

עבודה סמינריונית

תכנות מערכות דפנסיבי

סמינר: תכנות מערכות דפנסיבי 20928

מגיש: עמית סידס

בהנחיית: פרופ' לאוניד ברנבוים

תעודת זהות: 206768780

תאריך:

# תוכן עניינים

[תוכן עניינים 1](#_Toc57503663)

[מבוא 3](#_Toc57503664)

[שפת C++ 4](#_Toc57503665)

[מבנה ותחביר השפה 4](#_Toc57503666)

[מחלקות, אובייקטים וירושה 4](#_Toc57503667)

[פולימורפיזם 5](#_Toc57503668)

[מודל הזיכרון 5](#_Toc57503669)

[איומי אבטחה 6](#_Toc57503670)

[חולשת Stack Buffer Overflow 6](#_Toc57503671)

[מתקפת Return-oriented Programming 6](#_Toc57503672)

[דריסת משתנים וטבלאות וירטואליות 7](#_Toc57503673)

[מנגנוני אבטחה 7](#_Toc57503674)

[שפת Python 8](#_Toc57503675)

[מבנה ותחביר השפה 8](#_Toc57503676)

[דינאמיות השפה 8](#_Toc57503677)

[מחלקות, אובייקטים וירושה 8](#_Toc57503678)

[איומי אבטחה 8](#_Toc57503679)

[חולשות ב-Deserialization 8](#_Toc57503680)

[טעינת קובץ זדוני 8](#_Toc57503681)

[חילוץ קבצים זדוניים 8](#_Toc57503682)

[שימוש בפונקציות מסוכנות 8](#_Toc57503683)

[מנגנוני אבטחה 8](#_Toc57503684)

[תקשורת 9](#_Toc57503685)

[אבטחה בתקשורת 9](#_Toc57503686)

[פרוטוקולי הצפנה ואימות 9](#_Toc57503687)

[אלגוריתם Diffie–Hellman 9](#_Toc57503688)

[הצפנת RSA 9](#_Toc57503689)

[פרוטוקול SSL 9](#_Toc57503690)

[פרוטוקול TLS 9](#_Toc57503691)

[מתקפות שונות על פרוטוקולי תקשורת 9](#_Toc57503692)

[מתקפת POODLE 9](#_Toc57503693)

[מתקפת BEAST 9](#_Toc57503694)

[מתקפת CRIME 9](#_Toc57503695)

[מתקפת Heartbleed 9](#_Toc57503696)

[מתקפה על מנגנון האימות של TLS 9](#_Toc57503697)

[אבטחת מערכת 10](#_Toc57503698)

[מתקפות MiTM 10](#_Toc57503699)

[מתקפת ARP Poisoning 10](#_Toc57503700)

[מתקפת DNS Poisoning 10](#_Toc57503701)

[מתקפות מניעת שירות 10](#_Toc57503702)

[מתקפת DOS 10](#_Toc57503703)

[מתקפת DDOS 10](#_Toc57503704)

[מתקפת DRDOS 10](#_Toc57503705)

[ביבליוגרפיה 11](#_Toc57503706)

# מבוא

# שפת C++

שפת C++ נוצרה לראשונה בשלהי שנות ה-70, ע"י מדען המחשבים ביארן סטרוסטרופ כשעבד על עבודת הדוקטורט שלו [1]. בעבודתו, הוא נהג להעזר בשפת התכנות Simula, שהייתה אחת מבין שפות התכנות הראשונות שתמכו במודל התכנות-מונחה עצמים. לטעמו, השפה הזו הייתה איטית מידי לשימושים מעשיים ועל כן החליט לעבוד על פיתוח שפה חדשה המובססת על שפת C אשר תאשר שימוש במחלקות. בזמנו, הוא כינה אותה בשם “C with Classes”, ועם השנים שמה שונה ל- C++. כפי שהשם מרמז, מדובר על שפת C עם עוד תוספות ושדרוגים, כפי שעושה האופרטור ++ אשר מגדיל את ערכו של משתנה ב-1.

במהלך השנים, שפת C++ התקדמה והשתדרגה באמצעות עוד תוספות ושדרוגים לשפה כמו: מחלקות, ירושה, מנגנון טיפול בחריגות (Exceptions), פונקציות וירטואליות, העמסת פונקציות, תבניות, ניהול זיכרון ועוד... [2] תוך כדי פורסמו מספר סטנדרטים המגדירים את מבנה, תחביר ומאפייני השפה, כשבכל אחד מהם הוספו, עודכנו ושונו הרבה מאוד תכונות בשפה שעזרו לה להתפתח והביאו אותה עד למצבה היום. כל 3 שנים, מאז שנת 2011 שבה יצא הסטנדרט C++11, שכלל שינויים גדולים ומשמעותיים, פורסם סטנדרט חדש שהוספו אליו כל התוספות והשינויים שסיימו לפתח עד פרסום הסטנדרט [3]. הסטנדרט האחרון שפורסם ממש לאחרונה (בעת כתיבת המאמר) הוא C++20 [4].

