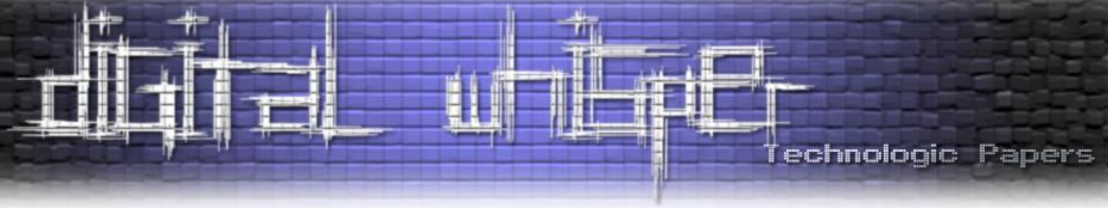


Grsecurity - Security Features for the Linux Kernel

מאת גילי ינקוביץ'

הקדמה

מאמר זה מובא כחלק מקהילת KernelTLV.com המקיימת מפגשים חודשיים בנושאי Linux Kernel ו-Kernel Security. האירוע הוא אירוע חינמי אשר מטרת המפגשים היא לקבץ ולהעשיר את קהילת מפתחי ה-Kernel בארץ. אתם מוזמנים לבקר באתר ולהתעדכן לגבי המפגשים הקרובים.

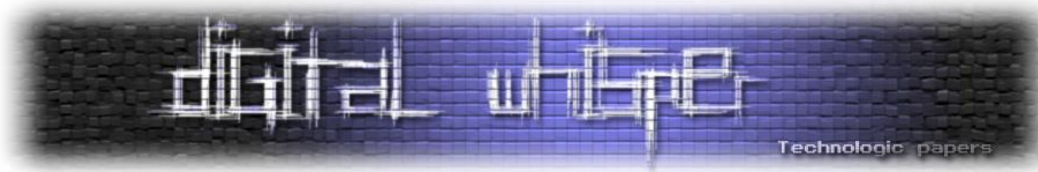
פוסט-הקדמה

Linux היא מערכת הפעלה פופולרית השולטת בתחום הרשתות ומפעילה מגוון מוצרי רשת כגון נתבים ומרכזיות. חשוב מאוד שמערכות אלו יהיו אמינות ובטוחות, אבל לא כולם שמים את נושא האבטחה בראש סדר העדיפויות. לינוס טורבלדס, למשל, התבטא בנושא פעמים רבות. הציטוט הבא מסכם את הגישה של לינוס לנושא האבטחה:

> "One reason I refuse to bother with the whole security circus is that I think it glorifies - and thus encourages - the wrong behavior.
> It makes "heroes" out of security people, as if the people who don't just fix normal bugs aren't as important."

ללינוס חשובים מאוד הביצועים של המערכת, לפעמים יותר מהאבטחה. לפעמים הדרישה לביצועים גוברת על הרצון לספק אבטחה ברמה הגבוהה ביותר האפשרית. למזלנו, קיימות קבוצות אשר עבורן, זהו הנושא אשר נמצא במרכז עיסוקן במערכת ההפעלה. הכירו: Grsecurity.

Grsecurity הינו patch ל-Linux Kernel, הנתמך בגרסאות 3.2.72 ו-3.14.54 בגרסאת ה-stable ו-4.3.5 בגרסאת ה-test. ה-patch היה חופשי לשימוש לכולם עד לאוגוסט 2015, אז יצאה הקבוצה בהצהרה שתפסיק לספק את אותו באופן חופשי מפאת פגיעה בזכויות יוצרים. כעת, ניתן להשתמש בגרסאת ה-test באופן חופשי או להפוך ללקוח רשמי שלהם ואז להשתמש בו. במאמר נתייחס לגרסאת ה-test החופשית, למטרות סקירה בלבד.



ה-patch תומך ברוב הארכיטקטורות ש-Linux תומך בהן. כמובן שלא כל הפיצ'רים נתמכים בכל ארכיטקטורה, אבל הרוב המשמעותי נתמך בארכיטקטורות המרכזיות: x86, x86_64, arm, powerpc, mips. במאמר אתמקד ב-x86_64.

הערה: אין כותב המאמר אחראי לכל נזק שיגרם כפועל יוצא של שימוש בתכניו, ואין הוא אחראי על מידת האבטחה המסופקת על-ידי התכנים המוסברים בו.

למרות האמור בהערה מעלה, Grsecurity הינו תוסף אבטחה מצוין ומומלץ לכל מערכת Linux אשר ניתן לשלב אותו בתוכה. התוסף מספק פיצ'רים שפועלים חלקם באופן פאסיבי וחלקם באופן אקטיבי, אשר משפרים את רמת האבטחה במערכת באופן גבוהה מאוד.

יש שישאלו אם התוסף כל-כך טוב, למה הוא לא ב-mainline? כנראה משיקולי ביצועים. כאמור, לינוס מעדיף אותם על פני אבטחה. כמו שנראה בהמשך, קיימים שינויים שנעשים בכוח, שלא מוגדרים תחת ifdef כלשהו ולכן ה-patch מאוד נחשב לפולשני, מה שמקשה על השילוב כחלק מה-mainline.

קונפיגורציה

את ה-patch כמובן, מחילים באופן הבא:

```
$ patch -p1 < grsecurity-3.1-4.3.5-201602032209.patch
```

וזהו. כעת ה-Kernel מעודכן בשינויים של Grsecurity, אבל חלק מהפיצ׳רים דורשים הפעלה אקטיבית ממערכת הקונפיגורציה של ה-Kernel.config. בסקירה הזו, אסקור אך ורק חלק מהפיצ'רים האבטחתיים של התוסף וכיצד הם פועלים. עבור רכיבים נוספים, ניתן לפנות באופן פרטי.

כמו כל פיצ'ר של Linux, נתחיל:

```
make menuconfig
      Security options --->
            Grsecurity --->
```

ניתן לבקש מ-Grsecurity להגדיר את הקופיגורציה שלו באופן אוטומטי, לפי מטרת השימוש ב-Kernel, עדיפות לביצועים/אבטחה. אבל אנחנו נרצה לבחון את הקונפיגורציות באופן פרטני. לכן, נבחר ב-Customize Configuration.

חלק מהתוסף הינו PaX ("שלום" בלטינית). PaX נחשב לרכיב נפרד ב-Grsecurity אבל בפועל, הוא מכיל את רוב הפיצ'רים האבטחתיים ברמת החומרה וניהול הזיכרון, הכוללים שיפורים לקריאות מערכת ASLR, Memory Sanitization, ועוד... במאמר זה לרוב אסקור מנגנונים של PaX, שהם אלו המתערבים במנגנוני Kernel עמוקים יותר, ולכן לדעתי יותר מעניינים.

במאמר זה אסקור את המנגנונים הבאים:

