digital whisper
Technologic Papers

על ארכיטקטורת Yan85 ומתקפות Side-channel

מאת יובל גל

הקדמה

במאמר זה אציג את הפתרון שלי לתרגיל האחרון בסדרת אתגרי ה-Reversing של האתר pwn.college. לאורך השלבים ניצבת לפנינו תוכנה שמממשת אימולטור לארכיטקטורה מומצאת בשם Yan85. כדי להשיג את ה-flag נאלץ להבין את הארכיטקטורה.

האתר pwn.college משמש כפלטפורמה לימודית למי שמוענין ללמוד ולתרגל קונספטים יסודיים באבטחת מידע. הוא נועד לקדם את מי שנמצא ברמה של "חגורה לבנה", בנושא אבטחת מידע, ל-"חגורה כחולה", הבנה מספקת לפתרון אתגרי CTF ברמה בינונית. האתר נוסד ע"י יאן שושיטאישבילי וקונור נלסון מאוניברסטית אריזונה והוא משמש כפלטפורמת התרגול של קורס אבטחת המידע באוניברסיטה.

השלבים הקודמים אילצו אותי לכתוב Disassembler ל-bytecode של הארכיטקטורה ואף להתאים אותו כ-Assembler לצורכי כתיבת Shellcode. אך השלב שעניין אותי במיוחד היה האתגר האחרון שדרש ניצול של חולשת Side-Channel בתוכנה שתזליג מידע נחוץ כדי לכתוב ולהריץ את ה-Shellcode שלנו.

כדי להגיע לאתגר האחרון ולפתרון שלו, תחילה נבין איך הארכיטקטורה הפשוטה הזאת עובדת.

את המידע הנ"ל הסקתי באמצעות ניתוח סטטי ודינמי של האימולטור לאורך התרגילים. תהליך ההסקה הזה פחות מעניין ולכן לא נצלול אליו, אבל כן ניגע במימוש האימולטור בהמשך לצורך פתרון האתגר. את החלק הזה מומלץ לקרוא ברפרוף, אין סיבה להתעמק פה באמת - זו גם ככה לא ארכיטקטורה אמיתית P:

yancode

ה-bytecode שמייצג הוראה בארכיטקטורה, כל הוראה היא 3 בתים:

- opcode - ערך מספרי שמייצג את הפעולה
- ארגומנט ראשון - יכול להיות Immediate value (ערך מספרי), ערך מספרי המייצג register, syscall או condition flag
- ארגומנט שני - כמו הארגומנט ראשון, סוג הערך על פיו יפורש הארגומנט תלוי הפעולה המתבצעת

סדרם של שלושת הבתים אינו קבוע ועלול להשתנות בין התרגילים.

אוגרים

בדומה לכל ארכיטקטורה אחרת, משמשים כאבני היסוד להעברת מידע ועיבודו.

שם	תיאור
a	כללי, משמש כפרמטר 1 ל-syscall-ים
b	כללי, משמש כפרמטר 2 ל-syscall-ים
c	כללי, משמש כפרמטר 3 ל-syscall-ים
d	כללי
s	מצביע לראש המחסנית (מקביל ל-rsp)
i	מצביע לכתובת הפעולה המתבצעת בקוד (מקביל ל-rip)
f	משמש כ-register הדגלים, לצורכי השוואות בין register-ים
NONE	register ריק, חסר משמעות

פעולות

הפקודות שאותן המעבד שמדומה ע"י האימולטור יכול לבצע:

שם	ארגומנט 1	ארגומנט 2	תיאור
IMM	register	value	<value> = <register> IMM, שמה ערך מספרי ל-register.
ADD	register	register	<register> += <register> ADD, מחברת שני register-ים, שמה את התוצאה בראשון.
STK	register	register	<register> <register> STK, פעולות מחסנית על register-ים, • ארגומנט ראשון - עושה לערכו push אל המחסנית • ארגומנט שני - עושה pop מהמחסנית אליו מאפשרת לבצע גם push וגם pop בפעולה אחת. מקבלת NONE כ-register אפשרי במידה והפעולה לא רצויה.
STM	register	register	<register> = *<register> STM, שמה את הערך של ה-register השני בכתובת אליה מצביע ה-register הראשון.
LDM	register	register	*<register> = <register> LDM, שמה את הערך אליו מצביעה הכתובת ב-register השני ב-register הראשון.
CMP	register	register	<register> <register> CMP, משווה בין שני register-ים ומחזיר את התוצאה ב-register הדגלים בשם f.
JMP	register	condition flag	בהתאם ל-condition flag שניתן, משווה ל-register הדגלים בשם f ולפי התוצאה קופץ לכתובת בערך ה-register שניתן.
SYS	syscall	register	<register> -> <syscall> SYS, מבצע syscall (קריאת מערכת) ומחזיר את ערך החזרה שלה ב-register. מקבלת NONE כ-register אפשרי במידה וערך החזרה לא רצוי.

קריאות מערכת

אלה הן קריאות המערכת שמתאפשרות לביצוע באמצעות פעולת SYS. מאחורי הקלעים נקראות קריאות המערכת ה-Linux-יות המקבילות:

שם	תיאור	פרמטרים
open	פותחת קובץ	• a = שם הקובץ לפתיחה • b = מצב פתיחת הקובץ
read_code	קוראת מקובץ לכתובת של קוד	• a = מזהה קובץ פתוח לקריאה (fd) • b = כתובת הקוד אליה יכתב הנקרא • c = מספר הבתים לקרוא
read_memory	קוראת מקובץ לכתובת של זיכרון	• a = מזהה קובץ פתוח לקריאה (fd) • b = כתובת הזיכרון אליה יכתב הנקרא • c = מספר הבתים לקרוא
write	כותבת מכתובת של זיכרון לקובץ	• a = מזהה קובץ פתוח לכתיבה (fd) • b = כתובת הזיכרון ממנה לקרוא את שיכתב • c = מספר הבתים לכתוב
sleep	ישנה ל-N שניות	• a = מספר השניות לישון
exit	יוצאת עם סטטוס חזרה	• a = סטטוס החזרה

babyrev level22

אחרי שהכרנו את Yan85 ובתקווה אנחנו מספיק yanheads כדי להמשיך, הגיע הזמן לחשוף את האתגר:

```
hacker@babyrev_level22:~$ ls -la /flag
-r-------- 1 root root 56 Oct  8 23:19 /flag
hacker@babyrev_level22:~$ ls -la /challenge/babyrev_level22.1
-rwsr-xr-x 1 root root 18600 Sep 17  2021 /challenge/babyrev_level22.1
```

