
28           uc.emu_stop()
```

לאחר הרצת הקוד, הפלט בשורה 31 מייצג את הפורמט שבו ה-Buffer צריך להיראות:

```
➜ python git:(master) ✗ python3 debug.py
>>> Before line 26: bytearray(b'\x00AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA')
>>> After line 26: bytearray(b'FLAG{\x00AAAAAAAAAAAAAAAAAAAAAAAAA')
>>> After line 27: bytearray(b'FLAG{rainy\x00AAAAAAAAAAAAAAAAAAA')
>>> After line 28: bytearray(b'FLAG{rainyrainy\x00AAAAAAAAAAAAAA')
>>> After line 29: bytearray(b'FLAG{rainyrainyrainy\x00AAAAAAAAAA')
>>> After line 30: bytearray(b'FLAG{rainyrainyrainyrainy\x00AAAAA')
>>> After line 31: bytearray(b'FLAG{rainyrainyrainyrainy}\x00AAAA')
>>> Emulation done.
```

Hook נוסף שצירפנו לקוד אפשר לנו לראות את ה-CRC:

```
>>> D2 after line 31: 0xd8e2
```

הפתרון

פתרון כל אתגר זיכה את הקבוצות ב"דגל" שאותו היו צריכים להזין למערכת על מנת לקבל את הנקודות. אחרי שהצלחנו לדבג ולבצע הנדסה לאחור של הפונקציות הרלוונטיות, הגענו למצב שאנחנו יכולים לכתוב סקריפט שיבצע Brute Force על הדגל. ביצענו שוב אמולציה לאותן כתובות זיכרון, אבל הפעם התחלנו בכתובת 0x80000646c - פונקצית ה-CRC. פונקציה זו מצפה לקבל שני ארגומנטים:

- הכתובת שבה "הדגל" אמור להיות
- אורך "הדגל"

הסקריפט שכתבנו מבצע לולאה שמנסה את כל המילים האפשריות מהמילון שקיבלנו, כותב את הדגלים לכתובת 0x70000000 וקורא לפונקציה שמחשבת את ה-CRC בכתובת 0x80000646c0.

```python
from __future__ import print_function
from unicorn import *
from unicorn.tricore_const import *
import struct
import itertools

def hook_code(uc, address, size, user_data):
    if address == 0x80006514:
        crc = hex(uc.reg_read(UC_TRICORE_REG_D2))
        if crc == '0xd6be':
            print('>>> Found a flag: ' + uc.mem_read(0x70000000,26).decode())
        uc.emu_stop()

def init_regs(mu):
    mu.reg_write(UC_TRICORE_REG_D4, 0x1F)
    mu.reg_write(UC_TRICORE_REG_A4, 0x70000000)
    mu.reg_write(UC_TRICORE_REG_FCX, 0x070E71) #will crash without

with open('start_0x80000000_0x802FFFFF', 'rb') as f:
    TRICORE_CODE_0x80000000 = f.read()

with open('start_0x70000000_0x7003BFFF', 'rb') as f:
    TRICORE_CODE_0x70000000 = f.read()

ADDRESS_0x80000000 = 0x80000000

ADDRESS_0x70000000 = 0x70000000

mu = Uc(UC_ARCH_TRICORE, UC_MODE_LITTLE_ENDIAN)

init_regs(mu)

mu.mem_map(ADDRESS_0x80000000, 0x300000)

mu.mem_map(ADDRESS_0x70000000, 0x3C000)

mu.mem_write(ADDRESS_0x80000000, TRICORE_CODE_0x80000000)

mu.mem_write(ADDRESS_0x70000000, TRICORE_CODE_0x70000000)

mu.hook_add(UC_HOOK_CODE, hook_code)

words = ['spoon', 'scary', 'asked']
for first, second, third in itertools.permutations(words):
    for i in range(0x00004004, 0x00007E7D, 4):
        offset = struct.unpack('<L', TRICORE_CODE_0x70000000[i:i+4])[0]
        word = TRICORE_CODE_0x80000000[offset-0x80000000:offset-0x80000000+5]
        mu.mem_write(ADDRESS_0x70000000, b'FLAG{' + bytes(f'{first}{second}{third}', 'utf-8') + word + b'}\x00AAAA')
        mu.emu_start(0x800064C6, ADDRESS_0x80000000 + len(TRICORE_CODE_0x80000000))

print('>>> Emulation done.')
```

עם סיום הריצה, קיבלנו את הדגל:

```
→ python git:(master) ✗ python3 solution.py
>>> Found a flag: FLAG{scaryspoonaskedquite}
>>> Emulation done.
```

לסיכום

עולם הרכב נמצא בעיצומה של מהפכה המייצגת שינוי פרדיגמה ומציעה פוטנציאל גדול לחדשנות, התאמה אישית וחוויית משתמש משופרת. מינוף תוכנה וקישוריות יכול להפוך כלי רכב לפלטפורמות חכמות שיודעות להסתגל לצרכי המשתמש המשתנים ולטכנולוגיות המתפתחות. אך הפוטנציאל הזה מגיע עם סיכוני אבטחת סייבר גבוהים שדורשים מענה ותעדוף משמעותי.

לצד המשך ההתקדמות הטכנולוגית, התעשייה חייבת לתת מענה לנושא האבטחה - החל משלב התכנון והפיתוח ועד פריסה ותחזוקה. כך יהיה ניתן להבטיח שהיתרונות של כלי רכב עתידיים ימומשו מבלי לפגוע בבטיחות ובפרטיות של נוסעי הרכב.

על המחברים

ויטלי בריזק הוא ראש צוות מחקר ובדיקות חדירה ב-General Motors, והוא בעל שנות ניסיון רבות בפריצות חומרה ותוכנה.

דן פלד הינו מנהל קבוצת הסייבר ב-General Motors ישראל. מצאתם את המאמר מעניין? חושבים שיש לכם את מה שנדרש כדי לעסוק בתחום? הצוות של דן מגייס ומחפש חוקרי/ות חולשות שיטצרפו לשורותיו. ניתן להגיש מועמדות בקישור הבא:

https://lnkd.in/dbgskJgD

המאמר פורסם לראשונה באתר גיק-טיים:

https://www.geektime.co.il/how-cars-get-hacked-by-rolling-code-vulnerabilities/