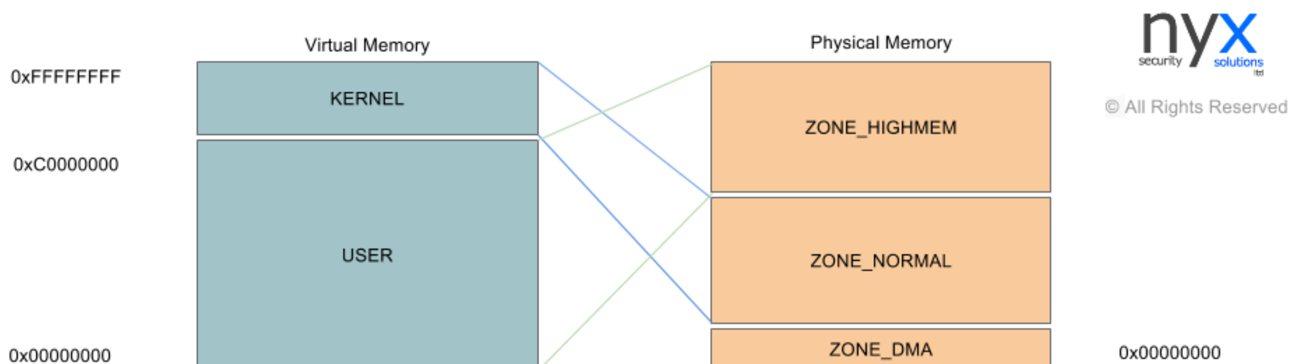
1. PAX_USERCOPY
2. PAX_MEMORY_SANITIZE
3. PAX_ASLR
   a. PAX_RANDSTACK

ניהול זיכרון ממש-על-קצה-המזג

ניהול זיכרון היא אחת הפעולות המורכבות יותר שמערכת הפעלה צריכה להתמודד איתן ובעצם, אחת מהפונקציונאליות היחידות שתוכנה צריכה לנהל כדי שתקרא "מערכת הפעלה" (ר' Microkernel). לכל מחשב, מאז ומעולם, היה קיים זיכרון נדיף (RAM - Random Access Memory) אשר הוא הזיכרון ה"אקטיבי" של המחשב. הוא יותר מהיר מ-HDD, אבל יותר איטי מאוגרי המעבד ו/או ה-cache-ים שלו. על הזיכרון נשמר מידע לטווח קצר. זיכרון זה הינו רכיב פיזי המחובר למחשב. הזיכרון הזה, כמו כל דבר פיזי, הינו מוגבל בנפחו. מה גם, שעם התפתחות מערכות ההפעלה, נוצר צורך לנהל מספר זכרונות נפרדים זה מזה. לכן הומצא הזיכרון הוירטואלי, המחלק את את הזיכרון לתהליכים בגדלי זיכרון של עד 4GB במכונות 32 ביט (לדוגמא).

אבן הבניים הבסיסית של זיכרון בכל מחשב הינם דפים. כל הזיכרון מחולק לדפים בגודל אחיד של 4KB או 32KB, בהתאם להגדרת מערכת ההפעלה. הדפים האלה הינם ישות לוגית הממופה לרכיב פיזי בעל כתובת ב-RAM הנקרא frame. באמצעות תמיכת ה-MMU מערכת ההפעלה מנהלת את מיפויי הזיכרון הוירטואלי (הדפים) מול המסגרות הפיזיות הממוקמות ב-RAM.

את הלוגיקה הזו מנהל ה-Kernel של מערכת ההפעלה. Linux מחלק את הזיכרון הפיזי של המכונה עליה הוא רץ לשלושה Zone-ים באופן הקלאסי. ב-Kernel-ים חדשים קיימים יותר Zone-ים, אבל נתעלם מהעובדה הזו כרגע, לצורך הפשטות:



1. ZONE_DMA - איזור השמור לקריאות / כתיבות מרכיבי DMA. איזור זה נגיש לבקרי DMA של רכיבים נוספים על ה-board (למשל, כרטיס רשת). בקרים אלה מבצעים offloading לכתיבה מרכיבים חיצוניים ל-RAM, כאשר הכתיבות נעשות על-פי כתובות פיזיות. בעבר, בקרי DMA היו נגישים אך ורק לכתובות של 16 ביט, לכן כתובות DMA הן בדרך-כלל נמוכות, על-מנת לאשר לבקרי DMA לפעול כראוי.
2. ZONE_NORMAL - האיזור אליו ממופה ה-Kernel, ה-Buddy System ומעליה ה-Slab Allocator (נדבר עליו בהמשך), kmalloc.
3. ZONE_HIGHMEM - איזור הזיכרון אליו ממופות כתובות זיכרון וירטואליות אשר אינן רציפות פיזית. כלומר: כל הקצאת זיכרון ב-Kernel על-ידי kmap ,vmalloc וזיכרון וירטואלי של אפליקציות usermode.

נתמקד ב-ZONE_NORMAL: איזור זה מממש את ה-Buddy System לזיכרון רציף פיזית. הוא מאגד דפים הרציפים פיזית בחזקות של 2. יש מספר סיבות שבגללן נרצה להקצות זיכרון רציף פיזית ולא רק וירטואלית. במקרה הזה, הסיבה העיקרית היא מהירות הגישה לזיכרון (ובשביל זה, בנו את ה-CMA). ה-Buddy System הוא מנגנון ניהול הזיכרון הבסיסי ביותר ב-Kernel. מעליו נבנה מנגנונים מתוחכמים יותר לניהול הזיכרון ברמה לוגית. כמובן זיכרון רציף פיזית הינו "יקר יותר" מזיכרון שרציף רק לוגית, לכן יש לודא שימוש נכון בו.

מעל ה-Buddy System, ה-Linux Kernel מממש מנגנון Slab/Slob/Slub: Slab Allocation. הקצאות הזיכרון ב-slab הינן על-פי cache-ים מוגדרים מראש בגודלם. בדרך-כלל, נרצה להגדיר cache נפרד לכל מבנה זיכרון מרכזי ב-Kernel ב-cache משל עצמו. כך למשל קיים task_struct cache עבור מבני זיכרון לתהליכים, cache-ים נפרדים ל-mm_struct, vma ועוד.

כמובן שניתן להקצות איזורי זיכרון ב-Kernel גם בגודל שרירותי. לכן קיימים slab-ים מיוחדים עבור הקצאות אלה. עבור הקצאות קטנות מ-128KB, ניתן להשתמש ב-kmalloc אשר מקצה זיכרון מעל ה-Buddy System. עבור הקצאות זיכרון מעל 128KB, עדיף השתמש ב-vmalloc, על-מנת לא לבזבז זיכרון רציף פיזית.

במאמר זה אתמקד בעיקר ב-Slub, שהוא ה-memory allocator החדש יותר ב-Kernel. לא אסביר עליו יותר מדי. יש עליו הסבר נהדר ב-lwn.

PAX_USERCOPY

כחלק מהיותו Kernel של מערכת הפעלה, Linux צריך לבצע אינטראקציה מול קלט מהמשתמש ופלט חזרה אל איזורי זיכרון שבשליטת המשתמש. פונקציונאליות זו נעשית על-ידי הפונקציות copy_from_user / copy_to_user.

```
static __always_inline __must_check
unsigned long __copy_from_user_nocheck(void *dst, const void __user *src,
unsigned long size)
{
	size_t sz = __compiletime_object_size(dst);
	unsigned ret = 0;

	if (size > INT_MAX)
		return size;

	check_object_size(dst, size, false);
```

[arch/x86/include/asm/uaccess_64.h]

המטרה הכללית במנגנון ההגנה הזה, הוא נסיון למנוע כתיבות של usermode data למבנים פנימיים של ה-Kernel מצד אחד ומניעת זליגת זיכרון של מבנים פנימיים של ה-Kernel חזרה ל-usermode. הגנה זו משולבת בקריאות copy_[from/to]_user, שהן הדרכים הסטנדרטיות למעבר מידע בין Kernel ו-user והפוך. כמובן שניתן לבצע פעולות אלו גם ללא עזרת הפונקציות הללו, אבל זה לא קורה כמעט אף פעם.

