****

**שם התלמיד: הדר רבין**

**ת"ז: 206448367**

**מנחה: מיכאל צ'רנובילסקי**

**בית הספר: תיכון חדש הרצליה**

**כיתה: י"ב**

**מאי 2015**

**תוכן עניינים**

[תקציר כולל ורציונל הפרויקט 3](#_Toc418019495)

[מבוא ורקע כללי 4](#_Toc418019496)

[תיאור המוצר המוגמר 6](#_Toc418019497)

[שפת התכנות וסביבת העבודה 7](#_Toc418019498)

[ניסוח וניתוח הבעיה האלגוריתמית 8](#_Toc418019499)

[תיאור אלגוריתמים קיימים 9](#_Toc418019500)

[הפתרון הנבחר 11](#_Toc418019501)

[פיתוח פתרון הקוד עם שפת התכנות 14](#_Toc418019502)

[תיאור המודולים המרכיבים את המוצר 19](#_Toc418019503)

[פירוט מבני הנתונים 12](#_Toc418019504)

**הקדמה**

מטרת המסמך היא להציג בפירוט את הפרויקט ולהסביר את החלקים השונים הנמצאים בפרויקט.

מסמך זה מיועד לקוראי המסמך, למשתמשי הפרויקט ולכל אדם המעוניין ללמוד על הפרויקט.

# תקציר כולל ורציונל הפרויקט

בעת השימוש במחשב ישנם מספר משאבים עיקריים שמנוצלים – זמן מעבד (CPU), זיכרון (RAM) שימוש בדיסקים קשיחים ועוד. שימוש מופרז באחד או יותר ממשאבים אלה מצביע לעתים קרובות על פעילות בעייתית המתרחשת במחשב, שיכולה להיגרם כתוצאה מהרצת תוכנה זדונית או כתוצאה משימוש לא נכון במחשב (למשל הרצת קוד עם דליפות זיכרון או הרצת קוד עם חישובים כבדים הפועלים ברציפות). בכל מקרה, שימוש מופרז במשאבי המחשב הוא מצב שגורם למחשב להיות איטי או להיתקע לגמרי, זה מונע מהתוכנות במחשב לרוץ כראוי ולכן אין ספק שמצב כזה דורש התייחסות וטיפול.

הרצון שלי לפתח תוכנה שתנטר את השימוש במשאבי המחשב ובעת הצורך תוכל להצביע על דליפות זיכרון, הרצת קודים לא נכונים ואף ךפעול באופן אקטיבי לסגירתם נבעה מהניסיון הרב שלי בתכנות. לעתים קרובות כתבתי תוכנות ששגיתי בהערכתי לגבי שימושן במשאבי המחשב, וכתוצאה מכך המחשב הפסיק להגיב(קרס). כאשר מדובר על רשת מחשבים של חברה אין ספק שהחשיבות עולה פי עשרות מונים וכדי לנסות ולמנוע את התקלות מבעוד מועד נדרש להריץ תוכנה אשר ביכולתה לפקח על השימוש במשאבי המחשב ולמנוע או\ו להתריע על שימוש מופרז.

התוכנה שלי אמורה לספק בקרה על משאבי המחשב ע"י שליטה מרחוק, היא תאסוף מידע לגבי השימוש במשאבים של המחשב ותיתן למנהל הרשת התראות בעת זיהוי פעילות חריגה באחד מההבאים: שימוש במעבד,שימוש בזיכרון RAM, שימוש בדיסקים, ומידע כללי לגבי המערכת. וכן תתערב ממש בפעילות המחשב ותסגור תהליכים שצורכים יותר מדי משאבים ואף תתריע על כך למנהל הרשת.

# מבוא ורקע כללי

הפרויקט שלי מבוסס על **שליטה מרחוק** (**Remote Control**). בין מחשבי הרשת השונים נוצר חיבור עם השרת המרכזי, וחיבור שכזה מאפשר שליטה מרחוק עליהם (שליטה מוגבלת, בהתאם לאפשרויות שהתוכנה מספקת). שליטה זו מאפשרת איסוף מידע ממחשבי הקצה והרצת פקודות עליהם (כגון סגירת תהליכים), וזהו עיקר הבסיס למודול התקשורת במיזם זה.

הנושא המרכזי של המיזם הוא **ניתוח משאבי מערכת והשימוש בהם באופן דינאמי**. למערכות המחשב יש סוגים שונים של משאבים, כגון: זמן מעבד, זיכרון פיזי, דיסק קשיח, תעבורת רשת ועוד. את המשאבים חולקים ביניהם כל התהליכים הרצים במערכת.

