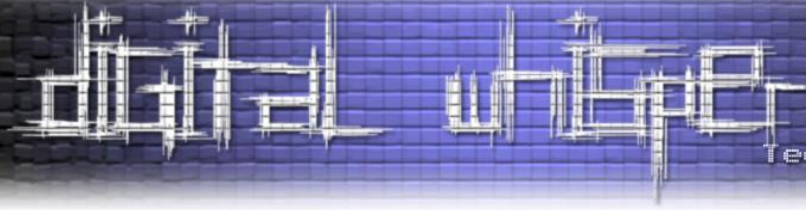


If You Build It - It Will (Cross) Compile

מאת ענבר (Blondy314) רותם

הקדמה

דמיינו את התרחיש הבא: כתבתי קוד שארצה להריץ על מכונה מסוימת, נניח על טלפון Android שכידוע מריץ Linux על מעבד ARM, בעוד סביבת הפיתוח שלי היא Ubuntu שרץ על מעבד Intel x86. אם אקמפל את הקוד על המכונה שלי באמצעות הקומפיילר שיש עליה ייווצר לי בינארי אשר יוכל לרוץ רק על מעבדי Intel. במידה ואעתיק אותו לטלפון שלי ואריץ אותו אקבל שגיאה:

```
./main
sh: ./main: not executable: 32-bit ELF file
```

למה זה קורה?

כאשר מקמפלים קוד עם קומפיילר מסוים הוא מתרגם את השפה "העילית" (במקרה שלנו שפת C) לשפת מכונה ומייצר קובץ בינארי המורכב מקוד אסמבלי עם op-code-ים שהמעבד מכיר. המעבדים השונים תומכים בפקודות ובמאפיינים שונים ולכן לא יכולים להריץ קוד שקומפל עבור מעבד אחר.

ניתן לחשוב על עניין זה כסט הוראות שאתם מנסים להשליך ממנגנון אחד למנגנון שני שעובד אחרת. לדוגמא, הפעולות שנבצע בהגה של מטוס שונות מהגה של רכב - במידה ונמשוך את הגה המטוס כלפינו, הוא יתרומם כלפי מעלה ואילו אם נעשה זאת ברכב - לא יקרה כלום.

אז איך בכל זאת אוכל לקמפל את הקוד עבור מעבד ARM? האם הדרך היחידה היא לקמפל אותו על המעבד הספציפי?

כמובן שהתשובה היא שלילית (אחרת המאמר הזה היה די קצר..)

במאמר הקרוב אענה בהרחבה על שאלה זו, ואציג את הפתרונות לעניין. אתייחס לפיתוח בסביבת לינוקס אך המנגנון רלוונטי גם למערכות הפעלה אחרות כגון Windows או Mac.

לפני שנתחיל לצלול לעומק הנושא בואו ניישר קו עם מספר מושגי בסיס בתחום:

- **ELF** - Executable and Linkable Format, קובץ בינארי בפורמט לינוקסי (יכול להיות קובץ הרצה, ספרייה, obj)
- **Open Source** - אוסף של קוד פומבי שניתן להורדה חינם. לדוגמא Linux הינו open source שניתן להוריד, לקרוא ולשנות כרצוננו ואילו מערכת ההפעלה Windows אינה (מה שנקרא Closed Source)
- **GNU** - פרויקט open source שמהווה את רוב סביבת ה-user mode של לינוקס (לינוקס מתייחס בעיקר ל-Kernel). GNU הינם ראשי תיבות מחזוריות - GNU Not Unix (זהו טרנד נפוץ בתחום המחשבים לעשות ראשי תיבות מחזוריות כגון PHP, CURL, Nagios, XBMC ועוד)
- **gcc** - GNU Compiler Collection, קומפיילר של GNU שתומך בשלל שפות תכנות (C, Objective-C, Java, GO ועוד) ומהווה את הקומפיילר הסטנדרטי ברוב המכונות שמבוססות UNIX
- **libc** - C Standard Library, ספרייה אשר עוטפת את ה-syscall-ים במערכת ומספקת למשתמש סט של פונקציות הנוחות לשימוש
- **glibc** - GNU libc, הספרייה הנפוצה ביותר בשימוש בלינוקס
- **Makefile** - קובץ המאפשר להריץ את הקומפיילר ובו מפורטות "הוראות" עבורו

וכעת, לענייננו...

Cross Compiling

Cross Compiling (להלן CC) הינה שיטה לקמפל קוד על מכונה A (ה-Host) אשר יוכל לרוץ על מכונה B (ה-Target). באמצעות CC אנו בעצם מקמפלים קומפיילר אשר ידע ליצור קובץ בינארי שיוכל לרוץ על ה-Target.

השוני בין ה-Host לבין ה-Target יכול להיות במספר מאפיינים:

1. **מעבד**: ישנו מגוון רב של סוגי מעבדים: Intel, ARM, MIPS, PPC, SPARC ועוד
    - יכול להיות שונה גם ב-Bitness (32 \ 64 ביט)
    - יכול להיות שונה גם ב-Endianness אשר קובע את הסדר בו הבתים מסודרים בזיכרון (Little \ Big)
    - לכל מעבד יש גם תת דגם שיכול להיות בעל Instruction Set שונה. למשל עבור ARM ישנם: Armv9, Armv7, Cortex ועוד

2. **מערכת הפעלה**: לעומת עולם ה-Windows בו יש רק מערכת הפעלה אחת (אין הבדל בקמפול בין XP, Win7, Win10 וכו' אלא רק 64 \ 32 ביט), בעולם ה-Unix יש שלל מערכות הפעלה ששונות ביניהן בקמפול:
   - Linux
   - FreeBSD
   - Darwin (מערכת ההפעלה מבוססת Unix של חברת Apple כגון Mac ו-iPod \ iPhone)
   - Solaris (מערכת הפעלה שמשמשת בין היתר רכיבי Oracle)
   - ועוד

כאשר מדברים על Cross Compiling, ניתן להתבלבל עם קונספט שונה של המרת קוד משפה לשפה שנקרא Source To Source. למשל המרת קוד ישן שנכתב בשפת Fortran לשפת C. תהליך כזה מתבצע ע"י רכיב המכונה Trans-compiler או Transpiler אך הנושא הזה אינו ב-scope שלנו ולכן לא ניכנס אליו.