אתמקד כאן בניתוח copy_from_user, אשר (כמובן) הינה architecture dependant. במימוש עצמו של הפונקציה ימצא בפונקציה פנימית יותר: __copy_from_user_nocheck. נתמקד ב-check_object_size. ניתן לשים לב מתוך ה-patch שגם כאשר מחילים את Grsecurity על ה-Kernel, נעשים שינויים אשר אינם קונפיגורביליים. כלומר, רק בהחלת ה-patch, הועלתה רמת האבטחה של המערכת. כעת, נתבונן ב-check_object_size:

```
void __check_object_size(const void *ptr, unsigned long n, bool to_user, bool
const_size)
{
#ifdef CONFIG_PAX_USERCOPY
	const char *type;
#endif
...
#ifdef CONFIG_PAX_USERCOPY
	if (!n)
		return;
	type = check_heap_object(ptr, n);
	if (!type) {
		int ret = check_stack_object(ptr, n);
		if (ret == 1 || ret == 2)
			return;
		if (ret == 0) {
			if (check_kernel_text_object((unsigned long)ptr, (unsigned
long)ptr + n))
				type = "";
			else
				return;
```

```
        } else
                type = "";
    }
    pax_report_usercopy(ptr, n, to_user, type);
#endif
}
```

[fs/exec.c]

ננתח את הפונקציה הזו עד לסיום תקין. (כלומר, ללא קריאה ל-pax_report_usercopy, אשר מתריעה על אירוע חריג):

```
#ifdef CONFIG_PAX_USERCOPY
const char *check_heap_object(const void *ptr, unsigned long n)
{
      struct page *page;
      struct kmem_cache *s;
      unsigned long offset;

      if (ZERO_OR_NULL_PTR(ptr))
            return "<null>";

      if (!virt_addr_valid(ptr))
            return NULL;

      page = virt_to_head_page(ptr);

      if (!PageSlab(page))
            return NULL;
      s = page->slab_cache;
      if (!(s->flags & SLAB_USERCOPY))
            return s->name;

      offset = (ptr - page_address(page)) % s->size;
      if (offset <= s->object_size && n <= s->object_size - offset)
            return NULL;

      return s->name;
}
#endif
```

[mm/slub.c]

בתחילת הפונקציה, ניתן לראות sanity checks פשוטות. בדיקת NULL, בדיקה לקיום מיפוי הכתובת על-ידי *virt_addr_valid*. לאחר מכן, נבצע מספר בדיקות פשוטות מאוד: קבלת ה-struct page הראשון של הכתובת הנתונה. מכיוון שאנחנו רוצים להגן על מבני בקרה של ה-Kernel, נרצה לטפל בגישות לשרשרת הדפים שהוקצאה במקור. כמו שהוזכר קודם, מבני נתונים פנימיים של ה-Kernel, לרוב, יהיו שייכים ל-slab כלשהו. לכן נבדוק ראשית אם הדף הזה שייך ל-slab כלשהו. כמובן, שאם אינו שייך ל-slab כלשהו, לא נבדוק עוד מכיוון שזהו אומנם מצביע ב-Kernel אבל הוא אינו שייך לאף מבנה בקרה פנימי.

אז הבנו שהאובייקט הנבדק מוקצא על slab כלשהו. כפי שהוזכר מקודם, קיימים cache-ים מיוחדים עבור הקצאות זיכרון באמצעות kmalloc. ניתן למצוא את האיתחול של מבנים אלה ב-*mm/slab_common.c* והם מאותחלים עם הדגל SLAB_USERCOPY. כלומר, אלו slab cacheמיוחדים אשר ניתן לבצע העתקות ביניהם ובין זיכרון המוקצה לתהליך usermode.. הם מוגדרים על-ידי הדגל SLAB_USERCOPY על ה-slab, כפי שניתן לראות.

כעת, כל שנותר לבדוק זה שההעתקה לא גולשת בין אובייקטים. מכיוון שכל האובייקטים מוקצים על- slab cache ומכיוון שכל slab cache מוגדר מעל ה-Buddy System, כל אובייקט מתוך slab כלשהו יתחיל בכתובת שהינה Page Aligned. על-כן, נחשב את ה-offset להעתקה לתוך ה-page של אובייקט ה-slab ונבדוק שההעתקה מתחילה ונגמרת בתוך תחום אותו אוביקט.

אז צלחנו עד כאן והצלחנו לגרום לפונקציה להחזיר NULL. לכן או שהאובייקט נמצא על slab כלשהו אבל הוא על slab "בסדר" ואינו גולש, או שהוא לא על slab בכלל. לכן, נעבור לבדיקה על ה-Stack, כאשר הבדיקות הראשונות טריוויאליות:

1. נבדוק שאין integer overflow
2. נבדוק אם בכלל האובייקט על המחסנית (של ה-Kernel!!)
3. ושהאובייקט לא דורס את כל המחסנית.

```
#ifdef CONFIG_PAX_USERCOPY
/* 0: not at all, 1: fully, 2: fully inside frame, -1: partially
(implies an error) */
static noinline int check_stack_object(const void *obj, unsigned long
len)
{
	const void * const stack = task_stack_page(current);
	const void * const stackend = stack + THREAD_SIZE;

#if defined(CONFIG_FRAME_POINTER) && defined(CONFIG_X86)
	const void *frame = NULL;
	const void *oldframe;
#endif
	if (obj + len < obj)
		return -1;
	if (obj + len <= stack || stackend <= obj)
		return 0;
	if (obj < stack || stackend < obj + len)
		return -1;
```

[fs/exec.c]

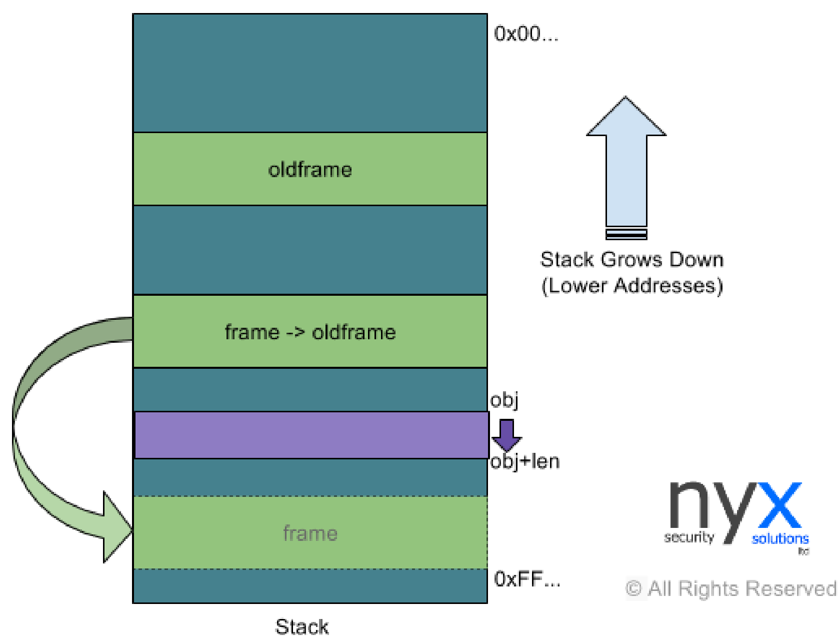
בגדול, כאן נגמרת הבדיקה. אבל בהמשך קיימת בדיקה נוספת מאוד מעניינת. אם ה-Kernel נבנה ל-x86 עם תמיכה ב-Frame Pointer (כלומר, ללא הדגל -fomit-frame-pointer של gcc(1)), נעשה שימוש בפונקציונאליות builtin של gcc:

```
#if defined(CONFIG_FRAME_POINTER) && defined(CONFIG_X86)
      oldframe = __builtin_frame_address(1);
      if (oldframe)
            frame = __builtin_frame_address(2);
...
      while (stack <= frame && frame < stackend) {
...
            if (obj + len <= frame)
                  return obj >= oldframe + 2 * sizeof(void *) ? 2 : -1;
            oldframe = frame;
            frame = *(const void * const *)frame;
      }
      return -1;
#else
      return 1;
#endif
}
```

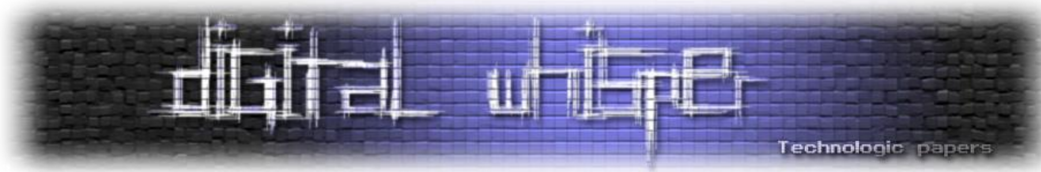
[fs/exec.c]

הבדיקה מנסה לבדוק האם האיזור להעתקה מוכל בתוך איזור המשתנים הלוקאליים של פונקציה. השימוש ב-__builtin_frame_address מספק את ה-stack frame האחרון על-פי ה-level המתאים, [לפי התיעוד של gcc](#).