תוכנה זו עוסקת בהצגת הנתונים הבסיסיים של המערכת למנהל, כמו כן בניתוח של נתוני השימוש במשאבי המערכת הבאים:זמן מעבד, זיכרון פיזי ושימוש בדיסקים הקשיחים.התוכנה מנטרת הן את נתוני השימוש הכולל במשאבי המערכת, והן את נתוני השימוש של תהליכים מסוימים בפני עצמם במשאבי המחשב.

מטרת ניתוח הנתונים היא לזהות תהליכים שגוזלים יותר משאבים מהרף שנקבע לגביהם וכך פוגעים בתפקודו שלא לצורך.

לפרויקט יש לא מעט מרכיבים, אולם העיקריים שבהם מוצגים לפניכם כאן:

* מודול עיקרי בפרויקט שלי הוא מודול ה**תקשורת**. תקשורת בין מחשביתם מתבצעת היום לרוב באמצעות Sockets, שהם נקודות קצה עבור זרם נתונים בתקשורת בין תהליכים על גבי רשת מחשבים. רוב הסוקטים כיום מבוססים על ה- Internet Protocol (IP).

בפרויקט שלי התקשורת בין השרת ללקוחות מתבצעת באמצעות Internet Sockets, כאשר בשכבת התעבורה יש שימוש בפרוטוקול ה- TCP, שייחודו בכך שהוא מוודא שכל הנתונים שנשלחו התקבלו ובסדר הנכון, ואחרת שולח אותם שוב. בפרויקט שלי יש חשיבות לכך שלא יהיה איבוד מידע ופרוטוקול ה- TCP/IPמתאים בדיוק למטרה זו.

* מרכיב חשוב נוסף בפרויקט הוא קבלת מידע לגבי תהליכים. המודול האחראי על קבלת המידע לגבי התהליכים הרצים במחשב. משתמש ב- API של windows. עם שלל הפונקציות שהתוכנה משתמשת בapi הזה, התוכנה מקבלת את הנתונים הרצויים שהיא צריכה כדי להגיע להחלטות לגבי תהליכים (האם חשודים או לא). הנתונים כוללים: ניצול מעבד, ניצול של תהליך מסויים מהמעבד, ניצול זיכרו, ניצול תהליך מסוים מהזיכרון, ניצול של כל כונן (ועוד כמה פרטים על כל כונן), איזה תהליכים רצים, סוג מערכת הפעלה, שם המחשב ועוד…
* בנוסף חשוב לציין שהפרויקט שלי כולל גם מודול המאפשר לשרת לתקשר באופן בטוח עם הלקוחות. המודול אחראי על הצפנת המידע העובר ומתן הכלים למודול התקשורת לבצע את החלפת המפתחות הכפולה. החלפת המפתחות תתואר באיור והסבר על הפרוטוקול בהמשך.



# תיאור המוצר המוגמר

שם המוצר: Network Managment.

המוצר אמור לבצע ניטור ובקרה על השימוש במשאבים של מחשבים ברשת מקומית, לזהות פעילות חריגה של שימוש מופרז במשאבים, ולטפל בה. המוצר למעשה נותן למנהל הרשת כלי לפיקוח על הנעשה במחשבים שברשת האינטרנט.

המטרה המרכזית של המוצר היא מניעת עומס על המחשבים הקיימים ברשת ,על ידי זיהוי וטיפול בפעילות חריגה (העלולה להעיד על וירוסים במחשב). המוצר מבצע זאת ע"י ניטור ובקרה על מחשבי הרשת באמצעות חיבור לשרת מרכזי. הוא מקבל אליו מידע שוטף לגבי השימוש במשאבים של כל אחד מהמחשבים ומאפשר הגבלה ובקרה על השימוש במשאבים באמצעות התראות, סגירת תהליכים ואפשרויות נוספות. המטרה היא לאפשר למנהל הרשת לפקח על הנעשה במחשבי הרשת – לקבל התראות לגבי פעילות חריגה ולפקח עלת וכנה שמטרתה לשמור על דרך פעולה תקינה של המחשבים ברשת באופן אוטומטי .

השאלות שהמערכת עונה עליהן:

1. כיצד להגן על המידע הרגיש מפני גורמים לא מורשים? התוכנה מצפינה את כל התקשורת בין מחשבי הרשת לשרת, כך שגורם לא מורשה לא יוכל להפיק שום תועלת מהאזנה למידע הנשלח, הוא לא יוכל לגלות אילו תהליכים רצים על מחשבי החברה וכמה משאבים הם צורכים ולא אף מידע אחר הנשלח אל השרת.
2. כיצד לעבוד בו-זמנית עם כל המחשבים ברשת? כאשר מחשבי הרשת מתחברים לשרת, הוא פותח Thread נפרד לכל אחד ומסוגל לתקשר במקביל עם כולם, לקבל מהם מידע ולשלוח להם פקודות, ולכל מחשב ברשת יש בצד השרת מסד נתונים נפרד שבו מאוחסן כל המידע שנאסף עליו.



