# מבנה הקרנל הלינוקסי וטכניקות ניצול שלו

מאת ירין הרמן

## הקדמה

דמיינו שאתם נכנסים למסעדה מפוארת, שלושה כוכבי מישלן. השולחנות ערוכים, המלצרים לבושים בחליפות, והריחות מהמטבח מבטיחים חוויה בלתי נשכחת. אתם פותחים את התפריט ומזמינים מנה עילאית, אבל כשהמנה שלכם מוכנה, היא נזרקת אליכם בתוך קופסת קרטון פשוטה של פלאפל. הטעם אולי מצוין, אבל ברור לכם שהחוויה השתנתה לגמרי. האריזה, הסדר, והאופן שבו הדברים מוגשים – משפיעים לא פחות מהחומר עצמו.

[באיור - המחשה של מסעדת המישלן- (מקור)]

ככה זה גם במחשבים. המשתמשים רואים את הממשק, את התוכנות, את האייקונים היפים – אבל מאחורי הקלעים ישנו מנגנון עצום שמסדר את הכל: מחליט איזה תהליך יקבל זמן ריצה, מי ישמור על הזיכרון, ואיך המשאבים יחולקו. זהו הקרנל של מערכת ההפעלה, והארגון הפנימי שלו הוא לא פחות מה"מסעדה" שבה כל שאר המרכיבים מתקיימים.

אבל כמו שבמסעדה טובה יש גם מי שמנסה להתגנב למטבח האחורי ולגנוב את המתכון הסודי של קציצת הסרטן, גם הקרנל לא חסין. החוליות החלשות שלו, כשנחשפות, הופכות לנתיב כניסה והסלמת הרשאות מפתה מאוד בעבור תוקפים.

במאמר הזה אציג איך הקרנל הלינוקסי בנוי, מה קורה שם מתחת למכסה המנוע, ואיך בדיוק אותם מנגנונים מופלאים שנועדו לשמור על סדר – עלולים להפוך לכלי משחק בידי התוקפים.

## תזמון תהליכים ו-Context Switching

ה-scheduler של ה-kernel פועל כמו פקח תנועה, שמחליט איזה תהליך (או thread) ירוץ על ה-CPU הבא ולכמה זמן. לינוקס משתמש ב-preemptive scheduler. כלומר, ה-kernel יכול להפסיק תהליך רץ כדי לעבור לאחר, וכך למנוע מתהליך יחיד להשתלט על ה-CPU. ה-scheduler ברירת המחדל הוא CFS, קיצור של Completely Fair Scheduler. תפקידו הוא לחלק את זמן ה-CPU באופן הוגן בין התהליכים.
ב-CFS, לכל תהליך מוקצה משקל (weight) המבוסס על priority או ערך ה-"nice" שלו, והוא נשמר במבנה של כל התהליכים ה-runnable, כאשר המפתח הוא ה-virtual runtime - ערך זמן שגדל מהר יותר עבור תהליכים עם עדיפות נמוכה. ה-scheduler תמיד יבחר את ה-task עם ה-virtual runtime הקטן ביותר (זה שקיבל הכי מעט זמן CPU ביחס לאחרים) כדי להריץ אותו הבא, מה שמדמה CPU multitasking "הוגן ואידאלי". בצורה זו, כל תהליך מקבל עם הזמן חלק שווה והוגן מה-CPU. ממש קומוניזם בקרנל.

בעוד ש-CFS מטפל בעומסי עבודה רגילים (מדיניות scheduling מסוג SCHED_NORMAL), לינוקס מספקת גם מחלקות scheduler נוספות לצרכים מיוחדים. לדוגמה, real-time scheduling מאפשרת למשימות בעלות עדיפות גבוהה לבצע preempt לאחרות ולהמשיך לרוץ עד לסיום או למשך זמן קבוע. קיימת גם מחלקת SCHED_DEADLINE (מבוססת Earliest Deadline First) עבור משימות רגישות לדדליינים, ומחלקת IDLE ששמה כן היא - עבור עבודות ברקע בעדיפות נמוכה מאוד. לכל מחלקת scheduling יש חוקים משלה, אך כולן משתלבות במסגרת ה-scheduler של ה-kernel.

אז הכל טוב ויפה? יש ויסותים לתהליכים כאילו מדובר בויסותים לחדר אוכל? בזה נגמר הסיפור? אז לא. בוא נקביל את התהליכים לחדר אוכל. יש ויסותים לקורס א' ב-12 ולקורס ב' ב-12:10. מה יקרה כאשר קורס א' מאחר? או שקורס א' מחכה למפקד שלו שישחרר אותו לחדר אוכל? בכל הזמן הזה קורס ב' לא יוכל להיכנס לאכול? מצב בעייתי בלשון המעטה. בשביל זה יש אדם שעומד בקדמת חדר האוכל שיכול לומר לקורס ב' להיכנס לפני קורס א'. אם נצא מההקבלה הזאת, מה קורה כאשר תהליך מסוים צריך לחכות למשהו? או שהוא רץ כבר למספיק זמן? (כלומר לקח יותר מדיי זמן לבחור את המנה הבשרית)? בדיוק בשביל זה יש לנו context switch.