אין לנו הרשאות מספקות כדי לקרוא את הקובץ שמכיל את ה-flag, אבל התוכנה של האתגר בעלת הרשאת setuid - משמע היא תרוץ בתור המשתמש root שיכול לקרוא את הקובץ הרצוי:

```
hacker@babyrev_level22:~$ /challenge/babyrev_level22.1
[+] Welcome to /challenge/babyrev_level22.1!
[+] This challenge is an custom emulator. It emulates a completely custom
[+] architecture that we call "Yan85"! You'll have to understand the
[+] emulator to understand the architecture, and you'll have to understand
[+] the architecture to understand the code being emulated, and you will
[+] have to understand that code to get the flag. Good luck!
[+]
[+] This level is a full Yan85 emulator. You'll have to reason about yancode,
[+] and the implications of how the emulator interprets it!
[?] This challenge is special! It randomizes the Yan85 VM based on
```

```
[?] the value of the flag. This means that there is no way for you
[?] to know the opcode and argument encodings...
[?]
[?] Keep in mind that the encoding that you observe in practice mode
[?] is going to be different than the actual encoding, because the
[?] practice mode flag is different. How will you adapt?
[?]
[?] Is there maybe a clever side channel you can utilize?
[?] ... Done! VM is randomized!
[!] This time, YOU'RE in control! Please input your yancode:
```

נפרוט לנקודות דברים מעניינים שאנחנו מסיקים מפלט התוכנה:

- ה-opcode-ים והערך המספרי של ארגומנטים (syscall-ים, register-ים וכו') מוגרלים על סמך ה-flag
- פתרון האתגר טמון בניצול Side-channel מתוחכם
- התוכנה מקבלת yancode דרך stdin אותו היא תריץ
  - נרצה להעביר shellcode שיפתח את flag וידפיס לנו אותו

הגרלה מבוססת flag

בהקדמה הזכרתי שכתבתי בעבר Assembler לארכיטקטורה, אז לכאורה כל מה שאנחנו צריכים הוא ליצר באמצעותו את ה-Shellcode, לא?

כזכור, ל-opcode-ים ולערכים שמועברים כארגומנט לפעולות יש ערך מספרי, ובמקרה הזה הוא מוגרל, לכן נצטרך להסיק אותו עבור כל אבן בניין שנדרשת על מנת לבצע את ה-Shellcode. הגיע הזמן לשלוף את ה-IDA ולהציץ על מימוש מנגנון ההגרלה. הפונקציה rerandomize נקראת מתוך main:

```
call    _puts
lea     rdi, aIsThereMaybeAC ; "[?] Is there maybe a clever side channe"...
call    _puts
mov     eax, 0
call    rerandomize
lea     rdi, aDoneVmIsRandom ; "[?] ... Done! VM is randomized!"
call    _puts
lea     rdi, aThisTimeYouReI ; "[!] This time, YOU'RE in control! Pleas"...
mov     eax, 0
call    _printf
lea     rax, [rbp+buf]
mov     edx, 300h       ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read
lea     rax, [rbp+buf]
mov     rdi, rax
call    interpreter_loop
main endp
```

בתוכה, עבור כל קבוצה שמגרילים, נקראת shuffle_values:

```
call    shuffle_values
movzx   eax, byte ptr cs:VALUES
mov     cs:SPEC_REG_A, al
movzx   eax, byte ptr cs:VALUES+1
mov     cs:SPEC_REG_B, al
movzx   eax, byte ptr cs:VALUES+2
mov     cs:SPEC_REG_C, al
movzx   eax, byte ptr cs:VALUES+3
mov     cs:SPEC_REG_D, al
movzx   eax, byte ptr cs:VALUES+4
mov     cs:SPEC_REG_S, al
movzx   eax, byte ptr cs:VALUES+5
mov     cs:SPEC_REG_I, al
movzx   eax, byte ptr cs:VALUES+6
mov     cs:SPEC_REG_F, al
mov     eax, 0
call    shuffle_values
movzx   eax, byte ptr cs:VALUES
mov     cs:INST_IMM, al
movzx   eax, byte ptr cs:VALUES+1
mov     cs:INST_STK, al
```

ולאחר מכן מתבצעת השמה של בתים מתוך VALUES לגלובליים של הקבוצה:

```
                public VALUES
VALUES          dq 8040201008040201h
```

shuffle_values אחראית לקרוא בלולאה לפונקצית rand מתוך libc ולהשתמש בתוצאה כדי לסדר מחדש את הבתים ב-VALUES:

```
loc_14FC:
cmp     [rbp+var_C], 0FFFEh
jle     short loc_148D

loc_148D:
call    _rand
cdq
shr     edx, 1Dh
add     eax, edx
and     eax, 7
sub     eax, edx
mov     [rbp+var_8], eax
call    _rand
cdq
shr     edx, 1Dh
add     eax, edx
and     eax, 7
sub     eax, edx
mov     [rbp+var_4], eax
mov     eax, [rbp+var_8]
cdqe
lea     rdx, VALUES
movzx   eax, byte ptr [rax+rdx]
mov     [rbp+var_D], al
mov     eax, [rbp+var_4]
cdqe
lea     rdx, VALUES
movzx   edx, byte ptr [rax+rdx]
mov     eax, [rbp+var_8]
cdqe
lea     rcx, VALUES
mov     [rax+rcx], dl
mov     eax, [rbp+var_4]
cdqe
lea     rcx, VALUES
movzx   edx, [rbp+var_D]
mov     [rax+rcx], dl
add     [rbp+var_C], 1

nop
nop
leave
retn
; } // starts at 1478
shuffle_values endp
```

כל זה נחמד ויפה, אבל איפה הקריאה ל-srand שאחראית לספק seed שבהסתמך עליו יגונרצו האקראיים?

נוכל למצוא אותה באמצעות xref (מציאת שימוש) לרשומה שלה בטבלת ה-import-ים:

0000000000005...	read@@GLIBC_2.2.5	.dynsym
0000000000005...	__libc_start_main@@GLIBC_2.2.5	.dynsym
0000000000005...	srand@@GLIBC_2.2.5	.dynsym
0000000000005...	setvbuf@@GLIBC_2.2.5	.dynsym
0000000000005...	open@@GLIBC_2.2.5	.dynsym

[srand בטבלת ה-import-ים]

לאחר ביצוע מספר xref-ים (שיקלפו את העטיפות של ה-import), נמצא את עצמנו ב-xref המעניין:

xrefs to _srand

Directio	Type	Address	Text
D...	p	flag_seed+17A	call _srand

Line 1 of 1

OK Cancel Search Help

הפונקציה flag_seed מתחילה בפתיחה של קובץ flag וקריאת 128 תווים ממנו:

```
mov     [rbp+var_18], 0
mov     esi, 0          ; oflag
lea     rdi, file       ; "/flag"
mov     eax, 0
call    _open
mov     [rbp+fd], eax
cmp     [rbp+fd], 0
jns     short loc_13B1
```

