digital whisper
Technologic Papers

# עכשיו! מתי נסנכרן?

מאת עומר כספי ו-0xDEAD6057

## הקדמה

ב-19 באוקטובר געשה קהילת הלינוקס ברחבי העולם בעקבות גילוי פרצה חמורה בקרנל, החולשה זכתה לכינוי: "dirtyc0w". הפרצה "ישנה" במשך כ-9 שנים(!) ונחשפה במקרה על ידי פיל אוסטר, חוקר אבטחה אשר קבצים חשודים עלו בחכתו. כעת, לא ידוע מי אותם גורמים אשר ניצלו את אותה החולשה או כמה זמן היא נמצאת בשימוש, אך השפעת הפרצה רחבה ומאפשרת בין השאר העלאת הרשאות. כרגע ידוע כי היא קיימת ברוב ההפצות המרכזיות של לינוקס וכן במערכת ההפעלה אנדרואיד.

הבאג עצמו מתרחש בשל Race Condition בקרנל, הליבה, של מערכת ההפעלה. מאמר זה יסקור את המשמעות של Race Condition, דרכים למנוע אותו וכן רקע מקדים לצורך הבנה של המושג. חשיבות המודעות לנושא תודגם בסוף המאמר באמצעות החולשה החדשה, אשר הייתה יכולה להימנע אם המתכנתים היו שמים דגש על ניהול משאבים בתוצרה תקינה, כפי שנציג במאמר.

## Context Switching

במחשב ביתי מודרני ממוצע יש למעבד הראשי (CPU) בין 2 ל-8 ליבות, אך בפועל רצים עליו מאות ולעתים גם אלפי תהליכים בו זמנית. אז איך מתרחש הפלא הטכנולוגי הזה? התשובה פשוטה: הם לא באמת רצים בו זמנית. בכל פרק זמן רצים מספר תהליכים בעוד השאר ממתינים לתורם וההחלפה מתרחשת מספיק מהר כדי לתת למשתמש חוויה כאילו הם קורים במקביל. על מנת לבצע זאת, יש לבצע פעולה הנקראת Context Switch (או בעברית, החלפת הקשר). ב-Context Switch נשמר תהליך שמוחלף ההקשר - מכלול התכונות הייחודיות הנוגעות למצב ריצתו (מיקום המחסנית, כתובת ההרצה הנוכחית וערכי אוגרי מעבד נוספים). התהליך המחליף יקבל את ההקשר שנשמר עבורו מהפעם האחרונה שבה רץ, או יקבל הקשר חדש אם זאת הפעם הראשונה שבה הוא מקבל זמן ריצה מהמעבד.

## זיכרון וירטואלי ודפדוף

כאשר תהליך רץ בלינוקס, מערכת ההפעלה מבטיחה לו שהזיכרון בו הוא משתמש נשאר פרטי ושאף תהליך אחר לא יבצע בו שינויים.[1] גם אם שני תהליכים כותבים ערך שונה לכתובת 0XDEAD6057, כל אחד מהם ייראה את הערך אשר נכתב במרחב הכתובות שלו מבלי להיות מודע לזה האחר. אך בסופו של דבר קיימת רק כתובת אחת ב-RAM שמספרה 0XDEAD6057, ורק ערך אחד יכול לאכלס אותה - כדי לפתור את הבעיה הנ"ל קיים **הזיכרון הווירטואלי**. זיכרון וירטואלי הוא למעשה מרחב כתובות אשר אינו מתאים בהכרח לזיכרון הפיזי - הזיכרון של ה-RAM. שכיח בזיכרון וירטואלי לציין כתובת אחת אשר בפועל מייצגת כתובת אחרת לגמרי בזיכרון הפיזי של המחשב. את תהליך ההמרה בין כתובת וירטואלית לכתובת פיזית מבצע רכיב הנקרא MMU (Memory Management Unit), אשר מוכל בתוך מרבית המעבדים המודרניים.

כאן לא נפתרות כל הסוגיות: מה קורה אם מאות התהליכים הרצים מנצלים בפועל יותר זיכרון וירטואלי מאשר הזיכרון הפיזי הקיים במערכת? לשם כך תוכנן מנגנון הנקרא דפדוף (Paging). המנגנון קובע כי כל מרחב הזיכרון יחולק לקטעים באורך קבוע - דפים[2]. דפים יוכלו להיות טעונים לרכיב זיכרון ראשי (RAM) או למשני (באופן שכיח, HDD או SSD). כאשר תהליך רץ מבקש גישה לזיכרון, תבדוק מערכת ההפעלה והמעבד[3] אם הדפים טעונים לזיכרון המשני, ואם כן, יעבירו אותם לזיכרון הראשי.