C++ מבוססת על השפה שקדמה לה, C. על כן, רוב התכנים של השפות זהה ואפילו יש לומר ש-C++ שומרת על תאימות לאחור עם C, וכך כל קוד C בסטנדרט הנפוץ יוכל להתקמפל (לעבור הידור – compilation) בעזרת compiler מתאים הבנוי עבור C++. לפיכך, לא אציין את כל מאפייני השפה אלא רק את הדברים שנוספו עבור C++ והופכים אותה לייחודית.

## מבנה ותחביר השפה

כמו C, גם C++ היא שפה פרוצדורלית. כלומר, ישנה חלוקה בין פרוצדורות (או פונקציות) אשר כל אחת אחראית על פעולה מסוימת וכל פרוצדורה יכולה לקרוא לאחרת על מנת שתבצע את הפעולה עליה היא אחראית. בכל פרוצדורה, ניתן להגדיר ולהשתמש במשתנים אשר מיוצגים ע"י אחד מהטיפוסים הבסיסיים של השפה, או להגדיר טיפוסים חדשים שיכילו את אחד או יותר מהטיפוסים הבסיסיים האחרים [5].

בנוסף, ישנן כמה דרכים שונות לשלוט בזרימת התוכנית. הדרך הבסיסית ביותר היא באמצעות פקודת ה-if המאפשרת לבדוק האם תנאי מסוים מתקיים או לא. בנוסף, קיימות 3 פקודות המאפשרות לבצע לולאה (קטע קוד החוזר על עצמו מספר פעמים) עד שתנאי כלשהוא יופר: for, do, while. יתר על כן, קיימת הפקודה switch המאפשרת להשוות ערך של משתנה למספר רב של ערכים קבועים בצורה יעילה (בעזרת jump table לרוב – תלוי compiler), וגם שימוש בפקודות כמו continue, break ו-goto אשר משנות את זרימת התוכנית בהתאם לשימוש שלהן בקוד.

## מחלקות, אובייקטים וירושה

מחלקה (Class) היא קבוצה של משתנים. בשונה מ-struct אשר קיים כבר ב-C, מחלקה כוללת בנוסף למשתני המידע גם פונקציות המוגדרות עבור המחלקה. אובייקט (Object) הוא מופע (Instance) של מחלקה. בדומה לכך שניתן להגדיר משתנים מטיפוסים בסיסיים, כך ניתן להגדיר משתנים שהם אובייקטים מטיפוס של המחלקה [6]. כל אובייקט שומר מצביע לפונקציות המחלקה המוגדרות עבורו.

ניתן ליצור מחלקה אשר יורשת ממחלקה אחרת. כך, כל אובייקט של המחלקה היורשת יכיל גם את המשתנים של המחלקה ממנה הוא יורש, ובנוסף גם את הפונקציות המוגדרות עבורה [7].

לכל משתנה במחלקה ניתן להגדיר מאפיין גישה (Access Specifier) אשר קובע מי יכול לגשת לאותו משתנה ומהיכן ניתן לגשת לאותו משתנה. מאפיין הגישה הדיפולטיבי הוא private המציין שניתן לגשת למשתנים המוגדרים באמצעותו אך ורק מפונקציות המוגדרות במחלקה של המשתנה. שאר המאפיינים הם:

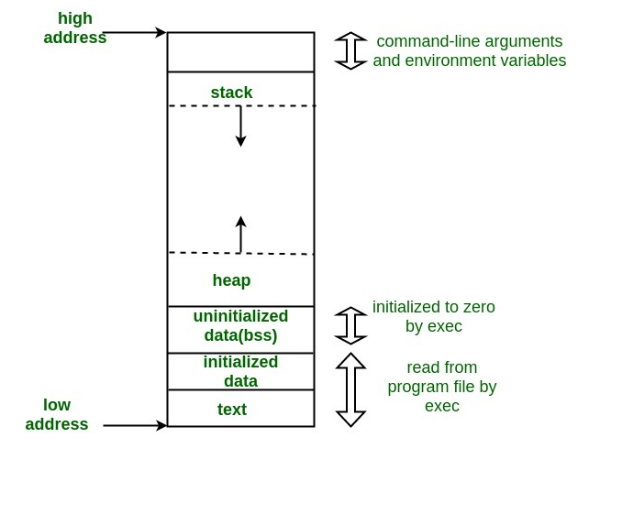
1. protected אשר מגדיר שניתן לגשת למשתנה במחלקה הנוכחית וכל מחלקה היורשת ממנה.
2. public אשר מאפשר לגשת למשתנה מכל פרוצדורה הנגישה לאובייקט של המשתנה. [6]

## פולימורפיזם

פולימורפיזם היא תכונה של שפות תכנות המאפשרת להשתמש באותם סימבולים, כמו שמות פונקציות או אופרטורים, למימוש שונה בהתאם לצורה שבה משתמשים בהם. C++ מממשת פולימורפיזם במספר רחב מאוד של צורות, ובכך מאפשרת תכנות גנרי אשר מונע כפילות קוד וחוסך עבודה למתכנת [8]. דוגמאות נפוצות לשימוש בפולימורפיזם בשפת C++ הן:

* העמסת פונקציות – הגדרת מספר פונקציות בעלות אותו שם אך עם חתימה שונה. הקומפיילר יודע בעזרת החתימה איזו פונקציה צריך להריץ. ניתן בצורה זו גם ליצור העמסה של אופרטורים ע"י הגדרת הפונקציות אשר ירוצו עבור כל אופרטור בשפה.
* דריסת פונקציות – החלפת מימוש של פונקציה במחלקה היורשת ממחלקה אחרת שבה הפונקציה מוגדרת כפונקציה וירטואלית. במצב כזה המצביע לפונקציה השמור באובייקט יתעדכן בהתאם לפונקציה של המחלקה אשר דרסה אותה.
* תבניות (Templates) – הגדרת פונקציה או מחלקה עבור טיפוסי מידע גנריים. בצורה זו ניתן להגדיר מימוש יחיד עבור פונקציה או מחלקה והקומפיילר ידע ליצור מימוש מתאים בכל פעם שמשתמשים בתבנית בהתאם לטיפוסים שאיתם עושים שימוש בתבנית. [9]

## מודל הזיכרון



מבנה הזיכרון במערכות הפעלה נפוצות. נלקח מהאתר geeksforgeeks.org [10]

כמו בכל תהליך מחשב במערכת הפעלה סטנדרטית, גם תוכניות הכתובות ב- C++ מחולקות במבנה הזיכרון למקטעים (Segments) מסויימים, חלקים הכרחיים וחלקם פחות. מקטעי הזיכרון ההכרחיים ביותר הם ה-text, שבו שומר הקוד של התהליך, וה-stack, שבו נשמרת מחסנית הריצה של התוכנית הכוללת משתנים סטטיים, פרמטרים וכתובות חזרה מקריאות של פונקציות. בנוסף להם, ישנם גם את המקטעים של ה-heap (ערימה), שבו נשמרים האובייקטים וקטעי הזיכרון אשר מוקצים בצורה דינאמית, ה-bss, שבו נשמרים המשתנים אשר אינם מאותחלים (ויאותחלו ל-0 בתחילת הריצה), וה-data, שבו נשמרים כל המשתנים הגלובליים שאינם שייכים לאף scope ריצה מסוים ותמיד צריכים להשאר בזיכרון (למעשה, המשתנים הגלובליים ואלו המוגדרים כסטטיים בחלקי הקוד). [10]

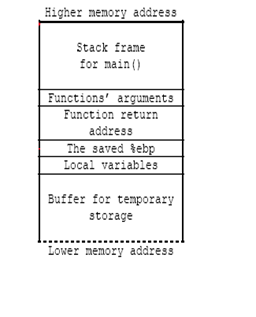
בשפת C++, כמו ב-C, משתנים אשר מוגדרים ב-scope של פונקציה ישמרו ב-stack, אשר הולכת וגודלת לכיוון המקטע של ה-heap. בנוסף להם, לעיתים גם נשמרים ב-stack הפרמטרים המועברים לפונקציות או אוגרים (registers) אשר רוצים לשמור את ערכם לצורך שימוש בהם. ויתר על כן, לרוב גם נשמרת שם כתובת החזרה מהקריאה לפונקציה או כתובת המחסנית של הפונקציה הקודמת, ולעיתים גם ה-stack canary (או stack cookie) אשר נשמרת על מנת להגן על תקינות המחסנית.

ב- C++, בשביל להקצות אובייקט בזיכרון בצורה דינאמית יש להשתמש באופרטור new ובאופרטור המקביל delete בשביל לשחרר את ההקצאה, זאת להבדיל מ-C שבה יש להשתמש בפונקציה malloc ומשפחתה ו-free לצורך השחרור (אולם ניתן להשתמש בהם גם ב- C++ כמובן). האופרטורים new ו-delete קוראים לפונקציית ה-constructor וה-destructor (בהתאמה) ומקצים או משחררים בלוקי זיכרון על ה-heap [11]. בנוסף, כדי למנוע מצבים של דליפות זיכרון עקב הקצאת אובייקטים ללא שיחרורם (אשר עלולים להגרם הרבה במצבים בהם נזרקת שגיאת ריצה), הוספו לשפה מחלקות המממשות מצביעים חכמים (Smart Pointers) ע"י מחלקות כמו unique\_ptr ו-shared\_ptr, אשר דואגים לשחרר את האובייקט ברגע שהמצביע אליו משוחרר מהזיכרון או כאשר המצביע האחרון לאותו אובייקט משתחרר או נדרס.

ניתן לשים לב כי קטעי הזיכרון של ה-stack וה-heap מתקרבים אחד לשני ככל שמשתמשים בהם יותר, לכן כמות המקום של כל אחד מהם מוגבלת ולפעמים הקצאות דינאמיות יכשלו או, במקרה הגרוע יותר, התוכנית תקרוס עקב דריסת מידע של קטע זיכרון כלשהו ע"י השני.

## איומי אבטחה

### חולשת Stack Buffer Overflow



דוגמא לפורמט המחסנית בארכיטקטורת x86 נלקח מאתר tenouk.com - [29]

Stack Buffer Overflow היא חולשה מסוג Buffer Overflow, שהיא קבוצה של חולשות אשר בהן נעשה שימוש זדוני בתוכנה בשביל לדרוס מידע בזיכרון מעבר למקום שהוקצה עבור ה-buffer שבו נשמר המידע הזדוני שנכתב.