הפונקציה תחפש מעלה במעלה המחסנית עד שתגיע לבסיסה, שם תיעצר. על-כן יש לשים לב שהעתקות הנעשות בפונקציות שהן מאוד פנימיות (דבר שבדרך-כלל לא קורה), עבור איזורי זיכרון שנמצאים בבסיס המחסנית, עלולות לקחת זמן במהלך ה-stack unwinding לחיפוש המצביע המתאים. בנוסף, נשים לב כי זו בדיקה מצוינת נגד stack based buffer overflows. עצם הבדיקה שהכתיבה נעשית רק בתוך איזור המשתנים הלוקאליים מודאת שלא נדרס אף מבנה בקרה של הפונקציה על המחסנית.



[שרטוט להמחשה של הבדיקה הנעשית על המחסנית]



כעת, כל שנותר לבדוק הוא שההעתקה לא מתבצעת על איזורי ה-text של ה-Kernel, שכן לא נרצה לכתוב עליו על-מנת למנוע Access Violation וכמובן שלא נרצה לאפשר למשתמש לקרוא את קוד ה-Kernel:

```
#ifdef CONFIG_PAX_USERCOPY
static inline bool check_kernel_text_object(unsigned long low, unsigned long high)
{
...
      unsigned long textlow = (unsigned long)_stext;
      unsigned long texthigh = (unsigned long)_etext;

      /* check against linear mapping as well */
      if (high > (unsigned long)__va(__pa(textlow)) &&
          low < (unsigned long)__va(__pa(texthigh)))
            return true;
      if (high <= textlow || low >= texthigh)
            return false;
      else
            return true;
}
#endif
```

[fs/exec.c]

באופן כללי, נשתמש ב-symbol-ים המסמנים את תחילת קוד ה Kernel *stext_* ו-*etext_*. נעשה וידוא מול היצוג הלינארי של הכתובות המוגדרות ב-symbol-ים האלה וגם עבור הכתובות עצמן. כך נוכל לודא האם האובייקט להעתקה נמצא חלק מה-Kernel במיפוי כלשהו למרחב הזיכרון. לבסוף, לאחר שוידאנו שהכל בסדר, נחזיר false, על-מנת לתת אינדיקציה ל-*check_object_size* שההעתקה בטוחה לביצוע.

באופן כללי, ניתן לראות שהמנגנון עצמו ממומש באופן מאוד פשוט וברור. נוסף על הכל, ניתן לראות שגם אם ה-Kernel נבנה ללא PAX_USERCOPY, קיימת הגנה חלקית על איזורי זיכרון בסיסיים. כמובן שמומלץ להפעיל את הגדרות PaX על רמת האבטחה הגבוהה ביותר בהתאם למתאר השילוב של ה-Kernel, אך קיימת הגנה גם ללא הגדרות ספציפיות.

PAX_MEMORY_SANITIZE

בכל הנוגע לאיזורי זיכרון רגישים - נרצה למנוע דליפות זיכרון למרות קיות המנגנון הקודם. על-כן, נשקיע את זמננו בלאתחל את איזורי הזיכרון בהם אנו משתמשים. כמו במנגנון הקודם, נגן על ה-Slab Allocator, שהוא הבסיס לכל הקצאות המבנים הפנימיים של ה-Kernel.

לפני הכל, אסביר מה הכוונה ב-Memory Sanitation: כאשר תוכנה משתמשת באיזורי זיכרון כלשהם, בין אם סטאטיים או דינאמיים, היא כותבת מידע פנימי של עצמה לצורך ניהול התוכנה על איזורי זיכרון אלה. מידע זה יכול להיות מספרים, מחרוזות, מצביעים וכו'. בעת שהתוכנה מסיימת להשתמש בזיכרון זה, היא

משחררת את הזיכרון הזה. כעת איזור הזיכרון חזר למאגר הפנוי לשימוש עבור מנגנון אחר בתוכנה. אם המנגנון החדש לא ממומש היטב, הוא עלול לעשות שימוש באיזורי זיכרון שהוא עצמו אינו אתחל לפני השימוש. בדרך-כלל gcc יתריע על כך (*-Wuninitialized, -Wmaybe-uninitialized*), אבל לא תמיד ובהחלט לא באופן אוטומטי. על-כן מנגנוני sanitation יגנו מפני מקרים כאלה על-ידי איפוס של ה-buffer בעת <u>השחרור</u> שלו, בדרך-כלל על-ידי אפסים. חשוב להדגיש כי מנגנון הגנה זה עלול לעלות בפגיעה קלה בביצועים, שכן בבסיסו, מתבצעת פעולת *memset* על כל ה--slab-ים אשר יסומנו לניקוי. על-כן, יש לקחת זאת בחשבון כאשר משלבים מנגנון זה.

כמובן, נתחיל בראשית: על-מנת לקבוע את רמת האבטחה הנדרשת, PaX נרשם ל- Kernel command line דרך קריאה ל-early_param, עם הפונקציה הבאה:

```
#ifdef CONFIG_PAX_MEMORY_SANITIZE
enum pax_sanitize_mode pax_sanitize_slab __read_only = PAX_SANITIZE_SLAB_FAST;
static int __init pax_sanitize_slab_setup(char *str)
{
	if (!str)
		return 0;
	if (!strcmp(str, "0") || !strcmp(str, "off")) {
		pax_sanitize_slab = PAX_SANITIZE_SLAB_OFF;
	} else if (!strcmp(str, "1") || !strcmp(str, "fast")) {
		pax_sanitize_slab = PAX_SANITIZE_SLAB_FAST;
	} else if (!strcmp(str, "full")) {
		pax_sanitize_slab = PAX_SANITIZE_SLAB_FULL;
	}
...
}
early_param("pax_sanitize_slab", pax_sanitize_slab_setup);
#endif
```

[mm/slab_common.c]

יחד עם פרסור ה-Kernel command line ניתן לבחור את רמת הסניטציה של ה-slab-ים, כאשר כברירת המחדל, נבחרת האפשרות המהירה, בניגוד למלאה, על-מנת לספק ביצועים טובים יותר במקרה הגנרי. בפועל, PAX_SANITIZE_SLAB_FAST לא באמת עושה שום דבר (נבין את הסיבה לכך עוד מעט). לכן, על-מנת לנקות את חוצצי הזיכרון של ה-slab, יש לשנות את ערך ברירת המחדל, או להוסיף ל- command line את הפקודה: pax_sanitize_slab=full.

```
struct kmem_cache *
kmem_cache_create(const char *name, size_t size, size_t align,
		  unsigned long flags, void (*ctor)(void *))
{
...
#ifdef CONFIG_PAX_MEMORY_SANITIZE
	if (pax_sanitize_slab == PAX_SANITIZE_SLAB_OFF || (flags &
SLAB_DESTROY_BY_RCU))
		flags |= SLAB_NO_SANITIZE;
	else if (pax_sanitize_slab == PAX_SANITIZE_SLAB_FULL)
		flags &= ~SLAB_NO_SANITIZE;
#endif
```

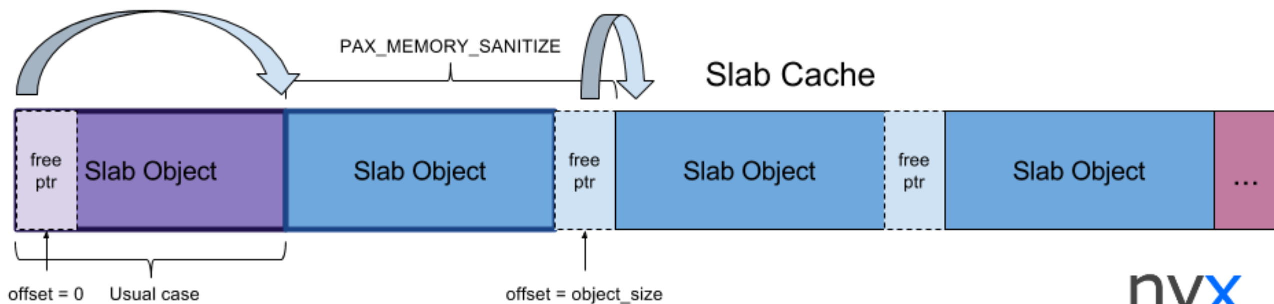
[mm/slab_common.c]

ביצירת ה-slab, נבדוק את ההגדרה כפי שהוגדרה בעליה מקודם. וכפי שניתן לראות, אין זכר ל-FAST. מכיוון שההקשחה היא בינארית - האם למחוק את המידע בעת שחרור הזיכרון או לא, נבחר ב-*PAX_SANITIZE_SLAB_FULL*.