אם נחזור לדוגמא הקודמת, על אותה מכונת Ubuntu אני אוכל לקמפל ELF המותאם לרוץ על טלפון המריץ Android.

אז למה בעצם צריך Cross Compiling? למה לא לקמפל על מכונה שמתאימה ל-spec של ה-target?

התשובה מורכבת ממספר סיבות:

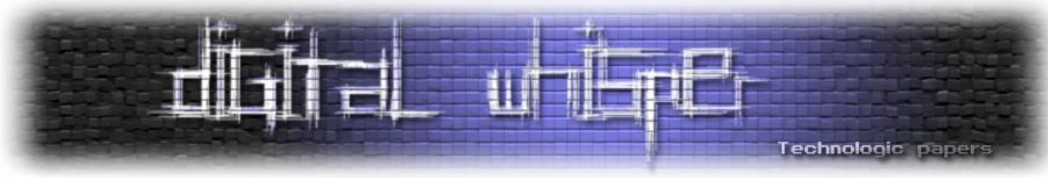
- תשתית - לא תמיד יש את התשתית לקמפול על מכונת היעד. למשל בסביבות Embedded שאינן חזקות מספיק כדי להריץ עליהן קומפיילר כדוגמת תנור מטבח או מכונת כביסה אשר מורץ עליהן לינוקס
- מהירות וביצועים - מכונת היעד יכולה להיות מאוד איטית ולכן ייקח זמן רב לקמפל קוד עליה
- זמינות - לא תמיד תהיה ברשותכם המכונה שאליה מיועד הבינארי שלכם. סיבה נוספת היא גם כלכלית, למה לרכוש רכיב אחר שיכול להיות יקר כשניתן לקמפל על מה שכבר יש
- רובוסטיות - אפשר להקים סביבה אחת בה ניתן לקמפל למגוון רב של סביבות אחרות באמצעות סקריפט אחד שמורץ בזמן ה-build של המוצר. דמיינו סט של makefile-ים: make-linux-armv7, make-mips, make-darwin וכו'

דוגמא מעניינת (ודי ישנה) הינה Canadian Cross Compiling שבה בונים Cross Compiler שבונה Compiler נוסף שמהווה Cross Compiler למכונה נוספת. כלומר, במכונה A מקמפלים קומפיילר אשר יוכל לרוץ על מכונה B עליה הוא יקמפל קומפיילר נוסף שיוכל לרוץ על מכונה C. בהרצה בשיטה זו ישתמשו בדגלים:

```
--build=[Compiler A Host] --host=[Compiler A Target] --target=[Compiler
Target]
```

דוגמא לשימוש בטכניקה זו היא קמפול של Cross Compiler על לינוקס עבור Windows אשר ירוץ על Windows ויקמפל בינארים עבור מעבד MIPS. הסתבכתם? גם אני...

הטכניקה נקראת כך מהסיבה המוזרה שבתקופה שבה חשבו עליה היו בקנדה שלוש מפלגות פוליטיות.



Binutils

במסגרת ה-Cross Compiling אנחנו נקבל, בין היתר, סט של Binutils (Binary Utilities) - בינאריים אשר מהווים כלים ליצירה וניהול של בינאריים שנוצרים בקמפול. דוגמאות לכלי Binutils חשובים:

- **as** - האסמבלר שמהווה את ה-backend של הקומפיילר (מוכר גם כ-GAS - GNU Assembler)
- **ld** - הלינקר שלוקח אחד או יותר קבצי obj (קבצי הביניים שהקוד מקומפל אליהם) ומקבץ אותם לכדי בינארי אחד (שיכול להיות executable, lib או obj אחר)
- **ar** - יוצר קבצי archive, בפרט קבצי ספריות סטטיות (למשל libc.a)
- **objdump** - מדפיס מידע אודות בינארי נתון ויכול לשמש כ-disassembler
- **readelf** - מציג מידע אודות מבנה הבינארי. למשל את ה-symbol table או את ה-header-ים של ה-ELF
- **strip** - מוריד מידע לא חיוני מהבינארי אשר מקטין את הגודל של הבינארי (כגון מידע עבור דיבוג ו-symbol-ים). על כן, בינארי שהוא stripped יהיה קשה יותר לדבג או לעשות לו Reverse Engineering

שימו לב כי כלי ה-Binutils הינם Platform Dependent ולכן גם הם נדבך של ה-Cross Compiling אך הם אינם כוללים את הקומפיילר עצמו. כלומר, אם נרצה להריץ objdump על בינארי שקומפל ל-MIPS נצטרך לקחת את ה-objdump שנוצר ב-CC ולא ב-objdump של ה-host (שככל הנראה לא יעבוד טוב).

Toolchain

Toolchain הינו אוסף הכלים הנדרשים בשביל קמפול של בינארי - קומפיילר, Binutils וה-libc (אוסף הספריות שעוטפות את ה-syscalls של המערכת). כאשר עוסקים ב-cross compiling ייווצר cross toolchain.

אז איך בעצם מקבלים toolchain באמצעותו אוכל לקמפל קוד למכונה אחרת? ובכן, יש שתי דרכים עיקריות:

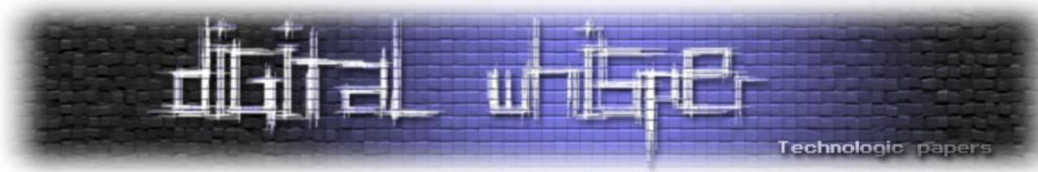
1. להוריד toolchain מוכן מהאינטרנט - ישנם לא מעט אתרים שבהם יש כבר toolchain-ים שמותאמים למגוון רחב של מערכות. למעשה, עבור לא מעט רכיבים, החברה אשר מייצרת אותם מספקת את ה-toolchain המתאים לפלטפורמה. לדוגמא:

https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

2. להשתמש ב-Buildroot - עליה נרחיב כעת

Buildroot

Buildroot הינה סביבה שנועדה להקל על בניית toolchain-ים אשר מורכבת מאוסף של makefile-ים. היא פורסמה לראשונה ב-2001 ונשארה מתוחזקת מאז כאשר עדכונים יוצאים מדי כמה חודשים. אגב, המערכת עצמה הינה open source שניתנת להורדה בחינם וניתן לעשות בה שינויים. היתרון של