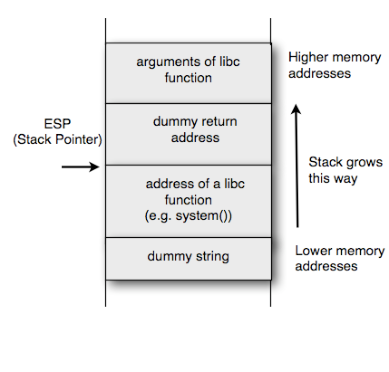
כידוע, buffer (חוצץ) הוא חלק רציף בזיכרון הנדיף של התוכנה, שבו נשמר מידע הנעשה בו שימוש זמני בתוכנה. כאשר מפתח התוכנה אינו בודק גבולות גישה עבור הגישות ל-buffer הזה, עלול להיווצר מצב שבו התוכנה ניגשת לחלקים בזיכרון שאינם נמצאים באיזור המוקצה ל-buffer, לרוב לחלקים שבאים אחריו (כלומר, בכתובת גבוהה יותר). לעומת Heap Overflow, שבה ה-buffer נשמר באיזור ה-heap ע"י הקצאה דינאמית, בחולשת ה-Stack Buffer Overflow ה-buffer מוקצה בצורה סטאטית על מחסנית הריצה של התוכנית. כפי שכבר הוזכר קודם, בהרבה מאוד ארכיטקטורות, באותו חלק בדיוק נשמרים נתונים חשובים כמו משתנים מקומיים, מצביע לתחילת המסגרת (frame) של הפונקציה, והחשוב מכל: כתובת החזרה שממנה הפונקציה תחזור לפונקציה שקראה לה. [12]

בהנתן שקיימת חולשה מסוג Stack Buffer Overflow, משתמש זדוני, אשר יש לו גישה למידע הנכתב לאותו ה-buffer, יכול לדרוס את המידע שנמצא אחריו ובכך להשפיע על הריצה של התוכנית ותקינות הקוד. בפרט, מכיוון שבהרבה ארכיטקטורות כתובת החזרה נשמרת על המחסנית אחרי ה-buffer, טכניקה נפוצה היא לדרוס את כתובת החזרה למיקום שנמצא בתוך ה-buffer עצמו, שלמשתמש הזדוני יש שליטה בתוכנו. בטכניקה זו המשתמש הזדוני יכתוב ל-buffer קטע קוד שאינו תלוי בחלקים אחרים בקוד או במיקום שלו, אשר נקרא shellcode, ובכך יוכל לדרוס את כתובת החזרה לכתובת של הקוד שלו ולהריץ קוד לפי בקשתו.

קיימות מספר רב מאוד של שיטות שונות ומגוונות אשר מציעות פתרונות על מנת לפתור בעיות הנוצרות מחולשות כאלו, שעליהן ארחיב בהמשך בתת-פרק על מנגנוני אבטחה.

### מתקפת Return-oriented Programming

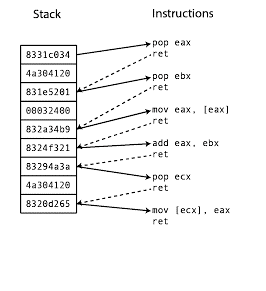
Return Oriented Programming היא טכניקה להרצת קוד אשר מבוססת על חולשת Stack Buffer Overflow. בחלק מהמקרים, בניסיון להשמיש חולשת Stack Buffer Overflow, לא נוכל לשלוח את ה-shellcode שלנו עקב מגבלות מסוימות כמו כמות תווים (למשל, הגבלה של גודל הודעה) או מניעה של תווים מסוימים במידע (למשל, רק תווים טקסטואלים). בנוסף, ישנן כמה הגנות, המפורטות ב-מנגנוני אבטחה, אשר אינן מאפשרות להריץ קוד אשר נמצא במקטע הזיכרון של ה-stack, ולכן גם במקרים אלו שליחת shellcode על גבי ה-buffer אינה מועילה לנו.



דוגמא להשמשת return-into-libc נלקח מהמאמר [30]

עבור מקרים אלו הומצאה טכניקת Return Oriented Programming, או בקיצור, ROP. בטכניקה זו, התוקף מסתמך על העובדה כי יש לו שליטה מלאה על התוכן אשר נמצא במחסנית הריצה, ע"י דריסה שלו עם תוכן ה-buffer. בהרבה ארכיטקטורות, ובפרט בארכיטקטורת x86, ה-calling convention הנפוץ מגדיר כי יש להעביר את הפרמטרים לפונקציות על גבי מחסנית הריצה. לפיכך, מכיוון שהתוקף יכול לשלוט במחסנית, הוא יכול, באופן תיאורטי, לקבוע אילו פרמטרים יכתבו למחסנית ובנוסף איזו פונקציה תקרא עם הפרמטרים הללו ע"י דריסה של כתובת החזרה לכתובת הפונקציה. לרוב, קריאה לפונקציה אחת אינה מספיקה. על כן, התוקף יבצע את אותו התהליך שוב ושוב ע"י בניית "מחסנית ריצה" הכוללת כתובות של פונקציות (בתור כתובת חזרה) ופרמטרים עבורן, על פי ה-calling convention המתאים. כמובן שהתהליך הזה יכול להתבצע מספר בלתי מוגבל של פעמים (בהיקפים סבירים בהתאם לחומרה), בהינתן שאין הגבלות המונעות זאת (כמו הגבלה של גודל ה-buffer). השימוש הזה ב-ROP נקרא return-into-library. זאת משום שהתוקף דורס את ה-buffer ברצף של קריאות לפונקציות אשר נמצאות בספריית פונקציות סטנדרטית כלשהיא, לדוגמא libc או STL. ספריות כאלו נחשבות למאוד נפוצות במערכות ההפעלה שלהן ולכן התוקף יכול להניח בנוחות כי המימושים לפונקציות נפוצות בהן גם נמצאים במרחב הזיכרון של התוכנית.