נקודה חשובה מאוד שחייבים לשים לב אליה היא הנקודה הבאה:

```
static int calculate_sizes(struct kmem_cache *s, int forced_order)
{
...
	if (((flags & (SLAB_DESTROY_BY_RCU | SLAB_POISON)) ||
#ifdef CONFIG_PAX_MEMORY_SANITIZE
	    (!(flags & SLAB_NO_SANITIZE)) ||
#endif
		s->ctor)) {
		/*
		 * Relocate free pointer after the object if it is not
		 * permitted to overwrite the first word of the object on
		 * kmem_cache_free.
		 *
		 * This is the case if we do RCU, have a constructor or
		 * destructor or are poisoning the objects.
		 */
		s->offset = size;
		size += sizeof(void *);
	}
…
static inline void set_freepointer(struct kmem_cache *s, void *object, void *fp)
{
	*(void **)(object + s->offset) = fp;
}
```

[mm/sub.c]



© All Rights Reserved

מכיוון ש-Slub הוא מנגנון "יעיל" יותר אשר משתמש בחוצץ עצמו עבור ה-metadata שלו (באופן דומה לפעולה של malloc על חוצצים משוחררים), מנגנון ה-Slub מנהל freelist בעזרת metadata על החוצץ עצמו. אבל מכיוון שנרצה לאפס את תוכן החוצץ, לא ניתן להשתמש בחוצץ עצמו למיקום ה-freepointer לחוצץ הפנוי הבא.

לכן, במקרה הזה נאלץ להגדיר את ה-offset המצביע על מיקום ה-freepointer לסוף האובייקט ולהצהיר על כך שהאובייקט גדול במצביע אחד נוסף.

```
static __always_inline void slab_free(struct kmem_cache *s,
                        struct page *page, void *x, unsigned long addr)
{
...
#ifdef CONFIG_PAX_MEMORY_SANITIZE
       if (!(s->flags & SLAB_NO_SANITIZE)) {
              memset(x, PAX_MEMORY_SANITIZE_VALUE, s->object_size);
              if (s->ctor)
                     s->ctor(x);
       }
#endif
```

[mm/sub.c]

ולבסוף, בעת שחרור של אוביקט slab כלשהו נבצע memset על מצביע האובייקט המשוחרר. כמובן שנקרא ל-Constructor במקרה וקיים, על-מנת לספק אובייקט חדש מאותחל. לכן יש לשים לב, כי במקרה והוגדר slab עם Constructor, הוא יקרא בעת השחרור של האובייקט לקראת הקצאתו הבאה, בניגוד לדעה הרווחת ב-OOP כי ה-Constructor נקרא לאחר ההקצאה. במקרה זה, הקריאה נובעת משיקולי יעילות וזמינות האובייקטים בזמן ההקצאה.

```
static bool free_pages_prepare(struct page *page, unsigned int order)
{
...
#ifdef CONFIG_PAX_MEMORY_SANITIZE
       unsigned long index = 1UL << order;
#endif
...
#ifdef CONFIG_PAX_MEMORY_SANITIZE
       for (; index; --index)
              sanitize_highpage(page + index - 1);
#endif
...

static int prep_new_page(struct page *page, unsigned int order, gfp_t gfp_flags,
int alloc_flags)
{
...
#ifndef CONFIG_PAX_MEMORY_SANITIZE
       if (gfp_flags & __GFP_ZERO)
              for (i = 0; i < (1 << order); i++)
                     clear_highpage(page + i);
#endif
```

[mm/page_alloc.c]

בנוסף לאתחול הדפים בהחזרה ל-slab-ים המקוריים, PaX אוכף אתחול גם בהקצאות ושחרורים של דפים באופן גנרי, מעל ה-Buddy System. באופן כללי, עבור כל הקצאה/שחרור של דף, PaX ימפה את הדף ל-High Memory, ינקה אותו ואז ימחק את המיפוי, לטובת המיפוי האמיתי, כפי שניתן לראות למעלה.

בהקצאת דפים, נעבור דף-אחר-דף, נמפה אותו ל-highmem אם נדרש, נאפס אותו ונשחרר את המיפוי.

```
static inline void clear_highpage(struct page *page)
{
	void *kaddr = kmap_atomic(page);
	clear_page(kaddr);
	kunmap_atomic(kaddr);
}

static inline void sanitize_highpage(struct page *page)
{
	void *kaddr;
	unsigned long flags;

	local_irq_save(flags);
	kaddr = kmap_atomic(page);
	clear_page(kaddr);
	kunmap_atomic(kaddr);
	local_irq_restore(flags);
}
```

[include/linux/highmem.h]

PAX_ASLR && PAX_RANDMAP

אחרון חביב ברשימת המנגנונים לסקירה הזו, הוא ה-ASLR של PaX. ראשי התיבות הוא Address Space Layout Randomization. זהו מנגנון ידוע במערכות הפעלה, אשר מגריל את כתובות הקצאות הזיכרון של התוכנית.

למה שנרצה לעשות דבר כזה בכלל? בעיקר על-מנת להקשות על תוקפים לנצל חולשות בצורה דטרמיניסטית. כאשר קיים תוקף המנסה לנצל חולשה, יתכן שיצטרך להסתמך על כתובות בזיכרון, למשל: כתובות של פונקציות על-מנת לבצע לוגיקה מורכבת יותר כחלק מהניצול. דוגמא נוספת היא למשל בנית ROP Chain. על-מנת לממש את ה-ROP יש לדעת היכן ה-gadgets נמצאים בזיכרון. מכאן, כאשר נגריל את מרחב הזיכרון, נקשה מאוד על תוקף לנצל את החולשות הללו.

ל-Linux יש כבר מנגנון ASLR מובנה, אבל האנטרופיה שלו לא מדהימה ובנוסף הוא אינו משנה את איזורי הזיכרון של התוכנית בצורה משמעותית. על-מנת להפעיל ולכבות את ה-ASLR במערכת, ניתן לשנות את ערכו של המשתנה הגלובאלי randomize_va_space בKernel. המשתנה הזה נגיש גם בתור root. על-מנת להפעיל את ה-ASLR, יש לבצע את הפקודה הבאה (בתור root):

```
$ echo 2 > /proc/sys/kernel/randomize_va_space
```

המשתנה הזה הוא int אשר לרוב נבדק בתור ערך בוליאני מלבד במקרה יחיד של אקראיות המחסנית. לכן, כדאי שיכיל את הערך 2, לעומת 1. מעתה, כל תהליך לחדש שיווצר, ישתמש ב-ASLR של Linux (או של PaX, כתלות בכך שהחלתם את ה-patch על המערכת שלכם).

כמו לכל רכיב ב-PaX, יש ל-ASLR מספר מצבי עבודה. PaX יכול להיות מוחל על קבצי ELF במספר דרכים:

1. החלה כוללת (ברירת המחדל).
2. כחלק מה-ELF Header.
3. כחלק מה-Extended Attributes של מערכת הקבצים.

בכל מקרה, על מנת שה-ASLR יפעל בכל תצורה שהיא, על המשתנה randomize_va_space להיות true, כלומר שונה מ-0. לכן, ודאו שזה אכן המצב לפני ההמשך, אחרת שום דבר אחר לא יעבוד. ניתן להשתמש בשתי הדרכים האחרות באמצעות הכלים chpax(1) ו-paxctl(1), אך על-מנת להשתמש בהם יש לבנות את ה-Kernel עם ההגדרות PAX_PT_PAX_FLAGS/PAX_XATTR_PAX_FLAGS. אמליץ על ברירת המחדל, מכיוון שבמקרה זה האקראיות תחול על כל הרצה שהיא.

