סמינר בתכנות מערכות דפנסיבי

מגיש: אלון שנקלר 206280547

Table of Contents

[מבוא – סמינר בתכנות מערכות דפנסיבי 3](#_Toc528442911)

[חלק ראשון – תוכנת לקוח בשפת C++ 4](#_Toc528442912)

[שפת C++ תכונות ומרכיבים 4](#_Toc528442913)

[תכנות פרוצדורלי 4](#_Toc528442914)

[User-Defined Types 5](#_Toc528442915)

[תכנות מונחה עצמים 6](#_Toc528442916)

[Generic Programming 7](#_Toc528442917)

[איומי אבטחה במנגנוני זיכרון בשפת C++ 8](#_Toc528442918)

[דריסת חוצץ – Buffer Overflow 8](#_Toc528442919)

[התמודדות עם דריסת חוצץ 8](#_Toc528442920)

[Dangling Pointer attacks 10](#_Toc528442921)

[התמודדות עם Dangling Pointers 10](#_Toc528442922)

[חלק שני – תוכנת שרת בשפת פייתון 11](#_Toc528442923)

[שפת פייתון תכונות ומרכיבים 11](#_Toc528442924)

[איומי אבטחה בשפות דינאמיות ושפת פייתון 14](#_Toc528442925)

[הזרקה דינאמית של משתנים 14](#_Toc528442926)

[התמודדות עם הזרקה דינאמית של משתנים 14](#_Toc528442927)

[חולשה במנגנון הסריאליזציה המובנה של פייתון – pickle 15](#_Toc528442928)

[התמודדות עם החולשה במנגנון הסריאליזציה המובנה pickle 15](#_Toc528442929)

[חלק שלישי – תקשורת בין הרכיבים השונים 17](#_Toc528442930)

[תקשורת רשתית בסיסית 17](#_Toc528442931)

[איומים בתקשורת רשתית 18](#_Toc528442932)

[התחזות ו-Man in the Middle 18](#_Toc528442933)

[התמודדות עם תקיפות התחזות 18](#_Toc528442934)

[גניבת מידע וריגול 19](#_Toc528442935)

[התמודדות עם גניבת מידע 19](#_Toc528442936)

[חלק רביעי – איומי אבטחה מערכתיים 20](#_Toc528442937)

[איום ראשון – מניעת שירות (/מבוזרת) D/DoS 20](#_Toc528442938)

[התמודדות עם תקיפות מניעת שירות 20](#_Toc528442939)

[איום שני – חולשות במערכת ההפעלה 21](#_Toc528442940)

[התמודדות עם חולשות במערכת ההפעלה 21](#_Toc528442941)

[חלק חמישי – סיכום החלק התיאורטי והצגת החלק המעשי 22](#_Toc528442942)

[סיכום 22](#_Toc528442943)

[החלק המעשי 22](#_Toc528442944)

[Bibliography 23](#_Toc528442945)

# מבוא – סמינר בתכנות מערכות דפנסיבי

עבודת סמינר זו נכתבת כחלק מהקורס **סמינר בתכנות מערכות דפנסיבי** באוניברסיטה הפתוחה.

עיסוק עבודה זו הוא בחינה של תהליך הפיתוח של מערכות תוכנה גדולות אשר מורכבות מרכיבים שונים אשר כתובים בשפות תכנות שונות וערוצי התקשורת בין הרכיבים השונים במערכת, תוך שימת דגש עיקרי על ההיבטים של אבטחת המערכת.

במהלך העבודה נסקור את שפות התכנות השונות אשר הולך להיעשות בהן שימוש במערכת.  
סקירת כל שפת תכנות תכלול סקירה ודוגמאות של תכונות שונות של השפה.  
לאחר הצגת תכונות השפה, נציג את השפה בהיבט אבטחתי ונסקור את הסכנות ופרצות האבטחה אשר השימוש בשפה זו מוסיף למערכת שלנו.

לאחר סקירת שפות התכנות אנו נסקור את ההיבטים המערכתיים של המערכת שלנו אשר מציבים אותה בסכנה.  
חלק זה כולל שני היבטים, ההיבט התקשורתי וההיבט המערכתי.

בהיבט התקשורתי נסקור איומי אבטחה אשר נובעים מהעובדה כי על הרכיבים השונים של המערכת שלנו לתקשר מעל תווך רשתי.  
בהיבט המערכתי נבחן מספר איומים אשר ניצבים מול המערכת שלנו מעצם היותה *מערכת תוכנה* אשר מורכבת ממספר רכיבים.

עבור כל איום אבטחה שנסקור בעבודת סמינר זו נציע דרכי התמודדות שונות, בעלות רמות שונות של הרמטיות הגנה.

לאחר סקירת השפות והאיומים השונים, נעבור לחלק המעשי.  
בחלק המעשי של סמינר זה נפתח מערכת תוכנה בעלת המאפיינים אשר נסקרו במהלך החלק התיאורטי של עבודה זו, כלומר:

* מערכת תוכנה בעלת מספר רכיבים
* הרכיבים יכתבו בשפות תכנות שונות
* הרכיבים יתקשרו ביניהם מעל תווך רשתי

במהלך כתיבת החלק המעשי ניישם פתרונות אבטחה אשר יוצגו לאורך הפרקים השונים של החלק התיאורטי של עבודה זו.

# חלק ראשון – תוכנת לקוח בשפת C++

## שפת C++ תכונות ומרכיבים

C++ היא שפת תכנות *לא ספציפית* – כלומר התוכנות אשר השפה מיועדת לשמש לכתיבתן אינן למטרה ספציפית, אלא כללית, לכל סוג תוכנה קיים. [1] [2]

עם זאת, היישום הנפוץ של השפה הוא בתחום *תכנות המערכות* [התחום העוסק בכתיבה של תוכנות אשר או דורשות ביצועים גבוהים (למשל משחקי מחשב או מערכות מוטמעות בחומרה) או תוכנות אשר אינן נגישות למשתמש ישירות ומשמשות כמצע\פלטפורמה ליישומי משתמש].

שפת C++ היא super-set של שפת C. כלומר, שפת C++ היא שפת C בתוספת יכולות ותחביר (עד כדי פרטים זניחים).  
אחת התופעות של תכונה זו היא שכל תכנית בשפת C, למעט מקרי קצה ייחודים, היא גם תכנית תקינה בשפת C++.

### תכנות פרוצדורלי

פרדיגמת התכנות הבסיסית ביותר הנתמכת ע"י שפת C++ היא *תכנות פרוצדורלי*.

Decide which procedures you want; use the best algorithms you can find. -Bjarne Stroustrup [1]

פרדיגמת התכנות הפרוצדורלי מתארת את הפעולה של כתיבת תוכנה כרצף הבא:

1. מציאת האלגוריתמים המתאימים לפתרון בעיה נתונה
2. תרגום כל אלגוריתם בסיסי לפונקציה בשפת התכנות
3. כתיבת התוכנה כרצף של קריאות לפונקציות והעברת פרמטרים ותוצאות מקריאה לקריאה

כדי לבצע את הפעולה של העברת פרמטרים ותוצאות ממקום למקום אנו מגדירים *משתנים*, שהם צימוד בין שם בקוד לבין פיסת מידע.  
ב-C++ יש לציין את הטיפוס של כל פיסת מידע לצורך שימוש בה, וקיימת התאמה קרובה בין סוגי המידע (*הטיפוסים* השונים) הנתמכים לחומרה אשר התוכנה רצה עליה.