במקרים אחרים, בהם ה-calling convention אינו מאפשר לנו לשלוח את הפרמטרים לפונקציות על גבי המחסנית, אנו נאלץ למצוא דרך שבה נוכל לשנות את ערכי האוגרים (registers) של המעבד, שבהם מועברים הפרמטרים לפונקציות. בנוסף, במקרים שבהם קשה למצוא את מיקום הפונקציות (בגלל ASLR לדוגמא) נשתמש באותן דרכים בשביל להשפיע על ריצת התוכנית ואולי אפילו לקרוא לקריאות מערכת (syscalls) בעצמנו ולא בעזרת פונקציית ספריה.



דוגמא ל-ROP עם gadget-ים. נלקח מהמאמר [31]

בשביל שנוכל לשנות ערכים של אוגרים בצורה נוחה ופשוטה שתאפשר לנו לבצע מספר רחב של פעולות, אנו נרצה לחפש gadget-ים רבים ככל שנוכל. Gadget הוא רצף של פקודות מעבד (instructions) אשר מבצע מספר מצומצם של פעולות כשבסופו מופיע הפקודה ret, אשר מבצעת חזרה מקריאה לפונקציה, ובפועל שולפת את כתובת החזרה שלנו מהמחסנית וקופצת אליה. למשל, בארכיטקטורת x86, gadget טוב היכול לשמש להרבה דברים הוא ה-gadget הכולל את הפקודות: pop eax; ret. ב-gadget הזה מבצעים שליפה מהמחסנית (שבתוכן שלה אנו שולטים) לאוגר ה-eax ולאחר מכן מבצעים ret וקופצים לכתובת החזרה (שגם עליה ניתן לשלוט). תוקף, אשר מעוניין להריץ קוד מבלי לשלוח אותו על גבי ה-buffer, יכול לבנות מחסנית ריצה שלמה המבוססת על gadget-ים כאלו, אשר רצים אחד אחרי השני כשבכל אחד מהם מבצעים מספר קטן של פקודות, המקדמות את לוגיקת הקוד שהתוקף רוצה להריץ, ואחריהן מבצעים ret וקופצים ל-gadget הבא...

באופן תיאורטי, אם תוקף נגיש ל-gadget מהסוג הזה עבור כל אוגר אשר קיים במעבד, הוא למעשה יכול לשנות כל אוגר לכל ערך שיבחר ובכך להריץ כל סוג קוד שרק יבחר, בהתאם למגבלות על תוכן ה-buffer.

ישנם סוגים רבים של gadget-ים אפשריים המאפשרים דברים שונים. ישנו gadget הנקרא gadget קריאה, אשר מבצעים בו טעינה של מידע מכתובת השמורה באוגר מסוים לתוך אוגר אחר. בהנחה שניתן לשלוט בתוכן האוגר שממנו קוראים, ניתן לקרוא מכל כתובת שהתוכנית נגישה אליה. בהתאם, ישנו gadget הנקרא gadget כתיבה אשר מבצע את הדבר ההפוך ומאפשר לכתוב ערך של אוגר לכתובת השמורה באוגר אחר, ובכך התוקף יכול לכתוב כל מידע שיבחר לכל כתובת הנגישה לתוכנית. שילוב של gadget-ים כאלו מאפשר לתוקף להזליג מידע חשוב לגבי התוכנה, ובהנחה שמדובר בשרת רשתי, דבר כזה עלול להיות מאוד מסוכן לנתקף.

### דריסת משתנים וטבלאות וירטואליות

בחלק מהמקרים שבהם ננסה להשמיש חולשת Stack Buffer Overflow, אנו עלולים להתקל בהגנות מסוימות (כמו stack canary) אשר מונעות מאיתנו לדרוס את כתובת החזרה על מחסנית הריצה. אבל, הגנות אלו עדיין אינן פותרות באופן מלא השמשות אפשריות של החולשה. למשל, בדוגמאת הקוד המופיע משמאל ניתן לראות פונקציה בשם authenticate האחראית על בדיקה אם המשתמש מאומת, ולפי זה מאפשרת לו לבצע פעולות שמשתמש רגיל לא יכול. בדוגמא זו, ניתן להשפיע על בקרת הזרימה (flow control) של התוכנית מבלי לדרוס בכלל את כתובת החזרה או כתובת מסגרת המחסנית השמורה. הכנסת הסיסמא 01234567890123456789012345678, אשר ארוכה יותר מגודל ה-buffer ששמו password, תגרום לכך שהמשתנה השני המוגדר על המחסנית, authenticated, ידרס ע"י התווים של הסיסמא שהוכנסה ומכיוון שהתווים הטקסטואליים שונים מהתו ‘\0’ הערך הבוליאני שיוחזר מהפונקציה יהיה true.