כעת, נעבור לאתחול תהליך. עבור כל ELF ש-Linux עושה עבורו execve, ה-Kernel קורא ל-load_elf_binary. הפונקציה אחראית לקרוא את קובץ ה-ELF, למפות אותו לזיכרון, למפות את ה-Interpreter (אם יש לו) ואז להריץ את ה-Interpreter.

אנצל את ההזדמנות כדי להזכיר דבר חשוב מאוד: אין משמעות כמעט בכלל לשימוש ב-ASLR עם בינאריים שעברו Linkage סטאטי. כלומר: בעת הבניה של ה-elf, יש להשתמש בדגלים `-fPIC` עבור בניה של קבצי ELF להרצה ובדגלים `-fPIE` ו-`-pie` עבור בניה של Shared Objects. יש לקרוא את man gcc(1) להסבר לא הרבה יותר מפורט.

בעצם, כאשר נאמר שנרצה לגרום למרחב הזיכרון להיות אקראי יותר, לאילו רכיבים במרחב הזיכרון נרצה לשנות את המיקום? בעצם, לכולם. אבל בכל-זאת נמנה אותם:

1. איזור ה-brk (עבור malloc, למשל).
2. ה-ELF עצמו.
3. הקצאות mmap.
   a. הקצאות עבור data.
   b. מיפויי זיכרון של ספריות דינאמיות.
4. המחסנית.

נתחיל מאיזור ה-brk, מכיוון שמקרה זה הוא הפשוט ביותר. מלבד ההגרלה שנעשית באופן טבעי, PaX מבצע הגרלה משלו.

איזור זה נחשב ל-heap של התהליך וכמו-זה, malloc עצמו משתמש בו. על-כן, נגריל גם את תחילתו.

```
static int load_elf_binary(struct linux_binprm *bprm)
{
...
#ifdef CONFIG_PAX_RANDMMAP
	if (current->mm->pax_flags & MF_PAX_RANDMMAP) {
		unsigned long start, size, flags;
		vm_flags_t vm_flags;

		start = ELF_PAGEALIGN(elf_brk);
		size = PAGE_SIZE + ((pax_get_random_long() & ((1UL << 22) - 1UL))
<< 4);
		flags = MAP_FIXED | MAP_PRIVATE;
		vm_flags = VM_DONTEXPAND | VM_DONTDUMP;
		down_write(¤t->mm->mmap_sem);
		start = get_unmapped_area(NULL, start, PAGE_ALIGN(size), 0, flags);
		...
		if (retval == 0)
			retval = set_brk(start + size, start + size + PAGE_SIZE);
...
	}
#endif
```

[fs/binfmt_elf.c]

נמשיך בכך שנגריל את כתובת ההקצאה עבור מיפוי קובץ ה-ELF אותו נריץ כעת. נשים לב ל"טריק" ש-PaX עושה כדי לודא ש-Linux יציית לכתובת שהוגרלה (elf_flags |= MAP_FIXED). אנקדוטה: טריק זה גם נעשה בכל dynamic loader כדי לודא שה-offset-ים אשר הוקצו ישארו נכונים במיפוים חוזרים.

```
static int load_elf_binary(struct linux_binprm *bprm)
{
...
			load_bias = ELF_ET_DYN_BASE - vaddr;
			if (current->flags & PF_RANDOMIZE)
				load_bias += arch_mmap_rnd();
			load_bias = ELF_PAGESTART(load_bias);
#ifdef CONFIG_PAX_RANDMMAP
			/* PaX: randomize base address at the default exe base if
requested */
			if ((current->mm->pax_flags & MF_PAX_RANDMMAP) &&
elf_interpreter) {
#ifdef CONFIG_SPARC64
				load_bias = (pax_get_random_long() & ((1UL <<
PAX_DELTA_MMAP_LEN) - 1)) << (PAGE_SHIFT+1);
#else
				load_bias = (pax_get_random_long() & ((1UL <<
PAX_DELTA_MMAP_LEN) - 1)) << PAGE_SHIFT;
#endif
				load_bias = ELF_PAGESTART(PAX_ELF_ET_DYN_BASE - vaddr
+ load_bias);
				elf_flags |= MAP_FIXED;
			}
#endif
...
		error = elf_map(bprm->file, load_bias + vaddr, elf_ppnt,
				elf_prot, elf_flags, total_size);
...
	}
#endif
```

[fs/binfmt_elf.c]

בסופו של דבר, הKernel יטען את הקובץ הניתן לו (bprm->file) לכתובת ה-FIXED שנתנה.

כעת, נבחן את קטע הקוד הבא, אשר ישמש בתור אבני הבניין להגרלת איזורי הזיכרון של ה-stack ושל הקצאות ה-mmap:

```
static int load_elf_binary(struct linux_binprm *bprm)
{
...
#ifdef CONFIG_PAX_ASLR
	if (current->mm->pax_flags & MF_PAX_RANDMMAP) {
		current->mm->delta_mmap = (pax_get_random_long() &
				((1UL << PAX_DELTA_MMAP_LEN)-1)) << PAGE_SHIFT;
		current->mm->delta_stack = (pax_get_random_long() &
				((1UL << PAX_DELTA_STACK_LEN)-1)) <<
PAGE_SHIFT;
	}
#endif
```

[fs/binfmt_elf.c]

שני המשתנים *delta_mmap* ו-*delta_stack* הולכים לשמש כנקודות מפתח בהמשך. לכן, אם נאתחל את RANDMAP לפי אחת האפשרויות המוזכרות מעלה, נוכל לאתחל את המשתנים הללו. שימו לב כי PaX מספק אפילו את פונקציות ה-Random בעצמו. אבל זו בעצם קריאה לפונקצית random גנרית של Linux: prandom_u32. יש לשים לב שזהו אינו urandom.

אחרי שאתחלנו משתנים שיגדירו את האקראיות של המחסנית ושל mmap, באמת צריך לגרום לאקראיות הזו לקרות. לכן, נתחיל מהמחסנית.

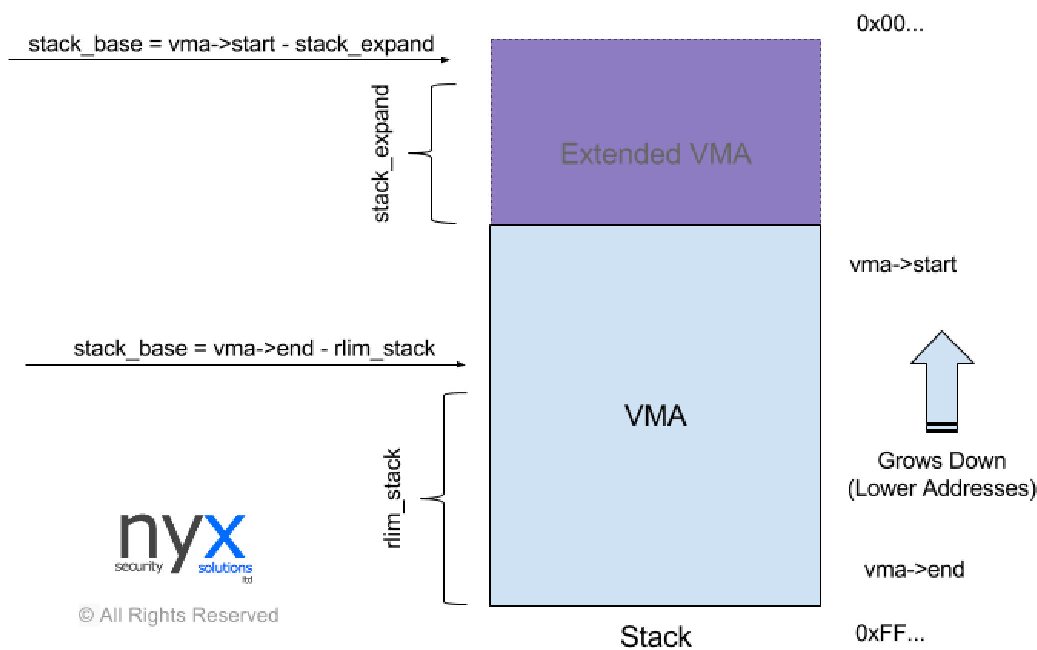
```
#define STACK_TOP ((current->mm->pax_flags &
MF_PAX_SEGMEXEC)?SEGMEXEC_TASK_SIZE:TASK_SIZE)
```