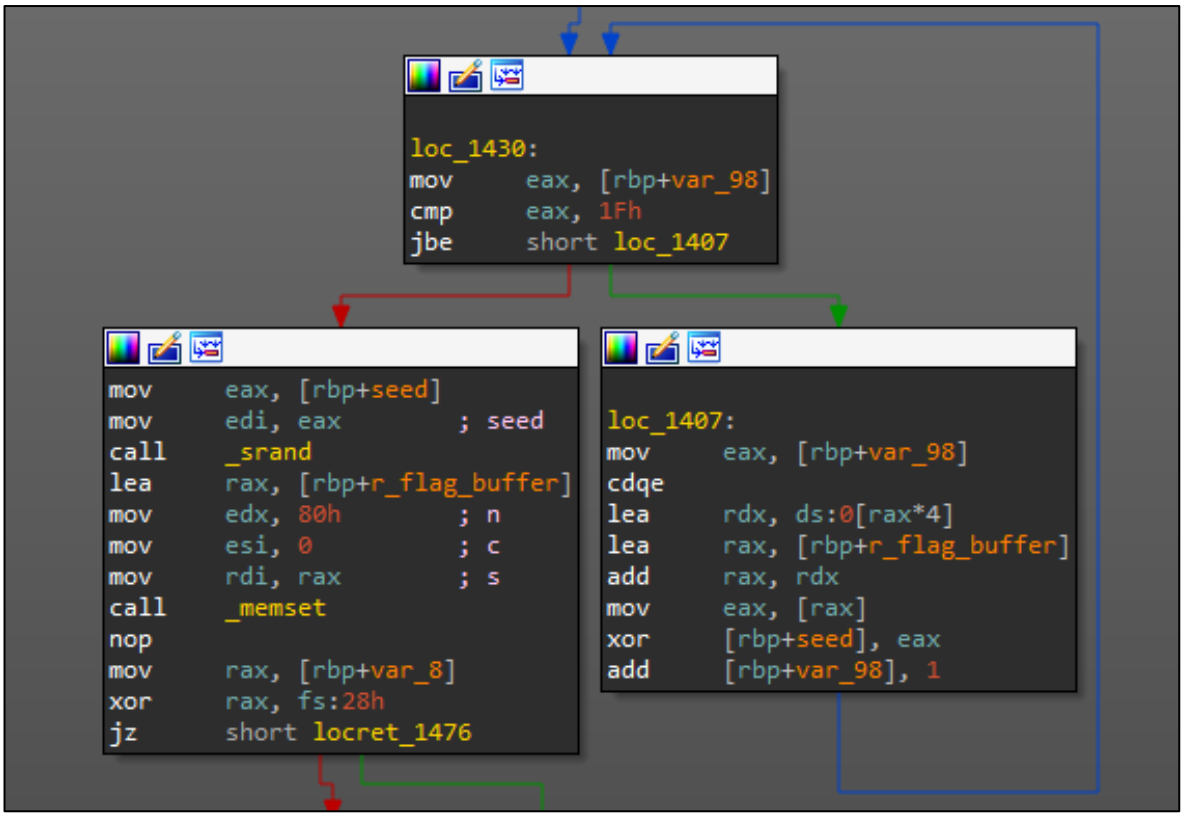
```
lea     rcx, __PRETTY_FUNCTION___3544 ; "flag_seed"
mov     edx, 11h        ; line
lea     rsi, aStdin     ; "<stdin>"
lea     rdi, assertion  ; "fd >= 0"
call    ___assert_fail
```

```
loc_13B1:
lea     rcx, [rbp+r_flag_buffer]
mov     eax, [rbp+fd]
mov     edx, 128        ; nbytes
mov     rsi, rcx        ; buf
mov     edi, eax        ; fd
call    _read
test    rax, rax
jg      short loc_13F1
```

```
lea     rcx, __PRETTY_FUNCTION___3544 ; "flag_seed"
mov     edx, 12h        ; line
lea     rsi, aStdin     ; "<stdin>"
lea     rdi, aReadFdFlag1280 ; "read(fd, flag, 128) > 0"
call    ___assert_fail
```

```
loc_13F1:
mov     [rbp+seed], 0
mov     [rbp+var_98], 0
jmp     short loc_1430
```

לאחר מכאן מתבצעת פעולה בלולאה על התווים, נוצר seed והוא מועבר ל-srand, לבסוף אף מאפסים את התווים שנקראו בזיכרון:



נראה לג'יט.

לנחש את קובץ הדגל יהיה שקול לפתירת האתגר - לכן זוהי לא הדרך הקלה לנצח. לא נוכל גם לדבג בדרכים המקובלות את דרכנו לנצחון - התוכנה נשענת על קובץ שדורש הרשאות קריאה של root בשביל לגנרץ את הערכים האקראיים וכשנדבג אותה היא לא תרוץ כ-setuid.

משחק עם האימולטור

בואו נתחיל בלשחק קצת עם האימולטור ונראה מה ההתנהגות שלו בהינתן קלט.

קלט לא תקין:

```
hacker@babyrev_level22:~$ /challenge/babyrev_level22.1
...
[!] This time, YOU'RE in control! Please input your yancode: ABCDEF
[+] Starting interpreter loop! Good luck!
Machine CRASHED due to: unknown register
```

מניב קריסה עם הודעת שגיאה של register לא מוכר.

קלט תקין:

```
hacker@babyrev_level22:~$ echo -e "\x80\x20\x40"|/challenge/babyrev_level22.1
...
[!] This time, YOU'RE in control! Please input your yancode: [+] Starting interpreter
loop! Good luck!
```

נראה שקלט תקין לכאורה (שלושה בתים אקראים מתוך VALUES) מניב לולאה (התוכנה רצה ללא הפסקה וללא חיוויים).

שימו לב ששם התוכנה הוא babyrev_level22.**1**, זוהי הגרסה הקשה של התרגיל - ללא סימבולים וללא חיוויים, לפניה יש גרסת "גלגלי עזר" בשם babyrev_level22.**0** - מכילה סימבולים בבינארי ומדפיסה דיבוג של ה-yancode שרץ מעליה. כמובן ששתיהן משמשות תרגילים שונים ולכל אחת מהן יש flag שונה.

נשתמש בגרסת "גלגלי עזר" כדי לחקור את מצב הלולאה:

```
hacker@babyrev_level22:~$ echo -e "\x80\x20\x40"|/challenge/babyrev_level22.0
...
[!] This time, YOU'RE in control! Please input your yancode: [+] Starting interpreter
loop! Good luck!
[V] a:0 b:0 c:0 d:0 s:0 i:0x1 f:0
[I] op:0x20 arg1:0x40 arg2:0x80
[s] STM *d = i
[V] a:0 b:0 c:0 d:0 s:0 i:0x2 f:0
[I] op:0 arg1:0 arg2:0xa
[V] a:0 b:0 c:0 d:0 s:0 i:0x3 f:0
[I] op:0 arg1:0 arg2:0
...
[V] a:0 b:0 c:0 d:0 s:0 i:0xff f:0
[I] op:0 arg1:0 arg2:0
[V] a:0 b:0 c:0 d:0 s:0 i:0 f:0
[I] op:0 arg1:0 arg2:0
[V] a:0 b:0 c:0 d:0 s:0 i:0x1 f:0
[I] op:0x20 arg1:0x40 arg2:0x80
[s] STM *d = i
...
```

- מתבצעת פעולה תקינה (במקרה הזה - STM)
- ואז האימולטור ממשיך לפרש פקודות ריקות, אפילו שהקלט שנתנו הוא רק ארבעה תווים (\x80\x20\x40\n).
- הוא מגיע לבסוף ל-Instruction ה-0xff ואז מריץ את i=0 ולבסוף חוזר להתחלה i=1 וחוזר חלילה.