# שפת התכנות וסביבת העבודה

הפרויקט נכתב ב- Python וב- C#. כל הליבה של הפרויקט נכתבה ב- Python בשל הפשטות והנוחות של השפה, וה- GUI של הפרויקט נכתב ב- C# בעזרת ה- Visual Studio ואז לחבר אותו לליבה שנכתבה ב- Python בעזרת pipes.

סביבת העבודה עם Python הייתה בתחילה IDLE אך מהר מאוד עברתי ל- PyCharm שהיא תוכנה הרבה יותר מקצועית ונוחה לעבודה על פרויקטים מרובי מודולים וקבצים. היא שימשה גם לפיתוח הקוד ועבודה במקביל עם כל המודולים השונים, וגם להרצתו בשביל בדיקות.

סביבת העבודה עם C# הייתה Microsoft Visual Studio 2010 ששימשה לכתיבת ה- GUI באמצעות גרירת פקדים מה- toolbox, וה- debugger המובנה שימש לבדיקות. הבחירה בסביבת עבודה זו הייתה משום שה- Visual Studio הוא IDE טוב מאוד לעבודה ב- C#.



# ניסוח וניתוח הבעיה האלגוריתמית

במהלך פיתוח המיזם נתקלתי ועודני נתקל בבעיות רבות, שחלקן אלגוריתמיות ולהלן:

* מימוש שרת מולטי-קליינט:

הבעיה הראשון שעמדה לפניי בפיתוח המוצר היא יצירת שרת מולטי-קליינט. כלומר, הייתי צריך לממש שרת שמתחברים אליו מספר לקוחות בו-זמנית והוא מסוגל לדבר עם כולם במקביל. בתוך בעיה אלגוריתמית זו ישנן מספר תתי-בעיות:

* פיתוח פרוטוקול התקשרות: השרת צריך לתקשר עם הלקוחות בדרך מסוימת, שתקבע את אופן ההתנהלות ביניהם שבשגרה. כדי לאפשר זאת היה עליי לפתח פרוטוקוך שמגדיר את ההתקשרות ביניהם ולכתוב קוד בהתאמה לכל אחד מהצדדים.
* השמת דגש על אבטחת המידע: כדי למנוע מהמידע לעבור בין השרת ללקוחות באופן חשוף הייתי צריך לממש אבטחת מידע, כלומר לגרום למידע המועבר להיות קריא אך ורק לצד השולח ולצד המקבל
* חיבור עם ה- GUI : השרת צריך להתחבר ל- GUI לשם הצגת מידע והתראות למנהל באופן גרפי.
* השגת מידע על כל התהליכים במחשב:
* **השגת ניצול המעבד והשגת ניצול של כל תהליך מהמעבד**:
* זהו הבעיה הראשונה שנתקלתי בה ובין הקשות שנתקלתי בפרויקט כולו. הבעיה הזאת דרשה מחקר של כמה מים. אחרי מחקר קטן הבנתי שאין פונקצייה ספציפית שעושה זאת, ולכן הייתי צריך לחשוב עם עצמי איך להשיג את הנתונים האלו.
* **השגת התהליכים הרצים במחשב**:
* היה לי קשה בהתלחה למצוא את התהליכים הרצים במחשב. הבעיה המרכזית הייתה **עידכון** התהליכים הרצים, שכאשר שתהליך נסגר צריך לעדכן את רשימת התהליכים בתוכנה או שתהליך נוצר צריך לעדכן את רשימת התהליכים בתוכנה.כמו כן לא קיים קוד בpython שמאפשר להשיג נתוני מערכת ולכן נאלצתי לכתוב אחד בעצמי באמצעות חבילות שאפשרו לתכנת בC.
* בעיה נוספת שהפריעה רבות במימוש שרת המולטי כליינט, היא סינכרון מספר לקוחות שיכולים לשלוח מידע לגוי באופן אסינכרוני, בעוד שהגוי לא יוכל להבדיל ביניהם. לצורך כך בכל פעולה שוטפת של הגוי הוא תמיד מקבל הודעה שמכילה את כתובת הip ולאחריה נקודתיים. בפורמט הבא: 127.0.0.1:38., וכך הגוי יודע האם המידע שהגיע תואם את הIP המבוקש.

# תיאור אלגוריתמים קיימים

* **מימוש שרת מולטי-קליינט:**

המימוש המקובל של תקשורת בין מחשבים הוא באמצעות Sockets, ואולם ישנם מספר סוגים:

* Datagram Socket – מכונה גם connectionless, משתמש בפרוטוקול UDP בשכבת התעבורה ומשמש להעברת מידע במהירות רבה תוך ויתור על הבטחת data integrity.
* Stream Socket – מכונה גם connection-oriented, משתמש בפרוטוקול TCP בשכבת התעבורה ומשמש להעברת מידע תוך וידוא שכל הפקטות מגיעות ליעדן ומגיעות בסדר הנכון.