בנוסף, קיים טיפוס משתנה פרימיטיבי נוסף מסוג מצביע, אשר מאפשר לגשת אל הכתובת בזיכרון הווירטואלי אשר בה שמור ערכו של משתנה אחר, לצורך ביצוע פעולות כמו החזרה של תוצאות מרובות מפונקציה.

שתי תכונות אלה, יחד עם תכונות אחרות משייכות את C++ לקבוצת *Low-Level Programming Languages.*  
שפות אלו מתאפיינות בקרבה לחומרה מבחינת הטיפוסים הנפוצים בהן, מבחינת היכולת לתרגם קוד ישירות משפות מקבוצה זו אל שפת מכונה ובגישה שהן מספקות אל מנגנוני חומרה כגון גישה ישירה לזיכרון הווירטואלי.

עם זאת, ראוי לציין כי ניתן לשייך את C++ לקבוצת *שפות התכנות העליות*, וזאת מכיוון שניתן לבצע בשפה אבסטרקציות שונות אשר אינן מוכרות ע"י המכונה (אשר יוזכרו בהמשך הפרק) ובכך השפה "מורחקת" מהמכונה ומיתר שפות הסף.

לכן C++ הינה שפה היברידית מבחינת השיוך שלה למחלקות Low/High Level Programming Languages, והמסקנה המתבקשת היא שלא מדובר בשיוך לקבוצה אלא במיקום על ציר המרחק מהמכונה.

לצורך תיאור אלגוריתמים בקוד ניתן לבצע על המשתנים השונים פעולות אריתמטיות ופעולות "מבחן" או *control flow* אשר מאפשרות לשנות את מסלול ריצת הקוד באמצעות בחינת ערך האמת של פסוקים אריתמטיים שונים.

להלן דוגמת קוד אשר עוסקת בהגדרת משתנים בסיסיים, מצביעים ו-control flows שונים:

/\*\*  
 \* A function to safely divide a by b  
 \* @param a - The number to be divided  
 \* @param b - The number to divide by  
 \* @param out - The out parameter where the result is saved (using a pointer)  
 \* @return 1 if succeeded and 0 otherwise  
 \*/  
**int** divide\_values(**int** a, **int** b, **int** \*out) {  
 **if** (a < b) { // A simple if test control-flow statement  
 **return** 0;  
 }  
 **if** (0 == b) {  
 **return** 0;  
 }  
 \*out = a / b; // Saving value to the integer pointed to by `out`  
 **return** 1;  
}  
  
**int** main() {  
 **int** a = 20, b = 4, c; // Definition of multiple variables of type int  
 **int** \*p = &c; // Definition of a variable of type pointer, and accessing `c`s address.  
  
 **int** succeeded = divide\_values(a, b, p); // Function call  
  
 **if** (succeeded) {  
 cout << "a / b: " << c;  
 }  
 **return** 0;  
}

שפות רבות מתנהלות בצורה דומה, ולכן נדלג על יתר הקונספטים הבסיסיים המשותפים לרוב שפות התכנות ונסתכל על תכונות ייחודיות יותר של C++.

### User-Defined Types

שפת C++ מאפשרת למתכנת להוסיף סוגי מידע חדשים שהוא יוכל לעבוד איתם בהמשך, באמצעות הגדרת טיפוסים חדשים עם המילה השמורה *class*.

המתכנת יכול לייצר טיפוס חדש באמצעות הרכבה של מספר טיפוסים בסיסיים ואוסף פעולות שהטיפוס החדש יתמוך בהן.

הטיפוסים שהמתכנת מגדיר בעצמו מתנהגים כמעט לחלוטין כמו הטיפוסים המובנים של השפה ומגדילים את ארגז הכלים הזמין למתכנת.

באמצעות המילה השמורה *virtual* ניתן להגדיר טיפוסים מופשטים, או ממשקים אשר מאפשרים להסתיר את פרטי המימוש של טיפוסים, ולחשוף למשתמש בהם רק את אוסף הפעולות שהן מאפשרות לו לבצע. תכונה זו מסייעת בהפיכת אותם טיפוסים חדשים לכלים יסודיים ולא לקוד שהמשתמש מכיר.

### תכנות מונחה עצמים

חיסרון בולט של אותם טיפוסים שמוגדרים ע"י משתמש שהצגנו בסעיף הקודם הוא חוסר הגמישות שלהם.  
לאחר הגדרת טיפוס שכזה, לא ניתן להוסיף לו יכולות ולכן אם יש לנו מספר טיפוסים בעלי תכונות והתנהגויות משותפות עלינו לשכפל את החלקים המשותפים בהגדרה שלהם.  
ב-C++ פותרים את הבעיה הזו באמצעות תמיכה ב*תכנות מונחה עצמים*.

תכנות מונחה עצמים הוא פרדיגמת תכנות אשר נתמכת ע"י C++.  
פרדיגמה זו מנחה את המתכנת להסתכל על הבעיה שהוא מנסה לפתור במונחים של העצמים שהבעיה מערבת והתכונות שלהם.  
אובייקט (בעברית *עצם*) הוא ישות בעלת אוסף פעולות שניתן להפעיל עליה, *מצב\תצורה* אשר משתנים בעקבות הפעלת פעולות וטיפוס. [3]

ניתן לראות כי אותם User-Defined Types מתאימים להגדרת אובייקט!  
אבל, אותו חיסרון שתיארנו קודם נפתר בפרדיגמה של תכנות מונחה עצמים ע"י הרחבת הגדרת הטיפוס ל*מחלקה (class)* והגדרת קשרים בין מחלקות.  
קשרים אלו נקראים ירושה, ומאפשרים הרחבה של טיפוס קיים באמצעות הוספת טיפוסים בסיסיים למבנה המרוכב והוספת פעולות ש*תת-הטיפוס* יתמוך בהן, מעבר לפעולות המוגדרות ב*מחלקת האב*.

להלן דוגמת קוד אשר מגדירה מחלקה חדשה מסוג מלבן, מייצרת אובייקט שהוא מופע של המחלקה ועושה בו שימוש:

// Based on http://www.cplusplus.com/doc/tutorial/classes/  
  
// Definition of a rectangle class with 2 members  
**class** Rectangle {  
 **int** width, height;  
**public**:  
 // Declaring public setter method for updating the members of the class  
 **void** set\_values(**int**, **int**);  
  
 // A public method for calculation using the members of the class  
 **int** area() { **return** width \* height; }  
};  
  
  
// Implementation of the formerly declared setter method  
**void** Rectangle::set\_values(**int** x, **int** y) {  
 width = x;  
 height = y;  
}  
  
**int** main() {  
 Rectangle rect{}; // Initializing a new object of type Rectangle  
 rect.set\_values(3, 4); // Calling it's setter method  
 cout << "area: " << rect.area(); // Printing the result of execution of an object's method  
 **return** 0;  
}

### Generic Programming

בעקבות הקרבה של הטיפוסים בשפת C++ לטיפוסים הנתמכים ע"י החומרה והבחירה של מעצבי C++ לדרוש מהמתכנת להצהיר בצורה נוקשה על הטיפוס של כל משתנה, נקלענו למצב בו יש צורך לכתוב מחדש כל אלגוריתם וכל מחלקה עבור כל טיפוס שהם פועלים מעליו (או עבור כל צירוף של טיפוסי משתנים שהם פועלים מעליהם!).  
למשל, אנו נאלץ לכתוב אלגוריתם למיון רשימת ערכים מספריים לפי גודל פעם אחת עבור כל אחד מהטיפוסים:

* int (מספר שלם עד ערך מוחלט מסוים)
* float (מספר ממשי עד ערך מוחלט מסוים)
* double (מספר ממשי עד ערך מוחלט אחר מ- float)

ועוד!