המקרים הללו של דריסה משתנים אומנם אינה נפוצה, בעיקר בגלל שרוב ה-compiler-ים כיום דואגים לשמור מערכים בסוף רשימת המשתנים כך שלא יהיה ניתן לדרוס בעזרתם משתנים אחרים (ולשם כך הוסף הדגל של -fno-stack-protector בדוגמא), אך הם עדיין קיימים: אם המשתנה השמור על המחסנית הוא struct אשר בו שמור מערך, ה-compiler לא יוכל לבצע את אותה אופטימיזציה מכיוון שסדר משתני המידע ב-struct מובטח שישאר זהה, או, כאשר ישנם 2 מערכים, האחד שנמצא בכתובת גבוהה יותר עלול להדרס ע"י השני מבלי שניתן לזהות זאת בקלות ללא פגיעה משמעותית בביצועים.

בנוסף, עוד דריסה אפשרית אשר עלולה להגרם בעקבות Stack Buffer Overflow ומאפשרת להתחמק מהגנות דומות היא דריסה של פונקציות וירטואליות, שהוא קונספט ייחודי של C++ אשר אינו קיים ב-C. כזכור, כל אובייקט של מחלקה שבה מוגדר פונקציות וירטואליות מכיל, כחלק ממשתני המידע שלו, עוד מצביע נוסף: מצביע לטבלה וירטואלית המכילה כתובות של פונקציות וירטואליות של האובייקט [13]. הטבלה הזו, אומנם, לרוב שמורה במקטע זיכרון מיוחד בשם rodata, אשר מאפשר קריאה בלבד, ולכן לא ניתן לדרוס את הכתובות של הפונקציות. אבל, מה שכן ניתן לעשות עם חולשת Stack Buffer Overflow שבה מוגדר אובייקט על המחסנית, הוא לדרוס את המצביע לטבלה הוירטואלית שלו, ולגרום לכך שיצביע, באופן תיאורטי, לכל מקום שנבחר. בפרט, כנראה שנרצה שהוא יצביע למקום כלשהו בתוך ה-buffer שלנו. במקום זה נוכל לקבוע בקלות את כתובת הקוד שירוץ ברגע שיקראו לפונקציה של אותו אובייקט. מתקפות כאלו ואחרות מוסברות בפירוט במאמר על VTint [14], שגם מסביר כיצד ניתן להגן מפניהן.

## מנגנוני אבטחה

ישנם אינספור מנגנוני אבטחה אשר מגנים מפני שלל בעיות וחולשות נפוצות בתוכנה, חלקם ממומשים בעזרת ה-compiler וחלקם בנויים כחלק ממערכת ההפעלה והמערכת המריצה את התוכנה. בחלק זה אציג מספר שיטות שונות שבעזרתן ניתן למנוע אחד או יותר מאיומי האבטחה שהצגתי למעלה:

1. NX bit – מנגנון ה-NX bit (No-eXecute bit) הוא מנגנון הממומש במעבדים אשר מאפשר למערכת הפעלה לקבוע באילו חלקים בזיכרון יהיה ניתן להריץ קוד. רוב מערכות ההפעלה משתמשות במנגנון זה בשביל לממש את הקונספט של (W xor X), אשר קובע כי כל קטע בזכרון יהיה ניתן או לכתיבה או להרצה, ואף פעם לא יאופשרו את שניהם יחדיו. כך, מערכת הפעלה אשר מממשת תמיכה במנגנון תקבע כי מקטע ה-text לא יאפשר כתיבה, ואילו מקטע ה-stack לא יאפשר הרצה [15]. בגרסאות החדשות של מע' ההפעלה Windows, לדוגמא, המימוש הזה נקרא DEP – Data Execution Prevention, וניתן (אך לא מומלץ) לכבות אותו בהגדרות. בכך, ניתן למנוע את השיטה הקלאסית של כתיבת shellcode לתוך buffer והרצה שלו. אך עדיין, הרבה איומי אבטחה אחרים שהוזכרו מעלה מצליחים להתגבר על הגנה זו.
2. ASLR – Address Space Layout Randomization, או בקיצור ASLR, היא שיטה הממומשת במערכת ההפעלה אשר גורמת לכך שבכל ריצה של תוכנית, החלקים השונים בזיכרון ימופו לכתובות אקראיות. כך, הרצה של שיטות מסוג ROP, ובפרט return-into-library, יהיו הרבה יותר קשות לביצוע, שכן התוקף יאלץ לדעת לאיזו כתובת נטענה הספרייה בזיכרון ורק לאחר מכן יוכל לקפוץ לפונקציות הספרייה או ל-gadget-ים הרלוונטיים שמצא.
3. Stack Canary – מנגנון זה, אשר נקרא גם Stack Cookie לפעמים, הוא מנגנון הממומש ע"י ה-compiler ומאפשר הגנה בזמן ריצה על כתובת החזרה השמורה על המחסנית. מימוש המנגנון לרוב מתבצע ע"י כתיבת 4 בתים או יותר של מידע בין כתובת החזרה לשאר משתני הפונקציה, ל-4 הבתים האלה קוראים canary או cookie. המיקום של ה-canary הוא אינו מקרי, בהנחה שהדריסה של המחסנית היא רציפה, בשביל לדרוס את כתובת החזרה, יאלץ התוקף גם לדרוס את ה-canary. כשכל פונקציה מסיימת את ריצתה, היא קודם כל בודקת שה-canary לא נדרסה, ורק לאחר מכן מבצעת את הקפיצה לכתובת החזרה. אם בזמן ריצה זיהינו שה-canary נדרסה, התוכנית תזרוק שגיאה ותפסיק את הריצה.
4. Reversed Stack – מנגנון נוסף, הפחות מוכר מהשאר, הוא מנגנון הקובע כי כיוון המחסנית יהיה הפוך מהמקובל כיום. משמעות הדבר היא שכעת, במקום שה-buffer ידרוס את תוכן המחסנית, הוא ידרוס מידע אקראי המופיע אחריה, ושברוב המקרים לא יפריע לריצת התוכנית. בארכיטקטורת ARM המנגנון הזה מובנה במעבד (למרות שאינו בשימוש נפוץ) ואילו ב-x86 פירסמו מאמרים המציגים כיצד ניתן לבצע זאת עם פגיעה מינימאלית ביעילות התוכנית, לדוגמא, המאמר הבא: [15].