Context Switch מתרחש כאשר ה-scheduler מחליף את ה-CPU מתהליך אחד לאחר. זה יכול לקרות כאשר ה-timeslice של התהליך מסתיים, כאשר התהליך נחסם (למשל בהמתנה ל-I/O), או כאשר תהליך בעל עדיפות גבוהה יותר מתעורר. תהליך ה-context switching כולל שמירה של מצב התהליך הנוכחי ושחזור מצב התהליך הבא. ה-kernel מחזיק עבור כל תהליך מבנה task control block (או בקיצור TCB) מסוג struct, עם כל המידע הדרוש לחידוש הריצה (רשומות CPU, מיפויי זיכרון ועוד). כאשר מתבצע context switch, פונקציית ()schedule של ה-scheduler בוחרת את התהליך הבא להרצה. שגרת ה-context switch ברמה הנמוכה שומרת את רשומות ה-CPU ואת מצב הביצוע של התהליך היוצא, וטוענת את המצב השמור של התהליך הנכנס.

Context Switch תוכנן להיות תהליך מהיר אבל בפועל הוא מוסיף גם הרבה עומס על המעבד. בכל context switch הקרנל צריך:

לשמור את כל המידע של התהליך הנוכחי (מצב רשומות ה-CPU, מצב הזיכרון, מיקום בקוד).
לטעון את כל המידע של התהליך הבא.
לעיתים גם לרוקן (flush) את ה-CPU cache וה-TLB, כי הנתונים שם שייכים לתהליך הקודם ואי אפשר להשתמש בהם בצורה בטוחה עבור התהליך החדש.
לומר את תפילת הדרך.

המשמעות היא שכל זמן שה-CPU משקיע במעבר הזה הוא זמן שבו הוא לא מריץ קוד אמיתי של אף תהליך. אם המעברים האלה קורים לעיתים קרובות מדי, ה-CPU מבזבז יותר זמן על שמירה ושחזור מצבים ופחות זמן על עבודה אמיתית - וזה פוגע בביצועים.

## טכניקות ניצול

באגים ב-scheduler עלולים להוביל ל-race condition exploits.

Race Condition הוא מצב שבו התוצאה הסופית של פעולה תלויה בסדר ובעיתוי המדויק שבו שניים או יותר תהליכים/threads מבצעים פעולות - ואם הסדר הזה משתנה, התוצאה עלולה להיות שונה, ולעיתים לא צפויה או לא בטוחה.

ואם נדבר סייברית, ויותר ספציפית בהקשר של הסלמת הרשאות, חולשות Race condition מנצלות את הפער בזמן שבין בדיקת האבטחה לבין השימוש במשאב, כך שהתוקף יכול לשנות את המשאב בזמן הזה ולעקוף את הבדיקה.

ה-scheduler, על ידי ביצוע מהיר של context switching בין threads, עלול בלי כוונה לאפשר מרוצים כאלה אם קוד הקרנל לא מסונכרן. לדוגמה, הפגיעות המפורסמת (Dirty COW CVE-2016-5195) ניצלה Race condition בין שני kernel threads - אחד כתב באופן מתמשך ל- copy-on-write mapping של קובץ ( copy on write - מנגנון בזיכרון שגורם לכך שכאשר שני תהליכים חולקים דף זיכרון לקריאה בלבד, ואם אחד מהם מנסה לכתוב אליו - ה-kernel עושה עותק פרטי עבורו, במקום לשנות את המקור), בעוד השני קרא שוב ושוב ל-Syscall שנועדה לאפס את המיפוי הזה. בעצם, במילים פשוטות, מה שקרה הוא:

Thread אחד שניסה ליצור עותק פרטי של קובץ כלשהו כדי שיוכל לכתוב אליו, ו-Thread שני שמנסה לשחרר את המיפוי של אותו קובץ.

התוצאה של זה הייתה הרסנית. משתמש ללא הרשאות הצליח לקבל גישת כתיבה לקבצים בבעלות root שהוגדרו לקריאה בלבד. Dirty COW הצליח כי ה-scheduler ביצע context switches בין ה-threads בדיוק ברגעים הנכונים כדי להטעות את מנגנון ה-copy-on-write.

הסבר יותר מפורט על פגיעות זו קיים בגיליון 77.

אם אתם רוצים לפתח קרנל (א' כל בבקשה אל), אתם חייבים להשתמש ב-locks כדי להגן על נתונים משותפים. אחרת, תוקף עלול לנצל את המרוץ הזה כדי להסלים הרשאות (למשל, על ידי שינוי קובץ לאחר בדיקת האבטחה אך לפני השימוש).

דוגמה קלאסית היא Time-of-check-to-time-of-use במערכות קבצים: תוכנית setuid עם הרשאות root עשויה לבצע ()stat על קובץ (לבדוק הרשאות) ואז ()open עליו. באותו הזמן, תוקף יכול לתזמן rename על הקובץ, כך שייפתח קובץ אחר לגמרי (אולי שלו, אולי זדוני, מי יודע). מרוצים כאלה אינם באג ישיר ב-scheduler, אך ה-scheduler מאפשר את ההתקפה על ידי context switching בזמן המדויק.

בעבר גם מתקפות (Denial-of-service DoS) היו נושא נוסף לדאגה. תוקף היה יכול לבצע fork למספר גדול של תהליכים או להעמיס על ה-CPU כדי לגרום לכך שמשימות קריטיות לא יקבלו משאבים. ה-scheduler כולל מנגנוני הגנה (כמו nice levels) כדי להתמודד עם זה, אך היסטורית נעשה שימוש ב-fork-bombs או לולאות כבדות ל-CPU כדי לנעול מערכות. כיום ניתן להגביל את כמות תהליכים למשתמש פשוט – באמצעות RLIMIT_NPROC (פקודת ulimit -u מציגה כמה תהליכים משתמש יכול ליצור), אשר מונעת ממשתמש רגיל ליצור מספר אינסופי של תהליכים.