ב-main קוראים 0x300 תווים של yancode:

```
mov     edx, 300h       ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read
lea     rax, [rbp-410h]
mov     rdi, rax
call    interpreter_loop
main endp
```

גודל ה-yancode מסתדר עם התנהגות האימולטור שמריץ 0x100 פעולות (0x300/3 - כל פעולה היא 3 בתים). אנחנו מסיקים מכך כי האימולטור לא קורס כאשר הוא מרציץ פעולה שה-opcode שלה אינו תקין.

ה-Shellcode המושלם

לפני שאנחנו מתעסקים בהזלגה, מה בכלל אנחנו צריכים להזליג? מה הן אבני היסוד שדרושות לנו כדי להדפיס את ה-flag למסך?

הנה פיסת פסודו-קוד של Yan85 שתשמש אותנו:

```
# ‘a’ register should point to flag filename in memory, will use STM
*a = '/flag'
IMM b = 0 # read mode
SYS open -> a # open(file=’/flag’, mode=0), return fd in ‘a’ register

IMM b = 0x60 # read flag to arbitrary address in memory
IMM c = 55 # flag is 55 chars
SYS read -> NONE # read(fd=a, buf=0x60, count=55)

IMM a = 1 # stdout
SYS write -> NONE # write(fd=1, buf=0x60, count=55)
```

ולפיכך האקראיים שנצטרך לגלות הם:

פעולות	אוגרים	קריאות מערכת
IMM	a	open
STM	b	read
SYS	c	write

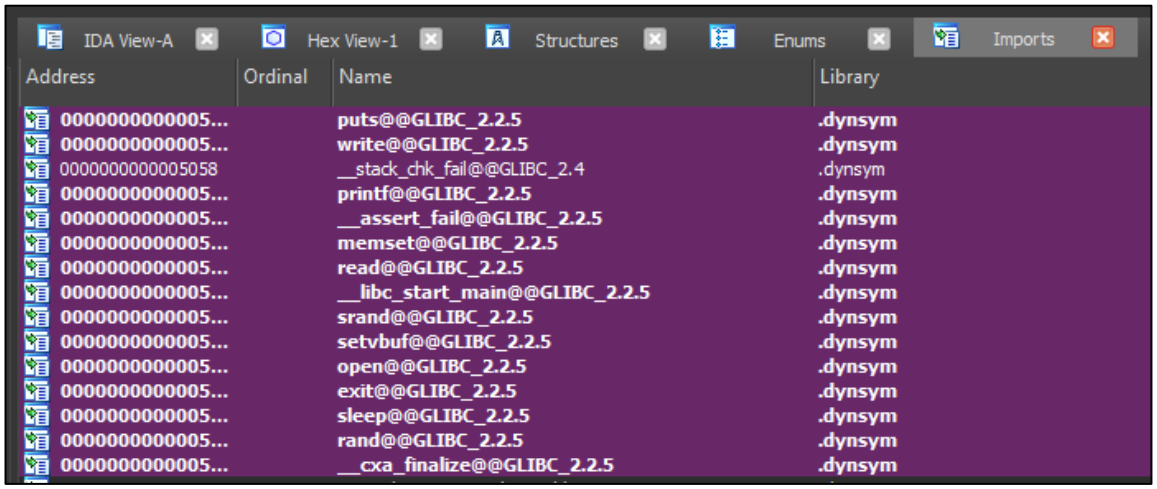
ערוצים-צדדיים באימולטור

אז מה אפשר לעשות בשביל להזליג אותם?

ניתוח פלט

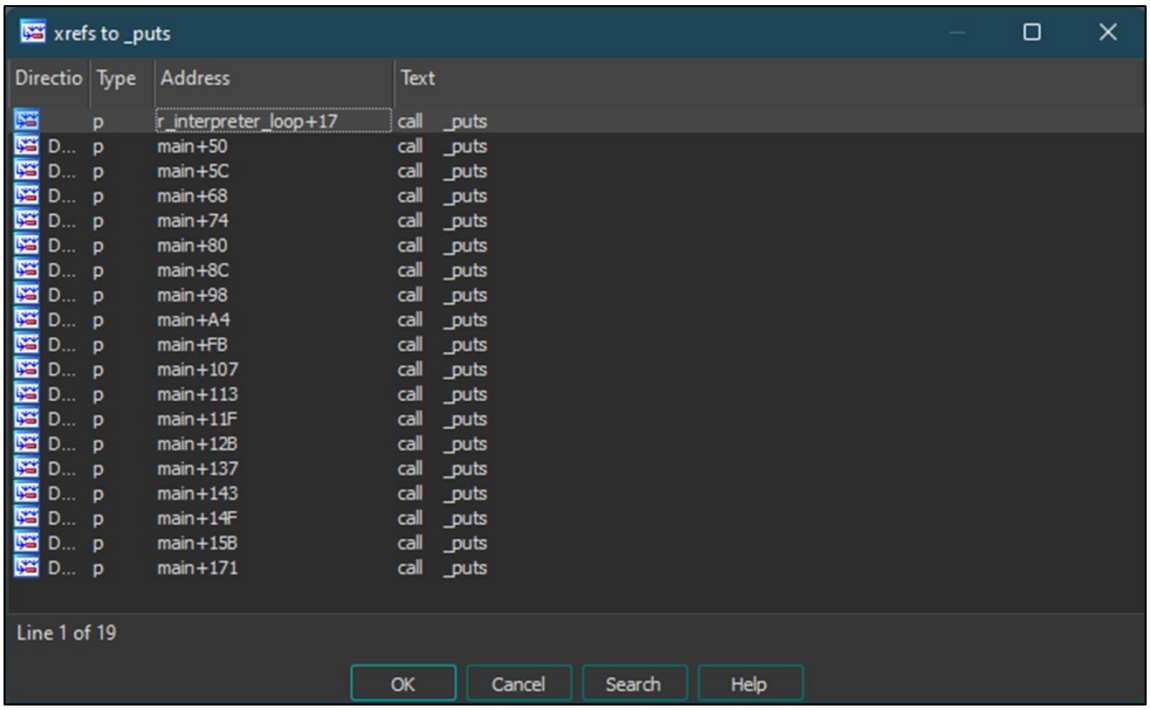
אמנם ראינו כי התוכנה (ברמה הקשה) לא מוסרת הרבה מידע, מלבד שגיאות, אולי פספסנו פלט כלשהו שיוכל לעזור לנו להבין מה הערכים המוגרלים? נרברס אותה ונחפש xref-ים לפונקציות שמדפיסות למסך:

כלל ה-import-ים של התוכנה:



Address	Ordinal	Name	Library
0000000000005...		puts@@GLIBC_2.2.5	.dynsym
0000000000005...		write@@GLIBC_2.2.5	.dynsym
0000000000005058		__stack_chk_fail@@GLIBC_2.4	.dynsym
0000000000005...		printf@@GLIBC_2.2.5	.dynsym
0000000000005...		__assert_fail@@GLIBC_2.2.5	.dynsym
0000000000005...		memset@@GLIBC_2.2.5	.dynsym
0000000000005...		read@@GLIBC_2.2.5	.dynsym
0000000000005...		__libc_start_main@@GLIBC_2.2.5	.dynsym
0000000000005...		srand@@GLIBC_2.2.5	.dynsym
0000000000005...		setvbuf@@GLIBC_2.2.5	.dynsym
0000000000005...		open@@GLIBC_2.2.5	.dynsym
0000000000005...		exit@@GLIBC_2.2.5	.dynsym
0000000000005...		sleep@@GLIBC_2.2.5	.dynsym
0000000000005...		rand@@GLIBC_2.2.5	.dynsym
0000000000005...		__cxa_finalize@@GLIBC_2.2.5	.dynsym

קריאות לפונקציה puts:



Directio	Type	Address	Text
	p	r_interpreter_loop+17	call _puts
D...	p	main+50	call _puts
D...	p	main+5C	call _puts
D...	p	main+68	call _puts
D...	p	main+74	call _puts
D...	p	main+80	call _puts
D...	p	main+8C	call _puts
D...	p	main+98	call _puts
D...	p	main+A4	call _puts
D...	p	main+FB	call _puts
D...	p	main+107	call _puts
D...	p	main+113	call _puts
D...	p	main+11F	call _puts
D...	p	main+12B	call _puts
D...	p	main+137	call _puts
D...	p	main+143	call _puts
D...	p	main+14F	call _puts
D...	p	main+15B	call _puts
D...	p	main+171	call _puts

Line 1 of 19

OK Cancel Search Help

כלל הקריאות האלו לא מעניינות והן מכילות את ההודעות שבתחילת התוכנית.

קריאות לפונקציה write - הקריאה היחידה מתבצעת כאשר ה-yancode מבצע קריאת מערכת מסוג write (באמצעות פעולת SYS):

xrefs to _write

Directio	Type	Address	Text
D...	p	r_sys+243	call _write

Line 1 of 1

OK Cancel Search Help

קריאות לפונקציה printf:

xrefs to _printf

Directio	Type	Address	Text
D...	p	r_crash+23	call _printf
D...	p	main+44	call _printf
D...	p	main+182	call _printf

Line 1 of 3

OK Cancel Search Help

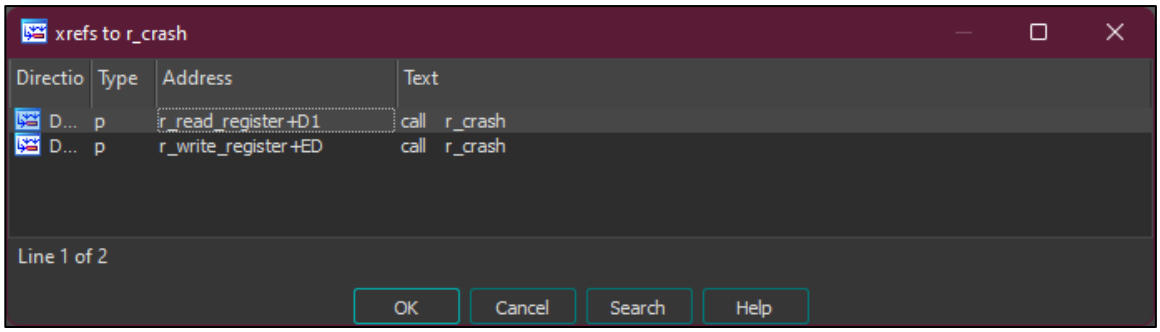
הפונקציה r_crash:

```
; Attributes: noreturn bp-based frame

r_crash proc near
; __unwind {
endbr64
push    rbp
mov     rbp, rsp
sub     rsp, 10h
mov     [rbp-8], rdi
mov     rax, [rbp-8]
mov     rsi, rax
lea     rdi, format     ; "Machine CRASHED due to: %s\n"
mov     eax, 0
call    _printf
mov     edi, 1          ; status
call    _exit
; } // starts at 168E
r_crash endp
```

ניתוח שגיאות

נראה שמצאנו את פונקציית השגיאות, איפה משתמשים בה ואיזה מידע מעניין היא יכולה להזליג? הפונקציה r_crash נקראת ב-r_read_register וב-r_write_register:



נראה שהשגיאה היחידה שנקבל מהתוכנה היא בהינתן register שגוי (בנסיון לקרוא או לכתוב אליו).

ניתוח זמן ריצה

אנחנו יודעים שבהינתן פקודה תקינה (מכילה register תקין) התוכנית אמורה לרוץ בלולאה אינסופית, אך האם אנחנו יכולים למנף את ההנחה הזאת לכדי הדלפה של האקראיים שחשובים לנו?

למעוניינים בתרגיל מחשבתי, לפני שאתם ממשיכים לעמוד הבא, קחו כמה רגעים כדי להעלות אפשרויות למתקפה שתזליג את המידע הרצוי. עכשיו זה זמן טוב לחזור אחורה ולהיעזר במדריך ל-**Yan85** ☺

תוקפים בערוץ-צדדי

אחרי שאספנו פרמיטיבים שמאפשרים הזלגה של מידע על התוכנית שרצה באימולטור והסקנו איזה אקראיים צריך להזליג, אפשר להתחיל להרכיב את הפאזל לכדי תוכנית פעולה שנוכל לממש.

הרעיון

נוכל להרכיב מספר תוכניות קטנות ובכל אחת מהן להגריל את ה-opcode-ים, האוגרים וקריאות המערכת (מתוך VALUES אפשריים) שמרכיבים אותן עד שנקבל תוצאה רצויה (לדוגמא לולאה אינסופית) שתפליל כי המספרים שהגרלנו אכן מתאימים לערכים שניסינו למצוא.

ניזכר באקראיים שעלינו להזליג, הפעם הטבלה תכיל גם את שאר המוגרלים מכיוון שאולי נצטרך להעזר בהם כדי להבין את החשובים לנו:

פעולות	IMM	ADD	STK	STM	LDM	CMP	JMP	SYS
אוגרים	a	b	c	d	s	i	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	write	sleep	exit		

מציאת IMM

פעולת IMM היא הפעולה היחידה שמקבלת גם אוגר וגם ערך מספרי (מספר שלא תלוי ב-VALUES) כארגומנטים. בהינתן תוכנית מינימאלית של פקודה אחת (שמקבלת ערך מספרי) שאחריה פקודה לא תקינה, קיימת רק אופציה אחת לקבלת לולאה אינסופית:

```
IMM i = 2 # jump over invalid op
INVALID_OP # 0xff 0xff 0xff
```

השמה של כתובת קוד 2 ל-i תגרום לדילוג על הפקודה הלא תקינה ותמנע קריסה. נשים לב כי בסוף הרצת פעולה ערכו של i יעלה ב-1 ולכן בסוף פעולת IMM ערכו אמור להיות 3, זאת כתובת הקוד של הפעולה לאחר INVALID_OP.