# שפת Python

שפת Python היא שפת תכנות עילית (high-level), דינאמית ומפורשת (interpreted). היא שפה יחסית מודרנית הנחשבת לאחת משפות התכנות הקלות והפשוטות ביותר להתחיל לפתח בהן, עבור אנשים שאינם מגיעים מתחום הפיתוח. היא הופצה לראשונה בתחילת שנות ה-90 ע"י חידו ואן רוסום אשר המשיך לעבוד עליה ולפתח אותה עד לאחרונה בשנת 2018 שבה הפסיק לפקח על התפתחותה, כשבדרך הוא נעזר בקהילת המתכנתים הרחבה שהתפתחה לשפה ותרמה רבות לפיתוח שלה. השפה פותחה במטרה לאפשר למפתחים לכתוב קוד ברור, קריא ופשוט תוך זמן קצר ביחס לשאר השפות הקיימות. פשטות השפה, שבה ניתן לכתוב קוד במבנה הדומה מאוד לשפה האנגלית, בשילוב עם אינספור ספריות הקוד-הפתוח הקיימות עבורה ברשת הופך את תהליך הפיתוח לקל ופשוט בהרבה גם עבור פרויקטים גדולים ורחבים. [17]

ל-Python קיים מנהל חבילות בשם Pip המאפשר להוריד ולעדכן בקלות כל חבילה, או מאגר (repository), אשר כוללת פיתוח של אדם או חברה אחרת המתחזקים אותה, ובכך להשתמש באותה חבילה בפיתוח תחת הרישיון שבאמצעותו החבילה מופצת. מרבית החבילות מופצות באמצעות רשיונות עם הגבלות מינימליות, כדוגמאת MIT, BSD ו-GPL [18], ועל כן כל מפתח או חברה יכולים להשתמש בהם בחופשיות ובקלות.

במהלך השנים שוחררו מספר רב של גרסאות ועדכונים עבור השפה, שהעיקריים מביניהם היו בשנת 2000 שבה שוחררה גרסאת Python 2.0 ובשנת 2008 שבה שוחררה גרסאת Python 3.0, שניהם מכילים עדכונים ראשיים (major release) והביאו לשינויים משמעותיים בשפה, הן במימוש והן בפונקציונאליות שלה.

## מבנה ותחביר השפה

## דינאמיות השפה

## מחלקות, אובייקטים וירושה

## איומי אבטחה

### חולשות ב-Deserialization

[16]

### טעינת קובץ זדוני

[17]

### חילוץ קבצים זדוניים

[18]

[19]

[20]

### שימוש בפונקציות מסוכנות

## מנגנוני אבטחה

# תקשורת

## אבטחה בתקשורת

## פרוטוקולי הצפנה ואימות

### אלגוריתם Diffie–Hellman

[21]

### הצפנת RSA

### פרוטוקול SSL

### פרוטוקול TLS

## מתקפות שונות על פרוטוקולי תקשורת

[22]

### מתקפת POODLE

### מתקפת BEAST

### מתקפת CRIME

### מתקפת Heartbleed

### מתקפה על מנגנון האימות של TLS

[23]

[24]

# אבטחת מערכת

## מתקפות MiTM

### מתקפת ARP Poisoning

[25]

[26]

### מתקפת DNS Poisoning

[27]

## מתקפות מניעת שירות

[28]