Buildroot והגורם להצלחה של המערכת הינו הקלות והנוחות של השימוש בה. אז במקום להכביר במילים על הסביבה בואו נצלול פנימה להדגמת השימוש בה...

לשם ההדגמה אשתמש ב-VM של Ubuntu x86 עם גישה לאינטרנט:

- הורידו את הגרסה האחרונה מ-https://buildroot.org/download.html (בעת כתיבת שורות אלו, הגרסא האחרונה הינה 2017.02.8). תקבלו קובץ tar (קובץ archive בדומה ל-zip או rar)
- העתיקו את הקובץ tar ל-VM (באמצעות scp או העתק-הדבק במידה ומותקן לכם vmware tools או בכל דרך הנוחה לכם)
- פתחו את הקובץ ונווטו לתיקיה שנוצרה:

```
tar -xvf buildroot-2017.02.8.tar.gz
cd buildroot-2017.02.8
```

בדומה לחבילות לינוקסיות אחרות ניתן לקבל תפריט עם "GUI" סטייל DOS ע"י הפקודה הבאה:

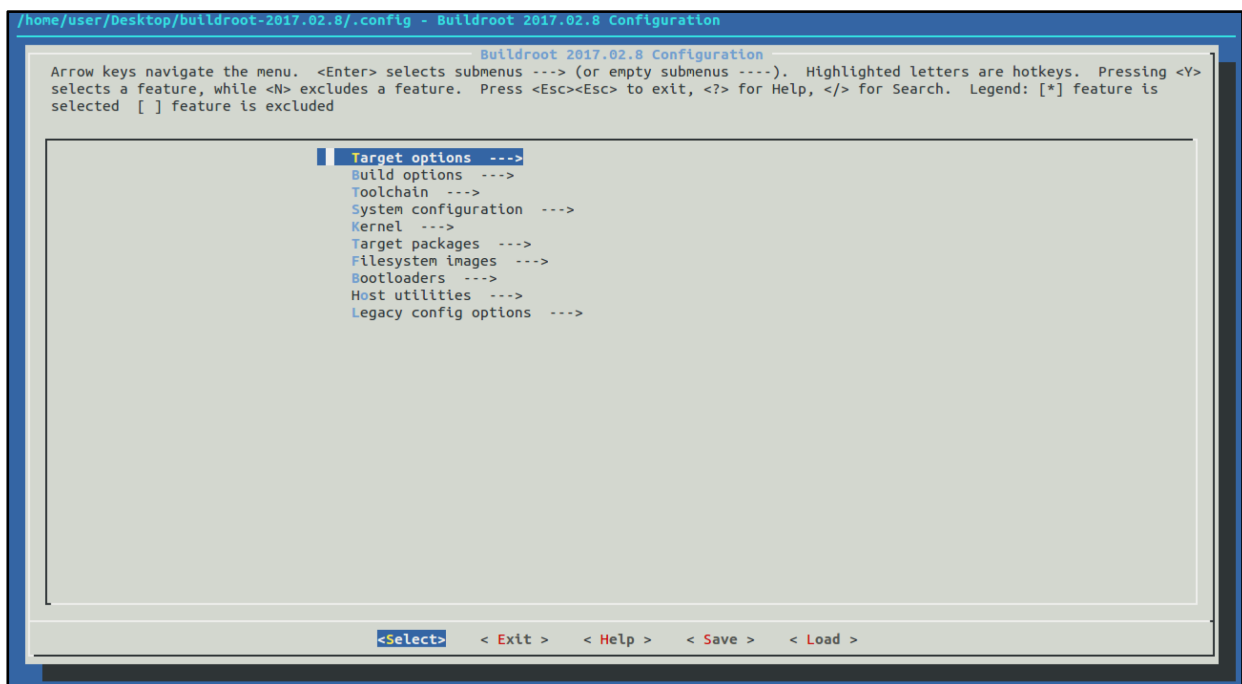
```
make menuconfig
```

יתכן כי ה-make יכשל וידרוש את הספרייה של ncurses[1], התקינו ע"י:

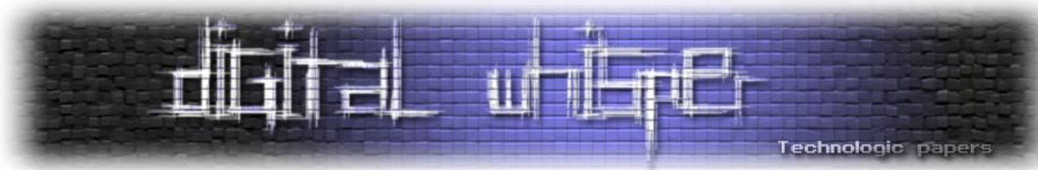
```
apt-get install libncurses-dev
```

ניתן לנווט בקלות ע"י Enter לכניסה פנימה, ESC לחזרה לשלב הקודם ו-"/" עבור חיפוש

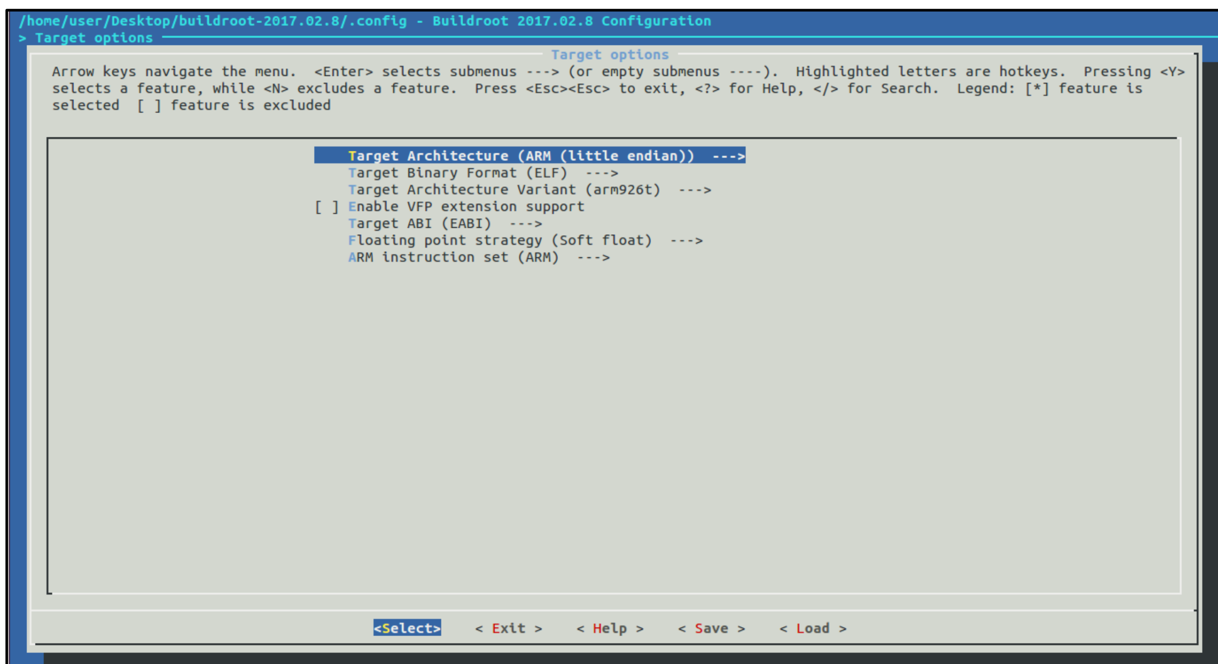
- בשלב הזה יבוצע תהליך קצר של קומפילציה ולאחריו יפתח המסך הבא:

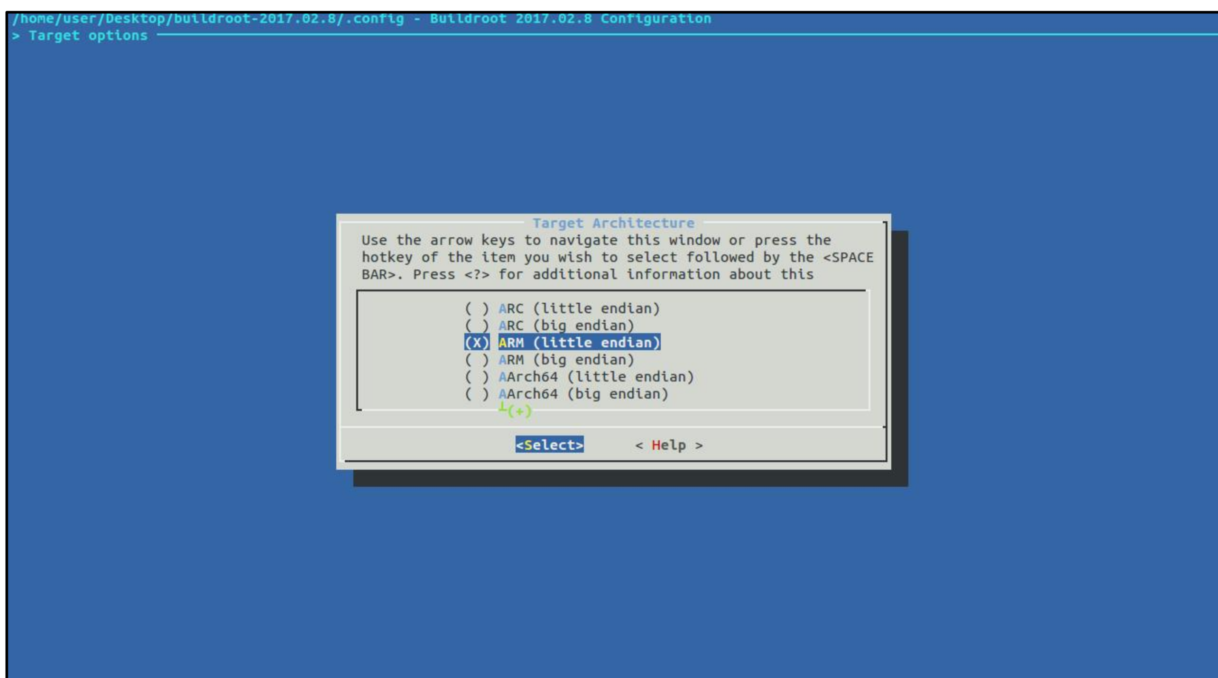


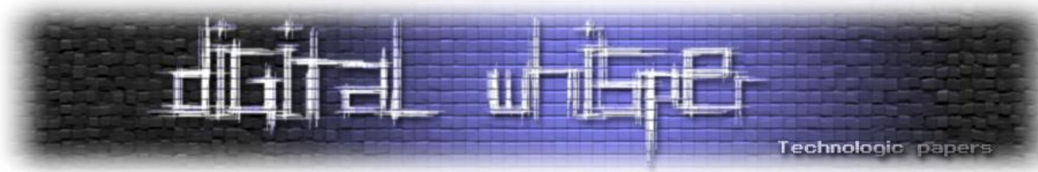
[1] (new curses) ncurses הינה חבילה המאפשרת פיתוח של אפליקציות עם GUI הרצות תחת Terminal



