אוניברסיטת אריאל בשומרון, המחלקה למדעי המחשב
הגנת פרוטוקולי תקשורת, סמסטר ב' תשפ"ג

Network Diode

ליאור וינמן יועד תמר

18 במאי 2023

תוכן עניינים

1 מבוא

בפרק זה נדבר על המבוא למטלה שכתבנו, לוגיסטיקת הקבצים, הוראת הרצה והסברים נוספים.

במטלה זו, מימשנו רכיב תקשורתי המדמה דיודת רשת עבור מידע העובר בתקשורת נכנסת. אפשר לחשוב על כך, כמו על דיודה פיזית (רכיב חשמלי) המחוברת בין שני מעגלים חשמליים. אופן הפעולה של הדיודה הפיזית הוא ההעברה של הזרם החשמלי רק בכיוון אחד ויחיד כך שזרם שנכנס לא יוכל לחזור אחורה, כך גם במטלה זו - ניתן לחשוב על כך בצורה הבאה, נניח אנו נמצאים בתוך ארגון (נניח, חברת סייבר כלשהי) וברצוננו שמידע רגיש מתוך הארגון לא ידלוף החוצה לעולם אבל אנחנו כן מעוניינים לקבל מידע המגיע מלקוחות שעובדים איתנו או מחברות אחרות שאנו בשיתוף פעולה עימן - כאן באה לידי ביטוי הדיודה, כדי לקדם את אבטחת המידע שלב אחד קדימה, נבנה רכיב תקשורתי המדמה דיודה כך שהיא תעביר את המידע "ממעגל חשמלי" הנמצא מחוץ לארגון לתוך ה"מעגל החשמלי" בתוך הארגון, ובכך שום מידע לא יצא החוצה אבל כן כל מידע שנרצה יכנס פנימה.
אפשר להרחיב את אופן העבודה של דיודת הרשת לרעיון של "חומת אש חד-כיוונית", כיוון שכל המידע הנכנס לתוך הארגון עובר אך ורק דרך הדיודה, זוהי עוד אפשרות לסנן מידע נכנס אם נרצה בכך (על מנת לשפר את אבטחת המידע בארגון). דיודת רשת תורמת לנו רבות בשמירה על דליפות המידע ולמעשה כיום כמעט לכל חברה יש דיודת רשת (אמנם אצלם זהו ממש רכיב פיזי ואנו מימשנו רכיב תקשורתי לא ממשי) אשר כל המידע עובר דרכה.

במטלה זו מימשנו ארבעה קבצים אשר מהווים - שולח, שני שרתי פרוקסי ומקבל. אופן פעולת הדיודה הוא שהשולח ישלח קובץ לשרת פרוקסי הראשון, משם הקובץ יעבור בפרוטוקול יחודי לשרת הפרוקסי השני אשר הוא יעביר אותו למקבל.

נציין בנוסף, שאנו בחרנו לכתוב את המטלה בשפת פייתון ולהריץ אותה בסביבת דוקרים.

1.1 קבצים

להגשה זו מצורפים 10 קבצים.

1. $Sender.py$ - קוובץ שבו נכתב השולח, אשר שולח את הקובץ.
2. $Proxy1.py$ - קובץ שבו נכתב שרת הפרוקסי הראשון, אשר אמור לקבל את הקובץ מהשולח ולהעביר לשרת הפרוקסי השני.
3. $Proxy2.py$ - קובץ שבו נכתב שרת הפרוקסי השני, אשר אמור לקבל משרת הפרוקסי הראשון את הקובץ ולהעביר למקבל.
4. $Receiver.py$ - קובץ שבו נכתב המקבל, אשר הוא הנקודה האחרונה בתקשורת ואמור לקבל משרת הפרוסקי השני את הקובץ.
5. $docker-compose.yml$ - קובץ אשר מרים את הקונטיינרים עבור הדוקר, בקובץ זה כתובות כל הגדרות הקונטיינרים שעליהם אנו נריץ את כל הקוד.
6. $1MB\_0\%Loss.pcapng$ - הקלטת Wireshark שבה אנו מתעדים את התעבורה דרך כל המערכת של קובץ במשקל של אחד מגהבייט וללא איבודי חבילות.
7. $1MB\_15\%Loss.pcapng$ - הקלטת Wireshark שבה אנו מתעדים את התעבורה דרך כל המערכת של קובץ במשקל של אחד מגהבייט עם 15 אחוזים של איבודי חבילות.
8. $213081763\_213451818.pdf$ - כמובן גם קובץ הסברים זה.
9. $vid.mp4$ - סרטון הרצה והסבר של המטלה.
10. $file$ - קובץ לדוגמה בגודל אחד מגהבייט שניתן לשלוח דרך הדיודה.

1.2 הוראות הרצה

את המטלה יש להריץ על על מערכת הפעלה Linux Ubuntu 22.04 LTS בלבד (זוהי מערכת ההפעלה שעליה אנו מימשנו, באופן תיאורטי המטלה יכולה גם לרוץ על סביבות אחרות). לפני ההרצה יש להוריד מפרש פייתון בגרסה 3, סביבת דוקרים וכלי לאיבוד חבילות. ניתן לבצ זאת על ידי:

```
SUDO APT-GET UPDATE
SUDO APT-GET INSTALL PYTHON3
PIP3 INSTALL TQDM
SUDO SNAP INSTALL DOCKER
SUDO APT INSTALL IPROUTE2
```

כעת, נרצה להרים את סביבת הדוקרים על המערכת. לשם כך, נרצה לפתוח טרמינל ולהריץ את הפקודה הבאה:

```
DOCKER-COMPOSE UP
```

הדבר, ירים לנו את כל הקונטיינרים של הדוקר הכתובים בקובץ ההגדרות המצורף (5).
נשים לב שבתיקה הנוכחית נוצרה תיקיה בשם volumes, זוהי תיקיה הנורצת לאחר הרמת הקונטיינרים ומהווה תיקיה משותפת בין כולם. נרצה להעביר לתיקיה זו את קבצי המטלה על ידי:

```
SUDO CP  ./*PY  ./VOLUMES/
```

כעת, נרצה ממש לפתוח את הקונטיינרים כמחשבים לעבוד עליהם, לכן, נפתח טרמינל ונריץ בו:

```
SUDO DOCKER PS
```

הדבר יתן לנו תמונת מצב על הקונטיינרים שהרמנו קודם לכן, כעת, נרצה לפתוח עוד שלושה טרמינלים (בסופו של דבר, כל טרמינל שפתחנו עד כה יהוו הקונטיינרים עצמם). בכל טרמינל שפתחנו עד כה נריץ:

```
SUDO DOCKER EXEC -IT <ID> /BIN/BASH
```

כאשר יש להחליף את <ID> במספר הנמצא בטבלת הפלט מהרצת הפקודה PS כעת, קיבלנו שיש ארבעה טרימינלים ובכל אחד מהם פתוח קונטיינר אשר מהווה מחשב במערכת. כעת, נרצה להסתכל בקובץ הגדרות הקונטיינרים על מנת לראות איזה מחשב מהווה איזה חלק בארכיטקטורה ולאחר שנוודא זאת, ניכנס לתיקיית הקבצים המשותפת ונריץ משם את קבצי הפייתון בהתאה:

```
CD  ./VOLUMES
PYTHON3 RECEIVER.PY
PYTHON3 PROXY2.PY
PYTHON3 PROXY1.PY
PYTHON3 SENDER.PY <FILE>
```

כאשר יש להריץ בסדר הנ"ל (מהמקבל לשולח) ויש להחליף את <FILE> בקובץ אותו אנחנו רוצים לשלוח. (ניתן להעביר קובץ לבחירתם לתקיית הקבצים המשותפת שראינו קודם לכן). אם נרצה להפעיל איבוד חבילות של X%, תחילה נבדוק על איזה כרטיס רשת אנחנו ולאחר מכן נשתמש בתוסף שהורדנו:

```
IFCONFIG
TC QDISC ADD DEV <IFACE> ROOT NETEM X%
TC QDISC DEL DEV <IFACE> ROOT NETEM
```

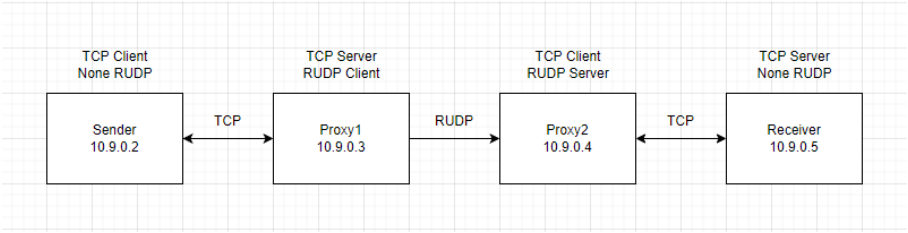
1.3 הסטוריית גרסאות

המטלה הועלתה גם לריפוסיטורי ב־Github, ניתן לראות שם היסטוריית commit־ים.

```
HTTPS://GITHUB.COM/LIORVI35/DATADIODE.GIT
```