כמו כן, למימוש השרת כמולטי-קליינט שתומך בתקשורת עם מספר לקוחות בו-זמנית יש שתי שיטות המוכרות לי:

* באמצעות ספריית select – ב- Python יש ספרייה בשם select שבתוכה יש פונקציה עם אותו שם, והיא יכולה לקבל רשימה של Sockets שמהם נרצה לקבל מידע ורשימה של Sockets שנרצה לשלוח דרכם מידע, והיא מחזירה רשימה של Sockets שמוכנים לקריאת מידע מתוכם ורשימה של Sockets שמוכנים לשליחת מידע דרכם לצד השני.
* באמצעות multi-threading – יהיה Socket מאזין שיאזין ללקוחות חדשים שמתחברים ב- Thread הראשי (בלולאה אינסופית), ובכל פעם שמתחבר לקוח חדש ייפתח עבורו Thread חדש ובו תתנהל התקשורת עם לקוח זה. בצורה כזו השרת ידבר עם כל לקוח ב- Thread נפרד וכך יוכל לדבר עם כולם בו-זמנית. ספריית threading של Python מאפשרת לעשות זאת.

לשם אבטחת מידע קיימים אלגוריתמי הצפנה רבים, למשל בהצפנה אסימטרית יש את El-Gamal ואת RSA, ובהצפנה סימטרית יש DES ו- AES כמו כן יש פונקציות גיבוב רבות(hash) שמאפשרות לוודא אם המידע שהתקבל זהה לזה שנשלח.

* **השגת השימוש הכללי ושל כל תהליך ממשאבי העיבוד**

כדי לקבל מהמערכת את הנתונים הללו יש פרוצדרות קבועות שצריכות להתבצע. הפעולות האלה נמצאות בAPI של windows:

השגת השימוש הכללי במשאבי העיבוד:

1. יש להשיג את זמני המערכת, זמן הkernel, זמן הuser, וזמן הidle.
2. יש לקרוא לפונקציה הזו מספר פעמיים בהפרשי זמן קבועים, ולחשב את ההפרש בין הkernel הראשון והשני, ובין הuser הראשון והשני.
3. חיבור הפרש המשתמש וחיבור הפרש הליבה כדי להשיג את זמן המערכת עצמו שייקרא system time.
4. יש להציב בנוסחה הזאת את הפרמטרים: ((system\_time- idle\_time)/system\_time) במידה ורוצים לקבל את הביטוי כאחוזים יש להכפיל ב100% כי זהו היחס בין השימוש כרגע לבין השימוש האפשרי הכולל.

השגת השימוש של תהליך מסוים ממשאבי המעבדף

1. יצירת handle לתהליך שאנחנו רוצים לבדוק.
2. ביצוע פונקציות 2-3 רק עם פונקציית getprocesstimes.
3. לחבר את זמן התהליך במצב user וזמן התהליך במצב kernel, ולקבל את זמן התהליך במערכת.
4. process\_system\_time / system\_time התוצאה שהתקבלה היא מהווה את היחס בין השימוש של התהליך מהמעבד לכלל השימוש האפשרי במשאבי המעבד, לכן כדי לקבל תוצאה באחוזים יש להכפיל ב100%.

* **השגת מידע על כל התהליכים במחשב:**

זוהי הבעיה האלגוריתמית הראשית שבה נתקלתי בהכנת המיזם. אין כלי בשפת python שמאפשר להשיג את המידע שחיפשתי, כמו השגת התהליכים הרצים, או אחוז הניצול של המעבד, אחוז הניצול של כל תהליך מהמעבד, המקום הפנוי בכל דיסק, כמות זיכרון הRAM ועוד.

לכן נאלצתי לכתוב בעצמי קוד בpython אולם קוד בשפת תכנות נמוכה(התאפשר באמצעות חבילות מסוימות בpython) ולהשיג את הנתונים האלה ממש מהמערכת עצמה.

# הפתרון הנבחר

* **מימוש שרת מולטי-קליינט:**

את השרת מולטי-קליינט מימשתי באמצעות Stream Socket (בפרוטוקול TCP) שמאזין ב- Thread הראשי, ובכל פעם שמתחבר לקוח, נפתח ללקוח הזה Thread נפרד משלו שנשלח לפונקציית session with client שאחראית על כל ההתקשרות עם הלקוח באופן תמידי. בצורה כזאת השרת מסוגל לדבר בו-זמנית עם מספר לקוחות בת'רדים נפרדים וגם להמשיך להאזין לקבלת חיבורים חדשים. אחרי מחקר רב שכלל מידע על פונקציות Select ופונקציות דומות או אחרות שמאפשרות מימוש דומה, הגעתי להחלטה באופן עצמאי שהמימוש שיעבוד באופן מיטבי עם המיזם שלי יהיה Multi-Threading.