[arch/x86/include/asm/processor.h]

```
static unsigned long randomize_stack_top(unsigned long stack_top)
{
#ifdef CONFIG_PAX_RANDUSTACK
	if (current->mm->pax_flags & MF_PAX_RANDMMAP)
		return stack_top - current->mm->delta_stack;
#endif

/* … In load_elf_binary … */
retval = setup_arg_pages(bprm, randomize_stack_top(STACK_TOP),
				executable_stack);
```

[fs/binfmt_elf.c]

כמובן שלפונקציה יש המשך, אבל כאן זה ממש ברור ש-PaX מכריח את הKernel להתייחס אליו קודם, לפני כל דגל אחר של מערכת ההפעלה. וכעת, ננסה למפות את הכתובת שהגרלנו. נקווה שנצליח, הרי פשוט הגרלנו מספר.




```
int setup_arg_pages(struct linux_binprm *bprm,
                    unsigned long stack_top,
                    int executable_stack)
{
      struct vm_area_struct *vma = bprm->vma;
...
      stack_top = arch_align_stack(stack_top);
      stack_top = PAGE_ALIGN(stack_top);
      bprm->p -= stack_shift;
...
      stack_expand = 131072UL; /* randomly 32*4k (or 2*64k) pages */
      stack_size = vma->vm_end - vma->vm_start;

      rlim_stack = rlimit(RLIMIT_STACK) & PAGE_MASK;
      if (stack_size + stack_expand > rlim_stack)
            stack_base = vma->vm_end - rlim_stack;
      else
            stack_base = vma->vm_start - stack_expand;

      current->mm->start_stack = bprm->p;
      ret = expand_stack(vma, stack_base);
```

[fs/exec.c]

דבר ראשון, ה-vma המוזכר כאן הוא vma שהוקצא כאשר קראו ל-execve. הוא מצביע לרשימה מקושרת (ועץ) של ה-vma-ים במרחב הזיכרון של התהליך הנוכחי. ה-vma-ים ממוינים לפי הכתובות שלהם, אבל זה לא משנה כל-כך כרגע, מכיוון שזהו ה-vma היחיד שמוקצא כרגע, והוא מוקצא בקצה הזיכרון (נבין בקרוב איך זה קרה).

מה שה-Kernel מנסה לעשות כאן זה להגדיל את המחסנית באופן די שרירותי. בנוסף, ניתן לראות כאן את השימוש ב-rlimit לגודל המחסנית. הקוד קצת מבלבל, אבל בפועל הוא נכון (ר' שרטוט להמחשה) ולבסוף מבקש להגדיל את ה-VMA לכתובת שחושבה. נשים לב גם ל-p. אמנם זהו שם שלא נותן אינדיקציה מאוד טובה למשמעותו, אבל תפקידו הוא להצביע על הכתובת הגבוהה ביותר במרחב הזיכרון, ומכיוון ש-Linux מקצא את המחסנית בכתובת הגבוהה ביותר, נעדכן את p לכתובת המחסנית.

כעת נבדוק את איזורי הזיכרון המוקצים דינאמית על-ידי מערכת ההפעלה. כידוע, הקצאה זו נעשית על-ידי mmap(2). הקצאה זו מבקשת איזור זיכרון חדש בגודל N דפים ממערכת ההפעלה וזו ממפה N דפים חדשים, מאותחלים ל-0, למרחב הזיכרון של התוכנית (שימו לב: בניגוד ל-malloc, לא ניתן להקצות זיכרון בגודל שאינו גודל של דף שלם).

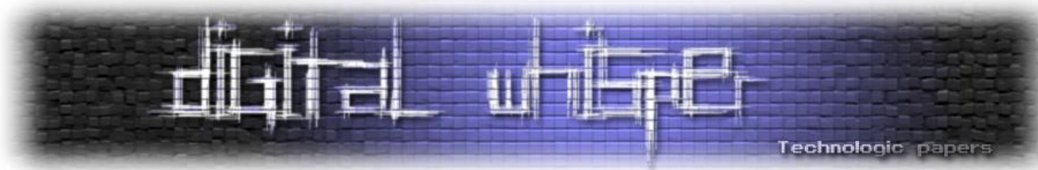
גם את הקצאות אלה נרצה לקבל בצורה אקראית, לטובת טעינה של ספריות דינאמיות הנטענות באמצעות mmap על-ידי ה-dynamic loader בעליית כל תהליך. לכן נמצא את המקום בו משתמשים ב-delta_mmap.

והרי התוצאה הלא מפתיעה (אך הלא-מאוד טריוואלית):

```
void arch_pick_mmap_layout(struct mm_struct *mm)
{
        unsigned long random_factor = 0UL;

#ifdef CONFIG_PAX_RANDMMAP
        if (!(mm->pax_flags & MF_PAX_RANDMMAP))
#endif
        if (current->flags & PF_RANDOMIZE)
                random_factor = arch_mmap_rnd();

        mm->mmap_legacy_base = mmap_legacy_base(mm, random_factor);

        if (mmap_is_legacy()) {
                mm->mmap_base = mm->mmap_legacy_base;
                mm->get_unmapped_area = arch_get_unmapped_area;
        } else {
                mm->mmap_base = mmap_base(mm, random_factor);
                mm->get_unmapped_area = arch_get_unmapped_area_topdown;
        }

#ifdef CONFIG_PAX_RANDMMAP
        if (mm->pax_flags & MF_PAX_RANDMMAP) {
                mm->mmap_legacy_base += mm->delta_mmap;
                mm->mmap_base -= mm->delta_mmap + mm->delta_stack;
        }
#endif

}
```

[arch/x86/mm/mmap.c]

mmap_base הינו האיבר אשר ישמש את ה-memory manager להגבלת איזורי הזיכרון מהם ניתן להקצות למשתמש. באופן זה, מגביל PaX את טווח הכתובות האפשרי לבחירה וממומש ASLR ברמת



איזורי הזיכרון הניתנים להקצאה. בנוסף, ניתן לראות בבירור את ההגרלה הנעשית על-ידי Linux באופן סטנדרטי על-ידי arch_mmap_rand והשמה ל-mmap_base.

```
unsigned long
arch_get_unmapped_area_topdown(struct file *filp, const unsigned long addr0,
                              const unsigned long len, const unsigned long pgoff,
                              const unsigned long flags)
{
...
        info.flags = VM_UNMAPPED_AREA_TOPDOWN;
        info.length = len;
        info.low_limit = PAGE_SIZE;
        info.high_limit = mm->mmap_base;
        info.align_mask = 0;
        info.align_offset = pgoff << PAGE_SHIFT;
        if (filp) {
                info.align_mask = get_align_mask();
                info.align_offset += get_align_bits();
        }
        info.threadstack_offset = offset;
        addr = vm_unmapped_area(&info);
```

[arch/x86/kernel/sys_x86_64.c]

אין באמת מה לראות כאן. כל העבודה באמת נמצאת ב-vm_unmapped_area שמוביל בסוף ל-unmapped_area_topdown או unmapped_area כתלות בבקשת ה-info. לא אצטט את הקוד כאן מפאת הכמות המאסיבית שלו, אך אסביר את הלך הרוח בו, עבור unmapped_area_topdown, כאשר השניה היא זהה לחלוטין, מלבד החיפוש שהוא bottom-up. יש לשים לב אך ורק ל-high_limit אשר הינו האיבר שמושפע מתוספת ה-ASLR של PaX כפונקציה של mmap_base.