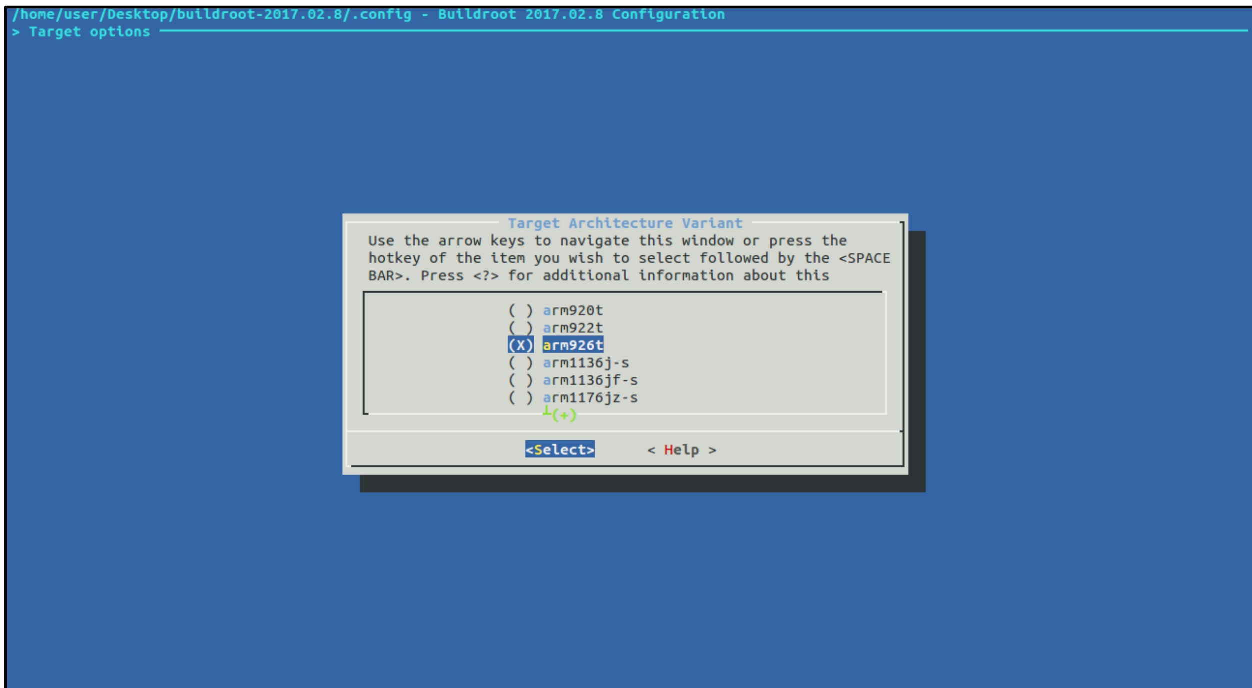
התפריט הנ"ל מאפשר להגדיר כמעט כל פרמטר אותו תרצו לשנות עבור ה-toolchain שלכם, לדוגמא, אם ניכנס לתפריט ה-Target options ולאחריו ל-Target Architecture נוכל לקבוע את הארכ' של המעבד. בדוגמא שלנו נרצה לבחור ARM. שימו לב כי ניתן להגדיר גם את ה-Endianness של המעבד (Little or Big):





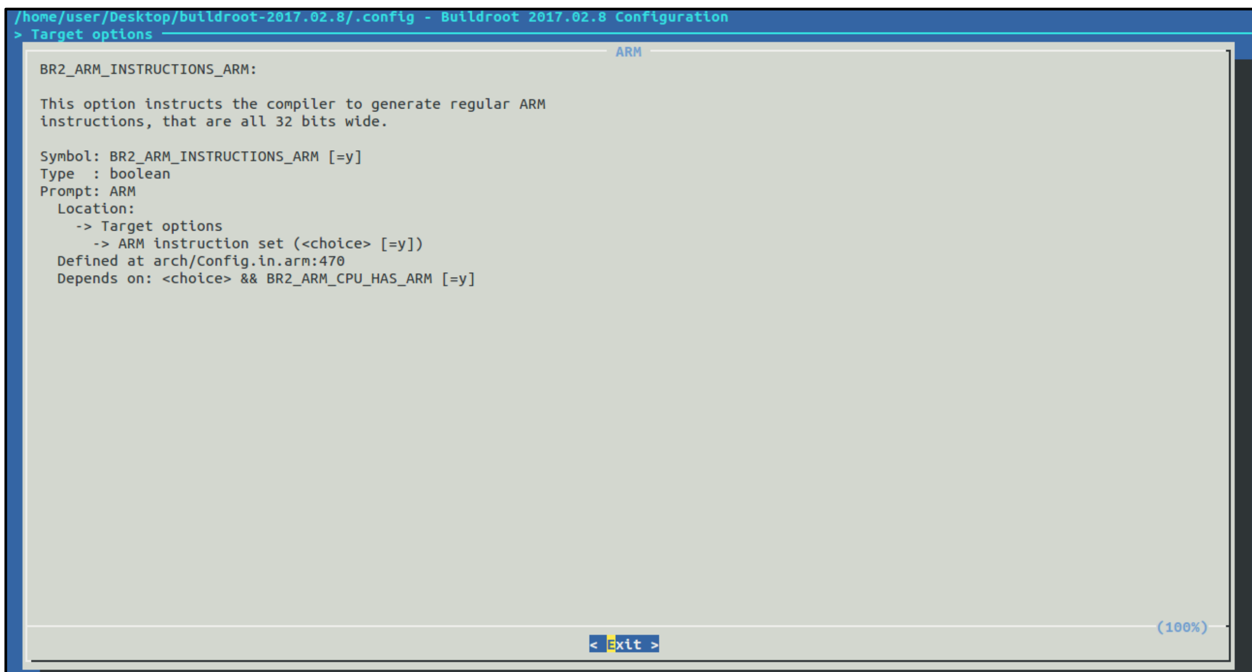


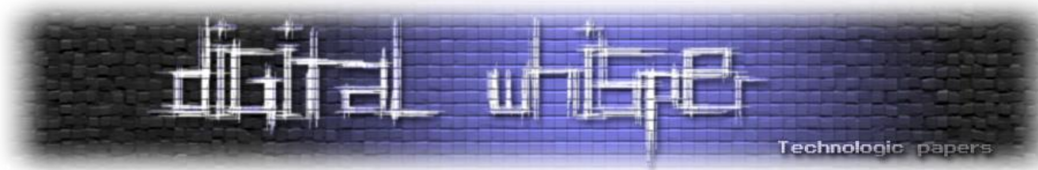
בהינתן שבחרנו מעבד, ניתן גם לבחור את הסוג היותר מפורט שלו, ע"י התפריט " Target Architecture Variant":



כל שינוי בתפריט משפיע על התפריטים האחרים שנקבעים על פיו.