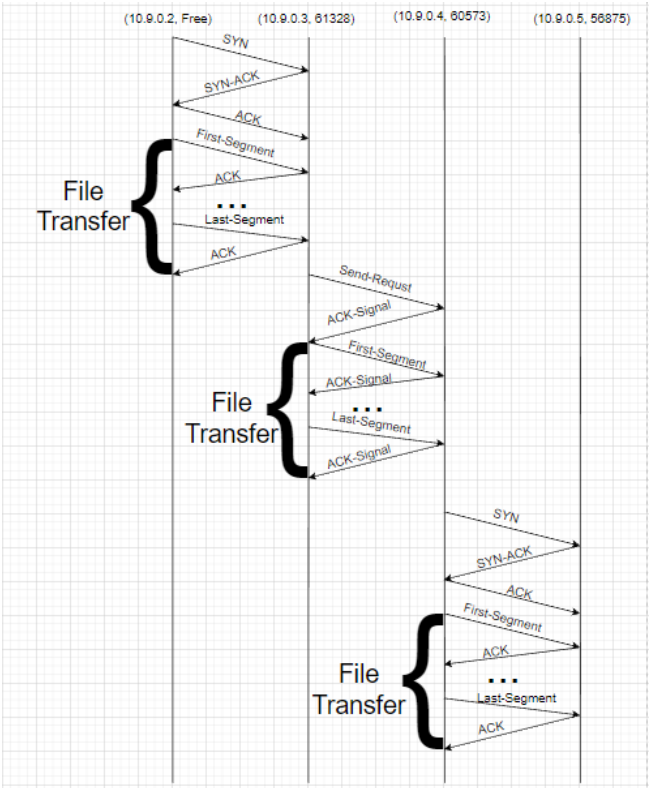
1.4 ארכיטקטורה

בעת מימוש המערכת, השתמשנו בשני פרוטוקולים עיקריים - הראשון TCP והשני, זהו פרוטוקול ייחודי אשר אנחנו בנינו והוא RUDP (פרוטוקול UDP הבנוי בצורה אמינה). בהמשך נסביר מדוע ישנו צורך בפרוטוקול הייחודי.
המערכת בנויה בצורה הבאה:



כאן ניתן לראות את ארבעת המכונות הבונות את המערכת, כפי שהזכרנו, המערכת לוקחת קובץ ושולחת אותו דרך הדיודה לנקודת הסף, אצלנו במימוש - השליחה של הקובץ מתבצעת בצורה הבאה - ישנו השולח שהוא לקוח TCP, בפרוטוקול זה הוא שולח את הקובץ לשרת הפרוקסי הראשון שהוא שרת TCP המאזין להתחברויות של לקוחות-שולחים ומקבל מהם את הקובץ, שרת הפרוקסי הראשון הוא גם לקוח בפרוטוקול RUDP וכאמור הוא שולח את הקובץ לשרת הפרוקסי השני אשר הוא שרת RUDP המאזין לשרת הפרוקסי הראשון לאחר קבלת הקובץ על ידו, שרת הפרוקסי השני כלקוח TCP שולח בפרוטוקול זה את הקובץ למשתמש הסף, לתוך הארגון כלומר למקבל וכך הקובץ הגיע לנקודת היעד, כאשר עבר דרך דיודת תקשורת.

נתבונן בזרימת החבילות במערכת:



כפי שתואר, תחילה הקובץ זורם מהשולח לפרוקסי הראשון, הדבר קורה בפרוטוקול TCP, כידוע פרוטוקול זה הוא פרוטוקול דו כיווני אשר דורש התחברות מלאה וקיום של SESSION בין שתי הנקודות. כעת, ישנו שרת הפרוקסי הראשון, הוא מקבל את הקובץ בפרוטוקול TCP. כעת נקודה חשובה, כל המימוש של ארכיטקטורת הדיודה במטלה בא לידי ביטוי בקשר ובתקשורת בין שני שרתי הפרוקסי. מטרת החיבור בין שני שרתי הפרוקסי (נציין כי במצב אמיתי, שניהם אמורים להיות מחוברים פיזית) היא לייצר את הדיודה ה-"רעיונית". זרימת מידע בצורה חד כיוונית אמיתית, ניתן לקבל רק בעזרת פרוטוקול UDP או מקביליו. בין שני שרתי הפרוקסי ישנה תקשורת בפרוטוקול RUDP שזהו כמובן, גם גם פרטוקול דו כיווני בהסתכלות ראשונית. אמנם, אם נתבונן פנימה לתוך הפרוטוקול

נשים לב כי **המידע** אכן זורם רק בכיוון אחד, וכי גם בתרשים מוצג כי הסגמנטים של הקובץ נעים מפרוקסי אחד לפרוקסי השני בלבד ולא הפוך, הדבר היחיד שחוזר אחורה הוא **סיגנלים אודות קבלת המידע** - זאת כיוון שנכון שאנחנו רוצים שמידע רק יכנס לארגון ולא יצא ממנו (וזה באמת מה שקורה!) אבל גם חשוב לנו שהארגון יקבל את כל המידע בשלמותו (כי חצי מידע, כלל לא עוזר). לבסוף, שרת הפרוקסי השני שולח למקבל את הקובץ וכך המידע נכנס לארגון.

כעת, נרצה להסביר מדוע באמת הזרימה פה מדמה דיודה - נשים לב כי בכל פעם שאנחנו מתקדמים לתחנה הבאה אנחנו לא חוזרים אחורה והמידע, הקובץ, תמיד זורם רק בכיוון אחד ויחיד (ימינה - בתרשים), בין כל שתי תחנות, המידע שחוזר שמאלה הוא "אישורי קריאה" בלבד, זאת מפני שחשוב לנו מאוד לשמור על אמינות לאורך כל הדרך ולכן עלינו לדאוג שהמידע נשלח בצורה אמינה, כעת, ברור כי TCP מהגדרתו תומך בכך בצורה אוטומטית וכך הוא בנוי (שולח חבילות עם דגל ACKNOWLEDGEMENT), לגבי הפרוטוקול שאנחנו בנינו, אנחנו שולחים חבילה עם מידע מאוד מסויים אשר אומר לשולח שהמידע שנשלח התקבל. על כן, הקובץ עצמו זורם תמיד רק לצד ימין ומה שחוזר לצד שמאל אלו אישורי קבלת המידע.

לבסוף, ההסבר המובטח, אם RUDP מנסה לדמות את פרוטוקול TCP מדוע אנחנו צריכים פרוטוקול יעודי כזה ולמה לא להשתמש פשוט ב־TCP? התשובה - כדי להבטיח את זרימת המידע רק בכיוון אחד עלינו לוודא שלא ניתן לבצע עלינו מניפולציות, כלומר, עליו לוודא בכל תוקף, שאם מצליחים להשתלט על התחנה של הפרוקסי הראשון, אין מצליחים להוציא מידע שכבר יש בתוך הארגון - הפתרון לכך, הוא יצירת פרוטוקול ייחודי חדש (ממש כמו יצירת שפה חדשה) אשר כיוון שזהו פרוטוקול ייעודי שאנחנו בנינו, כנראה שאף אחד לא יודע איך להשתמש בו (בהנחה שלא הפצנו אותו, וכמובן לא הפצנו כי על זה מושתת הארגון שלנו) לכן אם קיימת כאן תקשורת אשר אף אחד מלבד הארגון יודע איך להשתמש בו, כנראה שגם הצלחנו להוריד את סיכויי הנזק שיכולים להגרם לנו למינימום האפשרי, שכן, נניח שאיבדנו את הפרוקסי הראשון מי שהשתלט עליו, לא יודע איך להמשיך הלאה כלומר גם באיבוד תחנה, אנחנו מצליחים מלשמור על המידע מלזלוג לכיוון השני. זוהי בדיוק תפקידה של הדיודה.

מבחינת המימוש, הדיודה כאן היא "תפיסה" שאנו מנסים לבטא ונציין כי לא קיים רכיב במערכת הנקרא "דיודה" הדיודה היא עקרון שאנחנו רוצים להשיג בתקשורת בין שני שרתי הפרוקסי.

2 אלגוריתמיקה

בפרק זה נעבור בצורה יסודית על קבצי המימושים, נסביר את האלגוריתמיקה מאחורי המימוש.

את כל אחת מתחנות התעבורה כתבנו בשפת Python ראשית הסבר קצר מדוע בחרנו דווקא בשפה הזו - נגיד כי לכל משימה יש את הכלי הכי נכון בשביל לגשת ולפתור אותה. שפת Python היא שפה מאוד קלה ונוחה וכי מה שניתן לבצע בלפחות 10 שורות בשפת C ניתן לבצע בשלוש שורות ב־Python (כדומה פתיחת סוקט), זאת בנוסף לפשטות הכתיבה.

2.1 השולח

כאן נעבור על הקוד של השולח, הנקודה הראשונה בזרימת המידע.

כאן הגדרנו את אזור הקבועים, הגדרנו את הכתובת של שרת הפרוקסי הראשון, שכן השולח אמור לשלוח אליו את הקובץ, הגדרנו את גודל הבאפר - אנחנו החלטנו להשתמש בגודל הבאפר הדיפולטיבי של פייתון שהוא 8KB = 8192MB וכמובן, לצורך קריאות הקוד, הגדרנו שני קבועים המסמנים את אופי סיום התוכנית - 0 התוכנית הסתיימה בהצלחה 1 - קרתה תקלה בהרצה.

```
FIRST_PROXY_ADDR = ("10.9.0.3", 61328)  # address of proxy1
BUFFER_SIZE = io.DEFAULT_BUFFER_SIZE  # buffer size for send/receive
SUCCESS = 0  # ok
FAIL = 1  # !ok
```

בשלב הראשון של זרימת המידע, על השולח לבחור קובץ ולבצע לו גיבוב לפורמט MD5, על מנת לבצע את הגיבוב הזה כתבנו את הפונקציה הנ"ל אשר מקבלת קובץ (נציין שזהו אינו מיקום של קובץ, אלא אובייקט קובץ שהוא כבר פתוח במצב של קריאה־בינארית) קוראת מהקובץ ומבצעת גיבוב לפורמט הנדרש. לבסוף אנחנו מחזירים את המצביע של הקובץ לתחילתו (כדי שנוכל לשלוח אותו החל מההתחלה) ולבסוף מחזירים ייצוג הקסאדצימלי של הגיבוב.

