סמינר בתכנות מערכות דפנסיבי

מגיש: אלון שנקלר 206280547

# תוכן הסמינר - בניית מערכת מאובטחת לביזור משימות המורכבת מרכיבים בשפות שונות

# חלק ראשון – תוכנת לקוח בשפת C++

## שפת C++ תכונות ומרכיבים

מקורות:

* Stroustrup, Bjarne. *The C++ programming language*. Pearson Education India, 2000.
* Lippman, Stanley B. *C++ Primer*. Pearson Education India, 2005.
* Smith, Ben. "Object-oriented programming." *AdvancED ActionScript 3.0: Design Patterns*. Apress, 2011. 1-25.

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

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

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.

תכונה זו, יחד עם תכונות אחרות משייכות את C++ לקבוצת *Low-Level Programming Languages.*

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.

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.

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*.

### 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++

מקורות:

|  |
| --- |
|  |
| * Larochelle, David, and David Evans. "Statically detecting likely buffer overflow vulnerabilities." (2001). * Buchlovsky, Peter, and Adam Butcher. "BUFFER OVERFLOW VULNERABILITIES." * Hund, Ralf, Carsten Willems, and Thorsten Holz. "Practical timing side channel attacks against kernel space ASLR." *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013. * Szekeres, Laszlo, et al. "Sok: Eternal war in memory." *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013. |

### 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.

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.

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!

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.  
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

מקורות:

* Van Rossum, Guido. "Python Programming Language." *USENIX Annual Technical Conference*. Vol. 41. 2007.
* Paulson, Linda Dailey. "Developers shift to dynamic programming languages." *Computer* 40.2 (2007).

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

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.  
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).

*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.

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.

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

מקורות:

* Van Rossum, Guido. "Python Programming Language." *USENIX Annual Technical Conference*. Vol. 41. 2007.
* Slaviero, Marco. "Sour Pickles."

### 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.

### חולשה במנגנון הסריאליזציה המובנה של פייתון – pickle

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)