כדי למנוע את הכפילות המשמעותית הזו הוסיפו לשפת C++ תמיכה ב*תכנות גנרי*, צורת עבודה שבאמצעות הגדרת *תבניות* של טיפוסים ואלגוריתמים.  
התבנית היא אותו קטע קוד בשינוי הקל שאחד או יותר מהטיפוסים מוחלף ב-placeholder כך שניתן יהיה לשכפל את קטע הקוד הזה עם החלפת ה-placeholder בטיפוס אמיתי ולקבל את המימוש עבור אותו טיפוס.  
בתהליך ההידור (compilation) משוכפלת התבנית עם החלפת ה-placeholder בהתאם לשימושים שנעשים בה בתוכנית.

## איומי אבטחה במנגנוני זיכרון בשפת C++

### דריסת חוצץ – Buffer Overflow

כששפת C תוכננה, סדר העדיפויות היה קודם ביצועים ויעילות בזיכרון ורק לאחר מכן בטיחות.

לכן בשפות C ו-C++ בעקבותיה האחריות על ניהול הזיכרון אשר נצרך על ידי התוכנית נמצאת בידי המתכנת.

כאשר המתכנת מעוניין להשתמש בזיכרון לצורך אחסון מידע כמו מחרוזות של תווים (טקסט) על המתכנת להצהיר כמה תאי זיכרון הוא מעוניין להקצות למידע.  
חולשות buffer overflow מתאפשרות מאחר והשפה אינה מוודאת שגישה אל אחד מתאי המידע (*המערך*) אכן נמצאת בתוך הגודל שהוקצה למערך.  
לכן, אם הקצנו למערך שיכיל את קלט המשתמש לסיסמה שלו 20 תווים, וננסה לגשת לתו ה-21, הקוד עדיין יתקמפל ואכן יהיה ניסיון לבצע גישה זו. [4]

הגרסה הנפוצה של ניצול חולשות דריסת חוץ היא *Smashing the stack (ריסוק המחסנית)*.  
כל תוכנה שרצה על המחשב זוכה לכמות זיכרון משלה.  
לזיכרון הזה שני חלקים – Stack (מחסנית) ו-Heap (ערימה).  
בזיכרון המחסנית נשמרים בין היתר המשתנים המקומיים של פונקציות ורצף הקריאות לפונקציות בתוכנית, המיוצג על ידי מחסנית של קריאות.  
בכל קריאה לפונקציה נדחפים למחסנית הארגומנטים להרצת הפונקציה, הכתובת שאליה על הקוד לחזור לאחר סיום הריצה, משתנים מקומיים ועוד.

כעת, אם בתכנית מסוימת מקצים במשתנה מקומי מערך שמתמלא ע"י קלט משתמש, והמתכנת לא וידא שהוא אינו חורג מגבולות המערך אנו יכולים להכניס קלט זדוני אשר יחרוג מגבולות המערך וידרוס את הכתובת שאליה על הקוד לחזור לאחר סיום הריצה.

כלומר, אנו יכולים לגרום לתוכנה להריץ קוד אחר מהמתוכנן.  
ניצול מתוחכם מעט יותר יהיה לדחוף בתוך אותו המערך שהוקצה לנו קוד זדוני אשר נרצה להריץ ואז לגרום לתוכנה "לחזור" אליו ולהריץ קוד שרירותי!  
לקוד זה קוראים “shellcode” מאחר ואחד השימושים המקוריים של הרצת קוד זדוני הייתה להריץ shell על המכונה בהרשאות גבוהות ולקבל שליטה מלאה.

כמובן שקיימות דרכים שונות לניצול חולשות דריסת חוצץ אשר כולן מתבססות על חוסר בדיקת גבולות המערך.

### התמודדות עם דריסת חוצץ

עם השנים והפרצות השונות המודעות לבעיות דריסת חוצץ עלתה.

אחת ההתמודדויות הלא תמיד מעשית היא שימוש בשפת תכנות אחרת מ-C ו-C++, כזו אשר מגנה על המתכנת מחריגות ממערך (בין אם בזמן קומפילציה ובין אם בזמן ריצה).  
כמובן שלא נרחיב כאן על פתרון זה.

שיטה נוספת היא להשתמש בספריות קלט וניהול זיכרון "בטוחות".  
הפונקציה *gets* אשר משמשת לקבלת קלט משתמש ידועה לשמצה בתור מקור לחולשות דריסת חוצץ, מאחר והיא מעתיקה את הקלט לתוך המערך המועבר לה ללא בדיקת גבולות.  
במקום להשתמש בקריאה זו, המתכנת יכול להשתמש במקום בפונקציה *fgets* אשר מקבלת כפרמטר נוסף אורך מקסימלי לקריאה, ובכך מונעת חריגה מגבולות. [5]

פתרונות מהסוג של החלפת קריאות מקובלים מאוד ולהשתמש בהם זה סטנדרט בתעשייה.  
עם זאת, מאחר ופתרונות אלו משאירים את האחריות בידי המתכנת ודורשים ממנו לזכור בכל קריאה המתעסקת בזיכרון לדעת את הסיכונים, קיים צורך בפתרונות מתוחכמים יותר.

שיטת התמודדות נוספת, אשר מובנית במערכות הפעלה רבות ואינה דורשת פעולות אקטיביות מצד המתכנת היא *Non-Executable Memory* (*זיכרון שאינו ניתן להרצה*).  
שיטה זו מקשה על התוקף להריץ קוד שרירותי משלו באמצעות הוספת ההגבלה שאזורים מסוימים בזיכרון אינם ניתנים להרצה.  
לכן, אם התוקף ניסה לדרוס את מצביע החזרה כך שיצביע אל ה-shellcode שהכניס למערך, כאשר התוכנית תנסה לקפוץ אל הקוד הזדוני הנ"ל היא תזהה שהקוד נמצא בזיכרון שאינו מיועד להרצה ותמנע את הרצת הקוד הזדוני! [6]

אפילו שיטה זו אינה מבטיחה חסינות מתקיפות מסוג זה מאחר ועדיין ניתן לקפוץ אל אזורים אחרים בקוד הלגיטימי של התוכנה או הספריות אשר היא עושה בהן שימוש.