מילה נוספת לגבי הגיבוב עצמו, פורמט MD5 הוא גיבוב אשר נותן לכל טקסט קיים ייצוג חד חד ערכי באורך של 128 ביטים. על כן, אם אנחנו מבצעים גיבוב של הקובץ לפני השליחה ולבסוף גם מבצעים את הגיבוב והתוצאות זהות לגמריי, אנחנו יכולים להיות בטוחים שהקובץ הגיע ליעד בשלמותו (בגלל חד חד ערכיות), על כן, זוהי שיטה נוחה מאוד לאימות הגעת הקובץ ליעד בצורה אמינה.

```
def hash_file(file):
    """
    this function hashes a file into MD5
    :param file: object of file
    :return: hexadecimal representation of the hash
    """
    try:
        md5 = hashlib.md5()
        data = file.read(io.DEFAULT_BUFFER_SIZE)
        while data:
            md5.update(data)
            data = file.read(io.DEFAULT_BUFFER_SIZE)
        file.seek(0)
        return md5.hexdigest()
    except Exception as e:
        print(f"Error: {e}.")
        sys.exit(FAIL)
```

כאן זוהי הפונקציה הראשית של השולח. אופן השימוש בשולח, כפי שכבר ציינו, הוא:
PYTHON3 SENDERPY <FILE> כפי שניתן לראות, הרצת השולח דורשת מתן של ארגומנט כבר בשורה בטרמינל, הסיבה לכך היא שלא רצינו להגביל את השולח לקובץ קבוע ועל כן רצינו שהמשתמש יבחר איזה קובץ לשלוח.
בשלב הראשון, השולח בודק שאכן קיבל ארגומנט (הסיבה שאנו בודקים שמספר הארגומנטים שווה ל־2 היא כי לכל תוכנית פייתון תמיד יש ארגומנט ראשון והוא שם התוכנית - מועבר למפרש של השפה), אם הקובץ לא קיים תודפס הודאה בהתאם. לאחר מכן, אנו פותחים את הקובץ במצב של קריאה בינארית (קריאת ביטים ולא תווי ASCII) ומדפיסים את הגיבוב. לאחר מכן, פותחים סוקט בפרוטוקול TCP ומתחברים לשרת הפרוקסי הראשון. פותחים סרגל התקדמות ומתחילים לשלוח את הקובץ לבסוף, אחרי השליחה - ננתק את השולח (SHUTDOWN - ניתוק חד כיווני) מהשרת ולאחר מכן נסגור את הסוקט, כמובן שאם קוראת שגיאה נצא באמצע.

```python
def main():
    """
    main function, firstly it opens the file and hashes into MD5,
    then opens a connection with first proxy server and sends him the file
    """
    if len(sys.argv) != 2:
        print("Usage: 'python3 Sender.py <file>'.")
        sys.exit(FAIL)

    try:
        with open(sys.argv[1], "rb") as file:
            print(f"MD5 = '{hash_file(file)}'.")

            try:
                with socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP) as main_sock:
                    main_sock.connect(FIRST_PROXY_ADDR)
                    print(f"Established connection with: '{FIRST_PROXY_ADDR}'.")

                    with tqdm.tqdm(total=os.path.getsize(sys.argv[1]), unit="B", unit_scale=True, desc="Sending") as pb:
                        data = file.read(BUFFER_SIZE)
                        while data:
                            main_sock.sendall(data)
                            pb.update(len(data))
                            data = file.read(BUFFER_SIZE)
                    print("File has been sent.")
                    main_sock.shutdown(socket.SHUT_RDWR)
            except KeyboardInterrupt:
                print("\nClosing sender.")
            except (Exception, socket.error) as e:
                print(f"Error: {e}.")
            sys.exit(FAIL)
    except FileNotFoundError:
        print("File does not exists")
        sys.exit(FAIL)
```

כמובן לבסוף (התנאי האהוב עלינו) ישנו התנאי המגדיר את אופן ריצת התוכנית ומוודא כי התוכנית תרוץ ממש רק כאשר רצה במפרש ולא אם היא מיובאת כספריה לתוכנית אחרת. לבסוף אם התוכנית הראשית התבצעה בהצלחה, נצא עם קוד חיובי.

```python
if __name__ == "__main__":
    main()
    sys.exit(SUCCESS)
```

2.2 פרוקסי ראשון

כאן נעבור על הקוד של שרת הפרוקסי הרשון, הנקודה השניה בזרימת המידע.

כאן הגדרנו את אזור הקבועים, הגדרנו את הכתובת של שרת הפרוקסי הבא, זאת כי שרת הפרוקסי הנ"ל צריך להעביר אליו את הקובץ, הגדרנו את הכתובת של השרת הנוכחי, הגדרנו את גודל הבאפר - הדיפולטיבי פי שתיים כדי לייעל את התהליך, הגדרנו את השם הזמני של הקובץ שכן בשרת הנוכחי מקבל את הקובץ ועליו גם לשמור אותו ואז לשלוח, הגדרנו את המבנה של בקשת התחברות בפרוטוקול שבנינו - RUDP, הגדרנו זמן מקסימלי לקבלת חבילות בפרטוקול שבנינו, הגדרנו קוד הצלחה כ-0, הגדרנו מבנה של סיגנל אישור קבלה בפרוטוקול שבנינו, הגדרנו קוד כשלון כ-1, הגדרנו את המפצל של פיצול המידע בחבילות RUDP, הגדרנו הודעה ריקה, הגדרנו את מספר החיבורים המקסימליים בפרוטוקול TCP.

```
SECOND_PROXY_ADDR = ("10.9.0.4", 60573)  # address of proxy2
FIRST_PROXY_ADDR = ("10.9.0.3", 61328)  # address of proxy1
BUFFER_SIZE = 2 * io.DEFAULT_BUFFER_SIZE   # buffer size for send/receive
TEMP_FILE_NAME = "p1"  # temporary file name, saved as: p1_addr
SEND_REQ = b"N1:1:S"  # request to send
TIMEOUT = 0.15  # packet receive timeout
SUCCESS = 0  # ok
ACK = b"A"  # RUDP's acknowledgment signal
FAIL = 1  # !ok
SPLITTER = b":"  # payload info splitter
EMPTY_MESSAGE = b""  # empty data
MAX_CONNECTIONS = 300  # maximal TCP clients
```

פרוטוקול RUDP הוא למעשה שפה חדשה בין שרתי הפרוקסי, הגדרת המילים בשפה היא למעשה הפונקציה הנ"ל. כל חבילה הנשלחת ב-RUDP, תכיל מידע בצורה הנ"ל - תוו N או תוו F, התוו מסמל האם זהו הסגמנט האחרון או שיש אחריו עוד סגמנטים N מסמל NotFinal וכמובן F מסמל Final המספר שמגיע אחרי התו הוא מספר הרצף - מספר הסגמנט שנשלח. לאחריו יש נקודותיים (ניזכר במפצל שהגדרנו) אחרי הנקודותיים מגיע אורך המידע שאנו שולחים - חשוב לשמור את האורך המקורי שכן חבילה יכולה להגיע פגומה מהסוף ואז המידע נפגע ולבסוף לאחרי הנקודותיים הנוספות מגיע המידע עצמו, הסגמנט עצמו של הקובץ שאותו אנחנו עכשיו שולחים.

```
def build_payload(final, seq, data):
    """
    this function builds the RUDP packet's payload
    :param final: True if this is final segment, else false
    :param seq: segment's relative sequence number
    :param data: the segment itself
    :return: encoded string in format: <N/F><Seq>:<length(Data)>:<Data>
    """
    ch = "F" if final else "N"
    return f"{ch}{seq}:{len(data)}:{data.decode()}".encode()
```

כאן כתבנו פונקציה אשר מאמתת קבלה - פרוטוקול RUDP מושתת כולו על פרוטוקול UDP שזהו פרוטוקול שאינו אמין. אחת הדרכים לאמת קבלה, היא טיימר. נניח ששלחנו חבילה, אנו מצפים לקבל אישור קבלה תוך זמן מסויים (לרוב זמן שהוא קצת גבוה מהזמן האמיתי שעל החבילה להגיע) אם לא קיבלנו, ההנחה היא שהחבילה אבדה ועלינו לשלוח שוב, כך עד שהמידע יתקבל וככל שמצב הרשת פגום יותר ככה נשלח יותר חבילות עד שנקבל אישור קבלה.

כאן ישנה נקודה לציין, כי פרוטוקול חד כיווני אינו יכול להיות אמין כלל. פרוטוקול חד כיווני פירושו לשלוח את הסגמנטים ברצף ללא מעצור וללא ווידוא קבלה, על כן, אם ישנו איבוד חבילות אין הבטחה שכל הסגמנטים ששלחו יגיעו ליעד - או יגיעו לפי סדר ההגעה הנכון על כן, ישנו צורך באישורי קבלה על מנת להתגבר על המכשול. על כן, המסקנה עד כה היא שאין תקשורת חד כיוונית ואמינה. בנוסף לכך, נרצה לציין כי כאשר אנו כותבים פה "תקשורת חד כיוונית" הכוונה היא באופן כללי כל זרימה החבילות ולא רק זרימה המידע. אצלנו במטלה ובמימוש הדיודה, הזרימה היא חד כיוונית מבחינת המידע, אמנם ישנם אישורי קבלה שחוזרים אחורה אך המידע זורם רק בכיוון אחד ויחיד וישנו טיפול בכך שהוא לא יחזור אחורה.

```python
def check_recv(sock, packet):
    """
    this file checks if RUDP packet has been received or not
    :param sock: object of UDP socket to send data via
    :param packet: packet to send
    :return: (True,msg) if packet has been sent (data is what received), else (False,None)
    """
    sock.settimeout(TIMEOUT)
    while True:
        try:
            msg, address = sock.recvfrom(BUFFER_SIZE)
            sock.settimeout(None)
            return True, msg
        except (socket.timeout, socket.error):
            sock.sendto(packet, SECOND_PROXY_ADDR)
            continue
    sock.settimeout(None)
    return False, None
```