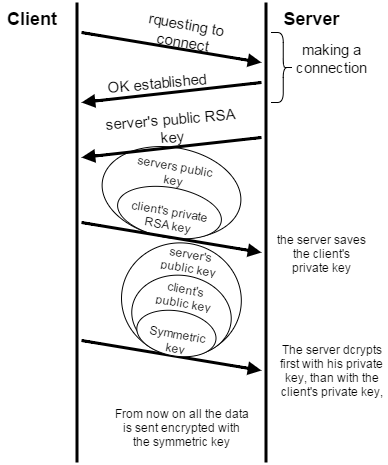
* **פיתוח פרוטוקול התקשרות בין השרת ללקוחות המאפשר לאבטח את העברת המידע:** פיתחתי פרוטוקול משלי בשביל התקשורת בין השרת ללקוח. ראשית כל מתחילים מפונקציית החלפת המפתחות. בפונקציה זו נעשה שימוש בפרוטוקול וידוא והצפנה כפול שפריצתו קשה פי עשרות מונים מפונקציית וידוא יחידה.

ראשית כל השרת והלקוח מייצרים כל אחד מפתח RSA ציבורי ופרטי, כידוע הציבורי משמש להצפנה והפרטי משמש לפיענוח, כמו כן הלקוח יוצר מפתח סימטרי מסוג AES שהמטרה הסופית היא שהלקוח והשרת יתקשרו אך ורק באמצעותו.

תהליך החלפת המפתחות מתחיל שהשרת שולח את המפתח הציבורי שלו אל הלקוח, וגם HASH של המפתח הזה, הלקוח אמור לקבל את שניהם וליצור HASH משלו למפתח הציבורי של השרת ולהשוות למפתח הציבורי של הלקוח. השלב הבא הוא שהמפתח שולח את המפתח הפרטי שלו מוצפן עם המפתח הציבורי של השרת, כמו כן הוא יוצר HASH ושולח את שניהם.

השרת מצדו מקבל את המפתח הפרטי של הלקוח מוצפן ומפענח אותו, יוצר לו HASH ומשווה לזה שקיבל. כרגע אנו במצב שבו לשרת יש את המפתח הפרטי של הלקוח, וללקוח יש את המפתח הציבורי של השרת. עכשיו הלקוח שולח את המפתח הסימטרי שלו מוצפן עם המפתח הציבורי שלו ומוצפן עם המפתח הציבורי של השרת. השרת מפענח ראשית את ההצפנה עם המפתח הפרטי שלו ואז עם המפתח הפרטי של הלקוח. כך מתבצעת העברת המפתחות.

ניתן לראות את כל השלבים שתוארו לעיל בתרשים הבא:



* **חיבור עם ה- GUI :** החיבור מתבצע באמצעות pipe כמתואר באיור הבא:

C:\Users\User\Downloads\Gui.png

* **השגת מידע על כל התהליכים במחשב:**

בניתי פונקצייה אשר מחזירה את רשימת התהליכים הרצים במחשב באותה שנייה שהתוכנה הופעלה.

קבלת התהליכים הרצים במחשב התבצעת ע"י 2 פוקציות עיקריות (ברגע שאחד הפונקציות לא עובדות הן מנסות את הפונקצייה השנייה). לפונקצייה הראשונה קוראים: EnumObjectItems שמחזירה רשימת שמות של התהליכים. כדי למצוא PID של כל תהליך הייתי צריך לערוך עוד מחקר ומצאתי שבספריית PDH ב-API של windows יש כמה פונקציות שעזרו לי עם זה.

הפונקצייה השנייה משתמשת ב-WMI של windows. אני מתחבר ל- WMI (באמצעות win32com.client) ולאחר מכן אני מקבל ממנו את המידע שאני צריך באמצעות שאילתות SQL מתאימות (במקרה שלנו זה שמות התהליכים והPID של כל אחד).

* **השגת הניצול של המעבד ואחוז הניצול של כל תהליך:**

לאחר מחקר של כמה ימים הגעתי לפתרון. נעזרתי רבות בפיתרון הקיים שהוצג בשפת- C (נעזרתי יותר ממקור אחד). כתבתי את הקוד ב- Python והצלחתי להשיג את הפרמטרים האלו, כדי להציב בנוסחה המפורטת להעיל ובהתאם להשיג את אחוז השימוש הנמוכחי בכל רגע נתון לגבי המעבד בכלליות, ושל כל תהליך מתוך משאבי המעבד.