עם מנגנון הדפדוף באה בעיה גדולה: תהליך העברת הדפים בין הזיכרון הראשי והמשני אורך זמן רב וגורם לתקורה (Overhead) משמעותית. לשם כך פותחו מספר מנגנונים. אחד מהם הוא ה-Dirty Bit. זהו ביט בזיכרון אשר קיים לכל דף ונדלק כאשר מבצעים כתיבה אליו. אם המערכת רוצה להעביר דף מהזיכרון הראשי למשני, היא תבצע את האלגוריתם הבא (בהפשטה):

1. אם ה-Dirty Bit מכובה:
    a. אם קיים עותק של הדף בזיכרון המשני:
        i. אם העותק בזיכרון המשני לא נדרס:
            1. קפוץ להוראה 3
2. בצע העתקה של הדף מהזיכרון הראשי אל הזיכרון המשני
3. סמן את ההעברה כהושלמה

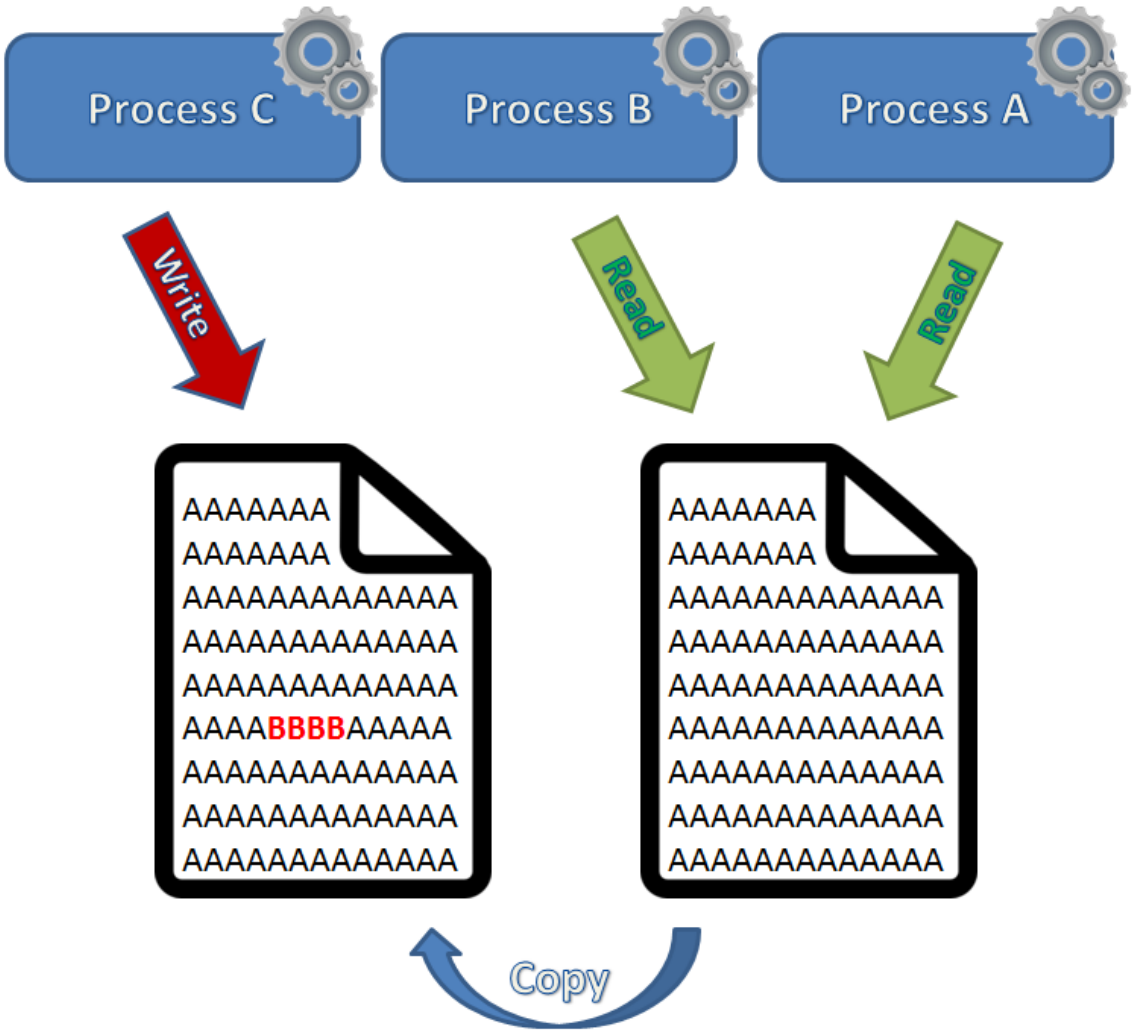
[1] במערכות הפעלה מסוימות קיימים API-ים ייעודיים לביצוע פעולות על מרחבי כתובות זרים. ב-linux, לדוגמה, קיימת הפסיקה ptrace שמאפשרת קריאה וכתיבה לזיכרון כמו גם שינוי מצב האוגרים ועוד מגוון אפשרויות.