כאן ישנה פונקציה אשר שולחת את הקובץ, הלאה משרת הפרוקסי הראשון לשרת הפרוקסי השני. תחילה אנחנו פותחים סוקט בפרוטוקול UDP ושולחים לשרת הפרוקסי השני, בקשת התחברות לאחר מכן, אם ההתחברות הצליחה, אנו פותחים את הקובץ שברצוננו לשלוח במצב קריאה־בינארית ופותחים סרגל התקדמות, מגדירים את מספר הרצף להיות 2 (שכן כבר שלחנו חבילה אחת שהיא הבקשה) ומתחילים לקרוא מהקובץ, כמובן רצים על כל הקובץ קוראים ושולחים עד שאין מה לקרוא יותר. כאן ישנו מימוש של מנגנון RETRANSMISSION, לאחר כל חבילה שהגיעה השרת שולח אישור קבלה עם מספר הרצף האחרון שקיבל. אם הגיעה חבילה עם מספר רצף שאינו מצופה (יכול לקרות במצב איבוד חבילות) השרת ישלח אישור קבלה על החבילה האחרונה שקיבל - הלקוח תמיד מצפה לאישורי קבלה ולכן רק יבדוק אם מספר הרצף שהגיע אישור לגביו זהה לגבי מה שנשלח, אם לא ההנחה היא שהחבילה אבדה ועל כן יש לשלוח אותה שוב, לכן נזיז אחורה את המצביע על הקובץ ונשלח את הסגמנט הקודם שאבד. כמובן, לאחר כל שליחה נעדכן את סרגל ההתקדמות.

```python
def send_file(addr):
    """
    this function sends RUDP packets
    :param addr: client address to send to
    """
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP) as flow_sock:
        flow_sock.sendto(SEND_REQ, SECOND_PROXY_ADDR)
        status, data = check_recv(flow_sock, SEND_REQ)
        if not status:
            sys.exit(FAIL)
        with open(f"{TEMP_FILE_NAME}_{addr}", "rb") as file:
            with tqdm.tqdm(total=os.path.getsize(f"{TEMP_FILE_NAME}_{addr}"),
                           unit="B", unit_scale=True, desc="Sending") as pb:
                seq = 2
                data = file.read(BUFFER_SIZE)
                while data:
                    flow_sock.sendto(build_payload(False, seq, data), SECOND_PROXY_ADDR)
                    status, to_send = check_recv(flow_sock, build_payload(False, seq, data))
                    if not status:
                        sys.exit(FAIL)
                    if int(to_send.split(SPLITTER)[1].decode()) != seq:
                        ack = int(to_send.split(SPLITTER)[1].decode())
                        file.seek(0)
                        file.read(ack)
                        seq = ack + 1
                        data = file.read(BUFFER_SIZE)
                        continue
                    pb.update(len(data))
                    data = file.read(BUFFER_SIZE)
                    seq += 1
                flow_sock.sendto(build_payload(True, seq, EMPTY_MESSAGE), SECOND_PROXY_ADDR)
                status, data = check_recv(flow_sock, build_payload(True, seq, EMPTY_MESSAGE))
                if data != ACK:
                    sys.exit(FAIL)
```

כאן זוהי הפונקציה הראשית, כיוון ששרת הפרוקסי הראשון משמש כשרת TCP, נפתח סוקט כזה נקבע אותו על הכתובת שציינו ונאזין להתחברויות נכנסות, לאחר שקיבלנו התחברות ניצור קובץ חדש שאליו נשמור את הקובץ ששולחים לנו - קובץ זה הוא זמני בלבד וימחק לאחר ההעברה לפרוקסי הבא, מקבלים את הקובץ ב־TCP, מנתקים את הלקוח, קוראים לפונקציה אשר שולחת את הקובץ ולבסוף מוחקים את הקובץ הזמני.

```python
def main():
    """
    main function, it creates a TCP socket which receives the file from the Sender
    then in opens a (R)UDP socket that sends the file to seconds proxy
    :return:
    """
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP) as main_sock:
            main_sock.bind(FIRST_PROXY_ADDR)
            main_sock.listen(MAX_CONNECTIONS)
            print("Listening for incoming connections...\n")

            while True:
                client_sock, client_addr = main_sock.accept()
                print(f"Accepted connection from: '{client_addr}'.")
                with open(f"{TEMP_FILE_NAME}_{client_addr}", "wb") as file:
                    with tqdm.tqdm(total=None, unit="B", unit_scale=True, desc="Receiving") as pb:
                        data = client_sock.recv(BUFFER_SIZE)
                        while data:
                            file.write(data)
                            pb.update(len(data))
                            data = client_sock.recv(BUFFER_SIZE)
                    print("File has been received.")
                client_sock.close()

                send_file(client_addr)
                print("File has been sent.\n")

                os.remove(f"{TEMP_FILE_NAME}_{client_addr}")
    except KeyboardInterrupt:
        print("\nClosing proxy1.")
        sys.exit(SUCCESS)
    except (Exception, socket.error) as e:
        print(f"Error: {e}.")
        sys.exit(FAIL)
```

כמובן כאן שוב תנאי ההרצה, כאן הוא ללא יציאה עם קוד 0 לאחר הפונקציה הראשית, כיוון שהפרוקסי הראשון הוא שרת לכן הוא תמיד מאזין (WHILE TRUE), כלומר הדרך היחידה לעצור אותו היא KEYBOARDINTERRUPT - אבל שם אנחנו יוצאים עם קוד 0, לכן הכל תקין.

```python
if __name__ == "__main__":
    main()
```

2.3 פרוקסי שני

כאן נעבור על הקוד של שרת הפרוקסי השני, הנקודה השלישית בזרימת המידע.

כאן הגדרנו את אזור הקבועים - הגדרנו את הכתובת של הפרוקסי השני והראשון, כמובן גם הגדרנו את כתובת המקבל, הגדרנו את גודל הבאפר - הגודל הדיפולטיבי שפייתון מציעה אך עם תוספת קנס על בניית החבילה כפי שבנינו אותה קודם לכן בעזרת הנקודותיים והתו והמספר על מנת לקבל את הגודל המדויק של התוספת, החלטנו להשתמש בפונקציית המערכת - שכן תוספת קבועה לא בהכרח נכונה וכי הגדלים משתנים בין מעבדים וארכיטקטורות, לאחר מכן הגדרנו את שם הקובץ הזמני, הגדרנו הודעת סיגנל של אישור קבלה, הגדרנו בקשת שליחה ב־RUDP, הגדרנו זמן המתנה לחבילה, הגדרנו מזהה של החבילה האחרונה, הגדרנו את המפגל של החבילה, והגדרנו קודם של הצלחה - 0 וכישלון - 1.

```
SECOND_PROXY_ADDR = ("10.9.0.4", 60573)  # address of proxy2
FIRST_PROXY_ADDR = ("10.9.0.3", 61328)  # address of proxy1
RECEIVER_ADDR = ("10.9.0.5", 56875)  # address of receiver
BUFFER_SIZE = (2 * io.DEFAULT_BUFFER_SIZE) + (2 * sys.getsizeof(b":")) + (2 * sys.getsizeof(int)) \
              + max(sys.getsizeof(b"A"), sys.getsizeof(b"N"), sys.getsizeof(b"F"))  # buffer size for send/receive
TEMP_FILE_NAME = "p2"  # temporary file name, saved as: p2_addr
ACK = b"A"  # RUDP's acknowledgment signal
SEND_REQ = b"N1:1:S"  # request to send
TIMEOUT = 1
LAST_PACKET = b"F"  # last packet identifier
SPLITTER = b":"  # payload info splitter
FAIL = 1  # !ok
SUCCESS = 0  # ok
```

כאן הגדרנו פונקציה אשר שולחת אישורי קבלה על סגמנטים שהתקבלנו בהתאם למספר הרצף. נציין שוב כי זהו הדבר היחיד שחוזר אחורה (וזהו תנאי הכרחי לצורך קיום תקשורת אמינה).

```
def send_acknowledgement(sock, client_addr, seq):
    """
    this function sends an ack signal about received segments
    :param sock: object of (R)UDP socket
    :param client_addr: client address to send to
    :param seq: None if should be sent a "common" signal, or a last segment sequence number
    """
    if seq is None:
        sock.sendto(ACK, client_addr)
    else:
        sock.sendto(f"{ACK.decode()}:{seq}".encode(), client_addr)
```

כאן ישנה פונקציה אשר שולחת את הקובץ הלאה - למקבל. כאן אנחנו פותחים שקע בפרוטוקול TCP שכן בו המקבל אמור לקבל את הקובץ הסופי, פותחים את הקובץ לקריאה־בינארית, פותחים סרגל התקדמות ומתחילים לקרוא ולשלוח, לבסוף מתנתקים מהמקבל שהוא מהווה השרת.

```
def send_file(addr):
    """
    this function opens a TCP socket and sends through it a file
    :param addr: client's address
    """
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP) as sock:
            sock.connect(RECEIVER_ADDR)
            with open(f"{TEMP_FILE_NAME}_{addr}", "rb") as file:
                with tqdm.tqdm(total=os.path.getsize(f"{TEMP_FILE_NAME}_{addr}"),
                               unit="B", unit_scale=True, desc="Sending") as pb:
                    data = file.read(BUFFER_SIZE)
                    while data:
                        sock.sendall(data)
                        pb.update(len(data))
                        data = file.read(BUFFER_SIZE)
            sock.shutdown(socket.SHUT_RDWR)
        print("File has been sent.\n")
    except (Exception, socket.error) as e:
        print("Error: {e}.")
        sys.exit(FAIL)
```