### Dangling Pointer attacks

Dangling Pointers הם מצביעים אשר אינם מצביעים לאובייקט תקין.  
מצביעים סוררים שכאלה נוצרים כתוצאה משגיאות תכנות כמו:

* שחרור זיכרון של מצביע ללא איפוס המצביע.  
  נניח ויש לנו מצביע שאולקץ לו זיכרון באמצעות קריאה ל-malloc.  
  לאחר מכן אין יותר צורך באובייקט בזיכרון שאליו הצביע אותו מצביע ובוצעה קריאה ל-free.  
  אם לא זכרו לאפס את המצביע בשלב זה, הוא מצביע לכתובת בזיכרון אשר אינה מוקצית לתוכנית.  
  שימוש בו כעת הוא באג מסוג *use after free*.
* שמירת המצביע לאחר יציאה מה-scope של המשתנה אליו הצביע.  
  נניח והיה לנו מצביע מ-scope עליון, וכעת אנו משנים אותו להצביע על משתנה מה-scope הנוכחי.  
  לאחר יציאה מה-scope הנוכחי אותו המשתנה אינו קיים יותר אך המצביע עדיין מצביע על הכתובת שלו.

הבאג של dangling pointers הופך לחולשה של ממש במקרים הבאים:

* הזיכרון אשר שימש בעבר את המצביע הסורר משמש בשלב מאוחר יותר לצורך אחסון מידע רגיש.  
  כעת במקרים מסוימים התוקף יכול לגשת למידע רגיש אשר הוא לא היה אמור לגשת אליו ולקרוא או לשנות אותו.  
  במידה ואותו המידע משמש לבדיקת הרשאות בהמשך, ניתן לנצל את הגישה אליו לצורך *Privilege Escalation* ומעבר של התוקף מהרשאות נמוכות להרשאות גבוהות יותר.
* אם נעשה במצביע סורר שכזה שימוש לצורך קריאה לפונקציה, התוקף עשוי להריץ קוד שרירותי או קוד זדוני אחר בדומה לנעשה בניצול חולשות דריסת חוצץ.

### התמודדות עם Dangling Pointers

מאחר וגם Dangling Pointers וגם Buffer Overflows הינן חולשות של השחתת זיכרון (memory corruption) חלק מהתמודדויות איתן משותפות.

למשל, גם מ-Dangling Pointers אפשר להתחמק באמצעות שימוש בשפות תכנות בעלות זיכרון מנוהל, שם קיים Garbage Collector אשר מנקה אובייקטים מהזיכרון אך ורק לאחר שהסתיים השימוש בהם.

גם שיטת Non-Executable Memory מקשה על תוקפים לנצל חולשות Dangling Pointers מאחר ואם הם מנסים להשפיע על המצביע הסורר להצביע על קוד זדוני שרירותי, הוא עושי לא לרוץ מאחר והוא ממוקם בקטע קוד שאינו להרצה.

שיטה נוספת להקשות על תוקפים לנצל חולשות של memory corruption נקראת ASLR, שמממשת Address Space Randomization technique. [7]  
אם Non-Executable Memory נועד להקשות על התוקף להריץ קוד שרירותי, אזי שיטה זו נועדה למנוע מהתוקף לגרום לתוכנה לקפוץ אל קוד לגיטימי, אבל עדיין לבחירתו (למשל אל ספריות טעונות).  
שיטה זו דואגת לטעון חלקים שונים בזיכרון של התכנית לכתובות אקראיות.  
לכן אם התוקף מנסה להריץ קוד של ספריה חיצונית מסוימת, עליו לנחש היכן ספריה זו טעונה בזיכרון.  
זו פעול קשה, אך עם זאת עדיין לא בלתי אפשרית

# חלק שני – תוכנת שרת בשפת פייתון

## שפת פייתון תכונות ומרכיבים

פייתון היא שפת תכנות *עילית* לא-ספציפית, שנועדה לאפשר כתיבת תוכנות אשר הינן *קריאות וקצרות*. [8]

עבור הרבה מתכנתים ומנהלי רשתות שפת התכנות פייתון משמשת כשפת *סקריפטים*, כלומר כותבים בה תוכנות קטנות וקצרות אשר מיועדות לביצוע פעולות פשוטות ובדידות.  
למשל, הרבה מהפעולות אשר מבצעות מניפולציה על קבצים, תוכן של קבצים ו-File System עברו להיכתב מ-bash לפייתון.

השפה נהגתה ונכתבה בשנות ה-80 על ידי Guido Van Rossum כאשר גווידו המשיך לתפקד כמפתח ליבה ומוביל של הקהילה וזכה לתואר “Benevolent Dictator For Life”.  
לאחרונה, ב-2018 אחרי תקופה ארוכה של הובלת הקהילה של שפת התכנות גווידו פרש מהתפקיד הלא רשמי, אך עדיין יישאר מעורב בקהילת השפה.

לעומת שפת C, פייתון היא שפה עילית אשר רצה בסביבת הרצה וירטואלית ומגיעה עם פיצ'רים תואמים כמו *ניהול זיכרון אוטומטי, ודינאמיות*.

פייתון רצה בסביבת הרצה וירטואלית, כלומר הקוד אינו מתקמפל ישירות לשפת מכונה אלא לשפת ביניים, bytecode.  
כאשר מריצים תכנית פייתון בעצם מריצים את סביבת ההרצה הווירטואלית ובה קיים ה-Interpreter אשר רץ סדרתית על כל statement בשפת הביניים, מתרגם אותו לפעולות הנכונות בשפת המכונה ומריץ אותן.  
ההשפעה של ריצה שכזו היא שתכנת פייתון יכולה לרוץ על כל מחשב אשר קיימת בו סביבת ההרצה של השפה, ללא צורך בקמפול התוכנית לארכיטקטורה הספציפית של המחשב.

ניהול הזיכרון האוטומטי בשפת פייתון מבוסס על השיטה של *Garbage Collection.*בשיטה זו משתמשים בסביבת ההרצה הווירטואלית כדי לשמור מעקב בזמן-ריצה אחר כל אובייקט שנשמר בזיכרון ואחר כל ההתייחסויות (references) אליו.  
כאשר כל ההתייחסויות אל אובייקט מסוים מסתיימות הרי שהסתיימה העבודה עם האובייקט והוא מנוקה מהזיכרון.  
באמצעות מנגנון זה המתכנת אינו צריך לאלקץ ולשחרר זיכרון עבור האובייקטים בתכנית שלו.  
כתוצאה מכך, מעבר להפיכת התכנות למשימה קלה יותר, אנו נמנעים מחולשות של memory corruption.  
החיסרון של ניהול הזיכרון האוטומטי הוא שהוא דורש סביבת ריצה ווירטואלית ולכן העלות שלו בביצועים ובצריכת זיכרון הינה גבוהה.

פייתון היא שפת תכנות דינאמית, לתכונה זו מספר משמעויות. [9]  
משמעות אחת היא שלעומת שפות כמו C++, בתוכנות פייתון אין צורך להצהיר על טיפוס המשתנה כאשר מגדירים אותו, או בחתימות של פונקציות.  
יתרון בתכונה זו הוא חיסכון משמעותי בקוד באמצעות הסרת הצהרות הטיפוסים של כל משתנה וארגומנטים וערכי החזרה בפונקציות.  
בנוסף, ניתן להגיע לאותה תוצאה כמו בתכנות גנרי בשפת C++ ללא הוספת תחביר מורכב אלא באמצעות *Duck Typing* שמשמעותו שכאשר מנסים לבצע פעולה על משתנה לא בודקים מה הטיפוס שלו, אלא רק אם הפעולה נתמכת על ידו.  
מקור השם Duck Typing הוא בסוג מסוים של הסקה לוגית שהתפרסם במשפט:

If it walks like a duck and it quacks like a duck, then it must be a duck

שפת פייתון נכתבה במטרה לאפשר למתכנת לכתוב קוד בכמה שיותר פשטות.  
אחת הדרכים שהשפה מקיימת מטרה זו היא באמצעות תמיכה מובנית בשפה במספר מבני נתונים נפוצים.