[2] טרמינולוגית, דפים (pages) הם קטעי זיכרון וירטואלי בעוד מסגרות (frames) הן קטעי זיכרון פיזי. המאמר לא עושה הבחנה בין המושגים מפני שמדובר לרוב במונחים מקבילים.

[3] אופן ביצוע חיפוש הדפים משתנה בין מערכות הפעלה וארכיטקטורות מעבד שונות.

באופן בדומה, קיימת טכניקת COW (**C**opy-**O**n-**W**rite). בטכניקה זו, יכול להתקיים דף המשותף למספר תהליכים אשר חלה עליו מדיניות קריאה בלבד[4], אך אם אחד התהליכים יבצע כתיבה אל הדף, תוכנו יועתק אל דף חדש **ורק בעותק יבוצעו השינויים**. כך, אם תהליך מסוים ידרוס לדוגמה דפים של ספריות משותפות טעונות (לדוגמה, ספריות של Java או libc), הוא ישנה את התוכן רק לעצמו בעוד שאר התהליכים ייראו את התוכן המקורי.

להלן תרשים המציג את המקרה המדובר:




[4] קריאה בלבד (מאנגלית, Read Only) היא תכונה של משאב אשר ניתן רק לקרוא ממנו אך אי אפשר לכתוב אליו. מקובל לעתים לכנות גם משאבים עם הרשאות הרצה כ-Read Only, כאשר הדגש הוא שהמידע יישאר כפי שהיה.

## Race Conditions

לאחר שהסברנו כמה מנגנונים מרכזיים להבנת הפרצה dirty c0w, עלינו להסביר מה בעצם גורם לבאג, לטובת זאת נגדיר מהו Race Condition ושיטות פתרון שונות לסוגיה זו.

Race Condition הוא מצב אשר מתרחש כאשר בעקבות אי סנכרון בין תהליכים או תהליכונים (threads) אשר ניגשים לקטע קריטי[5] יכולה להיווצר התנהגות לא צפויה. את המשמעות של Race Condition וקטע קריטי נמחיש באמצעות דוגמה. בדוגמה יש שני threads אשר משתמשים במחסנית משותפת וכל אחד מהם מנסה להוציא מידע מהמחסנית:

```
import java.util.Stack;
class RaceClass extends Thread {
	public Stack<Integer> s;

	public RaceClass(Stack<Integer> s) {
		this.s = s;
	}
	public void run() {
		this.pop(s);
	}

	public void pop(Stack<Integer> s) {
		boolean isempty[6] = false;
		while (isempty == false) {
			//start of critical section
			if(s.empty() == false) {
				s.pop();
			} else {
				isempty = true;
			}
			//end of critcial section
		}
	}
}

public class main {
	public static void main(String[] args) {
		Stack<Integer> s = new Stack<Integer>();
		for(int i = 0; i < 50000 ; i++) {
			s.push(1);
		}
		Thread popper1 = new RaceClass(s);
		Thread popper2 = new RaceClass(s);
		popper1.start();
		popper2.start();
	}
}
```

[5] קטע קריטי הוא קטע קוד הניגש למשאב משותף בין מספר תהליכים. ביצוע הקוד עלול להשתבש אם מספר תהליכים נמצאים במקביל בקטע קריטי.