כאן מוגדרת הפונקציה הראשית, ראשית אנחנו פותחים סוקט UDP ומקבעים אותו. מתחילים להאזין לתקשורת נכנסת, אם הגיעה החבילה האחרונה, נשלח אישור קבלה ונמשיך, אם לא, נמתין לבקשת שליחה - נקבל חבילת RUDP - נבדוק אם גודל החבילה זהה לגודל המידע שהגיע בפועל, אם לא נמשיך לחבילה הבאה, לאחר מכן נפתח קובץ לכתיבה של הסגמנטים ונפתח סרגל התקדמות, נאתחל את מספר הרצף שאנו שולחים ונאתל את מספר הרצף שאנו בפועל מצפים לקבל מהמשתמש. כעת ישנם מספר מנגנוני אמינות, אחנחו בודקים האם גודל המידע שהתקבל שווה ממש לגודל שחושב לפני כן, אם לא נשלח הודעת קבלה על הסגמט הקודם, בנוסף נבדוק האם מספר הרצף שקיבלנו הוא מספר הרצף שאנו מצפים לקבל, אם לא, נפעל כנ"ל. לבסוף אם הכל תקין, נעדכן את מספרי הרצף, נעדכן את סרגל ההתקדמות ונכתוב לקובץ, לבסוף נשלח את הקובץ בעזרת הפונקציה הקודמת. אחרי ששלחנו נמחק את הקובץ הזמני.

```python
def main():
    """
    main function, opens a (R)UDP socket and receiving the file,
    then opens a TCP socket and sends the file to receiver
    """
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP) as sock:
        sock.bind(SECOND_PROXY_ADDR)
        print("Listening for incoming connections...\n")
        while True:
            data, client_addr = sock.recvfrom(BUFFER_SIZE)
            if data[0:1] == LAST_PACKET:
                send_acknowledgement(sock, client_addr, None)
                continue
            if data != SEND_REQ:
                continue
            send_acknowledgement(sock, client_addr, 1)
            print(f"Accepted connection from: '{client_addr}'.")
            data, client_addr = sock.recvfrom(BUFFER_SIZE)
            if int(data[1:2]) != 2:
                continue
            with open(f"{TEMP_FILE_NAME}_{client_addr}", "wb") as file:
                with tqdm.tqdm(total=None, unit="B", unit_scale=True, desc="Receiving") as pb:
                    seq = 1
                    expected_seq = 2
                    while data[0:1] != LAST_PACKET:
                        if int(data.split(SPLITTER)[1]) != len(data[(data.find(SPLITTER, data.find(SPLITTER) + 1)) + 1:]):
                            send_acknowledgement(sock, client_addr, seq)
                            data, client_addr = sock.recvfrom(BUFFER_SIZE)
                            continue
                        elif int(data.split(SPLITTER)[0][1:]) != expected_seq:
                            send_acknowledgement(sock, client_addr, seq)
                            data, client_addr = sock.recvfrom(BUFFER_SIZE)
                            continue
                        seq = expected_seq
                        expected_seq += 1
                        pb.update(len(data[(data.find(SPLITTER, data.find(SPLITTER) + 1)) + 1:]))
                        file.write(data[(data.find(SPLITTER, data.find(SPLITTER) + 1)) + 1:])
                        send_acknowledgement(sock, client_addr, seq)
                        data, client_addr = sock.recvfrom(BUFFER_SIZE)
                    send_acknowledgement(sock, client_addr, None)
            print("File has been received.")
            send_file(client_addr)
            os.remove(f"{TEMP_FILE_NAME}_{client_addr}")
```

כאן שוב התנאי, הגישה לתפיסת השגיאות כאן הייתה היא לנסות להריץ את הפונקציה הראשית, אם שגיאה נזרקת היא תעלה למעלה וכאן נתפוס אותה. לבסוף נחזיר 1 או 0 בהתאם.

```python
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("\nClosing proxy2...")
    except (Exception, socket.error) as e:
        print(f"Error: {e}.")
        sys.exit(FAIL)
    sys.exit(SUCCESS)
```

2.4 המקבל

כאן נעבור על הקוד של המקבל, הנקודה האחרונה בזרימה המידע.

כאן הגדרנו את אזור הקבועים, הגדרנו קודיי הצלחה וכישלון התוכנית, הגדרנו את מספר החיבורים המקסימליים ב־TCP, הגדרנו את הכתובת של המקבל והגדרנו את שם הקובץ אותו נשמור בסוף התוכנית.

```
FAIL = 1  # !ok
SUCCESS = 0  # ok
MAX_CONNECTIONS = 300  # maximal TCP clients
RECEIVER_ADDR = ("10.9.0.5", 56875)  # address of receiver
FILE_NAME = "recv"  # name for received file, saved as: recv_addr
```

כאן הגדרנו את הפונקציה המגבבת את הקובץ לפורמט MD5, ראינו כבר את הפונקציה הזו - זוהי אותה פונקציה בדיוק כמו שיש לשולח וזוהי נקודה עקרונית כאן - עצם השימוש באותה פונקציה בדיוק מבטיח דיוקים בגיבוב וכן הסיכוי לקבל אותו פלט (כי החישובים נעשים מאחורי הקלעים עבור המשתמש) בעבור קלטים שונים מתאפס.

```
def hash_file(file):
    """
    this function hashes a file into MD5
    same function that Sender has
    :param file: object of file
    :return: hexadecimal representation of the hash
    """
    try:
        md5 = hashlib.md5()
        data = file.read(io.DEFAULT_BUFFER_SIZE)
        while data:
            md5.update(data)
            data = file.read(io.DEFAULT_BUFFER_SIZE)
        file.seek(0)
        return md5.hexdigest()
    except Exception as e:
        print(f"Error: {e}.")
        sys.exit(FAIL)
```

כאן זוהי הפונקציה הראשית של המקבל, אנחנו פותחים סוקט של TCP ומאזינים לבקשות נכנסות, לאחר מכן מאשרים ומתחילים לקבל את הקובץ. לבסוף נגבב את הקובץ לפורמט הרצוי לשם בדיקת תקינות ושלמות המידע שעבר במערכת. נציין כי כאן אנו לא מוחקים את הקובץ שקיבלנו, בשונה משאר התחנות כיוון שכאן זוהי התחנה הסופית וברצוננו שהקובל שהגיע ישאר בתוך הארגון (הריי לשם זה בנינו את הדיודה) לכן הקובץ פשוט נשמר.

```
def main():
    """
    main function, it opens a TCP socket and receives the file from Proxy2
    then hashes the file into MD5
    :return:
    """
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM, socket.IPPROTO_TCP) as sock:
        sock.bind(RECEIVER_ADDR)
        sock.listen(MAX_CONNECTIONS)
        print("Listening for incoming connections...\n")
        while True:
            client_sock, client_addr = sock.accept()
            print(f"Accepted connection from: '{client_addr}'.")

            with open(f"{FILE_NAME}_{client_addr}", "wb") as file:
                with tqdm.tqdm(total=None, unit="B", unit_scale=True, desc="Receiving") as pb:
                    data = client_sock.recv(io.DEFAULT_BUFFER_SIZE)
                    while data:
                        file.write(data)
                        pb.update(len(data))
                        data = client_sock.recv(io.DEFAULT_BUFFER_SIZE)

                client_sock.close()

            with open(f"{FILE_NAME}_{client_addr}", "rb") as file:
                print(f"File has been received.\nMD5 = '{hash_file(file)}'.")
```

כאן, לפי אותה גישה כמו בפרוקסי השני, אנחנו מנטרים את השגיאות ומחזירים קוד סיום בהתאם למצב.

```
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("\nClosing proxy2...")
    except (Exception, socket.error) as e:
        print(f"Error: {e}.")
        sys.exit(FAIL)
    sys.exit(SUCCESS)
```

2.5 הרכבת הקונטיינרים

אמנם אין חלק זה קשור באלגוריתמיקה עצמה של המימוש, אך כיוון שהפתרון חייב לרוץ על סביבת דוקרים (וספיציפית אך ורק הקונטיינרים המצורפים) נעבור גם על קובץ זה כאן.

כאן אנו מרכיבים את הקונטיינר של השולח, כתובתו היא 10.9.0.2.

```
Sender:
    image: handsonsecurity/seed-ubuntu:large
    container_name: Sender-10.9.0.2
    tty: true
    cap_add:
            - ALL
    networks:
        net-10.9.0.0:
            ipv4_address: 10.9.0.2
    volumes:
            - ./volumes:/volumes
    command: bash -c "
                /etc/init.d/openbsd-inetd start  &&
                tail -f /dev/null
             "
```

כאן אנו מרכיבים את הקונטיינר של שרת הפרוקסי הראשון, כתובתו היא 10.9.0.3.

```
Sender:
    image: handsonsecurity/seed-ubuntu:large
    container_name: Sender-10.9.0.2
    tty: true
    cap_add:
            - ALL
    networks:
        net-10.9.0.0:
            ipv4_address: 10.9.0.2
    volumes:
            - ./volumes:/volumes
    command: bash -c "
                /etc/init.d/openbsd-inetd start  &&
                tail -f /dev/null
             "
```

כאן אנו מרכיבים את הקונטיינר של שרת הפרוקסי השני, כתובתו היא 10.9.0.4.

```
Sender:
    image: handsonsecurity/seed-ubuntu:large
    container_name: Sender-10.9.0.2
    tty: true
    cap_add:
            - ALL
    networks:
        net-10.9.0.0:
            ipv4_address: 10.9.0.2
    volumes:
            - ./volumes:/volumes
    command: bash -c "
                /etc/init.d/openbsd-inetd start  &&
                tail -f /dev/null
             "
```

כאן אנו מרכיבים את הקונטיינר של המקבל, כתובתו היא 10.9.0.5.

```
Sender:
    image: handsonsecurity/seed-ubuntu:large
    container_name: Sender-10.9.0.2
    tty: true
    cap_add:
            - ALL
    networks:
        net-10.9.0.0:
            ipv4_address: 10.9.0.2
    volumes:
            - ./volumes:/volumes
    command: bash -c "
                /etc/init.d/openbsd-inetd start  &&
                tail -f /dev/null
             "
```

כאן אנו מגידירים את טווח כתובות ה־IP עבור כל רשת הדוקרים.

```
networks:
    net-10.9.0.0:
        name: net-10.9.0.0
        ipam:
            config:
                - subnet: 10.9.0.0/24
```

3 תעבורה

בפרק זה נעבור על תעבורת המערכת, נעבור על קטעי תעבורה נבחרים ונסביר את התעבורה לעומק.

3.1 ללא איבוד חבילות

3.1.1 TCP

כאן ניתן לראות פתיחת קשר בין השולח לבין שרת הפרוקסי הראשון.

1	0.000000000	10.9.0.2	10.9.0.3	TCP	74	51290 → 61328 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=454724723 TSecr=0 WS=128
2	0.000092731	10.9.0.3	10.9.0.2	TCP	74	61328 → 51290 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM=1 TSval=4124653757 TSecr=454724723 WS=128
3	0.000172911	10.9.0.2	10.9.0.3	TCP	66	51290 → 61328 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=454724723 TSecr=4124653757

כאן ניתן לראות מספר חבילות של העברת סגמנטים של הקובץ.