*List,* או רשימה בעברית היא מערך בגודל דינאמי (כלומר הגודל שלו יכול להשתנות לאחר אתחול) וזהו אחד ממבני הנתונים הנפוצים ביותר בעולם התוכנה.  
מאחר והרשימה היא מבנה נתונים נפוץ כל כך, הרשימה נתמכת בשפת פייתון בצורה יסודית וכחלק מתחביר השפה ומיוצגת על ידי סוגריים מרובעים המכילים רצף איברים מופרדים בפסיקים (לדוגמה: *[1, 2, 3]* היא רשימה המכילה את האיברים 1, 2 ו-3).

להלן דוגמת קוד אשר מכילה הגדרה של רשימה בצורות שונות וביצוע פעולות נפוצות עליה:

# Different ways of defining a list with items 0,1,2 and 3:  
a = [0, 1, 2, 3]  
b = list(range(4))  
c = [i for i in a]  
  
print(f'a: {a}\nb: {b}\nc: {c}')  
  
# Useful list operations:  
print(f'The list with reversed values: {reversed(a)}')  
print(f'The list [3,1,4,2,0] sorted: {sorted([3, 1, 4, 2, 0])}')  
print(f'Every item in the list in an odd index: {a[1::2]}')

*Dictionary (dict),* או מילון בעברית הוא מבנה נתונים מבוסס טבלאות גיבוב אשר מאפשר שמירה של מידע בתצורת מיפוי בין מפתח לערך (*key-value*) ואף הוא נפוץ ושימושי מאוד ולכן נתמך על ידי השפה ברמה יסודית הכוללת תחביר מיוחד לשימוש במבנה נתונים זה, עם הייצוג של סוגריים מסולסלות לאובייקטי מילון.

להלן דוגמת קוד אשר מכילה הגדרות של מילונים בצורות שונות וביצוע פעולות נפוצות עליהם בצירוף ייצוג של אובייקט מורכב, עץ בינארי, באמצעות מילון:

# Different ways of defining a dict from ABC letters to their ascii code:  
a = {'A': ord('A'), 'B': ord('B'), 'C': ord('C')}  
b = {letter: ord(letter) for letter in 'ABCDEFG'}  
c = dict([('D', ord('D')), ('E', ord('E')), ('F', ord('F'))])  
  
print(f'a: {a}\nb: {b}\nc: {c}')  
  
# Useful dict operations:  
print(f'\nKeys of b as list: {list(b.keys())}')  
print('Translation of key "B" to value: {}'.format(b['B']))  
  
print('\nBinary tree represented as a dict:')  
print({'root': {'value': 5, 'left': {'value': 6, 'left': None, 'right': None},  
 'right': {'value': 7, 'left': None, 'right': None}}})

שפת פייתון, כמו שפת C++ תומכת אף היא בפרדיגמת התכנות מונחה העצמים.  
כדי לתמוך בפרדיגמה זו קיים בשפה התחביר להגדרת טיפוסים חדשים בשפה באמצעות מחלקות ובנוסף להגדרת היררכיה וירושה בין המחלקות השונות.

להלן דוגמת קוד אשר מציגה הגדרת מחלקות בעלות היררכיה, אתחול מופעים שלהן ושימוש בפולימורפיזם:

class Animal: # Definition of a new Animal class:  
 def \_\_init\_\_(self, sound):  
 self.sound = sound  
  
 def talk(self):  
 print(f'{self.sound}')  
  
  
class Dog(Animal): # Definition of a Dog subclass:  
 # Definition of a class level constant  
 DOG\_SOUND = 'woof'  
  
 # Overriding super-class's constructor  
 def \_\_init\_\_(self):  
 # Calling super-class's constructor with value  
 super(Dog, self).\_\_init\_\_(sound=self.DOG\_SOUND)  
  
  
# Initializing instances of each class  
horse = Animal('Neigh')  
dog = Dog()  
animals = [horse, dog]  
  
# Using polymorphism to make all the animals talk  
for animal in animals:  
 animal.talk()

בצורה מעניינת, כל אובייקט בשפת פייתון מתנהג לפחות חלקית כמילון, המכיל את הטיפוס שלו ואת המיפוי בין שמות המאפיינים והפונקציות שלו אל הערכים עצמם.

## איומי אבטחה בשפות דינאמיות ושפת פייתון

### הזרקה דינאמית של משתנים

בשפות דינאמיות, וספציפית בשפת פייתון לא קיימת הגנה על משתנים והגדרות בחלק קוד אחד מחלקי קוד אחרים באותה התוכנה.  
למשל, אם הגדרנו מחלקה בקוד שלנו, וייבאנו ספרייה חיצונית ללא בדיקה קפדנית שלה, אנחנו חשופים למצב בו הספרייה החיצונית תשנה את אופן פעולתה של המחלקה שלנו!

להלן דוגמת קוד קצרה שמדגימה דריסה מבחוץ של התנהגות של מתודה פנימית של מחלקה:

# Defining a custom class:  
class MyString:  
 def \_\_init\_\_(self, message):  
 self.message = message  
  
 def \_\_str\_\_(self):  
 return f"The message is {self.message}"  
  
  
# Overriding an inner method of the class from outside:  
  
old\_str = MyString.\_\_str\_\_  
  
  
def new\_str(self):  
 print("Doing evil stuff!")  
 return old\_str(self)  
  
  
MyString.\_\_str\_\_ = new\_str  
  
# Evil stuff happens on normal usage  
a = MyString('hello!')  
  
print(str(a))

כמו שניתן לראות, בשימוש הרגיל במתודה str של המחלקה MyString נקרא כעת הקוד החיצוני לה ותודפס גם ההודעה:  
Doing Evil Stuff!  
אם חבילה חיצונית שתוכנת פייתון משתמשת בה הינה זדונית, ניתן היה להכניס התנהגות אשר תדליף מידע פנימי של התכנה החוצה.

### התמודדות עם הזרקה דינאמית של משתנים

למעשה אין התייחסות רשמית לתכונה זו של שפות דינאמיות כחולשה, מאחר ולמתכנת קיימת האפשרות והאחריות לבדוק לעומק את הספריות אשר הוא משתמש בהן בקוד שלו ובנוסף להימנע משימוש בספריות ממקור לא אמין.  
מאחר ופייתון אינה שפה מתקמפלת, הרי שהמתכנת יכול להסתכל על קוד המקור של הספריות אשר הוא משתמש בהן בתוכנה שלו. [8]

בנוסף, למרות שאין הבטחה כי בספריות פופולאריות לא יהיה קוד זדוני, ניתן להיות יחסית בטוחים כי ספריות פופולאריות אשר זכו לשימוש רב וזוגות רבים של עיניים עברו על קוד המקור שלהן יהיו בטוחות.