תהליכים זדוניים יכולים גם לנצל עדיפויות scheduling - למשל, קביעת real-time priority ללא מגבלות מתאימות עלולה לאפשר לתהליך עוין להשתלט על ה-CPU.

## ניהול זיכרון: Virtual Memory, Paging, and Allocation

בדומה לווינדוס, לינוקס משתמשת בזיכרון וירטואלי כדי לתת לכל תהליך מרחב כתובות מבודד. מרחב הכתובות הווירטואלי של כל תהליך מחולק ל-user-space (לקוד ונתוני האפליקציה) ו-kernel-space (שמורה למערכת ההפעלה). למשל, ב-64 bit, החלק התחתון של מרחב הכתובות הוא זיכרון המשתמש (שונה לכל תהליך), בעוד החלק העליון הוא מרחב הכתובות של ה-kernel (ממופה בכל process לשם יעילות, אך מוגן כך שקוד משתמש לא יכול לגשת אליו).

MMU (קיצור של Memory Management Unit) וטבלאות עמודים (page tables) מטפלות בתרגום בין כתובות וירטואליות לכתובות פיזיות ב-RAM. הקרנל שומר עבור כל תהליך טבלת עמודים מרובת-שלבים (למשל level-4 או level-5 ב-x86-64 ) שממפה עמודים וירטואליים ל-page frames פיזיים. כאשר תהליך מנסה לגשת לכתובת זיכרון, ה-MMU בודק בטבלת העמודים את המיקום הפיזי המתאים.

מנגנון זה לא רק יוצר את האשליה שלפיה לכל תהליך יש זיכרון רציף שמתחיל בכתובת 0, אלא גם אוכף בידוד (תהליך אחד לא יכול לראות את הזיכרון של אחר אם הוא לא ממופה בטבלת העמודים שלו) והגנה (למשל, עמודים יכולים להיות מוגדרים כ-read-only, או no-execute). אם תהליך ניגש לכתובת שאין לה מיפוי תקף, ה-CPU מייצר page fault, והקרנל או טוען את העמוד (אם הוא זיכרון חוקי שנמצא בדיסק - swapping) או הורג את התהליך במקרה של גישה לא חוקית.

דיברנו על pages בזיכרון, איך הקרנל יודע להקצות ולנהל אותם? הכל בזכות מערכת חברית במיוחד (סטגדיש) בשם Buddy system!

Buddy System מחלק את הזיכרון לבלוקים בגודל של N^2 דפים. הזיכרון הפנוי נשמר ברשימות של בלוקים (buddies) בגדלים שונים של חזקות של 2. כאשר מתקבלת בקשה להקצות זיכרון (למשל, 8 דפים), ה-buddy allocator יחפש את הבלוק הקטן ביותר שיכול לענות על הבקשה. אם נמצא רק בלוק גדול יותר (נניח 16 דפים), הוא יחלק אותו לשניים, ישמור אחד כהקצאה והשני יסמן כזמין. כאשר משחררים זיכרון, ה-buddy allocator ינסה למזג buddies בחזרה לבלוקים גדולים יותר אם אפשר. השיטה הזו יעילה להקצאות בגודל של דף ומעלה, אבל עלולה לסבול מפרגמנטציה בהקצאות קטנות.

להקצאות קטנות יותר (אובייקטים של buffers לדוגמה), הקרנל משתמש ב-slab allocator שמנהל caches של אובייקטים. ה-slab allocator שומר caches עבור סוגי אובייקטים בשימוש תדיר, כאשר כל cache מחלק מראש דפים ל-slots בגודל קבוע עבור אותו אובייקט. כאשר הקרנל צריך מבנה חדש (למשל task_struct או inode), הוא יכול פשוט לקחת אובייקט מאותחל מראש מה-slab cache המתאים. שיטת ה-slab caching משפרת ביצועים וממחזרת זיכרון ביעילות, תוך הפחתת פרגמנטציה בהשוואה להקצאה ישירה מ-buddy system.

באופן כללי בלינוקס הזיכרון מחולק לשני אזורים עיקריים - User Space ו-Kernel Space. User Space הוא האזור שבו רצות האפליקציות והתהליכים של המשתמש, בעוד Kernel Space הוא האזור שבו רץ הקרנל וכל הקוד הקריטי שמנהל תהליכים, זיכרון וכו'.

הקוד והנתונים של ה-kernel נמצאים ב-kernel-space מוגן, שנגיש רק כשה-CPU נמצא ב-Kernel Mode. תהליכי user-space לא יכולים לקרוא או לשנות ישירות זיכרון של ה-kernel או של תהליכים אחרים. גם כאשר תהליך מפעיל system call ונכנס ל-kernel mode, כל מצביע (pointer) שהוא שולח מה-user memory חייב להיבדק ולהיות נגיש רק באמצעות פונקציות בטוחות כמו copy_from_user/copy_to_user, שמוודאות שהכתובות שסיפק המשתמש תקינות ונגישות. כך נשמרת ההפרדה המלאה בין תחומי הזיכרון.

