שאלה 1

סעיף א

```
char* SHELLCODE = "..."
typedef void voidfunc(void)
void my_function(){
char* allocated_addr = SafeMem(1); // allocate memory with read write
memcpy(allocated_addr, SHELLCODE, 0x1000); // copy shellcode to allocated
memory (size = 0x1000B = 4096B)
SafeMem(4); // change permisions for address to allow execution
// call allocated address as function
voidfunc* f = (voidfunc*)allocated_addr;
f();
}
```

הסבר קצר: מקצים אזור לקריאה וכתיבה, מעתיקים אליו את הקוד, ואז משנים את ההרשאות שלו לקריאה וביצוע קוד. לבסוף נקפוץ לאותו אזור ע"י קריאה לפונקציה.

סעיף ב

באופן כללי, הצורה בה אנחנו קוראים לפונקציה ומסדרים את הארגומנטים על המחסנית לקראת הקריאה אינה חד משמאית. לנושא זה קוראים calling convention ו- cdecl__ הוא אחד מהם.
כאשר אנו יודעים שפונקציה נקראת עם קונבנציה cdecl__ זה אומר שהארגומנטים מוספים למחסנים מהסוף להתחלה, כלומר הארגומנט הראשון בכתובת הקטנה ביותר, השני בכתובת גבוהה יותר, וכן הלאה. בנוסף, הפונקציה אינה מנקה את הארגומנטים של עצמה, ויש לנקות אותם מהמחסנית לאחר שהיא מסתיימת (או לשמור מקום ייעודי לארגומנטים של פונקציות מראש באזור של המשתנים הלוקליים, כפי שאנחנו רואים ב- crackmes).
המשמעות של cdecl__ כאשר רוצים לעשות ROP היא שעלינו לנקות את הארגומנטים של הפונקציה בעצמנו, כלומר אם קפצנו לכתובת של פונקציה הנקראת באופן cdecl__, על כתובת החזרה להיות כתובת ל- gadget שעושה pop מספר פעמים כמספר הארגומנטים בפונקציה, או כל gadget אחר שיקטין את ה- esp בכמה שצריך כדי לעקוף את הארגומנטים הללו. במקרה של SafeMem מדובר בסך הכל ב- pop בודד עבור ארגומנט ActionTyp.

סעיף ג

מאוד הגיוני שבכתובת של gadget כלשהו או בערך מספרי/תוי כלשהו שנרצה לשים ב- stack באמצעות ה- overflow עשויים להכיל ביתים עם ערך 0 (a.k.a. null bytes). כאשר מדובר ב- overflow מבוסס ASCII, לדוגמה אחד הנובע מ- strcpy, אז נפסיק להעתיק את הערכים כאשר נגיע ל- null byte הראשון, שכן הוא מסמן על סוף מחרוזת. על מנת להתמודד עם זה, יש לשים ערכים שאינם מכילים null bytes ולשנות את ערכם תוך כדי פעילות ה- ROP chain, לדוגמה עם פעולות אריתמטיות פשוטות או not/neg.

סעיף ד

Stack	meaning
G: mov ebx, [edx]	get address to shellcode
G: mov ebp, esp	← EBP points here for the rest of the ROP chain
G: mov [ebp + 48], ebx	Write shellcode address as src addr for memcpy
G: not [ebp + 16]	Not SafeMem address (original contains null bytes)
G: neg [ebp + 24]	Neg SafeMem ActionType arg
Not &SafeMem	0xFFBEDCBF
G: pop edx	pop cdecl argument
0xFFFFFFFF	neg of 1 (no null bytes)
G: mov [ebp + 68], eax	Write returned value from SafeMem as ret addr for SafeMem
G: mov [ebp + 52], eax	Write returned value from SafeMem as memcpy dst
G: not [ebp + 56]	not op on memcpy size arg (original contains null bytes).
&memcpy	0x71727374
G: not [ebp + 64]	Ret addr of memcpy, will do NOT on second SafeMem call. NOTE: args are skipped due to winAPI call (stdcall)
0xdeadbeef	EBP+48 – write shellcode addr here (memcpy src)
0xdeadbeef	EBP+52 – write SafeMem ret here (memcpy dst)
0xFFFFEFFF	EBP+56 – not 0xFFFFEFFF = 0x1000 = 4096B (memcpy size)
G: not [ebp + 72]	Not on SafeMem ActionType arg (← ESP is here after memcpy)
Not &SafeMem	EBP+64 – 0xFFBEDCBF (not of original address)
0xdeadbeef	EBP+68 – write shellcode addr here (memcpy ret)
0xFFFFFFFB	not 4

הסבר השרשרת:

תחילה אנו אוספים את הכתובת ל- shellcode וקובעים את ה- ebp להיות ה- esp הנוכחי. מכאן יש 5 סוגי פקודות:

1. פקודה רגילה, כלומר סתם gadget שעוזר לנהל את ה- stack או להשיג את המטרה
2. אדום – פקודה שדורסת זיכרון על ה- stack.
3. סגול – ערך שיידרס במהלך השרשרת.
4. כחול – פעולות על ביטים המונעות null bytes.
5. ירוק – ערכים שיש לפעול על הביטים שלהם כדי לקבל את הערך הרצוי (למניעת null bytes)

המחסנית כבר מוכנה לקריאה הראשונה של SafeMem אבל לפני שנקרא לה, נשמור את כתובת ה- shellcode ששלפנו קודם אל המקום הרלוונטיים: ארגומנט src של. זה חוסך לנו להתעסק עם volatile registers כי אין יותר מידע שאנחנו מעוניינים לשמור למעט ebp שהוא לא volatile.

הכתובת של SafeMem מכילה null bytes וגם הערך 1 שהוא הארגומנט הראשון שאנחנו צריכים. לכן עלינו לבצע את פקודות ה- not ו- neg שישנו את הערכים לרצויים. לאחר העיוותים האלה, הפונקציה תקרא עם ארגומנט 1. הפונקציה היא cdecl אז כתובת החזרה שלנו שהיא ל- pop כלשהו מתייחסת לארגומנט הבודד של SafeMem. לאחר הקריאה, הוקצה עמוד ויש לנו את הכתובת אליו ב- eax.

לאחר מכן כותבים את ערך החזרה של SafeMem להיות הארגומנט dst של memcpy memcpy וכתובת החזרה של הקריאה השנייה של SafeMem, ואז הופכים את הביטים של ארגומנט size כי במקור הוא 0x1000 המכיל null bytes. מכאן אפשר לקרוא ל- memcpy כרגיל. נזכור שזהו winAPI call, כלומר הוא מנקה את המחסנית מארגומנטים. לכן ישר ניתן להפוך את הכתובת של SafeMem השנייה ושל הארגומנט שלה, הפעם 4 כדי לשנות את ההרשאות ל- read_execute.

לאחר הקריאה ל- SafeMem השנייה, איזור הזיכרון שהוקצה הוא בגודל עמוד, מכיל את התוכן של ה- shellcode, ובעל הרשאות קריאה וריצה. דאגנו שכתובת החזרה של SafeMem השני תהיה ערך החזרה של SafeMem הראשון, כלומר האזור המוקצה. לכן, לאחר הקריאה השנייה ל- SafeMem, נקפוץ מייד ל- shellcode המועתק ונתחיל להריץ את הקוד שלנו.

שאלה 2

סעיף א

הפונקציה הנתונה מקבלת מערך charים שמייצג כתובת כלשהי, ובודקת אם הוא מתחיל ב0xE8, שהוא הopcode של הפקודה jump. אם לא מוחזר null, אם כן משתמשים בשאר האיברים במערך (אם זו קידוד פקודת call חוקית אז יהיה שם את הoffset של הקפיצה) כדי לחשב לאיזה כתובת נקפוץ כשנבצע את הפקודת call.

סעיף ב

מכיוון שלפונקציה יש מספר נקודות כניסה, נעדיף לא לשים הוק בכל נקודה, זה יהיה מסורבל, ועלול לגרום גם לבעיות נכונות. במקום זאת נשים הוק ב**סוף** הפונקציה, ונשתמש בו כדי לקרוא לפונקציה שוב עם הפרמטרים הנכונים, ולדרוס את ערך החזרה המקורי עם ערך החזרה שנוצר מהקריאה השנייה.

לשם כך נגדיר את ההוק הבא:

```
__declspec(naked) void secretHook()
{
	log_file << "entered hook" << endl;
	if (calledFromItself) {
		calledFromItself = false;
		i++;

		__asm {
			mov eax, [ebp - 4]
			leave
			pop ecx
			leave
			retn
		}
	}
	else {
		__asm {
			mov eax, [ebp + 8]
			mov x, eax
			mov eax, [ebp + 4]
			mov originalRet, eax
		}

		f = (HOOK_TYPE)findCallAddress(originalRet - 5);

		if (f == NULL) {
			log_file << "oops" << endl;
```

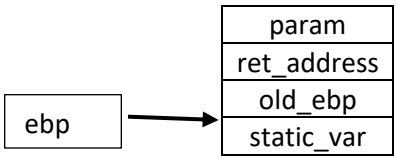
```
                exit(1);
            }

            calledFromItself = true;
            f(x + i);
        }
}
```

נחדיר אותו בסוף הפונקציה, וכמובן שנצטרך לדרוס כמות instructions שלמה (לא חלק כי אז הקריאה תקרוס), למזלנו שלוש הפקודות בסוף הפונקציה הם בדיוק 5 בתים, שזה מה שאנחנו צריכים בשביל להחדיר שם פקודת jmp. נעשה זאת ונכוון את הjmp להוק שלנו.