וכדי למצוא את הקומבינציה הזאת נוכל להגריל כך, כאשר כל הגרלה מתבצעת מסט הערכים האפשרי שמוגדר כ-VALUES:

```
<random instruction> <random register> = 2 # jump over invalid op
INVALID_OP # 0xff 0xff 0xff
```

בקבלת לולאה אינסופית נדע כי מצאנו את IMM ואת האוגר i.

נשים לב כי זה אפשרי ליצור לולאה אינסופית עם פקודה אחת גם בצורות שונות (לדוגמא STK i NONE), אך בהינתן שאנו מעבירים כארגומנט שני ערך מספרי שהוא לא תקין בתור אוגר (לא נמצא ב-VALUES) - פעולה שהיא לא IMM תגרום לקריסת התוכנית:

פעולות	~~IMM~~	ADD	STK	STM	LDM	CMP	JMP	SYS
אוגרים	a	b	c	d	s	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	write	sleep	exit		

מציאת STM

התוכנית הבאה תניב קריסה כאשר STM נכון, אחרת היא תכלא ללולאה מכיוון שמתבצעת השמה מתוך כתובת שלא נכתב אליה אל i אז i=0:

```
IMM reg = 0x30 # arbitrary register points to arbitrary memory address
IMM s = 4 # use a known register (abusing stack pointer purpose)
STM *reg = s # store 4 in arbitrary memory address
LDM i = *reg # load 4 code address from address to instruction pointer
INVALID_OP # 0xff 0xff 0xff
```

מתאר זה מסתמך על כך שאנו יודעים את האקראיים המאוזכרים חוץ מ-STM, לכן נצטרך להתחיל בלמצוא את האקראיים שבשימוש בו.

פעולות	~~IMM~~	ADD	STK	STM	LDM	CMP	JMP	SYS
אוגרים	a	b	c	d	s	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	write	sleep	exit		

מציאת s

התוכנית הבאה תניב לולאה אינסופית כאשר s נכון, זאת מכיוון שבשורת קוד 3 יתבצע דילוג על INVALID_OP:

```
IMM s = 2 # set stack pointer to 2
STK NONE, i # push i, now s = 3
STK i, s # push stack pointer, pop to instruction pointer (i = s = 4)
INVALID_OP # 0xff 0xff 0xff
```

לצורך מציאת s נצטרך למצוא קודם את הפעולה STK.

פעולות	~~IMM~~	ADD	STK	STM	LDM	CMP	JMP	SYS
אוגרים	a	b	c	d	s	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	write	sleep	exit		

מציאת STK

כבר דננו בנושא התוכנית הבאה (כשחיפשנו את IMM):

```
STK i, NONE # pop instruction pointer
INVALID_OP # 0xff 0xff 0xff
```

הנ"ל יניב לולאה אינסופית בהינתן STK נכון, זאת מכיוון שאנחנו מאפסים את i (לא דחפנו כלום ל-stack).

כעת נוכל לסמן כי מצאנו גם את STK וגם את s שהסתמך עליו:

פעולות	~~IMM~~	ADD	~~STK~~	STM	LDM	CMP	JMP	SYS
אוגרים	a	b	c	d	~~s~~	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	write	sleep	exit		

מציאת LDM

התוכנית תניב לולאה אינסופית בהינתן LDM נכון, זאת מכיוון שאנחנו מאפסים את i:

```
LDM i = *s # dereference s (0) to instruction pointer, i=0
INVALID_OP # 0xff 0xff 0xff
```

נשים לב כי במידה והיינו מגרילים בטעות את הערך של IMM במקום LDM גם היינו עלולים לקבל לולאה אינסופית (IMM i = ?), בהינתן שהערך המספרי של s גדול מ-1. אך מכיוון שכבר מצאנו את IMM מקרה זה מתבטל.

עכשיו נוכל לסמן כי מצאנו גם את LDM וגם את STM שהסתמך על הקודמים שמצאנו:

פעולות	~~IMM~~	ADD	~~STK~~	~~STM~~	~~LDM~~	CMP	JMP	SYS
אוגרים	a	b	c	d	~~s~~	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	write	sleep	exit		

מציאת SYS ו-write

בהינתן ש-SYS ו-write נכונים, יכתב ל-stdout תו שהערך המספרי שלו הוא 0xf9:

```
IMM reg = 1 # set arbitrary register to 1
IMM s = 0xf9 # set s (stack pointer purpose abuse) to 0xf9
STM *reg = s # store 0xf9 in memory address 1
IMM all_regs = 1 # set all possible registers (besides known) to value 1
SYS write -> NONE # write(fd=1, buf=1, count=1)
INVALID_OP # 0xff 0xff 0xff
```

במקום להגריל גם את ערכי האוגרים עד שנמצא את b, a ו-c, כשגם ככה אנחנו צריכים להגריל כבר שני ערכים, פשוט נבצע השמה לכל האוגרים האפשריים ועדיין נקבל את התוצאה הרצויה. זאת מכיוון שאנחנו מסדרים את התוכנית כך שהערך 1 יתאים לכל הארגומנטים של קריאת המערכת write.

נבחר לכתוב את התו 0xf9 מכיוון שהוא לא אמור להופיע בפלט רגיל של התוכנית - ù ולכן נוכל לזהות אותו כדי להבין שההגרלה שלנו נכונה.

מכיוון שיש יותר ערכים אפשריים (VALUES=[1,2,4,8,0x10,0x20,0x40,0x80]) ממספר האוגרים התקינים (NONE + 7), נצטרך למצוא את הערך השגוי שעלול להקריס לנו את התוכנית.

מציאת אוגרים תקינים

בהינתן שהאוגר נכון, התוכנית אמורה להיכנס ללולאה אינסופית:

```
IMM reg = 0xff # set arbitrary register to 0xff
```

אחרי שמצאנו את הערך השגוי ולמדנו מה הערכים התקינים שיכולים לייצג אוגר, נוכל להשלים את החיפוש:

פעולות	~~IMM~~	ADD	~~STK~~	~~STM~~	~~LDM~~	CMP	JMP	~~SYS~~
אוגרים	a	b	c	d	~~s~~	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	~~write~~	sleep	exit		

מציאת b, a ו-c

בהינתן ש-b, a ו-c נכונים, יכתב ל-stdout רצף הבתים 0x696a6b:

```
IMM a = 1 # set fd = 1
IMM b = 0x69 # arbitrary memory address
STM *b = b # [0x69] = 0x69
IMM b = 0x6a # next continuous byte
STM *b = b # [0x6a] = 0x6a
IMM b = 0x6b # next continuous byte
STM *b = b # [0x6b] = 0x6b
IMM b = 0x69 # set buf = *0x69 = [0x69, 0x6a, 0x6b]
IMM c = 3 # set count = 3
SYS write -> NONE # write(fd=1, buf=0x69, count=3)
INVALID_OP # 0xff 0xff 0xff
```