# פיתוח הפתרון הקוד עם שפת התכנות

* **מימוש שרת מולטי-קליינט:**

הבסיס לשרת המולטי-קליינט הוא:

self.socket.bind((**'0.0.0.0'**, PORT))  
self.socket.listen(5)  
**while** True:  
 client\_socket, client\_address = self.socket.accept()  
 s = threading.Thread(target=self.SessionWithClient, args=(client\_socket,client\_address[0],))  
 self.writeTGui(**"Start new client"**)  
 s.start()

ה- socket מאזין ובכל פעם שמתחבר לקוח חדש הוא יוצר בשבילו socket ואני פותח בשבילו Thread חדש בפונקציית SessionWithClient שמאפשרת את המשך העבודה מול הלקוח.

* **פיתוח פרוטוקול משלי ומימוש אבטחת מידע:**

ראשית קבעתי את גודל ה- buffer והשתמשתי בפונקציות שמקבלות את הרשימה של הsocket והמםתחות, וההודעה, היא עוטפת את ההודעה בהצפנה ושולחת:

**def** sendTclient(self, csocket, data):  
 encrypted\_data = self.client\_keys[csocket].encrypt\_data(data)  
 csocket.send(encrypted\_data)  
  
**def** recvFclient(self, csocket):  
 encrypted\_data = csocket.recv(LEN\_UNIT\_BUF)  
 data = self.client\_keys[csocket].decrypt\_data(encrypted\_data)  
 **return** data

*# Encrypt the given data with the encryption cypher***def** encrypt\_data(self,data\_to\_encrypt):  
 **try**:  
 plaintext\_padded = AddPadding(data\_to\_encrypt)  
 encrypted = self.AESobj.encrypt(plaintext\_padded)  
 **return** b64encode(encrypted)  
  
 *# Catch any general exception* **except** Usage, err:  
 **print** >>sys.stderr, err.msg  
 **print** >>sys.stderr, **"for help use --help"  
 return** None  
*# Decrypt the given encrypted data with the decryption cypher***def** decrypt\_data(self,encrypted\_data):  
 **try**:  
 decoded\_encrypted\_data = b64decode(encrypted\_data)  
 decrypted\_data = self.AESobj.decrypt(decoded\_encrypted\_data)  
 **return** StripPadding(decrypted\_data )  
 **except** Usage, err:  
 **print** >>sys.stderr, err.msg  
 **print** >>sys.stderr, **"for help use --help"  
 return** None

**def** StripPadding(data):  
 **return** data.rstrip(PAD).rstrip(INTERRUPT)  
*# Pad your data before encryption (with pad and interrupt\_***def** AddPadding(data):  
 new\_data = **''**.join([data, INTERRUPT])  
 new\_data\_len = len(new\_data)  
 remaining\_len = BLOCK\_SIZE - new\_data\_len  
 to\_pad\_len = remaining\_len % BLOCK\_SIZE  
 pad\_string = PAD \* to\_pad\_len  
 **return ''**.join([new\_data, pad\_string])

* **חיבור עם ה- GUI :**

החיבור ל- GUI מתבצע באמצעות Pipe:

self.f = open(**r'\\.\pipe\myPipee'**, **'r+b'**, 0)

ושליחת המידע ל- GUI וקבלתו מתבצעות באמצעות הפונקציות הבאות:

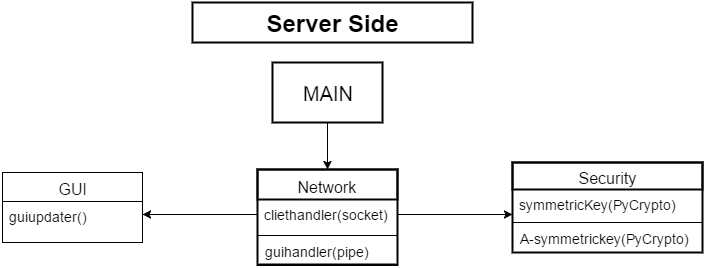
**def** writeTGui(self, s):  
 self.f.write(struct.pack(**'I'**, len(s)) + s)  
 self.f.seek(0)  
  
**def** readFGui(self):  
 n = struct.unpack(**'I'**, self.f.read(4))[0]  
 s = self.f.read(n)  
 self.f.seek(0)  
 **return** s