### חולשה במנגנון הסריאליזציה המובנה של פייתון – pickle [10]

סריאליזציה ודה-סריאליזציה הן בהתאמה הפעולות של ייצוג משתנים בזיכרון על ידי טקסט והשחזור של משתנים מתוך טקסט נתון.

המטרה של סריאליזציה היא העברת מידע בין תכנות אשר אינן רצות באותו התהליך ולכן אינן חולקות זיכרון משותף.  
(למשל מודולים שונים של מערכת הרצים על שרתים שונים, או העברת מידע מתקדם בין שרת ולקוח)

שפת פייתון, כמו שפות רבות אחרות מגיעה עם מנגנון סריאליזציה ודה-סריאליזציה מובנה.  
מנגנון זה הוא ספרייה הנקרית pickle.

עובדה ידועה, ואחת אשר אפילו מופיעה בתיעוד הרשמי של הספרייה היא שספריית pickle אינה מאובטחת וכאשר משתמשים בה לדה-סריאליזציה של מידע ממקור לא אמין היא עלולה להוביל להרצה של קוד שרירותי.  
(אפילו אני, הכותב של הסמינר, בתרגיל כניסה למשרה קיבלתי את המשימה לכתוב ניצול להרצת קוד מרוחק שרירותי על שרת נתון המריץ pickle והצלחתי לעשות זאת במספר שעות)

דוגמה ליצירת מחרוזת pickle זדונית אשר בטעינה שלה תורץ הפקודה ls:

import pickle  
import subprocess  
  
  
class RunLs(object):  
 def \_\_reduce\_\_(self):  
 return subprocess.Popen, (('/bin/ls',),)  
  
  
malicious\_pickle\_string = pickle.dumps(RunLs())  
  
print(malicious\_pickle\_string)  
  
pickle.loads(malicious\_pickle\_string)

### התמודדות עם החולשה במנגנון הסריאליזציה המובנה pickle

ההתמודדות המוצעת בתיעוד הרשמי של הספרייה היא פשוט לא לבצע דה-סריאליזציה של מחרוזות pickle ממקורות לא מוכרים או לא אמינים.  
כלומר, יש לשמור את השימוש במנגנון זה לשימוש פנימי בתוך רכיבים של המערכת, ולא לבצע בו שימוש לצורך תקשורת מול גורם חיצוני.  
בנוסף, יש להיזהר ולשים לב היכן שומרים מחרוזות pickle ובאיזה תווך שולחים אותן מהחשש של התערבות גורם חיצוני במחרוזות המאוחסנות או הנשלחות, בין אם בתקיפת *man in the middle* או באמצעות גישה למקום המשמש לאגירת המחרוזות (מסד נתונים או דיסק).

צורת התמודדות נוספת היא שימוש במנגנוני סריאליזציה אחרים, בטוחים יותר.  
למשל, שפת פייתון תומכת בצורה מובנית גם במנגנון סריאליזציה מעל פורמט *JSON (JavaScript Object Notation).* [11]באמצעות פורמט זה ניתן לייצג אובייקטים מטיפוסים מובנים כמו מחרוזות, מספרים, בוליאניים, רשימות ומילונים.  
כלומר, פורמט זה הוא מעט חלש יותר כאשר משתמשים בו מתוך פייתון מאחר ואובייקטים מטיפוסים שאינם הטיפוסים המובנים והבסיסיים לא ניתנים לסריאליזציה בצורה מידית.  
עם זאת, מאחר וכל האובייקטים בשפת פייתון הם מבוססי מילון, ניתן בעלות לא גבוהה במיוחד להעביר את רוב האינפורמציה הרצויה באמצעות פורמט JSON.

# חלק שלישי – תקשורת בין הרכיבים השונים

## תקשורת רשתית בסיסית

לצורך תקשורת בין הרכיבים השונים במערכת ביזור המשימות שלנו אנו צריכים שהם יתקשרו מעל רשת IP.

מערכות הפעלה מודרניות מאפשרות לנו לייצר חיבורים מעל רשתות IP בצורה הבסיסית ביותר באמצעות אובייקטי *socket*. [12]

אובייקטים אלו מהווים חיבור נקודה-לנקודה בין שתי כתובות ברשת באמצעות אבסטרקציה מעל רכיבי הניתוב השונים הקיימים ברשת.

כל אחד מהרכיבים שלנו מזוהה ברשת באמצעות כתובת המכילה שני חלקים:

1. כתובת *IP* אשר מזהה את המחשב עליו רץ הרכיב
2. *Port* אשר משמש כמזהה של "השירות" המבוקש מעל אותו המחשב, כאשר אנחנו נציין את רכיב השרת שלנו בפניות מהלקוח

במערכת ביזור המשימות שלנו אנו נשתמש בפרוטוקול *HTTPS* המוכר גם בשם *HTTP Over SSL* לצורך תקשורת בין הלקוח לשרת. [13]

פרוטוקול זה הוא הרחבה מאובטחת של פרוטוקול *HTTP (Hyper Text Transfer Protocol)*, כאשר ההבדלים ביניהם יוסברו בהמשך תחת סעיף איומים בתקשורת רשתית.  
נתחיל בתיאור של HTTP.  
פרוטוקול זה הוא פרוטוקול תקשורת במודל *בקשה-תגובה* מעל מודל *שרת-לקוח*.  
כלומר, כאשר משתמשים בפרוטוקול אחד מהצדדים מוגדר בתור השרת, אליו פונים מספר כלשהו של לקוחות ושולחים אליו בקשות.  
לכל בקשה השרת מגיב בתגובה כלשהי, אשר מציינת את הפעולה שביצע השרת בתגובה (אם בכלל!) ואת תוצאת הפעולה.

פרוטוקול זה מתאים לצורכנו מאחר ויש לנו שרת אשר ממתין לבקשות מלקוחות, כל לקוח שולח אל השרת קוד שברצונו להריץ והשרת שולח בתגובה את תוצאת הרצת הקוד.

## איומים בתקשורת רשתית

### התחזות ו-Man in the Middle

תקיפות התחזות הן תקיפות בהן נעשית פניה רשתית מגורם א' אל גורם ב', אך הפניה מגיעה אל גורם ג' זדוני במקום, אשר מתחזה אל גורם ב' ומתנהל מול גורם א'.

תקיפות מסוג זה יכולות להתבצע ברמה רשתית ואף ברמה חברתית.  
תקיפות התחזות ברמה החברתית נפוצות מאוד בצורה הנקרית *Fishing* כאשר גורם זדוני מתחזה אל גורם אמין ופונה אל אנשים במטרה לגנוב מהם מידע או לגרום להם לבצע פעולות כרצונו.  
אנחנו לא נתמקד בתקיפות חברתיות מאחר והמערכת שלנו מורכבת מרכיבי תוכנה.

תקיפת התחזות רשתית יכולה להתבצע במגוון רב של צורות.  
צורה נפוצה של תקיפות התחזות רשתית היא באמצעות "חטיפת" בקשות ARP או DNS ומענה עליהן באמצעות מידע שקרי אשר מוביל את הנתקף לפנות את הגורם הזדוני במקום אל השירות המבוקש על ידו.

מאחר והרכיבים במערכת שלנו מתקשרים מעל הרשת זהו ציר תקיפה שאנחנו חשופים אליו.