כאשר יש אפשרות שאתם לא בטוחים לגביה ניתן ללחוץ על Help (או בקיצור ללחוץ על H) ולקרוא את ההסבר על האופציה, ובנוסף לכך, לקבל את המידע אודות המשתנה אותו היא מגדירה ב-Makefile:



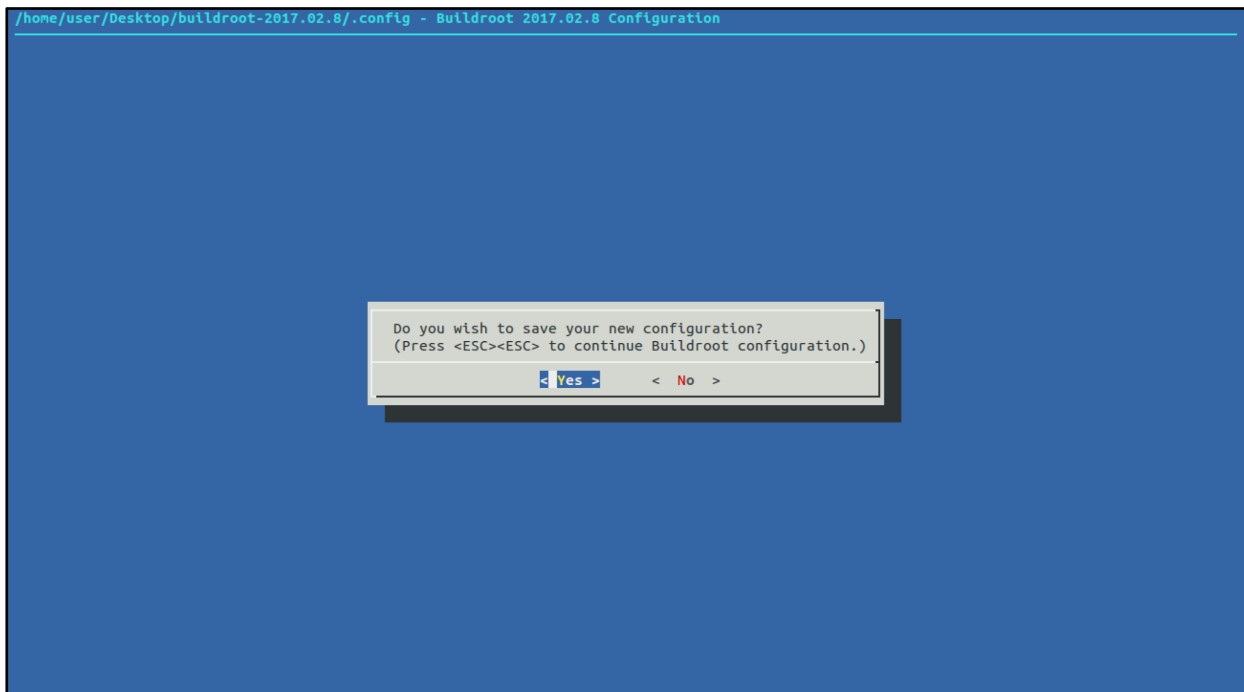


אפשרויות מעניינות נוספות הקיימות בתפריטים אלו:

- **Toolcahin -> Enable C++ support** - יש לבחור באפשרות הזו במידה ואתם רוצים לקמפל קוד C++. אחרת לא מומלץ לסמן אותה כיוון שזה יגדיל את זמן יצירת ה-toolchain וינפח את גודלו.
- **Toolchain -> C library** - ניתן לבחור באיזה libc להשתמש (uClibc, glibc, musl ועוד) - לכל ספרייה יתרונות וחסרונות משלה (גודל, תאימות לאחור, מהירות ריצה, תמיכה במעבדים וכו')

הרגישו חופשי לשוטט במגוון האפשרויות ולשחק איתן.

צאו מהתפריט ושמרו את השינויים. השינויים משפיעים על הקובץ config. שנמצא בתיקייה הנוכחית:



- כעת, הריצו את הפקודה make, בשלב זה אתם יכולים ללכת להכין קפה או אפילו ללכת להביא קפהTake away כיוון שפעולה זו עלולה לקחת הרבה זמן (סדר גודל של חצי שעה)
- את קובץ ה-config. מומלץ להעתיק הצידה ולתת לו שם ייחודי (למשל config-arm926le.). וכך בכל פעם שרוצים לבצע שינויים עבור פלטפורמה מסוימת ניתן להעתיק אותו לתיקייה של ה-Buildroot בשם config. (שימו לב שכל שינוי בתפריט דורס את הקובץ הזה)

- הפלט של הבילד נמצא בתיקיית /output/host/usr/bin/. הסתכלו על מה נוצר באמצעות:

```
ls -al ./output/host/usr/bin
```