### מתקפת DOS

### מתקפת DDOS

### מתקפת DRDOS

# ביבליוגרפיה

z

|  |  |
| --- | --- |
| [1] | Albatross, "History of C++," [Online]. Available: http://www.cplusplus.com/info/history/. |
| [2] | Albatross, "C++ - Brief Description," [Online]. Available: http://www.cplusplus.com/info/description/. |
| [3] | "C++11," [Online]. Available: https://en.wikipedia.org/wiki/C++11. |
| [4] | "C++20," [Online]. Available: https://en.wikipedia.org/wiki/C++20. |
| [5] | "Wikipedia - C++," [Online]. Available: https://en.wikipedia.org/wiki/C++. |
| [6] | "Classes," [Online]. Available: http://www.cplusplus.com/doc/tutorial/classes/. |
| [7] | "Friendship and Inheritance," [Online]. Available: http://www.cplusplus.com/doc/tutorial/inheritance/. |
| [8] | "Wikipedia - Polymorphism," [Online]. Available: https://he.wikipedia.org/wiki/פולימורפיזם\_(מדעי\_המחשב). |
| [9] | "Templates," [Online]. Available: https://www.cplusplus.com/doc/oldtutorial/templates/. |
| [10] | "Memory Layout," [Online]. Available: https://www.geeksforgeeks.org/memory-layout-of-c-program/. |
| [11] | "Dymanic Memory," [Online]. Available: http://www.cplusplus.com/doc/tutorial/dynamic/. |
| [12] | P. LACROIX and J. DESHARNAIS, "Buffer Overflow Vulnerabilities in C and C++," 7 August 2008. [Online]. Available: http://www2.ift.ulaval.ca/~desharnais/Recherche/RR/DIUL-RR-0803.pdf. |
| [13] | "Exploiting C++ VTABLES: Instance Replacement," 11 May 2013. [Online]. Available: https://defuse.ca/exploiting-cpp-vtables.htm. |
| [14] | C. Zhang, C. Song, K. Z. Chen and Z. Chen, "VTint: Protecting Virtual Function Tables' Integrity," February 2015. [Online]. Available: https://www.researchgate.net/profile/Chengyu\_Song/publication/281784405\_VTint\_Protecting\_Virtual\_Function\_Tables'\_Integrity/links/55f8743d08ae07629dd5e648/VTint-Protecting-Virtual-Function-Tables-Integrity.pdf. |
| [15] | A. G. M. F. Babak Salamat, "Reverse Stack Execution in a Multi-Variant Execution Environment," April 2012. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.218.7742&rep=rep1&type=pdf. |
| [16] | K. Tanaka and T. Saito, "Python Deserialization Denial of Services," in *Computational Science/Intelligence & Applied Informatics*, 2018, pp. 15-25. |
| [17] | A. Shaw, "10 common security gotchas in Python and how to avoid them," 16 June 2018. [Online]. Available: https://hackernoon.com/10-common-security-gotchas-in-python-and-how-to-avoid-them-e19fbe265e03. |
| [18] | A. Abraham, "Exploiting insecure file extraction in Python for code execution," 28 September 2017. [Online]. Available: https://ajinabraham.com/blog/exploiting-insecure-file-extraction-in-python-for-code-execution. |
| [19] | MITRE, "CVE-2019-9674 - denial of service via a ZIP bomb," 4 February 2020. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-9674. |
| [20] | MITRE, "CVE-2019-20907 - TAR archive leading to an infinite loop," 13 July 2020. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2019-20907. |
| [21] | Wikipedia, "Diffie–Hellman key exchange," [Online]. Available: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman\_key\_exchange. |
| [22] | A. Prodromou, "TLS Security 6: Examples of TLS Vulnerabilities and Attacks," 31 March 2019. [Online]. Available: https://www.acunetix.com/blog/articles/tls-vulnerabilities-attacks-final-part/. |
| [23] | D. Wagner and B. Schneier, "Analysis of the SSL 3.0 Protocol," November 1996. [Online]. Available: https://www.usenix.org/legacy/publications/library/proceedings/ec96/full\_papers/wagner/wagner.pdf. |
| [24] | N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov and B. Preneel, "A cross-protocol attack on the TLS protocol," October 2012. [Online]. Available: https://www.researchgate.net/publication/262208728\_A\_cross-protocol\_attack\_on\_the\_TLS\_protocol. |
| [25] | B. Fleck and J. Dimov, "Wireless Access Points and ARP Poisoning," December 2013. [Online]. Available: https://digilander.libero.it/SNHYPER/files/arppoison.pdf. |
| [26] | C. Nachreiner, "Anatomy of an ARP Poisoning Attack," 18 November 2012. [Online]. Available: http://csci6433.org/Papers/Anatomy%20of%20an%20ARP%20Poisoning%20Attack%20\_%20WatchGuard.pdf. |
| [27] | S. Son and V. Shmatikov, "The Hitchhiker’s Guide to DNS Cache Poisoning," in *Security and Privacy in Communication Networks*, The University of Texas, Austin, 2010, pp. 466-483. |
| [28] | Prolexic, "An Analysis Of DrDoS DNS Reflection Attacks," [Online]. Available: http://vertassets.blob.core.windows.net/download/74db6f36/74db6f36-56e7-4f4f-a6b4-a1880089f28a/analysis\_of\_drdos\_dns\_reflection\_attacks\_white\_paper\_us\_031513.pdf. |
| [29] | "Buffer Overflow," [Online]. Available: https://www.tenouk.com/Bufferoverflowc/Bufferoverflow4.html. |
| [30] | S. K. F. A. E.-S. O. Tamar Fatayer, "A key-agreement protocol based on the stack-overflow software vulnerability," June 2010. [Online]. Available: https://www.researchgate.net/publication/221505572\_A\_key-agreement\_protocol\_based\_on\_the\_stack-overflow\_software\_vulnerability. |
| [31] | D. W. Nicholas Carlini, "ROP is Still Dangerous: Breaking Modern Defenses," [Online]. Available: https://people.eecs.berkeley.edu/~daw/papers/rop-usenix14.pdf. |