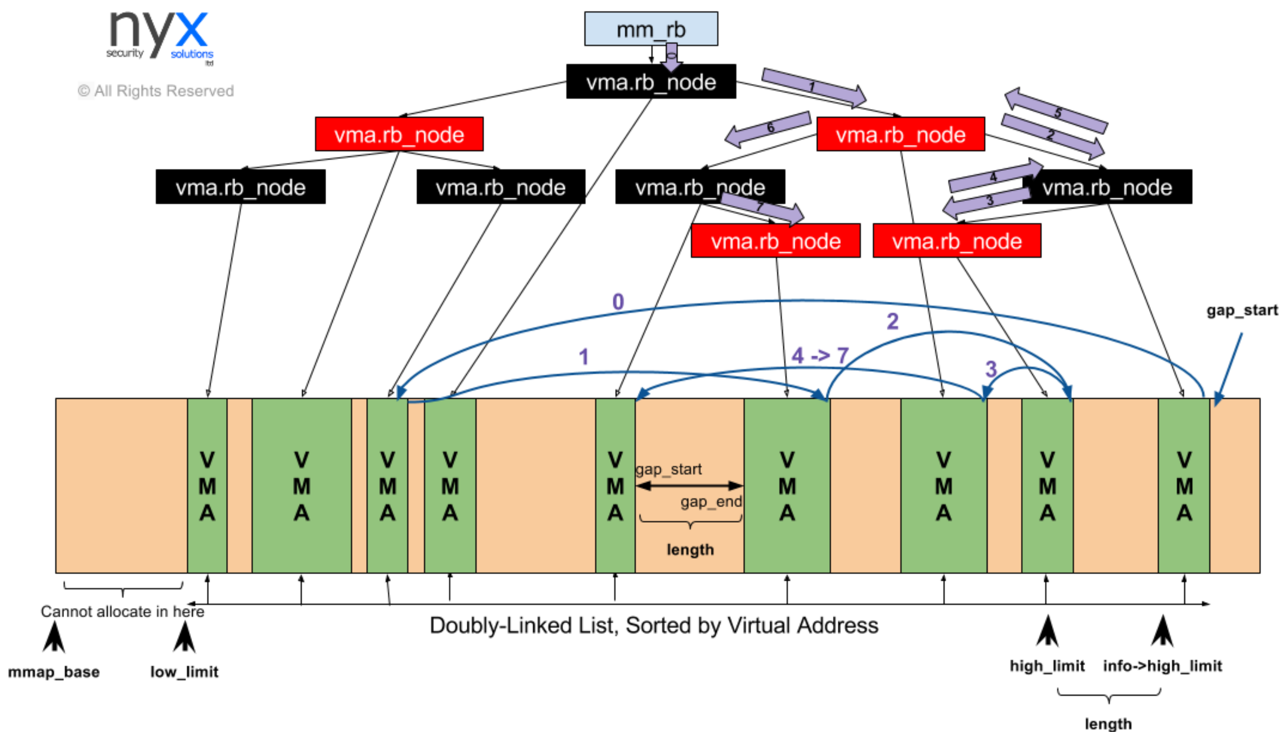
באופן כללי, איזורי זיכרון ב-Linux מיוצגים על-ידי מבנה נתונים הנקרא (vritual memory area) vma. כל vma מכיל שני תת-מבני נתונים, המקשרים בין כל ה-vma-ים של אותו תהליך: רשימה מקושרת דו כיוונית ועץ אדום-שחור. שניהם ממוינים לפי הכתובת הוירטואלית, כפי שניתן לראות בשרטוט מטה. הרשימה המקושרת מצביעה בכל vma על ה-vma הבא והקודם במרחב הזיכרון (מבחינת כתובות וירטואליות) והעץ הינו עץ אדום-שחור מאוזן גם הוא על-פי הכתובות. ברשימה בדרך-כלל משתמשים על מנת לבצע איטרציה על כלל ה-vma-ים או על-מנת לגשת לאיזורים שכנים ובעוד שמשתמשים במבנה העץ על-מנת לבצע חיפוש אחר איזורי זיכרון.

```
struct vm_area_struct {
      /* The first cache line has the info for VMA tree walking. */

      unsigned long vm_start;          /* Our start address within vm_mm. */
      unsigned long vm_end;            /* The first byte after our end address
                                   within vm_mm. */

      /* linked list of VM areas per task, sorted by address */
      struct vm_area_struct *vm_next, *vm_prev;

      struct rb_node vm_rb;
```

```
        /*
         * Largest free memory gap in bytes to the left of this VMA.
         * Either between this VMA and vma->vm_prev, or between one of the
         * VMAs below us in the VMA rbtree and its ->vm_prev. This helps
         * get_unmapped_area find a free area of the right size.
         */
        unsigned long rb_subtree_gap;
...
} __randomize_layout;
```

[include/linux/mm_types.h]

אוסיף ואומר כי rb_subtree_gap מחזיק את ה-gap המקסימלי ב-subtree אשר ה-vma הנוכחי הוא הראש שלו. איבר זה במבנה הנתונים מסייע בחיפוש אחר gap מתאים לאכלס בו את ההקצאה החדשה, כל-עוד היא בין ה-high_limit ל-low_limit המוגדרים.



נתחיל את החיפוש ראשית מסוף הזיכרון בתור מקרה קצה, למקרה וקיים זיכרון פנוי בסוף הזיכרון. נבחן האם קיים מרווח מתאים בין קצה איזור הזיכרון האחרון (gap_start) ל-high_limit שהוגדר. באם לא, נחל את החיפוש על העץ, כאשר המועמד הראשון הוא ראש העץ.

בכל איטרציה, ננסה לבחור את ה-vma אשר הינו הבן הימני, מכיוון שכתובתו גבוהה יותר. לאחר מכן, לכשהגענו לבן הימני ביותר, נבדוק את מרחקו משכנו (vm_prev). אם המרחק מספיק, נחזיק את ה-gap

הזה בתור המקום להקצאה החדשה. אחרת, נתחיל לבחור בבנים השמאליים של הענף הימני ביותר אליו הגענו. בכל בן שמאלי שוב ננסה להגיע לעלה הימני ביותר וכך הלאה.

במקרה והגענו ל-gap כלשהו אשר מכיל מספיק מקום אך נמצא מעל ה-high_limit, נאלץ לחזור אחורה עד שנגיע לצומת בה נבחר כעת בבן השמאלי, במקום הימני שנבחר. בקיצור: חיפוש על עץ בינארי ממוין.

כמובן שיש לשים לב, שה-ASLR כאן מגיע לידי ביטוי בנקודה אחת ויחידה: אותו delta_mmap אשר הגרלנו, כעת מגביל את ההקצאה מאיזור הכתובות הגבוהות. כלומר: ה-ASLR של PaX מגביל באופן אקראי את הכתובת הגבוהה ובכך מנסה לדחוס מעלה (הרי ההקצאות הן topdown) את רוב ההקצאות החל מכתובת גבוהה אקראית.

סיכום

ראינו ממש מעט מהנגיעות הקטנות של PaX ב-Memory Manager. זהו אך ורק קצה הקרחון לאבטחה המסופק ע"י התוסף. יש לקהילת האבטחה ב-Linux הרבה מאוד מה ללמוד מהתוסף הנ"ל והוא פותח הרבה אפשרויות לאבטחה (וסוגר פרצות פוטנציאליות). חשוב ללמוד ממנו וכמובן לשלב אותו במקומות הנכונים, בקונפיגורציה המתאימה.

מנגד, אי אפשר לומר שהאבטחה של Grsecurity ו-PaX מושלמת. קיימים מנגנונים נוספים הניתנים למימוש מעל Linux ולא רק ברמת ה-Kernel. מקרה זה הוא מאוד קיצוני בו יש להחליף את ה-Kernel, כאשר זה לא תמיד מצב אפשרי.

על הכותב

גילי ינקוביץ' (giliy@nyxsecuritysolutions.com) הוא מנכ"ל וחוקר האבטחה הראשי בחברת Nyx Software Security Solutions, בעל 7 שנים ניסיון באבטחת מידע, רשתות תקשורת ו- Embedded Security and Development. החברה מספקת שירותי אבטחה ומייצרת פתרונות אבטחה ייעודיים בתחום ה-Anti Exploitation.