### התמודדות עם תקיפות התחזות

התמודדות אפשרית עם תקיפות התחזות היא באמצעות דרישה מהשרת אשר פונים אליו להוכיח את אמיתות זהותו.  
אחת הדרכים לדרישת הוכחה היא באמצעות דרישה מהשרת לספק *סרטיפיקט*, אשר מכיל את המידע על זהותו ומאושר באמצעות גורם אשרור מוכר (*Certificate Authority*).  
קיימים מספר גורמי אשרור אשר מוכרים על ידי כמעט כל הדפדפנים ומערכות ההפעלה בעולם.

אותו מסמך אישור צריך להכיל מעבר למידע על השרת גם אמצעי לאתגר את זהות השרת.  
בפרוטוקול HTTPS הדרך בה מאותגר השרת היא שחלק מהסרטיפיקט המאושר שלו מכיל מפתח פומבי במנגנון הצפנה א-סימטרי. [14]  
באמצעות שימוש במפתח הפומבי הזה, הלקוח אשר מעוניין לאמת את זהות השרת יכול להצפין הודעה שרירותית ולאתגר את השרת לפענח אותה.  
מאחר ורק מי שמחזיק במפתח הפרטי המתאים לאותו המפתח הפומבי יכול לפענח את אותה ההודעה, הרי שאם השרת הצליח לפענח את ההודעה הרי שהוא מחזיק בפרט מידע אשר נמצא אך ורק בידיו של מי שקיבל את המסמך החתום מהגורם אשר הלקוח סומך עליו.

זוהי כמובן גרסה פשוטה ולא שלמה של מודל האמון הנהוג בפרוטוקולים הללו, והגרסה המלאה והחסינה יותר מורכבת משמעותית יותר.

### גניבת מידע וריגול

מאחר והרכיבים השונים במערכת שלנו מתקשרים רשתית, ומאחר וברשתות IP עבור רכיבים רשתיים שאינם מחוברים ישירות התקשורת עוברת דרך רכיבי ביניים, הרי שבאמצעות השתלטות על אחד מרכיבי הביניים בתקשורת בין הרכיבים במערכת שלנו, ניתן להאזין לתקשורת בין ברכיבים שלנו "ולגנוב" מידע שעשוי להיות רגיש.

למשל, במידה והמערכת שלנו משתמשת בסיסמה על מנת לזהות את הלקוחות מול השרת, אנו בסכנה של גניבת הסיסמה ושימוש במערכת על ידי לקוחות לא מורשים. [15]

### התמודדות עם גניבת מידע

התמודדות שאינה פרקטית במקרה שלנו אך הינה אידאלית במקרים אחרים עם גניבת מידע במהלך תעבורה רשתית היא להקים חיבור נקודה-לנקודה פיזי בין כל רכיבי המערכת.

למשל, במערכות צבאיות נהוג לחבר בין בסיסים באמצעות סיבים ייעודיים וישירים אשר אינם נמצאים בשימוש על ידי גורמים שאינם מתוך הארגון.  
כך, ניתן להיות בטוחים יותר כי אין גורם שלישי אשר מאזין לתקשורת בין שני הבסיסים.

במקרה שלנו, לא נקים תשתית פיזית חדשה לצורך תקשורת בין רכיבי המערכת שלנו.  
לכן עלינו להניח כי התקשורת בין הרכיבים ניתנת להאזנה על ידי גורם שלישי.

הדרך להתמודד עם האיום של גניבת מידע במקרה זה היא באמצעות *הצפנת* התקשורת.  
הצפנה היא שיטה לחלוק מידע בצורה מאובטחת מעל תווך שאינו מאובטח באמצעות מניפולציה של המידע העובר בתווך הגלוי.  
שיטות הצפנה קיימות אלפי שנים, החל משיטות הצפנה עתיקות כמו אתב"ש אשר כוללת החלפה של אותיות הא"ב במסר המוצפן באות ההפוכה בסדר הא"ב ועד שיטות מודרניות אשר מבוססות על מניפולציה מתמטית של הייצוג הבינארי של ההודעה.

כמו שציינו מקודם, באמצעות הסרטיפיקט החתום של השרת הלקוח יכול לראות את המפתח הפומבי של השרת במנגנון הצפנה א-סימטרי כלשהו.  
אבל, הצפנה א-סימטרית הינה פעולה איטית ומורכבת ולכן לא נהוג להשתמש בה לצורך הצפנת תקשורת ארוכה.  
לכן, בפרוטוקול HTTPS משתמשים במפתח הפומבי הידוע כדי לתאם בצורה מאובטחת מפתח חדש שישמש להצפנת יתר התקשורת בין השרת והלקוח באמצעות מנגנון הצפנה סימטרית.  
פרוטוקול HTTPS תומך במגוון רחב של מנגנוני הצפנה סימטרית, ויבחר באחד מהם על פי מידע על המנגנונים בהם תומכים השרת והלקוח. [14]

# חלק רביעי – איומי אבטחה מערכתיים

## איום ראשון – מניעת שירות (/מבוזרת) D/DoS

תקיפת מניעת שירות הינה ניסיון מכוון של תוקפים למנוע ממשתמשים לגיטימיים גישה/שימוש בשירות רשתי מסוים. [15]

דרכים שונות לבצע תקיפות מניעת שירות הן:

1. להציף את השרת בבקשות, כך שמשאבים קריטיים לשירות לא יהיו פנויים בכדי לשרת לקוחות לגיטימיים.
2. לשלוח אל השרת בקשות לא סטנדרטיות, שנבנו במטרה להקפיא את השירות או לגרום לשירות או אפילו לשרת לקרוס.

ישנן עוד דרכים לבצע תקיפות מניעת שירות, ולמעשה זוהי תקיפה שמוגדרת על ידי מטרתה ולא על ידי הטכניקות הטכנולוגיות לבצעה.

וריאציה נוספת של תקיפות מניעת שירות היא תקיפת מניעת שירות מבוזרת (***Distributed*** *Denial of Service*) אשר מיוחדת בכך שמשתמשים במספר רב מאוד של מחשבים לצורך הצפת השרת הנתקף בבקשות.

במקרים רבים, התוקפים משתמשים ב-Bot net, אוסף של מחשבים אשר נפרצו בשלב מוקדם יותר על ידי התוקפים, ומוסיפים את אוסף המחשבים הללו למחשבים אשר שולחים בקשות לשרת הנתקף.

### התמודדות עם תקיפות מניעת שירות

שיטה אחת להתמודד עם תקיפות מניעת שירות היא באמצעות החזקת כפילות של תשתיות.  
כך, אם תשתית אחת נמצאת תחת התקפה וזמינותה יורדת, ניתן יהיה להעלות תשתית אחרת במקומה ולשמור על הזמינות של השירות.

שיטה נוספת להתמודד עם תקיפות מסוג זה היא באמצעות זיהוי התקיפה ברכיב רשתי אשר קודם לרכיב הרגיש במערכת. [15]  
לדוגמה אם במערכת ישנו שרת Web אשר מבצע פעולות מורכבות בתגובה לבקשות, אם ניתן יהיה לבלום את הבקשות הלא לגיטימיות עוד ברכיב הקודם לו, רמת הפגיעה תהיה משמעותית נמוכה יותר.

כמובן שניתן גם ברמת השירות לזהות בקשות לא לגיטימיות ולהתעלם מהן, אך זה מסובך יותר ופוגע בעקרון של הפרדת סמכויות במערכת.