[6] הבאג משתחזר גם בלי המשתנה הלוקאלי isempty, המשתנה נועד לצורכי קריאות

בדוגמא הנ"ל יכול לקרות מצב הקיצון הבא:

- Thread מס' 1 יבדוק האם המחסנית ריקה, ויכנס לתנאי
- יתבצע Context Switch ל-thread מס' 2
- Thread מס' 2 יבדוק האם המחסנית ריקה, יכנס לתנאי ויבצע את ההוצאה מהמחסנית
- יתבצע Context Switch ל- thread מס' 1
- Thread מס' 1 יחזור לנקודה שבא עצר וימשיך משם. כלומר: הוא ינסה להוציא מהמחסנית
- המחסנית ריקה - לכן, בשפות שתומכות ב-exception לרוב יזרק exception. ובשפות אחרות יכולה להיות התנהגות חריגה (לדוגמה, ב-C יכולה להתבצע **כתיבה לזיכרון שלא שייך למחסנית**).

התוכנית הנ"ל תוציא במקרה ויזרק exception את ההודעה הבאה:

```
Exception in thread "Thread-0" java.util.EmptyStackException
      at java.util.Stack.peek(Unknown Source)
      at java.util.Stack.pop(Unknown Source)
      at raceClass.popper(main.java:21)
      at raceClass.run(main.java:12)
```

אך מה ניתן לעשות על מנת לפתור זאת? נוכל להשתמש במנגנוני סנכרון שמערכת ההפעלה ושפות התכנות מספקות לנו.

נתאר כמה מהם פה ונראה איך הם פותרים את הבעיה:

- **Mutex -** קיצור של Mutual Exclusion, מנגנון להגבלת גישה לקטע קריטי. בכל פעם ש-thread ירצה להיכנס לקטע הקריטי הוא ינסה לקחת בעלות על ה-mutex (lock); במידה והצליח, ייכנס לקטע הקריטי ולאחר שיסיים את הקטע הקריטי ישחרר את ה-mutex (unlock). במידה והוא לא הצליח לקחת בעלות על ה-mutex, ה-thread יחכה (כלומר יוותר על זמן העיבוד שלו) עד שזה שלקח את ה-mutex ישחרר אותו.

- **Semaphore** - מנגנון להגבלת גישה לקטע קריטי למספר מסוים של תהליכים. ה-Semaphore ממומש על ידי מונה שמאותחל למספר התהליכים אשר מורשים לגשת בו זמנית לקטע הקריטי. כאשר תהליך ירצה לגשת לקטע זה, הוא ינסה להוריד את המונה ב-1, אם המונה גדול מ-0 המונה ירד ב-1 והתהליך יכנס לקטע הקריטי. לעומת זאת, אם המונה הוא 0 התהליך יכנס לתור המתנה ויוותר על זמן העיבוד שלו עד שהמונה גדול מ-0. אז התהליך הראשון בתור יתעורר, יוריד את ה-semaphore ב-1 וייכנס לקטע הקריטי.

- **Spinlock** - מנגנון המתנהג בדומה ל-mutex, אך בניגוד אליו, כאשר הנעילה לא מצליחה הוא לא מוותר על זמן העיבוד.

קיימות עוד מגוון שיטות סנכרון כגון Barriers, Condition Variables ו-Futex אשר עליהן לא נפרט אך מידע אודותיהן מופיע במקורות המופיעים בסוף המאמר.

יש לציין שעל מנת שגם בנעילות לא יהיה מצבים של Race Conditions, המנגנונים הנ"ל ממומשים על ידי הוראות אטומיות (Atomic Instructions): הוראות אשר מערכת ההפעלה מתחייבת שיבוצעו ללא Context Switch (לרוב מדובר בפקודה אחת של המעבד).

כעת, אם נשתמש באחד ממנגנוני הסנכרון שהוסברו (mutex) הפעולה pop תראה כך:

```
import java.util.Stack;
import com.sun.corba.se.impl.orbutil.concurrent.Mutex;

class RaceClass extends Thread {
	static Mutex m = new Mutex();
	Stack<Integer> s;

	public RaceClass(Stack<Integer> s) {
		this.s = s;
	}

	public void run() {
		this.pop(s);
	}

	public void pop(Stack<Integer> s) {
		boolean isempty = false;
		while (isempty == false) {
			try {
				RaceClass.m.acquire();
			} catch (InterruptedException e) {
				continue;
			}
			//start of critical section
			if(s.empty() == false) {
				s.pop();
			} else {
				isempty = true;
			}
			//end of critcial section
			RaceClass.m.release();
		}
	}
}
```

```
public class main {

	public static void main(String[] args) {
		Stack<Integer> s = new Stack<Integer>();
		for(int i = 0; i < 50000 ; i++) {
			s.push(1);
		}
		Thread popper1 = new RaceClass(s);
		Thread popper2 = new RaceClass(s);
		popper1.start();
		popper2.start();
	}
}
```

ה-Mutex מבטיח שרק thread אחד כל פעם יוכל להיכנס לקטע הקריטי ולהוציא מהמחסנית, וכך ייפתר ה-Race Condition המדובר.