לכל תהליך במערכת יש מרחב כתובות וירטואלי נפרד, אך החלק של ה-Kernel זהה בין כל התהליכים. במצב User Mode לא ניתן לגשת לחלק הזה, וניסיון לעשות זאת יגרום לשגיאת גישה (segmentation fault). המבנה הזה מאפשר שכאשר תהליך מבצע System Call, המעבר ל-Kernel Mode יהיה מהיר, שכן ה-CPU יכול להשתמש באותו זיכרון Kernel ללא צורך בהעתקות נוספות.

בשנת 2018 התגלתה חולשת Meltdown, שאפשרה במעבדים מסוימים לקרוא זיכרון קרנלי מתוך User Space באמצעות מנגנון Speculative Execution, ובכך לעקוף את ההפרדה הזו. הפתרון שהוטמע בלינוקס נקרא KPTI (Kernel Page Table Isolation). בעזרת KPTI, כאשר ה-CPU נמצא במצב User, הזיכרון הקרנלי כלל אינו ממופה ב-Page Table של התהליך, למעט Stub מינימלי הדרוש למעבר למצב לקרנל. ברגע שהמעבד עובר ל-Kernel Mode, לדוגמה בזמן System Call או Interrupt, ה-Kernel מחליף ל-Page Table שמכיל גם את זיכרון הקרנל, וכאשר חוזרים ל-User Mode מוחזר ה-Page Table המצומצם.

אם תוכנית במצב user מנסה לקרוא כתובת של הקרנל, היא תקבל fault או (עם KPTI) הכתובת כלל לא תופיע ב-page table שלה. הפרדה זו מונעת מפוגענים ב-userland לשנות או להשפיע ישירות על מרחב הכתובות הקרנלי.

## טכניקות ניצול

**Buffer Overflows:** טכניקה זו מתרחשת כאשר ה-kernel מעתיק יותר נתונים ל-buffer בזיכרון ממה שהוקצה לו, ובכך כותב על זיכרון סמוך. בקרנל מצב כזה מסוכן במיוחד, משום שהוא יכול לשכתב מבנים קריטיים כמו מצביעי פונקציות, פרטי הרשאות (credentials), או משתנים קריטיים למערכת.

לדוגמה, ב-2021 התגלתה חולשה במודול הרשת TIPC (CVE-2021-43267) שהייתה heap buffer overflow: הקוד הקצה buffer על סמך ערך גודל שנשלט ע"י התוקף, אך השתמש בערך אורך אחר שסופק גם הוא ע"י התוקף (בלי בדיקה מספקת) בעת קריאת memcpy. תוקף היה יכול לשלוח פקטה זדונית שגרמה לקרנל להקצות מקטע heap קטן ואז לכתוב מעבר לגבולותיו, מה שהוביל לשכתוב נתונים סמוכים ב-heap. ניצול overflow כזה אפשר השגת הרצת קוד מרוחק על הקרנל. לא נעים בלשון המעטה. לרוב, ניצולים מהסוג הזה ישכתבו control data (כמו כתובות החזרה ב-stack), או state data חשוב (כמו credentials או flags) כדי לבצע הסלמת הרשאות.

**Use-After-Free ו-Heap Spraying:** מתקפות UAF מתרחשות כאשר ה-kernel ממשיך להשתמש במצביע לאובייקט לאחר שהוא שוחרר והוחזר ל-allocator. תוקפים יכולים לנצל זאת ע"י גרימה לכך שה-allocator יקצה מחדש את אותו זיכרון לאובייקט שנשלט ע"י התוקף, ובכך לגרום ל-kernel לעבוד על נתונים זדוניים כאילו היו האובייקט המקורי.

לדוגמה, תוקף פותח את הקובץ /dev/ptmx המון פעמים (למשל 100). כל פתיחה כזו גורמת לקרנל להקצות אובייקט חדש מסוג tty_struct בתוך ה-heap.

tty_struct הוא מבנה נתונים שנמצא בקוד שמנהל את מערכת הטרמינלים, הקונסולות ודברים דומים לכך. בין היתר, כל tty_struct מחזיק מצביע לטבלה שנקראת ops table - טבלה של מצביעי פונקציה.

הקרנל משתמש בה כדי לדעת אילו פונקציות להריץ כאשר תהליך של המשתמש עושה פעולות על הטרמינל (כמו ioctl נגיד). אם יש חולשת overflow או UAF, התוקף יכול לשכתב אחד ה-tty_struct האלו.

באופן ספציפי - הוא יכול לשכתב את ה-ops pointer שלו. במקום שאותו מצביע יצביע לטבלה אמיתית של פונקציות, הוא יצביע לכתובת שהתוקף בחר.

כדי להפוך ניצול כזה לאמין, נהוג להשתמש ב-Heap Spraying - הקצאת מספר רב של אובייקטים מסוג מסוים כדי "לרסס" את ה-heap בתבנית צפויה. המטרה היא למלא אזורי זיכרון פוטנציאליים של הקורבן בנתונים נשלטים. Heap Spraying היא למעשה שיטה לעצב את פריסת ה-heap. ה-slab allocators של הקרנל מנסים לבצע רנדומיזציה של מיקומי הקצאות ויש בהם מנגנוני איתור באגים כמו KASAN (קיצור של Kernel Address Sanitizer), אך תוקפים מעצמתיים (ומשועממים מאוד) יכולים לעיתים עדיין "לפסל" את ה-heap באמצעות רצפים מתוכננים של הקצאות ושחרורים כדי למקם את ההקצאה הבאה בדיוק היכן שירצו.