## איום שני – חולשות במערכת ההפעלה

עד כה התמקדנו בעיקר בבעיות אבטחה שניתן לפתור באמצעות שינוי הקוד שלנו.

סוג נוסף של בעיות אשר לא עסקנו בהן עד כה הן בעיות מתוקף העובדה שאנחנו רצים מעל מערכת הפעלה שנכתבה אף היא על ידי מתכנתים ומשתמשים בפרוטוקולי תקשורת אשר נכתבו על ידי אנשים ומשתמשים בשיטות הצפנה שפותחו על ידי אנשים.

לכן, אנחנו חשופים לתקיפות אשר מכוונות לכל אחד מהרכיבים שאנחנו מתבססים עליהם, בין אם מדובר בתקיפה על פרוטוקול HTTPS או חולשה במנגנון ההצפנה.

מדי פעם בפעם, מתגלות חולשות במערכות הנפוצות אשר גורמות לכך שחלק משמעותי מהשירותים באינטרנט פגיעים לתקיפות.

חולשות מסוג זה מכונות תקיפות Zero Day מאחר וכאשר נעשה בהן שימוש לראשונה הן לא מוכרות לעולם. [16]

### התמודדות עם חולשות במערכת ההפעלה

הדרך הטובה ביותר להתמודד עם חולשות במערכות הפעלה היא לוודא שעושים שימוש בגרסאות החדשות ביותר של כל המערכות החיצוניות והפרוטוקולים אשר משתמשים בהם.

בנוסף, יש להשתמש אך ורק במערכות ומוצרים חיצוניים פופולאריים ונפוצים.  
זאת מאחר ולאחר שמתגלה חולשה במוצרים פופולאריים היא מתפרסמת במהירות ועדכון תוכנה אשר מונע אותה מתפרסם במהירות אף הוא.

אמצעי נוסף להימנע מתקיפות הוא הגדרה טובה של מערכות אבטחה מקומיות כמו פיירוול, תוכנות ניטור שונות של לוגים ופרמטרים דומים ועוד.

בנוסף, כדאי לכבות ולחסום כל שירות תוכנתי אשר לא נעשה בו שימוש.

# חלק חמישי – סיכום החלק התיאורטי והצגת החלק המעשי

## סיכום

לאורך מסמך זה פירטנו רכיבים שונים של מערכת תוכנה וכיצד רכיבים אלו מעמידים אותה בסכנה.

ראינו כי כאשר כותבים תוכנה, קיימים סיכוני אבטחה אשר יש להיות מודעים אליהם.  
ראינו כי למרות שאופי הפרצות ודרכי הניצול שלהן משתנים בין שפת תכנות לשפת תכנות, עדיין *גם* בשפות תכנות בעלות יכולות סף *וגם* בשפות תכנות עיליות ודינמיות קיימות סכנות מרובות.

בנוסף ניתן ללמוד מעבודה זו כי כאשר מוסיפים למערכת תוכנה אלמנטים תקשורתיים היא נחשפת למספר רב נוסף של סכנות.  
הסכנות שסקרנו בתחום זה היו בעלות אופי ריגול, גניבת מידע והזדהות.

מהפרק אשר עוסק באיומי אבטחה מערכתיים הראינו כי הקוד והמימוש של המערכת אינם המקורות היחידים לפרצות אבטחה במערכות תוכנה, וכי יש להוסיף הגנות סביבתיות למערכות התוכנה שלנו.

עם כל הידע שנצבר בעבודת סמינר זו, ניגשתי אל משימת כתיבת המערכת, כמתואר בסעיף הבא.

## החלק המעשי

בכתובת:

<https://github.com/AlonSh/seminar-in-defensive-programming>

מאוחסן git repository עם המסמך הנוכחי, הקוד שנעשה בו שימוש במסמך והקוד של הפרויקט (החלק המעשי) של הסמינר.

כתבתי כנדרש מערכת בעלת שני רכיבים אשר מאפשרת ביזור של הרצת משימות.  
המערכת כוללת שרת פייתון להרצת משימות מרוחקות אשר מקבל בקשות לפעולות חישוב ומחזיר את תוצאותיהן, ולקוח בשפת C++ אשר מבקש מהמשתמש קלט ושולח בקשות מתאימות לחישוב אל השרת.

בהתאם למסקנות החלק השלישי של האיומים הרשתיים דאגתי שהתקשורת בין השרת והלקוח תהיה מוצפנת באמצעות שימוש בפרוטוקול HTTPS.

בהתאם למסקנות החלק השני אני שולח את המידע מהלקוח אל השרת בפורמט JSON, אשר בשונה מפורמט הסריאליזציה העיקרי של פייתון אינו פגיע.

בהתאם למסקנות החלק הראשון, הקפדתי לוודא את תקינות הקלט מהמשתמש ולהשתמש בקריאות ספריה מודרניות לניהול העתקת זיכרון.

# Bibliography

|  |  |
| --- | --- |
| [1] | B. Stroustrup, The C++ programming language, Pearson Education India, 2000. |
| [2] | S. B. Lippman, C++ Primer, Pearson Education India, 2005. |
| [3] | B. Smith, ""Object-oriented programming." AdvancED ActionScript 3.0: Design Patterns.," *Apress,* pp. 1-25., 2011. |
| [4] | P. Buchlovsky and A. Butcher., ""BUFFER OVERFLOW VULNERABILITIES."". |
| [5] | D. Larochelle and E. David, ""Statically detecting likely buffer overflow vulnerabilities."," 2001. [Online]. |
| [6] | L. e. a. Szekeres, ""Sok: Eternal war in memory."," in *Security and Privacy (SP), 2013 IEEE Symposium on*, 2013. |
| [7] | R. Hund, C. Willems and T. Holz, ""Practical timing side channel attacks against kernel space ASLR."," in *2013 IEEE Symposium on Security and Privacy.*, 2013. |
| [8] | G. Van Rossum, ""Python Programming Language."," in *Annual Technical Conference*, 2007. |
| [9] | L. D. Paulson, ""Developers shift to dynamic programming languages."," *Computer,* p. 40.2, 2007. |
| [10] | M. Slaviero, ""Sour Pickles."," [Online]. |
| [11] | D. Crockford, "The application/json media type for javascript object notation (json).," *No. RFC 4627.,* 2006. |
| [12] | K. R. a. W. R. S. Fall, TCP/IP illustrated, volume 1: The protocols, addison-Wesley, 2011. |
| [13] | E. Rescorla, "Http over tls," *No. RFC 2818,* 2000. |
| [14] | E. Rescorla, "Http over tls," *No. RFC 2818,* 2000. |
| [15] | J. a. P. R. Mirkovic, ""A taxonomy of DDoS attack and DDoS defense mechanisms."," *ACM SIGCOMM Computer Communication Review 34.2,* pp. 39-53, 2004. |
| [16] | L. e. a. Wang, ""k-zero day safety: Measuring the security risk of networks against unknown attacks."," in *European Symposium on Research in Computer Security.*, Springer, Berlin, Heidelberg, 2010. |
| [17] | D. A. S. a. B. W. Boneh, ""Functional encryption: Definitions and challenges."," in *Theory of Cryptography Conference.*, Springer, Berlin, Heidelberg, 2011. |