פעולות	~~IMM~~	ADD	~~STK~~	~~STM~~	~~LDM~~	CMP	JMP	~~SYS~~
אוגרים	~~a~~	~~b~~	~~c~~	d	~~s~~	~~i~~	f	~~NONE~~
קריאות מערכת	open	read_code	read_memory	~~write~~	sleep	exit		

מציאת open ו-read_memory

בהינתן ש-open ו-read_memory נכונים, תיכתב ל-stdout המילה "lol":

```
# ‘a’ register should point to filename in memory, will use STM
*a = '/tmp/lol'
IMM b = 0 # read mode
SYS open -> a # open(file=’/tmp/lol’, mode=0), return fd in ‘a’
register

IMM b = 0x60 # read file to arbitrary address in memory
IMM c = 3 # file is 3 bytes long
SYS read -> NONE # read(fd=a, buf=0x60, count=3)

IMM a = 1 # stdout
SYS write -> NONE # write(fd=1, buf=0x60, count=3)
```

בטח הבחנתם כי התוכנית זהה ל-Shellcode שבכוונתנו להריץ בסוף. בהחלט אפשר כבר להחליף את הנתיב הזמני בנתיב של קובץ ה-flag, אך זה לא קריטי לבצע את זה כרגע ולכן נעשה זאת בשלב הבא כשאנחנו יודעים את ערכם של האקראיים ☺

פעולות	~~IMM~~	ADD	~~STK~~	~~STM~~	~~LDM~~	CMP	JMP	~~SYS~~
אוגרים	~~a~~	~~b~~	~~c~~	d	~~s~~	~~i~~	f	~~NONE~~
קריאות מערכת	~~open~~	read_code	~~read_memory~~	~~write~~	sleep	exit		

הידד! מצאנו דרך להזליג את כל האקראיים שחשובים לנו! אבל רגע, מה עם הפסודו שכותב את נתיב הקובץ לזיכרון? איך הוא עובד? נענה על זה בעוד רגע - בשלב המימוש.

הופכים תכנון למציאות

אנחנו יודעים מה אנחנו רוצים לעשות כדי להזליג את האקראיים, ואפילו יכולים לעשות את זה ידנית במתן קלט לאימולטור, אבל זה יהיה הליך מייגע ומשעמם.

בואו נהפוך את התהליך לאוטומטי בעזרת Python וסיפריית Pwntools.

Pwntools היא תשתית CTF-ים ופיתוח השמשות ב-Python. מכילה מגוון רחב של כלים ועזרים שמפשטים את התהליך של אינטרקציה עם תוכנות, קבצים ושרתים. למי שלא הכיר לפני אני ממליץ בחום לקרוא עליה.

דרישות

נתחיל בלחשוב מה אנחנו צריכים מהאוטומציה?

- ממשק להרצה של האימולטור עם קלט של תוכנית לבחירתינו
  - עם אפשרות לבצע בדיקת ערוץ-צדדי
    - ✓ האם התוכנית גורמת ללולאה אינסופית?
    - ✓ האם התוכנית כתבה רצף בתים מסוים ל-stdout?
- ממשק להרכבת הוראה (פקודה עם ארגומנטים) לרצף בינארי שירכיב תוכנית
- דרך לעקוב אחרי ערכים אקראיים שמצאנו ולהגריל ערכים שעדיין לא מצאנו
- תיאור תוכנתי לרצף הפעולות שמצאנו לנכון לבצע כדי למצוא את כל הערכים האקראיים
- בהינתן מציאת כל האקראיים, להרכיב תוכנית שתדפיס את ה-flag

הפתרון שלי לדרישות הוא לא האלגנטי ביותר ולא המתוחכם ביותר, אבל הוא עושה את העבודה ולדעתי הוא גם פשוט להבנה. בואו נראה אותו.

ממשק להרצת האימולטור

התוכנית מריצה תהליך חדש של האתגר, שולחת לו את התוכנית שהתקבלה כפרמטר ומחזירה את אובייקט התהליך:

```
def test_vm(program: bytes):
    challenge_process = process(CHALLENGE, level='error')
    challenge_process.send(program)
    return challenge_process
```

לאחר מכן, מחכה מספר שניות לפני שהיא הורגת את תהליך האתגר וקובעת כי הוא נכנס ללולאה - בהינתן תוכנית:

```
def test_vm_is_loop(program: bytes):
    challenge_process = test_vm(program)
    challenge_process.wait(timeout=TIMEOUT)
    is_loop = not challenge_process.poll()

    if is_loop:
        challenge_process.kill()

    return is_loop
```

מריצה תהליך חדש ומחזירה את הפלט שלו - בהינתן תוכנית:

```
def test_vm_stdout(program: bytes):
    challenge_process = test_vm(program)
    challenge_process.wait()
    return challenge_process.recv()
```

ממשק להרכבת הוראה

הוראה (פעולה + ארגומנטים) שתגרום קריסה של התוכנית בהרצתה:

```
BAD_INSTRUCTION = b'\xff\xff\xff'
```

בהינתן פעולה וארגומנטים, מרכיבה רצף בתים מייצג (לפי הסדר שהוגדר באימולטור - ניתן למציאה ב-reversing):

```
def construct_instruction(opcode: int, arg1: int, arg2: int):
    return bytes(bytearray((arg1, opcode, arg2)))
```

מעקב אחרי אקראיים

סט הערכים האפשרי לאקראי:

```
VALUES = set([1, 2, 4, 8, 0x10, 0x20, 0x40, 0x80])
```

מילונים שיחזיקו את האקראיים שנמצאו. נשתמש בהם גם כדי להבין כמה ערכים עוד לא מצאנו. הערך הראשוני שלהם מסמל אקראי שלא מצאנו:

```
REGS = {
  'i': -1,
  ...
  'c': -1
}
OPCODES = {
  'IMM': -1,
  ...
  'SYS': -1
}
SYSCALLS = {
  'write': -1,
  'open': -1,
  'read': -1
}
```

בהינתן מילון של אקראיים (כמו אלה שהגדרנו למעלה) מחזירה את הערכים האפשריים לאלה שלא נמצא:

```
def get_possible_values(value_map):
    return VALUES - set(value_map.values())
```

נממש כל מציאת אקראי כפונקציה נפרדת ונקרא להן אחת אחרי השנייה:

```
def main():
    find_imm_and_i()
    print(f"IMM = {hex(OPCODES['IMM'])}, i = {hex(REGS['i'])}")
    find_possible_regs()
    find_stk()
    print(f"STK = {hex(OPCODES['STK'])}")
    find_s()
    print(f"s = {hex(REGS['s'])}")
    find_ldm()
    print(f"LDM = {hex(OPCODES['LDM'])}")
    find_stm()
    print(f"STM = {hex(OPCODES['STM'])}")
    find_sys_and_write()
    ...
```

נגריל את IMM מתוך הערכים האפשריים לפעולות, עבור כל אחד כזה, נגריל גם את i. לאחר מכן נבנה תוכנית המורכבת משתי הוראות, הוראה תקינה (IMM i = 0x60) והוראה לא תקינה (0xff 0xff 0xff).