9	0.010992648	10.9.0.3	10.9.0.2	TCP	66	61328 → 51290 [ACK] Seq=1 Ack=16385 Win=56064 Len=0 TSval=4124653768 TSecr=454724734
10	0.011015293	10.9.0.2	10.9.0.3	TCP	8258	51290 → 61328 [PSH, ACK] Seq=16385 Ack=1 Win=64256 Len=8192 TSval=454724734 TSecr=4124653768
11	0.011021029	10.9.0.3	10.9.0.2	TCP	66	61328 → 51290 [ACK] Seq=1 Ack=24577 Win=51968 Len=0 TSval=4124653768 TSecr=454724734
12	0.011038570	10.9.0.2	10.9.0.3	TCP	8258	51290 → 61328 [PSH, ACK] Seq=24577 Ack=1 Win=64256 Len=8192 TSval=454724734 TSecr=4124653768
13	0.011043998	10.9.0.3	10.9.0.2	TCP	66	61328 → 51290 [ACK] Seq=1 Ack=32769 Win=47872 Len=0 TSval=4124653768 TSecr=454724734
14	0.011066643	10.9.0.2	10.9.0.3	TCP	8258	51290 → 61328 [PSH, ACK] Seq=32769 Ack=1 Win=64256 Len=8192 TSval=454724734 TSecr=4124653768

לבסוף, כאן ניתן לראות את סגירת הקשר שלאחריו המעבר לפרוטוקול הבא.

197	0.013166764	10.9.0.2	10.9.0.3	TCP	66	51290 → 61328 [FIN, ACK] Seq=1100010 Ack=1 Win=64256 Len=0 TSval=454724736 TSecr=4124653770
198	0.013321508	10.9.0.3	10.9.0.2	TCP	66	61328 → 51290 [FIN, ACK] Seq=1 Ack=1100011 Win=1881984 Len=0 TSval=4124653770 TSecr=454724736
199	0.013338917	10.9.0.2	10.9.0.3	TCP	66	51290 → 61328 [ACK] Seq=1100011 Ack=2 Win=64256 Len=0 TSval=454724736 TSecr=4124653770

3.1.2 RUDP

כאן ניתן לראות את תעבורת ה־UDP, תחילה שרת הפרוקסי הראשון שולח בקשת שליחה לשרת הפרוקסי השני ולאחר מכן מתחיל לשלוח סגמנטים ולקבל אישורי קבלה.

200	0.013412020	10.9.0.3	10.9.0.4	UDP	48	49412 → 60573 Len=6
201	0.013485742	10.9.0.4	10.9.0.3	UDP	45	60573 → 49412 Len=3
202	0.013806208	10.9.0.3	10.9.0.4	UDP	16435	49412 → 60573 Len=16393
203	0.024496478	10.9.0.4	10.9.0.3	UDP	45	60573 → 49412 Len=3
204	0.024704955	10.9.0.3	10.9.0.4	UDP	16435	49412 → 60573 Len=16393
205	0.024821707	10.9.0.4	10.9.0.3	UDP	45	60573 → 49412 Len=3

3.2 15% איבוד חבילות

3.2.1 TCP

כאן ניתן לראות כי במצב של איבודי חבילות, קרו מספר תקלות תעבורה ־ חבילות לא הגיעו או הגיעו לא לפי סדר ההגעה הדרוש, כמובן TCP מטפל בכך בצורה אוטומטית.

22	0.007115721	10.9.0.2	10.9.0.3	TCP	8258	43340 → 61328 [PSH, ACK] Seq=57345 Ack=1 Win=64256 Len=8192 TSval=454514784 TSecr=4124443818
23	0.007132241	10.9.0.2	10.9.0.3	TCP	7306	43340 → 61328 [PSH, ACK] Seq=65537 Ack=1 Win=64256 Len=7240 TSval=454514784 TSecr=4124443818
24	0.007142513	10.9.0.2	10.9.0.3	TCP	8754	43340 → 61328 [PSH, ACK] Seq=72777 Ack=1 Win=64256 Len=8688 TSval=454514784 TSecr=4124443818
25	0.007795511	10.9.0.3	10.9.0.2	TCP	66	61328 → 43340 [ACK] Seq=1 Ack=81465 Win=27776 Len=0 TSval=4124443818 TSecr=454514784
26	0.007826384	10.9.0.2	10.9.0.3	TCP	11650	[TCP Previous segment not captured] 43340 → 61328 [PSH, ACK] Seq=95945 Ack=1 Win=64256 Len=11584 TSval=454514784 TSecr…
27	0.007834167	10.9.0.3	10.9.0.2	TCP	78	[TCP Dup ACK 25#1] 61328 → 43340 [ACK] Seq=1 Ack=81465 Win=27776 Len=0 TSval=4124443818 TSecr=454514784 SLE=95945 SRE=…
28	0.007842177	10.9.0.2	10.9.0.3	TCP	1514	[TCP Out-Of-Order] 43340 → 61328 [ACK] Seq=81465 Ack=1 Win=64256 Len=1448 TSval=454514784 TSecr=4124443818
29	0.007844679	10.9.0.2	10.9.0.3	TCP	10202	[TCP Out-Of-Order] 43340 → 61328 [PSH, ACK] Seq=82913 Ack=1 Win=64256 Len=10136 TSval=454514784 TSecr=4124443818
30	0.007849293	10.9.0.3	10.9.0.2	TCP	78	61328 → 43340 [ACK] Seq=1 Ack=82913 Win=26368 Len=0 TSval=4124443818 TSecr=454514784 SLE=95945 SRE=107529
31	0.007853476	10.9.0.2	10.9.0.3	TCP	2962	[TCP Out-Of-Order] 43340 → 61328 [PSH, ACK] Seq=93049 Ack=1 Win=64256 Len=2896 TSval=454514784 TSecr=4124443818
32	0.007858852	10.9.0.3	10.9.0.2	TCP	66	61328 → 43340 [ACK] Seq=1 Ack=107529 Win=1792 Len=0 TSval=4124443818 TSecr=454514784
33	0.007863341	10.9.0.2	10.9.0.3	TCP	1858	[TCP Window Full] 43340 → 61328 [PSH, ACK] Seq=107529 Ack=1 Win=64256 Len=1792 TSval=454514784 TSecr=4124443818
34	0.007934817	10.9.0.3	10.9.0.2	TCP	66	61328 → 43340 [ACK] Seq=1 Ack=109321 Win=13184 Len=0 TSval=4124443818 TSecr=454514784
35	0.007941686	10.9.0.2	10.9.0.3	TCP	5514	43340 → 61328 [PSH, ACK] Seq=109321 Ack=1 Win=64256 Len=5448 TSval=454514784 TSecr=4124443818
36	0.007965378	10.9.0.3	10.9.0.2	TCP	66	61328 → 43340 [ACK] Seq=1 Ack=114769 Win=64128 Len=0 TSval=4124443818 TSecr=454514784
37	0.007970889	10.9.0.2	10.9.0.3	TCP	8754	[TCP Previous segment not captured] 43340 → 61328 [PSH, ACK] Seq=122009 Ack=1 Win=64256 Len=8688 TSval=454514784 TSecr…
38	0.007974168	10.9.0.3	10.9.0.2	TCP	78	[TCP Window Update] 61328 → 43340 [ACK] Seq=1 Ack=114769 Win=81536 Len=0 TSval=4124443818 TSecr=454514784 SLE=122009 S…
39	0.213715280	10.9.0.2	10.9.0.3	TCP	1514	[TCP Retransmission] 43340 → 61328 [ACK] Seq=114769 Ack=1 Win=64256 Len=1448 TSval=454514990 TSecr=4124443818
40	0.213737256	10.9.0.3	10.9.0.2	TCP	78	61328 → 43340 [ACK] Seq=1 Ack=116217 Win=84352 Len=0 TSval=4124444024 TSecr=454514990 SLE=122009 SRE=130697
41	0.213747765	10.9.0.2	10.9.0.3	TCP	2962	[TCP Retransmission] 43340 → 61328 [PSH, ACK] Seq=116217 Ack=1 Win=64256 Len=2896 TSval=454514990 TSecr=4124444024
42	0.213750984	10.9.0.3	10.9.0.2	TCP	78	61328 → 43340 [ACK] Seq=1 Ack=119113 Win=90240 Len=0 TSval=4124444024 TSecr=454514990 SLE=122009 SRE=130697
43	0.213755864	10.9.0.2	10.9.0.3	TCP	2962	[TCP Retransmission] 43340 → 61328 [PSH, ACK] Seq=119113 Ack=1 Win=64256 Len=2896 TSval=454514990 TSecr=4124444024
44	0.213757199	10.9.0.2	10.9.0.3	TCP	1514	[TCP Previous segment not captured] 43340 → 61328 [ACK] Seq=132145 Ack=1 Win=64256 Len=1448 TSval=454514990 TSecr=4124…
45	0.213761550	10.9.0.3	10.9.0.2	TCP	66	61328 → 43340 [ACK] Seq=1 Ack=130697 Win=96000 Len=0 TSval=4124444024 TSecr=454514990
46	0.213760332	10.9.0.2	10.9.0.3	TCP	1514	[TCP Previous segment not captured] 43340 → 61328 [ACK] Seq=137937 Ack=1 Win=64256 Len=1448 TSval=454514990 TSecr=4124…
47	0.213768555	10.9.0.3	10.9.0.2	TCP	86	[TCP Window Update] 61328 → 43340 [ACK] Seq=1 Ack=130697 Win=101760 Len=0 TSval=4124444024 TSecr=454514990 SLE=137937 …
48	0.213774529	10.9.0.2	10.9.0.3	TCP	1514	[TCP Out-Of-Order] 43340 → 61328 [ACK] Seq=130697 Ack=1 Win=64256 Len=1448 TSval=454514990 TSecr=4124444024
49	0.213775537	10.9.0.2	10.9.0.3	TCP	1514	[TCP Out-Of-Order] 43340 → 61328 [ACK] Seq=133593 Ack=1 Win=64256 Len=1448 TSval=454514990 TSecr=4124444024
50	0.213780276	10.9.0.3	10.9.0.2	TCP	78	61328 → 43340 [ACK] Seq=1 Ack=133593 Win=104704 Len=0 TSval=4124444024 TSecr=454514990 SLE=137937 SRE=139385
51	0.213781207	10.9.0.3	10.9.0.2	TCP	78	61328 → 43340 [ACK] Seq=1 Ack=135041 Win=107520 Len=0 TSval=4124444024 TSecr=454514990 SLE=137937 SRE=139385
52	0.213785981	10.9.0.2	10.9.0.3	TCP	2962	[TCP Out-Of-Order] 43340 → 61328 [PSH, ACK] Seq=135041 Ack=1 Win=64256 Len=2896 TSval=454514990 TSecr=4124444024
53	0.213787123	10.9.0.2	10.9.0.3	TCP	2962	43340 → 61328 [PSH, ACK] Seq=139385 Ack=1 Win=64256 Len=2896 TSval=454514990 TSecr=4124444024
54	0.213790751	10.9.0.3	10.9.0.2	TCP	66	61328 → 43340 [ACK] Seq=1 Ack=139385 Win=113408 Len=0 TSval=4124444024 TSecr=454514990