פלט לדוגמא:

```
arm-buildroot-linux-uclibcgnueabi-addr2line
arm-buildroot-linux-uclibcgnueabi-ar
arm-buildroot-linux-uclibcgnueabi-as
arm-buildroot-linux-uclibcgnueabi-cc -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-cc.br_real
arm-buildroot-linux-uclibcgnueabi-c++filt
arm-buildroot-linux-uclibcgnueabi-cpp -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-cpp.br_real
arm-buildroot-linux-uclibcgnueabi-elfedit
arm-buildroot-linux-uclibcgnueabi-gcc -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-gcc-5.4.0 -> toolchain-wrapper
arm-buildroot-linux-uclibcgnueabi-gcc-5.4.0.br_real
arm-buildroot-linux-uclibcgnueabi-gcc-ar
arm-buildroot-linux-uclibcgnueabi-gcc.br_real
arm-buildroot-linux-uclibcgnueabi-gcc-nm
arm-buildroot-linux-uclibcgnueabi-gcc-ranlib
arm-buildroot-linux-uclibcgnueabi-gcov
arm-buildroot-linux-uclibcgnueabi-gcov-tool
arm-buildroot-linux-uclibcgnueabi-gprof
arm-buildroot-linux-uclibcgnueabi-ld
arm-buildroot-linux-uclibcgnueabi-ld.bfd
arm-buildroot-linux-uclibcgnueabi-ldconfig -> ldconfig
arm-buildroot-linux-uclibcgnueabi-ldd -> ldd
arm-buildroot-linux-uclibcgnueabi-nm
arm-buildroot-linux-uclibcgnueabi-objcopy
arm-buildroot-linux-uclibcgnueabi-objdump
arm-buildroot-linux-uclibcgnueabi-ranlib
arm-buildroot-linux-uclibcgnueabi-readelf
arm-buildroot-linux-uclibcgnueabi-size
arm-buildroot-linux-uclibcgnueabi-strings
arm-buildroot-linux-uclibcgnueabi-strip
```

לאחר שסיימנו להכין את ה-Buildroot, נקמפל תוכנה קטנה:[2]

```
#include <stdio.h>
#include <sys/utsname.h>

int main()
{
    struct utsname u = { 0 } ;
    uname(&u);

    printf("%s\n", u.sysname);
    printf("%s\n", u.machine);

    return 0;
}
```

ונריץ:

```
./output/host/usr/bin/arm-buildroot-linux-uclibcgnueabi-gcc main.c --static -o
unameit
```

[2] uname - הינו syscall אשר מחזיר מידע אודות הרכיב, מוזמנים לקרוא עליו ע"י man 2 uname

רצוי להוסיף את הדגל static-- בשביל שהבינארי יקומפל סטטית. ברוב המקרים הספריות במכונת ה-target לא יהיו תואמות לספריות איתן התקמפלנו או שאולי אפילו יהיו חסרות, לכן עדיף להביא את כל התלויות איתנו.

נבדוק מה הפורמט של הבינארי שיצא:

```
file ./unameit
./unameit:  ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
```

בואו ונבין את פלט הפקודה:

- **ELF 32-bit** - קובץ UNIX-י עבור מעבד 32 ביט
- **LSB** - Least Significant Bit כלומר הוא Little Endian (אחרת היה כתוב MSB)
- **ARM** - מעבד מתוצרת ARM
- **EABI5 version 1** - Embedded Application Binary Interface - הגרסה של ה-interface. משתמשים בזה עבור מעבדי ARM כדי להגדיר סטנדרטיזציה בין מנגנונים low level ל-high level
- **statically linked** - הבינארי מקומפל סטטית (כיוון שהוספנו את הדגל static--)
- **not stripped** - לא הורדנו ממנו את ה-symbol-ים

כעת נוכל לעלות אותו למכשיר האנדרואיד ולהריץ אותו (לאלו מכם עם iPhone - זה קצת יותר מאתגר להריץ בינאריים כיוון שצריך Jailbreak למכשיר). הפלט שנקבל:

```
192.168.1.25 - PuTTY
login as: root
SSHDroid
Use 'root' as username
Default password is 'admin'
root@192.168.1.25's password:
dreamqlteue:/data/data/berserker.android.apps.sshdroid/home $ ./unameit
Linux
armv8l
dreamqlteue:/data/data/berserker.android.apps.sshdroid/home $
```

ברכות, הרצתם את הבינארי ה-Cross Compiled הראשון שלכם!

Toolchain vs Buildroot

כפי שוודאי שמתם לב, Toolchain מוכן הינו הפתרון "המהיר". הוא חוסך את זמן הקמפול ב-Buildroot ואת ההתעסקות למי שלא מכיר כ"כ את הנושא (מי שלא קרא את המאמר הנ"ל...). מצד שני, הוא פחות גמיש ועל כן הוא מומלץ במקרים בהם יודעים בדיוק את הפלטפורמה אליה רוצים לבנות ולא רוצים להתעסק בדברים מסביב.

בנוסף, לא לכל פלטפורמה קיים Toolchain שניתן להורדה (או שלפעמים הוא אינו חינם), אך שוב, מצד שני, גם לא כל פלטפורמה נתמכת ב-Buildroot.

לכן, יש להפעיל שיקול דעת כאשר מחפשים דרך ליצור Toolchain לפלטפורמה מסוימת.

קמפול קוד Open Source חיצוני

ישנן לא מעט חבילות Open Source שהינן Cross Platform כלומר ניתן לעשות להן Cross Compiling.

עכשיו, כשיש לנו כבר toolchain בוא נראה איך נוכל לקמפל open source, לדוגמא tcpdump, כך שנהיה מסוגלים להריץ אותו על מכשיר האנדרואיד ולהסניף את התקשורת.

- בשביל נוחות, נוסיף ל-PATH שלנו את התיקייה עם ה-cross compiler שקמפלנו:

```
PATH=$PATH:/home/user/Desktop/buildroot-2017.02.8/output/host/usr/bin/
```

- בכדי לקמפל tcpdump יש צורך לקמפל גם את libpcap (הספרייה המשמשת בין היתר להסנפה)
- הורידו את הקוד מה-repository וכנסו לתיקייה:

```
apt-get source libpcap
apt-get source tcpdump
cd libpcap-1.5.3
```

- כעת נריץ את סקריפט ה-configure אשר יצור קובץ Makefile שמותאם לפרמטרים שניתן:

```
CC=arm-buildroot-linux-uclibcgnueabi-gcc ./configure --host=arm-linux --with-pcap=linux
```

  - **CC** : קובע מה הקומפיילר (בכדי שלא ייקח את ה-default של ה-Ubuntu)
  - **--host** : קובע שאנחנו מקמפלים עבור ARM
  - **--with-pcap** : קובע את סוג ה-packet capture. אנחנו רוצים לרוץ על מערכת Linux