בהינתן שהתוכנית מכניסה את האימולטור ללולאה אינסופית - מצאנו את האקראיים.

```
def find_imm_and_i():
    """
    To find IMM and i:
    > IMM i = 0x60
    > 0xff 0xff 0xff
    This is the only combo that will cause an infinite loop
    """
    for imm in get_possible_values(OPCODES):
        for i in get_possible_values(REGS):
            program = construct_instruction(imm, i, 0x60) + BAD_INSTRUCTION
            if test_vm_is_loop(program):
                INSTRUCTIONS['IMM'] = imm
                REGS['i'] = i
                return
```

הפעם נגריל שלושה אוגרים ולפיכך ההגרלה של כל אחד מהם מותנת גם בהגרלה של הקודמים לו. נבדוק האם פלט התוכנית (stdout) מכיל את רצף בתים המוגדר:

```
def find_a_b_c():
    """
    To find a, b and c:
    > IMM a = 1
    > IMM b = 0x69
    > STM *b = b (0x69)
    > IMM b = 0x6a
    > STM *b = b (0x6a)
    > IMM b = 0x6b
    > STM *b = b (0x6b)
    > IMM c = 3
    > SYS write -> NONE
    > 0xff 0xff 0xff
    This will print '\x69\x6a\x6b'
    """
    possible_regs = get_possible_values(REGS)
    for a in possible_regs:
        for b in possible_regs - set([a]):
            for c in possible_regs - set([a, b]):
                program = (construct_instruction(OPCODES['IMM'], a, 1) +
                  construct_instruction(OPCODES['IMM'], b, 0x69) +
                  construct_instruction(OPCODES['STM'], b, b) +
                  construct_instruction(OPCODES['IMM'], b, 0x6a) +
                  construct_instruction(OPCODES['STM'], b, b) +
                  construct_instruction(OPCODES['IMM'], b, 0x6b) +
                  construct_instruction(OPCODES['STM'], b, b) +
                  construct_instruction(OPCODES['IMM'], b, 0x69) +
                  construct_instruction(OPCODES['IMM'], c, 3) +
                  construct_instruction(OPCODES['SYS'], SYSCALLS['write'], 0) +
                      BAD_INSTRUCTION)
                if b'\x69\x6a\x6b' in test_vm_stdout(program):
                    REGS['a'] = a
                    REGS['b'] = b
                    REGS['c'] = c
                    return
```

הדפסת ה-flag

לבסוף, אחרי שמצאנו את כל האקראיים, נוכל לנצח. אבל לפניכן, זוכרים את הפסודו לכתיבת סטרינג לכתובת זיכרון? נרכיב תוכנית שכל תפקידה הוא לכתוב רצף בתים (input_bytes) לתוך כתובת בזיכרון

(address) ולבסוף לשים את הכתובת הזאת בתוך אוגר (dest_reg). התוכנית תעזר באוגר נוסף כדי לבצע זאת (helper_reg):

```
def construct_memory_bytes_instructions(dest_reg, helper_reg, address, input_bytes):
    program = b''
    address_index = address
    for byte in input_bytes:
        program += (construct_instruction(OPCODES['IMM'], dest_reg, address_index) +
                    construct_instruction(OPCODES['IMM'], helper_reg, byte) +
                    construct_instruction(OPCODES['STM'], dest_reg, helper_reg))
        address_index += 1
    program += construct_instruction(OPCODES['IMM'], dest_reg, address)
    return program
```

כעת אנחנו מוכנים להדפיס את ה-flag. נרכיב את ה-Shellcode שמדפיס את ה-flag באמצעות האקראיים שמצאנו, לבסוף נחזיר את ה-flag תוך חילוצו מ-stdout:

```
def print_flag():
    """
    To print flag:
    > *a = '/flag'
    > IMM b = 0
    > SYS open -> a
    > IMM b = 0x60
    > IMM c = 55
    > SYS read -> NONE
    > IMM a = 1
    > SYS write -> NONE
    > 0xff 0xff 0xff
    Will print flag
    """
    filename = '/flag'
    program = (construct_memory_bytes_instructions(REGS['a'], REGS['b'], 0x30,
filename.encode()) +
              construct_instruction(OPCODES['IMM'], REGS['b'], 0) +
              construct_instruction(OPCODES['SYS'], SYSCALLS['open'], REGS['a']) +
              construct_instruction(OPCODES['IMM'], REGS['b'], 0x60) +
              construct_instruction(OPCODES['IMM'], REGS['c'], 0xff) +
              construct_instruction(OPCODES['SYS'], SYSCALLS['read'], REGS['a']) +
              construct_instruction(OPCODES['IMM'], REGS['a'], 1) +
              construct_instruction(OPCODES['SYS'], SYSCALLS['write'], REGS['a']) +
              BAD_INSTRUCTION)
    return re.search("pwn\.college\{.{42}\}", test_vm_stdout(program).decode()).group()
```

את שאר המימוש תוכלו לראות בנספח שמכיל את קוד הפתרון.

הרצה

נריץ מול האתגר את התוכנית שבנינו:

```
hacker@babyrev_level22:~$ python3 babyrev/level22.1_leak.py
IMM = 0x1, i = 0x1
STK = 0x40
s = 0x2
LDM = 0x4
STM = 0x2
SYS = 0x8, write = 0x20
a = 0x80, b = 0x8, c = 0x40
sleep = 0x80
open = 0x1, read = 0x10
pwn.college{wId-qEIsprqOvZHN6wj6fVGTLke.QX4gjMsITM2IzW}
```

מזל טוב, קיבלנו את הדגל ☺

סיכום

אז מה היה לנו פה?

- למדנו על הארכיטקטורה המומצאת Yan85
- כתבנו Shellcode שידפיס לנו את ה-flag
- חקרנו את תוכנת האימולטור בחיפוש אחר ערוץ-צדדי להזליג ממנו מידע
- באמצעות ממצאי המחקר וההכרה של הפעולות בארכיטקטורה, הרכבנו תוכניות מינימאליות שיזליגו לנו ערכים אקראיים להרכבת ה-Shellcode
- מימשנו אוטומציה שמחפשת את האקראיים, מרכיבה את ה-Shellcode ומביאה את ה-flag

נהנתי מאוד מכתיבת המאמר וארצה לסיים אותו בכמה תודות: לאפיק קסטיאל שקיבל את ההצעה שלי בזרועות פתוחות ולכותבים הרבים שתורמים מהידע שלהם, עוזרים להחזיק את המגזין בחיים והסבו לי הנאה רבה בקריאת המאמרים שלהם לאורך השנים.

מקווה שעניינתי אתכם ותודה שקראתם (גם אם ברפרוף) את המאמר שלי!

קצת עלי

אני יובל גל, אוהב לגרום למחשבים לעשות מה שאני רוצה שהם יעשו, בעל חיבה גדולה למחקר ולפעמים גם זיקה לפיתוח.