**Memory Corruption:** בסופו של דבר, רוב טכניקות הניצול של זיכרון מכוונות להסלמת הרשאות. על-ידי פגיעה בזיכרון הקרנל, תוקף יכול למשל לדרוס את מבנה ה-struct cred של התהליך שלו כדי לקבל הרשאות root, או לבטל דגלי אבטחה. בקרנל, המבנה struct cred מכיל את מזהי המשתמש והקבוצה (UID/GID) ואת ה-capabilities של תהליך. ניצולים בקרנל לעיתים מחפשים בזיכרון את ה-struct cred של התהליך הנוכחי ודורסים אותו (או עושים שימוש חוזר באובייקטים של cred ששוחררו) כדי להגדיר UID=0.

קיימות גם טכניקות כמו "return-to-direct-mapped memory" - מאחר שהקרנל ממפה ישירות את הזיכרון הפיזי במרחב הכתובות שלו, overflow יכול למקם מבנה מזויף בזיכרון במקום שבו הקרנל עלול להשתמש בו.

**Bypassing Memory Isolation:** חלק מהניצולים מכוונים לעקוף את מנגנון ההפרדה הבסיסי של הזיכרון. לפני שפותחו מנגנוני הגנה, משתמש יכול היה למפות זיכרון בכתובת 0 ולנצל באגים של NULL pointer dereference בקרנל. אם הקרנל ניסה לגשת בטעות למצביע NULL, בפועל הוא היה ניגש לעמוד זיכרון שנשלט על-ידי המשתמש, ובכך אפשר להריץ קוד זדוני. בקרנלים מודרניים נמנע מיפוי כתובות נמוכות על-ידי משתמשים, ותכונות חומרה כמו SMAP למשל מונעות מהקרנל להריץ קוד ישירות מ-User Space או לגשת לזיכרון משתמש ללא בדיקה מתאימה. בנוסף, מתקפות Side-Channel כמו Meltdown CVE-2017-5754 הצליחו לפרוץ את מנגנון ההפרדה על-ידי קריאת זיכרון קרנל באמצעות speculative execution (מעין "ניחוש" של הקרנל איזה הוראה תתבצע הבאה). בתגובה לכך, פותח KPTI, שכתבתי עליו לעיל.

## System Calls - ממשק לקרנל

תוכניות שרצות ב-User Space אינן יכולות לגשת ישירות לחומרה או לפונקציות של הקרנל; עליהן לבקש שירותים באמצעות קריאות מערכת (sycalls) - נקודות הכניסה המוגדרות ממצב user ל-Kernel Mode. קריאת מערכת דומה לקריאה לפונקציה בקרנל, אך מיושמת באמצעות פקודת CPU מיוחדת או interrupt

(ארחיב על כך בחלק הבא של המאמר) שגורמות ל-trap - מצב שבו המעבד עוצר את רצף הביצוע הרגיל ועובר באופן מתוכנן למצב קרנל. לדוגמה, בארכיטקטורת x86-64, קוד משתמש מפעיל syscall על-ידי טעינת מספר הקריאה ל-rax, ואז ביצוע הפקודה syscall.

כאשר trap מתרחש, ה-CPU מבצע מספר פעולות אוטומטיות:

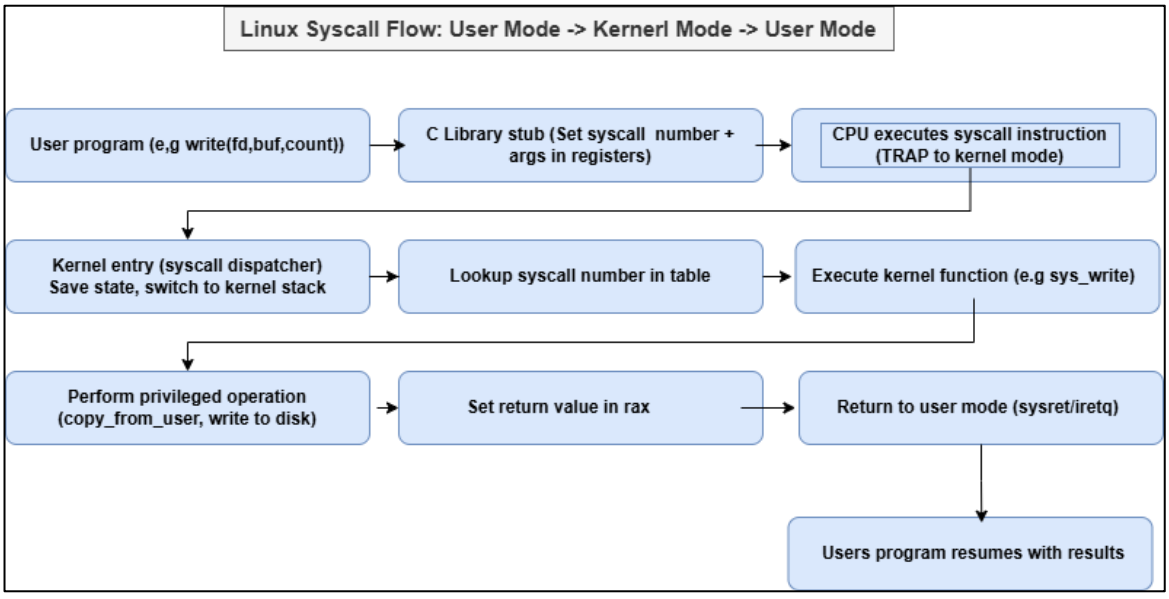
הוא עובר למצב 0 ring

עובר ל-stack קרנל ייעודי

קופץ לנקודת כניסה מוגדרת מראש בקרנל - ה-syscall dispatcher

בשלב זה התהליך נמצא "בתוך הקרנל" ומריץ קוד. ה-dispatcher של הקרנל שומר את הרישומים של המשתמש, ולאחר מכן משתמש במספר הקריאה כדי לאתר בטבלת ה-syscalls (מערך של מצביעי פונקציות) את המימוש של הקריאה. לדוגמה, אם הערך ב-rax הוא 60 (המספר של sys_exit ב-x86-64 ), ה-dispatcher יקרא לפונקציה של exit.

עכשיו, איך המעבר בין פונקציה פשוטה ב-User Mode להרצת קוד בקרנל מתרחש? נבין באמצעות דוגמה. תוכנית משתמש קוראת לפונקציה ()write. ספריית ה-C מכינה את הארגומנטים ומבצעת את קריאת המערכת write. ה-CPU עובר לKernel Mode, קוד הכניסה של הקרנל שומר את מצב ה-CPU ומזהה את מספר ה-syscall. לאחר מכן הוא קורא לפונקציה sys_write דרך טבלת ה-syscalls. המימוש של sys_write יעתיק את הנתונים מה-buffer של המשתמש לזיכרון הקרנל ויבצע את פעולת הכתיבה. בסיום, הוא קובע ערך חזרה (למשל מספר הבתים שנכתבו) ב-rax ומחזיר. בסופו של דבר, ה-dispatcher של ה-syscall מכין את החזרה ל-user mode: הוא משחזר את מצב ה-CPU והרישומים השמורים של התהליך ומחזיר את המעבד ל- user mode. מנקודת מבט של תוכנית המשתמש, היא פשוט קראה ל- ()write וקיבלה תוצאה בחזרה - מבלי לראות את כל מנגנון המעבר המורכב שמתרחש מאחורי הקלעים.



במהלך syscall, הקרנל רץ בהרשאות מלאות ולכן הוא חייב לאמת את כל הקלטים שמגיעים מ-User Space לפני שהוא משתמש בהם. הדבר כולל מצביעים - חייבים לוודא שהם מצביעים על זיכרון ששייך באמת לתהליך הקורא ונמצא ב-user space, ולא על זיכרון הקרנל - וגם גדלים, כדי למנוע buffer overflow בקרנל.

לדוגמה, הקרנל לא יבטח בצורה עיוורת במספר שמגיע מ-user space אם הוא מתכוון להשתמש בו כאינדקס במערך או כאורך - הוא יוודא שהוא נמצא בטווח הצפוי. אם syscall מצפה למצביע למבנה, הקרנל עשוי קודם להעתיק את המבנה הזה ל-buffer פנימי שלו, או לכל הפחות לוודא שכל שדה בו בטוח לשימוש.

בדיקה של מצביעים חשובה במיוחד: הקרנל לעולם לא ידרוס או ייגש ישירות לכתובת שסופקה מה-user אם היא מצביעה למרחב הקרנלי. אילו היה עושה זאת, תוקף יכול היה להעביר ל-syscall כתובת מתוך זיכרון הקרנל, ולגרום לכך שהקרנל יקרא או יכתוב לזיכרון שלו עצמו - מה שהיה מפר את מנגנון ההפרדה בין הרשאות. נאמר כספוילר להמשך - יש שהיו אומרים שמה שכתבתי עכשיו זה "הוראות שנכתבו בדם".

ניתן דוגמה היפותטית: אם הקרנל לא היה בודק את המצביע בקריאת write(fd, buf, count),

תוכנה זדונית הייתה יכולה להעביר מצביע למבנה פנימי של הקרנל. הקרנל היה מעתיק את התוכן הזה ואולי כותב אותו לקובץ, מה שהיה מוביל לדליפת מידע מהקרנל למשתמש. להיפך - אם בקריאת read() לא הייתה מבצעת בדיקת מצביע, ייתכן שהקרנל היה כותב לתוך כתובת בזיכרון הקרנלי, ובכך משחית את הנתונים שלו עצמו.

בקרנלים מודרניים נמנעים מכך באמצעות פונקציות כמו copy_from_user(), שנכשלות אם הכתובת שסיפק המשתמש אינה כתובת תקפה ב-user space. בנוסף, נעשה שימוש במנגנוני חומרה כמו SMAP (Supervisor Mode Access Prevention), שמונעים מהקרנל לגשת ישירות לזיכרון של משתמש אלא אם נאמר אחרת.

### טכניקות ניצול

קריאות מערכת הן משטח התקיפה העיקרי עבור תוקפים, משום שדרכו ה-userland מתקשר עם ה-kernel. רבות מהחולשות ההיסטוריות בלינוקס נבעו מפגיעויות במימושי syscalls. אפשר להתייחס לתקיפות הפוטנציאליות דרך syscalls ממש כמו השיר דיינו בהגדת פסח.