מכיוון שאנחנו נקרא בהוק שוב לפונקציה, אנחנו נגיע שוב להוק, ולכן כדי לבקר על מתי הגענו להוק מקריאה **מקורית** ומתי הגענו אליו מקריאה **מזויפת שאנחנו יצרנו**, נשמור משתנה בוליאני calledFromItself שיתחיל כfalse כמובן.

בתוך ההוק נעשה את הדבר הבא:

אם הגענו להוק מקריאה אמיתית (calledFromItself=false) אז נשלוף את x ואת כתובת החזרה של הפונקציה מהמחסנית. אנחנו יודעים את המיקום המדויק שלהם כי נתון לנו שהפונקציה לא מתעסקת איתם בשאלה, ולכן הם יהיו במקום הדיפולטי שלהם:



כפי שניתן לראות מהסרטוט, x יהיה ebp+8 והכתובת חזרה תהיה בebp+4.

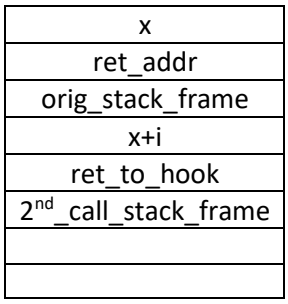
אבל למה אנחנו צריכים גם את הכתובת חזרה? כי נוכל להשתמש בה, בשילוב עם הפונקציה מסעיף א' כדי לחשב את נקודת **הכניסה** של הפונקציה!

הרי כתובת החזרה היא בדיוק 5 ביטים אחרי הcall, ולכן אם נשלח לפונקציה שלנו את הכתובת חזרה פחות 5 בתים נקבל ממנה את כתובת הקפיצה של הcall, כלומר נקודת הכניסה של הפונקציה באיטרציה זו.

לאחר שנשיג את שני הנתונים האלה, נדליק את calledFromItself (כי הפעם הבאה שאנחנו הולכים להכנס להוק זה בגלל קריאה לפונקציה מההוק עצמו), ונקרא לפונקציה עם x+i (כאשר i הוא גלובלי שמאותחל ל1 ועולה כל קריאה).

אחרי שכל מה שתיארנו עכשיו יקרה, אנחנו נגיע שוב להוק, אבל הפעם, זה יהיה כי אנחנו קראנו לפונקציה ולכן calledFromItself יהיה true. נכבה אותו (למען הקריאות הבאות לפונקציה) ונקדם את i (אותה סיבה).

לאחר מכן נשחזר את הפעולות שדרסנו עם הjmp להוק, אבל נעשה זו בצורה שונה קצת. כדי להבין למה בוא נסתכל על מבנה המחסנית בנקודה זו בזמן:



אנחנו רוצים בנקודה זו לחזור לפונקציה **שקראה לפונקצית חישוב שלנו**, ולכן נצטרך לקפל את כל החלק הזה של המחסנית ולחזור לret_address.

איך נעשה זאת?

א. נבצע את השמת ערך החזרה בeax (הפעולה mov eax, [ebp-4]), זה לא קשור לקיפול המחסנית, פשוט אנחנו רוצים לחזור עם ערך החזרה של x+i ולכן חייבים לבצע את הפעולה הזאת.
ב. נבצע leave – נקפל בעצם את 2^{nd}_stack_frame
ג. נוציא את ערך החזרה של הקריאה השנייה, שהוא בעצם הכתובת בסוף ההוק (אחרי השורה f(x+i)), באמצעות pop לרגיסטר **caller saved.**
ד. נבצע leave שוב, מה שleave עושה זה בעצם mov ebp, esp ואז pop ebp, כלומר הוא מקפל את המחסנית אחורה לנקודה לפני פתיחת הframe הכי נמוך, ולכן הוא ימחוק גם את x+i מהמחסנית וגם את orig_stack_frame.
ה. נבצע retn (סטנדרטי בווינדוס, גם עושה ret וגם מוחק את הפרמטרים של הפונקציה מהמחסנית).

וזהו, קיפלנו את כל המחסנית וחזרנו לפונקציה המקורית, כאשר ערך החזרה שנמצא בeax הוא ההפעלה על x+i!

כמובן שכל פעם שתהיה קריאה לפונקציה ההוק יעבוד מחדש באותה צורה, כי המשותף לכל הקריאות זה שהן מסתיימות באותה נקודה ושם מושתל הjmp להוק שלנו, ובנינו אותו בצורה repeatable אז הכל בסדר.