* **השגת מידע על כל התהליכים במחשב:**
* **def** get\_windows(self):  
   *"""  
   Return a list which contains the opened windows titles.* **:return***: title (list)  
   """* EnumWindows = ctypes.windll.user32.EnumWindows *# Filters all opened windows* EnumWindowsProc = ctypes.WINFUNCTYPE(ctypes.c\_bool, ctypes.POINTER(ctypes.c\_int),  
   ctypes.POINTER(ctypes.c\_int)) *# Callback function. Returns a tuple* GetWindowText = ctypes.windll.user32.GetWindowTextW *# Get all the window's title* GetWindowTextLength = ctypes.windll.user32.GetWindowTextLengthW *# Gets the exact buffer size of the title* IsWindowVisible = ctypes.windll.user32.IsWindowVisible *# Filters all the visible windows* titles = []  
    
   **def** foreach\_window(hwnd):  
   **if** IsWindowVisible(hwnd):  
   length = GetWindowTextLength(hwnd)  
   buff = ctypes.create\_unicode\_buffer(length + 1)  
   GetWindowText(hwnd, buff, length + 1)  
   titles.append(buff.value)  
   **return** True  
    
   EnumWindows(EnumWindowsProc(foreach\_window), 0) *# Callback* **return** titles
* **השגת הניצול של המעבד ואחוז הניצול של כל תהליך:**
* **def** GetSystemTimes(self):  
    
   *"""  
   Uses the function GetSystemTimes() (win32) in order to get the user mode time, kernel mode time and idle mode time* **:return***: user time, kernel time and idle time (Dictinary)  
   """* \_\_GetSystemTimes = windll.kernel32.GetSystemTimes  
   idleTime, kernelTime, userTime = FILETIME(), FILETIME(), FILETIME()  
    
   success = \_\_GetSystemTimes(  
    
   byref(idleTime),  
   byref(kernelTime),  
   byref(userTime))  
    
   **assert** success, ctypes.WinError(ctypes.GetLastError())[1]  
    
   **return** {  
   **"idleTime"**: idleTime.dwLowDateTime,  
   **"kernelTime"**: kernelTime.dwLowDateTime,  
   **"userTime"**: userTime.dwLowDateTime}  
    
  **def** cpu\_utilization(self):  
   *"""  
   Returns the total cpu usage  
    
   Source: http://www.codeproject.com/Articles/9113/Get-CPU-Usage-with-GetSystemTimes* **:return***: CPU usage (int)  
   """* FirstSystemTimes = self.GetSystemTimes()  
   time.sleep(1)  
   SecSystemTimes = self.GetSystemTimes()  
    
   **"""  
   CPU usage is calculated by getting the total amount of time  
   the system has operated since the last measurement  
   made up of kernel + user) and the total  
   amount of time the process has run (kernel + user).  
   """** usr = SecSystemTimes[**'userTime'**] - FirstSystemTimes[**'userTime'**]  
   ker = SecSystemTimes[**'kernelTime'**] - FirstSystemTimes[**'kernelTime'**]  
   idl = SecSystemTimes[**'idleTime'**] - FirstSystemTimes[**'idleTime'**]  
   self.sys = usr + ker  
   **return** int((self.sys - idl) \* 100 / self.sys)  
    
  **def** cpu\_process\_util(self, hproc):  
   *"""  
   Returns the process usage of CPU  
   \*\* self.cpu\_utilization() must run first!!  
   Source: http://www.philosophicalgeek.com/2009/01/03/determine-cpu-usage-of-current-process-c-and-c/* **:param** *hproc: Process handle* **:return***: Process CPU usage (int)  
   """  
    
   # hproc = proc* **if** hproc == 0:  
   **return** 0  
   FirstProcessTimes = win32process.GetProcessTimes(hproc)  
   time.sleep(1)  
   SecProcessTimes = win32process.GetProcessTimes(hproc)  
    
   **"""  
   Process CPU usage is calculated by getting the total amount of time  
   the system has operated since the last measurement  
   made up of kernel + user) and the total  
   amount of time the process has run (kernel + user).  
   """** proc\_time\_user\_prev = FirstProcessTimes[**'UserTime'**]  
   proc\_time\_kernel\_prev = FirstProcessTimes[**'KernelTime'**]  
    
   proc\_time\_user = SecProcessTimes[**'UserTime'**]  
   proc\_time\_kernel = SecProcessTimes[**'KernelTime'**]  
    
   proc\_usr = proc\_time\_user - proc\_time\_user\_prev  
   proc\_ker = proc\_time\_kernel - proc\_time\_kernel\_prev  
    
   proc\_total\_time = proc\_usr + proc\_ker  
    
   proc\_utilization = (100 \* proc\_total\_time) / self.sys  
   **return** proc\_utilization