## Race Condition ואבטחת מידע

ההשלכות של Race Condition מרחיקות לכת בכל הנוגע לסוגיות כתיבת קוד מאובטח. ככל שהקוד אשר קיים בו הבאג קריטי יותר לתפקוד המערכת, כך השפעתו עליה תהיה קשה יותר והוא עלול במקרי קיצון להוביל לקריסה כוללת או לפרצות אבטחה חמורות. הדבר רלוונטי במיוחד במערכות גדולות אשר מנהלות מספר עצום של משאבים משותפים, כמו במערכות מצילות חיים. ואם לא די בכך, הבאג מתרחש באופן סטטיסטי, כלומר רק לפעמים, ולכן ייתכן שלא יעלה בכלל בבדיקות אשר בוצעו למוצר למרות שקיים סיכוי להופעתו בעתיד. כדי להימנע ככל האפשר מבאג זה, יש לסקור את הקוד באופן ביקורתי ולזהות משאבים משותפים.

עם זאת, מתכנתים הם בני אדם, והם טועים, ולכן יש להגדיר גם את בדיקות המוצר בהתאם. בדיקות מוצר אשר כוללות בדיקות עומסים, יעלו לעתים באגים סטטיסטיים כמו Race Condition.

## You dirty, dirtyc0w

Dirtyc0w היא פרצה אשר נגרמת בשל Race Condition באופן מימוש ה-COW בקרנל של לינוקס. כאשר הקרנל מבצע כתיבה אל הדף החדש, הוא עלול ב-context אחר להוריד את הדף מהזיכרון הראשי אל המשני באופן לא מתואם עם פעולת ה-COW. הבאג מאפשר בפועל לכתוב אל דפי זיכרון אשר אליהם קיימות למשתמש הרשאות קריאה בלבד, ובכך המשתמש בתורו יוכל להשיג את היכולת לכתוב לאותם קבצים, ספריות משותפות ועוד. למעשה, בהינתן ניצול מוצלח של פרצה זו ניתן להשיג Privilege Escalation[7] ולהגיע למצב של הרצת פקודות כמשתמש root על המכונה הנתקפת.

ב-Linux קיימת הפונקציה madvise שמאפשרת למשתמש "להציע" לקרנל כיצד יש לנהוג בדפים שברשותו. באמצעות קריאה לפונקציה עם הפרמטר MADV_DONTNEED, ניתן להצהיר שאיננו עושים שימוש בדף מסוים ובכך להעלות את הסיכוי שהמעבד יוריד את דף הזיכרון אל הזיכרון המשני. כתיבה תכופה לדף זיכרון בהרשאות Read Only תוך כדי קריאה לפונקציה madvise מ-thread אחר עלולה לגרום לכך שפעולת הכתיבה תתבצע על הדף המקורי ולא על זה המועתק.

החלק הבא מתאר את הפרצה באופן מעמיק ויהיה טכני ביותר, בהמשך נדון באופן ניצול החולשה ובפתרון הקיים.

[7] Privilege Escalation היא טכניקה באבטחת מידע בה משתמש חסר הרשאות, משיג הרשאות גבוהות יותר באמצעות ניצול פרצות קיימות.

הפונקציה הראשונה הנוגעת לפרצה היא ***get_user_pages***. פונקציה זאת היא פונקציה קרנלית הנקראת כאשר מתבצעים syscall-ים רבים, ביניהם לדוגמה: open, read, write. תפקיד הפונקציה הוא לוודא שהדפים המבוקשים טעונים לזיכרון הראשי ולהחזיר מצביעים אליהם.