אם ה-syscall handler לא מאמת כראוי את הארגומנטים, תוקפים יכולים לנצל זאת, דיינו. כפי שצוין, היעדר ולידציה של מצביעים (pointers) עשוי לאפשר גישה לזיכרון ה-kernel. לדוגמה, נניח syscall היפותטי sys_mycopy(dst, src, len) שממומש בלי לבדוק ש-dst היא כתובת ב-user-space. תוכנה זדונית יכולה לקרוא ל-syscall עם dst שמצביע למבנה רגיש של ה-kernel ו-src לנתונים בשליטתה – ה-kernel עלול אז להעתיק את הנתונים למבנה שלו עצמו ולדרוס אותו. זאת הסלמת הרשאות פשוטה במלוא מובן המילה. רק חבל שהעולם לא כל כך פשוט (או בעצם לא חבל, תלוי את מי שואלים). כיום, מפתחי הקרנל דואגים מאוד לוולידציה על כל הפרמטרים וקריאות המערכות השונות.

דוגמה היסטורית נהדרת לניצול חולשה מסוג זה היא CVE-2014-3153. באג זה היה ב-futex syscall, שמנהל Userspace Mutex. הפגיעה התמקדה בפונקציה futex_requeue() שאיפשרה להעביר תהליכים מ-wait queue אחת לאחרת. הבעיה נבעה מבדיקות חסרות על פרמטרים שהגיעו מ-user space, במיוחד במנגנוני Priority Inheritance. תוקף יכל לנצל את החוסר בסנכרון ולגרום ל-race condition, שבסופו הקרנל עבד עם מצביעים לא תקינים או שכבר שוחררו. ניצול מעשי (כמו ה-Towelroot exploit באנדרואיד) שינה את מבנה ה-cred של התהליך הנוכחי כדי להפוך את ה-UID ל-0.

חלק מה-syscalls מקבלים נתונים מ-user space – לדוגמה, read(), או קריאות ioctl() שמעבירות מבנים. אם קוד הקרנל שמטפל בנתונים האלה לא מבצע בדיקות אורך בצורה נכונה, עלול להתרחש Buffer Overflow ולגרום לכתיבה מעבר לגבולות ה-kernel buffer, דיינו.

דוגמה חדשה יחסית היא החולשה בקריאת sendmsg() (CVE-2019-20386). קריאת מערכת זו מאפשרת שליחת הודעות עם control messages, כולל העברת descriptors בין תהליכים. כאשר תהליך שלח control message מסוג scm_rights עם נתונים מסוימים, הקרנל לא בדק את אורך הקלט מול גבולות buffer, ומשם הדרך להסלמת הרשאות או להקרסת המערכת קצרה מאוד.

ישנן syscalls שבבסיסן לגיטימיות אך עלולות להיות מסוכנות אם מופעלות ללא בקרות מתאימות, דיינו. תוקפים מחפשים דרכים לקרוא להן גם כשאין להם הרשאות לכך. למשל, ה-kexec_load syscall מאפשר למשתמש root לטעון קרנל חדש לזיכרון – ברור שמשתמש לא-מורשה לא אמור לקרוא לזה. אם מתרחשת תקלה זמנית בהרשאות (אפילו הרשאת ה-sudo הכי קטנה), תוקף יכול להשתמש ב-syscall כזה כדי להסלים את ההרשאות שלו לצמיתות, למשל על-ידי טעינת קרנל משלו.

אני אצטט עכשיו משפט שמפתחים מאוד אוהבים לומר - זה לא באג זה פיצ'ר. אלה הן קריאות מערכת לגיטימיות שקיימות בעבור פיתוח לגיטימי, אך אם לא מגבילים אותם בצורה מספקת - נחשפים למרחב תקיפה שלם.

אף על פי שלא מדובר בדרך כלל בהסלמת הרשאות ראשונית, לאחר שתוקף כבר השיג גישת כתיבה לקרנל הוא יכול לבצע hooking לטבלת ה-syscalls כחלק מ-rootkit. הרעיון הוא להחליף מצביע בטבלת ה-syscalls כך שיצביע לפונקציה זדונית במקום למימוש המקורי.

באופן זה התוקף יכול ליירט קריאות מערכת – למשל, לחבר מחדש את open כדי להסתיר קבצים מסוימים, את getdents כדי להסתיר תיקיות, או את execve כדי להעניק הרשאות גבוהות אוטומטית לכל תוכנית שמורצת.

במערכות מודרניות קיימות הגנות כמו קרנל עם זיכרון לקריאה בלבד, וכן מנגנוני integrity checkers שמסוגלים לעיתים לזהות טבלאות syscalls שהשתנו. צעדים אלה נועדו להפחית את הסיכוי ש-rootkits יצליחו להשתמש ב-Syscall Table Hooking מבלי להתגלות.

Interrupts הם אותות מחומרה או מתוכנה שעוצרים זמנית את ריצת ה-CPU ומעבירים אותו ל-handler routine. כאשר התקן (כמו מקלדת לדוגמה) דורש תשומת לב, הוא שולח Interrupt Request (IRQ) ל-CPU. ה-CPU, עם קבלת ה-interrupt, שומר את ההקשר הנוכחי שלו (בדומה ל-context switch) וקופץ ל-Interrupt Service Routine (ISR) של אותו interrupt. כל IRQ חומרתי ממופה ל-ISR, וה-kernel מגדיר זאת ב-Interrupt Descriptor Table (IDT). חשוב לציין שכאשר ISR רץ, הוא נמצא ב-context מיוחד: אין לו תהליך משתמש משויך. המשמעות היא ש-ISR לא יכול "לישון" או לבצע עיבוד ממושך - עליו לפעול במהירות ולאשר/לאפס את ה-interrupt בהתקן. דומה מאוד לאיזושהי מערכת הפעלה לא? שכחתי איך קוראים לה. הלכתי להסתכל מעבר לאדן החלון. אולי אזכר.