- בשלב זה יתכן ותיתקלו בשגיאה הבאה:

```
configure: error: Your operating system's lex is insufficient to compile libpcap
```

- לכן התקינו flex ו-bison[3] והריצו שוב:

```
apt-get install flex
apt-get install bison
```

- הריצו make:

```
make
```

- כעת נעשה דבר דומה גם ל-tcpdump ונקמפל אותו סטטית (שימו לב ש-libpcap צריך להיות תיקייה אחת מאחורי tcpdump):

```
cd ../tcpdump
CFLAGS=--static CC=arm-buildroot-linux-uclibcgnueabi-gcc ./configure --
host=arm-linux
make
```

- נבדוק איזה קובץ יצא לנו:

```
file ./tcpdump
./tcpdump: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
```

- נבדוק את גודלו:

```
ls -alh ./tcpdump
-rwxr-xr-x 1 root root 2.2M Dec 13 23:02 ./tcpdump
```

- נוריד ממנו את הסימבולים כך שיהיה קטן יותר:

```
arm-buildroot-linux-uclibcgnueabi-strip ./tcpdump
```

- ונבדוק שוב את גודלו ואת הפרטים עליו:

```
ls -alh ./tcpdump
-rwxr-xr-x 1 root root 1.5M Dec 13 23:08 ./tcpdump
file ./tcpdump
./tcpdump: ELF 32-bit LSB  executable, ARM, EABI5 version 1 (SYSV),
statically linked, stripped
```

שימו לב שהגודל ירד מ-2.2 מגה ל-1.5 מגה וכתוב שהוא stripped. קיבלנו בינארי של tcpdump שניתן להריץ אותו על מעבד ARM!

[3] Flex ו-Bison - מנתחים לקסיקלים אשר דרושים עבור libpcap על מנת לפרסר את ה-BPF (Berkley Packet Filter)

כיצד ניתן לדעת מה פלטפורמת היעד?

ישנם מקרים בהם נרצה לקמפל למכונה ונרצה לדעת את הפרמטרים שלה עבור ה-toolchain (כפי שציינתי בהתחלה - סוג מע"ה / Endianness / מעבד וכו').

ישנן מספר דרכים להשיג את המידע הרלוונטי:

- הרצת הפקודה "uname -a" - לרוב תיתן לנו מספיק פרטים על המערכת עצמה אך פקודה זו לא תמיד קיימת על הרכיב
- הרצת הפקודה "file" על אחד הבינארים במכונה - כפי שראינו בדוגמאות, פקודה זו מספקת לנו כמעט את כל הפרטים הרלוונטיים. לא תמיד יש את file על המכונה אך ניתן להוריד את הבינארי ולהריץ עליו במכונה בה יש את file (למשל Ubuntu). גם הרצת readelf יכולה לתת מידע נוסף.
- קריאה מ-/proc/cpuinfo - להלן דוגמא ממכשיר אנדרואיד:

```
cat /proc/cpuinfo
Processor       : AArch64 Processor rev 4 (aarch64)
model name      : ARMv8 Processor rev 4 (v8l)
Features        : half thumb fastmult vfp edsp neon vfpv3 tls vfpv4
idiva
                  idivt lpae evtstrm aes pmull sha1 sha2 crc32
model name      : ARMv8 Processor rev 4 (v8l)
model name      : ARMv8 Processor rev 1 (v8l)
```

- וכמובן - תמיד אפשר לנסות לחפש מידע על הרכיב באינטרנט...

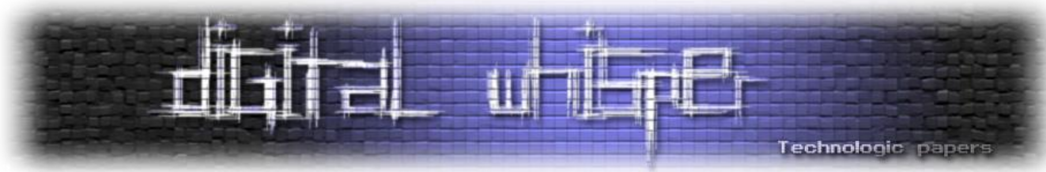
סיכום

במאמר הזה עברתי קצת על מושגים בסיסיים בקומפילציה והרחבתי על טכניקת ה-Cross Compiling אשר מאפשרת לקמפל בינאריים ממערכת אחת למערכת אחרת. כמו כן, הצגתי את הטכניקות השונות לעשות זאת והרחבתי בפירוט על אופן השימוש ב-Buildroot על מנת ליצור Toolchain שיתאים לפלטפורמה אליה אתם רוצים לקמפל.

המניע לכתוב את המאמר הנ"ל היה לפשט את נושא ה-Cross Compiling ולהנגיש אותו למי שלא התנסה בכך בעבר או למי שהרגיש שהוא לא מספיק יודע מה הוא עושה וע"י כך להפוך את הטכניקה ל"פחות מפחידה". המניע האישי עבורי, בתור מישהי שמתעסקת לא מעט ב-Cross Compiling, היה להיכנס יותר לעומק הנושא ולהבין כל שלב בתהליך.

יש לא מעט נושאים בתחום המחשבים והפיתוח שאנשים מתעסקים בהם ביום-יום ויודעים "לתפעל" אותם אך לא באמת מבינים מה עומד מאחוריהם וכיצד הם פועלים (מה שנקרא "נכנסים ל-Bits And Bytes"). לכן, המטרה שלי הייתה לעודד את קוראי המאמר להיכנס לעומק הנושא ולהבין יותר טוב איך המנגנון עובד ולתת את הכלים לעשות את כל העולה על רוחכם.

מקווה שהשגתי את המטרה ושחלקכם אף ניסו לעשות בעצמכם את הדברים שהדגמתי במאמר. תודה על הקריאה.



קישורים לקריאה נוספת

- https://buildroot.org/download.html
- https://elinux.org/Toolchains
- http://wiki.osdev.org/GCC_Cross-Compiler
- https://en.wikipedia.org/wiki/Source-to-source_compiler
- https://buildroot.org/
- https://en.wikipedia.org/wiki/ARM_architecture
- https://en.wikipedia.org/wiki/Recursive_acronym