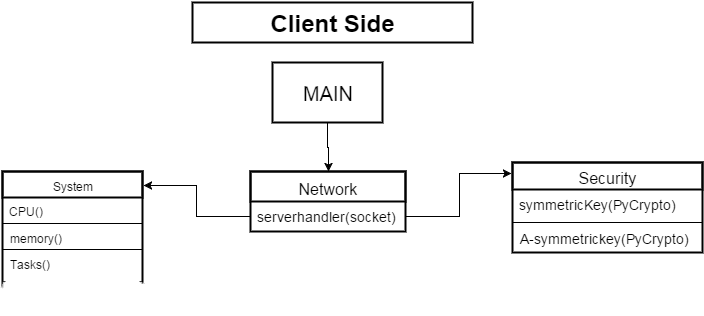
# 

# תיאור המודולים המרכיבים את המוצר

**עץ המודולים המרכיבים את המוצר בצד השרת:**



**עץ המודולים המרכיבים את המוצר בצד הלקוח:**



**פירוט על מחלקות השרת:**

* מודול תקשורת:
* מחלקת טיפול לקוח: אחראית לכל התקשורת בין השרת ללקוח כלומר אחראית להעביר הוראות או מידע ללקוח ולקבל ממנו מידע.
* מחלקת טיפול ב(GUI):אחראית לקבל מידע ממנהל הקצה באמצעות הGUI לעבד אותו ולשלוח אותו בהתאם ללקוח או להפך לקבל מידע מהלקוח ולהציג לGUI.
* מודול אבטחה:
* מחלקת מפתח א-סימטרי: אחראית על כל הקשור למפתח הא-סימטרי. המפתח הא-סימטרי הוא חלק ממכלול ההצפנה וההגנה על התקשורת.
* מחלקת מפתח סימטרי: אחראית על כל הקשור למפתח סימטרי. המפתח הסימטרי הוא חלק ממכלול ההצפנה וההגנה על התקשורת.
* מודול GUI:
* מחלקת עדכון הGUI: אחראית על טיפול בכל הקשור לפליטת המידע אלGUI ולקליטת המתקבל.

**פירוט על מחלקות הלקוח:**

* מודול התקשורת:
  + מחלקת טיפול בשרת:היא אחראית על נושא התקשורת עם השרת, לקבל ממנו מידע לנתח אותו ולהעביר את ההוראות המתאימות, כמו כן לשלוח את המידש שהתבקש.
* מודל האבטחה:
  + מחלקת מפתח א-סימטרי: אחראית על כל הקשור למפתח הא-סימטרי. המפתח הא-סימטרי הוא חלק ממכלול ההצפנה וההגנה על התקשורת.
  + מחלקת מפתח סימטרי: אחראית על כל הקשור למפתח סימטרי. המפתח הסימטרי הוא חלק ממכלול ההצפנה וההגנה על התקשורת.
* מודול המערכת:
* מחלקת CPU: מטפלת בכל מה שקשור למידע על הCPU. לדוגמא: סוג הCPU, אחוז הניצול הכללי ואחוז הניצול של כל תהליך מהמעבד, כמה תהליכים פועלים כרגע, מידע עליהם וכדומה.
* מחלקת Memory: מטפלת בכל מה שקשור למידע של זיכרון המחשב. לדוגמא:גודל הRAM, אחוז הניצול הכללי של זיכרון הRAM ואחוז הניצול של כל תהליך מהזיכרון.
* מחלקת Disk: מביאה אינפורמציה על כל הכונים שיש במחשב כמו: שמות של כוננים, כמה קיבולת הם יכולים להכיל, כמה קיבולת פנוי וכמה תהליך מנוצל מהכונן בו הוא נמצא.

# פירוט מבני הנתונים

|  |  |  |  |
| --- | --- | --- | --- |
| **מודול** | **שם המבנה** | **סוג** | **תפקיד** |
| System.py | self.processes | Dictionary | אחראית להחזיק פרטים על התהליכים באופן הבא:  מפתח-PID  ערך- רשימה המכילה במקום ה- 0 את שם התהליך, ובמקום ה- 1 את מספר הhandle שלה. |
| self.disk\_dict | Dictionary | מילון המחזיק פרטים על הכוננים הנמצאים על המחשב. המסודרים באופ הבא:  מפתח-שם הכונן  ערך-מילון המורה את כמות הקיבולת בכל כונן וכמות האחסון הפנויה. |
| Monitor.py | self.suspicious\_CPU\_processes | list | רשימה שמכילה את כל הנתונים של כל התהליכים במחשב החשודים כי הם צורכים מעבר לרף שנקבע מהמעבד. |
| Monitor.py | self.suspicious\_Memory\_processes | list | רשימה שמכילה את כל הנתונים של כל התהליכים במחשב החשודים כי הם צורכים מעבר לרף מהזכרון הדינאמי. |
| Monitor.py | self.disk | Dictionary | בודק האם מצב הדיסקים הקשיחים השתנה לאחרונה, על ידי השוואה לself.disk\_dict במודול הsystem. |