3.2.2 RUDP

כאן ניתן לראות כי ה־UDP שלח את אותה החבילה פעמיים כיוון שלא קיבל אישור קבלה בזמן הנכון. (טיפול באמינות).

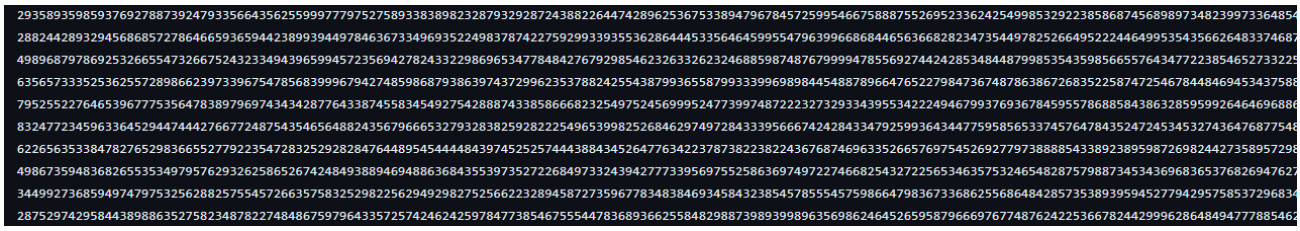
659	78.578267138	10.9.0.3	10.9.0.4	UDP	16436	58165 → 60573 Len=16394
660	79.029453911	10.9.0.3	10.9.0.4	UDP	16436	58165 → 60573 Len=16394
661	79.029869392	10.9.0.4	10.9.0.3	UDP	46	60573 → 58165 Len=4

4 פלט

בפרק זה נציג צילומי מסך נדרשים בהגשה.

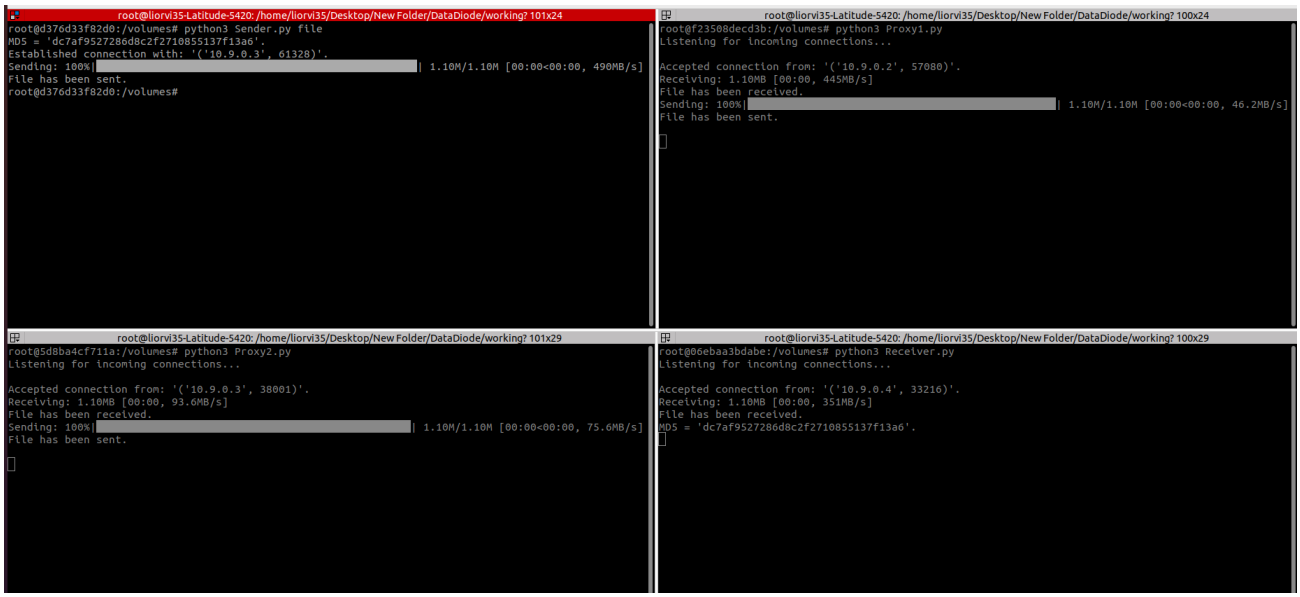
4.1 קובץ

הקובץ אותו אנחנו שולחים הוא קובץ המכיל ספרות 0-9. הקובץ גנרי ורנדומי לחלוטין, חסר כל משמעות. בעל משקל של ± 1 MB.



4.2 הדפסות

כך נראה הפלט של התוכנית לאחר סיום הריצה, על מנת להראות את כל חלונות הטרמינל בחלון יחיד לשם הנוחות אנו השתמשנו ב-"Terminator". כמובן, ניתן לראות שבכל טרמינל ישנו סרגל התקדמות, בין הפרוקסיים יש 2 אחד לקבלה ואחד לשליחה, יש הדפסות בהתאם המציינות התחברות, שליחה וקבלה וכמובן בנקודת ההתחלה ובנקודת הסוף מודפס גם MD5, ניתן כמובן לראות שהם זהים.



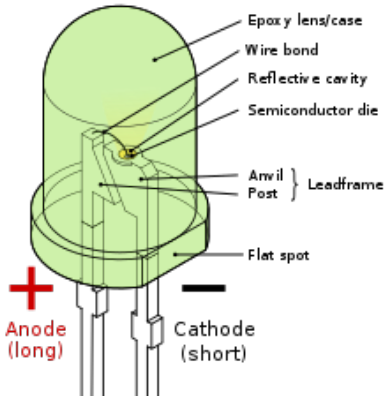
5 מענה על שאלות

בפרק זה נענה על השאלות הנוספות למטלה.

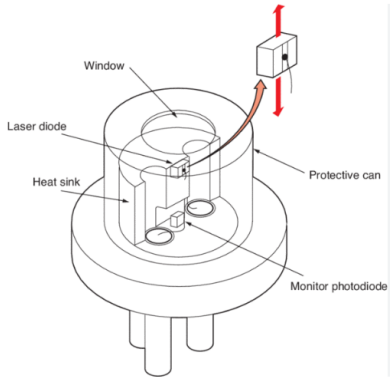
5.1 שאלה ראשונה

סוגי הדיודות השונות:

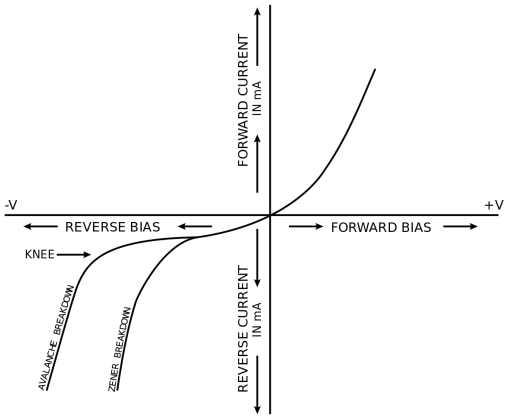
1. Light Emitting Diode - דיודה זו היא התקן (מנורת לד המוכרת) מוליך למחצה אשר פולט אור כאשר זרם חשמלי עובר דרכה.



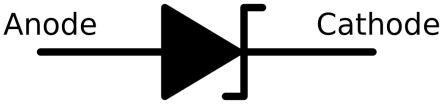
2. Laser diode - דיודת לייזר הפולטת אור קוהרנטי כאשר עובר זרם חשמלי. זוהי בעצם גרסה קומפקטית ויעילה של לייזר, והיא נמצאת בשימוש נפוץ במגוון רחב של יישומים כגון תקשורת אופטית, סיבים אופטיים, מדפסות לייזר, קוראי ברקוד ומצביעי לייזר.



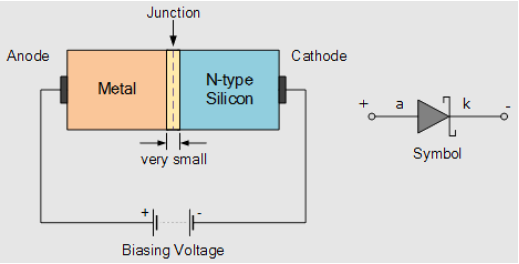
3. AVALANCHE DIODE - דיודה זו שייכת לסוג הטיה הפוכה ופועלת באמצעות אפקט המפולת. כאשר נפילת המתח קבועה ואינה תלויה בזרם, מתרחשת התמוטטות המפולת.