במערכות מרובות-מעבדים (שזה רוב המערכות כיום), ניתן לנתב Interrupts אל ליבות ספציפיות. כברירת מחדל, ייתכן שכל ה-hardware interrupts ינותבו ל-CPU0, אך זה אינו אופטימלי משום שזה עלול להעמיס על מעבד יחיד. IRQ balancing הוא הפרקטיקה של הפצת עומס הטיפול ב-interrupts בין ליבות שונות. הקרנל מאפשר להגדיר עבור כל interrupt ערך בשם IRQ affinity - כלומר bitmask של ליבות ה-CPU המורשות לטפל באותו interrupt.

במערכות רבות פועל תהליך משתמש בשם irqbalance, כלומר daemon שמבצע התאמות אוטומטיות בפרקי זמן קבועים: הוא משנה את ה-affinities כדי לאזן את העומס, ומעביר interrupts מליבה עמוסה לליבה פנויה.

### טכניקות ניצול

בצורה כללית אפשר להגיע למצב של Race Condition דרך Interrupts. כלומר, ה-Interrupt עצמו הוא לא הניצול, אך דרכו אפשר להגיע להרצת קוד בקרנל. לדוגמה, אם הקרנל מסמן buffer כמשוחרר ואז מפעיל פעולה חומרתית - interrupt שמגיע בזמן הזה יכול להקצות מחדש את ה-buffer לפני שהפעולה החומרתית משתמשת בו, מה שיוצר UAF. תוקפים שיש להם שליטה מסוימת על מקורות interrupts יכולים לתזמן מצבים כאלה. רבים מהדרייברים של ההתקנים כוללים interrupt handlers שהם חלק מהקרנל. באגים בתוך interrupt handler מסוכנים במיוחד משום שהם רצים ב-IRQ context, לרוב בהרשאות גבוהות ועם מעט מאוד הגנות (למשל, לעיתים לא כל בדיקות האבטחה של system calls מתבצעות שם). אם תוקף יכול לשלוח נתונים או אותות שמנצלים חולשה ב-ISR, הוא יכול להריץ קוד ב-Kernel Mode או להפיל את המערכת.

דוגמה קונקרטית היא ניצול בדרייבר USB - CVE-2016-2384. כאשר התקן כזה חובר, תת-המערכת של ה-USB בקרנל (שרצה בחלקה בזמן interrupt בעת חיבור התקנים) טיפלה באופן שגוי ב-descriptor לא צפוי ושחררה אובייקט פעמיים. כשהקרנל משחרר אובייקט פעמיים, ה-allocator של ה-heap עלול "לחשוב" שהזיכרון פנוי, ובפועל לשמור עליו ברשימת Free. תוקף מעצמתי יכול לעשות Heap Spraying (כתבתי על כך לעיל) ובסופו של דבר לקבל הרצת קוד ב-kernel.

## סיכום

בסופו של דבר, הקרנל של לינוקס הוא לא רק "צינור" שמחבר בין המשתמש לחומרה – הוא המוח שמנהל את כל המערכת. מהניהול של זמן המעבד בעזרת ה-CFS, דרך חלוקת הזיכרון במנגנוני Paging ו- Buddy Allocation, ועד לניהול קריאות המערכת והגנה על המרחב הקריטי של הקרנל – הכל עובד בהרמוניה עד שמישהו מנסה לשבור את הכללים. לאורך השנים הקרנל השתדרג כל הזמן, ועבר שיפורים שצמצו משטחי תקיפה קיימים אך גם יצרו משטחי תקיפה חדשים. הקרנל הלינוקסי הוא עולם אינסופי, שמכיל בתוכו כל הזמן הפתעות.

## על המחבר

שמי ירין הרמן, וזהו המאמר השלישי שלי. לכל שאלה, תגובה או טענה, מוזמנים לכתוב לי במייל:

yarinherman05@gmail.com

## מקורות

- https://documentation.ubuntu.com/real-time/latest/explanation/schedulers/
- https://linux-kernel-labs.github.io/refs/heads/master/so2/lec3-processes.html
- https://hackmd.io/@bachtam2001/BkZkudoLq
- https://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/dirty-cow/index.html
- https://www.form3.tech/blog/engineering/linux-fundamentals-user-kernel-space
- https://www.geeksforgeeks.org/operating-systems/operating-system-allocating-kernel-memory-buddy-system-slab-system/
- https://www.sentinelone.com/labs/tipc-remote-linux-kernel-heap-overflow-allows-arbitrary-code-execution/
- https://www.kernel.org/doc/gorman/html/understand/understand014.html
- https://santaclz.github.io/2024/01/20/Linux-Kernel-Exploitation-Heap-techniques.html
- https://thinkty.net/general/2024/04/29/bottom_half.html
- https://www.usenix.org/conference/usenixsecurity21/presentation/lee-yoochan
- https://enterprise-support.nvidia.com/s/article/what-is-irq-affinity-x
- https://xairy.io/articles/cve-2016-2384
- https://linux-kernel-labs.github.io/refs/pull/282/merge/lectures/syscalls.html