בתוך פונקציה פנימית אשר נקראת על ידי get_user_pages,מבוצע קטע הקוד הבא:

```
retry:

   cond_resched();
   page = follow_page_mask(vma, start, foll_flags, &page_mask);
   if (!page) {
      int ret;
      ret = faultin_page(tsk, vma, start, &foll_flags,
                                    nonblocking);
      switch (ret) {
         case 0:
            goto retry;
```

באופן כללי, קטע הקוד עושה את הפעולות הבאות:

1. מאפשר ל-thread-ים נוספים לרוץ (cond_resched)
2. מנסה להשיג דף זיכרון (follow_page_mask)
3. אם לא הצליח להשיג את הדף, בודק מה מהות השגיאה (faultin_page)
4. אם לא הוחזרה שגיאה (נדון על סוגיה זאת בהמשך) תחזור לשלב 1

בהנחה שהדף שרצינו להוריד לזיכרון כבר ירד, אנו קוראים כעת ל-follow_page_mask ומגלים שהדף לא טעון לזיכרון, לכן תיקרא הפונקציה faultin_page. בפונקציה פנימית ייבדק אם הדף קיים בזיכרון ולאחר מכן ייבדקו הדגלים של הדף, במידה ויימצא כי הדף משותף בין מספר תהליכים, ייווצר דף חדש. הדף החדש יהיה עם הרשאות read only אך ה-Dirty Bit שלו יהיה דלוק (על מנת למנוע באגים קודמים שהיו באזור הזה). הפונקציה מחזירה 0 כדי לנסות לכתוב מחדש, והפעם - **על הדף החדש שנוצר**.

בניסיון השני, כבר קיים דף חדש, אך עדיין אין לו הרשאות כתיבה ולכן follow_page_mask לא תחזיר את הדף. לאחר קריאה ל-faultin_page ייבדק בשנית אם הדף קיים בזיכרון. הפעם הדף באמת קיים ולכן תתבצע כתיבה אליו. הפונקציה הפנימית wp_reuse תגרום לדף לקבל הרשאות כתיבה לצורך כתיבת המידע בדף החדש. לאחר מכן הפונקציה תחזיר VM_FAULT_WRITE.

בהמשך הריצה של faultin_page ייבדק ערך ההחזרה של הפונקציות הפנימיות, ובאופן ספציפי, ייבדק התנאי הבא:

```
if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
      *flags &= ~FOLL_WRITE;
```

קטע הקוד למעלה בודק אם הפונקציות הפנימיות החזירו VM_FAULT_WRITE וגם לדף אין הרשאות כתיבה. flags מייצג את הצהרת הכוונות של המשתמש בשימוש בדף, ואם התנאי יתקיים, יהיה הדבר דומה לכך שהמשתמש לא ביקש לבצע כתיבה לדף. הפונקציה faultin_page תחזיר 0 בשנית.

כעת יש משמעות רבה ל-cond_resched. ב-Context Switch ייתכן שהמערכת תבצע הורדה מהזיכרון של הדף המטופל. כאשר יוחזר זמן הריצה שלנו, follow_page_mask יכריז שוב שאינו מוצא את הדף. faultin_page ייקרא שוב לפעולה, אבל הפעם לא יוצהר כי המשתמש רוצה לבצע כתיבה בדף. לכן תיקרא הפונקציה do_read_fault אשר תבצע בדיקה ב-cache האם קיים דף כזה. אם קיים, ייתכן באופן סביר שהדף שנמצא שם הוא הדף המקורי ובמקרה כזה, הוא זה שיוחזר מ-get_user_pages.

הקרנל לא מבצע וידוא במקרה כזה, **ולכן תתבצע בפועל כתיבה אל הדף המקורי ולא אל הדף המועתק!**

## פתרון

במקרה זה, המשאב המשותף הוא ה-cache בזיכרון אשר מביא דפים. הבאג נפתר בצורה פחות סטנדרטית על ידי יצירת דגל חדש הנוגע לתהליך ה-COW (אשר נותן אינדיקציה כי תהליך COW מתרחש) באופן ספציפי, ובכך מוריד את דו המשמעותיות סביב דגל הכתיבה שנבדק מספר פעמים ולבסוף מורד:

```
    if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags & VM_WRITE))
-       *flags &= ~FOLL_WRITE;
+         *flags |= FOLL_COW;
```