4. ZENER DIODE - דיודת זנר היא דיודה המיועדת לפעול במצב מוטה ולשמור על מתח כמעט קבוע על פני המסופים שלה, גם כאשר הזרם דרכה משתנה. דיודות זנר משמשות בדרך כלל כמווסת מתח, המספקת מתח ייחוס יציב למעגלים אלקטרוניים שונים.

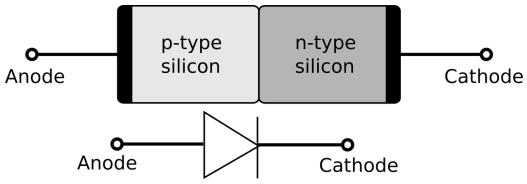


5. SCHOTTKY DIODE - דיודה שוטקי היא סוג של דיודה מוליכים למחצה המאופיינת במפלת המתח הנמוכה שלה קדימה ובמהירות המיתוג המהירה שלה. דיודות שוטקי נוצרות על ידי צומת מתכת מוליכים למחצה, כאשר שכבת מתכת מופקדת על חומר מוליכים למחצה כגון סיליקון או גליום ארסניד.

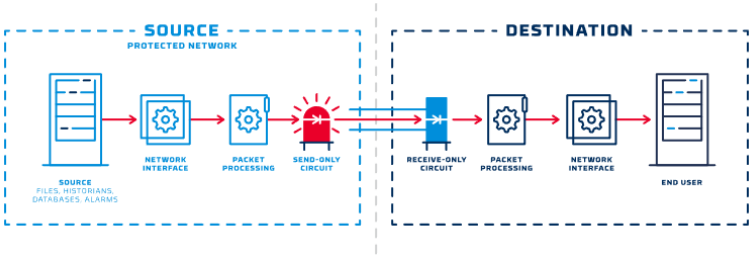


6. Photodiode - Photodiode פוטודיודה היא סוג של דיודה שנועדה לזהות אור על ידי המרת פוטונים חלקיקי אור לזרם חשמלי.

7. PN junction diode - דיודת צומת PN היא התקן מוליכים למחצה בסיסי המורכב מחומר מוליך למחצה מסוג p וחומר מוליך למחצה מסוג n המחוברים יחדיו ליצירת צומת p-n לחומר מסוג p יש עודף של חורים נשאי מטען חיובי, בעוד שלחומר מסוג n יש עודף של אלקטרונים נשאי מטען שלילי.



8. Data Diode - כמובן, דיודת רשת. דיודת נתונים היא התקן תקשורת רשת חד-כיווני המאפשר העברה בטוחה וחד-כיוונית של נתונים בין רשתות. תכנון דיודות הנתונים שומר על הפרדה פיזית וחשמלית בין רשתות מקור ויעד, תוך יצירת העברת נתונים חד-כיוונית סגורה לחלוטין בין רשתות ללא ניתוב.
כמו כן, בדיודה זו ישנו תהליך הנקרא שבירת פרוטוקול אשר הוא תהליך של סיום פרוטוקול העברת נתונים, שליחת מטען הנתונים באמצעות פרוטוקול אחר, ולאחר מכן כינון מחדש של הפרוטוקול המקורי לפני שהנתונים מגיעים ליעדם.



במטלה אנחנו מימשנו את דיודה מספר 8. מימשנו ארבע מכונות - שולח, פרוקסי ראשון, פרוקסי שני, מקבל. השולח שולח את המידע לשרת הפרוקסי הראשון בפרוטוקול TCP, לאחר מכן מתבצע שבירת פרוטוקול וכעת המידע נשלח בפרוטוקול ייחודי שאנחנו בנינו על בסיס UDP לפרוקסי השני ומשם עובר חזרה לפרוטוקול TCP ושולח את המידע למקבל. ניתן להכנס לדקויות המימוש בפרק על האלגוריתמיקה.
ברמה התקשורתית - המידע נע בכיוון אחד בלבד אמנם יש חבילות שכן חוזרות אחורה אך אלו חבילות מסוג ACK בלבד. תחילה בין השולח לשרת הפרוקסי הראשון מתבצעת לחיצת ידיים משולשת של TCP לאחר

מכן הקובץ מתחיל להישלח ולאחר כל סגמנט נשלח ACK. לבסוף יש ניתוק קשר. בין הפרוקסי הראשון לשני תחילה יש תבילה של בקשת שליחת הקובץ וחוזר ACK, לאחר מכן שוב הקובץ נשלח לבסוף בין הפרוקסי השני ולבין המקבל קורה אותו תהליך כפי שתיארנו בהתחלה כי שוב נשלח ב־TCP. לפירוט יותר מדויק הרחבנו בפרק על התעבורה.

5.2 שאלה שנייה

לשם השאלה, נניח שאנחנו ארגון בעל דיודת רשת. כלומר, כל מידע אשר אשר אנחנו מקבלים מלקוחותינו, עובר דרך הדיודה וכי לבסוף אין מידע היוצא מהארגון בגלל זרימת נתונים חד כיוונית.
הבעיות שיכולות להיווצר בגלל זרימה חד כיוונית הם:

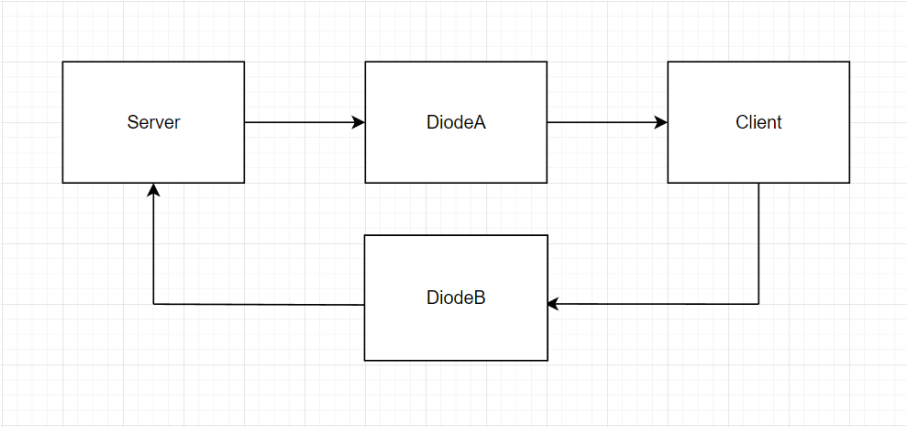
1. אנחנו לא יכולים להחזיר שום מידע ללקוח, אולי יש מידע שהלקוח צריך בשביל המשך התהליך אבל אנחנו לא יכולים להחזיר לו כלום.
2. לאחר שלקוח כבר העביר לנו מידע אבל אנחנו צריכים עוד מידע בזמן מסויים, אין לנו דרך לשלוח לו בקשה כזאת בגלל שזאת זרימה בכיוון ההפוך.

5.3 שאלה שלישית

מערכת שהיא "Air Gapped"זוהי מערכת אשר מבודדת לחלוטין מחיבור לרשת האינטרנט החיצונית, לרוב מבחינה פיזית. את המושג הנ"ל ניתן להשליך על דיודת הרשת שהמטלה עוסקת בה, כיוון שאם יש לנו ארגון בעל דיודה, אין נתונים שיוצאים ממנו כלומר גם אין שום נתונים אשר עוברים ברשת האינטרנט ולכן למעשה הארגון הוא "Air Gapped", שכן בפרט אפילו אין כלל נתונים היוצאים לרשתות אחרות כיוון שכל המידע הנכנס לארגון עובר דרך הדיודה, וזוהי נניח נקודת הכניסה/יציאה היחידה (כדי לשמור על אבטחת המידע) ולכן אכן, הארגון מנותק מכל רשת חיצונית.

5.4 שאלה רביעית

נציע דרך לאפשר זרימת מידע בשני הכיוונים ־ מהלקוח לארגון ומהארגון ללקוח. אם בעזרת דיודה אחת אנחנו מקבלים רמה גבוהה של אבטחה בזרימת מידע חד כיוונית, אז נשקול להגדיר דיודה נוספת ואז בסופו של דבר יהיו לנו שתי דיודות, אחת לקבלת המידע ואחת לשליחת המידע. נגדיר את הארכיטקטורה החדשה בצורה הבאה:



כעת, קיבלנו שיש אפשרות לזרימה בשני הכיוונים. כמובן, זה לחלוטין לא מספיק "AS IS" ויש צורך לטפל בחריגות אבטחה. טיפולים נוספים אפשריים הם להצפין את כל המידע היוצא ללקוח מהארגון ונכנס לארגון מהלקוח, כמובן, יש צורך גם לפתח מספר פרוטוקוליים יחודיים שאף אחד לא מכיר ואז כדי למנוע מצב של פגיעה ומניפולציות במערכת במקרה של השתלטות על אחת הדיודות (אנחנו לא נרצה לתת לתוקף את החשיבה שאם כל המידע היוצא והנכנס עובר דרך דיודות אז אם הוא השתלט על הדיודות הוא עקף את הארגון) ואפילו אולי כדאי לנתב את המידע דרך צד שלישי שאינו ידוע לאף אחד וכך זוהי דרך לשפר את האבטחה (אך הרעיון היסודי של הארכיטקטורה החדשה הוא התרשים).

6 ביבליוגרפיה

רשימת מקורות אשר עזרו לנו בעת כתיבת המסמך.

1. Cryptographic hash function
   https://w.wiki/6djM
2. MD5 Hash
   https://w.wiki/6djf
3. Diode Types
   https://byjus.com/physics/diodes/#types-of-diodes
4. Reliable UDP 1
   https://www.geeksforgeeks.org/reliable-user-datagram-protocol-rudp/
5. Reliable UDP 2
   https://w.wiki/6dja
6. Congestion-Control & Flow-Control for RUDP
   https://www.geeksforgeeks.org/difference-between-flow-control-and-congestion-control/?ref=rp
7. Unidirectional network
   https://w.wiki/6djd
8. Data Diode
   https://owlcyberdefense.com/blog/what-is-data-diode-technology-how-does-it-work/
9. Air-Gapped Networks and Data Diodes
   https://owlcyberdefense.com/blog/air-gapped-networks-and-data-diodes/