בנוסף, בפונקציה follow_pte_page אשר מהווה פונקציה פנימית ל-follow_page_mask מבוצעת בדיקה בהתאם לדגל החדש. בקצרה, הבדיקה מוסיפה על הלוגיקה הישנה גם וידוא האם קיים דגל COW, דגל FORCE והאם ה-Dirty Bit דלוק:

```
+/*
+ * FOLL_FORCE can write to even unwritable pte's, but only
+ * after we've gone through a COW cycle and they are dirty.
+ */
+static inline bool can_follow_write_pte(pte_t pte, unsigned int flags)
+{
+   return pte_write(pte) ||
+       ((flags & FOLL_FORCE) && (flags & FOLL_COW) && pte_dirty(pte));
+}
```

```
+

 static struct page *follow_page_pte(struct vm_area_struct *vma,
        unsigned long address, pmd_t *pmd, unsigned int flags)
 {
@@ -95,7 +105,7 @@ retry:
     }
     if ((flags & FOLL_NUMA) && pte_protnone(pte))
         goto no_page;
-    if ((flags & FOLL_WRITE) && !pte_write(pte)) {
+    if ((flags & FOLL_WRITE) && !can_follow_write_pte(pte, flags)) {
         pte_unmap_unlock(ptep, ptl);
         return NULL;
```

בכך נמנע כיבויו של דגל ה-WRITE, המערכת לא מנסה לחפש דפים עם הרשאות קריאה בלבד ב-cache ורק הדף אשר נוצר באמצעות COW יוחזר לבסוף מהפונקציה get_user_pages.

## סיכום

כאשר אנו שומעים על באג כל כך קריטי ששכב בקרנל במשך זמן כה רב, אנו עלולים לתפוס את עצמנו מופתעים. היה אולי מצופה מקהילה כל כך גדולה של תורמים להבחין בבעייתיות המנגנון, אך עם זאת, יש לזכור כי Race Condition הוא באג סטטיסטי אשר קשה לעתים לאתרו ולשחזרו. בסופו של דבר, מטעויות לומדים, וההשפעה החיובית של הפרצה לא מבוטלת בכלל. הפרצה הפכה את המודעות לסכנות בבאג מסוג זה לנושא חם בקהילות הפיתוח ואבטחת המידע ברחבי העולם, ובכך תרמה לפיתוח קוד מאובטח יותר.

תודה רבה שקראתם את המאמר ומקווים שנהניתם!

## על הכותבים

שמי עומר כספי אני מתכנת low level שמתעסק בזמנו הפנוי באבטחת מידע בתחום הפיתוח וגם בתחום בדיקות החדירות.

שמי 0xDEAD6057, ואני מפתח low level. בנוסף, אני נוהג להתעסק ב-Web, בסוגיות הנוגעות ל-Information Security ובנוסף ב-Internals של ה-Linux של kernel.

לכל שאלה / הערה / הארה או בקשה ניתן לפנות באימייל:

עומר: komerk0@gmail.com

## תודות

תודה לבן אגאי שעזר לנו בעריכה מקצועית והגהה של המאמר.

תודה לכל אלו אשר עזרו בבחירת הנושא ושלבי המחקר הראשוניים של הבאג.

תודה לצוות עורכי Digital Whisper אשר העניקו לנו את הבמה לשתף את המאמר.

תודה לכל המורים, המרצים והמדריכים המקצועיים אשר תמכו ועזרו לנו להמשיך להתפתח מבחינה מקצועית.

## ביבליוגרפיה

מידע על גילוי הפירצה:

http://www.v3.co.uk/v3-uk/news/2474845/linux-users-urged-to-protect-against-dirty-cow-security-flaw

מידע טכני על dirtyc0w:

https://dirtycow.ninja

https://github.com/dirtycow/dirtycow.github.io/wiki/VulnerabilityDetails

קוד הקרנל של לינוקס:

http://lxr.free-electrons.com

פתרון החולשה:

https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=19be0eaffa3ac7d8eb6784ad9bdbc7d67ed8e619

## נספחים

מידע על barriers ומימושם באמצעות Java:

http://blogs.sourceallies.com/2012/03/parallel-programming-with-barrier-synchronization/

מידע על barriers ו-Condition Variables:

http://ozark.hendrix.edu/~leonard/420-01/3-4-10.pdf

הסבר על Futex בלינוקס:

https://lwn.net/Articles/360699/

הוראות אטומיות:

http://faculty.ycp.edu/~dhovemey/spring2011/cs365/lecture/lecture20.html

מימושים שונים של פרצת dirtyc0w:

https://github.com/dirtycow/dirtycow.github.io/wiki/PoCs

הסבר על Page Cache:

http://www.tldp.org/LDP/lki/lki-4.html